Mamajanov R.Ya., Boynazarov I.M.

# ALGORITMLARNI LOYIHALASHTIRISH VA TAHLIL QILISH

O‘ZBEKISTON RESPUBLIKASI
OLIY TA’LIM, FAN VA INNOVATSIYALAR VAZIRLIGI

Mamajanov R.Ya., Boynazarov I.M.

# ALGORITMLARNI LOYIHALASHTIRISH VA TAHLIL QILISH

*O‘quv qo‘llanma*

ZEBO PRINTS
Denov 2025

**UO‘K 004.421**
**KBK 32.973.26**
**M-49**

Mamajanov R.Ya., Boynazarov I.M. Algoritmlarni loyihalashtirish va tahlil qilish. O‘quv qo‘llanma. – T.: “ZEBO PRINTS”, 2025. 268 bet.

O‘quv qo‘llanma 70610101- Kompyuter tizimlari va ularning dasturiy ta’minoti mutaxassislik talabalariga mo‘ljallangan. Ushbu o‘quv qo‘llanma Algoritmlarni loyihalashtirish va tahlil qilish, algoritmlar tahlili va ularni baholash usullari, berilganlarning abstract turlari, saralash algoritmlari, qidirish algoritmlari bo‘yicha ma’lumotlarni o‘z ichiga qamrab olgan. Bundan tashqari ushbu o‘quv qo‘llanmadan ushbu fanni mustaqil o‘rganuvchilar ham foydalanishlari mumkin.

O‘quv qo‘llanmada mavzular bo‘yicha ma’ruza matnlari, har bir bobdan keyin mustaqil ishlash uchun savollar va mashqlar keltirilgan.

**Taqrizchilar:**

**B.Sh.Rajabov** – Chirchiq davlat universiteti professori, texnika fanlari doktori.

**U.A.Tursunbadalov** – Denov tadbirkorlik va pedagogika instituti “Axborot texnologiyalari” kafedrasi v.v.b., dotsenti, texnika fanlari nomzodi.

**UO‘K 004.421**
**KBK 32.973.26**

**ISBN 978-9910-14-011-2**



## MUNDARIJA

## KIRISH

Mazkur o'quv qo'llanma magistraturaning 70610101- Kompyuter tizimlari va ularning dasturiy ta'minoti mutaxassisliklarifa tehsil olayotgan talabalarning o'zlashtirishi lozim bo'lgan bilimlari asosida tuzilgan bo'lib, kerak bo'lgan bilimlar va ko'nikmalar mazmunini o'z ichiga oladi: Algoritm tushunchasi va tarixi, algoritmlar tahlili, Rekurrent munosabatlar; Bo'l va zabt et algoritmi mazmuni; birlashtirib saralash algoritmi; strassenning matritsalarni ko'paytirish usuli; mustaqil ishlash uchun savollar va mashqlar; ochko'z algoritmi; Tangalarni almashtirish masalasi; minimal qamrov daraxtlarini qurish masalalari; Prim algoritmi; Kruskal algoritmi; Deykstra algoritmi; Dinamik dasturlash masalalari; Uorsholl algoritmi; Floyd algoritmi; Optimal Binar qidiruv daraxti; mustaqil ishlash uchun savollar va mashqlar; Shoxlanish va chegaralash usulining mazmuni ; Vazifani taqsimlash masalasi; Yuk xaltasi (Ryukzak) masalasi; Kommivoyajer masalasi, Saralash algoritmlari; Qo'yish orqali saralash; Tanlash asosida saralash usuli; Almashtirish usuli; Saralashning yaxshilangan usullari; Sheyker saralash algoritmi; Shell saralash algoritmi; Daraxt yordamida saralash; Binar uyum; To'dalash usuli (Heapify Method); Kiritish (Insert) va ajratib olish (Extract) usuli; Piramidali saralash (Heapsort); Tashqi saralash algoritmlari; Tez saralash usuli; Birlashtirish bilan saralash usuli; Oddiy birlashtirish bilan saralash; Tabiiy birlashtirish bilan saralash; Qo'shimcha o'rganish uchun materiallar; Hisoblash bilan saralash usuli; Razryad bo'yicha saralash (radix_sort); Qidiruv algoritmlari; Umumiy tushunchalar; Ketma-ket qidiruv algoritmi; Chiziqli va Binar qidiruv algoritmlari tahlili; qidiruvning yaxshilangan algoritmlari; Interpolyatsion qidiruv; Transpozitsiya usuli; Boshiga ko'chirish usuli; O'tishlar orqali qidirish algoritmi; Matnda qidirish algoritmlari; Chiziqli qidirish algoritmi; Knut-Morris-Pratt algoritmi; Boyer va Mur algoritmi; Ma'lumotlarni xeshlash algoritmlari; Xesh jadval va xesh funksiyalari; Mustaqil ishlash uchun savollar va mashqlar hamda dasturlash tilini o'rganganingizdan keyin bilish kerak bo'lgan muhim tushunchalarni o'z ichiga oladi.

# 1-BOB. ALGORITMNI TAHLIL QILISH USULLARI

## 1.1. Algoritm tushunchasi va tarixi

«Algoritm» atamasi xorazmlik buyuk olim Muhammad al-Xorazmiy (825 y.) nomi bilan bog'liq. Algoritm tushunchasi XX asrning boshlarida yashab ijod qilgan

D. Gilbert, K. Gyodel, S.Klini, A.Chyorch, E.Post, A.Tyuring, N.Viner, A.A.Markov kabi olimlarning ishlari orqali fanga kirib kelgan.

Sonli algoritmlarning eng qadimiysi Yevklid algoritmi [1] hisoblanadi. Bu algoritm ikkita sonning **eng katta umumiy bo'luvchisini (EKUB)** topish uchun ishlatiladi. Yevklid algoritmi **bo'lish amaliyoti** asosida ishlaydi:

1. Berilgan ikkita sonning kattasidan kichigini ayirib borish (klassik versiya).
2. Yoki kattasini kichigiga bo'lib, qoldiq bilan davom ettirish (zamonaviy versiya).

Zamonaviy algoritmlar nazariyasi nemis matematigi Kurt Gyodel (1931 yil) ning ishlaridan boshlangan bo'lib, bu olimning ishlarida izchil aksiomalar doirasida hal etilmaydigan masalalar mavjudligi ko'rsatib o'tilgan.

Algoritmlar nazariyasi bo'yicha birinchi fundamental ish 1936 yilda paydo bo'ldi. Tyuring va Post mashinalari va Chyorchning λ-hisoblagichi taklif etildi. Bu mashinalar algoritmlarni formallashtirishning bir ekvivalenti edi. Shu o'rinda "Algoritm" tushunchasiga berilgan bir nechta ta'riflarni ko'rib chiqamiz. Donald E. Knut[2] algoritmni quyidagicha ta'riflaydi. "**Algoritm** – bu aniq va chekli qoidalar majmuasi bo'lib, u muayyan turdagi masalani yechish uchun zarur bo'lgan harakatlar ketma-ketligini belgilaydi". Ushbu ta'rif

[1] **Yevklid algoritmi** eng qadimgi ma'lum bo'lgan sonli algoritmlardan biri bo'lib, u **eramizdan avvalgi III asrda** yunon matematigi **Yevklid** tomonidan bayon qilingan. Ushbu algoritm **"Elementlar"** nomli asarda bayon etilgan bo'lib, hozirgi kunda ham samarali usullardan biri hisoblanadi.

[2] Donald Ervin Knut – zamonaviy informatikaning asoschilaridan biri bo'lib, algoritmlar nazariyasi va dasturlash tillari sohasida katta hissa qo'shgan olim. U **algoritmlarni tahlil qilish** va **dasturiy ta'minotni strukturaviy loyihalash** bo'yicha ko'plab ilmiy ishlarni yaratgan.



bugungi kunda algoritmlarning nazariy va amaliy tushunchalarini belgilashda asosiy mezon hisoblanadi.

Knut o'zining **"The Art of Computer Programming"** kitobida algoritmning asosiy xususiyatlarini quyidagicha ajratib ko'rsatadi:

1. Tugallanganlik **(Finiteness)** – algoritm chekli qadamdan iborat bo'lishi kerak.
2. Aniqlanganlik **(Definiteness)** – har bir qadam aniq va tushunarli bo'lishi kerak.
3. Kirish **(Input)** – algoritm bir yoki bir nechta kirish ma'lumotlarini qabul qilishi kerak.
4. Chiqish **(Output)** – algoritm natija berishi kerak.
5. Samaradorlik **(Effectiveness)** – har bir qadam oddiy va bajarilishi mumkin bo'lgan amallardan iborat bo'lishi kerak.

A.N.Kolmogorov [3] algoritmni **qat'iy belgilangan qoidalarga asoslangan hisoblash jarayoni** sifatida ta'riflaydi. Ya'ni, «**Algoritm** – bu qat'iy belgilangan

qoidalarga muvofiq amalga oshiriladigan muayyan sondagi qadamlardan keyin masalaning yechimiga olib keluvchi hisoblash tizimidir". Bu ta'rif algoritmlarning asosiy tamoyillariga mos keladi:

1. **Aniqlik** – har bir qadam oldindan belgilangan.
2. **Cheklilik** – algoritm chekli qadamlarda yakunlanadi.
3. **Natijaviylik** – ma'lum kirish ma'lumotlari asosida yechimga olib keladi.

Kolmogorovning algoritm haqidagi ishlari **algoritmik informatsiya nazariyasi** va **hisoblash matematikasi** sohalariga katta ta'sir ko'rsatgan.

Andrey Markov[4] algoritmni **hisoblash jarayoni** sifatida tasvirlab, uni quyidagicha bayon qiladi: «**Algoritm** – bu ma'lum kiruvchi ma'lumotlardan izlanayotgan yechimga olib keluvchi hisoblash jarayoni

[3] Andrey Nikolaevich Kolmogorov (1903-1987) – XX asrning eng buyuk matematiklaridan biri bo'lib, ehtimollar nazariyasi, matematik statistika, topologiya, funktsional analiz, differensial tenglamalar, algebraik geometriya, matematik mantiq va algoritmik informatsiya nazariyasi sohalariga ulkan hissa qo'shgan.

[4] Andrey Andreevich Markov (1903-1979) – matematik mantiq, algoritmlar nazariyasi va algoritmik tillar sohalarida muhim ishlari bilan mashhur bo'lgan rus matematigi. Uning otasi, Andrey Andreevich Markov Sr., ehtimollar nazariyasidagi Markov zanjirlari tushunchasini yaratgan buyuk matematik edi.

to‘g‘risidagi ko‘rsatma”. U algoritmning quyidagi asosiy xususiyatlarini ta’kidlagan:

1. **Kirish (input)** – algoritm boshlanishida beriladigan ma’lumotlar mavjud.
2. **Qadamlar ketma-ketligi** – algoritm oldindan belgilangan amallarni bajaradi.
3. **Chiqish (output)** – algoritm natijada yechimni hosil qiladi.

Uning ta’rifi **Markov algoritmlari** deb nomlanuvchi **formallashtirilgan algoritmik tizim** doirasida keltirilgan bo‘lib, bu model Turing mashinalari va rekursiv funksiyalar bilan bog‘liq (Markov zanjirlari bilan adashtirmaslik kerak).

M.M. Rozental [5] tahriri ostida chop etilgan falsafa lug‘atida «**Algoritm** - bu bir turdagi masalalarni yechishga olib keladigan aniq operatsiya (amal)lar tizimini muayyan tartibda bajarish to‘g‘risidagi ko‘rsatma” deb ta’riflangan. Bu ta’rif algoritmning quyidagi asosiy xususiyatlarini ta’kidlaydi:

1. **Muayyan turdagi masalalar** – algoritm bir xil turdagi masalalarni yechishga mo‘ljallangan bo‘ladi.
2. **Aniq operatsiyalar tizimi** – har bir qadam oldindan belgilangan amallar asosida bajariladi.
3. **Muayyan tartib** – amallar aniq ketma-ketlik asosida bajarilishi kerak.

Bu ta’rif umumiy qilib aytganda, klassik algoritmik tushunchalarga mos keladi va ko‘plab matematik hamda informatika olimlari tomonidan qabul qilingan algoritm ta’riflari bilan uyg‘unlikda qabul qilingan.

1950-yillarga kelib Kolmogorov va Markovlar o‘z ishlari bilan algoritmlar nazariyasiga hissa qo‘shdilar. 1960-1970 yillarda algoritmlar nazariyasi bo‘yicha quyidagi ilmiy tadqiqot yo‘nalishlari paydo bo‘ldi:

1) **Klassik algoritmlar nazariyasi** - formal tillardan foydalanib masalalarni shakllantirish, qat’iylik masalalari tushunchasi, murakkab sinflarni joriy etish, P=NP(?) muammolarini shakllantirish, NP – to‘liq masalalarining ochiq klasslari va ularni tadqiq qilish;

[5] Mark Moiseevich Rozental (1906-1975) – falsafa, dialektik va tarixiy materializm bo‘yicha ishlari bilan mashhur.



2) **Algoritmlarni asimptotik tahlillar nazariyasi** - algoritmlarning qiyinlik darajasi va murakkabligi tushunchalari, algoritmlarni baholash mezonlari,

asimptotik bashoratlarni olish usullari, xususiy hollarda rekursiv algoritmlar uchun qiyinlik darajasi va bajarilish vaqtini asimptotik tahlil qilish;

3) **Hisoblash algoritmlarini amaliy tahlil qilish nazariyasi** - qiyinlik darajasining aniq funksiyasi olish, funksiyalarni intervalli tahlil qilish, algoritm sifatining amaliy mezonlari, ratsional algoritmlarni tanlash usullari, bu yo‘nalishdagi asosiy ishlardan biri sifatida D. Knutning **“The Art of Computer Programming”** (Kompyuter uchun dasturlash san’ati) nomli kitobini olish mumkin. ***Algoritmlar nazariyasining maqsad va vazifalari:***

- “algoritm” tushunchasini shakllantirish va formal algoritmik tizimlarni tadqiq qilish;
- algoritmik yo‘l bilan hal etilmaydigan masalalarning formal isboti;
- masalalarni klassifikatsiyalash, asosiy tushunchalar va klasslarning murakkabligini tadqiq qilish;
- algoritmlarning murakkabligini asimptotik tahlil qilish;
- rekursiv algoritmlar tahlili va tadqiqi;
- algoritmlarni qiyosiy tahlil qilish maqsadida qiyinlik funksiyasini ishlab chiqish;
- algoritmlar sifatini qiyosiy baholash mezonlarini ishlab chiqish.

Algoritm tushunchasini formallashtirish

**1-ta’rif.** Algoritm - bu mumkin bo‘lgan boshlang‘ich ma'lumotlar sinfi uchun umumiy bo‘lgan masalani yechish jarayonida bajariladigan elementar operatsiya (amal)larning chekli ketma-ketligini belgilaydigan ma'lum bir tilda berilgan chekli ko‘rsatma.

Masalaning dastlabki kiruvchi ma’lumotlari maydoni (to‘plami) D bo‘lsin, R esa mumkin bo‘lgan natijalar to‘plami bo‘lsin, u holda algoritm $\mathbf{D} \rightarrow \mathbf{R}$ ko‘rinishda ifodanadi. Bu ifoda to‘liq bo‘lmasligi mumkin.

Algoritm xususiy algoritm deyiladi, agar natijalar faqat ba'zi bir $d \in D$ lar uchun bo'lsa va to'liq algoritm deyiladi, agar algoritm barcha $d \in D$ lar uchun to'g'ri natija bersa.

**2-ta'rif.** Algoritm - bu chekli vaqt ichida ijrochining masalani yechish natijasiga erishish tartibini tavsiflovchi aniq ko'rsatmalar to'plami.

Algoritmning aniq yoki noaniq ta'riflari bir qator talablarni o'z ichiga oladi:

- algoritm chekli sondagi elementar ko'rsatmalarni o'z ichiga olishi, ya'ni yozuvning chekli bo'lishi talabini qanoatlantirishi kerak;
- algoritm masalani yechishda chekli qadamlarni bajarishi, ya'ni amallarning chekliligi talabini qanoatlantirishi kerak;
- algoritm barcha ruxsat etilgan dastlabki ma'lumotlar uchun bir xil bo'lishi kerak, ya'ni universallik talabini qanoatlantirishi kerak;
- algoritm masalaga nisbatan to'g'ri yechimga olib kelishi kerak, ya'ni to'g'rilik talabini qanoatlantirishi kerak. **Algoritmning formal xossalari:**

**Diskretlilik** - algoritm masalani hal qilish jarayonini bir nechta oddiy qadamlarning ketma-ket bajarilishi sifatida ifodalashi kerak. Algoritmning har bir bosqichini bajarish uchun chekli vaqt kerak bo'ladi, ya'ni dastlabki ma'lumotlarni natijaga aylantirish vaqt bo'yicha diskret ravishda amalga oshiriladi. Bu xossaning mazmuni – algoritmlarni doimo chekli qadamlardan iborat qilib bo'laklash imkoniyati mavjudligidadir. Boshqacha aytganda, uni chekli sondagi oddiy ko'rsatmalar ketma-ketligi shaklida ifodalash mumkin. Agar kuzatilayotgan jarayonlarni chekli qadamlardan iborat qilib bo'laklay olmasak, u holda uni algoritm deb bo'lmaydi.

**Aniqlik.** Vaqtning har bir momentida ishning keyingi qadami uchun tizimning holati bir qiymatli aniqlanadi. Shunday qilib, algoritm bir xil kiritilgan ma'lumotlar uchun bir xil natijani (javobni) beradi. Algoritmning xar bir qoidasi, undagi amallar va buyruqlar bir ma'noli bo'lishi kerak. Shu hossaga asosan algoritm ijrochisi buyruqlar ketma-ketligini mexanik bajarish imkoniyatiga ega bo'ladi. Ijrochiga berilayotgan ko'rsatmalar aniq mazmunda bo'lishi kerak. Chunki,



ko'rsatmadagi noaniqliklar mo'ljaldagi maqsadga erishishga olib kelmaydi.

**Tushunarlilik** - ijrochi uchun algoritm faqat uning buyruqlar tizimiga kiritilgan, unda (ijrochida) mavjud bo'lgan ko'rsatmalarni o'z ichiga olishi kerak. Algoritm ijrochisi buyruqlar ketma-ketligini qanday bajarishni aniq bilishi kerak. Algoritmning ijrochisi hamma vaqt ham inson bo'lavermaydi. Ijrochiga tavsiya etilayotgan ko'rsatmalar uning uchun tushinarli bo'lishi shart, aks holda ijrochi har qanday amalni ham bajara olmaydi.

**Tugallanganlik (natijaviylik)** - to'g'ri ko'rsatilgan kiruvchi ma'lumotlar uchun algoritm o'z ishini yakunlashi va natijani cheklangan miqdordagi qadamlarda berishi kerak. Bu hossaning mazmuni shundan iboratki, har qanday algoritmning ijrosi oxir-oqibat ma'lum bir yechimga olib kelishi kerak. Har bir algoritm chekli sondagi qadamlardan keyin, albatta, natija berishi shart. Bajariladigan amallar ko'p bo'lsa ham, baribir natijaga olib kelishi kerak. Chekli qadamdan keyin qo'yilgan masala yechimga ega emasligini aniqlash ham natija hisoblanadi. Agar ko'rilayotgan jarayon cheksiz davom etib, natija bermasa, uni algoritm deb ayta olmaymiz.

**Ommaviylik - universallik**. Algoritm faqatgina kiruvchi ma'lumotlari bilan farqlanuvchi bir turdagi masalalar sinfi uchun tuzilgan bo'lishi kerak. Masalani yechish algoritmi umumiy hollar uchun yaratiladi, ya'ni faqatgina boshlang'ich qiymatlari bilan farqlanuvchi bir turdagi masalalar sinfi uchun tuziladi. Bunda boshlang'ich qiymatlar algoritmning qiymatlar qabul qilishi mumkin bo'lgan sohadan olinadi. Har bir algoritm mazmuniga ko'ra bir turdagi masalalarning barchasi uchun ham o'rinli bo'lishi kerak, ya'ni masalaning boshlang'ich ma'lumotlari qanday bo'lishidan qat'iy nazar algoritm shu xildagi har qanday masalani yechishga yaroqli bo'lishi shart.

Algoritm noto'g'ri natijalarga olib keladigan yoki umuman natija bermasa, u holda bu algoritmda ***xatolar mavjud*** bo'ladi. Algoritm har qanday mumkin bo'lgan to'g'ri boshlang'ich qiymatlar uchun to'g'ri natijalarni bersa, u ***xatosiz*** hisoblanadi.

Bajarilayotgan ishning keyingi qadami tizimning joriy holati va yaratiladigan tasodifiy songa bog'liq bo'lgan ehtimoliy algoritmlar ham mavjud. Biroq, tasodifiy sonlarni yaratish usuli "kirish ma'lumotlari" ro'yxatiga kiritilganda, ehtimoliy algoritm oddiy algoritmga aylanadi. Kiruvchi va chiquvchi ma'lumotlar turi bo'yicha algoritmlarni quyidagicha turlarga ajratish mumkin:

- sonli masalalarni yechish uchun algoritmlar (birinchi paydo bo'lgan);
- sonli bo'lmagan algoritmlar.

**Hisoblash modellari** - bu hisoblash jarayonini matematik yoki mantiqiy shaklda ifodalash uchun ishlatiladigan abstrakt tushuncha. Eng mashhur hisoblash modellari quyidagilardan iborat:

a. **Turing mashinasi** - bu abstrakt hisoblash modelidir va istalgan algoritmik jarayonni tasvirlay oladi. Uning asosiy **elementlari** cheksiz lenta, o'qish va yozish qurilmasi, holatlar to'plami. Bu model har qanday hisoblash masalasining algoritmik yechilishi yoki yechib bo'lmasligini aniqlash uchun ishlatiladi.

b. **RAM modeli (Random Access Machine)** - bu real kompyuterning abstrakt modeli bo'lib, cheksiz xotiraga ega bo'lgan mashinani tasvirlaydi. Uning **xususiyatlari** har bir buyruq bajarilishi bitta vaqt birligini oladi, masofaga qaramasdan xotiraga kirish bir xil tezlikda amalga oshadi.

c. **Hisoblash graflari** - algoritmlar graflar orqali tasvirlanadi. Har bir tugun
amal (operatsiya)larni, yoylar esa ma'lumotlar oqimini ifodalaydi.

d. **Lambda hisoblash** - matematik funksiyalarni hisoblash uchun ishlatiladigan tizim. Bu model funksional dasturlash tillarining asosini tashkil qiladi. **Algoritmlar turlari:**

a. **Bo'lish va hukmronlik qilish (Divide and Conquer)** - masalani kichik
qismlarga bo'lib, har bir qismini alohida yechish, so'ng natijalarni birlashtirish bilan aniqlangan algoritmlar. Masalan, Merge Sort, Quick Sort algoritmlar.



b. **Dinamik dasturlash** - masalani kichik qism masalalarga ajratish va natijalarni qayta ishlatish. Masalan, Fibonacci sonlarini hisoblash, Bellman-Ford algoritmi.

c. **Ochko'z (Greedy) algoritmlar** - har bir bosqichda eng yaxshi lokal qaror
qabul qilish. Masalan, Kruskal algoritmi, Prim algoritmi.

d. To'liq tekshirish (Brute Force) **yoki Orqaga qaytish (Backtracking)** -
masalani barcha ehtimollarni sinab ko'rish orqali hal qilish. Masalan, Sudoku yechish, N-queen[6] masalasi, parollarni tekshirish, grafda eng qisqa yo'l topish (Travelling Salesman Problem - TSP), matndan so'zni qiditrish.

**Algoritm - to'liqlik (completeness)** - algoritm har doim javob topa olishi, **optimal yechim** - eng yaxshi natijaga olib kelishi, **murakkablik** - resurslardan samarali foydalanish darajasi kabi **asosiy xususiyatlarga** ega bo'lishi kerak.

Algoritmlarning murakkabligi ularning ishlashi uchun kerak bo'ladigan resurslar (vaqt va xotira) miqdorini baholaydi.

a. **Vaqt murakkabligi (Time Complexity)** - bu algoritmning bajarilishi uchun zarur bo'lgan vaqtni tavsiflaydi.

b. **Xotira murakkabligi (Space Complexity)** - algoritmning ishlashi uchun
zarur bo'lgan xotira miqdorini o'lchaydi.

c. **Eng yaxshi, o'rtacha va eng yomon holat** - har bir algoritm uchun turli kirish ma'lumotlariga qarab bajarilish vaqti har xil bo'lishi mumkin. Masalan, eng yaxshi holat (Best Case) - eng kam vaqt talab qilinadigan kirish ma'lumotlarida.

O'rtacha holat (Average Case) - o'rtacha vaqt talab qilinadigan holat, eng yomon holat (Worst Case) - eng ko'p vaqt talab qilinadigan kirish ma'lumotlarida.

[6] N-Queen (N-Ferz) masalasi - bu shaxmat doskasida N ta ferzni joylashtirish masalasidir. Shunday joylashtirish kerakki, hech qaysi ferz bir-birini urmasin.

**Murakkablik sinflari** - hisoblash murakkabligi nazariyasida masalalar sinflarga bo'linadi. Algoritmlarning murakkabligi – bu ularning **ishlash vaqti** yoki **xotira sarfi** qanday o'zgarishini o'lchaydigan tushunchadir. Murakkablik **Big-O notatsiyasi** bilan ifodalanadi.

**1. Asosiy murakkablik sinflari**

| Murakkablik | Tavsif | Misollar |
|---|---|---|
| $O(1)$ – doimiy vaqt | Kiritilgan ma'lumot hajmidan qat'i nazar, algoritm bir xil vaqt ishlaydi. | Array indeksiga murojaat (arr[i]) |
| $O(\log N)$ – logarifmik vaqt | Ma'lumot hajmi ortgan sari, bajariladigan amallar soni sekin ortadi. | Binary Search (ikkilik qidiruv) |
| $O(N)$ – chiziqli vaqt | Ma'lumot hajmi ortsa, amallar soni ham shu nisbatda ortadi. | Oddiy qidiruv, massivni bosib o'tish (for tsikli) |
| $O(N \log N)$ – chiziqli-logarifmik vaqt | Ko'pgina samarali tartiblash algoritmlari shu sinfga kiradi. | Merge Sort, Quick Sort |
| $O(N^2)$ – kvadratik vaqt | Ma'lumot hajmi ortsa, ish vaqti kvadratga oshadi. | Bubble Sort, Selection Sort |
| $O(2^n)$ – eksponensial vaqt | Juda tez o'suvchi murakkablik; katta N qiymatlar uchun ishlatish mumkin emas. | Fibonachchi rekursiv hisoblash |
| $O(N!)$ – faktorial vaqt | Kombinatorik masalalar uchun xos; juda sekin ishlaydi. | Traveling Salesman Problem (TSP) |

**2. Kompyuterda qabul qilingan sinflar - murakkablik sinflari hisoblash nazariyasi** bo'yicha tasniflanadi:

| Sinf | Tavsif | Misollar |
|---|---|---|



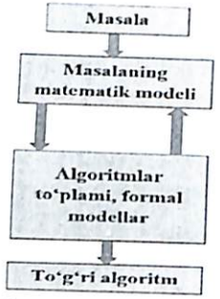
| | | |
|---|---|---|
| $P$ (polinom vaqt) | Muammo polinomial vaqt ichida yechilsa, ya'ni $O(N^k)$ shaklda bo'lsa. | BubbleSort - $O(N^2)$, MergeSort - $O(N \log N)$ |
| $NP$ (deterministik bo'lmagan polinom vaqti) | Muammo yechimi **tez tekshiriladi**, lekin uni topish **juda qiyin**. | Ko'chmanchi savdogar, Gamilton Sikli |
| $NP$-to'liq ($NP$-$C$) | Eng qiyin $NP$ masalalar; agar bitta $NP$-to'liq masalani polinom vaqt ichida yechsak, barcha $NP$ muammolarni shunday yechish mumkin bo'ladi. | 3-SAT, Vertex Cover, Knapsack Problem |
| $NP$-murakkab | $NP$ masalalarga qaraganda **yana ham murakkab** bo'lishi | Algoritmik hal qilinishi mumkin bo'lmagan |
| | mumkin; yechimi tekshirish uchun ham ko'p vaqt ketadi. | masalalar (Turing hal qilinmaslik teoremasi) |

Murakkablik sinflari algoritm samaradorligini baholash va katta hajmli ma'lumotlar bilan ishlashda qaysi algoritmni tanlashni bilish uchun juda muhimdir. Agar masala NP-to'liq yoki NP-murakkab bo'lsa, u holda samarali yechim topish o'rniga taxminiy (approximate) yoki evristik algoritmlar qo'llanadi.

Hisoblash modellari va algoritmlar texnologik rivojlanishning asosi hisoblanadi. Ularning murakkabligini tahlil qilish optimal algoritmlarni yaratish uchun zarurdir. Bu tushunchalar dasturlashda, sun'iy intellektda va katta ma'lumotlar tahlilida muhim ahamiyatga ega.

## 1.2. Algoritmlar tahlili

Amaliy masalalarni yechish uchun algoritmlarni qo'llashda algoritmni to'g'ri tanlash masalasi paydo bo'ladi (1.1-rasm).




**1.1-rasm**. *Algoritmni to'g'ri tanlash masalasi*

Tanlash masalasini hal qilish qiyosiy baholash tizimini qurish bilan bog'liq bo'lib, u o'z navbatida algoritmning formal modeliga bog'liq bo'ladi.

Algoritmning formal modeli bilan ishlash uchun quyidagilar mavjud bo'lgan abstrakt mashina ko'rib chiqiladi:

- adreslangan xotirani qo'llab-quvvatlovchi protsessor;
- yuqori darajali til bilan bog'langan elementar amallar to'plami.

Taxminlar:

- har bir buyruq belgilangan vaqtdan ko'p bo'lmagan muddatda bajariladi;
- algoritmning kirish ma'lumotlarining har biri $\alpha$ bitli $N$ ta mashina so'zlari bilan ifodalanadi.

Algoritmga ma'lumot kiritishda:

$N_\alpha = N * \alpha$ bit ma'lumotlar

Algoritmni amalga oshiradigan dastur $\beta$ bitlar uchun $M$ mashina ko'rsatmalaridan iborat bo'ladi:

$M_\beta = M * \beta$ bit ma'lumot

Algoritmni amalga oshirish uchun hisoblash modellari (abstrakt mashina)ning qo'shimcha resurslari:

- $S_d$– oraliq natijalarni saqlash uchun xotira;
- $S_\gamma$ – hisoblash jarayonini tashkil qilish uchun xotira (rekursiv chaqiruvlar va qaytarishlarni amalga oshirish uchun zarur bo'lgan xotira). Xotira so'zlari tomonidan berilgan aniq masalani yechishda algoritm abstrakt mashinaning cheklangan miqdordagi ($N + M + S_d + S_\gamma$) kabi "elementar" amallarini bajaradi. Shu munosabat bilan



algoritmning murakkabligi ta'rifi kiritiladi. Algoritmning murakkabligi ($n$) deganda aniq kirish ma'lumotlari bilan berilgan aniq masalani yechish uchun formal tizimda algoritm tomonidan bajariladigan "elementar" amallar ($n$) soni tushuniladi.

**Algoritm tahlili** aniq masalalarni hal qilish uchun algoritm talab qiladigan formal mashinaning resurslarini kompleks baholash asosida amalga oshirilishi mumkin. Bu quyidagicha murakkablik funksiyasi bilan aniqlanadi:

$$\varphi_A = c_1 F_a(n) + c_2 N_a + c_3 M_\beta + c_4 S_d + c_5 S_\gamma$$

Bu yerda, $c_i$ - resurslarning og'irlik koeffisienti. Turli xil qo'llash sohalari uchun resurslarning og'irligi har xil bo'ladi.

1. Murakkabligi bo'yicha miqdorlar soniga bog'liq algoritmlar. Ularning murakkablik funksiyasi faqat kiritilgan ma'lumotlarning o'lchamiga bog'liq va ularning qiymatlariga bog'liq emas:

$$F_a(n), \quad n = f(N).$$

*Masalan.* Massivlar va matritsalar bilan standart amallarni bajarish uchun algoritmlar: matritsalarni ko'paytirish, matritsani vektorga ko'paytirish va boshqa amallar.

2. Murakkabligi bo'yicha parametrlarga bog'liq algoritmlar. Ularning murakkablik funksiyasi qayta ishlangan xotira so'zlarining o'ziga xos qiymatlari bilan belgilanadi:

$$F_a(n), \quad n = f(p_1, \dots p_i).$$

Bunday algoritmlarda ikkita sonli qiymat kiritiladi: funksiya argumenti va aniqlik.

*Masalan.* Tegishli darajali sonli qatorlarni hisoblash orqali berilgan aniqlik bilan standart funksiyalarni hisoblash algoritmlari.

*a)* $x^k$ ni ketma-ket ko'paytirish orqali hisoblash: $F_a(a, k) = F_a(k)$.

*b)* $e^k = \sum(^x\_n^n!)$ ni $\varepsilon > 0$ aniqlikda hisoblash: $F_a = F_a(x, \varepsilon)$.

3. Murakkabligi bo'yicha miqdoriy-parametrlarga bog'liq algoritmlar. Ko'pgina amaliy holatlarda murakkablik funksiyasi kiritilgan ma'lumotlarning miqdoriga va ularning qiymatlariga bog'liq:

$$F_a(n), \quad n = f(N, p_1, \dots p_i).$$

Masalan, aniqlik bo'yicha parametrik bog'liq sikl va o'lchov bo'yicha miqdoriy jihatdan bog'liq bo'lgan sikl mavjud bo'lgan sonli usul algoritmlari. Parametrik bog'liq algoritmlar orasida bir guruh algoritmlar ajratilib ko'rsatiladi, ular uchun amallar soni kiruvchi ob'ektlar tartibiga bog'liq bo'ladi. Masalan:

- Saralash algoritmlari;
- Massivda minimum/maksimumni topish algoritmlari.

**Funksiyalarning asimptotik tahlili.** Algoritmlarning murakkabligini tahlil qilishdan maqsad – berilgan masalani yechish uchun optimal algoritmni topishdan iborat. Optimallik mezoni – berilgan algoritm yordamida masalani yechish uchun bajarilishi zarur bo'lgan elementar operatsiyalar soni. Asimptotik tahlil qilishdan maqsad – quyidagicha masalani hal qilish uchun mo'ljallangan turli algoritmlar bo'yicha tizim resurslari sarfini taqqoslash:

- aynan bitta masalani;
- katta hajmdagi kirish ma'lumotlari bilan.

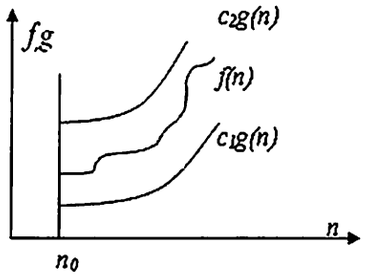
Asimptotik tahlilda qo'llaniladigan murakkablik funksiyasini baholash ***algoritmning murakkabligi deb ataladi*** va ma'lumotlar hajmining oshishi bilan algoritmning murakkabligi qanchalik tez o'sishini aniqlash imkonini beradi. Asimptotik tahlilda funksiyaning o'sish tezligini ko'rsatuvchi quyidagicha belgilashlar qo'llaniladi: $\Theta$(tetta) baholash, $O$ (O katta) baholash, $\Omega$ (Omega) baholash.

**1. $\Theta$ (tetta)** baholash. $f(n)$ va $g(n)$ – musbat argumentli musbat funksiyalar berilgan bo'lsin, $n \in \mathbb{N}$, u holda (1.2-rasm):

$$\theta(g(n)) = \{f(n): barcha\ n \geq n0lar\ uchun\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

*shartni qanoatlanturuvchi musbat* $c_1$, $c_2$ *va* $n_0$ *mavjud*






**1.2-rasm.** *$\Theta$ baholash*

Odatda bu holda $g(n)$ funksiyasi $f(n)$ funksiyaning asimptotik aniq bahosi deb qaraladie, chunki ta'rifiga ko'ra, $f(n)$ funksiyasi $g(n)$ funksiyasidan o'zgarmas koeffitsientgacha farq qilmaydi. Masalan,

1) $f(n) = 4n^2 + nln(n) + 174, f(n) \to \theta(n^2)$;

2) $f(n) \to \theta$

Quyidagi yozuv $f(n)$ noldan farqli o'zgarmasga teng yoki $f(n)$ ning $\Theta$ dagi o'zgarmas bilan cheklanganligini bildiradi:

$f(n) = 7 + 1/n \to \theta.$

*Misollar:*

$c^1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$

$c^1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$

Tengsizliklar quyidagi holatlarda bajariladi:

$$n^{\geq} 7, c_1 \leq \frac{1}{14}$$

$$n \to \infty, c_2 \geq 1/2$$

Xuddi shunday:

$$\frac{2}{7} n^2 \leq \frac{n^2}{2} - 3n \leq \frac{1}{2} n^2$$

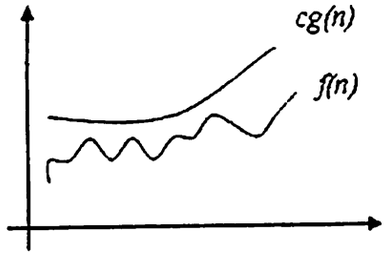
Misol.

$$c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2,\ a, b, c > 0,$$

$$an^2 \leq an^2 + bn + c \leq (a + b + c)n^2.$$

Ixtiyoriy $a, b, c$ lar uchun: $c_1 = \min\{a, a + b + c\}$, $c_2 = \max\{a, a + b + c\}$

**2. $O$ ($O$ katta) baholash.** $\Theta$ baholashdan farqli o'laroq, $O$ baholash faqat $f(n)$ funksiyasi ba'zi $n > n_0$ dan boshlab o'zgarmas koeffitsientgacha bo'lgan $g(n)$ dan oshmasligini talab qiladi (1.3-rasm):




**1.3-rasm.** *$O$ baholash*

$$O(g(n)) = \{f(n): \textit{barcha } n \geq n0 \textit{ lar uchun } 0 \leq f(n) \leq cg(n)$$
$$\textit{shartni qanoatlanturuvchi musbat } c \textit{ va } n_0 \textit{ mavjud}$$

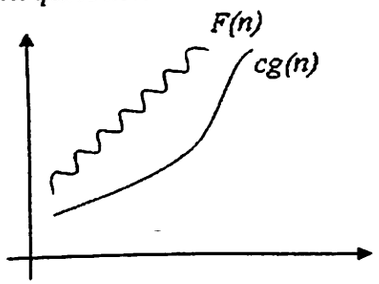
$O(g(n))$ yozuvi o'zgarmas koeffitsientgacha $g(n)$ funksiyasidan tez o'smaydigan funksiyalar sinfini bildiradi, shuning uchun ba'zida $g(n)$ funksiyani $f(n)$ kattalashtirishi aytiladi. Masalan, quyidagi barcha funksiyalar uchun, $O(n^2)$ baholash to'g'ri bo'ladi:

$$f(n) = \frac{1}{n}, \quad f(n) = 12, \quad f(n) = 3n + 17$$

$$f(n) = n \cdot \ln(n), \quad f(n) = 6n^2 + 24n + 77.$$

**3. $\Omega$ (Omega) baholash.** $\Omega$ baholash quyiga baholash usulidir, ya'ni o'zgarmas koeffitsientgacha $g(n)$ga nisbatan sekin o'smaydigan funksiyalar sinfini aniqlaydi (1.4-rasm).

$$\Omega(g(n)) = \{f(n): \textit{barcha } n \geq n0 \textit{ lar uchun } 0 \leq cg(n) \leq f(n)$$
$$\textit{shartni qanoatlanturuvchi musbat } c \textit{ va } n_0 \textit{ mavjud}$$




**1.4-rasm.** *$\Omega$ baholash*

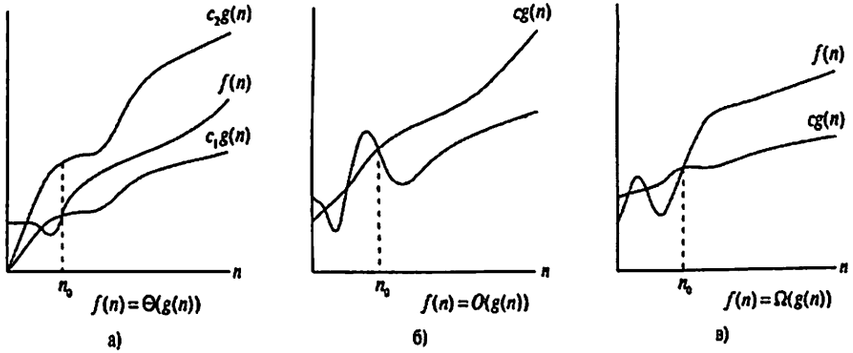


**Teorema.** Ixtiyoriy ikkita $f(n)$ *va* $g(n)$ *funksiyalar uchun* $f(n) = \Theta(g(n))$ *munosabat bajariladi, faqat va faqat* $f(n) = O(g(n))$ *va* $f(n) = \Omega(g(n))$ shartlar bajarilganda (1.5-rasm).

Masalan, $\Omega(n \cdot ln(n))$ yozuv, $g(n) = n \cdot ln(n)$ funksiyadan kam o'smaydigan funksiyalar sinfini belgilaydi. Bu sinfga $n > 2$ darajali barcha ko'phadlar va asosi birdan katta bo'lgan barcha ko'rsatkichli funksiyalari kiradi.

Funksiyalar juftligi uchun asimptotik munosabatlarning hech biri bajarilmaydigan holatlarga misollar mavjud. Masalan, $f(n) = n^{1+sin(n)}$ va $g(n) = n$. Algoritmlarning asimptotik tahlilida, ayniqsa, rekursiv algoritmlar sinfi uchun asimptotik baholarni olish uchun maxsus usullar ishlab chiqilgan. $\Theta$ baholash yuqori afzallikka ega.

Algoritmning murakkabligi funksiyasining asimptotikasini bilish dastlabki ma'lumotlarning katta o'lchamlari uchun yanada optimal algoritmni tanlash orqali bashorat qilish imkonini beradi.




**1.5-rasm.** $\Theta$, $O$ va $\Omega$ baholashlarga grafik misollar.

$n_0$ sifatida mumkin bo'lgan minimal qiymat qo'llaniladi, ya'ni ixtiyoriy $n > n_0$ qiymatlar uchun mos baholash bajarilgan bo'ladi.

Algoritmlarning murakkablik funksiyasini olishda quyidagi "elementar" amallar hisobga olinadi:

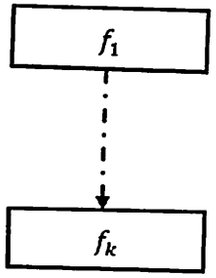
1. Oddiy ta'minlash: $a \leftarrow b$;
2. Bir o'lchovli indekslash $a[i]$: $(adres(a) + i * [element\ uzunligi])$;
3. Arifmetik amallar: $(*, /, -, +)$;

4. Taqqoslash amallari: $a < b$ (<, >,<>, ≤,≥); 5. Mantiqiy amallar: (or, and, not).

Tahlildagi ushbu operatsiyalardan adres bo'yicha o'tish ko'rsatmasi chiqarib tashlanadi, chunki u tarmoqlanishda taqqoslash operatsiyasi bilan bog'liq.

**"Chiziqli" algoritm konstruktsiyasi.** Bu konstruktsiyaning murakkabligi birin-ketin keladigan bloklarning murakkabligi yig'indisidir (1.6-rasm).

$$F"chiziqli" = f1 + \cdots + fk, k - bloklar\ soni$$




**1.6-rasm.** *"Chiziqli" algoritm konstruksiyasi*

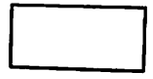
**"Tarmoqlanish" konstruksiyasi.** "Tarmoqlanish" konsturksiyasining umumiy murakkabligi «Then» va «Else» bloklariga o'tish ehtimolini tahlil qilishni talab qiladi va quyidagicha aniqlanadi (1.7-rasm):

$$F"tarmoqlanish" = f_then * p + f_else * (1 - p)$$

```
if ( 1 ) then
```

→ $f_{then}$ *p ehtimollik bilan*

```
else
```



→ $f_{else}$ *p-1 ehtimollik bilan*

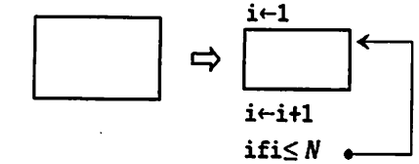
**1.7-rasm.** *«Tarmoqlanish» konsturksiyasi*

**«Sikl 1» konsturksiyasi.** Miqdorlarga bog'liq algoritm. Loyihani elementar operatsiyalarga qisqartirgandan so'ng, uning murakkabligi quyidagicha aniqlanadi (1.8-rasm):

$$F"sikl1" = 1 + 3 * N + N + f"sikl\ tanasi"$$



```
fori← 1 to N
```




```
end
```

**1.8-rasm.** *«Sikl 1» konstruksiyasi. Miqdorlarga bog'liq algoritm*

**«Sikl 2» konstruksiyasi.** Parametrlar soniga bog'liq algoritm. Yomon holatda $O(Nf_{ichki})$:

$$Fyomon(n) = 1 + 3 * N + N * fichki$$

$$f_ichki = n_1 + (n_shart + n_2)$$

Yaxshi holat $O(n)$:

$$Fyaxshi(n) = 1 + n1 + (nshart + n2)$$

O'rtacha holat $O(n)$:

$$Fo'rta(n) = (Fyaxshi + Fyomon)/2$$

```
for i ←1 to N
<>                  (n1)
   if <shart(i)>
   <>               (n2)
   else
   <>               (n3)
   break
   end
end
```

**Oddiy algoritmlar tahliliga misollar.**

| <> | |
|---|---|
| *Sum = 0* | *1* |
| *For i = 1 to n* | *n* |
| *For j = 1 to n* | *n* |
| *Sum = Sum + A(i, i)* | *4* |
| *Next j* | |
| *Next i* | |
| *Print (Sum)* | |
| <> | |

**1-misol.** Kvadrat matritsaning elementlari yig'indisini hisoblash masalasi Algoritm $n$ ning belgilangan qiymati uchun bir xil miqdordagi operatsiyalarni bajaradi va shuning uchun **miqdoriy jihatdan** bog'liq. "Sikl1" konstruktsiyasi metodikasini qo'llasak, quyidagi natijani beradi:

Ichki sikl: $f_1(n) = 1 + 3n + 4n$

Tashqi sikl: $f2(n) = 1 + 3n + nf_1(n)$
Umumiy: $F(n) = f_1(n) + 2(n)$

$$F(n) = 1 + 1 + 3n + n(1 + 3n + 4n) = 2 + 4n + 7n^2 \rightarrow O(n^2).$$

$n$ kvadrat matritsaning chiziqli o'lchami bo'lganligi uchun algoritmga kirish $n^2$ qiymatlar beriladi.

**2-misol.** Massivda maksimum qiymatni aniqlash masalasi.

| <> | |
|---|---|
| *Max = S(1)* | *2* |
| *For i = 2 to n* | *n-1* |
| *If Max < S(i)* | *2 (< u S[i])* |
| *Max = S(i)* | *2 (= u S[i])* |
| *end if* | |
| *Next i* | |
| *Print (Max)* | |
| <> | |

Ushbu algoritm miqdoriy va parametrik bo'lganligi sababli kirish ma'lumotlarning qat'iy o'lchami uchun eng yomon, eng yaxshi va o'rtacha holatlar uchun tahlil qilish kerak.

**Yomon holat**: Agar massiv elementlari o'sish tartibida saralangan bo'lsa, taqqoslashlar soni maksimal (siklning har bir o'tishida) bo'ladi. Shuning uchun ham bu holatda algoritmning murakkabligi quyidagicha:

$F_1(n) = 2 + 1 + 3(n-1) + (n-1)(2+2) = 7n - 4 \rightarrow O(n).$

**Yaxshi holat**: Agar massivning eng katta elementi birinchi o'rinda joylashgan bo'lsa, taqqoslashlar soni minimal bo'ladi. Shuning uchun ham bu holatda algoritmning murakkabligi quyidagicha:

$F_2(n) = 2 + 1 + 3(n-1) + (n-1)(2) = 5n - 2 \rightarrow O(n).$

**O'rtacha holat**: O'rtacha holat formulasidan quyidagi natija olinadi:

$F3(n) = F\_\_\_\_\_1(n) + 2F2(n) = 6n - 3 \rightarrow O(n).$

Ikki algoritmni murakkablik funksiyasi bo'yicha taqqoslash olingan natijalarga ba'zi bir xatoliklarni keltirib chiqaradi. Bu xatolikning asosiy sabablari:

- elementar operatsiyalarning turli chastotali yuzaga kelishi;
- real protsessorda ularning bajarilish vaqtidagi farq.



Shunday qilib, masalaning murakkablik funksiyasidan ma'lum bir protsessorda algoritmning ishlash vaqtini baholashga o'tish zarurati paydo bo'ladi.

$F(A)$ – algoritmning murakkablik funksiyasi qiymati berilgan. Algoritmning dasturiy tatbiqidagi ishlash vaqti $T(A)$ni aniqlash talab qilingan bo'lsin.

**Asosiy muammolar:** • algoritmlarni yozishning formal tizimi va protsessor buyruqlarining real tizimlariining mos kelmasligi;

- kuzatilayotgan dasturning bajarilish vaqtiga sezilarli ta'sir ko'rsatadigan arxitektura xususiyatlarining mavjudligi (konveyr; xotirani keshlash; buyruqlarni, ma'lumotlarni oldindan olish va boshqalar);
- real mashina ko'rsatmalarining turli xil bajarilish vaqtlaridagi farqlar;
- operandlar va ma'lumotlar turlarining qiymatlariga qarab bir xil turdagi ko'rsatmalarni bajarish vaqtidagi farq;
- kompilyatorning o'zi va uning sozlamalari sabab bo'lgan algoritmning dasturi kodini kompilyatsiya qilishning noaniqliklari.

**Vaqtni baholashga o'tish metodikalari:**

1. Operatsion tahlil bo'yicha
2. Gibson usuli
3. O'rtacha vaqtni bevosita aniqlash usuli

**1) Operatsion tahlil bo'yicha**

**1-qadam.** Ma'lumotlar turlarini hisobga olgan holda elementar operatsiyalarning har biri uchun operatsion murakkablik funksiyasini olish - $F_{A,oper,i}(n)$.

**2-qadam.** Muayyan kompyuterda berilgan elementar operatsiyaning o'rtacha bajarilish vaqtini tajriba yo'li bilan aniqlash - $t_{oper,i,o'rta}$. Kutilayotgan bajarilish vaqti operatsion murakkablik va o'rtacha ish vaqtlari ko'paytmasining yig'indisi sifatida hisoblanadi:

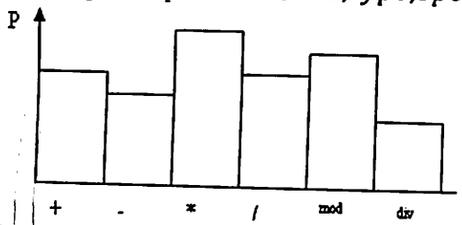
$$T_A(n) = \sum_i F_{A,oper,i}(n) \cdot t_{oper,o'rta,i}$$

**2) Gibbson usuli** - hal qilinayotgan muammo quyidagi turlardan biriga tegishli yoki yo'qligiga qarab vaqtni baholashga o'tishni nazarda tutadi:

- haqiqiy turdagi operandlar ustida operatsiyalar ko'pligi bilan ilmiytexnikaviy xarakterdagi masalalar;
- butun sonli operandlar ustida operatsiyalar ko'pligi bilan diskret matematika masalalari;
- satr tipidagi operandlar ustida operatsiyalar ko'pligi bilan ma'lumotlar bazasi masalalari.

Keyin, tegishli turdagi masalalarni yechish uchun haqiqiy dasturlar to'plamini tahlil qilish asosida operatsiyalarning paydo bo'lish chastotasi aniqlanadi (9-rasm).

Tegishli test dasturlari yaratiladi va berilgan turdagi masala bo'yicha operatsiyaning o'rtacha vaqti aniqlanadi - $t_o'rta,type,oper.$

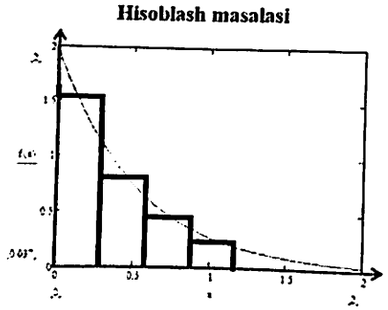



**1.9-rasm.** *Operatsiyalarning chastotali uchrashining mumkin bo'lgan ko'rinishi*

Olingan ma'lumotlar asosida algoritmning umumiy ishlash vaqti quyidagi ko'rinishda baholanadi:

$$toper,o'rt = \sum i\ ptype,oper,i \cdot toper,i$$

$$TA(n) = FA,type,oper(n) \cdot toper,o'rta$$




**1.10-rasm.** *Hisoblash masalasi*



$$p(t) = aexp(-at)$$

$$= \int_0^\infty tp(t)dt = -\frac{(1-at)}{a}\exp(-at)\,|_0^\infty = \qquad 1$$

$$ttype.o'rt_a$$

Aralash masalalar

$$p(t) = \frac{1}{b}$$

$$\int_0^b tp(t)dt = \frac{}{2b}|_0^b$$

$$ttype.o'rt = \qquad = \frac{t^2}{2} \qquad b$$

Belgili-mantiqiy masalalar:

$$p(t) = \frac{1}{\sqrt{2\pi}\sigma_t}\exp\left(-\frac{(t-a)^2}{2\sigma_t}\right)$$

$$ttype.o'rt = \int_0^\infty tp(t)dt = a$$

**3) O'rtacha vaqtni bevosita aniqlash usuli.** Bu usulda murakkablik bo'yicha chuqur tahlil o'tkaziladi:

- $F(n)$ aniqlanadi;
- turli xil o'lchamdagi kirish ma'lumotlari $N$ uchun bevosita eksperiment asosida berilgan dasturning o'rtacha ishlash vaqti $T_{o'rt}$ aniqlanadi;
- berilgan algoritm, kompilyator va kompyuter tomonidan yaratilgan umumlashgan elementar operatsiya uchun o'rtacha vaqt quyidagicha hisoblanadi:

$$To'rt = \frac{\sum i\ tN,iNop,i}{\sum i\ Nop,i}$$

$$tA,o'rt = \frac{To'rt}{n}$$

$$T(n) = FA(n)tA,o'rt$$

$N$ dagi o‘rtacha vaqt barqaror bo‘lsa, ushbu formulani masalaning kirish ma’lumotlari o‘lchamining boshqa qiymatlarida ham qo‘llash mumkin.

Operatsion tahlil masalani yechish uchun u yoki bu algoritmdan aniq foydalanishning nozik tomonlarini ochishga imkon beradi.

**Misol.** Ikkita kompleks sonni ko‘paytirish masalasi:

**A1:** $(a + bi) * (c + di) = (ac - bd) + i(ad + bc) = e + if$

**A2:** $(a + bi) * (c + di) = z1 - z2 + i(z1 + z3) = e + if,$

$$z1 = c(a + b), z2 = b(d + c), z3 = a(d - c)$$

1. A1 algoritmda ($e, f$ ni hisoblash uchun – 4 ta ko‘paytirish)

*MultComplex1(a, b, c, d; e, f)*

| | |
|---|---|
| $e \leftarrow a*c-b*d$ | $f_{A1}$= 8 ta amal |
| $f \leftarrow a*d+b*c$ | $f_{*}$= 4 ta amal |
| *Return(e, f)* | $f_{+,-}$= 2 ta amal |
| *End.* | $f_{\_}$= 2 ta amal |

2. A2 algoritmda ($e, f$ ni hisoblash 3 ta ko‘paytirish)

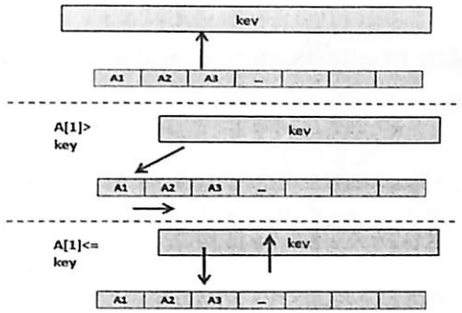
*MultComplex2(a, b, c, d; e, f)*

| | |
|---|---|
| $z_1 \leftarrow c*(a+b)$ | |
| $z_2 \leftarrow b*(d+c)$ | |
| $z_3 \leftarrow a*(d-c)$ | $f_{A2}$= 13 ta amal |
| $e \leftarrow z_1-z_2$ | $f_{*}$= 3 ta amal |
| $f \leftarrow z_1+z_3$ | $f_{+,-}$= 5 ta amal |
| *Return(e, f)* | $f_{\_}$= 5 ta amal |
| *End.* | |

Elementar amallarning umumiy soni bo‘yicha A2 algoritmi A1 algoritmidan pastroq, ammo haqiqiy kompyuterlarda ko‘paytirish operatsiyasi qo‘shish operatsiyasiga qaraganda ko‘proq vaqtni talab qiladi.

**Misol.** Qo‘yish orqali saralash algoritmi tahlili (1.11-rasm).






**1.11-rasm.** *Qo‘yish orqali saralash algoritmini tahlil qilish sxemasi*

| INSERTION_SORT(A) | | vaqt | takrorlanish soni |
|---|---|---|---|
| 1 | **for** $j \leftarrow 2$ **to** *length*[A] | $c_1$ | $n$ |
| 2 | **do** $key \leftarrow A[j]$ | $c_2$ | $n-1$ |
| 3 | ▷ A[j] elementni saralangan A[1..j-1] ketma-ketlikka qo'yish | 0 | $n-1$ |
| 4 | $i \leftarrow j-1$ | $c_4$ | $n-1$ |
| 5 | **while** $i > 0$ and $A[i] > key$ | $c_5$ | $\sum_{j=2}^{n} t_j$ |
| 6 | **do** $A[i+1] \leftarrow A[i]$ | $c_6$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| 7 | $i \leftarrow i-1$ | $c_7$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| 8 | $A[i+1] \leftarrow key$ | $c_8$ | $n-1$ |

**1.12-rasm.** *Qo‘yish orqali saralash algoritmi*

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1).$$

**Yaxshi holat:** massivning barcha elementlari oldindan saralangan ($c_6$ *va* $c_7$ vaqt sarflari yo‘q):

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$

$$T(n) = a(t)n + b(t)$$

**Yomon holat:** massiv elementlari teskari tartibda saralangan,

$$t_j = j:$$

$$\sum_{j=2}^{n} j = \frac{n(n-1)}{2} - 1, \sum_{jj=2}^{n} (j-2) = \frac{n(n-1)}{2},$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right)$$

$$+ c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \left(\frac{c5}{2} + \frac{c6}{2} + \frac{c7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c5}{2} - \frac{c6}{2} - \frac{c7}{2} + c_8\right)$$
$$- (c_2 + c_4 + c_5 + c_8).$$

20 $n$

$$T(n) = a(t)n^2 + b(t)n + c(t).$$

O'rtacha holatda $j/2$ tekshiruvlar o'tkazilishi kerak, shuning uchun bahosi yuqoridagi holatlar bilan bir xil deb olinadi.

### 1.3. Rekurrent munosabatlar

**Rekurrent munosabat** – bu ketma-ketlikni oldingi qiymatlariga asoslanib ifodalovchi tenglama. Ya'ni, ketma-ketlikning har bir $a(n)$ hadini oldingi $a(n-1), a(n-2), \ldots$ kabi hadlar orqali hisoblash mumkin. Agar algoritm rekurrent ravishda o'zini o'zi chaqirsa, uning ishlash vaqtini rekurrent munosabatlar yordamida tavsiflash mumkin. *Rekurrent munosabatlar (recurrence) - bu kichikroq argumentlar bilan funksiyani o'zidan foydalanib tavsiflovchi tenglama yoki tengsizlikdir.* ***Rekursiya*** – bu masalani kichik qism masalalarga bo'lish imkonini beruvchi dasturlash usuli bo'lib, har bir qism masala bitta va aynan



ushbu algoritm yordamida yechiladi. Rekurrent munosabatlar **dasturlash, matematik analiz, hisoblash nazariyasi va algoritmlarning murakkablik bahosida** keng qo'llaniladi. Rekurrent munosabatlarning turlari:

*Chiziqli rekurrent munosabatlar.* Bu turdagi rekurrent munosabatlar oldingi hadlarning chiziqli kombinatsiyasi shaklida bo'ladi:

$$an = c1an-1 + c2an-2 + \cdots + ckan-k + f(n).$$

**1-misol.** Fibonachchi sonlari:

$$Fn = Fn-1 + Fn-2, F0 = 0, F1 = 1.$$

Bu ikkita oldingi hadga bog'liq bo'lgan chiziqli rekurrent tenglama.

**2-misol.** Geometrik progressiya – bu har bir had oldingisining $q$ ga ko'paytirilishi bilan hosil bo'ladigan ketma-ketlik:

$$a_0,\ a_1,\ a_2,\ \ldots,\ a_n,\ \ldots$$

Umumiy formulasi:

$$a_n = a_0 \cdot q^n$$

*Chiziqsiz rekurrent munosabatlar.* Bu turdagi munosabatlar chiziqsiz amallar (ko'paytirish, bo'lish, darajaga ko'tarish, logarifm) ishlatadi.

**3-misol.** Faktorilani hisoblash funksiyasi:

$$n! = n \cdot (n-1)!, 0! = 1.$$

Bu **rekursiv** shaklda ifodalangan chiziqsiz rekurrent munosabatdir.

**4-misol.** Hanoy minorasi (Tower of Hanoi) haqidagi masala:

$$T(n) = 2T(n-1) + 1, \quad T(1) = 1.$$

Bu muammo disklarni ko'chirish uchun **optimal harakatlar sonini** ifodalaydi.

*Rekurrent munosabatlarni yechish usullari:*

*Iterativ usul* - oldingi qiymatlardan foydalanib, rekurrent munosabatni bosqichma-bosqich ochish.

**5-misol.** $a_n = 2a_{n-1}$, $a_0 = 3$. Buni qadamma-qadam (iterativ) tarzda ochish bosqichlari:

$$a_1 = 2 \cdot 3 = 6$$
$$a_2 = 2 \cdot 6 = 12$$
$$a_3 = 2 \cdot 12 = 24$$

Yechimi, $a_n = 3 \cdot 2^n$ ko'rinishda bo'ladi.

*Xarasteristik tenglama usuli* - bu usul **chiziqli rekurrent munosabatlar** uchun ishlatiladi. Masalan, Fibonachchi sonlari berilgan bo'lsin:

$$F_n = F_n{-}1 + F_n{-}2.$$

Uning xarasteristik tenglamasi $r^2 - r - 1 = 0$. Bundan ildizlarni topamiz va umumiy yechimni quramiz.

*Generatsion funksiyalar usuli.* Bu usulda **generatsion funksiya** yordamida rekurrent munosabat transformatsiya qilinadi va yechim olinadi. Generatsion funksiya – bu ketma-ketlikni matematik tarzda ifodalash usuli bo'lib, rekurrent munosabatlarni yechishda kuchli vosita hisoblanadi. Masalan, Fibonachchi sonlari, geometrik progressiyalar va kombinatorik hisob-kitoblarda keng ishlatiladi.

**6-misol.** Oddiy geometrik progressiya berilgan: $a_0 = 1$ va $q = r$ bo'lsa, progressiyaning hadlari quyidagicha ko'rinishga ega bo'ladi:

$$1,\ r,\ r^2,\ r^3,\ r^4,\ \ldots$$

Uning generatsion funksiyasi:

$$G^{(x)} = \sum_{n=0}^{\infty} r^n x^n.$$

Bu cheksiz geometrik progressiya bo'lib, uning yopiq ko'rinishi:

$$G(x) = \frac{1}{1 - rx},\ agar\ |x| < \frac{1}{|r|}.$$

Agar $a_0$ boshlang'ich had bo'lsa:

$$G^{(x)} = \sum_{n=0}^{\infty} a_0 \cdot r^n x^n$$

$$G(x) = \frac{a_0}{1 - rx},\ agar\ |x| < \frac{1}{|r|}.$$

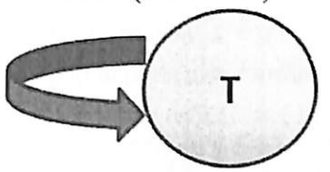


Agar, $a_0 = 2, r = 3$ deb olsak, u holda:

$$G(x) = \frac{2}{1 - 3x},\quad |x| < \frac{1}{3}$$

Bu generatsion funksiya *geometrik progressiyaning yopiq formulasi* bo'lib, har qanday $n$-hadni hisoblashda ishlatiladi.

**7-misol.** Faktorialni hisoblash (1.12-rasm).

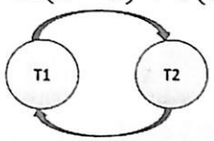



**1.12-rasm.** *To'g'ri rekursiya*

$F(0) = 1$
$F(n) = n * F(n-1)$
$T(n) = T(n-1) + \Theta(1)$

**Bilvosita rekursiya** (rekursiv funksiyasi boshqa funksiyani chaqiradi, bu esa o'z navbatida birinchisini chaqiradi) (1.13-rasm).

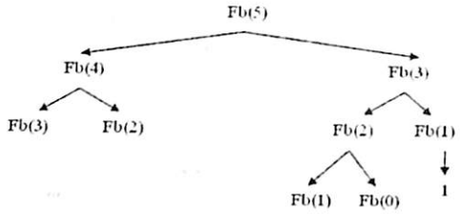
$$T_1(n) = T_2(n-1) + \Theta(1)$$
$$T_2(n) = T_1(n-1) + \Theta(1)$$




**1.13-rasm**. *Bilvosita rekursiya* **8-misol.** Fibonachchi sonlarini hisoblash (1.14-rasm).

$$F_n = F_n{-}1 + F_n{-}2$$

**Function**
$Fib(n)$ **if** $n \leq$
1: **return**
$Fib = 1$ **else**:
**return** $Fib(n) = Fib(n-1) + Fib(n-2)$




**1.14-rasm.** *Fibonachchi sonlarini hisoblash*

*Dasturning bajarish vaqtini tahlil qilish.* To‘xtash shartlaridan biri $n \leq 1$ necha marta bajariladi:

$$G(0) = 1, G(1) = 1, G(n) = G(n-1) + G(n-2), n > 1$$

Algoritm necha marta rekursiv qadamga yetadi

$$H(0) = 0, H(1) = 0, H(n) = 1 + H(n-1) + H(n-2), n > 1.$$

U holda $H(n) = Fib(n) - 1$ va $G(n) = Fib(n)$, umumiy holda:

$$T(n) = H(n) + G(n) = 2Fib(n) - 1.$$

Fibonachchi tenglamasining umumiy yechimi quyidagi ko‘rinishga ega:

$$F(n) = c_1\left(\frac{1+\sqrt{5}}{2}\right)^{-n} + c_2\left(\frac{1+\sqrt{5}}{2}\right)^{-n}$$

Boshlang‘ich shartlar qiymati $F(0) = 0$ va $F(1) = 1$. Shunga mos ravishda quyidagi tenglamalar sistemasiga ega bo‘lamiz:

$$\begin{cases} c_1 + c_2 = 0 \\ \frac{\sqrt{5}}{2}(c_1 - c_2) = 1 \end{cases}$$

Ushbu tenglamalar sistemasini yechib, quyidagilarga ega bo‘lamiz:

$$c_1 = -c_2 = \frac{1}{\sqrt{5}}$$

$$F(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n} - \left(\frac{1-\sqrt{5}}{2}\right)^{n}\right]$$



**Rekursiya xavfi**, cheksiz rekursiyaning paydo bo‘lishi va ko‘p xotiraning sarflanishi bilan aniqlanadi. Ushbu ortiqcha sarflarni kamaytirishning bir necha yo‘li mavjud:

- Juda ko‘p keraksiz o‘zgaruvchilardan foydalanmaslik. Agar qism dastur ularni ishlatmasa ham, lekin bu o‘zgaruvchilar uchun xotira ajratiladi.
- Global o‘zgaruvchilar yordamida stekdan foydalanishni kamaytirish mumkin.
- Statik o‘zgaruvchilardan foydalanilganda, tizim har safar qism dastur chaqirilganda o‘zgaruvchilarning yangi nusxalari uchun xotira ajratishi shart emas.

*Rekursiyadan asossiz foydalanish holatlari.* Yuqorida keltirilgan faktorial funksiyasi, eng katta umumiy bo‘luvchini topish, Fibonachchi sonlarini hisoblash masalalari albatta rekursiya yo‘li bilan hal qilinishi shart emas.

**Algoritm tahliliga doir namunaviy masala:** *Shkaf eshiklari masalasi*. Koridorda 1 dan $n$ gacha ketma-ket raqamlangan $n$ ta shkaf mavjud. Dastlab, barcha shkaf eshiklari yopiq. Siz har safar 1-shkafdan boshlab, shkaflar yonidan $n$ marta o'tasiz. $i$-chi o’tishda ($i = 1,2,\ldots,n$) har bir $i$-chi shkafning eshigini almashtirasiz: agar eshik yopiq bo'lsa, uni oching, agar u ochiq bo'lsa, uni yoping. Oxirgi o'tishdan keyin qaysi shkaf eshiklari ochiq va qaysi biri yopiq bo‘ladi? Ulardan nechtasi ochiq qoladi? Masalaning yechimi:.

Bu masala klassik "shkaf eshiklari" yoki "mahbuslar" masalasi deb yuritiladi. Quyida uning mantiqiy tahlili, yechimi va nechta eshik ochiq qolishini aniqlash bosqichma-bosqich ko’rsatib o‘tilgan.

*Masalaning parametrlarini aniqlash*: $n$ ta yopiq shkaf eshigi, $n$ marta eshiklarni ko‘rib chiqish kerak. Har bir $i$-chi ko‘rishda, faqat $i$ ga karrali raqamlangan shkaflar eshiklari holati (yopiq $\leftrightarrow$ ochiq)ni o‘zgartirish kerak.

*Savol:* oxirida qaysi shkaflar eshigi ochiq bo‘ladi? va nechtasi?

*Masalaning mantiqiy tahlili:* Shkaf raqami $k$ bo‘lgan eshik necha marta ochiladi/yopiladi? Har safar $i$–o‘tishda $k$–shkafga faqat $k$ soni $i$ soniga bo‘linadigan bo‘lsa $k$–shkaf eshigi holati o‘zgartiriladi. Demak,

$k$–raqamli shkafga uning ***bo'luvchilari soni*** marta o'zgarish kiritiladi. Misol uchun:

6-shkaf: bo'luvchilari – 1, 2, 3, 6 → 4 marta holati o'zgaradi → **yopiq** bo'ladi (juft son).

9-shkaf: bo'luvchilari – 1, 3, 9 → 3 marta → **ochiq** bo'ladi (toq son).

Xulosa qilib aytganda, faqat *toq sondagi bo'luvchiga ega son* bilan raqamlangan shkaflar eshigi ochiq qoladi. Bu son o'zidan kichik qaysidir sonning **kvadrati bo'lgan sonlar** bo'lishi mumkin. Chunki, juft bo'luvchilar har doim juft son hosil qiladi (masalan, 12 = 1×12, 2×6, 3×4 → 6 ta bo'luvchi). Faqat **kvadrat sonlar** (masalan, 9 = 3×3) juft bo'luvchilarga ega bo'lmaydi - bitta takroriy bo'luvchi bo'ladi, shuning uchun bo'luvchilar soni toq bo'ladi. *Yechim algoritmi:*

1. Dastlab barcha shkaflar yopiq.
2. Oxirida faqat $i^2 \le n$ bo'lgan sonlar - ya'ni $1^2, 2^2, 3^2, \ldots$ - raqamli shkaflar ochiq qoladi.
3. Demak, ochiq eshiklar soni = $\lfloor \sqrt{n} \rfloor$.

**Misol uchun,** $n = 10$ bo'lsin, bu songacha bo'lgan kvadrat sonlar: $1^2 = 1$, $2^2 = 4$, $3^2 = 9$. Shu sababli, 1, 4, 9–raqamli shkaflar ochiq qoladi. Javob: 3 ta shkaf eshigi ochiq.

**C++ dasturlash tilidagi yechimlari:** Dastlab barcha eshiklar yopiq (false).

Har bir o'tishda $i$ ga karrali raqamli eshiklar holati o'zgartiriladi (true ↔ false). Oxirida ochiq (true) eshiklar raqamini chiqaramiz. To'g'ri yechim:

```
#include <iostream>
#include
<vector>
#include
<cmath> using
namespace std;
int main() {
int n;
```



```
  cout << "Shkaflar sonini kiriting
(n): ";   cin >> n;
  vector<bool> eshik(n + 1, false); // 1-indeksdan boshlaymiz
  // Har bir o'tish   for (int
i = 1; i <= n; ++i) {
for (int j = i; j <= n; j += i)
{
      eshik[j] = !eshik[j]; // eshik holatini o'zgartirish
    }
  }
  cout << "\nOchiq qolgan eshiklar
raqami:\n";   int sanoq = 0;
  for (int i = 1; i <= n; ++i) {
    if (eshik[i]) {
cout << i << " ";
sanoq++;
    }
  }
  cout << "\n\nOchiq eshiklar soni: " << sanoq
<< endl;   return 0; }
```

**Natija:**

*1-holat:*

Shkaflar sonini kiriting (n): 10
Ochiq qolgan eshiklar raqami:
1 4 9
Ochiq eshiklar soni: 3 *2-holat:*
Shkaflar sonini kiriting (n): 21
Ochiq qolgan eshiklar raqami:
1 4 9 16
Ochiq eshiklar soni: 4

**Optimallashtirilgan yechim.** Agar faqat ochiq eshiklar kerak bo'lsa (ya'ni faqat kvadrat sonlarni chiqarish):

```
#include
<iostream>
#include
<cmath> using
namespace std;
int main() {
int n;
  cout << "Shkaflar sonini kiriting
(n): ";   cin >> n;
  cout << "\nOchiq qolgan eshiklar
raqami:\n";   int i = 1;   while (i * i
<= n) {      cout << i * i << " ";
    i++;   }
  cout << "\n\nOchiq eshiklar soni: " << i - 1
<< endl;   return 0; }
```

"Shkaf eshiklari masalasi"ning C++ yechimlari asosida vaqt va xotira murakkabligi (time & space complexity)ni **ikki xil yondashuv** uchun tahlil qilamiz:

**1-usul:** To'liq tog'ri C++ yechimi. Bu yechimda biz har bir o'tishda i-ga karrali raqamli eshiklarni yopamiz yoki ochamiz. Algoritm kodi qisqacha:

```
for (int i = 1; i <= n; ++i)
{   for (int j = i; j <= n; j
+= i) {      eshik[j] =
!eshik[j];   } }
```

**Vaqt murakkabligi** (Time Complexity):

1-chi o'tishda: $n$ ta eshik (1, 2, 3, ..., n) → $n$ ta amal

2-chi o'tishda: $n/2$ ta eshik (2, 4, 6, ...) → $n/2$ ta amal

3-chi o'tishda: $n/3$ ta eshik (3, 6, 9, …) → $n/3$ ta amal

...

$n$-chi o'tishda: 1 ta eshik (faqat $n$) → 1 ta amal Shu bilan,

$$umumiy\ amallar\ soni = \sum_{i=1}^{n} n$$



$\cdot i$=1 [i].

Bu yig'indi $n \cdot log(n)$ chegarasida baholanadi. Demak, vaqt murakkabligi:

$O(n \log n)$ ga teng.

**Xotira murakkabligi** (Space Complexity):

eshik vektorida $n + 1$ ta bool qiymat saqlanadi →

$O(n)$ yordamchi o'zgaruvchilar $(i, j, sanoq, ..)$

→ $O(1)$ Demak, xotira murakkabligi: $O(n)$.

**2-usul:** Kvadrat sonlar asosida optimallashtirilgan yechim kodi: int i = 1; while (i * i <= n) {   cout << i * i << " ";

i++; }

Bu usul faqat kvadrat sonlarni chiqaradi. Ya'ni: $1^2, 2^2, \ldots, k^2 \leq n$.

**Vaqt murakkabligi:** $i^2 \leq n \rightarrow i \leq \sqrt{n}$

$O(\sqrt{n})$

**Xotira murakkabligi:** Hech qanday qo'shimcha massiv yoki vektor ishlatilmayapti, shuning uchun $O(1)$. Qisqacha solishtirish jadvali:

| Yechim turi | Vaqt murakkabligi | Xotira murakkabligi | Izoh |
|---|---|---|---|
| To'g'ri yechim | $O(n \cdot \log n)$ | $O(n)$ | Haqiqiy model, har bir eshik holatini tekshiradi |
| Optimallashtirilgan | $O(\sqrt{n})$ | $O(1)$ | Faqat ochiq eshiklarni topish uchun |

Quyida berilgan mashqlarni shu tarzda tahlil qiling!

### 1.4. Mustaqil ishlash uchun savollar va mashqlar

1. Algoritmlarni qiyosiy baholash tizimi qanday maqsadda foydalaniladi?

2. Masalani kompleks tahlilini hisobga olgan holda yechish algoritmining murakkabligiga umumiy ta'rif bering.

3. Murakkabligi bo'yicha miqdoriy bog'liq algoritmlarga misollar keltiring.

4. Murakkabligi parametlarga bog'liq bo'lgan algoritmlarga misollar keltiring.

5. Murakkabligi bo'yicha Miqdoriy-parametrik bog'liq algoritmlarga misollar keltiring.

6. Algoritmlarning murakkablik funksiyalarini asimptotik tahlil qilishdan maqsad nima?

7. Murakkablik funksiyasining o'sishini $\Theta$–baholashga misollar keltiring.

8. Murakkablik funksiyasining o'sishini $O$ (katta) - baholashga misollar keltiring

9. Murakkablik funksiyasining o'sishini $\Omega$- baholashga misollar keltiring.

10. «Tarmoqlanish» va «sikl» konstruksiyalari ketma-ketligi murakkabligini aniqlovchi formulani yozing.

11. Ichma-ich «sikl» konstruksiyasining murakkabligini aniqlovchi formulani yozing.

12. $m$ marta ichma-ich sikl» konstruksiyasining murakkabligini aniqlaydigan formulani yozing.

13. Algoritmlar murakkabligini operatsion baholashdan vaqtni baholashga o'tishda qanday asosiy muammolar mavjud? 14. Vaqtni baholashga o'tishda yondashuvlar o'rtasidagi farq nimada?

15. Qo'yish orqali tartiblash algoritmining murakkabligini tahlil qiling.

16. Algoritmlar murakkabligining nazariy chegarasini aniqlang.

17. Tub sonlarni topish algoritmlari murakkabligining nazariy chegarasini taxmin qilish mumkinmi?

18. Rekurrent munosabatlarga ta'rif bering, misollar keltiring.

19. Algoritm tavsifidan talab qilinadigan aniqlik xossasini e'tiborga olib:



- o'quv binosidan uyingizga borish yo'nalishlarini yozing. - sevimli taomingizni pishirish jarayoni retseptini yozing.

20. Har qanday musbat butun $n$ uchun $[\sqrt{n}]$ ni hisoblash algoritmini tuzing.

Ta'minlash va taqqoslashdan tashqari, sizning algoritmingizda faqat to'rtta asosiy arifmetik amaldan foydalanish mumkin.

21. Ikki tartiblangan sonlar ro'yxatidagi barcha umumiy elementlarni topish algoritmini tuzing. Masalan, 2, 5, 5, 5 va 2, 2, 3, 5, 5, 7 ro'yxatlar uchun chiqish 2, 5, 5 bo'lishi kerak. Agar berilgan ikkita ro'yxatning uzunligi mos ravishda $m$ va $n$ bo'lsa, algoritm ishlashida taqqoslashning maksimal soni qancha bo'ladi?

22. Yevklid algoritmini qo'llash orqali $EKUB(31415, 14142)$ ni toping.

23. $\min\{m, n\}$ dan $EKUB(m, n)$ gacha bo'lgan ketma-ket butun sonlarni tekshirishga asoslangan algoritmga nisbatan Yevklid algoritmi bo'yicha

$EKUB(31415,14142)$ ni topish necha marta tezroq bo'lishini hisoblang.

24. Har bir $m$ va $n$ musbat sonlar juftligi uchun $EKUB(m, n) = EKUB(n, m \bmod n)$ ekanligini isbotlang.

25. $1 \leq m,\ n \leq 10$ barcha kirishlar orasida Yevklid algoritmi bo'yicha minimal bo'linishlar soni qancha bo'lishi kerak?

26. $1 \leq m,\ n \leq 10$ barcha kirishlar orasida Yevklid algoritmi bo'yicha maksimal qancha bo'linish mumkin?

27. Yevklidning asarida keltirilganidek, Yevklid algoritmi butun sonlarni bo'lishdan ko'ra ayirishlardan foydalanadi. Yevklid algoritmining ushbu versiyasi uchun psevdokod yozing.

28. Yevklidning o'yini doskadagi ikkita teng bo'lmagan musbat son bilan boshlanadi. Ikki o'yinchi navbatma-navbat doskaga berilgan ikkita sonning ayirmasiga teng musbat sonni yozishi kerak. Bu son yangi bo'lishi, ya'ni oldin taxtaga yozilgan sonlardan farq qilishi shart.

Ayirmani yoza olmagan o'yinchi o'yinni yutqazadi. Ushbu o'yinda yutish uchun birinchi boshlash kerakmi yoki ikkinchi?

29. Kengaytirilgan Yevklid algoritmi faqat ikkita musbat $m$ va $n$ sonlarning eng katta umumiy bo'luvchisi $d$ ni emas, balki $mx + ny = d$ tenglikni

qanoatlantiruvchi $x$ va $y$ butun sonlar (musbat bo'lishi shart emas)ni ham aniqlaydi,

a. Kengaytirilgan Yevklid algoritmining tavsifini aniqlang va uni o'zingiz

tanlagan dastrulash tilida dasturini yozing.

b. Shuningdek, koeffitsientlari $a$, $b$ va $c$ butun sonlardan iborat bo'lgan Diofant tenglamasi $ax + by = c$ ning butun sonli yechimlarini topish uchun dasturingizni o'zgartiring.

30. Shkaf eshiklari. Koridorda 1 dan $n$ gacha ketma-ket raqamlangan $n$ ta shkaf mavjud. Dastlab, barcha shkaf eshiklari yopiq. Siz har safar 1-shkafdan boshlab, shkaflar yonidan $n$ marta o'tasiz. $i$-chi o'tishda ($i = 1,2, \ldots, n$) har bir $i$-chi shkafning eshigini almashtirasiz: agar eshik yopiq bo'lsa, uni oching, agar u ochiq bo'lsa, uni yoping. Oxirgi o'tishdan keyin qaysi shkaf eshiklari ochiq va qaysi biri yopiq? Ulardan nechtasi ochiq?

31. Qadimgi dunyo boshqotirmasi. Dehqon daryo qirgogida bo'ri, echki va bir bosh karam bilan turibdi. U qayig'ida shu uchta narsani daryoning narigi tomoniga olıb o'tishi kerak. Biroq, qayiqda faqat dehqonning o'zi va yana bitta narsa (bo'ri yoki echki yoki karam) uchun joy bor. Dehqon yo'qligida echkini bo'ri, karamni echki yeb qo'yishi mumkin. Dehqon uchun bu masalani hal qiling yoki uning yechimi yo'qligini isbotlang. (Eslatma: Dehqon vegetarian, lekin karamni yoqtirmaydi va shuning uchun masalani hal qilish uchun na echkini va na karamni iste'mol qila olmaydi).

32. Yangi dunyo boshqotirmasi. To'rt kishi ko'prikdan o'tishni juda xohlaydi; ularning hammasi ko'prikning bir tomonida turgan deb hisoblang. Ular ikkinchi tomonga o'tishi uchun 17 daqiqa vaqt bor. Tungi payt bo'lganligi uchun ular chiroqdan foydalanishi kerak, ularda



bitta chiroq bor. Ko'prikdan bir vaqtning o'zida ko'pi bilan ikki kishi o'tishi mumkin. Bir yoki ikki kishi ko'prikdan o'tayotganda ularda albatta chiroq bo'lishi shart, uni tashlab bo'lmaydi. 1-shaxs ko'prikdan o'tish uchun 1 daqiqa, 2-shahs 2 daqiqa, 3-shahs 5 daqiqa va 4-shaxs 10 daqiqada o'tadi. Ikki kishi birga yurganda sekinroq yuradigan odamning tezligida birga yurishlari kerak. Ushbu masalaning yechimi uchun eng yaxshi algoritm tuzing va daturini tuzing.

33. Quyidagi formulalardan qaysi birini tomonlarining uzunliklari musbat $a$,

$b$ va $c$ sonlar bo'lgan uchburchakning yuzini hisoblash algoritmi deyish mumkin?

a. $S = \sqrt{p(p-a)(p-b)(p-c)}$, bu yerda $p = {}^{a}\_\_\_\_{}^{+b+c}$. 2

b. $S = \frac{1}{2}bc \sin A$, bu yerda $A - b$ va $c$ tomonlar orasidagi burchak.

c. $S = \frac{1}{2}ah_1\ _a$, bu yerda $h_a - a$ asosga tushirilgan balandlik.

34. Ixtiyoriy $a$, $b$ va $c$ haqiqiy koeffitsientlar uchun $ax^2 + bx + c = 0$ tenglamaning haqiqiy ildizlarini topish algoritmi uchun psevdokod yozing. ($sqrt(x)$ kvadrat ildiz funksiyasi mavjudligini hisobga oling).

35. Musbat o'nlik sanoq sistemasidagi butun sonning ikkilik sanoq sistemasidagi ko'rinishini topishning standart algoritmini yozib bering:

a. o'zbek tilida.

b. psevdokodda.

36. Sevimli bankomatingiz tomonidan naqd pul berishda ishlatiladigan algoritmni tasvirlab bering. (Siz o'zingizning tavsifingizni o'zbek tilida yoki psevdokodda yozishingiz mumkin).

37. a. $\pi$ sonini hisoblash masalasini aniq hal qilish mumkinmi?

b. Bu muammoning yechimi uchun nechta misol keltirish mumkin?

c. Internetdan ushbu muammoning algoritmini qidirib ko'ring.

38. Siz bilgan eng katta umumiy bo'luvchini hisoblash algoritmiga boshqa misol keltiring. Ulardan qaysi biri oddiyroq? Qaysi biri samaraliroq?

## 2-BOB. "BO'L VA ZABT ET" ALGORITMLARI

### 2.1. "Bo'l va zabt et" algoritmi mazmuni.

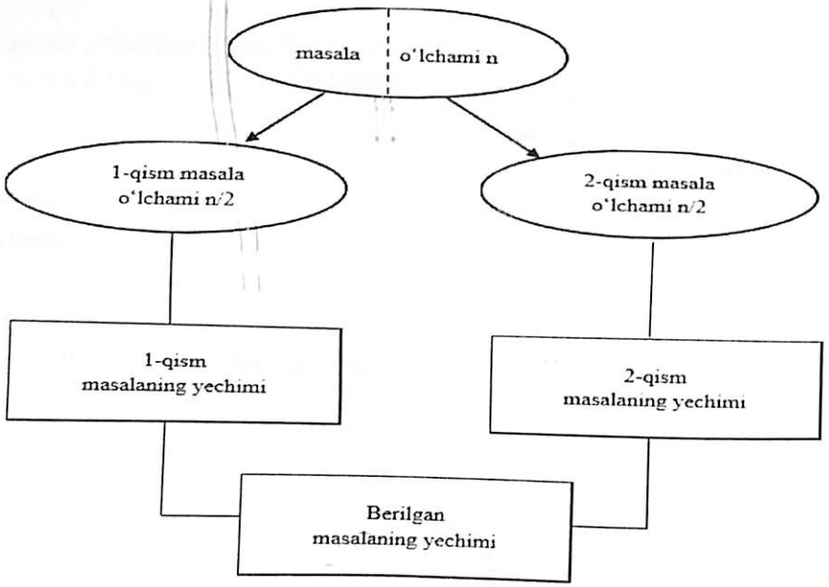
Algoritmlarni loyihalashning "Bo'l va zabt et" strategiyasi uning jozibali nomi tufayli mashhur bo'lsa-da, aslida ham juda ko'plab algoritmlar bu umumiy strategiya asosida qo'llanilganda yuqori samaeadorlikka erishgan. Bu algoritmlar quyidagi umumiy reja asosida ishlaydi:

1.Masalani bir xil turdagi, taxminan teng o'lchamdagi bir nechta kichik masalalarga (qismlarga) bo'lish.

2. Qism masalalarni alohida-alohida yechish (odatda rekursiv chaqiruv asosida, lekin ba'zida, ayniqsa qism masalalar yetarlicha kichik bo'lganda boshqa algoritm qo'llaniladi).

3. Zarurat bo'lsa, qism masalalarning yechimlarini birlashtirib, asosiy masalaning yechimini olish.

2.1-rasmda "Bo'l va zabt et" texnikasi diagrammasi berilgan. Unda qo'yilgan masala ikkita kichik masalaga bo'lish holati ko'rsatilgan. Bu holat bir protsessorli kompyuterda bajarish uchun mo'ljallangan, hozirgacha eng keng tarqalgan holat hisoblanadi.




**2.1-rasm.** *"Bo'l va zabt et" sxemasi (eng oddiy holat).*



Misol tariqasida $n$ ta s00onli $a_0, a_1, \ldots, a_{n-1}$ qatorning yig'indisini hisoblash masalasi qo'yilgan bo'lsin. Agar $n > 1$ bo'lsa, masalani ikkita bir xil masalaga ajratin olishimiz mumkin: birinchi $\lfloor n/2 \rfloor$ ta sonlar yig'indisini hisoblash va qolgan $\lceil n/2 \rceil$ ta sonlar yig'indisini hisoblash. (Albatta, agar $n = 1$ bo'lsa, shunchaki javob sifatida $a_0$ ni olamiz.) Ushbu ikkita yig'indining har biri bir xil usulni rekursiv qo'llash orqali hisoblab chiqilgandan so'ng, ko'rib chiqilayotgan yig'indini olish uchun ularning qiymatlarini qo'shishimiz mumkin:

$$a^0 + \cdots + a_{n-1} = \left(a_0 + \cdots + a_{\lfloor \frac{n}{2}-1 \rfloor}\right) + \left(a_{\lfloor \frac{n}{2} \rfloor} + \cdots + a_{n-1}\right).$$

Bu $n$ ta sonning yig'indisini hisoblashning samarali usulimi? Nima uchun? Sog'lom fikrlaydigan har qanday shaxs bu masala uchun tadbiq qilingan algoritm samarali emas deb javob beradi[7]. Chunki, odatda yig'indilar bu tarzda hisoblanmaydi, shunday emasmi?

Bu yerda biz faqat ketma-ket hisoblashlar misolida "Bo'l va zabt et" algoritmlarini ko'rib chiqsak ham, shuni yodda tutish kerakki, bu algoritm parallel hisoblashlar uchun juda mos keladi, bunda har bir qism masala bir vaqtning o'zida alohida protsessor tomonidan yechilishi mumkin bo'ladi.

Yuqorida ta'kidlab o'tilganidek, bo'lish va zabt etishning eng oddiy holatida berilgan $n$ o'lchamli masala $n/2$ o'lchamdagi ikkita qism masalaga bo'linadi. Umuman olganda, $n$ o'lchamdagi masalani $n/b$ o'lchamdagi $b$ ta qism masalaga bo'lish mumkin, ular bilan $a$ ni hal qilish kerak. (Bu yerda $a$ va $b$ konstantalar, $a \geq 1$ va $b > 1$.) Tahlilimizni soddalashtirish uchun $n$ o'lchamini $b$ ning kuchi deb faraz qilsak, $T(n)$ ish vaqt uchun quyidagi rekkurent munosabatni olamiz:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n), \qquad (2.1)$$

Bu yerda $f(n)$ funksiya $n$ o'lchamdagi masalani $n/b$ o'lchamli qism masalalarga bo'lish va ularning yechimlarini birlashtirishga sarflangan vaqtni hisoblaydi. (Yuqoridagi yig'indini hisoblash masalasi

[7] Taqribiy hisoblashlarda yig'indini "Bo'l va zabt et" algoritmi asosida amalga oshirish to'plangan yaxlitlash xatoligini sezilarli darajada kamaytirishi mumkin.

uchun $a = b = 2$ va $f(n) = 1$). (2.1) rekkurent munosabat ***umumiy bo'l va zabt et rekursiyasi*** deb ataladi. Shubhasiz, uning yechimi $T(n)$ ning o'sish tartibi $a$ va $b$ konstantalarning qiymatlariga hamda $f(n)$ funksiyaning o'sish tartibiga bog'liq. Ko'plab bo'l va zabt et algoritmlarining samaradorligini tahlil qilish quyidagi teorema yordamida sezilarli darajada soddalashtiriladi.

**Ustoz teoremasi.** Agar (2.1) rekkurent munsabatda $f(n) \in \Theta(n^d)$ bo'sa, bu yerda $d \geq 0$ , u holda:

$$T(n) = \begin{cases} \Theta(n^d) & agar\ a < b^d, \\ \Theta(n^d \log n) & agar\ a = b^d, \\ \Theta(n^{\log_b a}) & agar\ a > b^d. \end{cases}$$

Shunga o'xshash natijalar $O$ va $\Omega$ belgilashlar uchun ham amal qiladi.

Masalan, $n = 2^k$ o'lchamli kirish ma'lumotlari uchun bo'l va zabt et usulida yig'indini hisoblash algoritmi (yuqorida ko'rsatilgan) tomonidan amalga oshirilgan qo'shishlar soni $A(n)$ ning rekurrent ifodasi quyidagicha: $A(n) = 2A(n/2) + 1$

Shunday qilib, bu misol uchun $a = 2$, $b = 2$ va $d = 0$; demak, $a > b^d$ bo'lgani uchun:

$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$. Shuni ta'kidlash joizki, biz rekkurent tenglamani yechish mashaqqatiga kirishmasdan turib, yechimning samaradorlik sinfini aniqlashga erishdik. Biroq, albatta, bu yondəshuv yechimning o'sish tartibini faqat noma'lum ko'paytuvchi konstanta doirasida o'rnatishi mumkin. Aksincha, rekurrent tenglamani aniq boshlang'ich shart bilan yechish esa (hech bo'lmaganda $b$ ning darajalari bo'lgan $n$ qiymatlari uchun) aniq javob beradi.

### 2.2. Birlashtirib saralash algoritmi

Birlashtirib saralash (MergeSort) "bo'l va zabt et" prinsipi asosida ishlaydigan algoritmlardan biri hisoblanadi.

*G'oyasi:* Berilgan $A[0..n-1]$ massivni ikkita $A[0..n/2-1]$ va $A[n/2..n-1]$ qismlarga bo'lib, har bir qismni **rekursiv** tarzda



saralaydi, so'ngra ikkita kichikroq saralangan massivni bitta tartiblangan massivga birlashtirish orqali ishlaydi.

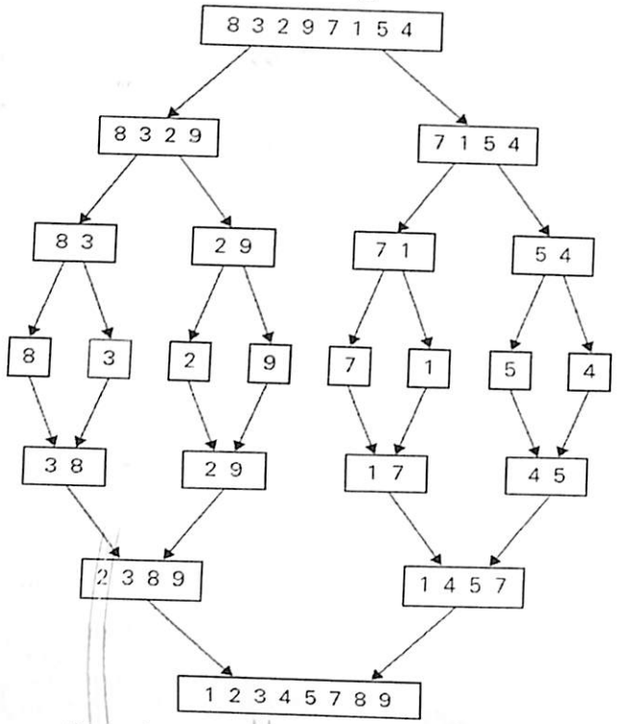
**ALGORITHM** Mergesort(A[0..n−1])
// A[0..n−1] massivni rekursiv birlashtirib saralaydi
// Kirish: A[0..n−1] tartiblanuvchi elementlar massivi // Natija: o'sish tartibida saralangan A[0..n−1] massiv **if** n>1 **copy** A[0..n/2−1] **to** B[0..n/2−1] **copy** A[n/2..n−1] **to** C[0..n/2−1]
**Mergesort**(B[0..n/2−1])
**Mergesort**(C[0..n/2−1])
**Merge**(B,C,A) //birlashtirish funksiyasi

Ikkita tartiblangan massivni birlashtirish quyidagicha amalga oshiriladi. Birlashtirilayotgan massivlarning birinchi elementlarini tanlash uchun ikkita ko'rsatkich (massiv indekslari) o'rnatiladi. Ko'rsatilgan elementlar taqqoslanadi va ularning kichigi yangi tuzilayotgan massivga qo'shiladi; so'ngra kichik element indeksi u nusxalangan massivdagi keyingi elementni ko'rsatish uchun bir birlikka oshiriladi. Bu jarayon berilgan ikkita massivdan biri tugaguncha takrorlanadi, shundan so'ng boshqa massivning qolgan elementlari yangi massivning oxiriga ko'chiriladi.

**ALGORITHM** Merge(B[0..p−1], C[0..q−1], A[0..p+q−1])
//Ikkita saralangan massivni bitta saralangan massivga birlashtiradi
//Kirish: ikkita saralangan B[0..p−1] va C[0..q−1] massivlar
//Chiqish: B va C elementlarining saralangan massivi
A[0..p+q−1] i←0; j←0; k←0
**while** i<p **and** j<q **do**
**if** B[i]≤C[j]
A[k]←B[i];
i←i+1 **else**
A[k]←C[j]; j←j+1
k←k+1 **if** i=p
**copy** C[j..q−1] **to** A[k..p+q−1] **else**

**copy** B[i..p−1] **to** A[k..p+q−1]

Algoritmning ishlash jarayoni {8, 3, 2, 9, 7, 1, 5, 4} massivni saralash misolida 2.2-rasmda tasvirlangan.




**2.2-rasm.** *Birlashtirib saralash (mergesort) algoritmining ishlash sxemasi.*

Birlashtirib saralash qanchalik samarali? Soddalik uchun $n$ ni 2 ning darajasi deb hisoblasak, asosiy taqqoslashlar soni $C(n)$ uchun rekurrent munosabat
quyidagicha bo‘ladi:

$$C(n) = 2C\left(\frac{n}{2}\right) + C_{merge}(n), bunda\ n > 1, C(1) = 0.$$

Keling, $C_{merge}(n)$ ni tahlil qilaylik. Bu birlashtirish bosqichida amalga oshirilgan asosiy taqqoslashlar sonidir. Har bir qadamda aynan bitta taqqoslash amalga oshiriladi, shundan so‘ng qayta ishlanishi kerak bo‘lgan ikkita massivdagi elementlarning umumiy soni 1 taga



kamayadi. Eng yomon holatda, ikkala massiv ham bo‘shab qolmaydi, toki ikkinchisi faqat bitta elementni o‘z ichiga olmaguncha (masalan, kichikroq elementlar galma-gal massivlardan kelishi mumkin). Shuning uchun eng yomon holat uchun $C_{merge}(n) = n - 1$ bo‘ladi va bizda quyidagi
rekursiv tenglik hosil bo‘ladi:

$$C_{yomon}(n) = 2C_{yomon}\left(\frac{n}{2}\right) + n - 1, bunda\ n > 1, C_{yomon}(1) = 0.$$

Shunday qilib, Ustoz (master) teoremasiga ko‘ra, $C_{yomon}(n) = \Theta(n \log n)$ bo’ladi (nima uchun?). Aslida, $n = 2^k$ bo‘lganda eng yomon holatdagi takrorlanishning aniq yechimini topish oson:

$$C_{yomon}(n) = n \log_2 n - n + 1.$$

Eng yomon holatda, birlashtirib saralash (mergesort) algoritmi amalga oshiradigan asosiy taqqoslashlar soni, umumiy taqqoslashga asoslangan har qanday saralash algoritmi uchun nazariy minimal chegara[8] ga juda yaqin keladi. $n$ ning katta qiymatlari uchun, bu algoritmning o‘rtacha holatdagi taqqoslashlar soni taxminan $0{,}25n$ ga kamroq bo‘ladi va shu sababli $O(n \log n)$ ichida joylashadi. Birlashtirib saralashning tezkor saralash (quicksort) va uyum saralash (heapsort) kabi ikkita muhim saralash algoritmlariga nisbatan e’tiborga loyiq afzalligi uning barqarorligidir (ushbu bo‘limdagi mashqlardagi 7-masalaga qarang). Birlashtirib saralashning asosiy *kamchiligi* algoritm talab qiladigan qo‘shimcha xotiraning chiziqli miqdoridir. Birlashtirish jarayoni joyida amalga oshirilishi mumkin bo‘lsada, natijada hosil bo‘lgan algoritm juda murakkab va faqat nazariy jihatdan qiziqarlidir.

Birlashtirishning bir nechta variantlariga olib keladigan ikkita asosiy g‘oya mavjud. Avval algoritmni pastdan yuqoriga massiv elementlari juftliklarini birlashtirish, so‘ngra saralangan juftliklarni birlashtirish orqali amalga oshirish mumkin va h.k. (Agar $n = 2^k$ bo‘lmasa, faqat ozgina hisob-kitob murakkabliklari paydo bo‘ladi.) Bu rekursiv chaqiruvlarni boshqarish uchun stekdan foydalanish vaqti va

[8] $\lceil \log_2 n! \rceil \approx \lceil n \log_2 n - 1.44n \rceil$ ekanligi ko‘rib chiqilgan

xotira sarfini kamaytiradi. Bundan tashqari, biz saralanishi kerak bo‘lgan ro‘yxatni ikkitadan ko‘proq qismga bo‘lishimiz, har birini rekursiv tarzda saralashimiz va keyin ularni birlashtirishimiz mumkin. Ikkilamchi xotira qurilmalarida joylashgan fayllarni saralashda ayniqsa foydali bo‘lgan bu usul ***ko‘p yo‘lli birlashtirib*** saralash deb ataladi.

**Algoritm tahliliga doir namunaviy masala:** $n = 100$ o‘lchamli massivni tasodifiy sonlar bilan to‘ldirib, Merge Sort algoritmi yordamida saralang va uning **vaqt va xotira murakkabligini** tahlil qiling. Amallar sonini, xotira sarfini va boshqa tahlillarni batafsil ko‘rib chiqing.

C++ dasturi: avval $n = 100$ o‘lchamli massivni tasodifiy sonlar bilan to‘ldirib, Merge Sort algoritmi yordamida saralashni amalga oshiramiz.

```
#include <iostream>
#include <vector>
#include
<cstdlib>
#include
<ctime> using
namespace std;
// Ikkita tartiblangan qismini birlashtiruvchi
funksiya void merge(vector<int>& arr, int left,
int mid, int right) {
```



```
  int n1 = mid - left +
1;   int n2 = right -
mid;   // Yordamchi
massivlar
vector<int> L(n1),
R(n2);   for (int i = 0;
i < n1; ++i)      L[i] =
arr[left + i];   for (int
j = 0; j < n2; ++j)
R[j] = arr[mid + 1 + j];
  // Birlashtirish
  int i = 0, j = 0, k = left;
while (i < n1 && j < n2) {
if (L[i] <= R[j]) arr[k++] =
L[i++];      else arr[k++] =
R[j++];
  }
  // Qolganlarini qo‘shish
while (i < n1) arr[k++] =
L[i++];   while (j < n2)
arr[k++] = R[j++];
}
// Merge Sort algoritmi (rekursiv)
void mergeSort(vector<int>& arr, int left,
int right) {   if (left < right) {      int mid =
left + (right - left) / 2;      mergeSort(arr,
left, mid);      mergeSort(arr, mid + 1,
right);      merge(arr, left, mid, right);
  }
}
//massiv      elementlarini      chiqarish
funksiyasi void printArray(vector<int>
&arr, int size) {
  for (int i = 0; i < size; ++i)
{      cout << arr[i] << " ";
  }
  cout << endl;
}      int
main() {
  srand(time(0)); // Tasodifiy sonlar uchun
dastlabki holat      int n = 100; // Massiv
o'lchami   vector<int> arr(n);
  // Massivni tasodifiy sonlar bilan to'ldirish
  for (int i = 0; i < n; ++i) {
```

```
        arr[i] = rand() % 1000;  // 0 dan 999 gacha tasodifiy son
    }
    // Boshlang'ich massivni
chiqarish    cout << "Boshlang'ich
massiv: ";    printArray(arr, n);
    // MergeSortni chaqirish
mergeSort(arr, 0, n - 1);    //
Saralangan massivni
chiqarish    cout <<
"\nSaralangan massiv: ";
    printArray(arr, n);
    return
0; }
```

*Vaqt murakkabligi* (Time Complexity) tahlili. Merge Sort algoritmi **bo'l va zabt et** yondashuvini ishlatadi, har bir rekursiv bosqichda massiv ikki qismga bo'linadi, ya'ni $log_2 n$ qismlarga ajratiladi. Har qism-massiv to'liq ko'rib chiqiladi, ya'ni $n$ ta element qayta birlashtiriladi.

Vaqt murakkabligini hisoblash. Har bir bosqichda $n$ amallarni bajaradi (birlashtirish bosqichi). Rekursiya chuqurligi soni $log_2 n$ ga teng. Shuning uchun Merge Sortning umumiy vaqt murakkabligi - $O(n \log n)$ bilan baholanadi.

Vaqtning amaldagi bajarilishiga ta'sir qiluvchi omillar: $n = 100$ o'lchamli massivda $\log 100 \approx 6$, shuning uchun 6 bosqichli rekursiya mavjud. Har bir bosqichda massivni ikkiga bo'lish va birlashtirish jarayonlari amalga oshiriladi, bu esa $n$ amallarni talab qiladi.

*Xotira murakkabligi* (Space Complexity) tahlili. Merge Sort yordamchi massivlar (L[] va R[])ni ishlatadi. Har bir bosqichda yordamchi massivlar hajmi $n$ bo'ladi. Rekursiya davomida bu yordamchi massivlar har bir bosqichda alohida xotiraga joylashadi.

Xotira murakkabligini hisoblash. Recursiv stek chuqurligi $O(\log n)$ bo'ladi. Yordamchi massivlar - har bir bosqichda $O(n)$ xotira talab qiladi. Shuning uchun umumiy xotira murakkabligi $O(n)$ bo'ladi.



***Amallar soniga nisbatan tahlil.*** $n = 100$ bo'lsa: vaqt murakkabligi: $O(100 \cdot \log 100) \approx O(100 \cdot 6) = O(600)$ amallar bajariladi. Xotira murakkabligi: $O(100)$ xotira sarflanadi (yordamchi massivlar uchun).

### 2.3. Strassenning matritsalarni ko'paytirish usuli

Bo'l va zabt et yondashuvi ikkita butun sonni ko'paytirishda bir xonali ko'paytirishlar sonini kamaytirishini ko'rganimizdan so'ng, matritsalarni ko'paytirishda ham xuddi shunday natijaga erishish mumkinligi ajablanarli emas. Bunday algoritm 1969-yilda V.Strassen tomonidan e'lon qilingan.

Algoritmning asosiy g'oyasi shundaki, biz ikkita 2×2 o'lchamli A va B matritsalarning C ko'paytmasini oddiy usul talab qiladigan sakkizta ko'paytirish o'rniga, atigi yettita ko'paytirish bilan topishimiz mumkin (2.3-bo'limdagi 3misolga qarang). Bu quyidagi formulalardan foydalanish orqali amalga oshiriladi:

$$[cc1000cc1101] = [aa1000aa1101] * [bb1000 \quad bb1101]$$
$$= [m1 + mm42 - + mm54 + m7 \; m1 + mm33 - + mm25 + m6],$$

Bu yerda,

$$m1 = (a00 + a11) * (b00 + b11),$$
$$m2 = (a10 + a11) * b00,$$
$$m3 = a00 * (b01 - b11),$$
$$m4 = a11 * (b10 - b00),$$
$$m5 = (a00 + a01) * b11,$$
$$m6 = (a10 - a00) * (b00 + b01), m7 = (a01 - a11) * (b10 + b11).$$

Shunday qilib, ikkita 2×2 matritsani ko'paytirish uchun Strassen algoritmi yettita ko'paytirish va o'n sakkizta qo'shish/ayirish amalini bajaradi, oddiy usul esa sakkizta ko'paytirish va to'rtta qo'shishni talab etadi. Biroq, bu sonlar bizni 2×2 matritsalarni Strassen algoritmi yordamida ko'paytirishga undamasligi kerak. Uning ahamiyati

matritsaning $n$-tartibi cheksizlikka intilganda namoyon bo‘ladigan asimptotik ustunligidan kelib chiqadi.

Ikkita $n \times n$ o‘lchamli $A$ va $B$ matritsalar berilgan bo‘lsin, bunda $n$ ni 2 ning darajasi deb olamiz. (Agar $n$ 2 ning darajasi bo‘lmasa, matritsalarni nollardan iborat satr va ustunlar bilan to‘ldirish mumkin.) Biz $A$, $B$ va ularning ko‘paytmasi $C$ ni, har birini to‘rtta $n/2 \times n/2$ o‘lchamli qism-matritsalarga quyidagicha ajratishimiz mumkin:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

To‘g‘ri ko‘paytmani olish uchun bu qism-matritsalarni sonlar deb qarash mumkinligini tekshirish qiyin emas. Masalan, $C_{00}$ ni $A_{00} * B_{00} + A_{01} * B_{10}$ yoki $M_1 + M_4 - M_5 + M_7$ ko‘rinishda hisoblash mumkin, bu yerda $M_1$, $M_4$, $M_5$,$M_7$ lar Strassen formulalari bo‘yicha topiladi, sonlar mos qism-matritsalar bilan almashtiriladi. Agar $n/2 \times n/2$ matritsalarning yettita ko‘paytmasi bir xil usulda rekursiv hisoblansa, matritsalarni ko‘paytirish uchun Strassen algoritmiga ega bo‘lamiz.

Ushbu algoritmning asimptotik samaradorligini baholaymiz. Agar $M(n)$ ikkita $n \times n$ (bu yerda $n$ 2 ning darajasi) matritsalarni ko‘paytirishda Strassen algoritmi tomonidan amalga oshirilgan ko‘paytirishlar soni bo‘lsa, u uchun quyidagi rekurrent munosabatni olamiz:

$$M(n) = 7M\left(\frac{n}{2}\right), n > 1, M(1) = 1.$$

$n = 2^k$ bo’lgani uchun,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2}) = \cdots = 7^iM(2^{k-i}) \ldots = 7^kM(2^{k-k}) = 7^k.$$

$k = \log_2 n$ bo’lgani uchun,

$$M(n) = 7\log 2\, n = n\log 2\, 7 \approx n2.807,$$

Bu qo‘pol kuch algoritmi talab qiladigan $n^3$ dan kichikroq hisoblanadi.



Bu ko‘paytirishlar sonini tejash qo‘shimcha qo‘shishlarni amalga oshirish hisobiga erishilganligi sababli, Strassen algoritmi bo‘yicha amalga oshirilgan qo‘shishlar soni $A(n)$ ni tekshirishimiz kerak. $n > 1$ o’lchamli ikkita matritsani ko‘paytirish uchun algoritm $n/2$ o’lchamli yettita matritsani ko‘paytirishi va o‘lchami $n/2$ bo‘lgan matritsalarni 18 marta qo‘shishi/ayirishi kerak; $n = 1$ bo‘lganda qo‘shish amalga oshirilmaydi, chunki ikkita son shunchaki ko‘paytiriladi. Shu tahlillar asosida quyidagi rekkurent munosabatga ega bo’lamiz:

$$A(n) = 7A\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2, n > 1, A(1) = 0.$$

Ushbu rekursiyaning yopiq shaklda yechimini olish mumkin bo‘lsa-da (shu bo‘limdagi 8-masalaga qarang), biz bu yerda faqat yechimning o‘sish tartibini aniqlaymiz. Ustoz teoremasiga ko‘ra, $A(n) \in \Theta(n^{\log 2\, 7})$ ga teng. Boshqacha aytganda, qo‘shishlar sonining o‘sish tartibi ko‘paytirish sonining o‘sish tartibiga teng. Bu Strassen algoritmini $\Theta(n^{\log 2\, 7})$ samaradorlik sinfiga kiritadi, bu esa oddiy usulning $\Theta(n^3)$ samaradorlik sinfidan yaxshiroqdir.

Strassen kashfiyotidan beri, haqiqiy sonlarning ikkita $n \times n$ matritsasini

$O(n^\alpha)$ vaqt ichida ko‘paytirishning tobora kichikroq $\alpha$ doimiylari bilan bir nechta yangi algoritmlar yaratildi. Hozirgacha eng tezkor algoritm Kupersmit va Vinograd

[Coo87] algoritmi bo‘lib, uning samaradorligi $O(n^{2.376})$ ga teng. Ko‘rsatkichlarning pasayishi ushbu algoritmlarning murakkablashuvi evaziga yuz berdi.

Ko‘rsatkichlarning kamayishi ushbu algoritmlarning murakkablashuvi hisobiga erishilgan. Katta ko‘paytiruvchi o‘zgarmaslar tufayli, ularning hech biri amaliy qiymatga ega emas. Biroq, ular nazariy nuqtai nazardan qiziqarli. Bir tomondan, ular matritsalarni ko‘paytirish uchun ma’lum bo‘lgan eng yaxshi nazariy quyi chegara – $n^2$ ko‘paytirishga tobora yaqinlashmoqda, garchi bu chegara va mavjud eng yaxshi algoritm o‘rtasidagi farq hali yechilmagan. Boshqa tomondan, matritsalarni

ko‘paytirish boshqa muhim masalalar, jumladan chiziqli tenglamalar tizimini yechish bilan hisoblash jihatidan teng kuchli ekanligi aniqlangan.

**Algoritmning tahliliga doir namunaviy masala:** Strassenning matritsalarni ko‘paytirish algoritmi - bu bo‘l va zabt et (Divide and Conquer) tamoyilini asoslangan eng mashhur va samarali matritsalarni ko‘paytirish algoritmlaridan biridir. Bu algoritm standart $O(n^3)$ vaqt murakkabligini pasaytiradi, bu esa matritsalarni katta o‘lchamda ko‘paytirishda sezilarli samaradorlikni ta'minlaydi.

Keling, Strassenning matritsalarni ko‘paytirish algoritmining ishglashini, algoritmik tahlilini quyidagi matritsalarni ko‘paytirish misolida ko‘rib chiqamiz. Berilgan matritsalar:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \times \begin{bmatrix} 16 & 15 & 14 & 13 \\ 12 & 11 & 10 & 9 \\ 8 & 7 & 6 & 5 \\ 4 & 3 & 2 & 1 \end{bmatrix}$$

Strassen algoritmining C++ dasturi:

```
#include
<iostream>
#include
<vector> using
namespace std;
typedef vector<vector<int>> Matrix;
// Matritsa qo‘shish
Matrix add(Matrix A, Matrix
B) {    int n = A.size();
  Matrix                C(n,
vector<int>(n));     for (int i
= 0; i < n; i++)
      for (int j = 0; j < n; j++)
C[i][j] = A[i][j] + B[i][j];
return C;
}
```



```
// Matritsa ayirish
Matrix subtract(Matrix A, Matrix B) {
   int n = A.size();
   Matrix C(n,
vector<int>(n));    for (int
i = 0; i < n; i++)       for
(int j = 0; j < n; j++)
C[i][j] = A[i][j] - B[i][j];
  return
C; }
// Oddiy (an’anaviy) matritsa ko‘paytirish (bazaviy holat uchun)
Matrix conventionalMultiply(Matrix A, Matrix
B) {    int n = A.size();
  Matrix C(n, vector<int>(n,
0));    for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
for (int k = 0; k < n; k++)
C[i][j] += A[i][k] * B[k][j];
return C;
}
// Strassen algoritmi
Matrix strassen(Matrix A, Matrix B) {
  int     n     =
A.size();    if (n
<= 2)
     return conventionalMultiply(A,
B);    int k = n / 2;
  Matrix A11(k,     vector<int>(k)),     A12(k,
vector<int>(k)),     A21(k,     vector<int>(k)),     A22(k,
vector<int>(k));
  Matrix B11(k,     vector<int>(k)),     B12(k,
vector<int>(k)),     B21(k, vector<int>(k)), B22(k,
vector<int>(k));    for (int i = 0; i < k; i++) {      for (int j = 0; j < k;
j++) {       A11[i][j] = A[i][j];
```

```
            A12[i][j] = A[i][j + k];
            A21[i][j] = A[i + k][j];
            A22[i][j] = A[i + k][j + k];
            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + k];
            B21[i][j] = B[i + k][j];
            B22[i][j] = B[i + k][j + k];
        }
    }
    Matrix P1 = strassen(add(A11, A22), add(B11, B22));
    Matrix P2 = strassen(add(A21, A22), B11);
    Matrix P3 = strassen(A11, subtract(B12, B22));
    Matrix P4 = strassen(A22, subtract(B21, B11));
    Matrix P5 = strassen(add(A11, A12), B22);
    Matrix P6 = strassen(subtract(A21, A11), add(B11, B12));
    Matrix P7 = strassen(subtract(A12, A22), add(B21, B22));
    Matrix C11 = add(subtract(add(P1, P4), P5), P7);
    Matrix C12 = add(P3, P5);
    Matrix C21 = add(P2, P4);
    Matrix C22 = add(subtract(add(P1, P3), P2), P6);
    Matrix C(n,
vector<int>(n));    for
(int i = 0; i < k; i++) {
for (int j = 0; j < k; j++) {
C[i][j] = C11[i][j];
        C[i][j + k] = C12[i][j];
        C[i + k][j] = C21[i][j];
        C[i + k][j + k] = C22[i][j];
    }
}
return
C;
}
```



```
// Natijani chiqarish void
printMatrix(Matrix M) {
for (const auto& row :
M) {      for (int val :
row)          cout << val
<< "\t";      cout <<
"\n";
  } } int
main() {
  Matrix A = {{1, 2, 3, 4},
              {5, 6, 7, 8},
              {9, 10, 11, 12},
              {13, 14, 15, 16}};
  Matrix B = {{16, 15, 14, 13},
              {12, 11, 10, 9},
              {8, 7, 6, 5},
              {4, 3, 2, 1}};
  Matrix C = strassen(A, B);
cout << "Natijaviy C
matritsa:\n";
printMatrix(C);    return 0;
}
```

Dastur natijasi:

Natijaviy C matritsa:

80 70 60 50

240 214 188 162

400 358 316 274

560 502 444 386

**Vaqt murakkabligi (nazariy)** - har bosqichda 7 ta rekursiv chaqiruv bo'ladi, ya'ni: $T(n) = 7\,T(n/2) + O(n^2)$. Matritsa o'lchami $n = 4$ bo'lgani uchun algoritm quyidagi tarzda ishlaydi:

$$T(4) = 7 * T(2) + O(4^2)$$

$$T(2) = 7 * T(1) + O(2^2)$$

$$T(1) = O(1)$$

```
            A12[i][j] = A[i][j + k];
            A21[i][j] = A[i + k][j];
            A22[i][j] = A[i + k][j + k];
            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + k];
            B21[i][j] = B[i + k][j];
            B22[i][j] = B[i + k][j + k];
        }
    }
    Matrix P1 = strassen(add(A11, A22), add(B11, B22));
    Matrix P2 = strassen(add(A21, A22), B11);
    Matrix P3 = strassen(A11, subtract(B12, B22));
    Matrix P4 = strassen(A22, subtract(B21, B11));
    Matrix P5 = strassen(add(A11, A12), B22);
    Matrix P6 = strassen(subtract(A21, A11), add(B11, B12));
    Matrix P7 = strassen(subtract(A12, A22), add(B21, B22));
    Matrix C11 = add(subtract(add(P1, P4), P5), P7);
    Matrix C12 = add(P3, P5);
    Matrix C21 = add(P2, P4);
    Matrix C22 = add(subtract(add(P1, P3), P2), P6);
    Matrix C(n,
vector<int>(n));    for
(int i = 0; i < k; i++) {
for (int j = 0; j < k; j++) {
C[i][j] = C11[i][j];
        C[i][j + k] = C12[i][j];
        C[i + k][j] = C21[i][j];
        C[i + k][j + k] = C22[i][j];
    }
}
return
C;
}
```



```
// Natijani chiqarish void
printMatrix(Matrix M) {
for (const auto& row :
M) {       for (int val :
row)          cout << val
<< "\t";      cout <<
"\n";
  } } int
main() {
  Matrix A = {{1, 2, 3, 4},
              {5, 6, 7, 8},
              {9, 10, 11, 12},
              {13, 14, 15, 16}};
  Matrix B = {{16, 15, 14, 13},
              {12, 11, 10, 9},
              {8, 7, 6, 5},
              {4, 3, 2, 1}};
  Matrix C = strassen(A, B);
cout << "Natijaviy C
matritsa:\n";
printMatrix(C);    return 0;
}
```

Dastur natijasi:

```
Natijaviy C matritsa:
80    70    60    50
240   214   188   162
400   358   316   274
560   502   444   386
```

**Vaqt murakkabligi (nazariy)** - har bosqichda 7 ta rekursiv chaqiruv bo‘ladi, ya’ni: $T(n) = 7\,T(n/2) + O(n^2)$. Matritsa o‘lchami $n = 4$ bo‘lgani uchun algoritm quyidagi tarzda ishlaydi:

$$T(4) = 7 * T(2) + O(4^2)$$
$$T(2) = 7 * T(1) + O(2^2)$$
$$T(1) = O(1)$$

Shuning uchun umumiy **rekursiya soni** - 1-daraja: 7 ta $T(2)$, 2-daraja: 7×7=49 ta T(1); **ko'paytirishlar soni (asosiy amallar)** - har bir $T(1)$ da bitta ko'paytirish → 49 ta ko'paytirish, qolgan $O(n^2)$ qo'shish-ayirish amallar (+, -): har bosqichda ~18 ta, 3 bosqichda: ~18×2=36 ta qo'shish-ayirish. Shunday qilib jami amallar soni (ko'paytirish + qo'shish/ayirish) - 49 ta ko'paytirish + ~36 ta qo'shishayirish = **~85 amal**

**Xotira murakkabligi** - har bir rekursiv chaqiriqda ajratiladigan qo'shimcha xotira - $n/2 \times n/2$ o'lchamli yordamchi matritsalar → 7 ta $P$ matritsa, 2-3 ta vaqtinchalik ($tempA/tempB$), har bosqichda ~10 ta 2×2 matritsa saqlanadi → 10×4=40 element, **rekursiya chuqurligi** - $n = 4 \rightarrow 2 \rightarrow 1 \rightarrow 3$.

Umumiy xotira sarfi (elementlar soni bo'yicha) - 3 daraja×40 element=~120 ta element → int turida: 120×4 bayt = ~480 bayt

An'anaviy va Strassen ko'paytiish algoritmini taqqoslash

| Turi | Ko'paytirishlar soni | Qo'shish/Ayirish | Umumiy amallar | Xotira |
|---|---|---|---|---|
| An'anaviy | 64 | 48 | 112 | 0 (inline) |
| **Strassen** | 49 | ~36 | ~85 | ~480 bayt |

Xulosa qilib aytganda, **Strassen algoritmi** katta o'lchamli matritsalar uchun samarali (masalan $n \geq 64$), yuqorida berilgan masaladagi $n = 4$ kabi kichik holatlarda **an'anaviy usul** tezroq va kam xotira talab qiladi.

### 2.4. Mustaqil ishlash uchun savollar va mashqlar

**1.** ***a***. $n$ ta sondan iborat massivdagi eng katta elementning joylashuv o'rnini topish uchun "bo'l va zabt et" algoritmining psevdokodini yozing.

*b.* Siz tuzgan algoritm bir nechta elementida eng katta qiymat mavjud bo'lgan
massivlar uchun qanday natija beradi?



*c.* Siz tuzgan algoritm bajaradigan asosiy taqqoslashlar soni uchun rekurrent
munosabatni tuzib, yechimini toping.

*d.* Ushbu algoritm mazkur masala uchun "dag'al kuch" algoritmi (to'g'ridan-
to'g'ri qidirish) bilan qanday taqqoslanadi?

**2.** ***a***. $n$ ta sondan iborat massivdagi ham eng katta, ham eng kichik elementlarning qiymatlarini topish uchun "bo'l va zabt et" algoritmining psevdokodini yozing.

*b.* Algoritmingiz bajaradigan asosiy taqqoslashlar soni uchun rekurrent
munosabatni tuzib, yechimini toping ($n = 2^k$ uchun).

*c.* Ushbu algoritm mazkur masala uchun "dag'al kuch" algoritmi bilan qanday
taqqoslanadi?

**3.** ***a***. $a^n$ ni hisoblash (darajaga ko'tarish masalasi) uchun "bo'l va zabt et" algoritmining psevdokodini yozing, bunda $n$ musbat butun son hisoblanadi.

*b.* Ushbu algoritm bajaradigan ko'paytirishlar soni uchun rekurrent
munosabatni tuzib, yechimini toping.

*c.* Ushbu algoritm mazkur masala uchun "dag'al kuch" algoritmi bilan qanday
solishtiriladi?

**4.** Algoritmning samaradorlik sinfini tahlil qilishda paydo bo'ladigan aksariyat holatlarda logarifmning asosi ahamiyatsizdir. Bu xususiyat Ustoz teoremasining logarifmlarni o'z ichiga olgan ikkala tasdiqlariga ham taalluqlimi?

**5.** Quyidagi rekursiv munosabatlar yechimlarining o'sish tartibini aniqlang.

*a.* $T(n) = 4T(n/2) + n, T(1) = 1$

*b.* $T(n) = 4T(n/2) + n^2, T(1) = 1$

*c.* $T(n) = 4T(n/2) + n^3, T(1) = 1$

**6.** E, X, A, M, P, L, E ro‘yxatni alifbo tartibida saralash uchun mergesortni qo‘llang.

**7.** Mergesort turg‘un saralash algoritmi hisoblanadimi?

**8.** ***a***. Eng yomon holatda birlashtirib saralash orqali amalga oshirilgan asosiy taqqoslashlar soni uchun rekurrent munosabatni yeching. Siz $n = 2^k$ deb qabul qilishingiz mumkin.

***b***. Eng yaxshi holatlarda kiruvchi ma’lumotlarni birlashtirish orqali amalga

oshirilgan asosiy taqqoslashlar soni uchun rekurrent munosabat o‘rnating va uni $n = 2^k$ uchun yeching.



# 3-BOB. “OCHKO‘Z” ALGORITMI

## 3.1. Tangalarni almashtirish masalasi

Keling, butun dunyo bo‘ylab millionlab kassirlar tomonidan hech bo‘lmaganda ongsiz ravishda duch keladigan ***tangalarni almashtirish masalasini*** ko‘rib chiqaylik: ma'lum bir $n$ miqdordagi pulni $d1 > d2 > . . . > dm$ nominalidagi tangalarning eng kam soni bilan almashtiring. Masalan, O’zbekistonda keng qo‘llanilayotgan pul birliklari $d1 = 10$ (ming), $d2 = 5$ (ming), $d3 = 2$ (ming) va $d4 = 1$ (ming) nominallarda mavjud. Masalan, $n = 28$ (yigirma sakkiz ming) so‘mni yuqorida berilgan pul nominallari bilan qanday almashtirish mumkin? Agar siz 2 ta $d1$, 1 ta $d2$, 1 ta $d3$ va 1 ta $d4$ degan javobni tanlasangiz, ongli ravishda mavjud muqobillar orasidan eng yaxshi tanlovlar ketma-ketligini yaratishning mantiqiy strategiyasiga amal qilgansiz. Darhaqiqat, birinchi bosqichda siz to‘rtta nominaldan eng kattasini tanlashdan boshlashimiz kerak edi. "Ochko‘z" fikrlash natijasida ikkita $d1$ nominalni tanlashga olib keladi, chunki u qolgan miqdorni eng ko‘p, ya'ni 8 (mingga) kamaytiradi. Ikkinchi bosqichda ham barcha nominallar bor, lekin $d1$ ni tanlay olmaymiz, chunki bu muammoning cheklovlarini buzgan bo‘lar edi. Shunday qilib, bu bosqichda eng yaxshi tanlov 1 ta $d2$ nominal, qolgan miqdorni 3 mingga kamaytiradi. Yana bitta $d3$ va bitta $d4$ tanlab olinadi. Yig‘indisi 28 ga teng bo‘lgan nominallarning mumkin bo’lgan holatlari ro‘yxati:

| № | $d1 = 10$ | $d2 = 5$ | $d3 = 2$ | $d4 = 1$ | Tangalar soni |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 12 | 15 |
| 2 | 1 | 1 | 1 | 11 | 14 |
| 3 | 1 | 1 | 2 | 9 | 13 |
| 4 | 1 | 1 | 3 | 7 | 12 |
| 5 | 1 | 1 | 4 | 5 | 11 |
| 6 | 1 | 2 | 1 | 6 | 10 |
| 7 | 1 | 2 | 2 | 4 | 9 |
| 8 | 1 | 2 | 3 | 2 | 8 |
| 9 | 1 | 3 | 1 | 1 | 6 |

| 10 | 2 | 1 | 1 | 1 | 5 |
|---|---|---|---|---|---|

Bu yechim optimalmi? Ha shunday, optimal yechim. Darhaqiqat, ochko'z algoritm ushbu pul nominallari bilan har bir musbat butun son uchun optimal yechimni berishini isbotlash mumkin. Shu bilan birga, ba'zi miqdorlar uchun optimal yechimni keltirmaydigan pul nominallariga misol keltirish oson - masalan, $d1 = 1$ ta, $d2 = 3$ ta, $d3 = 1$ ta, $d4 = 1$ va $n = 28$, nominallar soni 6 ta bo'ladi.

Tangalarni hisoblash masalasida tanlovni nominallarning eng kattasidan boshlash kabi qo'llaniladigan yondashuv ***ochko'zlik*** deb ataladi. Ochko'zlik yondashuvi, muammoning to'liq yechimiga erishilgunga qadar, har biri shu paytgacha olingan qisman qurilgan yechimni kengaytiradigan bosqichlar ketmaketligi orqali yechim qurishni taklif qiladi. Har bir qadamda tanlov quyidagilardan biri bo'lishi kerak:

- amalga oshirilishi mumkin, ya'ni masalaning cheklovlarini qanoatlantirishi kerak;
- lokal optimal, ya'ni ushbu bosqichda mavjud barcha mumkin bo'lgan tanlovlar orasida eng yaxshi lokal tanlov bo'lishi kerak;
- qaytarib bo'lmaydigan, ya'ni bir marta qilingan bo'lsa, uni algoritmning keyingi bosqichlarida o'zgartirib bo'lmaydigan bo'lishi kerak.

Ushbu talablar texnikaning nomini izohlaydi: har bir qadamda lokal optimal tanlovlar ketma-ketligi butun masalaga (global) optimal yechim beradi degan maqsadda, mavjud bo'lgan eng yaxshi muqobilni "ochko'zlik bilan" olishni taklif qiladi. Biz ochko'zlikning yaxshi yoki yomonligi haqida falsafiy bahs yuritishdan o'zimizni tiyamiz. Bizning algoritmik nuqtayi nazardan, bunday ochko'z strategiya ish beradimi yoki yo'qmi, degan savol tug'iladi. Ko'rib turganimizdek, shunday masalalar mavjudki, ular uchun lokal optimal tanlovlar ketma-ketligi ko'rib chiqilayotgan masalaning har bir holati uchun optimal yechim beradi. Biroq, bunday bo'lmagan boshqa masalalar ham mavjud; bunday masalalar uchun, agar bizni taxminiy yechim qiziqtirsa yoki qanoatlantirishi kerak bo'lsa, ochko'z algoritm muhim ahamiyat kasb etishi mumkin.



Bobning dastlabki ikki bo'limida *minimal qamrov daraxtini qurish*[9] masalasi uchun ikkita klassik algoritm: Prim algoritmi va Kruskal algoritmi muhokama qilamiz. Ushbu algoritmlarning diqqatga sazovor tomoni shundaki, ular ochko'zlik yondashuvini ikki xil usulda qo'llash orqali bir xil masalani hal qiladi va ikkalasi ham har doim optimal yechim beradi. Shunindek, yana bir klassik algoritm - vaznli grafda eng qisqa - yo'l masalasi uchun Deykstra algoritmini kiritamiz.

Qoida tariqasida, ochko'z algoritmlar ham intuitiv jozibador, ham sodda bo'ladi. Optimallashtirish masalasini hisobga olgan holda, odatda muammoning bir nechta kichik holatlarini ko'rib chiqqandan so'ng, ochko'zlarcha qanday harakat qilish kerakligini aniqlash oson bo'ladi. Odatda, ochko'z algoritm optimal yechim berishini isbotlash ancha qiyinroq hisoblanadi. Buni amalga oshirishning keng tarqalgan usullaridan biri keying bo'limda keltirilgan isbotda tasvirlangan: matematik induksiya yordamida, ochko'z algoritm har bir iteratsiyada olgan qisman qurilgan yechimni masalaning optimal yechimigacha kengaytirish mumkinligini ko'rsatib berilgan.

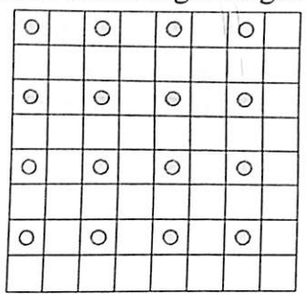
Ochko'z algoritmning optimalligini isbotlashning ikkinchi usuli shundaki, u har bir qadamda muammoning maqsadiga erishishda hech bo'lmaganda boshqa har qanday algoritmdan yaxshiroq harakat qilishini ko'rsatadi. Misol tariqasida quyidagi masalani ko'rib chiqamiz: shaxmat otining 100×100 o'chamdagi taxtaning bir burchagidan diagonal qarama-qarshi burchagiga o'tishi uchun zarur bo'lgan yurishlarning minimal sonini toping. (Otning harakatlari L-simon sakrashlardan iborat bo'lib: gorizontal yoki vertikal yo'nalishda ikkita kvadrat va perpendikulyar yo'nalishda bitta kvadrat). Bu yerda ochko'z yechim aniq: har bir harakatda maqsadga iloji boricha yaqinroq sakrash. Demak, uning boshlang'ich va oxirgi kvadratlari mos ravishda (1,1) va (100, 100) bo'lsa, 66 tadan iborat ketma-ketlikda
quyidagicha harakatlanadi: {(1,1) → (3,2) → (4,4) →. . . → (97,97) → (99,98) →

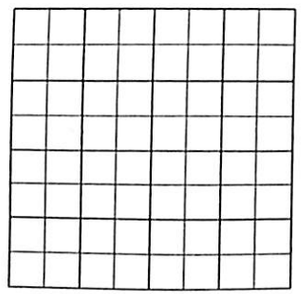
[9] Fan dasturi bo'yicha minimal karkas daraxtini qurish

(100,100)} masalaning yechimi. (Ikki yurishli ilgarilanmalar soni $k$ ni $1 + 3k = 100$ tenglamadan hosil qilish mumkin.) Nima uchun bu minimum harakatli yechim?

Chunki agar biz maqsadgacha bo'lgan masofani Manxetten masofasi bilan o'lchasak, ya'ni ko'rib chiqilayotgan ikkita kvadratning satr raqamlari va ustun raqamlari o'rtasidagi farqning yig'indisi bo'lsa, ochko'z algoritm har bir harakatda uni 3 ga kamaytiradi - bu otning qo'lidan keladigan eng yaxshi narsa.

Uchinchi yo'l - ochko'z algoritmning yakuniy natijasi algoritmning ishlash usuliga emas, balki uning natijasiga ko'ra optimal ekanligini ko'rsatishdir. Misol tariqasida 8×8 o'lchamli shaxmat doskaga maksimal miqdordagi shashka donalarini joylashtirish masalasini ko'rib chiqamiz, shunda ikkita shashka donasi bir xil yoki yonma-yon - vertikal, gorizontal yoki diagonal kvadratlarga joylashtirilmaydi. Ochko'zlik strategiyasiga amal qilish uchun biz har bir yangi shashka donasini keyingi donalar uchun imkon qadar ko'proq bo'sh kvadratlar qoldirish uchun joylashtirishimiz kerak. Masalan, doskaning yuqori chap burchagidan boshlab, 9.1arasmda ko'rsatilganidek, 16 ta shashka donasini joylashtirishimiz mumkin. Nima uchun bu yechim optimal? Buning sababini ko'rish uchun taxtani 9.1b-rasmda ko'rsatilganidek 16 ta 4×4 kvadratga bo'ling. Shubhasiz, bu kvadratlarning har biriga bittadan ortiq donani joylashtirish mumkin emas, bu esa doskadagi qo'shni bo'lmagan shashkalarning umumiy soni 16 tadan oshmasligini anglatadi.







**3.1-rasm** *(a) 16 ta donani qo'shni bo'lmagan kvadratlarga joylashtirish. (b) 16 tadan ortiq chipni joylashtirishning iloji yo'qligini isbotlovchi plataning bo'linishi.*

**Algoritmning tahlili uchun namunaviy misol**: Tangalarni almashtirish masalasi (Coin Change Problem) uchun ochko'z yondashuv eng katta pul birligini tanlab, kerakli summani kamroq sondagi nominallar bilan almashtirish tamoyiliga asoslanadi. Masala: Sizda quyidagi nominaldagi pul birliklari mavjud: $d1 = 10$ (ming), $d2 = 5$ (ming), $d3 = 2$ (ming), $d4 = 1$ (ming) so'm. Berilgan miqdor: $n =$ 28 ming so'mni **eng kam sonli nomanallar** bilan ifodalash.

**Ochko'z yondashuv uchun psevdokod:**

**Algorithm** CoinChangeGreedy(*n, nominals*):
//Berilgan miqdorni eng kam sondagi nominallar
//bilan almashtirish masalasini yechadi
//Kirish: Kamayish tartibida saralangan nominallar
//Chiqish: Almashtirish mumkin bo'lgan nominallar va ularning soni
1. Nominallarni kamayish tartibida saralash
2. **for** each coin **in** nominals: count ← n/coin;
   **print**(count and coin); n←n **mod** coin;
3. **End**

C++ dasturlash tilida:

```
#include <iostream> #include <vector> using namespace
std; void coinChangeGreedy(int amount, vector<int>
&denominations) {    cout << "Berilgan summa: " <<
amount << " ming so'm\n";    cout << "Tangalar taqsimoti
(ochko'z yondashuv):\n";
  for      (int      coin      :
denominations) {      int count
= amount / coin;
    if (count > 0) {
      cout << coin << " ming so'mlikdan: " << count << "
ta\n";          amount %= coin;
    }
```

```
}
if (amount > 0)        cout << "Qolgan summa: " << amount <<
" so'm (qoplab bo'lmaydi)\n";
} int main() {    vector<int> coins = {10,
5, 2, 1}; // ming so'mda    int n = 28; //
ming so'm    coinChangeGreedy(n, coins);
return 0; }
```

**Natija:**

Berilgan summa: 28 ming so'm
Tangalar taqsimoti (ochko'z yondashuv):
10 ming so'mlikdan: 2 ta
5 ming so'mlikdan: 1 ta
2 ming so'mlikdan: 1 ta
1 ming so'mlikdan: 1 ta

**Tahlil:** amallar soni: $O(m)$, bu yerda $m$ - pul birliklari soni, vaqt murakkabligi - $O(m)$, xotira murakkabligi: $O(1)$ (asosiy massivda nominal qiymatlar saqlanadi).

### 3.2. Minimal qamrov daraxtlarini qurish masalalari

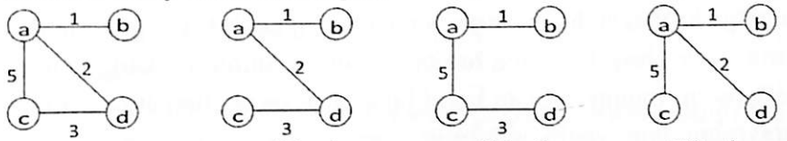
Ko'pgina amaliy vaziyatlarda quyidagi masala tabiiy ravishda paydo bo'ladi:

berilgan $n$ ta nuqtalarni iloji boricha kam xarajatli usulda shunday tutashtiringki, har bir nuqtalar juftliklari orasida yo'l bo'lsin. U aloqa, kompyuter, transport va elektr tarmoqlarini o'z ichiga olgan barcha turdagi tarmoqlarni loyihalashda to'g'ridanto'g'ri qo'llaniladi. Ma'lumotlar to'plamlaridagi nuqtalar klasterlarini identifikatsiyalaydi. U arxeologiya, biologiya, sotsiologiya va boshqa fanlarda tasniflash maqsadida qo'llanilgan. Shuningdek, u sayohatchi sotuvchi (kommivoyajer) masalasi kabi murakkabroq muammolarning taxminiy yechimlarini qurishda ham foydali.

Biz grafning uchlari bilan berilgan nuqtalarni, grafning yoylari bilan mumkin bo'lgan bog'lanishlarni va yoylarning vaznlari bilan bog'lanish xarajatlarini ifodalashimiz mumkin. U holda masalani minimal qamrovli daraxt masalasi sifatida qo'yish mumkin va u formal ravishda quyidagicha ta'riflanadi.



**Ta'rif.** Yo'naltirilmagan bog'langan grafning qamrov daraxti - bu grafning barcha uchlarini o'z ichiga olgan bog'langan asiklik qism grafdir (ya'ni daraxt). Agar bunday grafning yoylariga vaznlar qo'yilgan bo'lsa, minimal qamrovli daraxt uning eng kichik vazndagi qamrovli daraxti bo'lib, bunda daraxtning vazni uning barcha yoylaridagi vaznlarning yig'indisi sifatida aniqlanadi. Minimal qamrov daraxti masalasi - bu berilgan vaznli bog'langan graf uchun minimal qamrov daraxtini topish masalasi hisoblanadi. 3.2-rasmda ushbu tushunchalarni aks ettiruvchi oddiy misol keltirilgan.




graf $w(T_1) = 6$ $w(T_2) = 9$ $w(T_3) = 8$

**3.2-rasm.** *Graf va uning qamrov daraxtlari, bunda T1 minimal qamrov daraxti bo'ladi.*

Agar chuqur izlanishlar natijasida minimal daraxtni qurishga urinib ko'rsak, ikkita jiddiy to'siqqa duch kelamiz. Birinchidan, daraxtlar soni graf o'lchamiga qarab eksponensial ravishda o'sadi (hech bo'lmaganda zich graflar uchun). Ikkinchidan, berilgan graf uchun barcha qamrov daraxtlarni hosil qilish oson emas; aslida, bu masala uchun mavjud bo'lgan bir nechta samarali algoritmlar mavjud. Masalan, Prim algoritmi, Kruskal algoritmi, Deykstra algoritmi va boshqalar vaznli graf uchun minimal qamrovli daraxtni qurish masalalarini samarali yechimga olib keladi.

### 3.3. Prim algoritmi

Prim algoritmi kengayuvchi kichik daraxtlar ketma-ketligi orqali minimal qamrovli daraxtni quradi. Bunday ketma-ketlikdagi boshlang'ich qism daraxt graf tugunlarining $V$ to'plamidan ixtiyoriy ravishda tanlangan bitta tugundan iborat bo'ladi. Har bir qadamda

algoritm joriy daraxtni ochko‘zlik bilan kengaytiradi, shunchaki unga ushbu daraxtda bo‘lmagan eng yaqin tugunni[10] biriktiradi.

Grafning barcha tugunlari qurilayotgan daraxtga kiritilgandan so‘ng algoritm to‘xtaydi. Algoritm daraxtni har bir qadamida aniq bitta tugunga kengaytirgani uchun bunday qadamlarning umumiy soni $(n-1)$ ga teng bo‘ladi, bu yerda $n$ -
grafdagi tugunlar soni. Algoritm ishlashi natijasida hosil qilingan daraxt, daraxt kengaytmalari uchun ishlatiladigan yoylar to‘plami sifatida olinadi. Prim algoritmining mohiyati shundan iboratki, joriy qism daraxtga kirmagan har bir tugunni daraxt tuguni bilan bog‘laydigan eng qisqa yoy haqidagi ma’lumot bilan ta’minlash zarur. Bunday ma’lumotni tugunga ikkita belgi biriktirish orqali berishimiz mumkin: daraxtning eng yaqin uchining nomi va tegishli yoyning uzunligi (vazni). Joriy daraxtning hech bir tuhuniga qo‘shni bo‘lmagan tugunlarga ularning daraxt tugunlarigacha bo‘lgan "cheksiz" masofasini ko‘rsatuvchi $\infty$ belgisi va eng yaqin daraxt tuguni nomi uchun bo‘sh belgi berilishi mumkin. (Shuningdek, biz daraxtda bo‘lmagan tugunlarni ikki to‘plamga, "chetgi" va "ko‘rinmas"ga ajratishimiz mumkin. Chet faqat daraxtda bo‘lmagan, lekin daraxtning kamida bitta tuguniga qo‘shni bo‘lgan tugunlarni o‘z ichiga oladi. Bular daraxtning navbatda tanlanadigan tuguniga nomzod sifatida saqlanadi. Ko‘rinmas tugunlar esa grafning boshqa barcha tugunlari bo‘lib, ular "ko‘rinmas" deb ataladi, chunki ularga hali algoritm ta’sir qilmagan bo‘ladi.) Bunday belgilashlar yordamida, joriy $T =$
$\langle V_T, E_T \rangle$ daraxtga qo‘shiladigan keyingi tugunni topish $V - V_T$ to‘plamidagi eng kichik masofa belgisiga ega bo‘lgan tugunni aniqlashning oddiy masalasiga aylanadi. Daraxtdagi bog‘lovchi yoylarni ixtiyoriy ravishda o‘zgartirish mumkin. Bu algoritmning psevdokodi quyida berilgan.

**ALGORITM** *Prim(G)*

// Minimal qamrov daraxtni qurish uchun Prim algoritmi

[10] Eng yaqin tugun deganda boshlang‘ich daraxtda bo‘lmagan tugunni nazarda tutamiz, u daraxtdagi tugun bilan eng kichik vazndagi yoy orqali bog‘langan bo‘lishi kerak. Bog‘lanishlarni ixtiyoriy ravishda almashtirish mumkin.



// Kirish: Vaznli bog‘langan graf $G = \langle V, E \rangle$

// Chiqish: $E_T$, $G$ ning minimal qamrov daraxtini

// tashkil etuvchi yoylar to‘plami

$V_T \leftarrow \{v_0\}$ *// daraxt tugunlari to‘plamini istalgan*

*// tugun bilan boshlash mumkin*

$E_T \leftarrow \emptyset$ for $i \leftarrow 1$ to $V - 1$ do barcha $(v, u)$ yoylar orasida shunday eng kichik vaznli $e^* = (v^*, u^*)$ yoy topilsin, $v^*$ tugun $V_T$ da, $u$ esa $V - V_T$ ga tegishli bo‘lsin.

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

***return*** $E_T$

Daraxtga qo‘shilishi lozim bo‘lgan $u^*$ tugunni aniqlaganimizdan so‘ng, quyidagi ikki amalni bajarishimiz kerak:

- tanlangan $u^*$ tugunni daraxtga qo‘shish, ya’ni: $u^*$ tugun eng qisqa masofa bo‘yicha tanlanganidan so‘ng, $u^*$ tugunni $V - V_T$ to‘plamdan (ya’ni, hali daraxtga qo‘shilmagan tugunlar to‘plamidan) $V_T$ daraxt tugunlari to‘plamiga ko‘chirish;

- bog‘langan tugunlarni yangilash, agar $V - V_T$ to‘plamida qolgan har qanday $u$ tugun $u^*$ bilan bog‘langan bo‘lsa va ushbu bog‘lovchi yoy uzunligi $u$ tugunning hozirgi vaznidan kichikroq bo‘lsa, u holda, $u$ tugunning ota-tuguni sifatida $u^*$ belgilanadi hamda $u^*$ va $u$ tugunlar orasidagi yoyning vazni yangi vazn sifatida yangilanadi.

Bu jarayon har bir qadamda eng kichik yoyga ega bo‘lgan tugunni daraxtga qo‘shib borish orqali minimal qamrov daraxtini hosil qilishga yordam beradi. 3.3rasmda graf uchun Prim algoritmini qo‘llash ko‘rsatilgan.

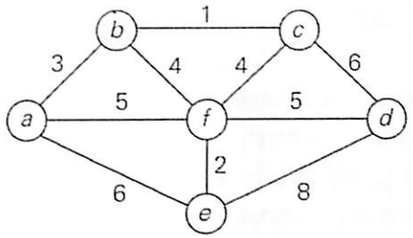
Prim algoritmi doimo minimal qamrov daraxtini hosil qiladimi? Bu savolning javobi ha. Keling, induksiya usuli bilan Prim algoritmi yordamida hosil qilingan har bir $T_i$ $(i = 0, \ldots, n-1)$ qism daraxtlar, qandaydir minimal qamrovli daraxtning bir qismi (ya’ni qism grafi) ekanligini isbotlaylik. (Bu, albatta, ketma-ketlikning oxirgi daraxti $T_{n-1}$ ning o‘zi minimal qamrovli daraxt ekanligini anglatadi, chunki u grafning barcha n ta tugunini o‘z ichiga oladi.) Induksiyaning asosi

oddiy, chunki $T_0$ bitta tugundan iborat daraxt va shu sababli har qanday minimal qamrovli daraxtning bir qismi bo'lishi shart. Induktiv qadam uchun $T_{i-1}$ qandaydir minimal qamrovli $T$ daraxtning bir qismi deb faraz qilaylik.

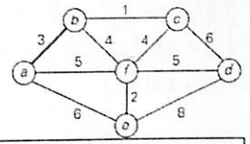
Biz Prim algoritmi bilan $T_{i-1}$ dan hosil qilingan $T_i$ ham minimal qamrovli daraxtning bir qismi ekanligini isbotlashimiz kerak. Biz buni qarama-qarshilik orqali isbotlaymiz, bunda grafning hech qanday minimal qamrov daraxti $T_i$ ni qamrab ololmaydi deb faraz qilamiz. Aytaylik, $e_i = (v, u)$ yoy $T_{i-1}$ ni $T_i$ ga kengaytirish uchun Prim algoritmi tomonidan qo'llaniladigan $T_{i-1}$ dagi tugundan $T_{i-1}$ dan tashqari tugungacha bo'lgan minimal vaznli yoy bo'lsin. Bizning farazimizga ko'ra, $e_i$ hech qanday minimal daraxtga, shu jumladan $T$ ga tegishli bo'la olmaydi. Shuning uchun, agar biz $T$ ga $e_i$ ni qo'shsak, sikl hosil bo'lishi kerak (3.4-rasm).

Bu grafda $e_i = (v, u)$ yoydan tashqari yana bir $(v', u')$ yoy bo'lishi kerak, bu yoy $v' \in T_{i-1}$ tugunni $T_{i-1}$ da bo'lmagan $u'$ tugun bilan bog'laydi. (aslida $v'$ ning $v$ bilan yoki $u'$ ning $u$ bilan ustma-ust tushishi mumkin, lekin ularning ikkalasi ham ustma-ust tushmaydi.) Agar endi bu sikldan $(v', u')$ yoyni olib tashlasak, butun grafning vazniligi $T$ ning vaznidan kichik yoki unga teng bo'lgan boshqa oraliq qamrov daraxtini hosil qilamiz, chunki $e_i$ ning vazni $(v', u')$ ning vaznidan kichik yoki unga teng bo'ladi. Demak, bu qamrov daraxti minimal qamrov daraxti bo'lib, bu hech qanday minimal qamrov daraxti $T_i$ ni o'z ichiga olmaydi degan taxminga zid keladi. Shu bilan Prim algoritmining to'g'riligini isbotlash tugallanadi.

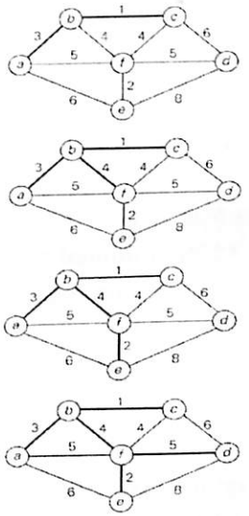





a(-,-) b(a, 3) c(-,∞) d(-,∞) e(a, 6) f(a, 5)



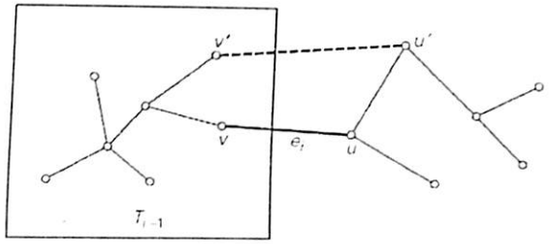
| **Daraxt tugunlari (VT)** | **Qolgan tugunlar (V-VT)** | **Tasviri (G=⟨V, E⟩)** |
|---|---|---|
| b(a, 3) | c(b, 1) d(-,∞) e(a, 6) f(b, 4) | |
| c(b, 1) | d(c, 6) e(a, 6) f(b, 4) | |
| f(b, 4) | d(f, 5) e(f, 2) | |
| e(f, 2) | d(f, 5) d(f, 5) | |



**3.3-rasm.** *Prim algoritmining qo'llanilishi. O'rtadagi ustunda tugunning qavsli yozuvlari daraxtning eng yaqin tuguni va yoyining*

*vaznini ko'rsatadi, tanlangan tugunlar va yoylar qalin chiziq bilan belgilangan.*

Prim algoritmi qanchalik samarali? Javob grafning o'zi va $V - V_T$ to'plamining ustuvor navbati uchun tanlangan ma'lumotlar tuzilmalariga bog'liq. Bu yerda $V - V_T$ to'plamidagi tugunlarning ustuvorligi daraxtning eng yaqin tugunlarigacha bo'lgan masofalar bilan belgilanadi. ($V - V_T$ to'plami haqiqatan ham ustuvorlik navbati sifatida ishlashini ko'rish uchun 3.3-rasmdagi misolni yana bir bor ko'rib chiqish zarur.)




**3.4-rasm**. *Prim algoritmining to'g'riligini isbotlash.*

Xususan, agar graf vazn matritsasi bilan ifodalansa va ustuvorlik navbati tartiblanmagan massiv ko'rinishida amalga oshirilsa, algoritmning ishlash vaqti
($V^2$) bo'ladi. Darhaqiqat, $|V| - 1$ takrorlanishning har birida ustuvorlik navbatini amalga oshiruvchi massiv eng kichik qiymatni topish va o'chirish, keyin esa, zarur bo'lsa, qolgan tugunlarning ustuvorliklarini yangilash uchun ko'rib chiqiladi.

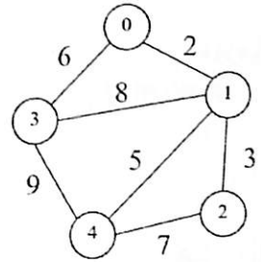
Biz ustuvor navbatni minimal uyum (min-heap) sifatida ham amalga oshirishimiz mumkin. Ya'ni, minimal uyum shunday to'liq ikkilik daraxtki, unda har bir element qiymati o'zining davomchi tugunidagi element qiymatidan kichik yoki teng bo'ladi. Uyumlarning barcha asosiy xususiyatlari minimal uyumlar uchun ham, ba'zi aniq o'zgarishlar bilan, o'z kuchini saqlab qoladi. Masalan, minimal uyumning ildizi eng katta emas, balki eng kichik elementni o'z ichiga oladi. Hajmi $n$ bo'lgan minimal uyumdan eng kichik elementni o'chirish va unga yangi elementni kiritish $O(\log n)$ amallarni bajarish orqali hosil bo'ladi, shuningdek, element ustuvorligini o'zgartirish ham xuddi shunday (ushbu bo'limdagi mashqlardan 15masalaga qarang).



Graf qo'shnilik ro'yxatlari bilan ifodalanganda va ustuvorlik navbati minheap sifatida amalga oshirilganda, algoritmning ishlash vaqti $O(|E| \log |V|)$ ga teng bo'ladi. Buning sababi algoritm eng kichik element $|V| - 1$ ni o'chirish uchun $|E|$ marta taqqoslashni amalga oshiradi $|V|$ dan oshmaydigan o'lchamdagi min-heapda elementlarning ustuvorligini o'zgartiradi. Yuqorida ta'kidlanganidek, bu amallarning har biri $O(\log |V|)$ vaqtni oladi. Shu sababli, Prim algoritmining bajarilishi $(|V| - 1 + |E|)O(\log|V|) = O(|E| \log |V|)$ vaqtni talab qiladi, chunki bog'langan grafda $|V| - 1 \leq |E|$ bo'ladi.

Keyingi bo'limda minimal qamrov daraxtini qurish masalasi uchun yana bir ochko'z algoritmni ko'rib chiqamiz, u Prim algoritmidan tubdan farq qiladi.

**Algoritmning tahlili uchun namunaviy misol:** Quyida berilgan 5 ta tugundan iborat vaznli yo'naltirilmagan graf uchun **Prim algoritmi** asosida **minimal qamrov daraxti (MST)** ni quring:




**Prim algoritmi – C++ kodi:**

```
#include <iostream>
#include
<vector>
#include
<limits.h> using
namespace std;
const int V = 5; // Tugunlar
soni // Eng yaqin tugunni
topish
int minKey(vector<int>& key, vector<bool>&
mstSet) {    int min = INT_MAX, min_index;
```

```
for (int v = 0; v < V; v++) {        if (!mstSet[v]
&& key[v] < min) {
        min = key[v], min_index = v;
      }
    }
    return min_index;
}
// Prim algoritmi
void primMST(int graph[V][V]) {
   vector<int> parent(V);    // MST uchun daraxt
vector<int> key(V, INT_MAX); // Tugunlar uchun kalit
qiymatlar    vector<bool> mstSet(V, false); // MSTga
kiritilgan tugunlar    key[0] = 0;      // Boshlang'ich
tugunni nol deb olamiz    parent[0] = -1;  // Boshlang'ich
tugunning ota elementi yo‘q    for (int count = 0; count <
V - 1; count++) {
     int u = minKey(key, mstSet); // Key eng kichik bo'lgan
tugunni topamiz        mstSet[u] = true;
     //     Qo‘shni      tugunlarni
yangilash        for (int v = 0; v <
V; v++) {
       if (graph[u][v] && !mstSet[v] && graph[u][v] <
key[v]) {              parent[v] = u;              key[v] =
graph[u][v];
       }
     }
   }
   // Natijani chiqarish
   cout << "Minimal qamrov daraxtining
yoylari:\n";    int totalWeight = 0;    for (int
i = 1; i < V; i++) {
     cout << parent[i] << " - " << i << "     og‘irlik: " <<
graph[i][parent[i]] <<
```



```
"\n";                    totalWeight +=
    graph[i][parent[i]];
    }
    cout << "Jami og‘irlik: " << totalWeight << "\n";
}      int
main() {
  // Og‘irliklar matritsasi (yoq
bo‘lsa 0)    int graph[V][V] = {
      {0, 2, 0, 6, 0},
      {2, 0, 3, 8, 5},
      {0, 3, 0, 0, 7},
      {6, 8, 0, 0, 9},
      {0, 5, 7, 9, 0}
  };
  primMST(graph);
return 0;
}
```

**Misolga izoh:** • graph[V][V] — tugunlar orasidagi og‘irlik matritsasi.

- parent[] — har bir tugunning MST dagi "ota" tugunini saqlaydi.
- key[] — har bir tugunga eng kam og‘irlikli yo‘l qiymatlari. • mstSet[] — tugun MSTga kiritildimi yoki yo‘qmi, shuni belgilaydi.

**Natija (Chiqarish):**

Dastur quyidagi ko‘rinishda minimal qamrov daraxtining yoylarini va ularning og‘irligini chiqaradi:

Minimal qamrov daraxtining yoylari:
0 - 1 og‘irlik: 2
1 - 2 og‘irlik: 3
0 - 3 og‘irlik: 6

1 - 4 og‘irlik: 5
Jami og‘irlik:
16

Algoritmning tahlili – demak, berilgan grafning vazn matritsasi:

$$graph = \begin{matrix} 0 & 2 & 0 & 6 & 0 \\ 2 & 0 & 3 & 8 & 5 \\ 0 & 3 & 0 & 0 & 7 \\ 6 & 8 & 0 & 0 & 9 \\ [0 & 5 & 7 & 9 & 0] \end{matrix}$$

Algoritm boshlang'ich tugun sifatida 0 ni tanlaydi. Matritsaga mos ravishda algoritm quyidagicha daraxt hosil qiladi: 0-1 (vazn=2), 1-2 (vazn=3), 0-3 (vazn=6), 1-4 (vazn=5). Demak, Minimal qamrov daraxtining vazni= **2+3+6+5=16** bo‘ladi.

**Vaqt murakkabligi -** Prim algoritmi oddiy massivlar yordamida (yuqoridagi kabi) amalga oshirilganda, har safar $V$ ta tugundan minKey() funksiyasi eng kichik qiymatni tanlash uchun $O(V)$ amalni bajaradi. Bu funksiya $V$ marta ishlaydi, shuning uchun umumiy holda $O(V^2)$ bo‘ladi. Berilgan misol uchun $V = 5$, demak: $O(5^2) = 25$ ta amallar bajariladi.

**Xotira murakkabligi - d**astur quyidagi elementlarni saqlaydi. Matritsa uchun
$O(V^2)$ va qolgan elementlar uchun $O(V)$

**Xulosa:**

| **O'lchov** | **Qiymat** |
|---|---|
| Tugunlar soni (V) | 5 |
| Yoylar soni (E) | 7 |
| MST vazni | 16 |
| Vaqt murakkabligi | $O(V^2) = 25$ |
| Xotira murakkabligi | $O(V^2) = 25$ |



| | |
|---|---|
| Afzalliklari | Oddiy yozilishi, tushunarli |
| Kamchiliklari | Katta graflar uchun sekin |

### 3.4. Kruskal algoritmi

Oldingi bo‘limda minimal qamrov daraxtini daraxtdagi mavjud tugunga eng yaqin tugunni ochko‘zlarcha qo‘shish orqali "kengaytiradigan" ochko‘z algoritmni ko‘rib chiqdik. Ajablanarli tomoni shundaki, minimal qamrov daraxt masalasi uchun doimo optimal yechim beradigan yana bir ochko‘z algoritm mavjud.

Kruskal algoritmi $G = \langle V, E \rangle$ bog‘langan grafning minimal qamrov daraxtini $|V| - 1$ ta yoydan iborat asiklik qism-graf sifatida tashkil qiladi, bunda yoylar vaznlarining yig‘indisi eng kichik bo‘ladi. (Bunday qism-grafning daraxt ekanligini isbotlash qiyin emas.) Shu sababli, algoritm minimal qamrov daraxtini har doim asiklik bo‘lgan, ammo algoritmning oraliq bosqichlarida bog‘langan bo‘lishi shart bo‘lmagan kengayuvchi qism-graflar ketma-ketligi sifatida quradi.

Algoritm grafning yoylarini ularning vaznlari o‘sishi tartibida saralashdan boshlanadi. So‘ngra, bo‘sh qism-grafdan boshlab, u ushbu saralangan ro‘yxatni ko‘rib chiqadi va agar bunday qo‘shish sikl hosil qilmasa, ro‘yxatning navbatdagi yoyini joriy qism-grafga qo‘shadi, aks holda yoyni shunchaki o‘tkazib yuboradi. **ALGORITHM** $Kruskal(G)$

// Minimal qamrov daraxtini qurish uchun Kruskal algoritmi
// Kirish: Vaznli bog‘langan grafik $G = \langle V, E \rangle$
// Chiqish: $E_T$, $E$ yoylar vaznlarining kamaymaydigan tartibidagi
// $G$ turidagi minimal qamrov daraxtni tashkil etuvchi yoylar
// to‘plami $w_{ei}1 \leq \ldots \leq w_{ei}|E|$ ;
$E_T \leftarrow \emptyset$; $ecounter \leftarrow 0$ //daraxt yoylari to‘plami va uning dastlabki o‘lchami
$k \leftarrow 0$ //ko‘rib chiqilgan yoylarlar soni **while** $ecounter < V - 1$ **do**

$k \leftarrow k + 1$

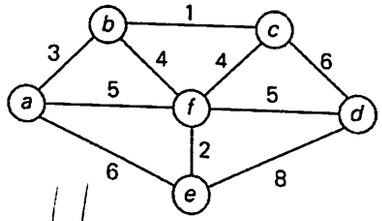
**if** $E_T \cup \{e_{ik}\}$ is acyclic

$E_T \leftarrow E_T \cup \{e_{ik}\}$;ecounter←ecounter+1

**return** $E_T$

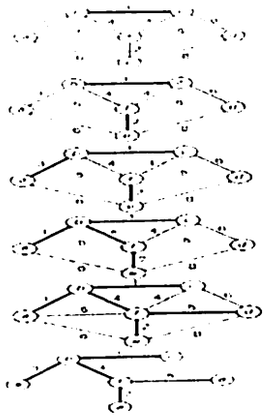
Kruskal algoritmining to'g'riligini oldingi bo'limda keltirilgan Prim algoritmini isbotlashning muhim bosqichlarini takrorlash orqali isbotlash mumkin. Prim algoritmida $E_T$ daraxt deb qaraladi, Kruskal algoritmida esa bu asiklik qism graf deb olinishi bilan farq qiladi.

3.5-rasmda 3.2-bo'limdagi Prim algoritmini tasvirlash uchun foydalangan grafga Kruskal algoritmining qo'llanilishi ko'rsatilgan. Algoritm amallari ketmaketligini kuzatib, ba'zi oraliq qismgraflarda bir-biriga bog'lanmagan qismlar mavjudligiga e'tibor bering.

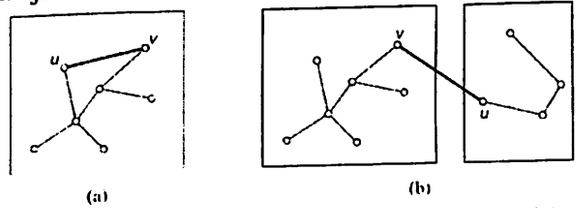



| **Daraxt yoylari** (G=⟨V, E⟩) | **Qirralarning saralangan ro'yxati** | **Tasviri** |
|---|---|---|
| | **bc** ef ab bf cf af df ae cd de | |
| | **1** 2 3 4 4 5 5 6 6 8 | |
| bc | bc **ef** ab bf cf af df ae cd de 1 | 1 **2** 3 4 4 5 5 6 6 8 |
| ef | bc ef **ab** bf cf af df ae cd de 2 | 1 2 **3** 4 4 5 5 6 6 8 |
| ab | bc ef ab **bf** cf af df ae cd de 3 | 1 2 3 **4** 4 5 5 6 6 8 |
| bf | bc ef ab bf cf af **df** ae cd de 4 | 1 2 3 4 4 5 **5** 6 6 8 |
| df | | |
| 5 | | |





**3.5-rasm.** *Kruskal algoritmining qo'llanilishi. Tanlangan yoylar qalin chiziqlar bilan ko'rsatilgan.* Prim va Kruskal algoritmlarini bir xil kichik grafga amaliy qo'llash, go'yo Kruskal algoritmi Prim algoritmidan soddaroqdek taassurot uyg'otishi mumkin. Biroq, bu taassurot noto'g'ri, chunki Kruskal algoritmining har bir takrorlanishida, tanlangan yoylarga navbatdagi yoyni qo'shish sikl hosil qilish-qilmasligini tekshirish zarur. Yangi sikl faqat va faqat yangi yoy oldingi yo'l bilan bog'langan ikkita tugunni bog'lasa, ya'ni ikkala tugun bir xil bog'langan komponentga tegishli bo'lsagina hosil bo'lishini ko'rish qiyin emas (3.6-rasm). Shuningdek, Kruskal algoritmi yordamida hosil qilingan qismgrafning har bir bog'langan komponenti daraxt hisoblanishini ta'kidlash lozim, chunki unda sikllar mavjud emas.




**3.6-rasm.** *Ikki uchni birlashtiruvchi yangi qirra (a) sikl hosil qilishi yoki (b) sikl hosil qilmasligi mumkin.*

Ushbu misollarni inobatga olgan holda, Kruskal algoritmining biroz boshqacha talqinidan foydalanish maqsadga muvofiq. Algoritm jarayonini berilgan grafning barcha tugunlari va ayrim yoylarini o'z ichiga olgan "o'rmon"lar ketmaketligi sifatida ko'rib chiqish mumkin. Boshlang'ich o'rmon $|V|$ ta oddiy daraxtdan iborat bo'lib, ularning har biri grafning bitta tugunidan tashkil topgan. Yakuniy o'rmon esa yagona daraxtdan iborat bo'lib, u grafning minimal qamrov daraxti hisoblanadi. Har bir takrorlanishda algoritm graf yoylarining tartiblangan ro'yxatidan navbatdagi $(u, v)$ yoyni oladi, $u$ va $v$ tugunlarini o'z ichiga olgan daraxtlarni aniqlaydi va agar bu daraxtlar bir xil bo'lmasa, $(u, v)$ yoyni qo'shish orqali ularni kattaroq daraxtga birlashtirib yuboradi.

Yoylarni qo‘shish uchun samarali algoritmlar ishlab chiqilgan, jumladan ikkita tugunning bir xil daraxtga tegishli yoki tegishli emasligini tekshirish uchun muhim bo‘lgan algoritmlar ham bor. Bular "unionfind" algoritmlari deb ataladi. Ular haqida keyingi bo‘limda batafsil to‘xtalamiz. Samarali "union-find" algoritmi yordamida Kruskal algoritmining ishlash vaqti asosan berilgan grafning yoy vaznlarini saralash uchun ketadigan vaqt bilan belgilanadi. Shu sababli, samarali saralash algoritmidan foydalanilganda, Kruskal algoritmining vaqt samaradorligi
$O(E \log E)$ ko‘rinishida ifodalanadi.

### 3.5. Deykstra algoritmi

Ushbu bo‘limda biz bir manbali eng qisqa yo‘llar masalasini ko‘rib chiqamiz: vaznli bog‘langan grafda manba deb ataladigan berilgan tugun uchun uning boshqa barcha tugunlariga eng qisqa yo‘llarni topish. Shuni ta’kidlash muhimki, bu yerda bizni manbadan boshlanib, qolgan barcha uchlarga yetib boradigan yagona eng qisqa yo‘l qiziqtirmaydi. Bu ancha murakkab masala hisoblanadi. Bir manbali eng qisqa yo‘llar masalasi har biri manbadan grafning boshqa tuguniga olib boradigan yo‘llar to‘plamini talab qiladi. Albatta, ba’zi yo‘llar umumiy tugunlarga ega bo‘lishi mumkin.

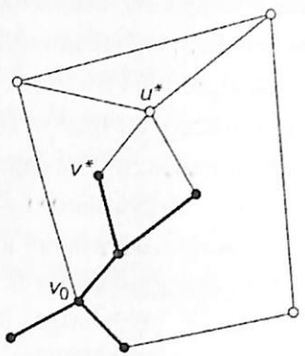
Eng qisqa yo‘llar masalasining turli xil amaliy qo‘llanilishi masalani juda mashhur tadqiqot obyektiga aylantirdi. Aniq, ammo eng keng qo‘llaniladigan holatlar, bu transport oqimini rejalashtirish va aloqa tarmoqlarini qurishda, shu jumladan Internetda paketlarni yo‘naltirish maslalaridir. Unchalik aniq bo‘lmagan holatlarga ijtimoiy tarmoqlarda eng qisqa yo‘llarni topish, nutqni tanib olish, hujjatlarni formatlash, robototexnika, kompilyatorlar va aviakompaniya ekipajini rejalashtirish kiradi. Ko‘ngilochar dunyo haqida gapirganda, video o‘yinlarda yo‘l topish va holat-fazo graflaridan foydalanib boshqotirmalarning eng yaxshi yechimlarini aniqlash kabi misollarni keltirish mumkin.



Eng qisqa yo‘llarni topish uchun bir nechta taniqli algoritmlar mavjud, shu jumladan keyingi bobda muhokama qilingan eng qisqa yo‘llar muammosi uchun Floyd algoritmi. Bu yerda bir manbali eng qisqa yo‘llar masalasi uchun eng ma’lum bo‘lgan va Deykstra algoritmi deb ataladigan algoritmni ko‘rib chiqamiz. Ushbu algoritm faqat manfiy bo‘lmagan vaznli yo‘naltirilmagan va yo‘naltirilgan graflar uchun qo‘llaniladi. Ko‘pgina ilovalarda bu shart bajarilganligi sababli, cheklov Deykstra algoritmining mashhurligiga ta’sir qilmaydi.

*Deykstra algoritmi* [11] grafning tugunlariga ularning berilgan manbadan uzoqligi tartibida eng qisqa yo‘llarni topadi. Dastlab manbadan unga eng yaqin tugungacha eng qisqa yo‘lni aniqlaydi, so‘ngra eng yaqin ikkinchi tugungacha va hokazo. Umuman olganda, uning $i$-iteratsiyasi boshlanishidan oldin, algoritm dastlab manbaga eng yaqin bo‘lgan boshqa $i-1$ tugunlarga eng qisqa yo‘llarni aniqlagan. Bu tugunlar, manba va manbadan ularga olib boruvchi eng qisqa yo‘llarning yoylari berilgan grafning $T_i$ qism daraxtini tashkil qiladi (3.7-rasm).
Barcha yoy vaznlari nomanfiy bo‘lganligi sababli, manbaga eng yaqin bo‘lgan keyingi tugunni $T_i$ ning tugunlariga qo‘shni bo‘lgan tugunlar orasidan topish mumkin. $T_i$ dagi tugunlarga qo‘shni bo‘lgan tugunlar to‘plamini "chekka tugunlar" deb atash mumkin; ular Deykstra algoritmi manbaga eng yaqin bo‘lgan keyingi tugunni tanlaydigan nomzodlardir. (Aslida, qolgan barcha tugunlarni daraxt tugunlari bilan cheksiz katta vaznlarning yoylari orqali bog‘langan tugunlar deb hisoblash mumkin.) $i$-chi eng yaqin tugunni aniqlash uchun algoritm har bir $u$ tuguni uchun daraxtning eng yaqin tuguni $v$ gacha bo‘lgan masofaning yig‘indisini $((v, u)$ bilan berilgan yoyning vaznini) va manbadan $v$ gacha bo‘lgan eng qisqa yo‘lning $d_v$ uzunligini (ilgari algoritm bilan aniqlangan) hisoblaydi va keyin bunday yig‘indi eng kichik bo‘lgan tugunni tanlaydi. Bunday maxsus yo‘llarning uzunliklarini taqqoslash kifoya ekanligi Deykstra algoritmining asosiy tushunchasidir

[11] Ushbu algoritmni 1950-yillarning o'rtalarida kompyuter fanlari va sanoatining taniqli golland olimi Edsger V. Dijkstra (1930–2002) kashf etgan. Dijkstra o'zining algoritmi haqida shunday dedi: "Bu men o'zim qo'ygan va hal qilgan birinchi graf masalasi edi. Ajablanarlisi shundaki, men uni nashr qilmaganman. O'sha paytda bu hayratlanarli emas edi. O'sha paytda algoritmlar deyarli ilmiy mavzu hisoblanmasdi".




**3.7-rasm**. *Deykstra algoritmining asosiy g'oyasi. Oldindan aniqlangan eng qisqa yo'llarning qism daraxti qalin chiziqlar bilan ko'rsatilgan. Manba $v_0$ tugunga eng yaqin bo'lgan keyingi $u^*$ tugun, qism daraxt yo'llarining uzunligiga qism daraxt tugunlariga qo'shni bo'lgan tugunlargacha bo'lgan masofalarni qo'shish orqali tanlanadi.*

Algoritmning ishlashini osonlashtirish maqsadida har bir tugunni ikkita belgi bilan belgilaymiz. Sonli belgi $d$ algoritm shu paytgacha topgan manbadan ushbu tugungacha bo'lgan eng qisqa yo'lning uzunligini ko'rsatadi; yangi tugun daraxtga qo'shilganda esa, $d$ manbadan shu qo'shilgan tugungacha bo'lgan eng qisqa yo'lning uzunligini ifodalaydi. Boshqa belgi esa bunday yo'ldagi oxirgi tugundan bitta oldingi tugunning nomini, ya'ni qurilayotgan daraxtdagi tugunning "ota" tugunini ko'rsatadi. (Manba $s$ va joriy daraxt tugunlarining hech biriga qo'shni bo'lmagan tugunlar uchun buni ko'rsatmasa ham bo'ladi.) Bunday belgilash tizimi bilan navbatdagi eng yaqin $u^*$ tugunni topish eng kichik $d$ qiymatga ega bo'lgan chetki tugunni topish kabi oddiy vazifaga aylanadi. Teng qiymatli holatlarni ixtiyoriy tarzda hal qilish mumkin, ya'ni manbadan yangi tugungacha qisqa yo'l aniqlanganda, ushbu tugungacha bo'lgan odingi bog'lanish yo'lini almashtirish mumkin.

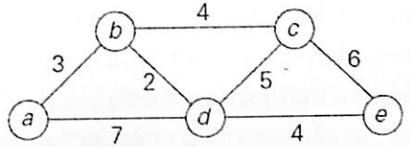
Daraxtga qo'shilishi kerak bo'lgan $u^*$ tugunni aniqlaganimizdan so'ng ikkita amalni bajarishimiz kerak:

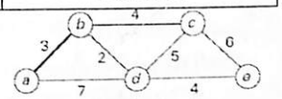
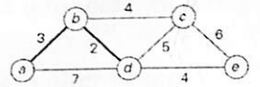
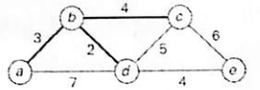
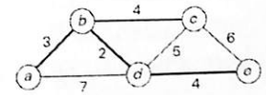
- $u^*$ ni chet tugunlar to'plamidan daraxt uchlari to'plamiga ko'chirish.



- $u^*$ bilan $w(u^*,u)$ vazndagi yoy orqali bog'langan va $d_{u^*} + w(u^*, u) < du$ shartni qanoatlantiruvchi har bir qolgan $u$ tugun uchun $u$ ning belgilarini mos ravishda $u^*$ va $d_{u^*} + w(u^*, u)$ ga o'zgartirish.

3.8-rasmda muayyan graf uchun Deykstra algoritmini qo'llash ko'rsatilgan. Deykstra algoritmining belgilashlari va mexanizmi Prim algoritmida qo'llanilgan belgilash va mexanizmga juda o'xshash. Ularning ikkalasi ham qolgan tugunlarning ustuvorlik ro'yxatidan navbatdagi tugunni tanlab, tugunlarning kengayuvchi minimal daraxtini quradi. Biroq, ularni aralashtirib yubormaslik muhim. Ular turli masalalarni yechadi va shuning uchun boshqa usulda hisoblangan ustuvorliklar bilan ishlaydi: Deykstra algoritmi yo'l uzunliklarini taqqoslaydi va shuning uchun yoy vaznlarini qo'shishi kerak, Prim algoritmi esa berilgan yoy vaznlarini taqqoslaydi.




| Daraxt tugunlari | Qolgan (chetki) tugunlar | Tasviri |
|---|---|---|
| a(-, 0) | b(a, 3) c(-,∞) d(a, 7) e(-,∞) |  |
| b(a, 3) | c(b, 3+4) d(b, 3+2) e(-,∞) |  |
| d(b, 5) | c(b, 7) e(d, 5 + 4) |  |
| c(b, 7) | e(d, 9) |  |
| e(d, 9) | | |

Eng qisqa yo'llar (chap ustundagi manzil tugunidan manba tugunigacha raqamsiz belgilarni teskari tartibda kuzatish orqali

aniqlanadi) va ularning uzunliklari (daraxt tugunlarining raqamli belgilari) bilan quyidagicha ifodalanadi:

---

$a$ dan $b$ gacha: $a - b$ uzunligi 3
$a$ dan $d$ gacha: $a - b - d$ uzunligi 5
$a$ dan $c$ gacha: $a - b - c$ uzunligi 7
$a$ dan $e$ gacha: $a - b - d - e$ uzunligi 9

---

**3.8-rasm**. *Deykstra algoritmining qo'llanilishi. Keyingi eng yaqin tugunga yo'l qalin chiziq bilan ko'rsatilgan.*

Endi biz Deykstra algoritmining psevdokodini taqdim etishimiz mumkin. U

Prim algoritmiga nisbatan batafsil tarzda, ya'ni belgilangan tugunlarning ikkita to'plami ustidagi aniq amallar nuqtai nazaridan bayon etilgan. Bular dastlab aniqlangan eng qisqa yo'l dagi $V_T$ tugunlar to'plami va chetki tugunlarning $Q$ ustuvorlik navbati. (E'tibor bering, quyidagi psevdokodda $V_T$ berilgan manba tugunini o'z ichiga oladi va yo'l 0-iteratsiya yakunlangandan so'ng unga qo'shni tugunlarni qamrab oladi.)

**ALGORITHM** $Dijkstra(G, s)$
//Bir manbali eng qisqa yo'llar uchun Dijkstra algoritmi
//Kirish: manfiy bo'lmagan vaznli bog'langan $G = \langle V, E \rangle$
// graf va uning tuguni $s$
//Chiqish: $V$ dagi har bir $v$ tugun uchun $s$ dan $v$ gacha bo'lgan eng
// qisqa yo'lning $d_v$ uzunligi va uning oxiridan oldingi $p_v$ tuguni
$Initialize(Q)$ //bo'sh ustivor navbatni
aniqlash **for** $V$ dagi har bir $v$ tugun
$d_v \leftarrow \infty; p_v \leftarrow null$
$Insert(Q, v, d_v)$ //ustuvor navbatdagi tugun ustuvorligini aniqlash $d_s \leftarrow 0$; $Decrease(Q, s, d_s)$ //$s$ ning ustuvorligini $d_s$ bilan yangilash



$V_T \leftarrow \emptyset$ **for** $i \leftarrow 0$ **to** $|V| - 1$ **do**
$u^* \leftarrow DeleteMin(Q)$ // eng kichik ustuvorlik elementini o'chirish
$V_T \leftarrow V_T \cup \{u^*\}$
**for** $u^*$ ga qo'shni bo'lgan $V \_ V_T$ dagi har bir u tugun **do** **if** $d_{u^*} + w(u^*, u) < d_u$ $d_u \leftarrow d_{u^*} + w(u^*, u)$; $p_u \leftarrow u^*$
$Decrease(Q, u, d_u)$

Deykstra algoritmining vaqt bo'yicha samaradorligi ustuvorlik navbatini amalga oshirish va kirish grafining o'zini ifodalash uchun ishlatiladigan ma'lumotlar tuzilmalariga bog'liq. Prim algoritmini tahlil qilishda tushuntirilgan sabablarga ko'ra, samaradorlik vazn matritsasi va tartiblanmagan massiv sifatida amalga oshirilgan ustuvorlik navbati bilan ifodalangan graflar uchun $\Theta(|V|^2)$ ga teng.

Qo'shnilik ro'yxati va min-uyum ko'rinishida amalga oshirilgan ustuvorlik navbati bilan ifodalangan graflar uchun esa $O(E \log |V|)$ bo'ladi. Agar ustuvorlik navbati Fibonachchi uyumi deb ataladigan murakkab ma'lumotlar tuzilmasi yordamida amalga oshirilsa, Prim va Deykstra algoritmlari uchun yanada yaxshiroq yuqori chegaraga erishish mumkin (masalan, [Cor09]). Biroq, uning murakkabligi va katta qo'shimcha xarajatlar bunday takomillashtirishni asosan nazariy jihatdan asosli ekanligini ko'rsatadi.

### 3.6. Ochko'z algoritm– afzalliklari va kamchiliklari

Ochko'z algoritm (Greedy algorithm) – bu har bir bosqichda eng yaxshi (lokal optimal) qarorni tanlab, umumiy muammoni yechishga urunadigan algoritmdir. Bu algoritmlar har doim joriy holatda eng foydali variantni tanlaydi, lekin har doim global eng yaxshi yechimga olib kelmasligi mumkin.

Algoritmning **afzalliklari:** *soddaligi* - oddiy ishlaydi, tushunish va dasturlash oson, *kam vaqt* talab qiladi - ko'pincha vaqt murakkabligi $O(n \log n)$ yoki $O(n)$ atrofida bo'ladi. Masalan, Kruskal yoki Prim

algoritmlari. *Kam xotira* talab qiladi dinamik dasturlash yoki teskari kuzatishga (backtracking) nisbatan kamroq xotira sarflaydi. Amaliy muammolarda *tez yechim beradi* - har doim eng optimal bo'lmasa ham, "yaxshi yechim" tez topiladi. *Parallellashtirishga mos* - mustaqil tanlovlar asosida qarorlar qabul qilinadi, bu uni ba'zan paralel ishlov berishga yaroqli qiladi.

**Kamchiliklari:** doim ham *global optimal yechim* bermaydi - har doim lokal eng yaxshi tanlov qilinsa ham, umumiy natija eng yaxshi bo'lmasligi mumkin. Masalan, "Yuk xaltasi" (Sumka - Knapsack) masalasi.

Ba'zi muammolarda *ishlamaydi* - ayrim kombinatorik masalalarda ochko'z yondashuv yaroqsiz bo'ladi. Oldindan *isbot talab qiladi* - dastur yozishdan oldin, ochko'z yondashuv to'g'ri yechim berishini matematik jihatdan isbotlash kerak. Ba'zan mukammal yechim kerak bo'lgan holatlarda *noto'g'ri tanlov qiladi* - har bir qaror keyingi bosqichlarga ta'sir qiladi, bu esa xatolikka olib keladi.

Qachon foydalanish kerak? Muammoning **ochko'z xossasi** (greedy choice property) va **optimal qismlarga ajratish mumkin** bo'lsa. Tez va taxminiy (yaqin) yechim kifoya qiladigan hollarda. Resurslar (vaqt, xotira) cheklangan bo'lsa.

Mashhur ochko'z algoritmlarga misollar: Kruskal algoritmi va Prim algoritmi (Minimum qamrov daraxti), Deykstra algoritmi (Eng qisqa yo'l), Intervalni rejalashtirish (Activity selection), Tangalarni almashtirish (Coin change, ba'zi holatlarda).

### 3.7. Mustaqil ishlash uchun savollar va mashqlar

**1. Tangalarni almashtirish** masalasi uchun ochko'z algoritmning psevdokodini yozing. Kirish ma'lumotlari sifatida $n$ miqdor va $d1 > d2 > \ldots > dm$ tanga nominallari berilsin. Algoritmingizning vaqt samaradorligi sinfini aniqlang.

**2. Vazifalarni taqsimlash** masalasi uchun ochko'z algoritmini ishlab chiqing

(3.4-bo'limga qarang). Siz taklif qilgan ochko'z algoritm har doim eng maqbul yechimni beradimi?



**3. Vazifalarni rejalashtirish** (Job scheduling) masalasi. Bitta protsessor tomonidan bajarilishi uchun davomiyligi ma'lum bo'lgan $t1, t2, \ldots, tn$ vaqtga ega

$n$ ta vazifani rejalashtirish masalasini ko'rib chiqaylik. Vazifalarni istalgan tartibda va bir vaqtning o'zida faqat bitta vazifani bajarish mumkin. Tizimdagi barcha vazifalarning umumiy vaqtini minimallashtiruvchi jadval tuzilsin. (Tizimdagi bitta ish uchun sarflangan vaqt ushbu ishning kutish vaqti va uni bajarish vaqtining yig'indisiga teng.)

Ushbu masala uchun ochko'z algoritm ishlab chiqing. Ochko'z algoritm har doim ham optimal yechimni beradimi?

**4. Mos intervallar.** Haqiqiy sonlar o'qida $n$ ta ochiq interval $(a1,b1), (a2,b2), \ldots, (an, bn)$ berilgan bo'lib, ularning har biri bir xil resursni talab qiladigan ma'lum faoliyatning boshlanish va tugash vaqtlarini ifodalaydi. Vazifa shundan iboratki, bu intervallarning eng ko'p sonini aniqlab olish kerak, bunda tanlangan intervallarning hech biri bir-biri bilan kesishmasligi lozim. Quyidagi uchta ochko'z algoritmni ishlab chiqing:

a. eng tez boshlanish vaqti bo'yicha.

b. eng qisqa davomiyligi bo'yicha.

c. eng tez tugash vaqti bo'yicha.

Ushbu uch algoritmning har biri uchun, algoritm doimo optimal yechimni berishini isbotlang yoki bu shunday emasligini ko'rsatuvchi misol keltiring.

**5. Ko'prikdan o'tish masalasi.** Ko'prikdan o'tish masalasining kengaytirilgan variantini (1.2-mashqdagi 2-masala). Bunda $n > 1$ kishining ko'prikdan o'tish vaqtlari $t1, t2, \ldots, tn$ bo'lsin. Masalaning qolgan barcha shartlari o'zgarmaydi: ko'prikdan bir vaqtning o'zida faqat ikki kishi o'ta oladi (ular ikkala kishidan sekinroq yuradiganining tezligida harakatlanadi) va ular o'zlari bilan guruhda mavjud bo'lgan yagona yorituvchi chiroqni olib o'tishlari shart.

Ushbu masala uchun ozko'z algoritm ishlab chiqing va bu algoritm yordamida ko'prikdan o'tish uchun qancha vaqt ketishini aniqlang. Ishlab chiqilgan algoritm masalaning har bir holati uchun eng qisqa

o‘tish vaqtini ta’minlaydimi? Agar shunday bo‘lsa, buni isbotlang; agar unday bo‘lmasa, bunday holat yuz beradigan eng kam sonli odamlar misolini keltiring.

**6. O‘rtacha kamaytirish masalasi.** $n > 1$ ta bir xil idish mavjud, ulardan birida $W$ miqdorda suv bor, qolganlari bo‘sh. Faqat quyidagi amalni bajarishga ruxsat berilgan: ikkita idishni olib, ulardagi suvning umumiy miqdorini teng ikkiga bo‘lish. Maqsad shunday amallar ketma-ketligi orqali dastlabki holatda barcha suv joylashgan idishdagi suv miqdorini iloji boricha kamaytib borishdir. Buni amalga oshirishning eng samarali usuli qanday?

**7. Mish-mish tarqatish.** $n$ ta odam bor, ularning har biri turli mish-mishlarga ega. Ular elektron xabarlar yuborish orqali barcha mish-mishlarni bir-biriga yetkazmoqchi. Aytaylik, jo‘natuvchi xabar yuborilgan paytda o‘zi bilgan barcha mish-mishlarni o‘z ichiga olgan xabarni faqat bitta manzilga yubora oladi.

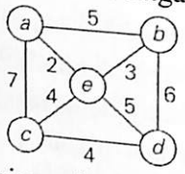
Ularning har biri barcha mish-mishlarni qabul qilishini kafolatlash uchun har doim yuborilishi kerak bo‘lgan xabarlarning minimal sonini beradigan ochko‘z algoritmni ishlab chiqing.

**8. Tarozi toshlari haqidagi masala.** 1 dan $W$ gacha bo‘lgan eng katta oraliqda har qanday butun sonli yukni tarozida tortish mumkin bo‘lishi uchun $n$ ta yukning optimal to‘plami $\{w1, w2, \ldots, wn\}$ ni toping, bunda:

**a.** toshlarni faqat tarozining bo‘sh pallasiga qo‘yish mumkin.

**b.** tarozining ikkala pallasiga ham tosh qo‘yish mumkin.

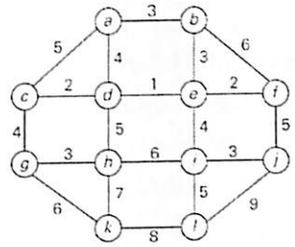
**9.** Quyidagi grafga Prim algoritmini qo‘llang. Daraxtda mavjud bo‘lmagan barcha tugunlarni ustuvor navbatga qo‘shing.




b. Quyidagi grafga Prim algoritmini qo‘llang. Ustuvorlik navbatiga faqat “chet” tugun (joriy daraxtga kirmagan, ammo



daraxtning kamida bitta tuguni bilan qo‘shni bo‘lgan tugun) larni kiriting.




**10.** Minimal qamrov daraxti tushunchasi bog‘langan vaznli grafga qo‘llaniladi. Prim algoritmini qo‘llashdan oldin grafning bog‘liqligini tekshirishimiz kerakmi yoki algoritm buni o‘zi aniqlay oladimi?

**11.** Prim algoritmi yoylarining vazni manfiy bo‘lgan graflarda har doim ham to‘g‘ri ishlaydimi?

**12.** $T$ Prim algoritmi yordamida olingan $G$ grafning minimal qamrov daraxti bo‘lsin. $G_{new}$ grafi $G$ ga yangi tugun va yangi tugunni $G$ ning ba’zi tugunlari bilan bog‘laydigan ba’zi yoylarni qo‘shish orqali hosil qilingan vaznli graf bo‘lsin. $T$ ga yangi yoylardan birini qo‘shib, $G_{new}$ ning minimal qamrov daraxtini qurish mumkinmi? Agar "ha" deb javob bersangiz, qanday bajarilishini; "yo‘q" deb javob bersangiz, nima uchun ekanligini tushuntiring.

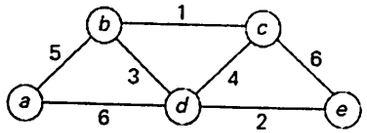
**13.** Yoylarida vazni bo‘lmagan bog‘langan grafning qamrov daraxtini topish uchun Prim algoritmidan qanday foydalanish mumkin? Bu masala uchun samarali algoritmmi?

**14.** Turli vaznlarga ega bo‘lgan har qanday vaznli bog‘langan graf faqat bitta minimal qamrov daraxtiga ega ekanligini isbotlang.

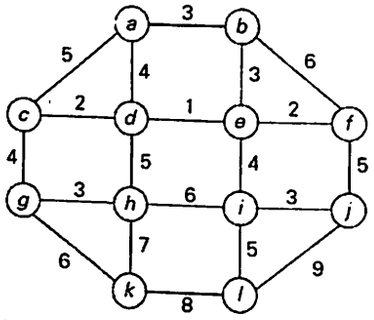
**15.** Min-heapda element qiymatini o‘zgartirishning samarali algoritmini tavsiflab bering. Algoritmning vaqt samaradorligi qanday bo‘ladi?

**16.** Quyidagi graflarning minimal qamrov daraxtini topish uchun Kruskal algoritmini qo‘llang.

a.




b.




**17.** Quyidagi mulohazalarning to‘g‘ri yoki noto‘g‘riligini ko‘rsating:

a. Agar *e* bog‘langan vaznli grafda minimal vaznli yoy bo‘lsa, u grafning
kamida bitta minimal qamrov daraxti yoylari orasida bo‘lishi kerak.

b. Agar *e* bog‘langan vaznli grafda minimal vaznli yoy bo‘lsa, u grafning har
bir minimal qamrov daraxti yoylari orasida bo‘lishi kerak.

c. Agar bog‘langan vaznli grafning barcha yoylari vaznlari turlicha bo‘lsa,
graf aniq bitta minimal qamrov daraxtiga ega bo‘lishi kerak.

d. Agar bog‘langan vaznli grafning barcha yoylari vaznlari bir-biridan farq
qilmasa, u holda graf bir nechta minimal qamrov daraxtiga ega bo‘lishi kerak.

**18.** Kruskal algoritmi ixtiyoriy graf uchun ***minimal qamrov o‘rmon***ni topishi uchun unga qanday o‘zgarishlar kiritish kerak? (Minimal qamrov o‘rmon - bu daraxtlari grafning bog‘langan komponentlarining ***minimal qamrov daraxtlari*** bo‘lgan ***o‘rmon***dir.)

**19.** Qirralarining vazni manfiy bo‘lgan graflarda Kruskal algoritmi to‘g‘ri ishlaydimi?



**20.** Vaznli bog‘langan grafning maksimal qamrovli daraxtini - mumkin bo‘lgan eng katta yoy vazniga ega bo‘lgan qamrov daraxtni topish algoritmini tuzing.

**21.** Kruskal algoritmining psevdokodini kesishmaydigan qism to‘plamlarning ADT operatsiyalari nuqtai nazaridan qayta yozing.

**22.** Kruskal algoritmining to‘g‘riligini isbotlang.

**23.** Tezkor birlashtirishning birlik o‘lchamli varianti uchun $find(x)$ amalining vaqt samaradorligi $O(\log n)$ ekanligini isbotlang.

**24.** Kruskal va Prim algoritmlari vizuallashtirilgan (animatsiya qilingan) kamida ikkita Web-sayt toping. Ularning yutuq va kamchiliklarini izohlab bering.

**25.** Turli o‘lcham va zichlikdagi tasodifiy graflarda Prim va Kruskal algoritmlarining samaradorligini empirik taqqoslash uchun tajriba o‘tkazing.

**26.** Shteyner daraxti. Evklid tekisligida birlik kvadratning uchlarida to‘rtta qishloq joylashgan. Sizdan ularni eng qisqa yo‘llar tarmog‘i bilan bog‘lash so‘raladi, shunda bu yo‘llar bo‘ylab joylashgan har bir juft qishloq o‘rtasida yo‘l mavjud bo‘ladi. Shunday tarmoqni toping.

**27.** Quyidagilar asosida tasodifiy labirint yaratuvchi dastur yozing:

a. Prim algoritmi

b. Kruskal algoritmi

**28.** Quyidagi masalalarni yechish uchun Deykstra algoritmi va/yoki asosiy grafga qanday tuzatishlar kiritish kerakligini tushuntiring.

**a.** Yo‘naltirilgan vaznli graflar uchun bir manbali eng qisqa yo‘llar masalasini
yeching.

**b.** Vazn berilgan graf yoki yo‘naltirilgan grafning ikki berilgan tuguni orasidagi eng qisqa yo‘lni toping. (Bu variant juft-tugunli eng qisqa yo‘l masalasi deb yuritiladi.)

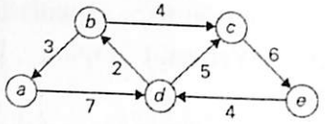
**c.** Vazn berilgan graf yoki yo‘naltirilgan grafning berilgan bir tuguniga

boshqa barcha tugunlardan eng qisqa yo‘llarni toping. (Bu variant yagona-manzilli eng qisqa yo‘llar masalasi deb ataladi.)

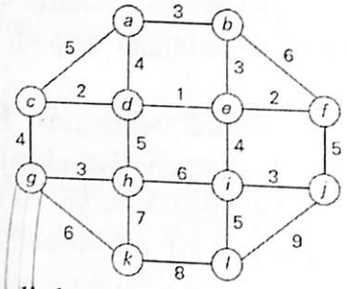
**d.** Uchlari manfiy bo‘lmagan sonlar bilan belgilangan grafda yagona-manbali
eng qisqa yo‘llar masalasini yeching (bunda yo‘l uzunligi yo‘ldagi tugunlar sonlarining yig‘indisi sifatida aniqlanadi).

**29.** Manba sifatida $a$ tugunga ega bo‘lgan bir manbali eng qisqa yo‘llar masalasining quyidagi misollarini yeching:

**a.**




**b.**




**30.** Manfiy vaznli bog‘langan graf uchun Deykstra algoritmi ishlamasligini ko‘rsatadigan qarshi misol keltiring.

**31.** Vaznli bog‘langan $G$ graf uchun bir manbali eng qisqa yo‘llar masalasini yechish jarayonida Deykstra algoritmi yordamida qurilgan $T$ daraxt berilgan bo‘lsin.

**a.** To‘g‘ri yoki noto‘g‘ri: $T$ daraxt $G$ ning qamrov daraxti hisoblanadimi?

**b.** To‘g‘ri yoki noto‘g‘ri: $T$ daraxt $G$ ning minimal qamrov daraxti hisoblanadimi?

**32.** Vazn matritsasi bilan ifodalangan grafning berilgan tugunidan qolgan barcha tugunlargacha bo‘lgan masofalarni (ya’ni eng qisqa yo‘llar uzunligini, lekin yo‘llarning o‘zini emas) topadigan Deykstra algoritmining soddalashtirilgan varianti uchun psevdokod yozing.

**33.** Musbat vaznli graflar uchun Deykstra algoritmining to‘g‘riligini isbotlang.



**34.** Qo‘shnilik ro‘yxatlari bilan ifodalangan yo‘naltirilgan asiklik graflar
(DAG) uchun bir manbali eng qisqa yo‘llar masalasini yechadigan chiziqli murakkablikdagi (vaqtli) algoritmni ishlab chiqing.

**35.** Eng qisqa yo‘lni modellashtirish: Faraz qiling, sizda tegishli uzunlikdagi (yoylarni ifodalovchi) iplar bilan bog‘langan sharlardan (tugunlarni ifodalovchi) yasalgan vaznli bog‘langan graf modeli bor.

**a.** Ushbu model yordamida bir juftlik eng qisqa yo‘l masalasini qanday
yechish mumkinligini tasvirlab bering.

**b.** Ushbu model yordamida bir manbali eng qisqa yo‘llar masalasini qanday
hal qilish mumkinligini tasvirlab bering.

# 4-BOB. DINAMIK DASTURLASH MASALALARI

## 4.1. Uorshall algoritmi

Dinamik dasturlash - bu juda qiziqarli tarixga ega algoritmlarni loyihalash texnikasi hisoblanadi. U AQShning taniqli matematigi Richard Bellman tomonidan

1950-yillarda ko‘p bosqichli qaror qabul qilish jarayonlarini optimallashtirishning umumiy usuli sifatida ixtiro qilingan. Shunday qilib, ushbu texnikaning nomidagi

"dasturlash" so‘zi "rejalashtirish" degan ma’noni anglatadi va kompyuter dasturini anglatmaydi. Amaliy matematikaning muhim vositasi sifatida o‘z qiymatini isbotlagandan so‘ng, dinamik dasturlash oxir-oqibat hech bo‘lmaganda informatika sohasida optimallashtirish masalalarining maxsus turlari bilan cheklanmaydigan algoritmlarni loyihalashning umumiy texnikasi sifatida ko‘rib chiqiladi.

Dinamik dasturlash bir-birini to’ldiruvchi qism-masalalarga ega masalalarni yechish texnikasidir. Odatda, bu masalalar berilgan masalaning yechimini uning qism masalalarining yechimlari bilan bog‘lovchi rekursiya natijasida paydo bo‘ladi. Dinamik dasturlash bir nechta qism masalalarni qayta-qayta yechish o‘rniga, qism masalalarning har birini faqat bir marta yechishni va natijalarni jadvalga yozishni taklif qiladi, shundan so‘ng dastlabki masalaning yechimini olish mumkin.

Ushbu bo‘limda biz ikkita mashhur algoritmni ko‘rib chiqamiz: yo‘naltirilgan grafning tranzitiv yopilishini[12] hisoblash uchun Uorsholl algoritmi va barcha juftliklar uchun eng qisqa yo‘llar masalasini yechish uchun Floyd algoritmi. Uorsholl va Floyd o‘z algoritmlarini e’lon qilishda dinamik dasturlash haqida so‘z yuritishmagan. Shunday bo‘lsa-da, bu algoritmlar, shubhasiz, dinamik dasturlash uslubiga ega va vaqt o‘tishi bilan ushbu texnikaning qo‘llanilishi sifatida qaraladigan bo‘ldi.

[12] Tranzitiv yopilish — bu yo‘naltirilgan graf ning barcha mumkin bo‘lgan yo‘llarini hisobga oladigan kengaytirilgan ko‘rinishidir. Matematik ta’rif: $G = (V, E)$ yo‘naltirilgan graf berilgan bo‘lsa, uning tranzitiv yopilishi $G^* = (V, E^*)$ bo‘lib, agar $(u, v) \in E^*$ bo‘lsa, u holda $u$ dan $v$ ga yo‘l mavjud, ya’ni, agar $G$ grafda $u$ dan $v$ ga bevosita yoki bir necha oraliq tugunlar orqali borish mumkin bo‘lsa, unda $(u, v) \in E^*$.



**Uorshall algoritmi.** Yo‘naltirilgan grafning $A = \{a_{ij}\}$ qo‘shnilik matritsasining $i$-satrida va $j$-ustunidagi qiymat 1 ga teng bo‘lgan mantiqiy matritsa hisoblanadi. Bunda 1 qiymati faqat va faqat grafning $i$-tugundan $j$-tuguniga yo‘naltirilgan yoy mavjud bo‘lgandagina qo’yiladi. Shuningdek, berilgan grafning tugunlari orasidagi ixtiyoriy uzunlikdagi yo‘naltirilgan yo‘llar mavjudligi haqidagi ma’lumotni o‘z ichiga olgan matritsa ham muhim ahamiyatga ega. Bu matritsa yo‘naltirilgan grafning tranzitiv yopilmasi deb ataladi. Bunday matritsa $j$-chi tugunga $i$-chi tugundan yetib borish mumkinligini doimiy vaqt (Constant Time[13], O(1)) ichida aniqlash imkonini beradi.

Quyida bir nechta amaliy misol keltirilgan. Elektron jadval katagidagi qiymat o‘zgartirilganda, jadval dasturi bu o‘zgarish ta’sir ko‘rsatgan boshqa barcha kataklar haqidagi ma’lumotni bilishi lozim. Agar elektron jadval tugunlari kataklar, yoylari esa kataklar orasidagi bog‘liqliklarni ifodalovchi yo’naltirilgan graf bilan modellashtiriladigan bo‘lsa, tranzitiv yopilish aynan shunday ma’lumotni

ta’minlaydi. Dasturiy ta’minot muhandisligida tranzitiv yopilish ma’lumotlar oqimi va boshqaruv oqimi bog‘liqliklarini tadqiq qilish, shuningdek, obyektga yo‘naltirilgan dasturiy ta’minotning vorislik xususiyatlarini sinovdan o‘tkazish uchun qo‘llaniladi. Elektron muhandislikda esa u raqamli sxemalardagi ortiqcha elementlarni aniqlash va sinov namunalarini yaratish uchun foydalaniladi.

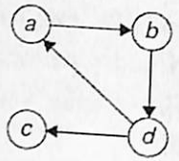
**TA’RIF:** $n$ ta tugunga ega bo‘lgan yo‘naltirilgan grafning tranzitiv yopiqligi $n \times n$ mantiqiy matritsa $T = \{t_{ij}\}$ sifatida aniqlanishi mumkin, unda $i$-tugundan $j$tugungacha musbat uzunlikdagi yo‘naltirilgan yo‘l mavjud bo‘lsa, $T$ matritsaning $i$satr va $j$-ustundagi element 1 ga teng; aks holda, 0 ga teng bo’ladi.

4.1-rasmda yo’naltirtilgan grafning qo‘shnilik va tranzitiv yopilish matritsasiga misol ko‘rsatilgan.

Yo’naltirilgan grafning tranzitiv yopilishini chuqurlik bo‘yicha qidiruv (DFS) yoki kenglik bo‘yicha qidiruv (BFS) yordamida hosil

[13] Agar tranzitiv yopilish matritsasi berilgan bo‘lsa, har qanday $(i, j)$ tugunlar juftligi orasida yo‘l bor yoki yo‘qligini oddiy matritsa elementi orqali tekshirish mumkin. Masalan, $M[i][j] = 1$ bo‘lsa, $i$ dan $j$ ga borish mumkinligini; $M[i][j] = 0$ bo‘lsa, yo‘l yo‘qligini anglatadi.

qilish mumkin. $i$-chi tugundan boshlab bu ikki usuldan birini qo'llash orqali undan yetib borish mumkin bo'lgan tugunlar haqida ma'lumot olinadi va shu asosda tranzitiv yopilish matritsasining $i$satrida 1 qiymati bo'lgan ustunlar aniqlanadi. Shunday qilib, har bir tugunni boshlang'ich nuqta sifatida olib, bu jarayonni takrorlash orqali to'liq tranzitiv yopilish hosil qilinadi.




(a)

$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

(b)

$$T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

(c)

**4.1-rasm.** *(a) yo'naltirilgan graf. (b) uning qo'shnilik matritsasi. (c) uning tranzitiv yopilish matritsasi.*

Ushbu usul bir xil yo'naltirilgan grafni bir necha marta ko'rib chiqishi sababli, yaxshiroq algoritm topilishiga umid qilishimiz kerak. Darhaqiqat, bunday algoritm mavjud. U Stiven Uorshall tomonidan kashf etilgan bo'lib, Uorshall algoritmi deb ataladi. Yo'naltirilgan grafning tugunlari va shunga ko'ra qo'shnilik matritsasining satr va ustunlari 1 dan $n$ gacha raqamlangan deb hisoblash qulay. Uorshall algoritmi tranzitiv yopilishni $n \times n$ o'lchamli mantiqiy matritsalar ketmaketligi orqali hosil qiladi:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)} \qquad (4.1)$$

Bu matritsalarning har biri grafda yo'naltirilgan yo'llar haqida ma'lum ma'lumotlarni taqdim etadi. Xususan, agar $R^{(k)}$ $(i, j = 1,2,\ldots,$ $n, k = 0,1,\ldots, n)$ matritsaning $i$-satri va $j$-ustunidagi $r_{ij}{}^{(k)}$ elementi 1 ga teng bo'lsa, demak grafning $i$-tugunidan $j$-tugunigacha musbat uzunlikdagi yo'naltirilgan yo'l mavjud va har qanday oraliq tugun (agar mavjud bo'lsa), $k$ dan katta bo'lmagan raqam bilan belgilangan bo'ladi. Shunday qilib, matritsalar qatori $R^{(0)}$ dan boshlanadi, bu o'z yo'llarida hech qanday oraliq tugunlarga yo'l qo'ymaydi; demak, $R^{(0)}$ yo'naltirilgan grafning qo'shnilik matritsasidan boshqa narsa emas. (Eslatib o'tamiz, qo'shnilik matritsasi oraliq tugunlari bo'lmagan yo'llar haqidagi ma'lumotlarni o'z ichiga oladi). $R^{(1)}$ birinchi tugunni oraliq



tugun sifatida ishlatishi mumkin bo'lgan yo'llar haqidagi ma'lumotni o'z ichiga oladi; shu sababli, aytish mumkinki, ko'proq imkoniyat bilan, u $R^{(0)}$ ga nisbatan ko'proq 1 larni o'z ichiga olishi mumkin. Umuman olganda, (4.1) matritsalar qatoridagi har bir keyingi matritsa o'zidan oldingi matritsaga qaraganda yo'llar uchun oraliq sifatida bitta ko'proq tugunga ega bo'ladi va shu sababli u ko'proq 1 larni o'z ichiga olishi mumkin, ammo bu shart emas. Qatordagi oxirgi matritsa $R^{(n)}$ grafning barcha $n$ tugunidan oraliq sifatida foydalanishi mumkin bo'lgan yo'llarni aks ettiradi va shuning uchun u grafning tranzitiv yopilishidan boshqa holatni ifodalamaydi.

Algoritmning asosiy g'oyasi shundaki, (4.1) qatordagi har bir $R^{(k)}$ matritsaning barcha elementlarini uning bevosita izdoshi bo'lgan $R^{(k-1)}$ matritsadan hisoblash mumkin. $R^{(k)}$ matritsaning $i$-satri va j-ustunidagi $r_{ij}{}^{(k)}$ element 1 ga teng bo'lsa, u holda grafning $i$-tuguni $v_i$ dan $j$-tuguni $v_j$ gacha yo'l mavjud bo'lib, unda har bir oraliq tugun $k$ dan katta bo'lmagan raqam bilan belgilangan bo'ladi, ya'ni $v_i$ dan $v_j$ gacha yo'lni quyidagicha tugunlar ro'yxati shalida ifodalash mumkin:

$v_i$, $k$ dan katta bo'lmagan raqamlangan oraliq tugunlar ro'yxati, $v_j$. **(4.2)**

Bu yo'lda ikki holat bo'lishi mumkin. Birinchisida uning oraliq tugunlari ro'yxati $k$-tugunni o'z ichiga olmaydi. U holda bu $v_i$ dan $v_j$ gacha bo'lgan yo'l $k-1$ dan katta bo'lmagan oraliq tugunlarga ega va shuning uchun $r_{ij}{}^{k-1}$ ham 1 ga teng bo'ladi. Ikkinchi ehtimol shundan iboratki, (4.2) yo'l haqiqatan ham oraliq tugunlar orasidagi $k$-tugun $v_k$ ni o'z ichiga oladi. Umumiylikni yo'qotmagan holda $v_k$ bu ro'yxatda faqat bir marta uchraydi deb faraz qilish mumkin. (Agar bunday bo'lmasa, bu xususiyat bilan $v_i$ dan $v_j$ gacha bo'lgan yangi yo'lni $v_k$ ning birinchi va oxirgi uchrashishlari orasidagi barcha uchlarni shunchaki yo'q qilish orqali hosil qilish mumkin bo'ladi.) Bu eslatma bilan (4.2) yo'lni quyidagicha qayta ifodalash mumkin:

$v_i$, oraliq tugunlar $\leq k-1$, $v_k$, oraliq tugunlar $\leq k-1$, $v_j$.

Bu ifodaning birinchi qismi har bir oraliq tugun $k-1$ dan katta bo'lmagan $v_i$ tugundan $v_k$ tugungacha yo'l mavjudligini (ya'ni, $r_{ik}{}^{k-1} =$

1), ikkinchi qismi esa har bir oraliq tugun $k-1$ dan katta bo'lmagan $v_k$ dan $v_j$ gacha yo'l mavjudligini bildiradi (ya'ni, $r_{kj}^{k-1}=1$).

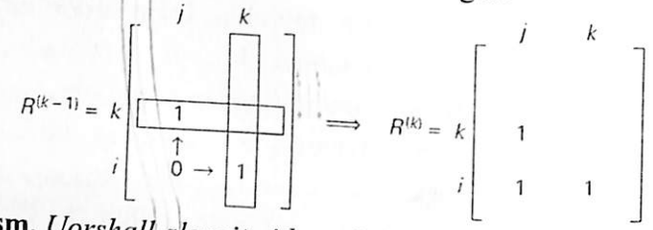
Bu isbotdan kelib chiqadiki, agar $r_{ij}^{k-1}=1$ bo'lsa, u holda yoki $r_{ij}^{k-1}=1$ yoki $r_{ik}^{k-1}=1$ va $r_{kj}^{k-1}=1$ bo'ladi. Bu tasdiqning teskarisi ham to'g'ri ekanligini ko'rish qiyin emas. Shunday qilib, $R^{(k-1)}$ matritsa elementlaridan $R^{(k)}$ matritsa elementlarini hosil qilish uchun quyidagi formula kelib chiqadi:

$$r_{ij}^{k}=r_{ij}^{k-1}\ yoki\ (r_{ik}^{k-1}\ va\ r_{kj}^{k-1})$$

(4.3).

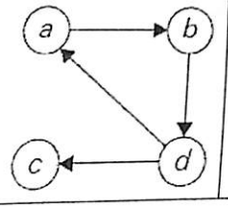
(4.3) formula Uorshall algoritmining asosini tashkil etadi. Bu formula $R^{(k-1)}$ matritsa elementlaridan $R^{(k)}$ matritsa elementlarini hosil qilishning quyidagi qoidasini nazarda tutadi, bu esa Uorshall algoritmini qo'lda bajarish uchun juda qulay ekanligini bildiradi:

- Agar $R^{(k-1)}$ matritsadagi $r_{ij}$ elementi 1 ga teng bo'lsa, u $R^{(k)}$ matritsada ham 1 bo'lib qoladi.
- Agar $R^{(k-1)}$ matritsadagi $r_{ij}$ elementi 0 bo'lsa va agar uning $i$-satri va $k$ustunidagi element hamda $j$-ustuni va $k$-satridagi elementlarning ikkalasi ham 1 bo'lsa, $R^{(k)}$da $r_{ij}$ elementni 1 ga o'zgartirish kerak. Bu qoida 4.2-rasmda tasvirlangan.




**4.2-rasm.** *Uorshall algoritmida nollarni almashtirish qoidasi*

Misol tariqasida 4.1-rasmdagi yo'naltirilgan grafga Uorshall algoritmining qo'llanilishi 4.3-rasmda ko'rsatilgan.

| | | |
|---|---|---|
|  | $R^{(0)}=$ | 1 lar oraliq tugunlari bo'lmagan yo'llarning mavjudligini aks ettiradi ($R^{(0)}$ faqat qo'shnilik matritsasi); alohida ajratib ko'rsatilgan katakli satr va |



| | a | b | c | d |
|---|---|---|---|---|
| a | 0 | 1 | 0 | 0 |
| b | 0 | 0 | 0 | 1 |
| c | 0 | 0 | 0 | 0 |
| d | 1 | 0 | 1 | 0 |



| | | |
|---|---|---|
| | | ustun $R^{(1)}$ ni hosil qilish uchun ishlatiladi. |
| | $R^{(1)}=$ | 1 lar oraliq tugunlari 1 tadan ko'p bo'lmagan, ya'ni faqat $a$ tugun bilan belgilangan yo'llarning mavjudligini aks ettiradi ($d$ dan $b$ gacha bo'lgan yangi yo'lga e'tibor bering); ajratilgan satr va ustun $R^{(2)}$ ni hosil qilish uchun ishlatiladi. |

| | a | b | c | d |
|---|---|---|---|---|
| a | 0 | 1 | 0 | 0 |
| b | 0 | 0 | 0 | 1 |
| c | 0 | 0 | 0 | 0 |
| d | 1 | **1** | 1 | 0 |

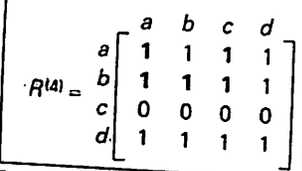
| | | |
|---|---|---|
| | $R^{(2)}=$ | 1 lar oraliq tugunlari 2 tadan ko'p bo'lmagan, ya'ni $a$ va $b$ bo'lgan yo'llarning mavjudligini aks ettiradi (ikkita yangi yo'lga e'tibor bering); ajratilgan satr va ustun $R^{(3)}$ ni hosil qilish uchun ishlatiladi. |

| | a | b | c | d |
|---|---|---|---|---|
| a | 0 | 1 | 0 | **1** |
| b | 0 | 0 | 0 | 1 |
| c | 0 | 0 | 0 | 0 |
| d | 1 | 1 | 1 | **1** |

| | | |
|---|---|---|
| | $R^{(3)}=$ | 1 lar oraliq tugunlari 3 tadan ko'p bo'lmagan, ya'ni $a$, $b$ va $c$ (yangi yo'llar yo'q) bilan belgilangan yo'llar mavjudligini aks ettiradi; |

| | a | b | c | d |
|---|---|---|---|---|
| a | 0 | 1 | 0 | 1 |
| b | 0 | 0 | 0 | 1 |
| c | 0 | 0 | 0 | 0 |
| d | 1 | 1 | 1 | 1 |

| | | |
|---|---|---|
| | | ajratilgan satr va ustun $R^{(4)}$ ni hosil qilish uchun ishlatiladi. |

| | | |
|---|---|---|
| | $R^{(4)} =$  | 1 lar oraliq tugunlari 4 tadan k'op bo'lmagan, ya'ni $a$, $b$, $c$, $d$ bilan belgilangan yo'llar mavjudligini aks ettiradi (beshta yangi yo'lga e'tibor bering). |



**4.3-rasm.** Uorshall algoritmining ko'rsatilgan yo'naltirilgan grafga qo'llanilishi. Yangi yo'llar qalin **1** bilan kiritilgan. Uorsholl algoritmining psevdokodi:

**ALGORITHM** *Uorshall*$(A[1..n, 1..n])$

// Tranzitiv yopilish matritsasini hisoblash uchun Uorshall algoritmi

// Kirish: $n$ ta tugunli yo'naltirilgan grafning $A$ qo'shnilik matritsasi

// Chiqish: Yo'naltirilgan grafning tranzitiv yopilish matritsasi

$R(0) \leftarrow A$ **for** $k \leftarrow 1$ **to**

$n$ **do for** $i \leftarrow 1$ **to** $n$ **do**

**for** $j \leftarrow 1$ **to** $n$ **do**

$R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$

**return** $R^{(n)}$

Uorshall algoritmi haqida bir nechta muhim kuzatish natijalari mavjud. Birinchidan, u juda ham qisqa va ixcham, shunday emasmi? Shunga qaramay, uning vaqt samaradorligi faqat $\Theta(n^3)$ bilan baholanadi. Aslida, qo'shnilik ro'yxati bilan ifodalangan ***siyrak graf***lar uchun ushbu bo'limning boshida aytib o'tilgan ***aylanib o'tishlarga*** asoslangan algoritm Uorshall algoritmiga qaraganda yaxshiroq asimptotik samaradorlikka ega (nima uchun?).

Uorshall algoritmining yuqoridagi tatbiqini ayrim kiritilgan ma'lumotlar uchun uning eng ichki siklini qayta tuzish orqali tezlashtirishimiz mumkin (ushbu bo'limdagi mashqlarning 4-masalasiga qarang). Algoritmni yanada tezroq ishlashini ta'minlashning yana bir usuli - matritsalar qatorini ***bit*** satrlar sifatida ko'rib chiqish va



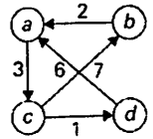
zamonaviy dasturlash tillarining aksariyatida mavjud bo'lgan ***bitli "yoki*** (or)" amalidan foydalanishdir.

Uorshall algoritmining xotira samaradorligiga kelsak, vaziyat Fibonachchi sonini hisoblash va boshqa ba'zi dinamik dasturlash algoritmlariga o'xshaydi. Algoritmning oraliq natijalarini qayd etish uchun alohida matritsalardan foydalangan bo'lsak-da, aslida bunga hojat yo'q. Ushbu bo'limdagi mashqlardan 3masalada kompyuter xotirasidan bunday isrofgarchilikka yo'l qo'ymaslik usulini topishni so'ralgan. Nihoyat, quyida biz Uorshall algoritmining asosiy g'oyasini vaznli graflarda eng qisqa yo'llar uzunligini topishning umumiyroq masalasiga qanday qo'llash mumkinligini ko'rib chiqamiz.

### 4.2. Floyd algoritmi.

Vaznli bog'langan graf (yo'naltirilmagan yoki yo'naltirilgan) berilgan bo'lsa, barcha juftliklar uchun eng qisqa yo'llar masalasi har bir tugundan boshqa barcha tugunlargacha bo'lgan masofalarni, ya'ni eng qisqa yo'llarning uzunliklarini topishni ifodalaydi. Bu graflardagi eng qisqa yo'llar masalasining bir nechta ko'rinishlaridan biri hisoblanadi. Aloqa, transport tarmoqlari va operatsiyalarni tadqiq qilishda muhim ahamiyatga ega bo'lgani sababli, bu masala yillar davomida chuqur o'rganib kelingan. Barcha juftliklar uchun eng qisqa yo'l masalasining so'nggi qo'llanilish sohalariga kompyuter o'yinlarida harakatlarni rejalashtirish uchun masofalarni oldindan hisoblash muammosini kiritish mumkin.

Eng qisqa yo'llar uzunligini $n \times n$ o'lchamli $D$ matritsa orqali ifodalash qulay bo'lib, bu matritsa masofalar matritsasi deb ataladi: bu matritsaning $i$-satri va $j$ustunidagi $d_{ij}$ elementi $i$-tugundan $j$-tugungacha bo'lgan eng qisqa yo'l uzunligini ifodalaydi. Misol uchun 4.4-rasmga qarang.





(a)

$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

(b)

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

(c)

**4.4-rasm.** *(a) Yo'naltirilgan graf. (b) Uning vazn matritsasi. (c) Uning masofa matritsasi.*

Masofa matritsasini Uorshall algoritmiga juda o'xshash bo'lgan usul bilan yaratish mumkin. Bu usul o'zining hammuallifi Robert V. Floyd[14] sharafiga Floyd algoritmi deb nomlanadi. U manfiy uzunlikdagi halqani o'z ichiga olmagan yo'naltirilmagan va yo'naltirilgan vaznli graflarga qo'llaniladi. (Bunday halqada istalgan ikki tugun orasidagi masofani siklni yetarlicha marta takrorlash orqali ixtiyoriy darajada kichik qilish mumkin.) Algoritmni nafaqat barcha tugunlar juftliklari uchun eng qisqa yo'llar uzunligini, balki eng qisqa yo'llarning o'zini ham topish uchun takomillashtirish mumkin (ushbu bo'lim mashqlaridagi 10-masalaga qarang).

Floyd algoritmi $n$ ta tugunli vaznli grafning masofa matritsasini $n \times n$ o'lchamli matritsalar ketma-ketligi orqali hisoblab chiqadi:

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)} \qquad (4.4)$$

Bu matritsalarning har biri qaralayotgan matritsa uchun ma'lum cheklovlarga ega bo'lgan eng qisqa yo'llarning uzunliklarini o'z ichiga oladi. Xususan, $D^{(k)}$ matritsaning $i$-satri va $j$ –ustunidagi $d_{ij}^{(k)}$ $(i, j = 1,2,\ldots, n, k = 0,1,\ldots, n)$ element

$i$-tugundan $j$-tugungacha bo'lgan barcha yo'llar orasidagi eng qisqa yo'lning uzunligiga teng, bunda har bir oraliq tugun, agar mavjud bo'lsa, $k$ dan ko'p bo'lmagan raqam bilan belgilangan. Ayniqsa, masofalar matritsalari qatori $D^{(0)}$ dan

boshlanadi, bu matritsadagi yo'llarda hech qanday oraliq tugunlar mavjud emas; shu sababli $D^{(0)}$ shunchaki grafning vazn matritsasi deyiladi. Qatordagi so'nggi matritsa

$D^{(n)}$ barcha $n$ ta tugundan oraliq sifatida foydalanishi mumkin bo'lgan yo'llar orasidagi eng qisqa yo'llarning uzunliklarini o'z ichiga oladi va shu bois u izlanayotgan masofa matritsasi hisoblanadi.

Xuddi Uorshall algoritmidagi kabi (4.4 qatorda berilgan) har bir $D^{(k)}$ matritsaning barcha elementlarini uning bevosita o'tmishdoshi

[14] Floyd o'z algoritmini taqdim etishda Uorshallning maqolasiga aniq havola qilgan [Flo62]. Uch yil oldin Bernard Roy xuddi shu algoritmni Fransiya Fanlar akademiyasi nashrlarida e'lon qilgan edi.



$D^{(k-1)}$ dan hisoblash mumkin. $D^{(k)}$ matritsaning $i$-satr va $j$-ustuni elementi $d_{ij}^{(k)}$ bo'lsin. Bu shuni anglatadiki, $d_{ij}^{(k)}$ $i$-tugun $v_i$ dan $j$-tugun $v_j$ gacha bo'lgan barcha yo'llar orasidagi eng qisqa yo'lning uzunligiga teng bo'lib, ularning oraliq tugunlari $k$ dan ko'p bo'lmagan sonlar bilan belgilanadi:

$v_i$, $k$ dan katta bo'lmagan raqamlangan oraliq tugunlar ro'yxati, $v_j$ **(4. 5)** Bunday yo'llarning barchasini ikkita kesishmaydigan qism to'plamlarga ajratishimiz mumkin: $k$-tugun $v_k$ dan oraliq sifatida foydalanmaydigan va foydalanadigan. Birinchi qism to'plamning yo'llari oraliq uchlari $k - 1$ dan ko'p bo'lmagan sonlar bilan belgilanganligi sababli, ularning eng qisqasi, matritsalar ta'rifiga ko'ra, $d_{ij}^{(k-1)}$ uzunlikka ega.

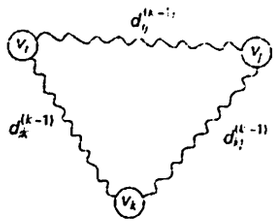
Ikkinchi qism to'plamdagi eng qisqa yo'lning uzunligi qancha? Agar grafda manfiy uzunlikdagi sikl mavjud bo'lmasa, ikkinchi qism to'plamdagi faqat bir marta $v_k$ tugunini oraliq tugun sifatida ishlatadigan yo'llarga e'tibor qaratish mumkin (chunki $v_k$ ga bir martadan ko'proq tashrif buyurish faqat yo'lning uzunligini oshirishi mumkin). Bunday yo'llarning barchasi quyidagi ko'rinishga ega:

$v_i$, oraliq tugunlar $\leq k - 1$, $v_k$, oraliq tugunlar $\leq k - 1$, $v_j$.

Boshqacha aytganda, yo'llarning har biri har bir oraliq tugun $k - 1$ dan katta bo'lmagan $v_i$ dan $v_k$ gacha bo'lgan yo'ldan va har bir oraliq tugun $k - 1$ dan katta bo'lmagan $v_k$ dan $v_j$ gacha bo'lgan yo'ldan iborat. 4.5-rasmda ushbu holat ramziy ma'noda tasvirlangan.

Sonlari $k - 1$ dan ko'p bo'lmagan oraliq tugunlardan foydalanuvchi yo'llar orasida $v_i$ dan $v_k$ gacha bo'lgan eng qisqa yo'lning uzunligi $d_{ik}^{(k-1)}$ ga va sonlari $k - 1$ dan ko'p bo'lmagan oraliq tugunlardan foydalanuvchi yo'llar orasida $v_k$ dan $v_j$ gacha bo'lgan eng qisqa yo'lning uzunligi $d_{kj}^{(k-1)}$ ga teng bo'lgani uchun, $k$tugundan foydalanuvchi yo'llar orasida eng qisqa yo'lning uzunligi $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ ga teng. Ikkala qism to'plamdagi eng qisqa yo'llarning uzunliklarini hisobga olish quyidagi rekursiv munosabatga olib keladi:

$$dij(k) = \min\{dij(k-1), dik(k-1) + dkj(k-1)\}, dij(0) = wij. \qquad (4.6)$$



**4.5-rasm.** *Floyd algoritmining asosiy g'oyasi.*

Boshqacha qilib aytganda, joriy masofa matritsasi $D^{(k-1)}$ ning $i$-satr va $j$ustunidagi element, faqat va faqat agar $i$-satr va $k$-ustundagi hamda $j$-ustun va $k$satrdagi elementlar yig'indisi uning hozirgi qiymatidan kichik bo'lsa, shu yig'indi bilan almashtiriladi. 4.4-rasmdagi grafga Floyd algoritmining qo'llanilishi 4.6rasmda ko'rsatilgan.

Quyida Floyd algoritmining psevdokodi keltirilgan. U (4.6) ketma-ketlikdagi keyingi matritsani o'zidan oldingi matritsaga yozish mumkinligidan foydalanadi.

**ALGORITHM** *Floyd*$(W[1..n, 1..n])$

//Barcha juftliklar uchun eng qisqa yo'llar masalasi uchun Floyd algoritmi

//Kirish: manfiy uzunlikdagi siklga ega bo'lmagan grafning $W$ vazn matritsasi

// Chiqish: Eng qisqa yo'llar uzunliklarining masofa matritsasi

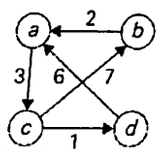
$D \leftarrow W$ //agar $W$ ni qayta yozish mumkin bo'lsa,

shart emas **for** $k \leftarrow 1$ **to** $n$ **do** **for** $i \leftarrow 1$ **to** $n$ **do for** $j \leftarrow 1$ **to** $n$ **do**

$$D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$$

**return** $D$

Shubhasiz, Floyd algoritmining vaqt bo'yicha samaradorligi xuddi Uorshall algoritmining vaqt bo'yicha samaradorligibilan bir xil kubik qiymatga ega.

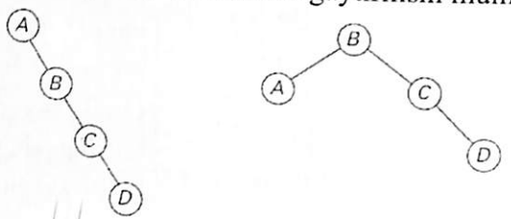
| | | |
|---|---|---|
|  | $D^{(0)} =$ | Oraliq tugunlari bo'lmagan eng qisqa yo'llarning uzunliklari ($D^{(0)}$- oddiy vazn matritsasi). |

| | | |
|---|---|---|
| | $D^{(1)} =$ | Oraliq tugunlari bo'lgan eng qisqa yo'llarning uzunliklari. 1 tadan ko'p bo'lmagan oraliq tugun, ya'ni faqat $a$ bilan belgilangan ($b$ dan $c$ gacha va $d$ dan $c$ gacha bo'lgan ikkita yangi eng qisqa yo'lga e'tibor bering). |
| | $D^{(2)} =$ | Oraliq tugunlari 2 tadan ko'p bo'lmagan eng qisqa yo'llarning uzunliklari, ya'ni $a$ va $b$ ($c$ dan $a$ gacha bo'lgan yangi eng qisqa yo'lga e'tibor bering). |
| | $D^{(3)} =$ | Oraliq tugunlari 3 tadan ko'p bo'lmagan, ya'ni $a$, $b$ va $c$ lar bilan belgilangan eng qisqa yo'llarning uzunliklari ($a$ dan $b$ gacha, $a$ dan $d$ gacha, $b$ dan $d$ gacha va $d$ dan $b$ gacha bo'lgan to'rtta yangi eng qisqa yo'llarga e'tibor bering). |
| | $D^{(4)} =$ | Oraliq tugunlari 4 tadan ko'p bo'lmagan eng qisqa yo'llarning uzunliklari, ya'ni $a$, $b$, $c$ va $d$ ($c$ dan $a$ gacha bo'lgan yangi eng qisqa yo'lga e'tibor bering). |

**4.6-rasm.** *Berilgan yo'naltirilgan grafga Floyd algoritmini qo'llash. Yangilangan elementlar qalin yozuvlar bilan ko'rsatilgan.*

### 4.3. Optimal Binar Qidiruv daraxti

Ikkilik qidiruv daraxti kompyuter fanidagi eng muhim ma'lumotlar tuzilmalaridan biri hisoblanadi. Uning asosiy qo'llanilish sohalaridan biri qidirish, qo'yish (element kiritish) va o'chirish amallariga ega elementlar to'plami bo'lgan lug'at tuzilmasini yaratishdir. Agar to'plam elementlarini qidirish ehtimolliklari ma'lum bo'lsa - masalan, oldingi qidiruvlar to'g'risida to'plangan ma'lumotlar mavjud bo'lganda - qidiruvda taqqoslashlarning o'rtacha soni eng kichik bo'lgan ***optimal binar qidiruv daraxtini*** qurish masalasi paydo bo'ladi. Soddalik uchun muvaffaqiyatli qidiruvda taqqoslashlarning o'rtacha sonini minimallashtirish bilan cheklanamiz. Usul muvaffaqiyatsiz qidiruvlarni ham o'z ichiga olgan holda kengaytirilishi mumkin.




**4.7-rasm.** *A, B, C va D kalitli, mumkin bo'lgan 14 ta holatdab 2 ta binar qidiruv daraxtlari.*

Misol tariqasida to'rtta A, B, C va D kalitlar mos ravishda 0.1, 0.2, 0.4 va 0.3 ehtimollik bilan izlanayotgan bo'lsin. 4.7-rasmda ushbu kalitlarni o'z ichiga olgan 14 ta mumkin bo'lgan binar qidiruv daraxtlaridan ikkitasi tasvirlangan. Ushbu daraxtlarning birinchisida muvaffaqiyatli qidiruvda taqqoslashlarning o'rtacha soni $0{,}1 \cdot 1 + 0{,}2 \cdot 2 + 0{,}4 \cdot 3 + 0{,}3 \cdot 4 = 2{,}9$ ga teng, ikkinchisi uchun esa $0{,}1 \cdot 2 + 0{,}2 \cdot 1 + 0{,}4 \cdot 2 + 0{,}3 \cdot 3 = 2.1$ ga teng.

Bu ikki daraxtning hech biri aslida optimal emas. (Qaysi binar daraxt optimal ekanligini ayta olasizmi?). Bu kichik misol uchun optimal daraxtni ushbu kalitlar bilan barcha 14 ta binar qidiruv daraxtlarini hosil qilish orqali topish mumkin. Umumiy algoritm sifatida bunday to'liq-qidiruv yondashuvi haqiqatga to'g'ri kelmaydi: $n$ kalitli binar qidiruv daraxtlarining umumiy soni $n$-Katalon soni:

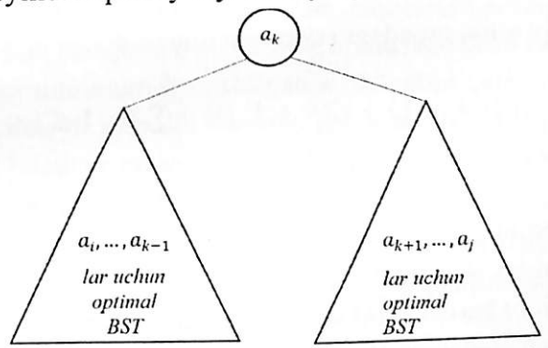
$$\frac{1}{n}\binom{2n}{n}$$



$$c(n) = \frac{1}{n}\binom{2n}{n}, n + 1n$$

bu yerda $n > 0$, $c(0) = 1$, u $4^n$.5 kabi tez cheksizlikka intiladi (ushbu bo'limdagi mashqlardagi 17-masalaga qarang).

Shunday qilib, eng kichigidan eng kattasigacha tartiblangan $a_1, \ldots, a_n$ turli kalitlar va ularni qidirish ehtimolliklari $p_1, \ldots, p_n$ lar berilgan bo'lsin. $C(i, j)$ esa $a_i, \ldots, a_j$ kalitlardan tashkil topgan $T_i^j$ binar qidiruv daraxtida muvaffaqiyatli
qidiruvda amalga oshirilgan taqqoslashlarning eng kichik o'rtacha soni bo'lsin (bu yerda $i$ va $j$ qandaydir butun indekslar, $1 \le i \le j \le n$). Klassik dinamik dasturlash yondashuviga ko'ra, biz faqat $C(1, n)$ ga qiziqsak ham, masalaning barcha kichikroq holatlari uchun $C(i, j)$ ning qiymatlarini aniqlaymiz. Dinamik dasturlash algoritmi asosida yotuvchi rekursiyani keltirib chiqarish uchun $a_i, \ldots, a_j$ kalitlar orasidan $a_k$ ildizni tanlashning barcha mumkin bo'lgan usullarini ko'rib chiqamiz. Bunday binar qidiruv daraxti uchun (4.8-rasm) ildizdagi $a_k$ kalit uchun, $T_i^{k-1}$ chap qism daraxtda
$a_i, \ldots, a_{k-1}$ kalitlar optimal joylashgan, $T_{k+1}^j$ o'ng qism daraxtda esa $a_{k+1}, \ldots, a_j$ kalitlar ham optimal joylashgan bo'ladi. (Bu yerda biz optimallik tamoyilidan qanday foydalanayotganimizga e'tibor bering.)




**4.8-rasm.** *Binar qidiruv daraxti (BST) ildizi* ak *va ikkita optimal binar qidiruv qism daraxtlari* $T_i^{k-1}$ *va* $T_{k+1}^j$.

Agar daraxt darajalarini kalitlarning darajalariga tenglashtirish uchun 1 dan boshlab sanasak, quyidagi takrorlanish munosabati kelib chiqadi:

$$C(i,j) = \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot (T_i^{k-1} + 1 \text{ dagi } a_s \text{ daraja}) + \sum_{s=k+1}^{j} p_s \cdot (T_{k+1}^{j} + 1 \text{ dagi } a_s \text{ daraja})\}$$

$$= \min_{i \le k \le j} \{\sum_{s=i}^{k-1} p_s \cdot (T_i^{k-1} + 1 \text{ dagi } a_s \text{ daraja}) + \sum_{s=k+1}^{j} p_s \cdot (T_{k+1}^{j} + 1 \text{ dagi } a_s \text{ daraja}) + \sum_{s=i}^{j} p_s\}$$

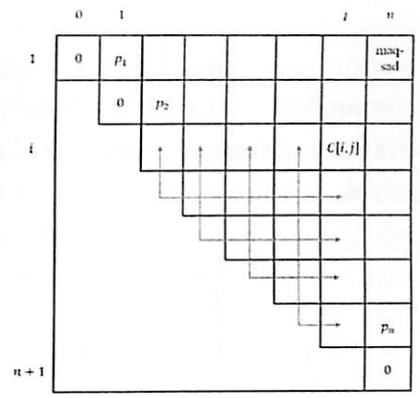
$$= \min_{i \le k \le j}\{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s.$$

Shunday qilib, quyidagi rekkurent munosabatga ega bo'lamiz:

$$C(i,j) = \min_{i \le k \le j}\{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s, \; 1 \le i \le j \le n \quad (4.7)$$

(4.7) formulada $1 \le i \le n+1$ uchun $C(i, i-1) = 0$ deb faraz qilamiz. Buni bo'sh daraxtdagi taqqoslashlar soni sifatida talqin etish mumkin. E'tibor berish kerakki, ushbu formula $1 \le i \le n$ uchun $C(i, i) = pi$ ekanligini anglatadi. Bu esa $a_i$ ni o'z ichiga olgan bir tugunli binar qidiruv daraxti uchun to'g'ri keladi.






**4.9-rasm**. *Optimal binar qidiruv daraxtini qurish uchun dinamik dasturlash algoritmi jadvali.*

4.9-rasmdagi ikki o'lchovli jadvalda (4.7) formula bo'yicha $C(i, j)$ ni hisoblash uchun zarur bo'lgan qiymatlar ko'rsatilgan: ular i-qatorda va j-ustundan chapdagi ustunlarda hamda j-ustunda va i-satrdan pastdagi satrlarda joylashgan. Strelkalar $C(i, j)$ qiymatining eng kichigini topish va yozish uchun hisoblanadigan yozuvlar juftligining yig'indisini ko'rsatadi. Bu jadvalni diagonallari bo'ylab to'ldirishni taklif qiladi. Asosiy diagonaldagi barcha nollardan boshlab, uning tepasida berilgan $p_i$ $(1 \le i \le n)$ ehtimolliklar bilan, yuqori o'ng burchak tomon harakatlanish orqali to'ldiriladi.

Yuqorida tasvirlangan algoritm $C(1, n)$ ni – ya'ni optimal binar daraxtda muvaffaqiyatli qidiruvlar uchun taqqoslashlarning o'rtacha sonini hisoblaydi. Agar optimal daraxtning o'zini ham olish zarur bo'lsa, (4.7) formuladagi minimumga erishilgan $k$ qiymatini qayd etish uchun yana bir ikki o'lchovli jadvalni yaratish kerak bo'ladi. Bu jadval 4.9-rasmdagi jadval bilan bir xil shaklga ega bo'lib, $1 \le i \le n$ uchun $R(i, i) = i$ yozuvlaridan boshlab aynan shu tartibda to'ldiriladi. Jadval to'ldirilgandan so'ng, uning yozuvlari optimal kichik daraxtlarning ildiz indekslarini ko'rsatadi. Bu esa berilgan butun to'plam uchun optimal daraxtni qayta tiklash imkoniyatini yaratadi.

**Misol.** Ushbu bo'limning boshida ishlatilgan to'rt kalitli to'plamga algoritmni qo'llash:

| kalit | A | B | C | D |
|---|---|---|---|---|
| ehtimolligi | 0.1 | 0.2 | 0.4 | 0.3 |

Dastlabki jadvallar quyidagi ko‘rinishda bo‘ladi:

asosiy jadval

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 1 | 0 | 0.1 | | | |
| 2 | | 0 | 0.2 | | |
| 3 | | | 0 | 0.4 | |
| 4 | | | | 0 | 0.3 |
| 5 | | | | | 0 |

ildiz jadval

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 1 | | 1 | | | |
| 2 | | | 2 | | |
| 3 | | | | 3 | |
| 4 | | | | | 4 |
| 5 | | | | | |

Ushbu berilganlardan $C(1,2)$ ni hisoblaymiz:

$$k = 1: C(1,0) + C(2,2) + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5$$

$$C(1,2) = \min\{k = 2: C(1,1) + C(3,2) + \sum 2ps = 0.1 + 0 + 0.3 = 0.4\} = 0.4$$

$s$=1

Demak, dastlabki ikkita kalitni o‘z ichiga olgan ikkita mumkin bo‘lgan binar daraxtdan, $A$ va $B$, optimal daraxtning ildizi indeksi 2 ga (ya’ni u $B$ ni o‘z ichiga oladi) va bu daraxtda muvaffaqiyatli qidiruvda taqqoslashlarning o‘rtacha soni 0,4 ga teng. Misoldagi hisoblashlarni ko’rsatilgan namuna asosida to’liq bajaring. Natijada quyidagi jadvallarga ega bo’limiz kerak:

asosiy jadval

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 1 | 0 | 0.1 | 0.4 | 1.1 | 1.7 |
| 2 | | 0 | 0.2 | 0.8 | 1.4 |
| 3 | | | 0 | 0.4 | 1.0 |
| 4 | | | | 0 | 0.3 |
| 5 | | | | | 0 |

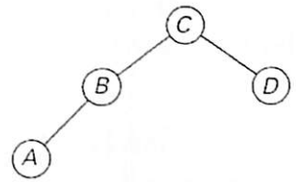
ildiz jadval

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 1 | | 1 | 2 | 3 | 3 |
| 2 | | | 2 | 3 | 3 |
| 3 | | | | 3 | 3 |
| 4 | | | | | 4 |
| 5 | | | | | |

Shunday qilib, optimal daraxtdagi asosiy taqqoslashlarning o‘rtacha soni 1.7 ga teng. $R(1,4) = 3$ bo‘lgani uchun optimal daraxtning ildizi uchinchi kalitni, ya’ni $C$ ni o‘z ichiga oladi. Uning chap qism daraxti $A$ va $B$ kalitlardan, o‘ng qism daraxti esa faqat $D$ kalitni o‘z



ichiga oladi (nima uchun?). Ushbu kichik daraxtlarning o‘ziga xos tuzilishini topish uchun ildiz jadvaliga yana bir bor murojaat qilib, ularning ildizlarini topamiz. $R(1,2) = 2$ bo‘lgani uchun $A$ va $B$ ni o‘z ichiga olgan optimal daraxtning ildizi $B$ bo‘lib, $A$ uning chap davomchisi (va bitta tugun daraxtining ildizi: $R(1,1) = 1$) bo‘ladi. $R(4,4) = 4$ bo‘lgani uchun bu bir tugunli optimal daraxtning ildizi uning yagona kaliti $D$ dir. 4.10-rasmda optimal daraxt to‘liq ko‘rsatilgan.




**4.10-rasm.** *Berilgan misol uchun optimal binar qidiruv daraxti.* Dinamik dasturlash algoritmining psevdokodi:

**ALGORITHM** *OptimalBST*$(P[1..n])$
//Dinamik dasturlash orqali optimal binar qidiruv daraxtini topadi
//Kirish: $n$ ta kalitdan iborat tartiblangan ro‘yxat uchun
//qidiruv ehtimolliklari massivi $P[1..n]$
//Natija: Optimal BSTdagi muvaffaqiyatli qidiruvlarda
// taqqoslashlarning o‘rtacha soni va optimal BSTdagi
//qism daraxtlar ildizlarining $R$
jadvali **for** $i \leftarrow 1$ **to** $n$ **do** $C[i, i-1] \leftarrow 0$
    $C[i, i] \leftarrow P[i]$
    $R[i, i] \leftarrow i$
$C[n+1, n] \leftarrow 0$
**for** $d \leftarrow 1$ **to** $n-1$ **do** //diagonal
    bo’yicha **for** $i \leftarrow 1$ **to** $n-d$ **do**
        $j \leftarrow i + d$ $minval \leftarrow \infty$ **for** $k \leftarrow i$ **to** $j$ **do if** $C[i, k-1] + C[k+1, j] < minval$
            $minval \leftarrow C[i, k-1] + C[k+1, j]$; $kmin \leftarrow k$
        $R[i, j] \leftarrow kmin$

$sum \leftarrow P[i]$; **for** $s \leftarrow i+1$ **to** $j$ **do** $sum \leftarrow sum + P[s]$
$C[i,j] \leftarrow minval + sum$
**return** $C[1,n], R$

Algoritmning xotira samaradorligi aniq kvadratik qiymat bilan, uning vaqt samaradorligi esa kubik qiymat bilan aniqlangan (nima uchun?). Chuqurroq tahlil shuni ko'rsatadiki, ildiz jadvalidagi qiymatlar har bir satr va ustun bo'ylab doimo kamaymaydigan tartibda joylashgan. Bu $R(i,j)$ qiymatlarini $R(i,j-1),\ldots,R(i+1,j)$ oralig'ida chegaralaydi va algoritmning ishlash vaqtini $\Theta(n^2)$ gacha qisqartirish imkonini beradi.

### 4.4. Mustaqil ishlash uchun savollar va mashqlar

1. Quyidagi qo'shnilik matritsasi bilan aniqlangan yo'naltirilgan grafning tranzitiv yopilishini topish uchun Warshall algoritmini qo'llang:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

2. ***a.*** Uorsholl algoritmining vaqt bo'yicha samaradorligi kubik ekanligini isbotlang.

***b.*** Nima uchun Uorshall algoritmining vaqt samaradorligi qo'shnilik ro'yxati bilan ifodalangan siyrak graflar uchun grafda o'tishga asoslangan algoritmdan past ekanligini tushuntiring.

3. Uorshall algoritmini qo'shimcha xotiradan foydalanmasdan, ya'ni algoritmning oraliq matritsa elementlarini saqlamasdan amalga oshirishni tushuntiring.

4. Uorshall algoritmining eng ichki siklini qayta tuzishni tushuntiring, shunda u ayrim kirish ma'lumotlarida tezroq ishlaydi.

5. Uorshall algoritmi psevdokodini qayta yozing, bunda matritsa satrlari bitli satrlar ko'rinishida ifodalangan bo'lib, ularga nisbatan bitli "yoki" amalini bajarish mumkin.

6. ***a.*** Berilgan yo'naltirilgan grafning asiklik (yo'naltirilgan asiklik graf - DAG) ekanligini aniqlash uchun Uorshall algoritmidan qanday



foydalanish mumkinligini tushuntiring. Bu masala uchun samarali algoritmmi?

***b.*** Yo'naltirilmagan grafning tranzitiv yopilishini topish uchun Uorshall algoritmini qo'llash maqsadga muvofiqmi?

7. Quyidagi vazn matritsasiga ega bo'lgan yo'naltirilgan graf uchun barcha juftliklar orasidagi eng qisqa yo'l masalasini yeching:

$$\begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix}$$

8. Floyd algoritmining (8.12) ketma-ketligidagi keyingi matritsani o'zidan oldingi matritsa ustiga yozish mumkinligini isbotlang.

9. Floyd algoritmi to'g'ri natija bermaydigan manfiy vaznli graf yoki yo'naltirilgan grafga misol keltiring.

10. Floyd algoritmini shunday takomillashtiringki, eng qisqa yo'llarning uzunligi emas, balki ularning o'zini topish mumkin bo'lsin.

11. *Jek Straus*. Jek somon poxollari o'yinida stolga bir nechta plastik yoki yog'och "poxollar" tashlanadi va o'yinchilar boshqa poxollarga xalaqit bermasdan ularni birin-ketin olib tashlashga harakat qilishadi. Bu yerda bizni turli juft poxollar bir-biriga tegib turgan poxollar yo'li bilan bog'langanmi-yo'qmi, degan masala qiziqtiradi, xolos. $n > 1$ ta somon uchun oxirgi nuqtalar ro'yxati berilgan (xuddi ular katta millimetrli qog'ozga tashlangandek), bog'langan barcha somon juftlarini aniqlang. E'tibor bering, tegib ketish bog'lashdir, lekin ikkita poya boshqa bog'langan poyalar orqali bilvosita bog'lanishi mumkin. [1994-yil ACM xalqaro talablar uchun dasturlash tanlovining Sharqiy-Markaziy mintaqalari].

12. ***a.*** OptimalBST algoritmining vaqt bo'yicha samaradorligi nima uchun kubik?

***b.*** OptimalBST algoritmining xotira samaradorligi nima uchun kvadratik?

**13.** Ildiz jadvalidan foydalanib, optimal binar qidiruv daraxtini chiziqli vaqt samaradorligiga ega bo'lgan algoritm uchun psevdokod yozing.

**14.** Optimal binar qidiruv daraxtini qurish uchun dinamik dasturlash algoritmida ishlatiladigan yig'indilar $\sum_{s=i}^{j} p_s$ni doimiy (konstanta) vaqtda (har bir yig'indi uchun) hisoblash usulini ishlab chiqing.

**15.** To'g'ri yoki noto'g'ri: Optimal binar qidiruv daraxtining ildizi doimo qidirish ehtimoli eng yuqori bo'lgan kalitni o'z ichiga oladi?

**16.** Agar barcha kalitlarni qidirish ehtimoli teng bo'lsa, $n$ ta kalitdan iborat to'plam uchun optimal binar qidiruv daraxtini qanday tuzgan bo'lardingiz? Agar $n = 2^k$ bo'lsa, bunday daraxtda muvaffaqiyatli qidiruv uchun taqqoslashlarning o'rtacha soni qancha bo'ladi?

**17.** ***a.*** $n$ ta tartibga solinadigan kalitlar to'plami uchun qurilishi mumkin bo'lgan alohida binar qidiruv daraxtlari soni $b(n)$ quyidagi rekurrent munosabatni qanoatlantirishini ko'rsating:

$$b(n) = \sum_{k=0}^{n-1} b(k)b(n-1-k), \quad n > 0, b(0) = 1.$$

***b.*** Ushbu rekursiv munosabatning yechimi Katalan sonlari bilan ifodalanishi
ma'lum. Bu tasdiqni $n = 1,2,\ldots,5$ qiymatlari uchun tekshirib ko'ring.

***c.*** $b(n)$ ning o'sish tartibini toping. Bu savolga javobning optimal binar qidiruv daraxtini yaratishning to'liq-qidiruv algoritmi uchun qanday ahamiyati bor?

**18.** Optimal binar qidiruv daraxtini topishning $\Theta(n^2)$ algoritmini tuzing.

**19.** Muvaffaqiyatsiz qidiruvlarni ham hisobga olgan holda optimal binar qidiruv algoritmini umumlashtiring.

**20.** Optimal binar qidiruv daraxti masalasi uchun xotira funksiyasining psevdokodini yozing. Muvaffaqiyatli qidiruvda kalitlarni taqqoslashning eng kam sonini topish bilan cheklanishingiz mumkin.



**21.** Matritsalar zanjirini ko'paytirish. O'lchamlari mos ravishda $d_0 \times d_1, d_1 \times d_2, \ldots, d_{n-1} \times d_n$ bo'lgan $n$ ta

$$A_1 \cdot A_2 \cdot \ldots \cdot A_n$$

matritsalar ko'paytmasini hisoblashda amalga oshiriladigan ko'paytirishlarning umumiy sonini minimallashtirish masalasini ko'rib chiqing. Barcha oraliq ko'paytmalar ikki matritsa uchun "qo'pol kuch" (ta'rifga asoslangan) algoritmi bilan hisoblangan deb faraz qiling.

***a.*** $(A_1 \cdot A_2) \cdot A_3$ va $A_1 \cdot (A_2 \cdot A_3)$ ko'paytmalaridagi amallar soni kamida 1000 marta farq qiladigan uchta matritsaga misol keltiring.

***b.*** $n$ ta matritsaning ko'paytmasini hisoblashning nechta turli usuli mavjud?

***c.*** $n$ ta matritsani ko'paytirishning optimal tartibini topish uchun dinamik
dasturlash algoritmini ishlab chiqing.

# 5-BOB. SHOXLANISH VA CHEGARALASH MASALALARI

## 5.1. Shoxlanish va chegaralash usulining mazmuni

Optimallashtirish masalasi odatda ba'zi cheklovlarga bog'liq bo'lgan biror maqsad funksiyasini (yo'l uzunligi, tanlangan elementlar qiymati, vazifaning narxi va boshqalar) minimallashtirish yoki maksimallashtirishga intiladi. Shuni ta'kidlash kerakki, optimallashtirish masalalarining standart atamalarida ***mumkin bo'lgan yechim*** - bu masalaning barcha cheklovlarini qondiradigan masalaning qidiruv fazosidagi nuqtadir (masalan, sayohatchi savdogar masalasida Gamilton zanjiri yoki sumka masalasida umumiy og'irligi sumka sig'imidan oshmaydigan narsalar to'plami), ***optimal yechim*** esa maqsad funksiyasining eng yaxshi qiymatiga ega bo'lgan yechimidir (masalan, eng qisqa Gamilton zanjiri yoki sumkaga mos keladigan eng qimmatli narsalar to'plami).

Shoxlanish va chegaralash (Branch-and-Bound) ikkita qo'shimcha elementni talab qiladi:

- holat-fazo daraxtining [15] har bir tuguni uchun tugun bilan ifodalangan qisman qurilgan yechimga qo'shimcha komponentlarni qo'shish orqali olinishi mumkin bo'lgan har qanday yechimdagi maqsad funksiyasining eng yaxshi qiymati bo'yicha bog'lanishni ta'minlash usuli;
- hozirgacha ko'rilgan eng yaxshi yechimning qiymati.

Agar bu ma'lumot mavjud bo'lsa, biz tugunning chegaraviy qiymatini shu paytgacha topilgan eng yaxshi yechim qiymati bilan solishtira olamiz. Agar chegaraviy qiymat shu paytgacha topilgan eng yaxshi yechim qiymatidan yaxshiroq bo'lmasa, ya'ni, minimallashtirish masalasi uchun kichikroq yoki maksimallashtirish masalasi uchun kattaroq bo'lmasa - tugun keraksiz hisoblanadi va uni tarmoqdan chiqarish (odatda bu jarayon "shoxni kesish" deb ataladi).

[15] Holat-fazo daraxti – bu algoritmlarning muammolarni yechish jarayonida yaratadigan barcha mumkin bo'lgan holatlari va ularning o'zaro bog'liqligini ifodalovchi daraxt ko'rinishidagi tuzilma.



Darhaqiqat, bunday tugundan olingan har qanday yechim mavjud bo'lgan yechimdan yaxshiroq natija bera olmaydi. Bu "shoxlanish va chegaralash" usulining asosiy g'oyasi hisoblanadi.

Umuman olganda, shoxlanish va chegaralash algoritmning holat-fazo daraxtida quyidagi uchta sababdan biriga ko'ra joriy tugunda qidiruv yo'lini tugatamiz:

- Tugun chegarasining qiymati shu paytgacha ko'rilgan eng yaxshi yechim qiymatidan yaxshi emas.
- Tugun mumkin bo'lgan yechimlarni ifodalamaydi, chunki masalaning cheklovlari allaqachon buzilgan.
- Tugun bilan ifodalangan mumkin bo'lgan yechimlarning qism to'plami bitta nuqtadan iborat (va shuning uchun boshqa tanlov qilish mumkin emas) - bu holda biz ushbu mumkin bo'lgan yechim uchun maqsad funksiyasining qiymatini shu paytgacha ko'rilgan eng yaxshi yechimning qiymati bilan taqqoslaymiz va agar yangi yechim yaxshiroq bo'lsa, ikkinchisini birinchisi bilan yangilaymiz.

## 5.2. Vazifani taqsimlash masalasi

Shoxlanish va chegaralash yondashuvini $n$ nafar shaxsga $n$ ta vazifani taqsimlash masalasiga qo'llaymiz. Bunda umumiy xarajat iloji boricha kam bo'lishi sharti qo'yiladi. Vazifani taqsimlash masalasining har bir holati $n \times n$ o'lchamli $C[i, j]$, $(0 \le i, j \le n-1)$ ***xarajatlar matritsasi*** bilan beriladi, shuning uchun masalani quyidagicha ifodalashimiz mumkin: matritsaning har bir satridan bittadan element shunday tanlansinki, tanlangan elementlarning hech qaysi ikkitasi bir ustunda bo'lmasin va ularning yig'indisi eng kichik bo'lsin. Biz bu masalani quyidagi rasmda berilgan kichik misol yordamida shoxlanish va chegaralash usuli bilan qanday yechish mumkinligini ko'rib chiqamiz (5.1-rasm):

$$C = \begin{array}{c} \begin{matrix} \text{1-v} & \text{2-v} & \text{3-v} & \text{4-v} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ a \text{ xodim} \\ b \text{ xodim} \\ c \text{ xodim} \\ d \text{ xodim} \end{matrix}$$

**5.1-rasm**. *Xarajatlar matritsasi (bu yerda 1-v, 2-v, 3-v, 4-v – vazifalar).*

Masalani amalda yechmasdan turib, optimal tanlov narxining quyi chegarasini qanday aniqlashimiz mumkin? Buni bir necha usul bilan amalga oshirishimiz mumkin. Masalan, har qanday yechimning, jumladan optimal yechimning narxi, matritsaning har bir satridagi eng kichik elementlar yig'indisidan past bo'la olmasligi aniq. Ushbu holatda, bu yig'indi 2+3+1+4=10 ga teng. Shuni ta'kidlash lozimki, bu har qanday qonuniy tanlovning narxi emas (3 va 1 matritsaning bir ustunidan olingan); bu shunchaki har qanday qonuniy tanlovning quyi chegarasidir.

Xuddi shu mantiqni qisman tuzilgan yechimlarga ham qo'llashimiz mumkin va qo'llaymiz. Misol uchun, birinchi qatordan 9 ni tanlaydigan har qanday qonuniy tanlov uchun quyi chegara 9+3+1+4=17 bo'ladi.

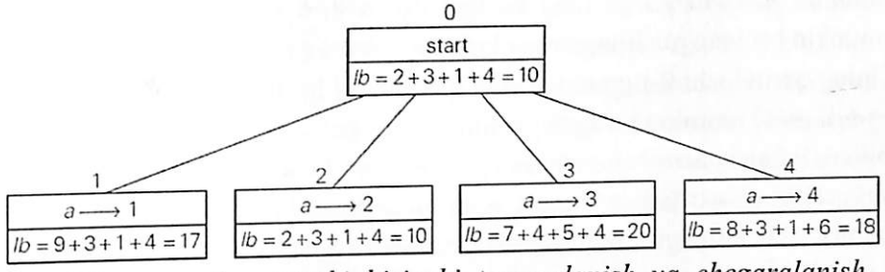
Masalaning holat-fazo daraxtini tuzishga kirishishdan oldin yana bir izohni keltirib o'tamiz. U daraxt tugunlari hosil qilinadigan tartib bilan shug'ullanadi. Teskari kuzatishdagi kabi oxirgi istiqbolli tugunning bitta farzandini hosil qilish o'rniga, biz joriy daraxtdagi tugallanmagan barglar orasidagi eng istiqbolli tugunning barcha farzandlarini hosil qilamiz. (Uzilmagan, ya'ni hali istiqbolli bo'lgan barglar tirik barglar deb ham ataladi.) Tugunlardan qaysi biri eng istiqbolli ekanligini qanday aniqlash mumkin? Buni biz tirik tugunlarning pastki chegaralarini taqqoslash orqali amalga oshirishimiz mumkin. Eng yaxshi bog'lanishga ega bo'lgan tugunni eng istiqbolli deb hisoblash mantiqan to'g'ri, garchi bu, albatta, optimal yechim oxir-oqibat holat-fazo daraxtining boshqa tarmog'iga tegishli bo'lishi ehtimolini istisno qilmaydi. Strategiyaning bu varianti ***eng yaxshi birinchi shoxlanish va chegaralash*** deb ataladi.

Shunday qilib, yuqorida berilgan o'zlashtirish masalasiga qaytib, xarajatlar matritsasidan tanlangan hech qanday elementga mos kelmaydigan ildizdan boshlaymiz. Yuqorida muhokama qilganimizdek, *lb* deb belgilangan ildizning quyi chegaralangan qiymati 10 ga teng. Daraxtning birinchi darajasidagi tugunlar matritsaning birinchi satridagi

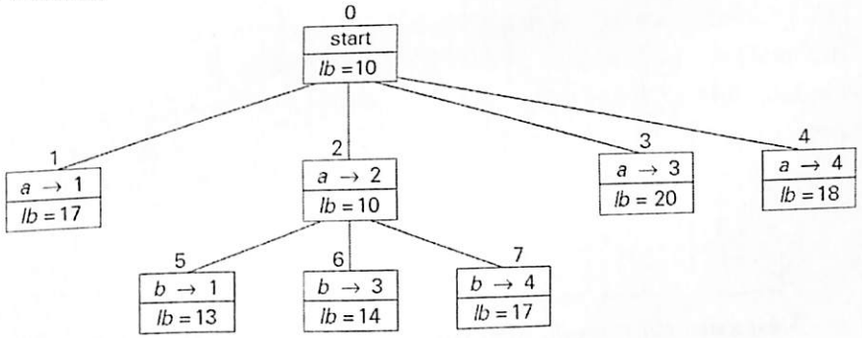


elementning tanlanishiga, ya'ni a shaxs uchun taqsimlangan vazifaga mos keladi (5.2-rasm).

Shunindek, bizda optimal yechimni o'z ichiga olishi mumkin bo'lgan to'rtta ***tirik barglar*** 1 dan 4 gacha bo'lgan tugunlar mavjud. Ulardan eng istiqbollisi 2tugun hisoblanadi, chunki u eng kichik quyi chegaraviy qiymatga ega. Bizning eng yaxshi qidiruv strategiyamizdan so'ng, biz birinchi navbatda ikkinchi ustunda emas, balki ikkinchi satrdan elementni tanlashning uch xil usulini - b shaxsga berilishi mumkin bo'lgan uch xil vazifani ko'rib chiqib, o'sha tugundan voz kechamiz (5.3rasm).

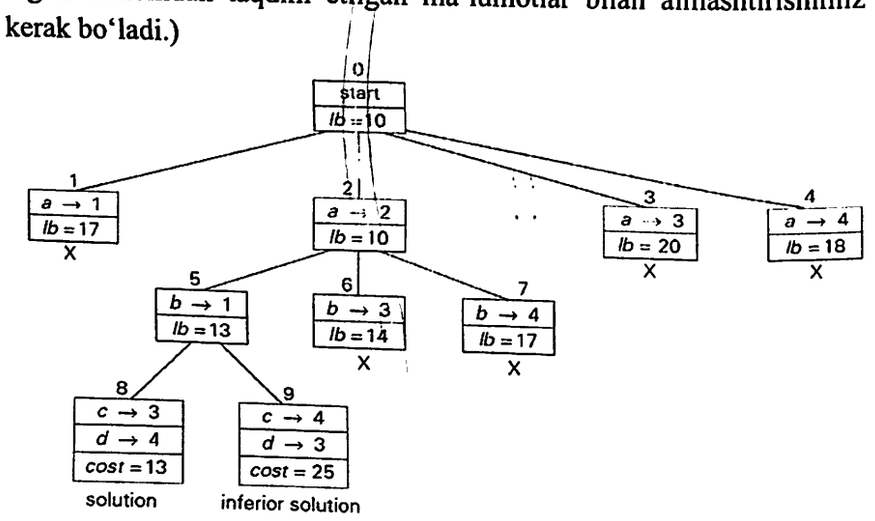



**5.2-rasm**. *Eng yaxshi birinchi tarmoqlanish va chegaralanish algoritm bilan yechiladigan o'zlashtirish masalasi uchun holat-fazo daraxtining 0 va 1 darajalari. Tugun ustidagi son tugunning hosil bo'lish tartibini ko'rsatadi. Tugunning maydonlari* a *shaxsga berilgan vazifa raqamini va ushbu tugun uchun quyi chegaraviy qiymat* lb *ni ko'rsatadi.*



**5.3-rasm.** *Eng yaxshi birinchi tarmoqlangan va chegaralangan algoritm bilan o'zlashtirish masalasi yechilayotgan holat-fazo daraxtining 0, 1 va 2 darajalari.*

Optimal yechimni o'z ichiga olishi mumkin bo'lgan oltita tirik barg - 1, 3, 4, 5, 6 va 7-tugunlardan yana eng kichik quyi chegaraga ega bo'lgan 5-tugunni tanlaymiz. Dastlab, uchinchi ustun elementini $c$ satridan tanlashni ko'rib chiqamiz
(ya'ni $c$ shaxsni 3-ishga tayinlash); bu bizga $d$ satrining to'rtinchi ustunidan elementni tanlashdan boshqa iloj qoldirmaydi ($d$ shaxsni 4-ishga tayinlash). Natijada 8 ta barg hosil bo'ladi (5.4-rasm), bu esa umumiy qiymati 13 ga teng bo'lgan $\{a \to 2, b \to 1, c \to 3, d \to 4\}$ mumkin bo'lgan yechimga mos keladi.
Uning qarindoshi 9-tugun umumiy qiymati 25 bo'lgan $\{a \to 2, b \to 1, c \to 4, d \to 3\}$ mumkin bo'lgan yechimga mos keladi. Uning narxi 8-tugun bilan ifodalangan yechim narxidan katta bo'lganligi sababli, 9-tugun shunchaki chiqariladi. (Albatta, agar uning narxi 13 dan kam bo'lsa, biz hozirgacha ko'rilgan eng yaxshi yechim haqidagi ma'lumotni ushbu tugun tomonidan taqdim etilgan ma'lumotlar bilan almashtirishimiz kerak bo'ladi.)




**5.4-rasm.** *Eng yaxshi birinchi shoxlanish va cegaralash algoritm bilan yechilgan vazifalarni taqsimlash masalasi misoli uchun to'liq holat-fazo daraxti.*

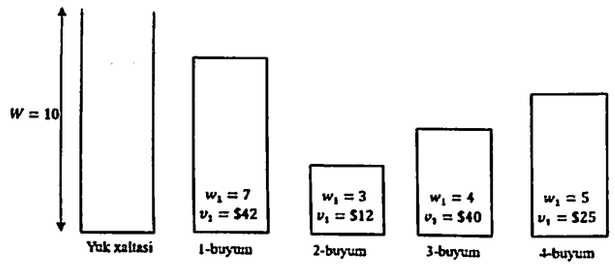


Endi 5.4-rasmdagi oxirgi holat-fazo daraxtining har bir tirik barglarini - 1, 3, 4, 6 va 7-tugunlarni ko'zdan kechirar ekanmiz, ularning quyi chegara qiymatlari 13 dan kichik emasligini aniqlaymiz, bu hozirgacha ko'rilgan eng yaxshi tanlov qiymati (8-barg) hisoblanadi. Demak, ularning barchasini tugatamiz va 8-tugun bilan ifodalangan yechimni masalaning optimal yechimi deb qabul qilamiz.

### 5.3. Yuk xaltasi (Ryukzak) masalasi

Algoritmlarni loyihlashning yana bir mashhur masalasi. $n$ ta buyumning og'irliklari $w_1, w_2, \ldots, w_n$ va qiymatlari $v_1, v_2, \ldots, v_n$ hamda $W$ sig'imli yuk xaltasi (rukzak) berilgan bo'lsa, yuk xaltasiga sig'adigan buyumlarning eng qimmatli to'plamini toping. Agar o'zingizni rukzagiga sig'adigan eng qimmatbaho o'ljani o'g'irlamoqchi bo'lgan o'g'ri o'rnida tasavvur qilish g'oyasi sizga yoqmasa, samolyot sig'imidan oshmagan holda eng qimmatbaho buyumlar to'plamini uzoq manzilga yetkazishi kerak bo'lgan yuk tashuvchi samolyotni misol qilib olishingiz mumkin. 5.5a-rasmda yuk xaltasi muammosining kichik bir namunasi keltirilgan.

Bu masalani yechishda to'liq qidiruv usuli qo'llanilganda, berilgan $n$ ta buyumlar to'plamining barcha qism to'plamlari hosil qilinadi, har bir qism to'plamning umumiy og'irligi hisoblanib, mumkin bo'lgan qism to'plamlar (ya'ni umumiy og'irligi yuk xaltasi sig'imidan oshmaydigan qism to'plamlar) aniqlanadi va ular orasidan eng katta qiymatga ega bo'lgan qism to'plam topiladi. Misol uchun, 3.8a-rasmdagi holatning yechimi 5.5b-rasmda ko'rsatilgan. $n$-elementli to'plamning qism to'plamlari soni $2^n$ ga teng bo'lgani sababli, alohida qism to'plamlar qanchalik samarali yaratilmasin, to'liq qidiruv $\Omega(2^n)$ algoritmi bilan yakunlanadi.




(a)

| Qism-to'plam | Umumiy vazni | Umumiy qiymati |
|---|---|---|
| Ø | 0 | $0 |
| {1} | 7 | $42 |
| {2} | 3 | $12 |
| {3} | 4 | $40 |
| {4} | 5 | $25 |
| {1,2} | 10 | $54 |
| {1,3} | 11 | Sig'im katta |
| {1,4} | 12 | Sig'im katta |
| {2,3} | 7 | $52 |
| {2,4} | 8 | $37 |
| {3,4} | 9 | $65 |
| {1,2,3} | 14 | Sig'im katta |
| {1,2,4} | 15 | Sig'im katta |
| {1,3,4} | 16 | Sig'im katta |
| {2,3,4} | 12 | Sig'im katta |
| {1,2,3,4} | 19 | Sig'im katta |

(b)

**5.5-rasm.** (*a*) *Yuk xaltasi masalasining yechimi.* (*b*) *Uning to'liq qidiruv yo'li bilan yechilishi. Optimal tanlash haqidagi ma'lumotlar qalin harflar bilan yozilgan.*

Endi yuk xaltasi masalasini yechishda shoxlanish va chegaralash usulini qanday qo'llash mumkinligini ko'rib chiqamiz. Bu masala yuqorida keltirilganidek, $n$ ta buyumning og'irliklari $w_i$ va qiymatlari $v_i$ $(i = 1,2,\ldots, n)$ hamda sig'imi $W$ bo'lgan yuk xaltasi berilgan bo'lsa, yuk xaltaga sig'adigan buyumlarning eng qimmatli to'plamini toping. Berilgan masala buyumlarini ularning qiymat-og'irlik nisbati bo'yicha kamayib borish tartibida joylashtirish maqsadga muvofiq. Shunda birinchi buyum og'irlik birligiga nisbatan eng yuqori foyda beradi, oxirgisi esa eng past foyda keltiradi. Bunda teng qiymatli buyumlar ixtiyoriy tartibda joylashtiriladi:

$$\frac{v1}{} \geq \frac{2}{} \geq v$$

$$v_n \quad \cdots$$

$$\geq \_\_$$



$$w1 \quad w2 \qquad wn$$

Ushbu masala uchun holat-fazo daraxtini binar daraxt ko'rinishida tuzish tabiiy hisoblanadi (5.6-rasmga misol sifatida qarang). Daraxtning $i$-bosqichidagi har bir tugun $(0 \leq i \leq n)$ birinchi $i$ ta tartiblangan elementdan tuzilgan ma'lum bir tanlovni o'z ichiga olgan $n$ ta elementning barcha qism to'plamlarini ifodalaydi. Bu tanlov ildizdan tugungacha bo'lgan yo'l orqali aniq belgilanadi: chapga yo'nalgan shoxcha keyingi elementning kiritilganini, o'ngga yo'nalgan shoxcha esa uning kiritilmaganini ko'rsatadi. Tugunda ushbu tanlovning umumiy og'irligi $w$ va umumiy qiymati $v$, shuningdek, bu tanlovga nol yoki undan ortiq elementlarni qo'shish orqali olinishi mumkin bo'lgan har qanday qism to'plamning qiymatining yuqori chegarasi $ub$ qayd etiladi.

Yuqori chegara $(ub)$ ni hisoblashning oddiy usuli quyidagicha: $v$ ga, ya'ni allaqachon tanlangan buyumlarning umumiy qiymatiga, yuk xaltasining qolgan sig'imi $(W - w)$ va qolgan buyumlar orasidagi eng yaxshi tanlovning $(v_{i+1}/w_{i+1})$ ko'paytmasini qo'shish orqali amalga oshiriladi:

$$v^{i+1}$$

$$ub = v + (W - w)\,(\_\_\_). \qquad (5.1)$$

$$wi{+}1$$

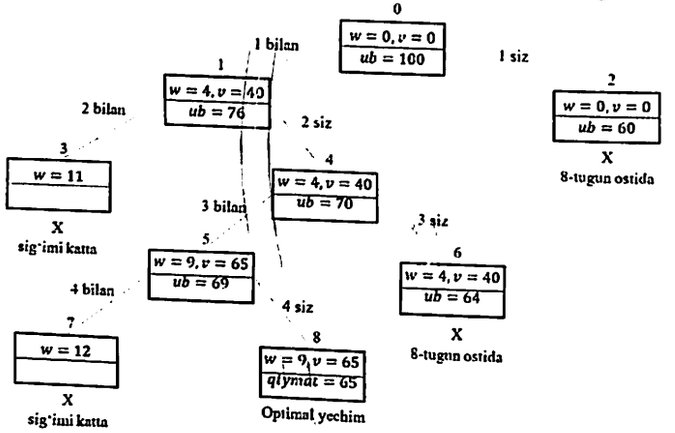
Xususiy misol tariqasida shoxlanish va chegaralash algoritmni yuqorida to'liq qidirish yo'li bilan yechilgan yuk xaltasi haqidagi masalaga tatbiq etamiz. Bunda buyumlarni ularning qiymat va og'irlik nisbatlarining kamayib borishi tartibida qayta joylashtirib chiqamiz.

| Buyum | Og'irligi | Qiymati | Qiymati/Og'irligi |
|---|---|---|---|
| 1 | 4 | $40 | 10 |
| 2 | 7 | $42 | 6 |
| 3 | 5 | $25 | 5 |
| 4 | 3 | $12 | 4 |

$W = 10$

Holat-fazo daraxtining ildizida (5.6-rasmga qarang) hali hech qanday elementlar tanlanmagan. Demak, tanlab olingan buyumlarning

umumiy og'irligi ham, ularning umumiy qiymati ham 0 ga teng. (5.1) formula bo'yicha hisoblangan yuqori chegaraning qiymati \$100 ga teng. 1-tugun, ildizning chap bolasi, 1-tugunni o'z ichiga olgan qism to'plamlarni ifodalaydi. Kiritilgan buyumlarning umumiy og'irligi va qiymati mos ravishda 4 va \$40 ni tashkil qiladi; yuqori chegaraning qiymati $40 + (10 - 4) * 6 = \$76$ ni tashkil qiladi. 2-tugun 1-tugunni o'z ichiga olmaydigan qism to'plamlarni ifodalaydi. Shunga ko'ra, $w = 0, v = \$0$ va $ub =$
$0 + (10 - 0) * 6 = \$60$. 1-tugun 2-tugunning yuqori chegarasidan kattaroq chegaraga ega bo'lganligi sababli, bu maksimallashtirish masalasi uchun istiqbolli bo'lgani uchun birinchi navbatda 1-tugunni shoxlaymiz. Uning bola tugunlari 3 va 4 tugunlar mos ravishda 1-tugun bilan va 2-tugun bilan va 2-tugunsiz qism to'plamlarni ifodalaydi. 3-tugun bilan ifodalangan har bir qism to'plamning umumiy og'irligi $w$ yuk xalta sig'imidan oshib ketganligi sababli, 3-tugun darhol tugatilishi mumkin. 4-tugunning ota-onasi bilan bir xil $w$ va $v$ qiymatlariga ega; $ub$ ning yuqori chegarasi $40 + (10 - 4) * 5 = \$70$ ga teng.




**5.6-rasm.** *Yuk xaltasi masalasi uchun eng yaxshi birinchi shoxlanish va chegaralash algoritmning holat fazosi daraxti.*

Keyingi tarmoqlanish uchun 4-tugunni 2-tugun ustidan tanlab (nima uchun?), 3-tugunni kiritish va chiqarib tashlash orqali mos ravishda 5- va 6-tugunlarni olamiz. Ushbu tugunlar uchun umumiy



og'irliklar va qiymatlar, shuningdek, yuqori chegaralar oldingi tugunlar uchun bo'lgani kabi hisoblanadi. 5-tugundan tarmoqlanish hech qanday mumkin bo'lgan yechimlarni ifodalamaydigan 7-tugunni va \$65 qiymatli faqat bitta $\{1, 3\}$ qism to'plamini ifodalovchi 8-tugunni beradi. Qolgan 2- va 6-tugunlar 8-tugun bilan ifodalangan yechim qiymatidan kichikroq yuqori chegaralangan qiymatlarga ega. Demak, ikkalasi ham tugatilib, 8-tugunning $\{1, 3\}$ qism to'plami masalaning *optimal yechimi* bo'lishi mumkin.

Yuk xaltasi masalasini shoxlanish va chegaralanish algoritm bilan yechish juda g'ayrioddiy xususiyatga ega. Odatda holat-fazo daraxtining ichki tugunlari masalaning izlash fazosining biror nuqtasini aniqlamaydi, chunki yechimning ba'zi komponentlari aniqlanmay qoladi. Yuk xaltasi masalasida esa daraxtning har bir tuguni berilgan elementlarning qism to'plamini ifodalaydi. Biz bu faktdan daraxtdagi har bir yangi tugunni yaratgandan so'ng hozirgacha ko'rilgan eng yaxshi qism to'plam haqidagi ma'lumotni yangilash uchun foydalanishimiz mumkin. Agar biz buni yuqorida o'rganilgan holat uchun qilganimizda, 8-tugun hosil bo'lishidan oldin 2 va 6-tugunlarni tugatishimiz mumkin edi, chunki ularning ikkalasi ham 5tugunning \$65 qiymatli qism to'plamidan pastroq.

## 5.4. Kommivoyajer masalasi

Kommivoyajer masalasi[16] o'zining sodda ko'rinishdagi ifodasi, muhim amaliy qo'llanilishi va boshqa kombinatorik masalalar bilan qiziqarli bog'liqligi
tufayli so'nggi 150 yil davomida tadqiqotchilarning e'tiborini tortib kelmoqda. Oddiy tilda aytganda, bu masala $n$ ta shahar berilgan bo'lib, ularning har biriga faqat bir marta tashrif buyurib, boshlang'ich shaharga qaytib kelgan holda eng qisqa yo'lni topishni talab qiladi. Bu masalani vaznli graf yordamida qulay tarzda modellash mumkin, bunda

[16] Traveling Salesman Problem (TSP) – Sayohatchi sotuvchi masalasining qo'yilishi: Bir sotuvchi bir nechta shaharlarga borib, har bir shaharga faqat bir marta kirib chiqishi, va oxir-oqibat dastlabki shaharga qaytishi kerak. Masalaning yechimi sifatida sotuvchi bosib o'tgan umumiy yo'l uzunligi eng qisqa (yoki sarf-xarajatlar eng kam) bo'ladigan optimal holatni aniqlash talab qilinadi.

$$lb = \lceil \_ \rceil \qquad (5.2)$$
$$2$$

Masalan, 5.8a-rasmdagi holat uchun (5.2) formulaning natijasi quyidagicha bo‘ladi:

$$lb = \lceil [1 + 3 + 3 + 6 + 1 + 2 + 3 + 4 + 2 + 3]/2 \rceil = 14.$$

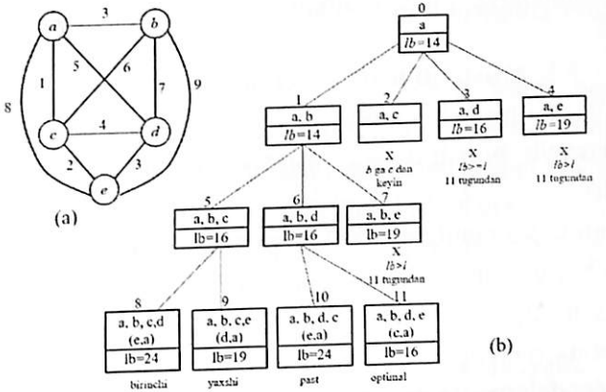
Bundan tashqari, berilgan grafning ma’lum yoylarini o‘z ichiga olishi shart bo‘lgan sayohat yo‘nalishlarining har qanday kichik to‘plami uchun (5.2) quyi chegarani tegishli ravishda o‘zgartirishimiz mumkin. Misol uchun, 5.8a-rasmdagi grafning $(a, d)$ yoyini o‘z ichiga olishi shart bo‘lgan barcha Gamilton sikllari uchun, $(a, d)$ va $(d, a)$ yoylarning majburiy kiritilishini hisobga olgan holda, har bir tugunga tutash eng qisqa ikkita yoyning uzunliklarini qo‘shib, quyidagi quyi chegarani hosil qilamiz:

$$\lceil [1 + 5 + 3 + 6 + 1 + 2 + 3 + 5 + 2 + 3]/2 \rceil = 16.$$

Endi (5.2) formula bilan berilgan chegaralovchi funksiyaga ega bo‘lgan “tarmoqlanish va chegaralasj” algoritmini qo‘llab, 5.8a-rasmdagi graf uchun eng qisqa Gamilton zanjirini topamiz. Birinchidan, umumiylikni yo‘qotmagan holda faqat $a$ dan boshlanadigan sayohatlarni ko‘rib chiqishimiz mumkin. Ikkinchidan, bizning graf yo‘naltirilmaganligi sababli, biz faqat $b$ ga $c$ dan oldin tashrif buyurilgan sayohatlarni yaratishimiz mumkin. Bundan tashqari, $n - 1 = 4$ ta shaharni ziyorat qilgandan so‘ng, sayohatchi tashrif buyurilmagan - qolgan shaharga tashrif buyurishdan va boshlang‘ich shaharga qaytishdan boshqa iloji yo‘q. Algoritmning qo‘llanilishini kuzatadigan holat-fazo daraxti 5.8b-rasmda keltirilgan. Avvalgi bo‘limning oxirida orqaga qaytishning kuchli va zaif tomonlari haqida aytgan izohlarimiz *shoxlar va chegaralar*ga ham tegishli. Asosiy fikrni yana bir bor eslatib o‘tamiz: ushbu holat-fazo daraxti usullari bizga murakkab kombinatorik masalalarning ko‘plab yirik misollarini yechish imkonini beradi. Biroq, odatda, qaysi holatlar real vaqt oralig‘ida hal bo‘lishini va qaysilari hal bo‘lmasligini oldindan aytib bo‘lmaydi.



O‘yin taxtasining simmetriyasi (masalan, shaxmat taxtasi) kabi qo‘shimcha ma’lumotlarni kiritish hal qilinadigan holatlar doirasini kengaytirishi mumkin. Shu yo‘nalishda, *shoxlar va chegaralar* algoritmi ba’zan oddiy bo‘lmagan mumkin bo‘lgan yechimning maqsad funksiyasi qiymatini bilish orqali tezlashtirilishi mumkin. Bu ma’lumot holat-fazoviy daraxtni ishlab chiqishni boshlashdan oldin ma’lumotlarning o‘ziga xos xususiyatlaridan foydalanish yoki hatto ba’zi muammolar uchun tasodifiy ravishda yaratilishi orqali olinishi mumkin. Keyin *shoxlanish va chegaralash* jarayoni bizni birinchi mumkin bo‘lgan yechimga olib kelishini kutib o‘tirmasdan, bunday yechimdan darhol eng yaxshi ko‘rilgan yechim sifatida foydalanishimiz mumkin.




**5.8-rasm**. *(a) Vaznli graf. (b) Ushbu grafda eng qisqa Gamilton siklini topish uchun shoxlanish va chegaralash algoritmining holat-fazosi daraxti. Daraxtning har bir tugunidagi graf tugunlari ro‘yxati tugun bilan ifodalangan Gamilton sikllarining boshlang‘ich qismini ko‘rsatadi.*

Orqaga qaytish [18] usulidan farqli o‘laroq, *shoxlanish va chegaralash* usuli bilan masalani yechishda tugunlarni yaratish tartibini tanlash va yaxshi chegaralash funksiyasini topish ham qiyinchilik, ham

[18] 18 Orqaga qaytish (Backtracking) - bu algoritmik yondashuv bo‘lib, murakkab kombinatorik masalalarni yechishda ishlatiladi. Bu usul yordamida yechimlar bo‘sh joyda birma-bir quriladi, va agar noto‘g‘ri yo‘l tanlansa, orqaga qaytib boshqa variantlar sinab chiqiladi. Asosiy g‘oyasi: “Agar tanlangan yo‘l muammo yechimiga olib bormasa, orqaga qaytib, boshqa yo‘lni sinab ko‘r!”

imkoniyat hisoblanadi. Yuqorida qo‘llagan eng yaxshi-birinchi qoidamiz mantiqli yondashuv bo‘lsa-da, u boshqa strategiyalarga qaraganda tezroq yechimga olib kelishi yoki kelmasligi mumkin. (Sun’iy intellekt tadqiqotchilari, ayniqsa, holat fazosi daraxtlarini rivojlantirishning turli strategiyalariga qiziqish bildiradilar.)

Yaxshi chegaralovchi funksiyani topish odatda oson ish emas. Bir tomondan, biz bu funksiyani hisoblash oson bo‘lishini istaymiz. Boshqa tomondan, u juda sodda bo‘lishi mumkin emas - aks holda, u o‘zining asosiy vazifasini imkon qadar tezroq holat-fazo daraxtining ko‘plab shoxlarini kesib tashlashni muvaffaqiyatsizlikka uchratgan bo‘lardi. Ushbu ikki raqobatdosh talab o‘rtasida to‘g‘ri muvozanatni o‘rnatish ko‘rib chiqilayotgan muammoning turli xil holatlari bilan intensiv tajriba o‘tkazishni talab qilishi mumkin.

### 5.5. Mustaqil ishlash uchun topshiriqlar

**1.** Eng yaxshi birinchi shoxlanish va chegaralash algoritmida faol tugunlarni kuzatib borish uchun qanday ma’lumotlar tuzilmasidan foydalaniladi?

**2.** Bo‘limda ko‘rsatilgan vazifalarni taqsimlash masalasining bir xil holatini, chegaralash funksiyasi bilan eng yaxshi birinchi shoxlanish va chegaralash algoritmi yordamida, satrlar emas, balki matritsa ustunlari asosida yeching.

**3.** ***a.*** Vazifalarni taqsimlash masalasi uchun shoxlanish va chegaralash algoritmning eng yaxshi holatiga misol keltiring.

***b.*** Eng yaxshi holatda, o‘zlashtirish masalasi uchun shoxlanish va chegaralash algoritmining holat-fazo daraxtida nechta tugun bo‘ladi?

**4.** Shoxlanish va chegaralash algoritmi yordamida o‘zlashtirish masalasini yechish dasturini yozing. Dastur yordamida tajriba o‘tkazish orqali ma’lum bir vaqt oralig‘ida (masalan, 1 daqiqada) yechim topiladigan xarajat matritsalarining o‘rtacha hajmini aniqlang.

**5.** Quyidagi ryukzak masalasini shoxlanish va chegaralash algoritmi yordamida yeching:



| buyum | vazni | qiymati |
|---|---|---|
| 1 | 10 | $100 |
| 2 | 7 | $63 |
| 3 | 8 | $56 |
| 4 | 4 | $12 |

$W = 16$

**6.** ***a.*** Ushbu bo‘limda qo‘llanilganiga nisbatan, ryukzak (yuk xaltasi) masalasini yechish uchun yanada murakkab chegaralash funksiyasini taklif qiling.

***b.*** 5-masala yechimi uchun qo‘llangan shoxlanish va chegaralash algoritmida o‘zingiz taklif qilgan yangi chegaralash funksiyasidan foydalanib masalani yeching.

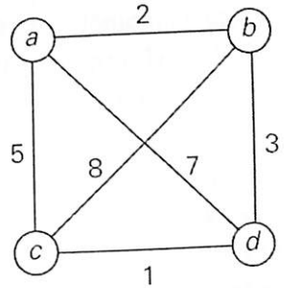
**7.** Shoxlanish va chegaralash algoritmi yordamida ryukzak (yuk xaltasi) masalasini yechish dasturini tuzing.

**8.** ***a.*** Shaharlararo masofalarning butun sonli qiymatlariga asoslangan simmetrik matritsalariga ega bo‘lgan kommivoyajer (ko‘chma savdogar) masalasi holat – daraxtlari uchun (12.2) formula bilan berilgan quyi chegaraning to‘g‘riligini isbotlang.

***b.*** Simmetrik bo‘lmagan masofa matritsalari uchun (12.2) quyi chegarani

qanday o‘zgartirish mumkin?

**9.** Quyida berilgan graf bo‘yicha kommivoyajer (ko‘chma savdogar) masalasini shoxlanish va chegaralash algoritmi yordamida yeching (bu masala 3.4bo‘limda ***to‘liq qidirish*** usuli bilan yechilgan edi):



**10.** Ilmiy tadqiqot loyihasi sifatida shaxmat, shashka va "krest-nol" kabi o'yinlarni dasturlashda holatlar dataxtlaridan qanday foydalanilishi haqida hisobot yozing. Bu tadqiqot loyihasini yozish uchun ikkita asosiy algoritmlar: minimaks algoritmi va alfa-beta kesish usullarini o'rganish zarur bo'ladi.



# 6-BOB. SARALASH ALGORITMLARI

## 6.1. Saralash algoritmlari: asosiy tushunchalar

**Saralash (tartiblash)** – bu berilgan ma'lumot elementlarining ba'zi bir xususiyatlariga ko'ra tartiblanishi (joylashtirilishi) hisoblanadi. Ko'p hollarda saralash mezoni (xusisiyati) sifatida aniq bir raqamli maydon qo'llaniladi va bu maydon kalit deb ataladi. Elementlarni kalit bo'yicha tartiblashda har bir keyingi elementning kaliti oldingisidan kichik bo'lsa kamayish tartibida, kalit maydon qiymati oldingisidan katta bo'lsa o'sish tartibida saralash deb ataladi.

**Saralashdan asosiy maqsad** - saralangan ma'lumotlarni qayta ishlash jarayonida zarur bo'ladigan elementni tez va oson qidirib topishni soddalashtirishdan iborat.

Saralash asosiy algoritmik dasturlash masalalaridan biri hisoblanadi. Saralash bilan bog'liq masalalarni yechish uchun ko'plab ilmiy izlanishlar olib borilgan va hozirgi kunda ko'plab algoritmlar ishlab chiqilgan.

Umuman olganda, saralash deganda, berilgan ob'ektlar to'plamini ma'lum bir mezon asosida qayta joylashtirish jarayoni tushunilishi kerak. Tartiblash dasturlashning barcha sohalarida: xoh u ma'lumotlar bazasi bo'lsin, xoh u matematik dasturlash sohasi bo'lsin istisnosiz qo'llaniladi.

Saralash algoritmi - bu ba'zi elementlar to'plamini tartibga solish algoritmi bo'ilb, odatda saralash algoritmi elementlar to'plamini o'sish yoki kamayish tartibida saralashni anglatadi.

Agar bir xil qiymatga ega elementlar mavjud bo'lsa, ular ketma-ket ixtiyoriy tartibda bir-birining yonida joylashtiriladi. Biroq, ba'zida bir xil qiymatlarga ega bo'lgan elementlarning kirish massividagi tartibini saqlash foydali hisoblanadi.

Saralash algoritmlarida ma'lumotlarning bir qismi saralash kaliti sifatida ishlatiladi. Saralash kaliti atribut (yoki bir nechta atributlar) deb ataladi, uning qiymati elementlarning tartibini belgilab beradi. Shunday qilib, massivlarni saralash algoritmlarini yozishda kalit ma'lumotlarga to'liq yoki qisman mos kelishini hisobga olish kerak.

Mavjud saralash algoritmlarini 3 qismga bo'lish mumkin:

- taqqoslash qismi - bir juft elementlarning tartibini aniqlash uchun taqqoslash;
- o‘rnini almashtirish qismi - bir juft elementlarni almashtiruvchi almashtirish;
- saralash algoritmining o'zi - u to'plamning barcha elementlari tartibga solinguncha elementlarni taqqoslaydi va qayta tartibga soladi.

Saralash algoritmlarini amaliyotda ko‘p uchratishimiz mumkin. Masalan, katta hajmdagi ma’lumotlarni qayta ishlash va saqlash jarayonida ularni ma’lum bir mezon asosida tartibga keltirish masalalarida.

Saralash algoritmlarining ishlash murakkabligini baholash

Boshqa hech qanday muammo saralash masalasi kabi turli xil yechimlarni keltirib chiqarmagan. Hozirda universal, eng yaxshi saralash algoritmi mavjud emas. Biroq, kiritilgan ma'lumotlarning taxminiy xususiyatlarini hisobga olgan holda, optimal ishlaydigan usulni tanlash mumkin. Buning uchun mavjud algoritmlarni baholash uchun zarur bo‘ladigan parametrlarni bilish zarur bo‘ladi.

Saralash vaqti – algoritmning tezkorligini baholaydigan asosiy parametr.

Xotira - asosiy parametrlardan biri bo'lib, bir qator saralash algoritmlari ma'lumotlarni vaqtincha saqlash uchun qo'shimcha xotira ajratishni talab qilishi bilan tavsiflanadi. Ishlatilgan xotirani baholashda dastlabki ma'lumotlar massivi egallagan joy va kirish ketma-ketligiga bog'liq bo'lmagan xotira sarfi, masalan, dastur kodini saqlash uchun xotira sarflari hisobga olinmaydi.

Barqarorlik (turg‘unlik) - bu saralash teng elementlarning nisbiy holatini o'zgartirmasligi uchun javobgar bo'lgan parametr.

Xulq-atvorning tabiiyligi - bu tartiblangan yoki qisman tartiblangan ma'lumotlarni qayta ishlashda tatbiq qilinadigan usulning samaradorligini ko'rsatadigan parametr. Algoritm kirish ketma-ketligining ushbu xususiyatini hisobga olsa va yaxshiroq ishlasa, o'zini tabiiy tutadi. Saralash algoritmlari klassifikatsiyasi.



Saralash algoritmlarining barcha xilma-xilligi va turli ko‘rinishga egaligi har xil mezonlarga ko'ra klassifikatsiya qilinishi mumkin, masalan, barqarorlik, xattiharakatlar, taqqoslash operatsiyalaridan foydalanish, qo'shimcha xotiraga bo'lgan ehtiyoj, qo‘llanilgan ma'lumotlar tuzilmasi haqidagi bilimga bo'lgan ehtiyoj, taqqoslash operatsiyasidan tashqariga chiqish va boshqalar.

Keling, saralash algoritmlarini qo'llash sohasi bo'yicha tasniflashni batafsil ko'rib chiqaylik. Bunday holatda saralashning asosiy turlari quyidagicha ajratib olinadi: **ichki saralash va tashqi saralash**. Ya’ni:

- ichki saralash algoritmlari – bu massivda saralash (tezkor xotirada);
- tashqi saralash algoritmlari – bu faylda saralash (tashqi xotirada).

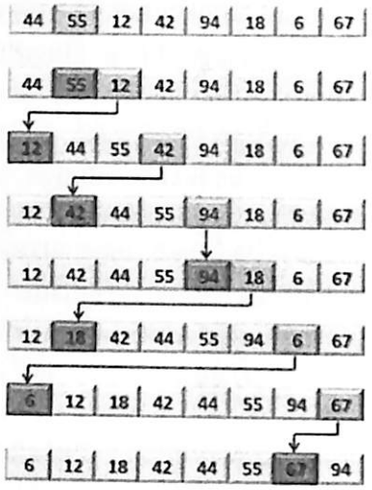
Shuni ta'kidlash kerakki, ichki saralash tashqi saralashdan ko'ra ancha samaralidir, chunki tezkor xotiraga kirish tashqi axborot tashuvchilarga qaraganda ancha kam vaqt talab etadi. Yaxshilangan (logarifmik) deb ataluvchi asosiy ichki saralash algoritmlarini ko'rib chiqamiz.

Ichki saralash usullarini 3 ta sinfga ajratish mumkin:

- qo‘yish orqali saralash;
- tanlash asosida saralash;
- almashtirish orqali saralash.

### 6.1.1. Qo‘yish orqali saralash

Algorotmning asosiy g‘oyasi: Massiv elementlari shartli ravishda oldindan tayyorlangan ketma-ketlik $a_1$, $a_2$, ..., $a_{i-1}$ va kiruvchi ketma-ketlik $a_i$, $a_{i+1}$, ..., $a_n$ kabi qismlarga ajratib olinadi. Oldindan tayyor ketma-ketlikda har bir $i$-element qulay joyga joylashtiriladi.




**6.1-rasm**. *Qo'yish orqali saralash sxemasi*

Ushbu algoritmning ishlashiga C++ dasturlash tilida misol. O'nta elementdan iborat butun sonli massiv berilgan. Massiv elementlarini o'sish tartibida saralang. Yechimi:

```
#include <iostream>
using namespace std;

void insertionSort(int arr[],
int n) {    for (int i = 1; i <
n; i++) {       int key =
arr[i];       int j = i - 1;

    // Elementlarni siljitish
while (j >= 0 && arr[j] > key)
{
        arr[j + 1] = arr[j];
        j-
-;      }
    arr[j + 1] = key;
  }
}
```



```
void printArray(int arr[],
int n) {    for (int i = 0; i <
n; i++) {       cout <<
arr[i] << " ";
  }
  cout << endl;
}

int main() {
  int arr[] = { 44, 55, 12, 42, 94, 18,
6, 67 };    int n = sizeof(arr) /
sizeof(arr[0]);    cout << "Dastlabki
massiv: ";
  printArray(arr, n);
insertionSort(arr, n);
cout << "Saralangan
massiv: ";
  printArray(arr, n);

return
0; }
```

Natija:

Dastlabki massiv: 44 55 12 42 94 18 6 67

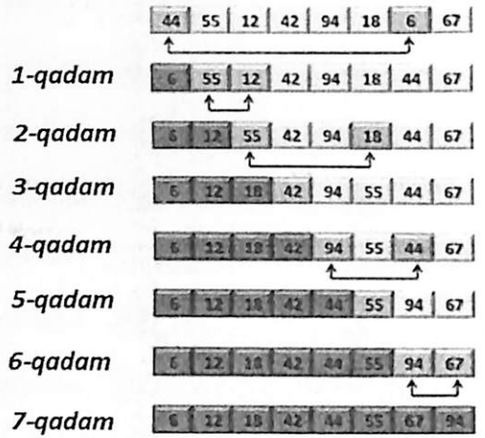
Saralangan massiv: 6 12 18 42 44 55 67 94

Algoritmning ishlash murakkabligi $O(n^2)$ ga teng. Shunday qilib, qo'yish orqali saralash usuli kompyuter uchun unchalik ham ma'qul emas, chunki bir nechta elementlar guruhini birdaniga surish samarali bo'lmaydi.

### 6.1.2. Tanlash asosida saralash usuli

To'g'ridan-to'g'ri tanlash usuli qandaydir ma'noda to'g'ridan-to'g'ri qo'yish usuliga ziddir. Bu yerda suriladigan elementlar faqat bitta

bo‘ladi va har bir surishdan keyin elementlarni taqqoslashlar soni bittaga kamayadi. Bu jarayon elementlar tugaguncha davom etadi.




**6.2-rasm**. To‘g‘ridan-to‘g‘ri tanlash orqali saralash

Bizga ***n*** ta element berilgan bo‘lsin. Biz shu elementlar ichidan eng kichigini topamiz va bu elementni massiv boshidagi element bilan almashtiramiz. Endi esa ikkinchi elementdan boshlab, ***n***-elementgacha bo‘lgan ***n***-1 ta elementdan eng kichigini topamiz. Topilgan minimumni qaralayotgan elementlarning boshiga ya’ni ikkinchi element o‘rniga qo‘yamiz. Ushbu jarayon massiv elementlari tugaguncha davom ettiriladi va natijada massiv elementlari o‘sish tartibida saralangan bo‘ladi.

To‘g‘ridan-to‘g‘ri tanlash orqali saralash usulining C++ dasturlash tilidagi algoritmi uchun misol. #include <iostream> using namespace std;

```
void selectionSort(int arr[],
int n) {    for (int i = 0; i < n
- 1; i++) {       int minIndex
= i;
      // Minimal elementni
qidirish       for (int j = i + 1;
j < n; j++) {          if (arr[j]
```



```
< arr[minIndex]) {
minIndex = j;
        }
      }
      // Minimal elementni
almashtirish       int temp =
arr[minIndex];
arr[minIndex] = arr[i];
arr[i] = temp;
   } } void printArray(int
arr[], int n) {    for (int i =
0; i < n; i++) {       cout
<< arr[i] << " ";
   }
   cout << endl;
} int main() {    int arr[] = { 64,
34, 25, 12, 22, 90, 11 };    int n =
sizeof(arr) / sizeof(arr[0]);    cout
<< "Dastlabki massiv: ";
   printArray(arr, n);
selectionSort(arr, n);
cout << "Saralangan
massiv: ";
   printArray(arr, n);
   return
0; }
```

Natija :

Dastlabki massiv: 64 34 25 12 22 90 11

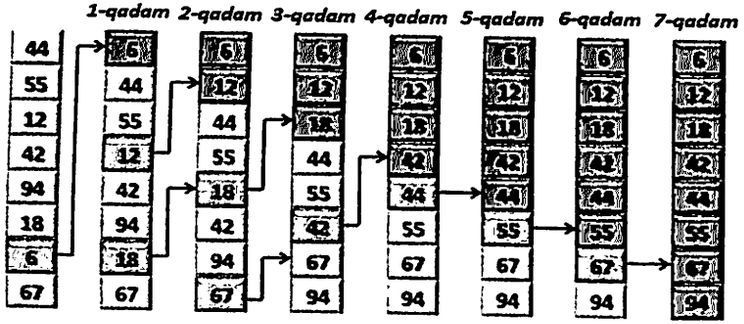
Saralangan massiv: 11 12 22 25 34 64 90

To‘g‘ridan-to‘g‘ri tanlash algoritmining ishlash murakkabligi $O(n^2)$ ga teng. Tanlash usuli qo‘yish usulidan ustunroq. Lekin agar kalitlar dastlab saralangan bo‘lsa, yoki qisman saralangan bo‘lsa to‘g‘ridan-to‘g‘ri qo‘yish usuli sal tezroq bo‘ladi.

### 6.1.3. Almashtirish usuli

To‘g‘ridan-to‘g‘ri almashtirish yoki pufaksimon saralash usuli – elementlar saralangunga qadar yonma-yon elementlarni saralashlar va almashtirishlar jarayoni.

Bu usulda xuddi to‘g‘ridan-to‘g‘ri saralash usuli kabi eng kichik element massiv boshiga ko‘chiriladi. Agar massivni vertikal ko‘rinishda deb tasavvur qilsak, minimal elementlarning tepaga ko‘chishi xuddi suvdagi pufakchalarning tepaga ko‘tarilishiga o‘xshaydi. Shuning uchun ham bu usulni “pufakchali usul” deyish mumkin.




**6.3-rasm.** To‘g‘ridan-to‘g‘ri almashtirish usuli

“Pufakcha” usulida saralashning C++ dastrulash tilidagi algoritmiga misol

```
#include <iostream> using
namespace std; void
bubbleSort(int arr[], int n) {
for (int i = 0; i < n - 1; i++) {
for (int j = 0; j < n - i - 1; j++)
{          if (arr[j] > arr[j + 1])
{             // Elementlarni
almashtirish
           int temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
        }
    }
```



```
  } } void printArray(int
arr[], int n) {    for (int i =
0; i < n; i++) {       cout
<< arr[i] << " ";
  }
  cout << endl;
}      int
main() {
  int arr[] = { 44, 55,12, 42, 94, 18, 6, 67 };
  int    n    =    sizeof(arr)    /
sizeof(arr[0]);   cout << "Berilgan
massiv: ";
  printArray(arr, n);
bubbleSort(arr, n);    cout
<< "Saralangan massiv: ";
  printArray(arr, n);

return
0; }
```

Natija:

Berilgan massiv: 44 55 12 42 94 18 6 67

Saralangan massiv: 6 12 18 42 44 55 67 94

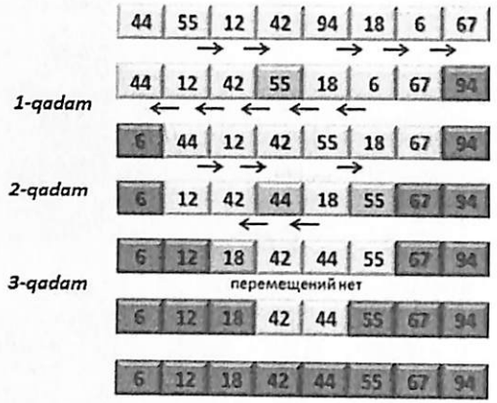
Algoritm ishlash murakkabligi $O(n^2)$ ga teng. «Pufakcha usulida saralash» samaradorligi bo‘yicha to‘g‘ridan-to‘g‘ri tanlash va to‘g‘ridan-to‘g‘ri qo‘yish usullari o‘rtasida desa ham bo‘ladi.

## 6.2. Saralashning yaxshilangan usullari

### 6.2.1. Sheyker saralash algoritmi

Sheyker saralash[19] usuli pufaksimon saralashning mukammallashgan usulidir. Pufakchali saralashda eng maksimal element massiv oxiriga boradi. Elementlarni bir marta to‘liq ko‘rib chiqganda elementlarning oxiridagisi saralangan bo‘ladi. Shuning uchun massivni bir marta qarab chiqqanimizdan keyin uni to‘liq tekshirmasdan $n-1$ elementigacha ko‘rib chiqish kifoya. Ushbu jarayon elementlar

tugaguncha davom etadi. Sheyker – saralash algoritmini misolda ko‘rib chiqamiz. Ketma-ketlik berilgan.




6.4-rasm. Sheyker saralash algoritmi

Sheyker-saralash algoritmining dasturi

Har bir while() siklining takrorlanishi saralash qadamini bildiradi.

---

[19] 1950–1960-yillar atrofida kompyuter fanining ilk davrlarida, dasturchilar Bubble sortning sekin ishlashini kamaytirish yo‘llarini izlashgan. Shunday optimallashtirishlardan biri - ikki yo‘nalishda yuruvchi pufakcha saralash bo‘lib, bu keyinchalik "Cocktail sort" deb nom oldi.

```
#include <iostream> using
namespace std; void
shakerSort(int arr[], int size)
{    bool swapped = true;
int start = 0;    int end = size
- 1;    while (swapped) {
swapped = false;       //
Chapdan o‘ngga yurish
for (int i = start; i < end; i++)
{          if (arr[i] > arr[i + 1])
{              swap(arr[i], arr[i
+ 1]);
```



```
        swapped = true;
      }
    }
    // Agar almashish bo‘lmagan bo‘lsa, massiv
saralangan       if (!swapped)          break;
    // O‘ngdan chapga yurish uchun tayyorgarlik
    swapped = false;
    end--;
    // O‘ngdan chapga yurish
for (int i = end - 1; i >= start; i--)
{          if (arr[i] > arr[i + 1]) {
swap(arr[i], arr[i + 1]);
swapped = true;
      }
}
start++;
  } } void printArray(int
arr[], int n) {    for (int i =
0; i < n; i++) {       cout
<< arr[i] << " ";
  }
  cout << endl;
}

int main() {    int arr[] = { 20, 10,
30, 40, 50, 25, 15 };    int size =
sizeof(arr) / sizeof(arr[0]);    cout
<< "Saralanmagan massiv: ";
  printArray(arr, size);
shakerSort(arr, size);    cout
<< "Saralangan massiv:  ";
printArray(arr, size);

return
```

0; }

Natija:

Saralanmagan massiv: 20 10 30 40 50 25 15

Saralangan massiv: 10 15 20 25 30 40 50

### 6.2.2. Shell saralash algoritmi

Shell saralash [19] usuli qo'yish usuli yordamida saralashning mukammallashtirilgan usuli hisoblanadi. Bu usulning asosiy g'oyasi, saralashning dastlabki bosqichidayoq bir-biridan yetarli uzoqda joylashgan elementlar juftligini taqqoslashdan iborat. Saralash usulining ushbu modifikatsiyasi uzoqdagi tartibsiz qiymatlar juftligini tez almashtirish imkonini beradi (bunday juftlarni saralash, odatda, faqat qo'shni elementlar solishtirilsa, ko'p miqdordagi almashtirishlarni talab qiladi).

Shell saralash usuli algoritmi. *N* ta elementdan iborat sonli massiv berilgan bo'lsin.

1-qadam. $1 < i < n/2$ shart uchun $n/2$ ta $(x_i, xn_{+i})$ juftliklarni tartiblash.

2

2-qadam. $1 < i < n/4$ shart uchun to'rtta elementdan iborat $n/4$ ta

$(x_i, xn4_{+i}, xn2_{+i}, x34n_{+i})$ elementlarni tartiblash.

3-qadam. Oldingi qadamda $n/4$ uchun tartiblangan guruhlardan 8 ta elementdan iborat yangi guruhni tartiblash va h.k.

Oxirgi qadamda birdaniga to'liq massiv $x_1, x_2, \ldots, x_n$ elementlari saralanadi.

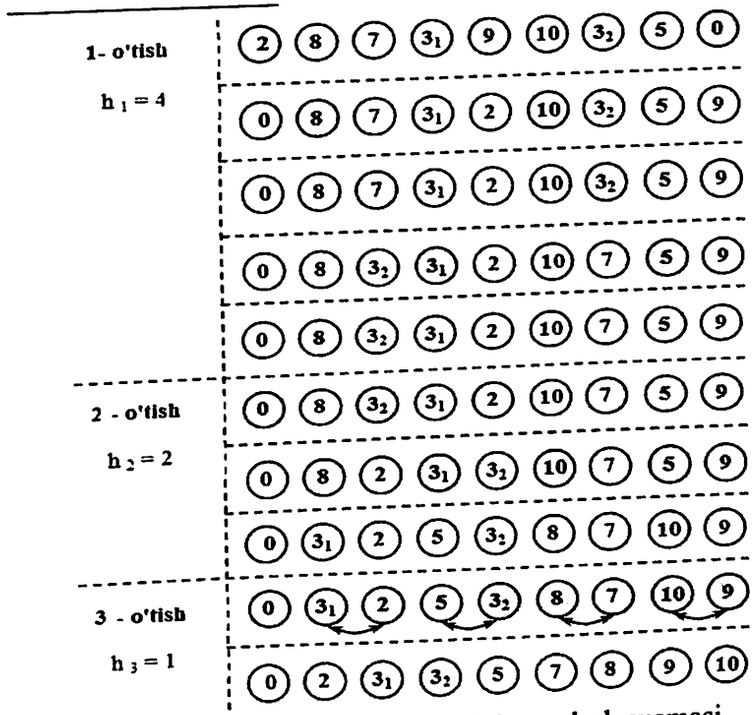
Har bir qadamda kichik guruhlarni saralash uchun qo'yish bilan saralash usuli qo'llaniladi. (6.5-rasm). // Shell sort algoritmi

```
void shellSort(int arr[], int size) {
// masofa h ning  boshlang'ich
qiymati     for (int h = size / 2; h >
0; h /= 2) {
```

[19] Bu saralash usuli 1959-yilda Donal'd Shell tomonidan taklif etilganligi uchun Shell nomini olgan.



```
    // h bo'yicha
almashish       for (int i =
h; i < size; i++) {
int temp = arr[i];
        int j = i;
        // Elementlarni joylashtirish
while (j >= h && arr[j - h] > temp) {
arr[j] = arr[j - h];
            j -= h;
        }
        arr[j] = temp;
    }
  }
}
```




6.5-rasm. Shell usulida saralash sxemasi

0; }

Natija:

Saralanmagan massiv: 20 10 30 40 50 25 15

Saralangan massiv: 10 15 20 25 30 40 50

### 6.2.2. Shell saralash algoritmi

Shell saralash [19] usuli qo‘yish usuli yordamida saralashning mukammallashtirilgan usuli hisoblanadi. Bu usulning asosiy g‘oyasi, saralashning dastlabki bosqichidayoq bir-biridan yetarli uzoqda joylashgan elementlar juftligini taqqoslashdan iborat. Saralash usulining ushbu modifikatsiyasi uzoqdagi tartibsiz qiymatlar juftligini tez almashtirish imkonini beradi (bunday juftlarni saralash, odatda, faqat qo'shni elementlar solishtirilsa, ko'p miqdordagi almashtirishlarni talab qiladi).

Shell saralash usuli algoritmi. $N$ ta elementdan iborat sonli massiv berilgan bo‘lsin.

1-qadam. $1 < i < n/2$ shart uchun $n/2$ ta $(x_i, xn_{+i})$ juftliklarni tartiblash.

2

2-qadam. $1 < i < n/4$ shart uchun to‘rtta elementdan iborat $n/4$ ta

$(x_i, xn4_{+i}, xn2_{+i}, x34n_{+i})$ elementlarni tartiblash.

3-qadam. Oldingi qadamda $n/4$ uchun tartiblangan guruhlardan 8 ta elementdan iborat yangi guruhni tartiblash va h.k.

Oxirgi qadamda birdaniga to‘liq massiv $x_1, x_2, \ldots, x_n$ elementlari saralanadi.

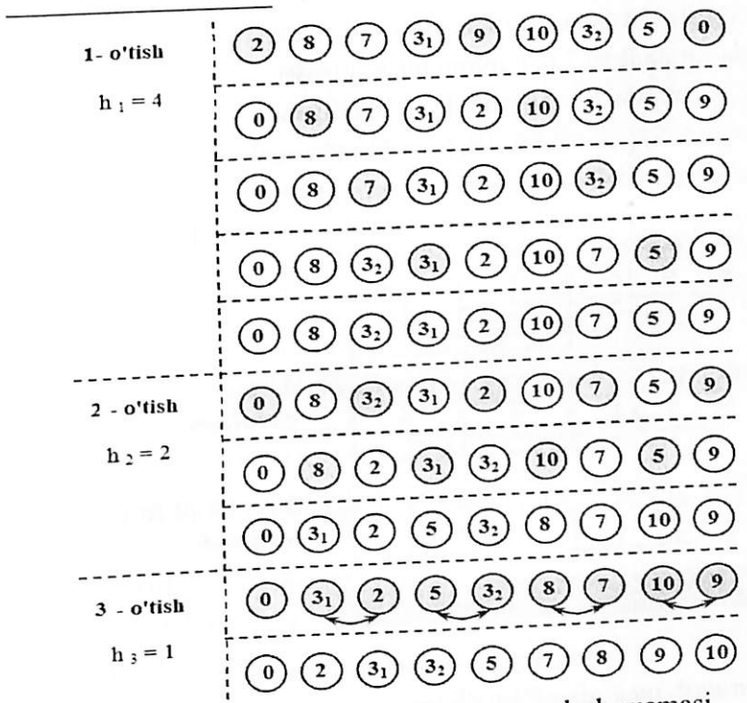
Har bir qadamda kichik guruhlarni saralash uchun qo‘yish bilan saralash usuli qo‘llaniladi. (6.5-rasm). // Shell sort algoritmi

```
void shellSort(int arr[], int size) {
// masofa h ning  boshlang'ich
qiymati    for (int h = size / 2; h >
0; h /= 2) {
```

[19] Bu saralash usuli 1959-yilda Donal’d Shell tomonidan taklif etilganligi uchun Shell nomini olgan.



```
    // h bo'yicha
almashish       for (int i =
h; i < size; i++) {
int temp = arr[i];
        int j = i;
        // Elementlarni joylashtirish
while (j >= h && arr[j - h] > temp) {
arr[j] = arr[j - h];
            j -= h;
        }
        arr[j] = temp;
    }
  }
}
```




6.5-rasm. Shell usulida saralash sxemasi

Donald L. Shell tomonidan taklif qilingan yuqoridagi usul o'z o'rnida *beqaror* saralash usullari qatoriga kiradi. Ushbu algoritmning samaradorligi siljitiiladigan elementlar tezroq o‘z o‘rnini egallashi bilan tavsiflanadi. Shell saralash algoritmining murakkabligi:

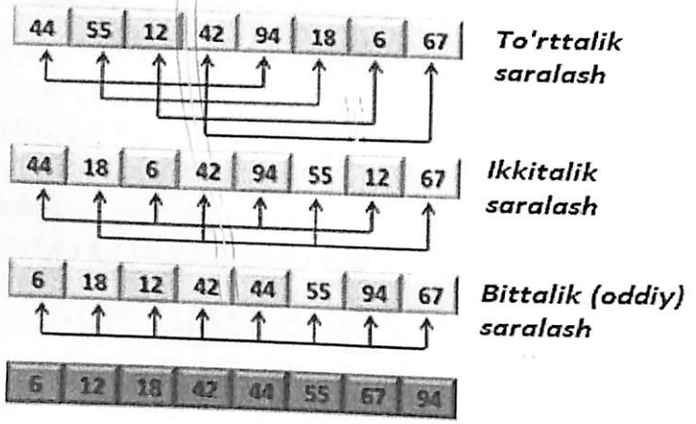
Vaqt murakkabligi - $O(n^2)$ eng yomon holat uchun (agar $h$ noto‘g‘ri tanlansa); $O(nlog^2n)$ yoki $O(n^{1.5})$ o‘rtacha holat uchun (agar $h$ optimallashtirilgan bo‘lsa), $O(n)$ eng yaxshi holat uchun (agar massiv saralangan bo‘lsa).

Xotira murakkabligi - $O(1)$ chunki Shell Sort **in-place** ishlaydi va faqat **doimiy xotira** talab qiladi.

Har bir to‘liq qarab chiqishda barcha elementlarni ishga tushirish mashinadan katta resurs talab qiladigandek ko‘rinadi, lekin har bir to‘liq qarab chiqishda kam elementlar saralanadi yoki yaxshi saralangan bo‘lsa, ularni shunchaki taqqoslab chiqish yetarli bo‘ladi.

Bu usul natijasida massiv saralangan bo‘ladi va har bir to‘liq qarab chiqish avvalgisiga nisbatan yaxshiroq bo‘ladi. (chunki har bir $i$-saralash $2i$-saralashda saralangan 2 ta guruhni birlashtiradi).

Shell saralash algoritmi qo‘llanilgan yana bir misol. $A[8] = \{44, 55,12, 42,94, 18, 6, 67\}$ butun sonli massivni Shell saralash usulidan foydalanib saralash dasturini tuzing.




6.6-rasm. Saralash sxemasi

Shell saralash usulining C++ tilidagi tadbiqi:



```
#include <iostream> using
namespace std; // Shell sort
algoritmi void shellSort(int arr[],
int size) {    // masofa h ning
boshlang'ich qiymati    for (int h
= size / 2; h > 0; h /= 2) {
      // h bo'yicha
almashish        for (int i =
h; i < size; i++) {
int temp = arr[i];
          int j = i;
          // Elementlarni joylashtirish
while (j >= h && arr[j - h] > temp) {
arr[j] = arr[j - h];
             j -= h;
          }
          arr[j] = temp;
      }
   }
}
//Massivni chiqarish void
printArray(int arr[], int n)
{    for (int i = 0; i < n;
i++) {
      cout << arr[i] << " ";
   }
   cout << endl;
} int main() {    int arr[] = { 44,
55,12, 42, 94, 18, 6, 67 };    int size
= sizeof(arr) / sizeof(arr[0]);    cout
<< "Saralanmagan massiv: ";
   printArray(arr,          size);
shellSort(arr, size);
   cout << "Saralangan massiv:   ";
```

Donald L. Shell tomonidan taklif qilingan yuqoridagi usul o'z o'rnida *beqaror* saralash usullari qatoriga kiradi. Ushbu algoritmning samaradorligi siljitiiladigan elementlar tezroq o‘z o‘rnini egallashi bilan tavsiflanadi. Shell saralash algoritmining murakkabligi:

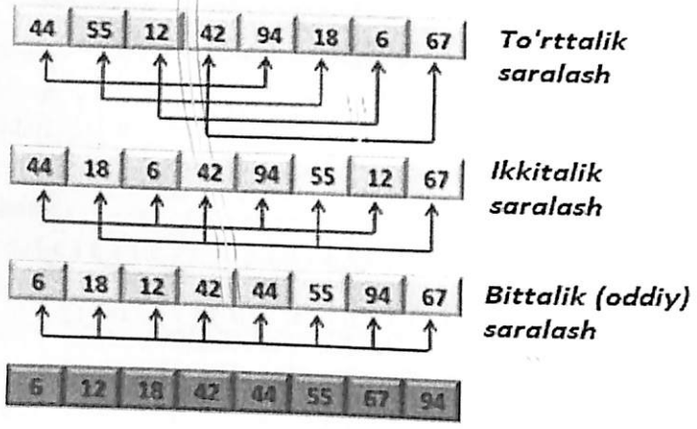
Vaqt murakkabligi - $O(n^2)$ eng yomon holat uchun (agar $h$ noto‘g‘ri tanlansa); $O(nlog^2n)$ yoki $O(n^{1.5})$ o‘rtacha holat uchun (agar $h$ optimallashtirilgan bo‘lsa), $O(n)$ eng yaxshi holat uchun (agar massiv saralangan bo‘lsa).

Xotira murakkabligi - $O(1)$ chunki Shell Sort **in-place** ishlaydi va faqat **doimiy xotira** talab qiladi.

Har bir to‘liq qarab chiqishda barcha elementlarni ishga tushirish mashinadan katta resurs talab qiladigandek ko‘rinadi, lekin har bir to‘liq qarab chiqishda kam elementlar saralanadi yoki yaxshi saralangan bo‘lsa, ularni shunchaki taqqoslab chiqish yetarli bo‘ladi.

Bu usul natijasida massiv saralangan bo‘ladi va har bir to‘liq qarab chiqish avvalgisiga nisbatan yaxshiroq bo‘ladi. (chunki har bir $i$-saralash $2i$-saralashda saralangan 2 ta guruhni birlashtiradi).

Shell saralash algoritmi qo‘llanilgan yana bir misol. $A[8] = \{44, 55,12, 42,94, 18, 6, 67\}$ butun sonli massivni Shell saralash usulidan foydalanib saralash dasturini tuzing.




6.6-rasm. Saralash sxemasi

Shell saralash usulining C++ tilidagi tadbiqi:



```
#include <iostream> using
namespace std; // Shell sort
algoritmi void shellSort(int arr[],
int size) {    // masofa h ning
boshlang'ich qiymati    for (int h
= size / 2; h > 0; h /= 2) {
      // h bo'yicha
almashish        for (int i =
h; i < size; i++) {
int temp = arr[i];
          int j = i;
          // Elementlarni joylashtirish
while (j >= h && arr[j - h] > temp) {
arr[j] = arr[j - h];
             j -= h;
          }
          arr[j] = temp;
      }
   }
}
//Massivni chiqarish void
printArray(int arr[], int n)
{    for (int i = 0; i < n;
i++) {
      cout << arr[i] << " ";
   }
   cout << endl;
} int main() {    int arr[] = { 44,
55,12, 42, 94, 18, 6, 67 };    int size
= sizeof(arr) / sizeof(arr[0]);    cout
<< "Saralanmagan massiv: ";
   printArray(arr,          size);
shellSort(arr, size);
   cout << "Saralangan massiv:   ";
```

```
printArray(arr, size);

return
0; }
```

Natija:

Saralanmagan massiv: 44 55 12 42 94 18 6 67

Saralangan massiv: 6 12 18 42 44 55 67 94

### 6.2.3. Daraxt yordamida saralash

Daraxt yordamida saralash asosini binar qidiruv daraxti tashkil etadi. Binar (ikkilik) qidiruv daraxti – quyidagi qo'shimcha shartlar bajariladigan binary daraxt hisoblanadi. Qidiruv daraxti xususiyatlari:

Ikkala shoxi ham – chap va o'ng ikkilik qidiruv daraxti hisoblanadi.

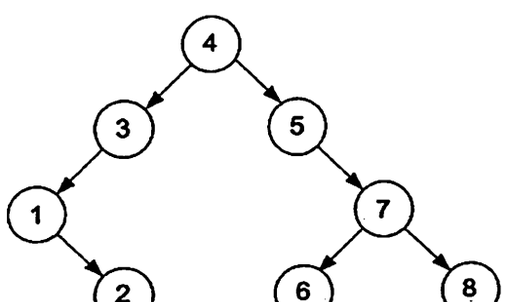
Istalgan chap shox kaliti o'zi chiqqan daraxtning kalitidan kichik.

Istalgan o'ng shox kaliti o'zi chiqqan daraxtning kalitidan kichik emas.

Daraxt yordamida saralash uchun berilgan ketma-ketlik daraxt ko'rinishidagi ma'lumotlar strukturasi bo'lishi kerak. Masalan, berilgan ketma-ketlik quyidagi ko'rinishga ega: { 4, 3, 5, 1, 7, 8, 6, 2 }.

Daraxrt ildizi ketma-ketlikning boshlang'ich elementi hisoblanadi. Qolgan elementlar ildizdan kichik bo'lsa chap shoxga joylashtiriladi, kattalari esa o'ng shoxga joylashtiriladi. Holbuki, bu shart barcha shoxlarda bajarilishi shart. Shundan keyin daraxt tuzilishiga joylashtirilgan barcha elementlarni infiks ko'rinishli aylanib chiqish yordamida chiqaramiz.






6.7-rasm. Binar qidiruv daraxtiga misol

Daraxt yordamida saralashning C++ dagi dasturi

```
#include <iostream> using
namespace std; struct tnode {
int field;          // ma'lumotlar
maydoni   struct tnode *left; //
chap avlod   struct tnode *right;
// o'ng avlod };
// daraxt tarmoqlarini chiqarish (infiks ko'rinishli aylanib
chiqish ) void treeprint(tnode *tree) {  if (tree != NULL) {
//toki bo'sh tarmoq uchramaguncha    treeprint(tree->left);
//chap shoxning rekursiv chiqarish funksiyasi    cout <<
tree->field << " "; //daraxt ildizini chiqaramiz
  treeprint(tree->right); // o'ng shoxning rekursiv chiqarish
funksiyasi
 } }
// daraxtga tarmoqlar qo'shish struct tnode * addnode(int x, tnode
*tree) {  if (tree == NULL)  { //agar daraxt mavjud bo'lmasa,
ildizni shakllantiramiz    tree = new tnode; // tarmoq osti xotira
tree->field = x;  //ma'lumotlar maydoni    tree->left = NULL;
  tree->right = NULL; //shoxlarni bo'sh yaratamiz
 }
 else    // aks holda
```

```
   if (x < tree->field)   //agar x element ildizdan kichik bo‘lsa
chapga ketamiz      tree->left = addnode(x, tree->left);
//elementni rekursiv qo‘shamiz     else  //aks holda o‘ngga
ketamiz
     tree->right = addnode(x, tree->right); // elementni rekursiv
qo‘shamiz     return(tree); }
//daraxt xotirasini bo‘shatish void freemem(tnode
*tree) {   if (tree != NULL)  {  // agar daraxt bo‘sh
bo‘lmasa     freemem(tree->left);   // chap shoxni
rekursiv o‘chiramiz     freemem(tree->right);  //
o‘ng shoxni rekursiv o‘chiramiz     delete tree;
// ildizni o‘chiramiz
  }   } int
main() {
 struct tnode *root = 0;    // Daraxt tuzilishini e'lon qilamiz
 int a;                  // joriy tarmoq
qiymati   for (int i = 0; i< 8; i++)
{
   cout << i + 1 << " - tugunni kiriting "<<
": ";     cin >> a;
   root  =  addnode(a,  root);  // daraxtga  kiritilgan  tarmoqni
joylashtiramiz
  }
cout<<"Saralangan massiv: "<<endl;
 treeprint(root);      freemem(root);     //
ajratilgan xotirani o‘chiramiz   return 0; }
```

Natija:

1-holatda:

```
1 - tugunni kiriting : 4
2 - tugunni kiriting : 3
3 - tugunni kiriting : 5
4 - tugunni kiriting : 1
5 - tugunni kiriting : 7
6 - tugunni kiriting : 8
7 - tugunni kiriting : 6
8 - tugunni kiriting : 2 Saralangan massiv: 1 2 3 4 5 6 7 8 2-holatda:
1 - tugunni kiriting : 9
2 - tugunni kiriting : 5
3 - tugunni kiriting : 7
4 - tugunni kiriting : 6
5 - tugunni kiriting : 3
6 - tugunni kiriting : 4
7 - tugunni kiriting : 8
8 - tugunni kiriting : 1 Saralangan massiv:
1 3 4 5 6 7 8 9
```

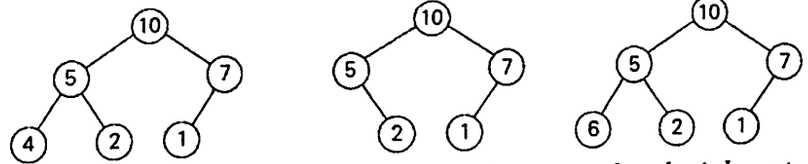


## 6.3. Piramidali saralash

### 6.3.1. Binar uyum

"Uyum" (Heap) deb ataladigan ma'lumotlar tuzilmasi, oddiy lug‘atdagi so‘z ta'rifida ko‘rsatilishi mumkin bo‘lgan tartibsiz elementlar uyumi emas. Aksincha, u ustuvor navbatlarni amalga oshirish uchun eng qulay bo‘lgan, qisman tartiblangan ma'lumotlar tuzilmasidir. Eslatib o‘tamiz, ustuvor navbat - bu elementning ustuvorligi deb ataladigan tartiblanadigan xususiyatga ega bo‘lgan elementlar to‘plamidir. U quyidagi amallarni bajarish imkonini beradi:

- eng yuqori (ya'ni eng katta) ustuvorlikka ega bo‘lgan elementni topish; - eng muhim ustuvorlikka ega bo‘lgan elementni o‘chirish; - to‘plamga yangi element qo‘shish.




**6.8-rasm.** *Uyum tuzilmasiga misollar: faqat eng chapdagi daraxt uyum hisoblanadi.*

Yuqorida sanab o'tilgan uyumlar ustida bajariladigan amallarning samarali tashkil etilishi muhim ahamiyatga ega va tuzilmani amaliyotda qo'llash uchun foydali qiladi. Kompyuter operatsion tizimlari

tomonidan vazifalarni bajarishni rejalashtirish va aloqa tarmoqlari tomonidan trafikni boshqarish kabi ilovalarda ustuvor navbatlar tabiiy ravishda paydo bo'ladi. Ular bir qator muhim algoritmlarda ham qo'llaniladi, masalan, Prim algoritmi, Deykstra algoritmi va shoxlanish va chegaralash usuli. Uyum, shuningdek, nazariy jihatdan muhim bo'lgan HeapSort yoki Piramidali saralash deb ataladigan saralash algoritmining asosiy ma'lumotlar tuzilmasi hisoblanadi. Biz ushbu algoritmni uyumni aniqlab, uning asosiy xususiyatlarini o'rganib chiqqanimizdan so'ng muhokama qilamiz.

**Ta'rif:** Uyumni quyidagi ikkita shart bajarilganda har bir tugunga bittadan kalit biriktirilgan binar daraxt sifatida aniqlash mumkin:

1. Shakl xususiyati - binar daraxt mohiyatan to'liq (yoki shunchaki to'liq), ya'ni uning barcha darajalari to'liq bo'lishi kerak (faqat oxirgi daraja bundan mustasno, bu yerda faqat ba'zi o'ng qism daraxtning chetki barglari yetishmasligi mumkin).

2. Ota tugun ustunligi yoki uyum xossasi - har bir tugundagi kalit uning davomchilaridagi kalitlardan katta yoki teng bo'lishi kerak – max-heap. (Bu shart barcha barglar uchun avtomatik ravishda bajarilgan deb hisoblanadi.)[20]

Misol uchun, 6.8-rasmdagi daraxtlarni ko'rib chiqaylik. Birinchi daraxt - uyum (heap) bo'ladi. Ikkinchisi emas emas, chunki daraxtning shakl xususiyati buzilgan. Uchinchisi ham uyum emas, chunki 5 kalitli tugun uchun ota-ona ustunligi qoidasi bajarilmagan.

E'tibor bering, uyumdagi kalit qiymatlari yuqoridan pastga qarab tartiblangan; ya'ni, ildizdan bargga boradigan har qanday yo'ldagi qiymatlar ketmaketligi kamayib boradi (agar teng kalitlarga ruxsat berilsa, kamaymasligi mumkin). Biroq, kalit qiymatlarida chapdan o'ngga tartib mavjud emas; ya'ni daraxtning bir xil darajasidagi yoki umuman olganda, bir xil tugunning chap va o'ng qism daraxtlaridagi tugunlarning kalit qiymatlari o'rtasida hech qanday bog'liqlik yo'q.

[20] Ba'zi mualliflar har bir tugundagi kalit uning davomchilaridagi kalitlardan kichik yoki teng bo'lishini talab qilishadi. Bunday uyum tuzilmasi min-heap deb ataladi.



Quyida uyumlarning isbotlash qiyin bo'lmagan muhim xossalari ro'yxati keltirilgan (bu xossalarni 6.9-rasmdagi uyum misolida tekshiring).

1. Aslida $n$ ta tugunga ega bo'lgan bittagina to'liq binar daraxt mavjud. Uning balandligi $\lfloor \log_2 n \rfloor$ ga teng bo'ladi.

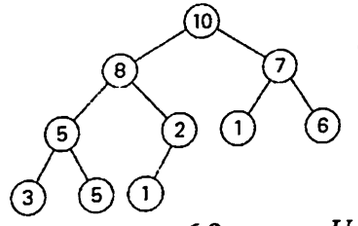
2. Uyumning ildizi har doim uning eng katta elementini o'z ichiga oladi.

3. Uyumning barcha avlodlari bilan birgalikda qaralgan tuguni ham uyumdir.

4. Uyumni massiv ko'rinishida ifodalash uchun uning elementlarini yuqoridan pastga, chapdan o'ngga ko'rinishda yozish orqali amalga oshirish mumkin. Uyum elementlarini bunday massivning 1 dan n gacha bo'lgan indekslarida saqlash qulay bo'lib, bunda H[0] ishlatilmaydi yoki u yerga qiymati uyumdagi har bir elementdan katta bo'lgan ***nazoratchini*** qo'yish mumkin. Ushbu tasavvur bilan,

*a.* ota tugun kalitlari massivning birinchi $\lfloor n/2 \rfloor$ indekslarida, barg kalitlari esa
oxirgi $\lceil n/2 \rceil$ indekslarida joylashadi;

*b.* massivning ota indeksidagi kalitning $i$ $(1 \le i \le \lfloor n/2 \rfloor)$ davomchilari $2i$ va
$2i + 1$ indekslarda bo'ladi va mos ravishda $i$ $(2 \le i \le n)$ indeksdagi kalitning ota tuguni $\lfloor 2i \rfloor$ indeksda bo'ladi.




massiv ko'rinishi

| indexlar | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| qiymatlar | | 10 | 8 | 7 | 5 | 2 | 1 | 6 | 3 | 5 | 1 |

Ota-ona tugunlar | Barg tugunlar

**6.9-rasm.** *Uyum va uning massiv ko'rinishi.*

Demak, uyumni H[1..n] massiv sifatida ham ta'riflashimiz mumkin ekan. Bunda massivning birinchi yarmidagi $i$-indeksidagi har bir element $2i$ va $2i + 1$ indekslardagi elementlardan katta yoki teng bo'ladi:

$H[i] \ge \max\{H[2i], H[2i+1]\}$, bu $i = 1, \ldots, n/2$ uchun o'rinli.

(Albatta, agar $2i + 1 > n$ bo'lsa, faqat $H[i] \geq H[2i]$ shartining bajarilishi kifoya.)

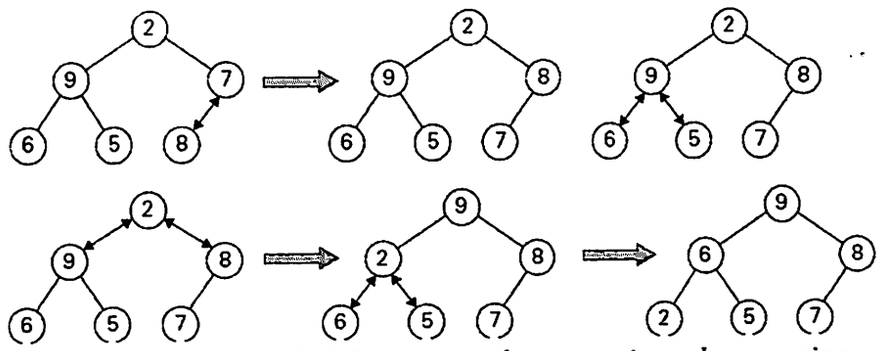
Uyumlar bilan ishlaydigan ko'pchilik algoritmlarning g'oyalarini tushunish uyumlarni binar daraxtlar sifatida tasvirlaganda osonroq bo'lsa-da, ularni amalda qo'llash odatda massivlar yordamida ancha sodda va samaraliroq bo'ladi.

### 6.3.2. To'dalash usuli (Heapify Method)

Berilgan kalitlar ro'yxati uchun uyumni qanday qurish mumkin? Buning uchun ikkita asosiy usul mavjud. Birinchisi 6.10-rasmda ko'rsatilgan ***pastdan yuqoriga uyum qurish*** (***bottom-up heap construction***) algoritmidir. U $n$ ta tugunli deyarli to'liq binar daraxtni kalitlarni berilgan tartibda joylashtirish orqali boshlang'ich holatga keltiradi va so'ngra daraxtni quyidagicha "uyumlashtiradi" (heapifies):

- oxirgi ota tugundan boshlab, algoritm ushbu tugundagi kalit uchun ota ustunligi mavjudligini tekshiradi.
- agar bu shart bajarilmasa, algoritm tugunning $K$ kalitini uning davomchilarining kattaroq kaliti bilan almashtiradi va $K$ ning yangi joylashuvida ota ustunligi saqlanishini tekshiradi.
- bu jarayon $K$ uchun ota ustunligi ta'minlanguncha davom etadi (chunki u bargdagi har qanday kalit uchun avtomatik ravishda o'rinli bo'lishi kerak.)
- joriy ota tugundagi qism daraxtning "uyumlashtirilishi" tugagach, algoritm tugunning bevosita ajdodi uchun ham xuddi shunday ishni bajaradi.
- algoritm bu jarayon daraxt ildizi uchun bajarilgandan so'ng to'xtaydi.





**6.10-rasm.** 2, 9, 7, 6, 5, 8 *ro'yxati uchun uyumni pastdan yuqoriga qurish. Ikki tomonlama strelkalar ota tugun ustunligini tasdiqlovchi asosiy taqqoslashlarni ko'rsatadi.*

**ALGORITM** *HeapBottomUp*($H[1..n]$)
//Berilgan massiv elementlaridan pastdan yuqoriga
//algoritmi orqali uyumni shakllantiradi
//Kirish: $H[1..n]$ tartiblanishi mumkin bo'lgan elementlar massivi
//Chiqish: $H[1. \ . \ n]$
uyumi **for** $i \leftarrow n/2$
**downto** 1 **do** $k \leftarrow i$; $v$
$\leftarrow H[k]$
$heap \leftarrow \boldsymbol{false}$
**while not** $heap$ **and** $2 * k \leq n$ **do**
$j \leftarrow 2 * k$
**if** $j < n$ //bu yerda ikkita avlod tugun bor
**if** $H[j] < H[j+1]$
$j \leftarrow j + 1$
**if** $v \geq H[j]$
$heap \leftarrow$
$\boldsymbol{true}$
**else** $H[k] \leftarrow H[j]$;
k←j
$H[k] \leftarrow v$

Bu algoritm eng yomon holatda qanchalik samarali? Soddalik uchun $n =$

$2^k - 1$ deb faraz qilaylik, shunda uyumning daraxti to‘liq bo‘ladi, ya’ni har bir darajada (daraxt bosqichida) mumkin bo‘lgan eng ko‘p tugunlar soni mavjud bo‘ladi. Daraxtning balandligi $h$ bo‘lsin. Bo‘lim boshidagi uyumlarning xususiyatlari ro‘yxatdagi birinchi xususiyatiga ko‘ra, $n$ ning aniq qiymatlari uchun $h = \lfloor \log_2 n \rfloor$ yoki $\lfloor \log_2(n+1) \rfloor - 1 = k - 1$ bo‘ladi. Daraxtning $i$-darajasidagi har bir kalit, uyum qurish algoritmining eng yomon holatida, $h$ barg darajasigacha yetib boradi. Keyingi quyi darajaga o‘tish uchun ikkita taqqoslash talab etiladi: biri kattaroq bola tugunni aniqlash uchun, ikkinchisi esa almashtirishning zarurligini belgilash uchun. Shu sababli, $i$-darajadagi kalitni o‘z ichiga olgan asosiy taqqoslashlarning umumiy soni $2(h - i)$ ga teng bo‘ladi. Shuning uchun asosiy taqqoslashlarning umumiy soni eng yomon holatda quyidagicha bo‘ladi:

$$C_{yomon}(n) = \sum_{i=0}^{h-1} \sum_{i-daraja\ kaliti} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)),$$

bu yerda oxirgi tenglikning to‘g‘riligini $\sum_{i=1}^{h} i2^i$ yig‘indi uchun yopiq formuladan foydalanib (A ilovaga qarang) yoki $h$ bo‘yicha matematik induksiya orqali isbotlash mumkin. Shunday qilib, ushbu pastdan yuqoriga yo‘naltirilgan algoritm yordamida $2n$ dan kam taqqoslashlar bilan $n$ o‘lchamli uyumni qurish mumkin.

### 6.3.3. Kiritish (Insert) va ajratib olish (Extract) usuli

*HeapBottomUp()* algoritmiga muqobil (va kamroq samarali) bo’lgan algoritm oldindan mavjud (ilgari qurilgan) uyumga yangi kalitni ketma-ket kiritish orqali uyumni quradi; ba’zilar bu algoritmni ***yuqoridan pastga qarab uyum qurish*** algoritmi deb atashadi. Xo‘sh, qanday qilib uyumga yangi $K$ kalitni kiritish (qo‘yish) mumkin?

1) Dastlab, mavjud uyumning oxirgi bargidan keyin kaliti $K$ bo‘lgan yangi tugunni biriktiring.

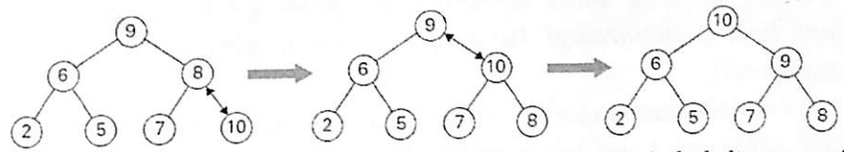
2) So‘ngra $K$ ni yangi uyumdagi tegishli joyigacha quyidagicha siljiting (ustivorligi mos o’ringacha):



$K$ ni uning asosiy kalit (ota tuguni) bilan taqqoslang: agar ota tugun kaliti $K$ dan katta yoki teng bo‘lsa, to‘xtang (uyum tuzilmasi hosil bo‘ldi); aks holda, bu ikki kalitni almashtiring va $K$ ni uning yangi asosiy kaliti bilan taqqoslang.

3) Bu almashtirish $K$ o‘zining oxirgi ota tugunidan katta bo‘lmaguncha yoki ildizga yetmaguncha davom etadi (6.11-rasmda tasvirlangan).

Shubhasiz, bu kiritish (qo‘shish) amali uyumning balandligidan ko‘proq kalit taqqoslashlarini talab qilmaydi. $n$ ta tugunli uyumning balandligi taxminan $\log_2 n$ bo‘lgani uchun, kiritishning vaqt samaradorligi $O(log\ n)$ hisoblanadi.




**6.11-rasm**. *6.10-rasmda qurilgan uyumga yangi 10 kalitli tugunni kiritish.*

*Yangi kalit o‘z ota tugunidan katta bo‘lmaguncha (yoki ildizda bo‘lmaguncha) o‘z ota tuguni bilan almashtirish orqali siljitib boriladi (elakdan o‘tkaziladi.)*

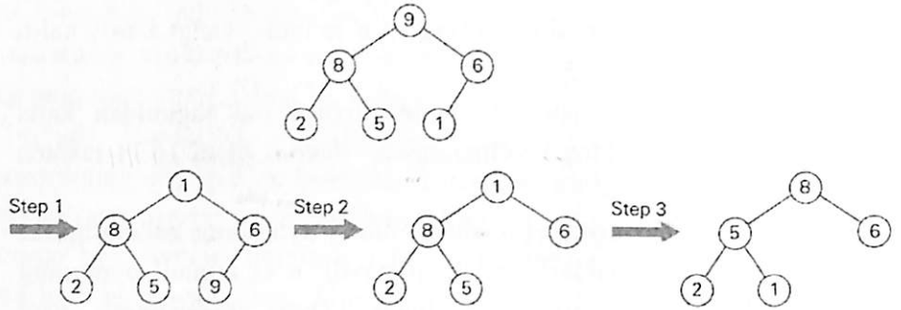
Uyumdan elementni qanday o‘chirish mumkin? Bu yerda faqat ildiz kalitini o‘chirishning eng muhim holatini ko‘rib chiqamiz, uyumdagi ixtiyoriy kalitni o‘chirish masalasini esa mashqlarga qoldiramiz. Uyumdan ildiz kalitini o‘chirish 6.12-rasmda tasvirlangan quyidagi algoritm yordamida amalga oshirilishi mumkin.

Uyumdan eng katta kalitni o‘chirish:

**1-qadam:** Ildiz kalitini uyumning oxirgi kaliti $K$ bilan almashtiring. **2-qadam:** Uyum hajmini 1 birlikka kamaytiring.

**3-qadam:** $K$ ni daraxt bo‘ylab pastga siljitish orqali kichikroq daraxtni "Heapify (uyumlanish)" holatiga keltiring, bu xuddi pastdan yuqoriga qarab uyum qurish algoritmidagidek bajariladi. Ya’ni, $K$ uchun ota ustunligini tekshiring: agar u bajarilsa, jarayon tugaydi; aks holda, $K$

ni bola tugunlarining kattarog'i bilan almashtiring va bu amalni $K$ uchun ota ustunligi sharti uning yangi holatida bajarilguncha takrorlang.




**6.12-rasm**. *Uyumdan ildiz kalitini o'chirish. O'chirilishi kerak bo'lgan kalit oxirgi kalit bilan almashtiriladi, shundan so'ng kichikroq daraxt ota ustunligi talabi qondirilgunga qadar o'z ildizidagi yangi kalitni bola tugunlaridagi kattaroq kalit bilan almashtirish orqali "uyumlanadi".*

O'chirish samaradorligi almashtirish amalga oshirilgandan va daraxt o'lchami 1 ga kamaytirilgandan so'ng daraxtni "uyumlash" uchun zarur bo'lgan asosiy taqqoslashlar soni bilan belgilanadi. Bu uyumning ikki baravar balandligidan ko'p asosiy taqqoslashlarni talab qila olmasligi sababli, o'chirishning vaqt samaradorligi ham $O(\log n)$ ga teng bo'ladi.

### 6.3.4. Piramidali saralash (Heapsort[21])

Endi saralashning qiziqarli algoritmi - ***heapsort***ni tavsiflashimiz mumkin. Bu ikki bosqichli algoritm bo'lib, quyidagicha ishlaydi.

**1-bosqich** (uyum qurish): Berilgan massiv uchun uyum qurish.

**2-bosqich** (ildizni o'chirishlar): Hosil bo'lgan uyumga $n - 1$ marta ildizni o'chirish amalini qo'llash.

Ushbu bosqichlardagi amallarning natijasida massiv elementlari kamayish tartibida yo'q qilinadi. Biroq, uyumlarning massiv orqali amalga oshirilishida o'chirilayotgan element oxirgi o'ringa

[21] Heap sort algoritmi 1954-yilda amerikalik matematik va informatika olimi John Uilliam Joseph Uilliams tomonidan ixtiro qilingan. Taxminan 1960-yillarda R. W. Floyd bu algoritmni takomillashtirib, heapify funksiyasini optimallashtirgan. Shu sababli, ba'zida bu algoritmni Uilliams–Floyd algoritmi deb ham atashadi.



joylashtirilgani sababli, hosil bo'lgan massiv aslida dastlabki massivning o'sish tartibida saralangan ko'rinishi bo'ladi. Heapsort algoritmining ishlashi 4.6-rasmda muayyan kirish ma'lumotlari asosida ko'rsatilgan. (4.3-rasmdagi bilan bir xil kirish ma'lumotlari ataylab ishlatilgan, bu

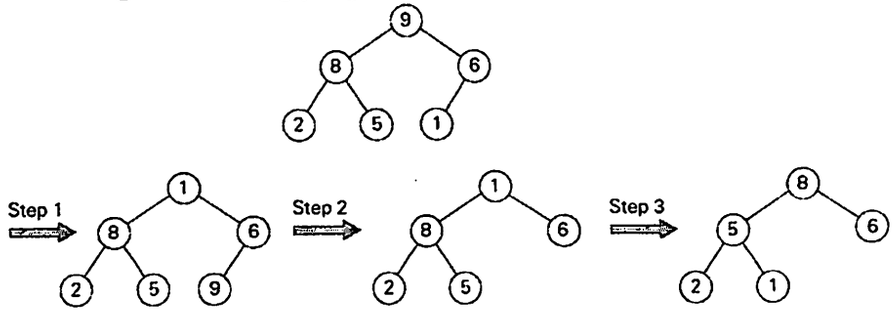
pastdan yuqoriga uyum qurish algoritmining daraxt va massiv orqali amalga oshirilishlarini taqqoslash imkonini beradi.)

| 1-bosqich (uyum qurish) | 2-bosqich (ildizni o'chirish) |
|---|---|
| 2 9 **7** 6 5 8 | **9** 6 8 2 5 7 |
| 2 **9** 8 6 5 7 | 7 6 8 2 5 \|**9** |
| **2** 9 8 6 5 7 | **8** 6 7 2 5 |
| 9 **2** 8 6 5 7 | 5 6 7 2 \|**8** |
| 9 6 8 2 5 7 | **7** 6 5 2 |
| | 2 6 5 \|7 |
| | **6** 2 5 |
| | 5 2 \|**6** |
| | **5** 2 |
| | 2 \|**5** |
| | 2 |

**6.14-rasm.** *2, 9, 7, 6, 5, 8 massivni heapsort bo'yicha saralash.* Algoritmning C++ dasturlash tilidagi ifodasi:

```
#include <iostream>
using namespace std;
// Heapni yaratish (heapify)
funksiyasi void heapify(int arr[], int
n, int i) {
  int largest = i; // Eng katta elementni ildiz sifatida
belgilang  int left = 2 * i + 1; // Chap bola  int
right = 2 * i + 2; // O'ng bola  // Agar chap bola
ildizdan katta bo'lsa  if (left < n && arr[left] >
arr[largest])
    largest = left;
```

ni bola tugunlarining kattarog'i bilan almashtiring va bu amalni $K$ uchun ota ustunligi sharti uning yangi holatida bajarilguncha takrorlang.




**6.12-rasm**. *Uyumdan ildiz kalitini o'chirish. O'chirilishi kerak bo'lgan kalit oxirgi kalit bilan almashtiriladi, shundan so'ng kichikroq daraxt ota ustunligi talabi qondirilgunga qadar o'z ildizidagi yangi kalitni bola tugunlaridagi kattaroq kalit bilan almashtirish orqali "uyumlanadi".*

O'chirish samaradorligi almashtirish amalga oshirilgandan va daraxt o'lchami 1 ga kamaytirilgandan so'ng daraxtni "uyumlash" uchun zarur bo'lgan asosiy taqqoslashlar soni bilan belgilanadi. Bu uyumning ikki baravar balandligidan ko'p asosiy taqqoslashlarni talab qila olmasligi sababli, o'chirishning vaqt samaradorligi ham $O(\log n)$ ga teng bo'ladi.

### 6.3.4. Piramidali saralash (Heapsort[21])

Endi saralashning qiziqarli algoritmi - ***heapsort***ni tavsiflashimiz mumkin. Bu ikki bosqichli algoritm bo'lib, quyidagicha ishlaydi.

**1-bosqich** (uyum qurish): Berilgan massiv uchun uyum qurish.

**2-bosqich** (ildizni o'chirishlar): Hosil bo'lgan uyumga $n - 1$ marta ildizni o'chirish amalini qo'llash.

Ushbu bosqichlardagi amallarning natijasida massiv elementlari kamayish tartibida yo'q qilinadi. Biroq, uyumlarning massiv orqali amalga oshirilishida o'chirilayotgan element oxirgi o'ringa

[21] Heap sort algoritmi 1954-yilda amerikalik matematik va informatika olimi John Uilliam Joseph Uilliams tomonidan ixtiro qilingan. Taxminan 1960-yillarda R. W. Floyd bu algoritmni takomillashtirib, heapify funksiyasini optimallashtirgan. Shu sababli, ba'zida bu algoritmni Uilliams–Floyd algoritmi deb ham atashadi.



joylashtirilgani sababli, hosil bo'lgan massiv aslida dastlabki massivning o'sish tartibida saralangan ko'rinishi bo'ladi. Heapsort algoritmining ishlashi 4.6-rasmda muayyan kirish ma'lumotlari asosida ko'rsatilgan. (4.3-rasmdagi bilan bir xil kirish ma'lumotlari ataylab ishlatilgan, bu

pastdan yuqoriga uyum qurish algoritmining daraxt va massiv orqali amalga oshirilishlarini taqqoslash imkonini beradi.)

| 1-bosqich (uyum qurish) | | | | | | 2-bosqich (ildizni o'chirish) | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 9 | 7 | 6 | 5 | 8 | **9** | 6 | 8 | 2 | 5 | 7 | |
| 2 | **9** | 8 | 6 | 5 | 7 | 7 | 6 | 8 | 2 | 5 | \|**9** | |
| **2** | 9 | 8 | 6 | 5 | 7 | **8** | 6 | 7 | 2 | 5 | | |
| 9 | **2** | 8 | 6 | 5 | 7 | 5 | 6 | 7 | 2 | \|**8** | | |
| 9 | 6 | 8 | 2 | 5 | 7 | 7 | 6 | 5 | 2 | | | |
| | | | | | | 2 | 6 | 5 | \|**7** | | | |
| | | | | | | **6** | 2 | 5 | | | | |
| | | | | | | 5 | 2 | \|**6** | | | | |
| | | | | | | **5** | 2 | | | | | |
| | | | | | | 2 | \|**5** | | | | | |
| | | | | | | 2 | | | | | | |

**6.14-rasm.** *2, 9, 7, 6, 5, 8 massivni heapsort bo'yicha saralash.* Algoritmning C++ dasturlash tilidagi ifodasi:

```
#include <iostream>
using namespace std;
// Heapni yaratish (heapify)
funksiyasi void heapify(int arr[], int
n, int i) {
  int largest = i; // Eng katta elementni ildiz sifatida
belgilang  int left = 2 * i + 1; // Chap bola  int
right = 2 * i + 2; // O'ng bola  // Agar chap bola
ildizdan katta bo'lsa  if (left < n && arr[left] >
arr[largest])
    largest = left;
```

```
    // Agar o'ng bola ham ildizdan katta
bo'lsa      if (right < n && arr[right] >
arr[largest])
        largest = right;
    // Eng katta element ildiz bo'lmasa
almashtirish    if (largest != i) {
        swap(arr[i], arr[largest]);
        // Almashishdan keyin heapify qilishni davom ettiramiz
heapify(arr, n, largest);
    }
}
// Heap sort algoritmi
void heapSort(int arr[], int
n) {    // Max heap
yaratish    for (int i = n / 2
- 1; i >= 0; i--)
heapify(arr, n, i);
    // Birma-bir elementlarni
olish    for (int i = n - 1; i >=
0; i--) {
        // Eng katta elementni oxiriga joylashtirish
swap(arr[0], arr[i]);
        // Qolgan elementlarni qayta heapify qilish
heapify(arr, i, 0);
    }
}
// Massivni chop etish
funksiyasi void
printArray(int arr[], int n)
{    for (int i = 0; i < n;
++i)        cout << arr[i] <<
" ";    cout << "\n";
}
```



```
//        Asosiy
dastur        int
main() {
    int arr[]= { 2, 9, 7, 6, 5, 8 };
int size = sizeof(arr) /
sizeof(arr[0]);    cout <<
"Dastlabki massiv: " << " ";
    printArray(arr,
size);    cout << endl;
heapSort(arr, size);
    cout << "Saralangan massiv: "<<" ";
    printArray(arr,
size); } Natija:
Dastlabki massiv: 2 9 7 6 5 8
Saralangan massiv: 2 5 6 7 8 9
```

Algoritmning uyum qurish bosqichi $O(n)$ ekanligi oldindan ma'lum, endi faqat ikkinchi bosqichning vaqt samaradorligini tekshirish kerak. $n$ dan 2 gacha kamayib boruvchi o'lchamdagi uyumlardan ildiz kalitlarini chiqarib tashlash uchun zarur bo'lgan kalit taqqoslashlar soni $C(n)$ uchun quyidagi tengsizlikni olamiz:

$$C(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2\sum_{i=1}^{n-1} \log_2 i$$

$$\le 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \le 2n \log_2 n.$$

Bu shuni anglatadiki, heapsortning ikkinchi bosqichi uchun $C(n) \in O(n \log n)$. Ikkala bosqich uchun esa, $O(n) + O(n \log n) = O(n \log n)$ bo'ladi.

Batafsilroq tahlil shuni ko'rsatadiki, uyumli saralashning vaqt samaradorligi, aslida, eng yomon va o'rtacha holatlarda $\Theta(n \log n)$ bo'ladi. Shunday qilib, heapsort vaqt samaradorligi mergesort

(birlashtirib saralash) bilan bir sinfga tegishli. Ikkinchisidan farqli o'laroq, heapsort bitta joyda ishlaydi, ya'ni qo'shimcha saqlash uchun xotira talab qilmaydi. Tasodifiy fayllar ustida o'tkazilgan vaqt bo'yicha tajribalar shuni ko'rsatadiki, heapsort quicksort (tez saralash) usuliga qaraganda sekinroq ishlaydi, ammo mergesort (birlashtirib saralash) usuli bilan raqobatbardosh bo'lishi mumkin.

### 6.4. Mustaqil ishlash uchun savollar va mashqlar

**1.** ***a.*** Pastdan yuqoriga algoritmi bo'yicha 1, 8, 6, 5, 3, 7, 4 ro'yxat uchun uyum yasang.

***b.*** Kalitlarni ketma-ket qo'yish orqali 1, 8, 6, 5, 3, 7, 4 ro'yxati uchun uyum
yasang (yuqoridan pastga algoritmi asosida).

***c.*** Pastdan yuqoriga va yuqoridan pastga yo'naltirilgan algoritmlar bir xil
ma'lumot uchun bir xil uyumni berishi har doim ham to'g'rimi?

**2.** $H[1. . n]$ massivning uyum ekanligini tekshirish algoritmini tuzing va uning vaqt bo'yicha samaradorligini aniqlang.

**3.** ***a.*** Balandligi $h$ bo'lgan uyumda joylashishi mumkin bo'lgan eng kichik va eng katta kalitlar sonini toping.

***b.*** $n$ ta tugunli uyumning balandligi $\log_2 n$ ga tengligini isbotlang.

**4.** Quyidagi tenglikni isbotlang:

$$\sum_{i=0}^{h-1} 2(h-1)2^i = 2(n - \log_2(n+1)), \text{ bu yerda } n = 2^{h+1} - 1.$$

**5.** ***a.*** Uyumdagi eng kichik qiymatga ega bo'lgan elementni topish va o'chirishning samarali algoritmini tuzing va uning vaqt bo'yicha samaradorligini aniqlang.

***b.*** $H$ uyumda berilgan $v$ qiymatli elementni topish va o'chirishning samarali
algoritmini tuzing va uning vaqt bo'yicha samaradorligini aniqlang.



**6.** Quyidagi ko'rinishda amalga oshiriladigan ustuvor navbatning uchta asosiy amalining vaqt samaradorligi sinflarini ko'rsating:

***a.*** saralanmagan massiv.

***b.*** saralangan massiv.

***c.*** binar qidiruv daraxti.

***d.*** AVL daraxti.

***e.*** uyum.

**7.** Quyidagi ro'yxatlarni heapsort bo'yicha saralashda uyumlarning massiv ko'rinishidan foydalaning.

***a.*** 1, 2, 3, 4, 5 (o'sish tartibida)

***b.*** 5, 4, 3, 2, 1 (o'sish tartibida)

***c.*** S, O, R, T, I, N, G (alifbo tartibida)

**8.** Heapsort barqaror saralash algoritmi hisoblanadimi?

**9.** Heapsort transform-and-conquer texnikasining qanday turini ifodalaydi?

**10.** Qaysi saralash algoritmi heapsortdan boshqa ustuvor navbatdan foydalanadi?

**11.** O'zingiz tanlagan tilda saralashning uchta ilg'or algoritmi - mergesort, quicksort va heapsortni amalga oshiring va ularning $n = 103, 104, 105$ va $106$ o'lchamli massivlarda ishlashini tekshiring. Ushbu o'lchamlarning har biri uchun quyidagilarni hisobga oling

***a.*** $[1. . n]$ oraliqdagi butun sonlarning tasodifiy hosil qilingan fayllari.

***b.*** $1,2,\ldots,n$ butun sonlarning o'suvchi fayllari.

***c.*** $n, n-1,\ldots,1$ butun sonlarning kamayuvchi fayllari.

**12.** Spagetti usulida saralash. Bir hovuch pishirilmagan spagetti, uzunligi saralanishi kerak bo'lgan sonlarni ifodalovchi alohida tayoqchalarni tasavvur qiling. Ushbu noan'anaviy ko'rinishdan foydalanadigan "spagetti saralash" - saralash algoritmini tuzing.

### 6.5. Tashqi saralash algoritmlari

Tashqi saralash – bu ma'lumotlarni tartiblash jarayonida tashqi xotira qurilmalari, qoida bo'yicha qattiq disk qo'llaniladigan saralash

algoritmi hisoblanadi. Tashqi saralash algoritmlari tezkor hotira joylashtirish imkoni bo‘lmagan katta hajmdagi ma’lumotlar ro‘yxatini qayta ishlash uchun ishlab chiqilgan. Turli tashqi axborot tashish vositalariga murojaat ushbu algoritmga qo'shimcha cheklovlarni qo'yadi: tashqi xotira qurilmalariga kirish ketma-ketlikda amalga oshiriladi, ya'ni vaqtning har bir bosqichida faqat joriy elementdan keyingi elementni o'qish yoki yozish mumkin, ma'lumotlar miqdori ularni tezkor xotiraga joylashtirishga imkon bermaydi.

Tashqi saralash algoritmlari eng ommalashgan turlari:

- birlashtirish bilan saralash (oddiy birlashtirish va tabiiy birlashtirish);
- yaxshilangan saralash (ko‘pfazali saralash va kaskadli saralash).

Taqdim etilgan tashqi saralash usullaridan birlashtirish bilan saralash usuli eng muhimi hisoblanadi. Birlashtirish bilan saralash algoritmini tavsiflashdan oldin, ba'zi ta'riflar bilan tanishib olamiz.

Tashqi saralashni qo‘llashning asosiy tushunchalaridan biri seriya (qator) tushunchasi hisoblanadi. Seriya (tartiblangan kesma) – bu kalit bo‘yicha saralangan elementlar ketma-ketligi. Seriyadagi elementlar soni seriyaning uzunligi deyiladi. Bitta elementdan iborat qator hammavaqt saralangan bo‘ladi. Oxirgi seriya fayl seriyasining qolgan qismidan qisqaroq bo'lishi mumkin. Fayldagi seriyalarning maksimal soni N (barcha elementlar tartiblanmagan). Seriyalarning minimal soni bitta (barcha elementlari tartiblangan).

Tashqi saralash algoritmlarining asosida ikkita protsedura yotadi: birlashtirish va ajratish. Birlashtirish - bu mavjud bo'lgan elementlarni to‘liq o‘qish orqali ikkita (yoki undan ortiq) tartiblangan seriyalarni bitta tartibli ketma-ketlikga birlashtirish jarayoni. Ajratish – saralangan seriyani ikki yoki undan ortiq yordamchi fayllarga bo'lish jarayoni.

Faza – bu elementlarning butun ketma-ketligini bir marta qayta ishlash uchun bajarilgan harakat. Ikki fazali saralash – ikkida amal: ajratish va birlashtirish alohida qo‘llaniladigan saralash algoritmi. Bir fazali saralash – ajratish va birlashtirish fazalarining birlashmasi orqali saralash algoritmi.



Berilgan ma’lumotlar ikkita yordamchi fayllarga ajratish orqali saralash ikki yo‘lli birlashtirish bilan saralash deb ataladi. Berilgan ma’lumotlar $N$ $(N > 2)$ ta yordamchi fayllarga ajratish orqali saralash ko‘p yo‘lli birlashtirish bilan saralash deb ataladi.

Birinchidan, seriya ikki yoki undan ortiq yordamchi fayllarga bo'linadi. Bu taqsimot navbati bilan amalga oshiriladi: birinchi seriya birinchi yordamchi faylga, ikkinchi seriya - ikkinchiga yordamchi faylga va oxirgi seriya oxirgi yordamchi faylga yoziladi. Keyin yana seriyalarga ajratish birinchi yordamchi fayldan boshlanadi. Barcha seriyalar taqsimlangandan so'ng, ular uzunroq saralangan seriyalarga birlashtiriladi, ya'ni birlashtirilgan har bir yordamchi fayldan bitta seriya olinadi. Agar seriya qaysidir faylda tugasa, keyingi seriyaga o'tish amalga oshirilmaydi. Saralash turiga qarab, yaratilgan uzunroq tartiblangan seriyalar kirish faylga yoki yordamchi fayllardan biriga yoziladi. Barcha yordamchi fayllarning barcha seriyalari yangi seriyalarga birlashtirilgandan so'ng, ularni yana qayta tarqsimlash boshlanadi va h.k. barcha ma'lumotlar saralanmaguncha davom etadi.

Birlashtirish bilan saralashning quyidagi asosiy xarakteristikalarini ajratib ko‘rsatish mumkin:

- saralashni amalga oshirishdagi faza (bosqich)lar soni; - seriyalar taqsimlanadigan yordamchi fayllar soni.

### 6.5.1. Tez saralash usuli

Tez saralash - almashinish prinsipiga asoslangan mukammallashgan saralashning bir usuli. Tez saralash usuli 1962 yilda CH.A.R.Xoar tomonidan taklif etilgan. Tez saralash - bu usulning unumdorligiga ta'sir qiluvchi muhim parametrni olishning turli yondashuvlarini aks ettiruvchi bir qator algoritmlarning umumiy nomi hisoblanadi.

Umuman tez saralash algoritmi kirish massivini bo‘laklarga bo‘lish g‘oyasiga asoslanadi. Bo‘laklarga ajratish tayanch elementni tanlab olishdan boshlanadi. Tayanch (yetakchi) element - massivning ma'lum bir qoida asosida tanlangan qaysidir elementi. Algoritmning

to'g'riligi nuqtai nazaridan olganda, tayanch elementni tanlash uchun alohida qoida shart emas. Algoritm samaradorligini oshirish nuqtai nazaridan o'rta qiymatli (mediana) elementini tanlash kerak, ammo ma'lumotlarning tartiblanganligi haqida qo'shimcha ma'lumotlarga ega bo'lmasdan uni tanlab olish odatda mumkin emas. Doimiy ravishda bir xil elementni (masalan, o'rtadagi yoki oxirgi) elementni tanlash yoki tasodifiy tanlangan indeksga ega elementni olish kerak.

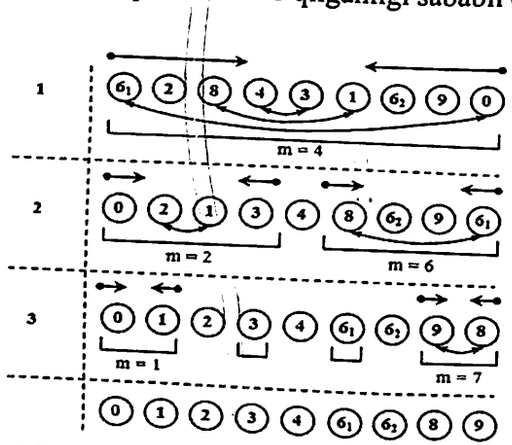
Xoarning tez saralash algoritmi: massiv $x[n]$ berilgan bo'lsin.

1-qadam. Massivning tayanch elementini tanlash.

2-qadam. Massivni tayanch elementga nisbatan ikkita – chap va o'ng qism massivlarga ajratish. Bunda chap qism massivdagi elementlar tayanch elementdan kichik, o'ng qismmassivdagi elementlar katta qiymatga ega bo'lishi e'tiborga olinishi kerak.

3-qadam. Keyin 2-qadamdagi amallar har ikkila qismmassivlar uchun takrorlanadi. Har bir qadamda hosil bo'lgan qismmassivlarda ikkita element qolmaguncha takrorlanadi. (6.15-rasm).

Quicksort, birinchi navbatda, uni amalga oshirish osonligi, turli xil kirish ma'lumotlarda yaxshi ishlashi va ko'p hollarda boshqa saralash usullariga qaraganda kamroq resurs talab qilganligi sababli ommalashib ketgan.




6.15-rasm. Tez saralash algoritmi ishlash sxemasi

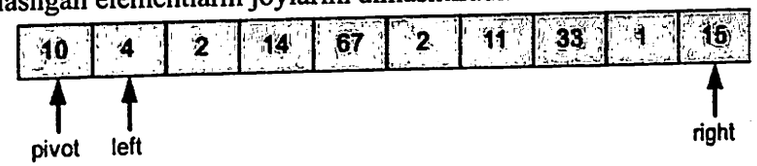
Tez saralash usulining asosiy jihati qayta tartiblash algoritmi hisoblanadi. Saralashni massiv misolida ko'rib chiqamiz:

$$A[10] = \{10,4,\ 2,\ 14,\ 67,2,\ 11,\ 33,\ 1,15\}.$$



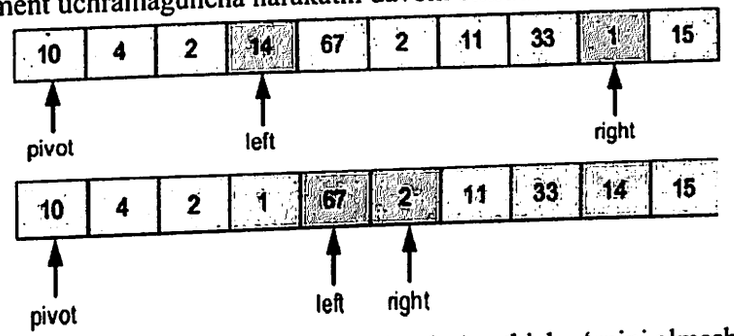
Qayta tartiblash algoritmini amalga oshirish uchun massivning chap chegarasi, ya'ni left ko'rsatgichi olinadi. Ko'rsatgich o'ngga harakatlanadi toki, tayanch elementdan kichik element topilmaguncha. Qayta tartiblash algoritmini amalga oshirish uchun massivning o'ng elementi ya'ni right ko'rsatgichi olinadi. Ko'rsatgich chapga harakatlanadi toki, tayanch elementdan katta.

Massivning chapdagi elementi tayanch pivot bo'lsin. left ko'rsatgichini undan keying elementga o'rnatamiz; right — eng oxirgi. Algoritm 10 ning to'g'ri joyini aniqlab va ish jarayonida noto'g'ri joylashgan elementlarni joylarini almashtiradi.




6.16-rasm. Tez saralash sxemasi

Agar joylashishi tayanch elementga nisbatan noto'g'ri bo'lsa, ko'rsatgichning harakati to'xtaydi. left ko'rsatgichi 10 dan katta element uchramaguncha harakatlanaveradi; right ko'rsatgichi 10 dan kichik element uchramaguncha harakatni davom ettiradi.




6.17-rasm. Chap va o'ng ko'rsatkich o'rnini almashtirish

Jarayon right ko'rsatkichi left dan chapda bo'lib qolmaguncha davom etadi.

Agar shu holat yuz bersa, tayanch element bilan right ko'rsatayotgan element joy almashadi (6.18-rasm).

to'g'riligi nuqtai nazaridan olganda, tayanch elementni tanlash uchun alohida qoida shart emas. Algoritm samaradorligini oshirish nuqtai nazaridan o‘rta qiymatli (mediana) elementini tanlash kerak, ammo ma'lumotlarning tartiblanganligi haqida qo'shimcha ma'lumotlarga ega bo‘lmasdan uni tanlab olish odatda mumkin emas. Doimiy ravishda bir xil elementni (masalan, o'rtadagi yoki oxirgi) elementni tanlash yoki tasodifiy tanlangan indeksga ega elementni olish kerak.

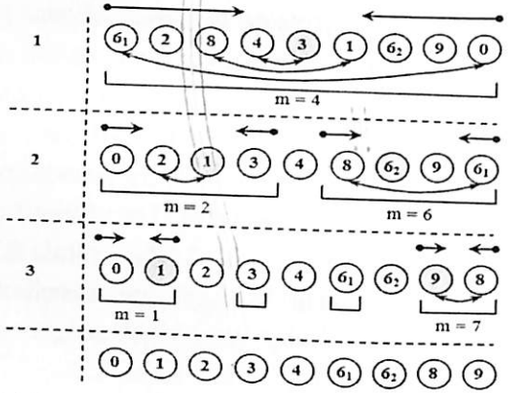
Xoarning tez saralash algoritmi: massiv $x[n]$ berilgan bo‘lsin.

1-qadam. Massivning tayanch elementini tanlash.

2-qadam. Massivni tayanch elementga nisbatan ikkita – chap va o‘ng qism massivlarga ajratish. Bunda chap qism massivdagi elementlar tayanch elementdan kichik, o‘ng qismmassivdagi elementlar katta qiymatga ega bo‘lishi e’tiborga olinishi kerak.

3-qadam. Keyin 2-qadamdagi amallar har ikkila qismmassivlar uchun takrorlanadi. Har bir qadamda hosil bo‘lgan qismmassivlarda ikkita element qolmaguncha takrorlanadi. (6.15-rasm).

Quicksort, birinchi navbatda, uni amalga oshirish osonligi, turli xil kirish ma’lumotlarda yaxshi ishlashi va ko'p hollarda boshqa saralash usullariga qaraganda kamroq resurs talab qilganligi sababli ommalashib ketgan.




6.15-rasm. Tez saralash algoritmi ishlash sxemasi

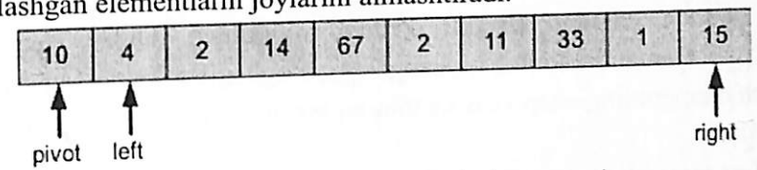
Tez saralash usulining asosiy jihati qayta tartiblash algoritmi hisoblanadi. Saralashni massiv misolida ko‘rib chiqamiz:

$$A[10] = \{10,4,\ 2,\ 14,\ 67,2,\ 11,\ 33,\ 1,15\}.$$



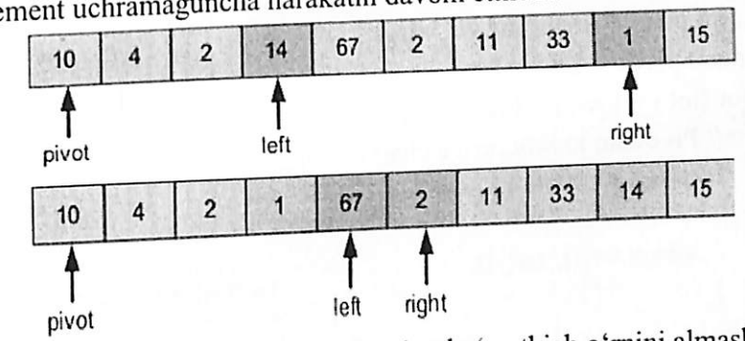
Qayta tartiblash algoritmini amalga oshirish uchun massivning chap chegarasi, ya’ni left ko‘rsatgichi olinadi. Ko‘rsatgich o‘ngga harakatlanadi toki, tayanch elementdan kichik element topilmaguncha. Qayta tartiblash algoritmini amalga oshirish uchun massivning o‘ng elementi ya’ni right ko‘rsatgichi olinadi. Ko‘rsatgich chapga harakatlanadi toki, tayanch elementdan katta.

Massivning chapdagi elementi tayanch pivot bo‘lsin. left ko‘rsatgichini undan keying elementga o‘rnatamiz; right — eng oxirgi. Algoritm 10 ning to‘g‘ri joyini aniqlab va ish jarayonida noto‘g‘ri joylashgan elementlarni joylarini almashtiradi.




6.16-rasm. Tez saralash sxemasi

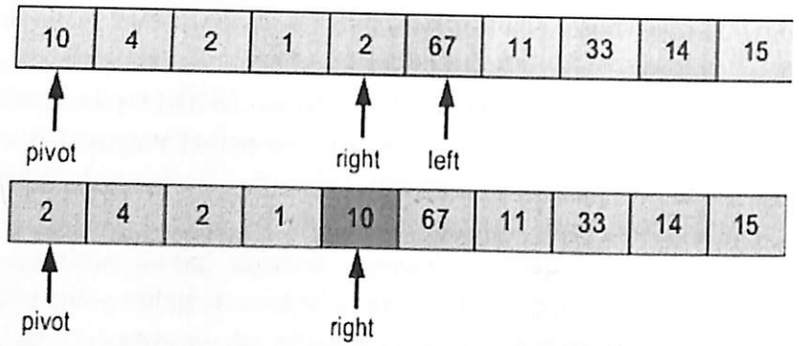
Agar joylashishi tayanch elementga nisbatan noto‘g‘ri bo‘lsa, ko‘rsatgichning harakati to‘xtaydi. left ko‘rsatgichi 10 dan katta element uchramaguncha harakatlanaveradi; right ko‘rsatgichi 10 dan kichik element uchramaguncha harakatni davom ettiradi.




6.17-rasm. Chap va o‘ng ko‘rsatkich o‘rnini almashtirish

Jarayon right ko‘rsatkichi left dan chapda bo‘lib qolmaguncha davom etadi.

Agar shu holat yuz bersa, tayanch element bilan right ko‘rsatayotgan element joy almashadi (6.18-rasm).




6.18-rasm. Tayanch element joyini aniqlash

Tayanch element kerakli o‘ringa joylashgan: undan chapda joylashgan elementlar kichik qiymatga ega, o‘ngdagilar- katta. Algoritm tayanch elementning chap va o‘ng tomonidagi massivlar uchun rekursiv chaqiriladi.

Tez saralash algoritmining C++ tilidagi dasturi

```
#include <iostream>
using namespace std;
// Elementlarni joylashtirish funksiyasi
int partition(int arr[], int low, int high)
{   int pivot = arr[high]; // Oxirgi
element pivot    int i = low - 1;
  for (int j = low; j < high; j++) {
    // Pivotdan kichik bo'lsa chap tomonga
    if (arr[j] < pivot) {
i++;
      swap(arr[i], arr[j]);
    }
  }
  // Pivotni o'rniga qo'yish
swap(arr[i + 1], arr[high]);
  return (i + 1);
}
// Tez saralash algoritmi (rekursiv)
void quickSort(int arr[], int low, int
```



```
high) {   if (low < high) {      //
Partition index
    int pi = partition(arr, low, high);
 // Rekursiv chaqiruv      quickSort(arr, low, pi - 1);
quickSort(arr, pi + 1, high);
  }
}
// Massivni chop etish void
printArray(int arr[], int size)
{   for (int i = 0; i < size;
i++)      cout << arr[i] << "
";
  cout << endl;
}
//        Asosiy
dastur        int
main() {
  int arr[] = {10, 4, 2, 14, 67, 2, 11, 33,
1, 15};   int n = sizeof(arr) /
sizeof(arr[0]);   cout <<
"Saralanmagan massiv: ";
  printArray(arr, n);
quickSort(arr, 0, n - 1);
cout << "Saralangan massiv:
";
  printArray(arr, n);

return
0; }
```

**Natija:**

Dastlabki massiv: 10 4 2 14 67 2 11 33 1 15
Saralangan massiv: 1 2 2 4 10 11 14 15 33 67

### 6.5.2. Birlashtirish bilan saralash usuli

Birlashtirish – bu ikkita yoki undan ko‘p tartiblangan qism massivlarni bitta tartiblangan massivga birlashtirishni anglatadi.

Birlashtirish bilan saralash eng oddiy saralash algoritmlaridan biri hisoblanadi (eng tezkor algoritmlar orasida). Ushbu algoritmning o'ziga xos xususiyati shundaki, u massiv elementlari bilan asosan ketma-ket ishlaydi, shuning uchun bu algoritm turli xil apparat cheklovlari bo'lgan tizimlarda (masalan, qattiq diskdagi ma'lumotlarni) saralashda qo'llaniladi. Bundan tashqari, birlashtirish bilan saralash - bog'langan ro'yxatlar kabi ma'lumotlar tuzilmalarini saralash uchun samarali ishlatilishi mumkin bo'lgan algoritm hisoblanadi.

Ushbu algoritm oraliq natijalarni saqlash uchun kirish massivining o'lchamiga o'xshash xotiradan foydalanish mumkin bo'lganda qo'llaniladi. U “bo‘laklarga bo‘l va hukmronlik qil” tamoyili asosida qurilgan. Birinchidan, masala bir nechta kichik masalalarga bo'linadi. Keyinchalik bu qism masalalan rekursiv chaqiruv bilan yoki ularning hajmi yetarlicha kichik bo'lsa, to'g'ridan-to'g'ri hal qilinadi. Keyin ularning yechimlari birlashtirilib, dastlabki masala yechimi olinadi (6.19-rasm). Birlashtirish bilan saralash algoritmi ikkita tartiblangan massivni talab qiladi. E'tibor bering, bitta elementdan iborat massiv, ta'rifiga ko'ra, tartiblangan hisoblanadi.

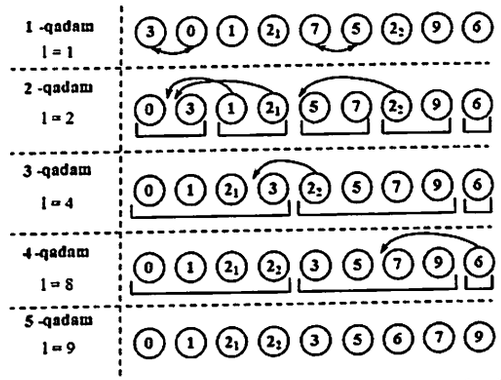
Birlashtirish bilan saralash algoritmi:

1-qadam. Massivning mavjud elementlarini juftliklarga ajratish va har bir qismning elementlarini birlashtirib, uzunligi 2 ga teng saralangan massivlar zanjirini hosil qilish (ehtimol, jufti bo'lmaganda bitta elementni olish mumkin).

2-qadam. Mavjud tartiblangan zanjirlar juftligini ajratib olish va har bir juftning zanjirlarini birlashtirish.

3-qadam. Agar saralangan zanjirlar uzunligi 1 dan katta bo‘lsa, 2-qadamni takrorlash.

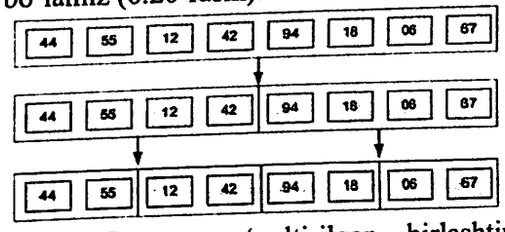




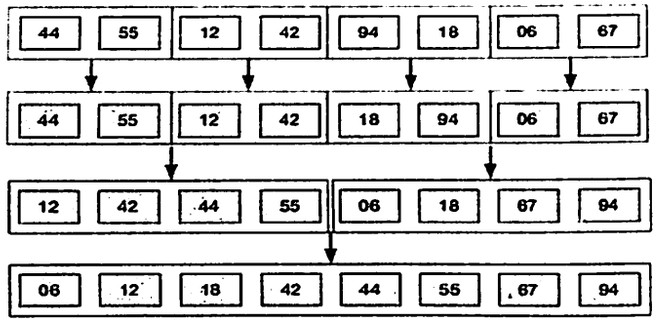

6.19-rasm. Birlashtirish bilan saralash algoritmining ishlash sxemasi

Algoritmning kamchiligi shundaki, u uzunligi $n$ ga teng qo'shimcha xotirani talab qiladi (yordamchi massivni saqlash uchun). Bundan tashqari, u bir xil qiymatlarga ega bo'lgan elementlarning tartibini saqlashga kafolat bermaydi. Lekin uning vaqt murakkabligi har doim $O(n \log n)$ ga teng bo‘ladi.

*Pastga yo'nalgan birlashtirish bilan saralash.* Chiquvchi ketma-ketlik 1 element bo‘ylab ketma-ketlikni olmagunimizcha o‘rtasidan rekursiv bo‘linadi. Olingan ketma-ketlikdan birlashtirish usulida tartiblangan juftliklarni, keyin choraklarni va h.k hosil qilamiz. Ketma-ketlikni 2 qismga bo‘lamiz (6.20-rasm).




6.20-rasm. Pastga yo‘naltirilgan birlashtirish bilan saralash Har bir ketma-ketlikni birlashtirish usuli bilan tartiblaymiz va tayyor ketmaketlikni olamiz.




6.21-rasm. Pastga yo‘naltirilgan birlashtirish bilan saralash

Pastga yo‘nalgan birlashtirishli saralash usulining C++ dagi dasturi

```
#include
<iostream>
#define SIZE 16
using namespace
std;
// pastga yo‘nalgan birlashtirishli saralanuvchi
funksiya void mergeSort(int *a, int l, int r) {
if (l == r) return; // chegaralar yopilguncha
 int mid = (l + r) / 2; // ketma-ketlik o‘rtasini
aniqlaymiz  //va har bir qism uchun saralash
funksiyasini rekursiv chaqiramiz mergeSort(a, l, mid);
mergeSort(a, mid + 1, r);  int i = l;  // birinchi yo‘l boshi
int j = mid + 1; // ikkinchi yo‘l boshi
 int *tmp = (int*)malloc(r * sizeof(int)); // qo‘shimcha massiv
 for (int step = 0; step < r - l + 1; step++) {
  // shakllanayotgan ketma-ketlikka ikkita yo‘lning kichigini
yozib qo‘yamiz
  // agar  j > r bo‘lsa, birinchining
qoldig‘i   if ((j > r) || ((i <= mid) &&
(a[i] < a[j]))) {     tmp[step] = a[i];
   i++;  }
else  {
tmp[step] =
```

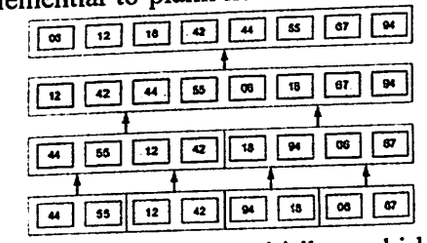


```
a[j];     j++;
}
 } // shakllangan ketma-ketlikni dastlabki massivga ko‘chiramiz
 for (int step = 0; step < r - l + 1;
step++)   a[l + step] = tmp[step];
} int main() {  int
a[SIZE];  for (int i = 0;
i<SIZE; i++)  {    a[i] =
(rand() % 100);
printf(" %d ", a[i]); }
 mergeSort(a, 0, SIZE - 1); // saralash funksiyasini chaqiramiz
 printf("\n");
 // saralangan massivni
chiqaramiz  for (int i = 0;
i<SIZE; i++)   printf(" %d
", a[i]);  getchar();  return 0;
}
```

Natija:

```
41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91
0 5 24 27 34 41 45 58 61 62 64 67 69 78 81 91
```

*Yuqoriga yo‘naltirib birlashtirishli saralash.* Yuqoriga yo‘naltirib birlashtirishli saralash usuli ketma-ketlikni teng ikkiga bo‘lish protsedurasi orqali ifodalanadi. Chiquvchi ketma-ketlik xuddi ketma-ket olingan elementlar to‘plami ko‘rinishida bo‘ladi.




6.22-rasm. Yuqoriga yo‘naltirilgan birlashtirish bilan saralash

Yuqoriga yo‘naltirib birlashtirishli saralash usulining C++ dagi dasturi

```
#include
<istream>
#define SIZE 15
using namespace
std;
// Yuqoriga yo‘naltirib birlashtirishli
usuli void mergeSort(int *a, int n) {
 int step = 1; //ketma-ketlikni bo‘lish qadami
 int *temp = (int*)malloc(n * sizeof(temp)); // qo‘shimcha
massiv  while (step < n) { // qadamlar massiv uzunligidan
kichik
  int index = 0;  //natijaviy massiv indeksi
int l = 0;  // maydonning chap chegarasi
int m = l + step; // maydonning o‘rtasi
int r = l + step * 2; // maydonning o‘ng
chegarasi  do {  m = m < n ? m : n
r = r < n ? r : n;
   int i1 = l, i2 = m; // taqqoslanayotgan elementlar
indeksi  for (; i1 < m && i2 < r; ) {
   if (a[i1] < a[i2]) { temp[index++] =
a[i1++]; }  else { temp[index++] = a[i2++];
}
   }
while (i1 < m) temp[index++] = a[i1++];
// saralanayotgan maydonning qolgan elementlarini
kiritamiz  while (i2 < r) temp[index++] = a[i2++];
//natijaviy massivda  l += step * 2; // keyingi
saralanayotgan maydonga o‘tamiz  m += step * 2;
r += step * 2;  } while (l < n);
// toki saralanayotgan maydonning chap chegarasi ketma-
ketlikning ichida  for (int i = 0; i < n; i++) // shakllangan
massivni a ga qaytaramiz  a[i] = temp[i];  step *= 2; //
Bo‘linish qadamini 2 barobar oshirish
```



```
 } } int
main() {
int
a[SIZE];
 //Massiv elementlarini
to‘ldiramiz  for (int i = 0;
i<SIZE; i++)  {  a[i] =
(rand() % 100);  printf("
%d ", a[i]);
 }
 mergeSort(a, SIZE); // saralash funksiyasini chaqiramiz
printf("\n");
 // saralangan massivni
chiqaramiz  for (int i = 0;
i<SIZE; i++)  printf(" %d
", a[i]); getchar(); return 0;
}
```

Natija:

```
41 67 34 0 69 24 78 58 62 64 5 45 81 27 61
0 5 24 27 34 41 45 58 61 62 64 67 69 78 81
```

### 6.5.3. Oddiy birlashtirish bilan saralash

Oddiy birlashtirish bilan saralash algoritmi ketma-ket birlashtirish protsedurasiga asoslangan eng oddiy tashqi saralash algoritmi hisoblanadi. Ushbu algoritmda seriya uzunligi har bir qadamda belgilab qo‘yiladi. Kirish faylida barcha seriyalar uzunligi 1 ga ega, birinchi bosqichdan keyin u 2, ikkinchidan keyin - 4, uchinchidan keyin - 8, k - chi bosqichdan keyin - 2k ga teng bo‘ladi.

Oddiy birlashtirish bilan saralash algoritmi

1-qadam: Kirish f faylini ikkita yordamchi f1 va f2 fayllarga ajratish;

2-qadam: f1 va f2 yordamchi fayllari f fayliga birlashtirilib, bunda har bir yordamchi fayldagi bitta elementlar saralangan juftligini hosil qilish.

3-qadam: Olingan natija f fayli 1 va 2 qadamlarda ko‘rsatilganidek yangidan qayta ishlanadi. Bunda saralangan juftliklarni saralangan 4 talikka o‘tkazish.

4-qadam: Birlashtirish qadamini to‘rttalikdan sakkiztalik seriyaga birlashtirish uchun takrorlash. Bu jarayonni har bir qadamda seriya uzunligini e’tiborga olib fayl to‘liq saralangan holatga kelgunga qadar takrorlash (8.30-rasm).

Har bir i- o‘tishda ikkita uzunligi 2i ga teng bo‘lgan faylga ega bo‘lamiz. Jarayon 2i>=n shart bajarilgandan keyin yakunlanadi. Bundan kelib chiqadiki, oddiy birlashtirish bilan saralash algoritmning murakkabligi O(log n) bilan baholanadi.

Oddiy birlashtirish bilan saralashning tugash belgilari quyidagi shartlar bilan aniqlanadi:

- seriyaning uzunligi fayldagi elementlar sonidan kam emas (birlashtirish bosqichidan keyin aniqlanadi);
- seriyalar soni 1 ta (birlashtirish bosqichida aniqlanadi).
- bir fazali saralash bilan, seriyani taqsimlashdan keyin ikkinchi yordamchi fayl bo'sh qoladi.

| | *Kirish fayli f: 5 7 3 2 8 4 1* | |
|---|---|---|
| | *Taqsimlash* | *Birlashtirish* |
| 1-o‘tish | *f1*: 5 3 8 1<br>*f2*: 7 2 4 | *f*: 5 7 2 3 4 8 1 |
| 2-o‘tish | *f1*: 5 3 4 8<br>*f2*: 2 3 1 | *f*: 2 3 5 7 1 4 8 |
| 3-o‘tish | *f1*: 2 3 5 7<br>*f2*: 1 4 8 | *f*: 1 2 3 4 5 7 8 |

6.23-rasm. Ikkiyo‘lli ikkifazali oddiy birlashtirish bilan saralash sxemasi //oddiy birlashtirish funksiyasi void Simple_Merging_Sort (char *name){
int a1, a2, k, i, j, kol, tmp;
FILE *f, *f1, *f2;
kol = 0;
if ( (f = fopen(name,"r")) == NULL )
printf("\nKirish faylini o‘qish mumkin



emas..."); else { while ( !feof(f) ) {
fscanf(f,"%d",&a1);

kol++;
}
fclose(f);
} k = 1; while ( k < kol ){
f = fopen(name,"r"); f1 =
fopen("smsort_1","w"); f2
= fopen("smsort_2","w");
if ( !feof(f) )
fscanf(f,"%d",&a1); while
( !feof(f) ){
for ( i = 0; i < k && !feof(f) ; i++ ){
fprintf(f1,"%d ",a1);
fscanf(f,"%d",&a1);
}
for ( j = 0; j < k && !feof(f) ; j++ ){
fprintf(f2,"%d ",a1);
fscanf(f,"%d",&a1);
} } fclose(f2);
fclose(f1); fclose(f); f =
fopen(name,"w"); f1 =
fopen("smsort_1","r"); f2 =
fopen("smsort_2","r"); if (
!feof(f1) )
fscanf(f1,"%d",&a1); if (
!feof(f2) )
fscanf(f2,"%d",&a2);
while ( !feof(f1) && !feof(f2) ){
i =
0; j
= 0;

```
    while ( i < k && j < k && !feof(f1) &&
!feof(f2) ) {       if ( a1 < a2 ) {
fprintf(f,"%d ",a1);

fscanf(f1,"%d",&a1);
i++;       }       else {
fprintf(f,"%d ",a2);

fscanf(f2,"%d",&a2);
j++;
      }
    }
    while ( i < k && !feof(f1) ) {
fprintf(f,"%d ",a1);

fscanf(f1,"%d",&a1);
i++;
    }
    while ( j < k && !feof(f2) ) {
fprintf(f,"%d ",a2);

fscanf(f2,"%d",&a2);
j++;

}
}
  while ( !feof(f1) ) {
fprintf(f,"%d ",a1);
    fscanf(f1,"%d",&a1);
  }
  while ( !feof(f2) ) {
fprintf(f,"%d ",a2);
fscanf(f2,"%d",&a2);    }
```



```
fclose(f2);    fclose(f1);
fclose(f);    k *= 2;
 }
 remove("smsort_1");
remove("smsort_2");
}
```

E'tibor bering, tashqi saralashni oddiy birlashtirish usuli bilan amalga oshirish uchun tezkor xotirada faqat ikkita o'zgaruvchini ajratish kerak - yordamchi fayllardan navbatdagi elementlarni (yozuvlarni) joylashtirish uchun. Kirish va yordamchi fayllar O(log n) marta o'qiladi va bir xil miqdorda yoziladi.

### 6.5.4. Tabiiy birlashtirish bilan saralash

Oddiy birlashtirish bilan saralangan ma'lumotlarning qisman tartiblangan bo‘lishi hech qanday afzallik bermaydi. Buning sababi, har bir o'tishda qat’iy uzunlikdagi seriyalar birlashtiriladi. Tabiiy birlashtirish bilan esa ketma-ketlik uzunligi cheklanmaydi, lekin har bir o'tishda ajratilgan qism ketma-ketlikdagi oldindan saralangan elementlarning soni belgilab qo‘yiladi.

Har doim ikkita mumkin bo'lgan eng uzun ketma-ketlikni birlashtiradigan usul - bu tabiiy birlashtirish usuli hisoblanadi. Bu usul maksimal uzunlikdagi seriyalarni birlashtiradi.

Tabiiy birlashtirish bilan saralash algoritmi

1-qadam. Kirish f fayli ikkita f1 va f2 yordamchi fayllarga ajratiladi. Ajratish quyidagi shaklda amalga oshiriladi: kirish ketma-ketligidagi ai yozuvlar navbati bilan quyidagicha o‘chiladi, agar qo‘shni yozuvlarning kalit qiymatlari f(ai)<=f(ai+1) shartni qanoatlantirsa, u holda ular birinchi f1 yordamchi fayliga yoziladi. Qachonki f(ai)>f(ai+1) kabi shartni qanoatlantiruvchi ai+1 yozuv paydo bo‘lsa, bu yozuv f2 yordamchi fayliga ko‘chiriladi. Bu jarayon kirish fayli to‘liq yordamchi fayllarga taqsimlab bo‘linmaguncha davom ettiriladi.

2-qadam. Yordamchi fayllar f1 va f2 ni f faylga birlashtirish, bunda bunda qator tartiblangan ketma-ketliklarni hosil qiladi.

3-qadam. Natija fayli f yana 1- va 2-qadamlarda ko‘rsatilganidek yana qayta ishlanadi.

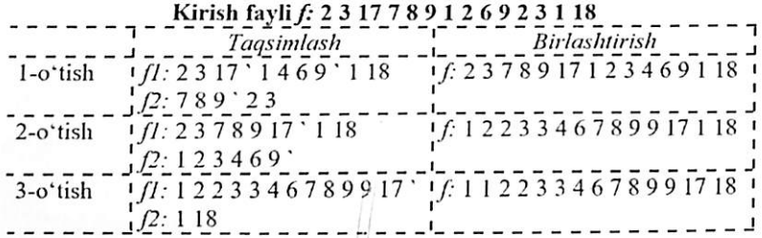
4-qadam. Saralangan ketma-ketliklarni birlashtirish qadamlari, fayl to‘liq saralanmaguncha takrorlanadi.

"`" belgisi ketma-ketlikning tuguganligini bildiradi.

Saralashning tugash belgilari quyidagi shartlar bilan aniqlanadi:

- seriyalar soni 1 ga teng (birlashtirish bosqichida aniqlanadi).
- bir fazali saralash bilan, seriyani taqsimlashdan keyin ikkinchi yordamchi fayl bo'sh qoldi.

Taqsimlash bosqichidan so'ng yordamchi fayllardagi seriyalar soni bir-biridan bittadan ko'p bo'lmagan farq qiladigan tabiiy birlashtirish muvozanatli birlashtirish, aks holda muvozanatsiz birlashtirish deb ataladi.




6.24-rasm. Ikkiyo‘lli ikkifazali tabiiy birlashtirish bilan saralash sxemasi

```
//tabiiy birlashtirish bilan saralash funksiyasi
void Natural_Merging_Sort (char
*name){   int s1, s2, a1, a2, mark;
FILE *f, *f1, *f2;   s1 = s2 = 1;
 while ( s1 > 0 && s2 > 0 ){
   mark = 1;    s1 = 0;
s2 = 0;    f =
fopen(name,"r");    f1 =
fopen("nmsort_1","w");
f2 =
fopen("nmsort_2","w");

fscanf(f,"%d",&a1)
;    if ( !feof(f) ) {
```



```
   fprintf(f1,"%d ",a1);
  }
  if ( !feof(f) ) fscanf(f,"%d",&a2);    while
( !feof(f) ){     if ( a2 < a1 ) {      switch
(mark) {       case 1:{fprintf(f1,"' "); mark
= 2; s1++; break;}       case 2:{fprintf(f2,"'
"); mark = 1; s2++; break;}
     }
    }
    if ( mark == 1 ) { fprintf(f1,"%d ",a2);
s1++; }     else { fprintf(f2,"%d ",a2); s2++;}
    a1 = a2;
    fscanf(f,"%d",&a2);
  }
  if ( s2 > 0 && mark == 2 ) {
fprintf(f2,"'");}    if ( s1 > 0 && mark
== 1 ) { fprintf(f1,"'");}    fclose(f2);
fclose(f1);    fclose(f);    cout << endl;
Print_File(name);
  Print_File("nmsort_1");
Print_File("nmsort_2");    cout
<< endl;    f =
fopen(name,"w");    f1 =
fopen("nmsort_1","r");    f2 =
fopen("nmsort_2","r");    if (
!feof(f1) )
fscanf(f1,"%d",&a1);    if (
!feof(f2) )
fscanf(f2,"%d",&a2);    bool
file1, file2;    while ( !feof(f1)
&& !feof(f2) ){     file1 = file2
= false;     while ( !file1 &&
!file2 ) {      if ( a1 <= a2 ) {
```

```
 fprintf(f,"%d ",a1);          file1
 = End_Range(f1);
        fscanf(f1,"%d",&a1);
      }       else {
 fprintf(f,"%d ",a2);
 file2 = End_Range(f2);
        fscanf(f2,"%d",&a2);
      }     }       while
 ( !file1 ) {
 fprintf(f,"%d ",a1);
 file1 =
 End_Range(f1);
      fscanf(f1,"%d",&a1);
    }      while ( !file2
) {       fprintf(f,"%d
",a2);       file2 =
End_Range(f2);
      fscanf(f2,"%d",&a2);
    }    }    file1 = file2 =
false;    while ( !file1 &&
!feof(f1) ) {
fprintf(f,"%d ",a1);
file1 = End_Range(f1);
fscanf(f1,"%d",&a1);
  }
  while ( !file2 &&
!feof(f2) ) {
fprintf(f,"%d ",a2);
file2 = End_Range(f2);
    fscanf(f2,"%d",&a2);
  }
  fclose(f2);
fclose(f1);
fclose(f);
```



```
 }
 remove("nmsort_1");
remove("nmsort_2");
}
//opredelenie kontsa bloka
bool End_Range (FILE
* f){  int tmp;  tmp =
fgetc(f);  tmp =
fgetc(f);
 if (tmp != '\') fseek(f,-
2,1);  else fseek(f,1,1);
 return tmp == '\' ? true : false;
}
```

Shunday qilib, tabiiy birlashtirish usulidan foydalanganda fayllarni o'qish yoki qayta yozish soni oddiy birlashtirish usulidan foydalangandan ko'ra yomonroq bo'lmaydi va o'rtacha hisobda yaxshiroq bo'ladi. Ammo bu usul ketma-ketlik oxiri belgisini tanib olish uchun zarur bo'lganlar hisobiga taqqoslashlar sonini oshiradi. Bundan tashqari, yordamchi fayllarning maksimal hajmi kirish faylining o'lchamiga yaqin bo'lishi mumkin, chunki seriya uzunligi ixtiyoriy bo'lishi mumkin.

Quyida Birlashtirib saralash (Merge Sort) algoritmining vaqt murakkabligi, xotira murakkabligi, va eng yaxshi, o‘rtacha, eng yomon holatlari bo‘yicha tahlil qilamiz. Shuningdek, algoritmni C++ tilida amalga oshirish misoli keltiriladi. Merge Sort algoritmining ishlash prinsipi:

Bo‘lish: Masalani kichik qismlarga bo‘lish (rekursiv tarzda 2 ga bo‘lish). Saralash: Har bir qismni alohida saralash.

Birlashtirish: Saralangan qismlarni birlashtirish orqali yakuniy tartiblangan ro‘yxat hosil qilish.

**Vaqt murakkabligi:**

Eng yaxshi holat (Best Case): Har doim massivni ikkiga bo‘lib, birlashtirish operatsiyalarini minimal darajada bajaradi. Vaqt murakkabligi: O(n·logn).

O‘rtacha holat (Average Case): Masalani o‘rtacha bo‘linish va birlashtirish bilan hal qiladi. Vaqt murakkabligi: O(n·logn).

Eng yomon holat (Worst Case): Bir xil bo‘linishlar bilan barcha elementlarni birlashtiradi. Vaqt murakkabligi: O(n·logn). **Xotira murakkabligi:**

Har bir bo‘linishda yangi yordamchi massivlar yaratiladi. Xotira murakkabligi: O(n).

**Merge Sort algoritmi (C++ da):**

```
#include
<iostream>
#include
<vector> using
namespace std;
// Massivlarni birlashtirish funksiyasi
void merge(vector<int>& arr, int left, int mid,
int right) {    int n1 = mid - left + 1; // Chap
massiv uzunligi    int n2 = right - mid;   // O'ng
massiv uzunligi
   vector<int> L(n1), R(n2); // Chap va o'ng yordamchi massivlar
   // Chap va o'ng qismlarni massivlarga ko‘chirish
   for (int i = 0; i < n1;
i++)       L[i] =
arr[left + i];    for (int j
= 0; j < n2; j++)
R[j] = arr[mid + 1 + j];
// Birlashtirish jarayoni
int i = 0, j = 0, k = left;
while (i < n1 && j <
n2) {       if (L[i] <=
R[j]) {          arr[k] =
L[i];          i++;       }
else {          arr[k] =
R[j];          j++;       }
k++;
```



```
   }
   // Qolgan elementlarni qo'shish
   while (i <
n1) {
arr[k] = L[i];
i++;
k++;
   }    while
(j < n2) {
arr[k] = R[j];
j++;
k++;
   }
}
// Rekursiv Merge Sort funksiyasi
void mergeSort(vector<int>& arr, int left, int
right) {    if (left < right) {
      int mid = left + (right - left) / 2; // O'rta indeksni topish
      // Bo'limlar bo'yicha saralash
mergeSort(arr, left, mid);
mergeSort(arr, mid + 1, right);
      // Birlashtirish
      merge(arr, left, mid, right);
   }
}
// Asosiy dastur
int main() {    vector<int> arr =
{12, 11, 13, 5, 6, 7};    cout <<
"Saralanmagan massiv: ";    for
(int x : arr) cout << x << " ";
cout << endl;    mergeSort(arr, 0,
arr.size() - 1);    cout <<
"Saralangan massiv: ";    for (int
```

```
x : arr) cout << x << " ";    cout
<< endl;    return 0; }
```

**Xulosa:**

Merge Sort algoritmi katta ma'lumotlarni tartiblash uchun samarali va barqaror bo'lib, asosan xotira hajmi muhim bo'lmagan holatlarda ishlatiladi. Bu algoritmning barqarorligi uni tezkor sortlash talab qilinadigan muammolar uchun ideal qiladi.

### 6.6. Qo'shimcha o'rganish uchun materiallar

Algoritmik saralash masalasi quyidagi ko'rinishda shakllantiriladi: **n** ta sondan tashkil topgan massiv berilgan. Massiv elementlari massivda o'sish (kamayish) tartibida saralangan holatda joylashtirish zarur. Juda ko'plam saralash algoritmlari mavjud bo'lib, ular turlicha murakkablikka ega – kvadratk $O(N^2)$ murakkablikdan, logirifmik $O(Nlog\ N)$ murakkablikkacha. Albatta, dasturlash bo'yicha olimpiadalarda muvaffaqiyatga erishish uchun ko'p sondagi saralash algoritmlarining mantig'ini tushunishib olish va STL kutubxonasining standart algoritmlaridan foydalana olish ko'nikmalariga ega bo'lish kerak.

STL kutubxonalarining standart saralash algoritmlari yordamida konteynerlarni saralash amalga oshiriladi. Bunda saralash algoritmlarini qo'llash uchun <algoritm> sarlavha faylini kiritish talab etiladi. Quyida saralash algoritmlari va partition algoritmlari ro'yxati keltirilgan, ularning mantig'ini tushunish tez saralashni o'rganishda zarur bo'ladi.

| Algoritm | Algoritm tavsifi |
|---|---|
| sort(it1, it2, Pred) | Ko'rsatilgan iteratorlar oralig'ida elementlarni o'sish tartibida yoki berilgan Pred peridikatida aniqlangan mezonga mos tartibda saralaydi. Algoritmning o'rtacha murakkabligi $O(NlogN)$. |
| stable_sort(it1, it2, Pred) | Saralanadigan elementlarning nisbiy tartibini saqlagan holda barqaror saralash yordamida elementlarni saralaydi. Algoritmning murakkabligi sort() algoritmining murakkabligidan kattaroq, eng yaxshi holatda $O(NlogN)$, eng yomon holatda $O(N(logN)^2)$. sort() dan sekinroq ishlaydi. |
| partition (it1, it2, Pred) | Iteratorlar oralig'idagi elementlarni ikkita kesishmaydigan qism to'plamlarga ajratadi. Ikkinchi qism to'plamning boshiga iteratorni qaytaradi. Ajratish algoritmining murakkabligi $O(N)$. |
| stable_partition (it1, it2, Pred) | Iteratorlar oralig'idagi elementlarni ikkita kesishmaydigan qism to'plamlarga ajratadi, massiv elementlarining nisbiy tartibini saqlab, unar predikatni qanoatlantirmaydigan elementlardan oldin ajratadi. Bu algoritmining murakkabligi $O(NlogN)$. |



Sanab o'tilgan algoritmlarning ishlashini sinab ko'rish va tasavvur qilish uchun a[16]=(5, 1, 9, 2, 0, 5, 7, 3, 4, 5, 8, 5, 5, 5, 10, 6) – massiv elementlariga ushbu algoritmlarning qo'llanishi quyidagi rasmlarda keltirilgan.

**Kirish massivi a[16]**

| 5 | 1 | 9 | 2 | 0 | 5 | 7 | 3 | 4 | 5 | 8 | 5 | 5 | 5 | 10 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**sort(a, a+16) algoritmi qo'llanilgan massiv**

| 0 | 1 | 2 | 3 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

partition (a,a+16,pr6) algorituni qo'llanilgan massiv, bu yerda pr6 – unar peridikat.

```
bool pr6(int val1){ return val1>6;}
```

| 10 | 8 | 9 | 7 | 0 | 6 | 2 | 3 | 4 | 5 | 1 | 5 | 5 | 5 | 5 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**stable_partition (a,a+16,pr6) algoritmi qo'llanilgan massiv**

| 9 | 7 | 8 | 10 | 5 | 1 | 2 | 0 | 5 | 3 | 4 | 5 | 5 | 5 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

6.25-rasm. Saralash sxemasi

Ba'zan elementlarni saralash bir nechta kalitlar bo'yicha amalga oshirish usullari qo'llaniladi. Misol uchun, agar sinf o'quvchilarining kodlangan ro'yxati ular yechimini olgan masalalar soni ko'rsatilgan bo'lsa va bu ro'yxatni yechilgan masalalar sonining kamayishiga qarab

tartiblash talab etilsa, unda beqaror saralash algoritmlari kirish ma'lumotlarini sezilarli darajada aralashtirishi mumkin. Ushbu ro'yxatni beqaror saralash algoritmi sort() va barqaror saralash algoritmi stable_sort() yordamida saralash holatlarini ko'rib chiqamiz. Ro'yxat yechilgan masalalar soni – kalit bo'yicha saralanishi kerak. Yuqorida ko'rib chiqilgan misoldan ko'rinib turibdiki, barqaror tartiblash kirish massividagi elementlarning nisbiy tartibini saqlab qoladi. STL kutubxonasining standart tartiblash algoritmlari natijalari o'rtasidagi farq: sort() tartiblash va barqaror tartiblash stable_sort() faqat kirish ma'lumotlarning katta massivlari qo'llanilishiga e'tibor bering.

| Kirish massivi | sort() qo'llanilgandan keyin | stable_sort() qo'llanilgandan keyin |
|---|---|---|
| Val0 6, Val1 6, Val2 4 | Val16 4, Val30 4, Val2 4 | Val2 4, Val5 4, Val6 4 |
| Val3 6, Val4 6, Val5 4 | Val29 4, Val27 4, Val5 4 | Val8 4, Val16 4, Val19 4 |
| Val6 4, Val7 6, Val8 4 | Val6 4, Val24 4, Val8 4 | Val21 4, Val22 4, Val24 4 |
| Val9 5, Val10 6, Val11 6 | Val19 4, Val22 4, Val21 4 | Val27 4, Val29 4, Val30 4 |
| Val12 5, Val13 6, Val14 5 | Val31 4, Val12 5, Val14 5 | Val31 4, Val9 5, Val12 5 |
| Val15 5, Val16 4, Val17 6 | Val15 5, Val32 5, Val18 5 | Val14 5, Val15 5, Val18 5 |
| Val18 5, Val19 4, Val20 6 | Val23 5, Val25 5, Val28 5 | Val23 5, Val25 5, Val28 5 |
| Val21 4, Val22 4, Val23 5 | Val9 5, Val10 6, Val17 6 | Val32 5, Val0 6, Val1 6 |
| Val24 4, Val25 5, Val26 6 | Val7 6, Val13 6, Val26 6 | Val3 6, Val4 6, Val7 6 |
| Val27 4, Val28 5, Val29 4 | Val4 6, Val20 6, Val3 6 | Val10 6, Val11 6, Val13 6 |
| Val30 4, Val31 4, Val32 5 | Val1 6, Val11 6, Val0 6 | Val17 6, Val20 6, Val26 6 |

6.26-rasm. Oddiy va barqaror saralash algoritmlarini taqqoslash

Sort() funksiyasi uchinchi parametr sifatida tartiblash mezoni (taqqoslovchi)ni qabul qilishi mumkin. Quyidagi list_fam tuzilmasining elementlarini vektorda saqlaydigan dastur kodi keltirilgan. Tuzilma ikkita ob'ektdan tashkil topgan elementlarni - talabaning familiyasi va baholarini tasvirlash imkonini beradi.

```
struct list_fam{
      string fam;
      int value;
};
```

Yuqorida tavsiflangan tuzilma elementlarini taqqoslash qandaydir saralash mezoni bo'lishi kerak. Misol uchun, elementlarni faqat o'quvchilarning baholari bo'yicha kamayish tartibida saralash uchun quyidagi taqqoslash mezonini kiritish mumkin:

```
bool comp (list_fam a, list_fam b){
return a.value>b.value;
```



```
}
```

Saralash algoritm bunday holatlarda ko'rsatilgan taqqoslash mezoni bilan chaqiriladi.

```
stable_sort(rating.begin(),rating.end(),comp);
```

Agar kamayish tartibida saralash talab qilingan bo'lsa, xususiy taqqoslash mezonini yozib, keyin saralashni amalga oshirish mumkin. Yoki oddiy saralashni amalga oshirgandan keyin konteyner elementlarini teskari tartibda joylashtiradigan reverse() metodini qo'llash mumkin bo'ladi. Shuningdek, rbegin(), rend iteratorlaridan foydalanish mumkin.

partition() algoritmi unar predikatida tavsiflangan mezon asosida massivni ikkita qism massivga ajratib olish asosida ishlaydi. Quyida keltirilgan misolda element 3 ga karrali bo'lmasa predikat true qiymatini qaytaradi. Partition() algoritmini qo'llanilgandan so'ng, massiv boshida 3 ga karrali bo'lmagan sonlar, keyin esa 3 ga karrali bo'lgan sonlar joylashadi. Shunday qilib, **a[]** massiv ikkita qism massivga ajratiladi – birinchisida {1, 2, 8, 4, 5, 7} sonlar, ikkinchisi {6, 3, 9} sonlari bo'ladi. partition() algoritmi ikkinchi qism massivning birinchi elementiga iteratorni qaytaradi.

```
int a[]={1, 2, 3, 4, 5, 6, 7, 8,
9}; bool comp (int l) { return
(l%3); } int main() {        int
*i;
    i = partition (a, a+9, comp);
      for (int *j=a; j!=a+9;++j)
      cout<<(*j)<<" "; cout<<endl;
for (int *j=a; j!=i;++j)
          cout<<(*j)<<" ";
```

Algoritmning murakkabligi O(N) ga teng, bu yerda N – massiv elementlari soni.

### 6.6.1. Hisoblash bilan saralash usuli

QuickSort saralashda elementlarni taqqoslash amali ishlatilgan. Hisoblash bilan Saralash (Counting Sort) taqqoslash amallaridan

foydalanmaydi va kichik oraliqdan qiymatlarni qabul qiluvchi diskret ma'lumotlar A (masalan, butun sonlar, belgilar) massivi uchun ishlatiladi $a_i = [0, K-1]$. Bu algoritm uchun C yordamchi massividan foydalaniladi, uning elementlari c[$i$] = $count$(A, $i$) , bu yerda $count$(A, $i$) - A massivdagi qiymati $i$ ga teng elementlar soni. C massivning maksimal o'lchami A massivning o'lchamiga teng. Quyida A massiv va C yordamchi massivga misol keltirilgan.

| a[i] | 2 | 5 | 3 | 0 | 2 | 3 | 0 | 3 |
|---|---|---|---|---|---|---|---|---|

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| c[i] | 2 | 1 | 3 | 2 | 2 | 3 | 2 | 3 |

6.27-rasm. Hisoblash bilan saralashga misol

C massiv orqali tartiblangan A massivni olish juda oson.

```
for (int i=0;i<n;++i)
        c[a[i]]++;
  int l=0;    for (int
i=0;i<k;++i){          for (int
j=1;j<=c[i];++j){
a[l]=i;
                    l++;
                    }
    }
```

Algoritm qo'shimcha ravishda O(K) xotirani ishlatadi va murakkabligi O(N+K) ga teng. Hisoblash bilan saralash algoritmi barqarorlik xossasini saqlab qoladi.

### 6.6.2. Razryad bo'yicha saralash (radix_sort)

Bu saralash algoritmining asosiy g'oyasi sonlar massivini bit qiymati bo'yicha saralashga asoslangan. Masalan, a[5]=(523, 153, 088, 554, 235) uch xonali sonlar massivini saralash talab qilingan bo'lsin. k=3 – sondagi xona birliklari (raqamlar) soni. Sonlar o'nlik sanoq sistemasida berilgan – m=10. Bundan tashqari, qayta ishlangan razryad raqami uchun 10 ta vektor talab qilinadi. Massivning sonlarini



razrayadlar vektoriga yozib olamiz: 0, 1, 2, ..., 9 kichik razradlar (birliklar xonasi) bo'yicha.

523, 153 – sonlari 3 raqamiga mos vektorga;

554 – soni 4 raqamiga mos vektorga;

235 – soni 5 raqamiga mos vektorga;

088 – soni 8 raqamiga mos vektorga yoziladi.

Yordamchi vektorlardan olingan sonlarni kirish vektorga joylashtiramiz. Natija: 523, 153, 554, 235, 088.

Natija massivdagi sonlarni ikkinchi razryad (o'nliklar xonasidagi) raqami bo'yicha yordamchi vektorga taqsimlab chiqamiz.

523 – soni 2 ga mos vektorga

235 – soni 3 ga mos vektorga

153 – soni 5 ga mos vektorga

554 – soni 5 ga mos vektorga

088 - soni 8 raqamiga mos vektorga yoziladi.

Yordamchi vektorlardan olingan sonlarni kirish vektorga joylashtiramiz. Natija: 523, 235, 153, 554, 088.

Keyin natija vektordagi sonlarni katta razryad bo'yicha vektorlarga qayta taqsimlaymiz

088

153

235

523

554

Yordamchi vektorlardan olingan sonlarni kirish vektorga joylashtiramiz.

Natija: 088, 153, 235, 523, 554. Bu saralash algoritmining murakkabligi O(kn+km) ga teng va qo'shimcha ravishda O(n+m) xotira talab qiladi.

## 6.7. Mustaqil ishlash uchun savollar va mashqlar

1. Saralash algoritmlarining xilma-xilligini qanday tushuntirish mumkin?

2. Nima uchun hozirda universal saralash algoritmi mavjud emas?

3. Barqaror (turg'un)lik va tabiiylik xususiyatlariga rioya qilish saralash algoritmining murakkabligiga qanday ta'sir qiladi?

4. Taqqoslash va o‘rnini almashtirish amallariini bajarishda tez saralash algoritmlari nima uchun afzallikka ega?

5. Quyidagi algoritmlardan qaysi biri deyarli tartiblangan massivlarda eng samarali hisoblanadi: binar piramidali saralash, birlashtirish bilan saralash, Shell saralash va Hoar saralash? Afzallikning sababi nimada?

6. Nima uchun tez saralash algoritmlari kichik o'lchamdagi massivlar uchun yuqori samara bermaydi?

7. Quyidagi saralash algoritmlarining bir-biriga nisbatan afzalliklari va kamchiliklari qanday: binar piramidali saralash, birlashtirish bilan saralash, Shell va Hoar saralash usullari?

8. Muammoni hal qilishda qaysi saralash algoritmiga ustunlik berish kerakligini qanday aniqlash mumkin?

9. Tashqi saralash algoritmlaridan foydalanishning sababi nimada?

10. Turli xil tashqi saralash algoritmlaridan foydalanganda tezkor xotira qanday sarflanadi?

11. Umumiy kalit bilan tartiblangan ikkita faylni birlashtirish uchun qaysi birlashtirish usuli samaraliroq: oddiy yoki tabiiy? Javobni asoslang.

12. Tashqi saralash algoritmlarining samaradorligini tahlil qilishda fazalar va yo'llar sonidan tashqari yana qanday omillarni hisobga olish kerak?

13. Muammoni hal qilishda qaysi tashqi samaraliroq algoritmiga ustunlik berish kerakligini qanday aniqlash mumkin?

14. STL kutubxonasining barqaror saralash va standart sort() saralash usullari natijalari orasidagi farqni aniqlash uchun test tuzing. Test, masalan, string va int turidagi ikkita kalitli yozuvlarni o‘z ichiga olishi mumkin.

15. Saralangan massiv: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 uchun tezkor saralash algoritmini tekshirib ko‘ring. Bu massivda algoritm tomonidan bajarilgan taqqoslash amallari sonini hisoblang.



16. 10 ta elementdan iborat massivga misollar keltiring, bunda tezkor saralash usuli kirish massivning eng yomon holatda saralanganligi kabi bir xil miqdordagi taqqoslashlarni amalga oshirishini sinovdan o‘tkazing.

17. O'sish tartibida saralangan ikkita butun sonli fayl berilgan. Ushbu ma'lumotlarga asoslangan uchinchi faylni yarating, u ham tartiblangan va kirish fayllari elementlari bo'yicha quyidagi amallar bajarilsin:

- birlashmasi (to'plamlarning kamida bittasiga tegishli sonni o'z ichiga oladi); - kesishmasi (har ikkala to'plamga tegishli sonlar);
- ayirmasi (birinchi to'plamga tegishli sonlar, lekin ikkinchisida mavjud bo‘lmagan);
- simmetrik ayirma (to'plamlar ayirmalarining birlashmasi).

18. N ta har xil uzunlikdagi kesmalar berilgan (N<=5000). Ushbu kesmalardan nechta uchburchak hosil qilish mumkinligini hisoblang.

19. O'lchami n ga teng bo‘lgan butun sonli kvadrat matritsa berilgan. Qiymatlarni quyidagicha saralang:

$a_{11}<=a_{12}<=...<=a_{1n}<=a_{21}<=a_{22}<=...<=a_{2n}<=...<=a_{n1}<=a_{n2}<=...<=a_{n}$

n.

20. Butun sonli massiv berilgan. Elementlar qiymatlari takrorlanmas ekanligini tekshirishni amalga oshiring. Takrorlanuvchi kirish qiymatlarini massivdan olib tashlang.

21. Oddiy birlashtirish bilan tashqi saralash algoritmining funksiyaga asoslanib, ikki tomonlama ikki fazali saralash algoritmidan foydalanuvchi dasturni yozing va sinab ko‘ring.

22. Berilgan funksiyadan foydalanib ikki tomonlama ikki fazali tabiiy birlashtirish bilan tashqi saralash algoritmini bajaruvchi dastur yozing va sinab ko‘ring.

23. Barcha mamlakatlarning to‘liq ro‘yxati berilgan, unga quyidagilar kiradi:
nomi, qit’asi, poytaxti, hududi, aholisi. Aholi sonining o'sishi bo'yicha ma'lum bir qit'aning shtatlari haqidagi ma'lumotlarni ko'rsating. Ikki tomonlama bir fazali oddiy birlashtirishdan foydalaning.

24. Kimyoviy moddalar haqida ma'lumot beriladi, unga quyidagilar kiradi: moddaning sinfi, moddaning nomi, moddaning molekulyar og'irligi. Belgilangan sinfning barcha moddalarini molekulyar og'irligi o'sib borish tartibida saralang. Ikki tomonlama ikki fazali tabiiy muvozanatli birlashtirish algoritmidan foydalaning.

25. Faylda o'zbekcha so'zlar ketma-ketligi mavjud. Uni alifbo tartibida saralang. Tashqi saralashdan foydalaning. E'tibor bering, o'zbek alifbosidagi harflar tartibi kodlash jadvalidagi harflar tartibiga mos kelmaydi.



# 7-BOB. QIDIRUV ALGORITMLARI

## 7.1. Umumiy tushunchalar

Aniq tuzilmaga ega bo'lgan ma'lumotlar ustida bajariladigan amallardan biri bu qidiruvdir. Qidiruv - bu oldindan yaratilgan ma'lumotlar to'plamidan aniq ma'lumotlarni topish jarayoni. Odatda, ma'lumotlar yozuvlar bo'lib, ularning har birida kamida bitta kalit mavjud. Qidiruv kaliti – yozuvning qidirilayotgan qiymat joylashgan maydoni. Kalitlar bir yozuvni boshqasidan ajratish uchun ishlatiladi. Qidiruvning maqsadi berilgan kalit qiymatiga ega bo'lgan barcha yozuvlarni (agar mavjud bo'lsa) topishdir.

Qidiruv jarayoni o'tkaziladigan ma'lumotlar tuzilmasini belgilar jadvali (nomlar jadvali yoki identifikatorlar jadvali) - kalitlar va ma'lumotlarni o'z ichiga olgan hamda ikkita amalni bajarishga ruxsat beruvchi tuzilma sifatida ko'rish mumkin. Bu amallar yangi elementni kiritish va berilgan kalit bilan elementni qaytarish amallaridir. Ba'zan belgilar jadvali so'zlarni alifbo tartibida saqlaydigan taniqli tizimga o'xshash lug'atlar deb ataladi: so'z - kalit, uning talqini - ma'lumotdir.

Qidiruv dasturlashda eng keng tarqalgan amallardan biridir. Ma'lumotlarning qanday tashkil etilishiga bog'liq bo'lgan juda ko'p turli xil qidiruv algoritmlari mavjud. Har bir qidiruv algoritmining o'ziga xos afzalliklari va kamchiliklari mavjud. Shuning uchun muayyan muammoni hal qilish uchun eng mos bo'lgan algoritmni tanlash muhimdir.

Ma'lum miqdordagi elementlardan iborat massiv sifatida tavsiflangan ma'lumotlar majmuasi berilgan bo'lsin. Berilgan kalitga mos qiymat massivda mavjudligini tekshirish talab qilingan. Agar qidirilayotgan qiymat massivda mavjud bo'lsa, u holda ushbu elementning indeksini, ya'ni berilgan kalitning (elementning) kirish massivida birinchi uchragan tartib raqamini aniqlash kerak.

**Qidiruv** – bu kompyuter xotirasida ma'lumotlarni qayta ishlash jarayonida qo'llaniladigan amallardan biri bo'lib, massiv elementlari orasidan berilgan elementga mos ma'lumot (element)ni aniqlash jarayonidir.

Barcha qidiruv algoritmlari quyidagi turlarga bo'linadi:

– saralangan ma'lumotlar tuzilmasidan qidirish uchun qo‘llaniladigan algoritmlar; – saralanmagan ma'lumotlar tuzilmasidan qidirish uchun qo‘llaniladigan algoritmlar.

Shunday qilib, ma'lumotlarni qidirishning umumiy algoritmi quyidagicha bo‘ladi:

1-qadam. Elementni hisoblash, bu ko'pincha elementning qiymatini, element kalitini va hokazolarni olishni talab qiladi.

2-qadam. Elementni kalit bilan solishtirish yoki ikkita elementni solishtirish (masalaning qo'yilishiga qarab).

3-qadam. To'plam elementlarini ko‘rib chiqish, ya'ni massiv elementlarini to‘liq qarab chiqish.

Turli xil qidiruv algoritmlarining asosiy g'oyalari qidiruv usullari va qidiruv strategiyasida jamlangan.

### 7.1.1. Ketma-ket qidiruv algoritmi

Ketma-ket (chiziqli) qidiruv – bu oldindan aniqlangan elementlar to‘plamidagi elementlar bilan berilgan kalit qiymatiga mos elementni ketma-ket taqqoslash yo‘li bilan qidirishning eng oddiy usuli hisoblanadi. Taqqoslash qidirilayotgan kalit qiymatga mos element topilmaguncha davom ettiriladi.

Bu usulning g‘oyasi quyidagicha: To‘plamning barcha elementlari ko‘rib chiqilmaguncha, ketma-ket ma’lum bir tartibda qarab chiqiladi (masalan, chapdan o‘ngga). Agar to‘plam elementlarini ko‘rib chiqishda qidirilayotgan element topilsa, qidiruv ijobiy natija bilan yakunlanadi (to‘xtatiladi). Agar to‘plamning barcha elementlari ko‘rib chiqilganda ham qidirilayotgan element topilmasa, algoritm salbiy neatija berishi kerak.

Chiziqli qidiruv algoritmi:

1-qadam. Siklning boshlang‘ich qiymati i=0.

2-qadam. Agar massivning x[i] elementi qiymati key kalit qiymatiga teng bo‘lsa, u holda qidirilayotgan elementning massivdagi tartib raqamini qaytarish va algoritm ishini yakunlash. Aks holda sikl o‘zgaruvchiga i=i+1 qiymatni berib, bir birlik oshirish.



3-qadam. Agar i<k bo‘lsa, 2-qadamni qayta bajarish, bu yerda k – x massivning elementlari soni, aks holda algoritm ishini yakunlash va -1 qiymatni qaytarish.

Massivda key qiymatli bir nechta elementlar mavjud bo‘lgan holatda, bu algoritm faqat birinchi uchraganini (eng kichik indekslisini) topadi.

```
int LinearSearch(int *x, int k,
int key){  int i = 0;  for (i=0;
i<k; i++)    if ( x[i] == key )
break;
 return i < k ? i : -1;
}
```

Bu algoritmning bajarilish vaqti haqiqiy sonlar uchun $n/\varepsilon$ ga teng, bu yerda $n$ – massivdagi elementlar soni, $\varepsilon$ – aniqlik. Butun soni massivlarda bajarilish vaqti, eng yomon holat uchun $n$ ga, o‘rtacha holat uchun $n/2$ga teng. Shunday qilib, vaqt bo‘yicha chiziqli qidiruv algoritmining murakkabligi O(n) bilan baholanadi. Chiziqli qidiruv algoritmi uchun kirish massivi elementlarining joylashish tartibi muhim emas.

Chiziqli qidiruv algoritmining kamchiligi eng yomon holatda massivni to‘liq ko‘rib chiqish bilan baholangan. Shuning uchun ham bu algoritm kichik sondagi elementlardan iborat to‘plamlarda qo‘llanilishi tavsiya etiladi.

Chiziqli qidiruv algoritmining afzalligi shundaki, uni tatbiq etish juda oson, qaralayotgan to‘plam elementlarini saralash, qo‘shimcha xotira va funksiyani qo‘shimcha tahlil qilish talab etilmaydi. Shuning uchun, istalgan manbadan to'g'ridan-to'g'ri ma'lumotlarni qabul qilishda oqim rejimida ishlashi mumkin.

Qidiruvni tezlashtiradigan chiziqli qidiruv algoritmining takomillashgan variantlari mavjud.

Ketma-ket qidiruv massiv kabi tashkil etilgan tuzilmaning barcha elementlarini ketma-ket qarab chiqishni talab qiladi. Chiziqli qidiruv algoritmi (Linear Search)ning ishlash prinsipi - har bir elementni ketma-

ket tekshirish orqali qidirilayotgan qiymatni topadi. Bu eng oddiy va intuitiv qidiruv usuli hisoblanadi.

Algoritmning *vaqt murakkabligi tahlili*:

- eng yaxshi holat (Best Case): Qidirilayotgan element massivning birinchi o'rnida joylashgan bo'lsa - $O(1)$.
- o'rtacha holat (Average Case): Element massivning o'rtasida joylashgan bo'lsa - $O(n/2)$, bu asosan $O(n)$ sifatida qaraladi.
- eng yomon holat (Worst Case): Element massivning oxirida yoki massivda yo'q bo'lsa - $O(n)$.

*Xotira murakkabligi*: Qo'shimcha xotira ishlatilmaydi - $O(1)$. Chiziqli qidiruvning C++ kod misoli:

```
#include
<iostream>
#include
<vector> using
namespace std;
int linearSearch(const vector<int>& arr, int target) {
  for (int i = 0; i < arr.size();
i++) {      if (arr[i] ==
target) {          return i; //
Element topildi
    }
  }
  return -1; // Element topilmadi
}     int
main() {
  vector<int> arr = {10, 20, 30, 40,
50};    int target = 30;
  int result = linearSearch(arr, target);
  if (result != -1) {
    cout << "Element " << target << " indeksda topildi: " << result
<< endl;
  } else {
    cout << "Element topilmadi." << endl;
```



```
  }
return
0;
}
```

### 7.1.2. Binar qidiruv algoritmi

Binar qidiruv – tartiblangan massivda amalga oshiriladi. Binar qidiruvda qidirilayotgan kalit massivdagi o'rta element kaliti bilan solishtiriladi. Agar ular teng bo'lsa qidiruv (eng yaxshi) yakunlanadi. Aks holda massivning chap yoki o'ng qismlarida qidiruv jarayoni davom ettiriladi.

Algoritm rekursiv yoki rekursiv bo'lmagan ko'rinishlarda bo'lishi mumkin.

Binar qidiruvni oraliqni teng ikkiga bo'lish usuli deb ham atash mumkin.

Qidiruvda qadamlar soni $\log_2 n \uparrow$ ga teng.

$n$ – elementlar soni;

$\uparrow$ - katta butun songa yaxlitlash.

Har bir qadamda o'rtasidan bo'lib qidiriladi va bunda quyidagi formuladan foydalaniladi:

$$mid = \frac{left - right}{2}$$

Agar qidirilayotgan element *mid* indeksli elementga teng bo'lsa, qidiruv yakunlanadi. Qidirilayotgan element *mid* elementdan kichik bo'lsa u holda bu qiymat *mid* dan oldin, aks holda (katta bo'lsa), *mid* qiymatdan keyin joylashtiriladi.

7.1-rasm. Binar qidiruv sxemasi

Binar qidiruv algoritmi:

1-qadam. Massivning o‘rta elementi indeksini aniqlash middle=(left+right)/2.

2-qadam. Agar qidirilayotgan element qiymati massivning o‘rta qiymatiga teng bo‘lsa, ushbu qiymat indeksini qaytarish va algoritm ishini yakunlash.

3-qadam. Agar qidirilayotgan element qiymati o‘rta qiymatdan katta bo‘lsa, u holda o‘rta qiymatdan o‘ng tomondagi barcha elementlarni, aks holda (kichik bo‘lsa) o‘rta qiymatdan kichik elementlarni olib, 1-qadamga qaytish.

Qidiruv qadamlar sonini kamaytirish uchun darhol qidiruv chegaralarini segmentning o‘rtasidan keyingi elementga o‘tkazish mumkin:

```
left = mid + 1
right = mid - 1
//binar qidiruv funksiyasi tavsifi
int BinarySearch(int *x, int k,
int key){  bool found = false;
int high = k - 1, low = 0;  int
middle = (high + low) / 2;
while ( !found && high >= low
){
  if (key ==
x[middle])     found
= true;    else if (key
< x[middle])     high
= middle - 1;    else
    low = middle + 1;
middle = (high + low) /
2;
 }
 return found ? middle : -1 ;
}
```

Binar qidiruv algoritmining ishlash jarayonida uning har bir qadamidan keyin qidiruv olib boriladigan maydon uzunligi 2 martaga



qisqarib boradi. Bu algoritmning ishlash murakkabligini $O(\log n)$ bilan baholashga asos bo‘ladi, bu yerda $n$ – to‘plam elementlari soni. Eng yaxshi holatda (Best Case): element birinchi taqqoslashda topilganda - $O(1)$. O‘rtacha holat (Average Case): har bosqichda massivning uzunligi yarmiga qisqaradi - $O(\log n)$. Eng yomon holat (Worst Case): Element massivning oxirida yoki massivda yo‘q - $O(\log n)$.

Xotira murakkabligi: iterativ usulda $O(1)$, rekursiv usulda funksiya chaqiruvlari uchun $O(\log n)$. Binar qidiruvning C++ kod misoli:

*Iterativ usul*:

```
#include
<iostream>
#include
<vector> using
namespace std;
int binarySearch(const vector<int>& arr, int target) {
  int left = 0, right = arr.size() - 1;
while (left <= right) {
    int mid = left + (right - left) / 2; //
Markazni topish      if (arr[mid] == target) {
return mid; // Element topildi      } else if
(arr[mid] < target) {
      left = mid + 1; // O'ng qismga o'tish
    } else {
      right = mid - 1; // Chap qismga o'tish
    }
  }
  return -1; // Element topilmadi
}     int
main() {
  vector<int> arr = {10, 20, 30, 40,
50};   int target = 40;
  int result = binarySearch(arr, target);
  if (result != -1) {
```

```
        cout << "Element " << target << " indeksda topildi: " << result
    << endl;
      } else {
        cout << "Element topilmadi." << endl;
      }
    return
    0;
    }
```

*Rekursiv usul*:

```
    #include
    <iostream>
    #include
    <vector> using
    namespace std;
    int binarySearchRecursive(const vector<int>& arr, int left, int
    right, int target)
{   if (left > right) {
        return -1; // Element topilmadi
      }
      int mid = left + (right - left) / 2; // Markazni
    topish    if (arr[mid] == target) {
        return  mid;  //  Element
    topildi    } else if (arr[mid] <
    target) {
        return binarySearchRecursive(arr, mid + 1, right, target); //
    O'ng qismga
o'tish
      } else {
        return binarySearchRecursive(arr, left, mid - 1, target); //
    Chap qismga
o'tish
    } } int
    main() {
      vector<int> arr = {10, 20, 30, 40, 50};
```



```
      int target = 50;
      int result = binarySearchRecursive(arr, 0, arr.size() - 1,
    target);    if (result != -1) {
        cout << "Element " << target << " indeksda topildi: " << result
    << endl;
      } else {
        cout << "Element topilmadi." << endl;
      }
    return
    0; }
```

### 7.1.3. Chiziqli va Binar qidiruv algoritmlari tahlili

Algoritmlar tahlili uchun namunaviy misol. Chiziqli va binar qidiruv algoritmlarini o‘zaro taqqoslash uchun har ikkala algoritmni C++ tilida dasturlash orqali murakkablikni taqqoslash uchun tajribaviy tahlil qilish.

**1: Chiziqli qidiruvni amalga oshirish**

```
    #include
    <iostream>
    #include
    <vector> using
    namespace std;
    int linearSearch(const vector<int>& arr, int target) {
      for (int i = 0; i < arr.size();
    i++) {      if (arr[i] ==
    target) {          return i; //
    Element topildi
        }
      }
      return -1; // Element topilmadi
    }      int
    main() {
      vector<int> arr = {15, 3, 7, 10, 20,
    8};    int target;
```

```
        cout << "Qidirilayotgan sonni
    kiriting: "; cin >> target;
        int result = linearSearch(arr, target);
        if (result != -1) {
            cout << "Element massivning " << result << "-indeksida
    topildi." <<
endl;
        } else {
            cout << "Element topilmadi." << endl;
        }
    return
    0;
    }
```

**2: Binar qidiruvni amalga oshirish**

```
#include
<iostream>
#include
<vector> using
namespace std;
int binarySearch(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
while (left <= right) {
        int mid = left + (right - left) / 2; //
O'rta indeks        if (arr[mid] == target) {
return mid; // Element topildi        } else if
(arr[mid] < target) {
            left = mid + 1; // O'ng qismga o'tish
        } else {
            right = mid - 1; // Chap qismga o'tish
        }
    }
    return -1; // Element topilmadi
}       int
main() {
```



```
    vector<int> arr = {3, 7, 10, 15, 20, 25}; // Oldindan tartiblangan
massiv    int target;
    cout << "Qidirilayotgan sonni
kiriting: "; cin >> target;
    int result = binarySearch(arr, target);
    if (result != -1) {
        cout << "Element massivning " << result << "-indeksida
topildi." <<
endl;
    } else {
        cout << "Element topilmadi." << endl;
    }
return
0; }
```

**3: Murakkablikni tajribaviy taqqoslashga doir misol.** Tartiblanmagan massivda chiziqli qidiruv va ushbu massivni tartiblash orqali binar qidiruvni tashkil qilish va ularning ishlashini taqqoslash. Massiv uzunligini oshirish, misol uchun, 100, 1000, 10000 va h.k. elementdan iborat massivlar yaratib, qidiruv uchun vaqtni o‘lchash (tajriba qilish). Buning uchun bitta dasturda ikkala algoritmni qo‘llash. Foydalanuvchidan massiv va qidiriladigan qiymatni qabul qilib, massivni tartiblash va binar qidiruvni ishlatish.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono> // Vaqt o'lchash
uchun using namespace std; using
namespace chrono; // Chiziqli
qidiruv
int linearSearch(const vector<int>& arr, int target) {
    for (int i = 0; i < arr.size();
i++) {        if (arr[i] == target)
return i;
    }
```

```
        cout << "Qidirilayotgan sonni
    kiriting: "; cin >> target;
        int result = linearSearch(arr, target);
        if (result != -1) {
            cout << "Element massivning " << result << "-indeksida
    topildi." <<
endl;
        } else {
            cout << "Element topilmadi." << endl;
        }
    return
    0;
    }
```

**2: Binar qidiruvni amalga oshirish**

```
#include
<iostream>
#include
<vector> using
namespace std;
int binarySearch(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
while (left <= right) {
        int mid = left + (right - left) / 2; //
O'rta indeks        if (arr[mid] == target) {
return mid; // Element topildi        } else if
(arr[mid] < target) {
            left = mid + 1; // O'ng qismga o'tish
        } else {
            right = mid - 1; // Chap qismga o'tish
        }
    }
    return -1; // Element topilmadi
}       int
main() {
```



```
    vector<int> arr = {3, 7, 10, 15, 20, 25}; // Oldindan tartiblangan
massiv    int target;
    cout << "Qidirilayotgan sonni
kiriting: "; cin >> target;
    int result = binarySearch(arr, target);
    if (result != -1) {
        cout << "Element massivning " << result << "-indeksida
topildi." <<
endl;
    } else {
        cout << "Element topilmadi." << endl;
    }
return
0; }
```

**3: Murakkablikni tajribaviy taqqoslashga doir misol.** Tartiblanmagan massivda chiziqli qidiruv va ushbu massivni tartiblash orqali binar qidiruvni tashkil qilish va ularning ishlashini taqqoslash. Massiv uzunligini oshirish, misol uchun, 100, 1000, 10000 va h.k. elementdan iborat massivlar yaratib, qidiruv uchun vaqtni o‘lchash (tajriba qilish). Buning uchun bitta dasturda ikkala algoritmni qo‘llash. Foydalanuvchidan massiv va qidiriladigan qiymatni qabul qilib, massivni tartiblash va binar qidiruvni ishlatish.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <chrono> // Vaqt o'lchash
uchun using namespace std; using
namespace chrono; // Chiziqli
qidiruv
int linearSearch(const vector<int>& arr, int target) {
    for (int i = 0; i < arr.size();
i++) {        if (arr[i] == target)
return i;
    }
```

```
    return -1;
}
// Binar qidiruv
int binarySearch(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
while (left <= right) {        int mid =
left + (right - left) / 2;        if
(arr[mid] == target) return mid;
else if (arr[mid] < target) left = mid +
1;        else right = mid - 1;
    }
    return -1;
}       int
main() {
    vector<int> arr = {15, 3, 7, 10, 20, 8,
25, 13};    int target = 20;    // Chiziqli
qidiruv    auto start =
high_resolution_clock::now();    int
linearResult = linearSearch(arr, target);
auto end =
high_resolution_clock::now();
    auto linearTime = duration_cast<microseconds>(end - start);
    // Tartiblash va binar qidiruv    sort(arr.begin(), arr.end());
start = high_resolution_clock::now();    int binaryResult =
binarySearch(arr, target);    end =
high_resolution_clock::now();    auto binaryTime =
duration_cast<microseconds>(end - start);    cout <<
"Chiziqli qidiruv: Topildi -> Indeks: " << linearResult
<< ", Vaqt: " << linearTime.count() << " mikrosekund." <<
endl;    cout << "Binar qidiruv: Topildi -> Indeks: " <<
binaryResult        << ", Vaqt: " << binaryTime.count() << "
mikrosekund." << endl;    return 0; }
```

Ushbu dasturda keltirilgan **<algorithm>** va **<chrono>** kutubxonalari quyidagicha ishlatiladi. **<algorithm>** kutubxonasidan



sort() funksiyasini ishlatish uchun foydalaniladi. sort() massivni yoki vektorni tartiblash uchun zarur, chunki binar qidiruv faqat tartiblangan massivlarda ishlaydi. Muqobil yondashuvi - agar tartiblash uchun boshqa qo'lda yozilgan algoritmdan foydalanishni xohlasangiz (masalan, Bubble Sort, Merge Sort, yoki Quick Sort), unda <algorithm> kutubxonasiga ehtiyoj qolmaydi. Quyida Bubble Sort misoli keltirilgan:

```
void bubbleSort(vector<int>&
arr) {    int n = arr.size();
for (int i = 0; i < n - 1; i++) {
for (int j = 0; j < n - i - 1; j++)
{          if (arr[j] > arr[j + 1])
{
            swap(arr[j], arr[j + 1]); // Elementlarni almashtirish
        }
    }
}
```

**Maslahat:** Algoritmlarning samaradorligini taqqoslash uchun sort() funksiyasini ishlatgan yaxshiroq, chunki u zamonaviy, tez va optimallashtirilgan tartiblash usulidan foydalanadi.

**<chrono>** kutubxonasi qidiruv algoritmlarining ishlash vaqtini o'lchash uchun ishlatiladi. high_resolution_clock, duration_cast, va microseconds kabi sinflar yordamida ikki vaqt nuqtasi orasidagi farq hisoblanadi. Muqobil yondashuv - agar vaqtni o'lchash shart bo'lmasa, <chrono> kutubxonasidan foydalanmasdan ham dastur ishlaydi. Vaqt o'lchashni olib tashlash uchun kodni quyidagicha o'zgartirish mumkin:

```
#include <iostream>
#include <vector> #include <algorithm>
using namespace std; // Chiziqli qidiruv int
linearSearch(const vector<int>& arr, int
target) {
    for (int i = 0; i < arr.size();
i++) {        if (arr[i] == target)
return i;
```

```
    }
    return -1;
}
// Binar qidiruv
int binarySearch(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
while (left <= right) {        int mid =
left + (right - left) / 2;        if
(arr[mid] == target) return mid;
else if (arr[mid] < target) left = mid +
1;        else right = mid - 1;
    }
    return
    -1;
}       int
main() {
    vector<int> arr = {15, 3, 7, 10, 20, 8,
25, 13};    int target = 20;    // Chiziqli
qidiruv
    int linearResult = linearSearch(arr, target);
    //   Tartiblash   va   binar   qidiruv
sort(arr.begin(), arr.end());
    int binaryResult = binarySearch(arr, target);    cout <<
"Chiziqli qidiruv: Topildi -> Indeks: " << linearResult << endl;
cout << "Binar qidiruv: Topildi -> Indeks: " << binaryResult
<< endl;    return 0; }
```

Agar maqsad faqat algoritmni amalda sinab ko‘rish bo‘lsa, har ikkala kutubxonadan foydalanmasdan ham ishlash mumkin, ammo dastur sodda holatda bo‘ladi. Ya’ni, <chrono> kutubxonasidan foydalanmasdan ham vaqtni baholash mumkin. Buning uchun platformaga xos vaqt o‘lchash usullaridan foydalaniladi. Masalan, C++ standarti bo‘yicha <ctime> kutubxonasidan foydalanilgan. Undagi clock() funksiyasi yordamida vaqtni o‘lchash mumkin. Misol:

```
#include <iostream>
```



```
#include <vector>
#include <algorithm>
#include <ctime> // Vaqtni o'lchash
uchun using namespace std; //
Chiziqli qidiruv
int linearSearch(const vector<int>& arr, int target) {
    for (int i = 0; i < arr.size();
i++) {        if (arr[i] == target)
return i;
    }
    return -1;
}
// Binar qidiruv
int binarySearch(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
while (left <= right) {        int mid =
left + (right - left) / 2;        if
(arr[mid] == target) return mid;
else if (arr[mid] < target) left = mid +
1;        else right = mid - 1;
    }
    return -1;
}       int
main() {
    vector<int> arr = {15, 3, 7, 10, 20, 8, 25,
    13}; int target = 20;
    // Vaqtni o'lchash: Chiziqli qidiruv
clock_t start = clock();
    int linearResult = linearSearch(arr, target);
clock_t end = clock();
    double linearTime = double(end - start) / CLOCKS_PER_SEC;
    // Tartiblash va binar
qidiruv    sort(arr.begin(),
arr.end());    start = clock();
```

```
    int binaryResult = binarySearch(arr,
target);    end = clock();
    double binaryTime = double(end - start) /
CLOCKS_PER_SEC;    cout << "Chiziqli qidiruv:
Topildi -> Indeks: " << linearResult        << ", Vaqt: "
<< linearTime << " soniya." << endl;    cout << "Binar
qidiruv: Topildi -> Indeks: " << binaryResult        <<
", Vaqt: " << binaryTime << " soniya." << endl;    return
0; }
```

Izoh: clock() funksiyasi dastur ichida o'tgan CPU vaqtini o‘lchaydi. CLOCKS_PER_SEC qiymati har bir sekunddagi soat tikanini bildiradi (odatda 1000 yoki 1 000 000 bo‘ladi).

Operatsion tizimga xos usullar - agar platformaga xos bo‘lgan kutubxonalar kerak bo‘lsa, tizimga mos usullarni ishlatish mumkin. Masalan, Windows uchun QueryPerformanceCounter va QueryPerformanceFrequency funksiyalaridan foydalaniladi. Linux uchun gettimeofday() funksiyasi yoki clock_gettime() ishlatiladi. Misol (Linux uchun gettimeofday):

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <sys/time.h> //
gettimeofday uchun using namespace
std; double
getTimeInMicroseconds() {    struct
timeval tv;    gettimeofday(&tv,
NULL);
  return tv.tv_sec * 1000000.0 + tv.tv_usec; // Mikrosekundlarda
vaqt
}    int
main() {
  vector<int> arr = {15, 3, 7, 10, 20, 8,
25, 13};    int target = 20;    // Chiziqli
qidiruv
```



```
    double start =
getTimeInMicroseconds();    int
linearResult = linearSearch(arr, target);
double end =
getTimeInMicroseconds();
    cout << "Chiziqli qidiruv vaqti: " << (end - start) / 1000000.0
<< " soniya."
<< endl;
    // Tartiblash va binar qidiruv
    sort(arr.begin(), arr.end());    start =
getTimeInMicroseconds();    int
binaryResult = binarySearch(arr,
target);
    end = getTimeInMicroseconds();
    cout << "Binar qidiruv vaqti: " << (end - start) / 1000000.0 << "
soniya."
<< endl;
    return
0; }
```

Qachon chrono, qachon ctime yoki OSga xos usul ishlatish kerak?

- <chrono>: Standartga mos, zamonaviy, eng tavsiya qilinadigan usul.
- <ctime>: Soddaroq va kichik loyihalar uchun ishlatilishi mumkin.
- OSga xos usullar: Tizim darajasida yuqori aniqlik talab qilingan hollarda.

Maslahat: Zamonaviy C++ dasturlashda iloji boricha <chrono> kutubxonasini ishlatish afzalroq.

## 7.2. Qidiruvning yaxshilangan algoritmlari

### 7.2.1. Interpolyatsion qidiruv

Navbatdagi o‘zgaruvchan o‘lchamlarni kamaytirish algoritmiga misol sifatida *interpolyatsion qidiruv* deb ataluvchi tartiblangan massivda qidiruv algoritmini ko‘rib chiqamiz. Binar qidiruvdan farqli

element jadval boshiga o‘tadi va keyingi ularga bo‘lgan murojaatlarda qisman tez topiladi.

Bu usul chiziqli qidiruv (linear search) algoritmining optimallashtirilgan varianti bo‘lib, tez-tez izlanadigan elementlarni keyingi qidiruvlar uchun *yaqinlashtirish maqsadi*da ishlatiladi.

Asosiy g‘oyasi - ma’lum element ko‘p marta izlansa, uni massiv boshiga yaqinlashtirish, keyingi qidiruvlarni tezlashtirish. Ya’ni, bu usulda, topilgan element kamida bir pozitsiyaga oldinga suriladi, ya’ni undan oldingi element bilan joyi almashtiriladi. C++ da Transpozitsion qidiruv:

```
#include <iostream>
using namespace std;
int transpositionSearch(int arr[], int n, int target) {
  for (int i = 0; i < n;
i++) {       if (arr[i] ==
target) {
      // Topilgan elementni avvalgisi bilan joyini
almashtirish          if (i != 0) {
          swap(arr[i], arr[i - 1]);
          return i - 1; // yangi pozitsiyasi
        }
        return i; // birinchi element bo‘lsa
      }
  }
  return -1; // topilmadi
} void printArray(int arr[],
int n) {   for (int i = 0; i <
n; i++)      cout << arr[i]
<< " ";
  cout << endl;
} int main() {   int arr[] =
{5, 8, 2, 4, 7, 10};   int n =
sizeof(arr) / sizeof(arr[0]);
```



```
cout << "Boshlang‘ich
massiv: ";
  printArray(arr,
n);   int target = 4;
  int pos = transpositionSearch(arr, n,
target);   if (pos != -1) {
    cout << "Element topildi, yangi indeks: " << pos <<
endl;      cout << "Yangi massiv holati: ";
    printArray(arr, n);
  } else {
    cout << "Element topilmadi.\n";
  }
return
0;
}
```

**Natija:**

Boshlang‘ich massiv: 5 8 2 4 7 10
Element topildi, yangi indeks: 2
Yangi massiv holati: 5 8 4 2 7 10

| Afzalliklari | Kamchiliklari |
|---|---|
| Tez-tez izlanadigan elementlar keyinroq tezroq topiladi | Har doim optimal emas |
| Oddiy va sodda implementatsiya | Faqat chiziqli qidiruvga asoslangan |
| Qidiruvdan keyin massiv o‘zgaradi | Ma’lumot tartibi buzilishi mumkin |

Transposition qidiruv - bu *adaptiv qidiruv algoritmi* bo‘lib, izlanayotgan elementlar takrorlansa, ularni yaqinlashtirib samaradorlikni oshiradi. Bu usul, ayniqsa, izlanadigan elementlar cheklangan va doimiy takrorlanuvchi tizimlarda foydalidir.

### 7.2.3. Boshiga ko‘chirish usuli

Bu usulda elementga bo‘lgan har bir murojaat uni jadvalning boshiga ko‘chirish bilan amalga oshadi. Natijada jadval boshida oxirida ishlatiladigan element paydo bo‘ladi. Boshiga ko‘chirish usuli – Move-to-Front Search (MTF) - bu qidiruvni tezlashtirish maqsadida ishlatiladigan oddiy adaptiv qidiruv algoritmi bo‘lib, tez-tez qidiriladigan elementlarni massivning boshiga olib chiqadi. Bu usul massivdagi elementlarning takroran qidirilishi ehtimoli mavjud bo‘lgan holatlarda samarali hisoblanadi.

G‘oyasi - massivda oddiy chiziqli qidiruv amalga oshiriladi. Izlanayotgan element topilgach: u massiv boshiga (0-indexga) olib chiqiladi. Qolgan boshqa elementlar o‘ngga siljitiladi.

**Algoritm qadamlari:**

1. Chiziqli qidiruv bilan elementni qidirish.
2. Element topilganda:
   - Elementni ajratib olish.
   - 0-indexga qo‘yish.
   - Qolgan elementlarni bir indeksga o‘ngga surish.

Masalan, massiv: [5, 2, 4, 9, 1] shaklida berilgan bo‘lsa, unga 4 elementni qidirish uchun yuqoridagi algoritmni qo‘llash natijasida: topilgan elementning indeksi 2 ga teng va 4 ni 0-indexga olib chiqiladi. Natija massiv: [4, 5, 2, 9, 1] Murakkabliklar:

| Ko‘rinishi | Murakkabligi |
|---|---|
| Vaqt (eng yomon) | $O(n)$ (chiziqli qidiruv va ko‘chirish) |
| Xotira | $O(1)$ (qo‘shimcha xotira talab qilinmaydi) |

Ushbu algoritmning C++ dagi kodi:

```
#include <iostream>
using namespace std;
int moveToFrontSearch(int arr[], int n, int target) {
```



```
    for (int i = 0; i < n;
i++) {        if (arr[i] ==
target) {            int temp
= arr[i];           // Chapga
surish           for (int j =
i; j > 0; j--) {
arr[j] = arr[j - 1];
        }
        arr[0] = temp; // Boshiga joylash
        return 0; // Endi boshida joylashdi
      }
    }
    return -1; // Topilmadi
} void printArray(int arr[],
int n) {    for (int i = 0; i <
n; i++)        cout << arr[i]
<< " ";    cout << endl;
} int main() {    int arr[] =
{5, 2, 4, 9, 1};    int n =
sizeof(arr) / sizeof(arr[0]);
cout << "Boshlang‘ich
massiv: ";
    printArray(arr,
n);    int target = 4;
    int pos = moveToFrontSearch(arr, n, target);
    if (pos != -1) {
      cout << "Element topildi va boshiga
ko‘chirildi.\n";       cout << "Yangi massiv holati: ";
      printArray(arr, n);
    } else {
      cout << "Element topilmadi.\n";
    }
return
```

0; }

**Natija:**

Boshlang‘ich massiv: 5 2 4 9 1

Element topildi va boshiga ko‘chirildi.

Yangi massiv holati: 4 5 2 9 1

| Afzalliklar | Kamchiliklar |
|---|---|
| Tez-tez izlanadigan elementlar tez topiladi | Massiv tartibi o‘zgaradi |
| Oddiy va samarali | Har doim optimal emas |
| Qo‘shimcha xotira kerak emas | Kam ishlatiladigan elementlar ortda qolaveradi |

Bu algoritm asosan, Kesh (cache), so‘zli qidiruv, ma’lumotlar tez-tez qayta ishlanadigan ilovalarda qo‘llaniladi.

### 7.2.4. O‘tishlar orqali qidirish algoritmi

O‘tishlar orqali qidirish (Jump Search) algoritmining g‘oyasi - bu chiziqli qidiruvga qaraganda tezroq, lekin binar qidiruvga qaraganda sekinroq ishlovchi algoritm bo‘lib, saralangan massivda ishlaydi. U ma’lumotlar oralig‘ida "sakrashlar (jump)" orqali harakatlanadi va kerakli element yaqinlashgach, chiziqli qidiruvga o‘tadi.

Ishlash prinsipi:

- massivdagi elementlar saralangan bo'lishi shart.
- massiv uzunligi $n$, va biz $m$ uzunlikda sakraymiz ($m = \sqrt{n}$ odatda eng optimal).
- har $m$ ta elementdan so‘ng to‘xtab, element izlangan qiymatdan katta bo‘lsa - orqaga qaytib, shu oraliqda chiziqli qidiruv amalga oshiriladi. Bosqichlar:
- bloklar bo‘yicha sakrash: massivda har $\sqrt{n}$ elementdan so‘ng tekshir.
- oraliq aniqlangach, o‘sha oraliqda oddiy chiziqli qidiruv bilan davom et.

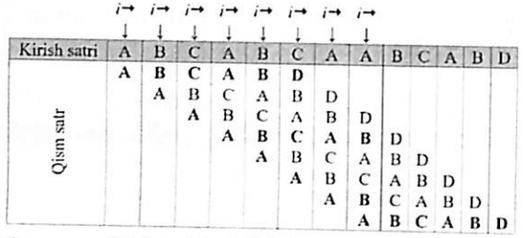


| Holat | Murakkablik |
|---|---|
| Eng yaxshi | $O(1)$ (birinchi urinishda topilsa) |
| O‘rtacha/eng yomon | $O(\sqrt{n})$ |
| Xotira | $O(1)$ (qo‘shimcha xotira kerak emas) |

C++ da algoritm kodi:

```
#include
<iostream>
#include
<cmath> using
namespace std;
int jumpSearch(int arr[], int n, int
target) {    int step = sqrt(n); //
sakrash hajmi    int prev = 0;
  // Sakrash orqali kerakli oraliqni
topish    while (arr[min(step, n) - 1] <
target) {
    prev = step;
step += sqrt(n);
if (prev >= n)
        return -1; // topilmadi
  }
  // Chiziqli qidiruv oraliqda
  for (int i = prev; i < min(step, n); i++) {
    if (arr[i] == target)
      return i;
  }
  return -1; // topilmadi
} int main() {    int arr[] = {1, 3, 5,
7, 9, 13, 17, 21, 25};    int n =
sizeof(arr) / sizeof(arr[0]);    int
target = 13;
```

Qism satrning bitta belgiga siljitilishi satr oxiriga yetib bormaguncha yoki qism satrning belgilari kirish satri belgilariga to‘liq mos kelmaguncha davom ettiriladi.




7.3-rasm. Chiziqli qidirish algoritmi ishlash sxemasi

```
//Satrdan qismsatrni chiziqli qidirish funksiyasi
int DirectSearch(char *string, char
*substring){  int sl, ssl;  int res = -1;
sl = strlen(string);  ssl =
strlen(substring);
 if ( sl == 0 )
  cout  <<  "Satr  noto‘g‘ri
berilgan\n";  else if ( ssl == 0 )
  cout << "Qism satr noto‘g‘ri
berilgan\n";  else  for (int i = 0; i
< sl - ssl + 1; i++)  for (int j = 0;
j < ssl; j++)  if ( substring[j] !=
string[i+j] )
      break;
    else if ( j == ssl - 1 ){
      res  =  i;
break;
    }
 return
res; }
```

Bu algoritm kam xarajatli va ma’lumotlarni oldindan ishlov berish va qo'shimcha joy talab qilmaydi. Chiziqli qidiruv algoritmidagi ko‘pchilik solishtirishlar ortiqcha bajariladi. Shuning uchun, eng yomon holatda, algoritm samarasiz bo'ladi, chunki uning murakkabligi O((n-



m+1)*m)ga proportsional bo'ladi, bu yerda n va m mos ravishda kirish satri va qismsatrlar uzunliklari.

### 7.3.2. Knut-Morris-Pratt algoritmi

Bu algoritm D. Knut va V.Pratt hamda ulardan mustaqil ravishda D.Morrislar tomonidan 1977-yilda taklif etilgan. Knut, Morris va Pratt (KMP)-algoritmi eng yomon holatda ham faqat O(n) ta solishtirishlarni talab qiladi. Ko'rib chiqilayotgan algoritm, qismsatrning boshlang'ich qismi kirish satrining mos keladigan belgilari bilan qisman mos tushgandan so'ng, kirish satrining ko‘rib chiqilgan qismini belgilab qo‘yish va uning yordamida ba'zi ma'lumotlarni hisoblash mumkinligi, shundan keyin tezda kirish satri bo‘ylab belgilab qo‘yilgan uzunlikda siljish mumkinligiga asoslanadi.

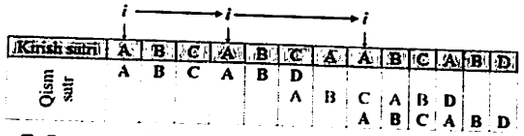
KPM-algoritmining chiziqli qidiruv algoritmidan asosiy farqi, har bir qadamda qismsatrni siljitish bittadan belgiga emas, balki bir nechta o‘zgaruvchan sondagi belgiga siljish mumkinligi bilan aniqlangan. Bundan kelib chiqadiki, navbatdagi siljishdan oldin, nechta belgiga siljish kerakligini aniqlash zarur bo‘ladi. Algoritm samaradorligini oshirish uchun har bir qadamda siljish imkon qadar katta bo'lishi kerak (7.4-rasm). Rasmda solishtirilgan elementlar qalin yozuv bilan berilgan.

Agar ixtiyoriy qismsatr uchun uning to‘liq bosh qismi bo‘lgan va bir vaqtning o'zida uning oxiri bo'lgan qismini aniqlab, ulardan eng uzunini tanlasak (albatta, satrning o'zini hisobga olmaganda), unda bunday protsedura odatda prefiks funksiyasi deb ataladi. Knut, Morris va Pratt algoritmini amalga oshirishda zarur bo'lgan qismsatni oldindan qayta ishlash qo'llaniladi, bunda prefiks funksiyasi ishlab chiqiladi. Buning uchun quyidagi g‘oya qo‘llaniladi: agar *i* uzunlikdagi satrning prefiksi bitta belgidan uzun bo‘lsa, u ham i-1 uzunlikdagi satrning prefiksi hisoblanadi. Shunday qilib, oldingi qismsatrning prefiksini tekshiramiz, agar u mos kelmasa, uning prefiksi va boshqalar. Shunday qilib, eng katta kerakli prefiksni aniqlab olamiz.

qismsatrning qaysidir (oxirgisi emas) belgisi kirish satridagi mos belgiga to‘g‘ri kelmasa, u holda bitta belgi o‘ng tomonga suriladi va yana qismsatrning oxirgi belgisidan boshlab solishtirish amallari bajariladi. Algoritm satrni to‘liq ko‘rib chiqmaguncha ishlashda davom etadi (7.5-rasm). Rasmda solishtirish amalga oshirilgan belgilar qalin yozuv bilan berilgan.

Oxirgi belgidagi nomuvofiqlik holatida siljish miqdori quyidagilarga asoslanib hisoblanadi: satrda qismsatrning paydo bo'lishini o'tkazib yubormaslik uchun qismsatrning siljishi minimal bo'lishi kerak. Agar qismsatrning solishtirilayotgan belgisi kirish satrida uchrasa, u holda qismsatrni kirish satrdagi ushbu belgining eng o'ng tomoniga mos keladigan tarzda siljitamiz. Agar kirish satrida bu belgi umuman bo'lmasa, u holda qismsatrni uning uzunligiga teng miqdorga siljitamiz, shunda qismsatrning birinchi belgisi kirish satridagi mos kelmagan belgidan keyingi belgisiga qo'yiladi.

Qismsatrning har bir belgisi uchun siljish qiymati faqat qismsatrdagi belgilar tartibiga bog'liq, shuning uchun siljishlarni oldindan hisoblash va ularni bir o'lchovli massiv sifatida saqlash qulay, bu yerda alifboning har bir belgisi qismsatrning oxirgi belgisiga nisbatan siljishga mos keladi.




7.5-rasm. BM-algoritmining ishlash sxemasi

```
//BM-algoritmi funksiyasi
int BMSearch(char *string, char
*substring){ int sl, ssl; int res = -1;
sl = strlen(string); ssl =
strlen(substring);
 if ( sl == 0 )
  cout << "Kirish satri noto‘g‘ri
berilgan\n"; else if ( ssl == 0 )
```



```
  cout << "Qismsatr noto‘g‘ri
berilgan\n"; else {
  int i, Pos;
int
BMT[256];
  for ( i = 0; i < 256; i ++ )
BMT[i] = ssl; for ( i = ssl-1; i >=
0; i-- ) if (
BMT[(short)(substring[i])] == ssl )
BMT[(short)(substring[i])] = ssl - i -
1;
  Pos = ssl - 1; while ( Pos < sl )
if ( substring[ssl - 1] != string[Pos] )
Pos = Pos +
BMT[(short)(string[Pos])]; else
    for ( i = ssl - 2; i >= 0; i-- ){ if (
substring[i] != string[Pos - ssl + i + 1] ) {
Pos += BMT[(short)(string[Pos - ssl + i + 1])] -
1; break; } else if ( i
== 0 ) return Pos - ssl + 1;
     cout << "\t" << i << endl;


} }
 return
res; }
```

Boyer va Mur algoritmi yaxshi satrlarda juda tez, yomon ma’lumotlar uchrash ehtimoli juda kam. Shuning uchun, qidiruv amalga oshiriladigan matnni oldindan qayta ishlashning iloji bo'lmagan hollardada ham optimal hisoblanadi. Shunday qilib, bu algoritm oddiy vaziyatlarda eng samarali hisoblanadi va uning ishlash unmdorligi qismsatr yoki alifbo ortishi bilan oshadi. Eng yomon holatda, ko'rib chiqilayotgan algoritmning murakkabligi O(m+n)ga teng.

"Yomon" holatlarda Knut, Morris va Pratt algoritmining o'ziga xos samaradorligini va "yaxshi" holatlarda Boyer va Mur algoritmining

samaradorligini birlashtirishga urinishlar bo‘lgan - masalan, turbo algoritmi, teskari Kolussi algoritmi va boshqalar.

Har bir qidiruv algoritmi o'zining masalalar sinfi uchun samarali ishlashga imkon beradi, turli xil tor yo'nalishlardagi takomillashtirishlar ham shunga yo‘naltirilganligi bilan ajralib turadi. Satrdagi qismsatrni qidirish algoritmi faqat dastur bajarishi kerak bo'lgan aniq masalaning qo‘yilishidan keyin tanlanishi kerak.

## 7.4. Ma’lumotlarni xeshlash algoritmlari

### 7.4.1.Xesh jadval va xesh funksiyalari

Oldingi bo‘limlarda biz uchta turdagi qidiruv usullarini ko'rib chiqdik: chiziqli qidiruv, ikkilik qidiruv va interpolyatsiya qidiruvi. Chiziqli qidiruv $O(n)$ vaqt murakkabligiga ega bo’lsa, ikkilik qidiruv $O(\log n)$ ga, bunda $n$ massivdagi elementlar soniga teng. Muhokama qilingan qidiruv algoritmlari samaralidir. Biroq, ularning qidirish vaqti massivdagi elementlar soniga bog'liq va ularning hech biri
$O(1)$ ga teng o’zgarmas vaqt ichida elementni qidira olmaydi. Biroq, chiziqli qidiruv, ikkilik qidiruv va boshqalar kabi barcha qidiruv algoritmlarida erishish juda qiyin, chunki bu algoritmlarning barchasi massivdagi elementlar soniga bog'liq. Bundan tashqari, oldingi qidiruv algoritmlari yordamida elementni qidirishda ko'plab taqqoslashlar amalga oshiriladi. Shuning uchun, bizning asosiy masala sifatida elementni o’zgarmas vaqt ichida qidirib topish va kamroq taqqoslashlarni bajarishga erishishdan iborat.

Faraz qilaylik, $N$ o'lchamdagi massiv berilgan va massivda saqlanadigan barcha qiymatlar takrorlanmas va 0 dan $N-1$ oralig'ida o’zgaradi. Shuning uchun yozuvlarni indeksi va qiymatlari bir xil bo'lgan qiymat asosida massivda saqlaymiz.
Shunday qilib, hech qanday asosiy taqqoslashlarsiz o’zgarmas vaqt ichida yozuvlarga kirish imkoni paydo bo’ladi. Buni quyidagi rasm yordamida batafsil tushunib olish mumkin:

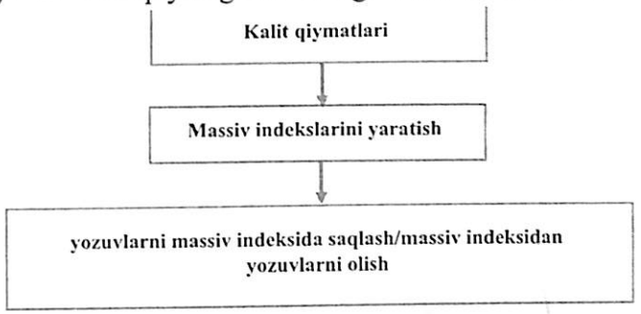
| arr[0] | arr[1] | arr[2] | arr[3] | arr[4] | arr[5] | arr[6] | arr[7] | arr[8] | arr[9] |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | | 3 | 4 | | 6 | | 8 | |

*7.6-rasm. Massiv*



7.6-rasmda beshta elementni o'z ichiga olgan massiv tasvirlangan. E'tibor bering, qiymatlar va massiv indeks raqamlari bir xil, ya'ni qiymat qiymati 3 bo'lgan yozuvga to'g'ridan-to'g'ri massivning indeksi $arr[3]$ bo’lgan elementi orqali kirish mumkin. Xuddi shunday, barcha yozuvlarga asosiy qiymatlar va massiv indeksi orqali kirish mumkin bo’ladi.

Shunday qilib, buni xeshlash orqali amalga oshirish mumkin, bu yerda qiymatni massiv indeksiga aylantiramiz va yozuvlarni massivda saqlaymiz. Buni quyidagicha amalga oshirish mumkin:

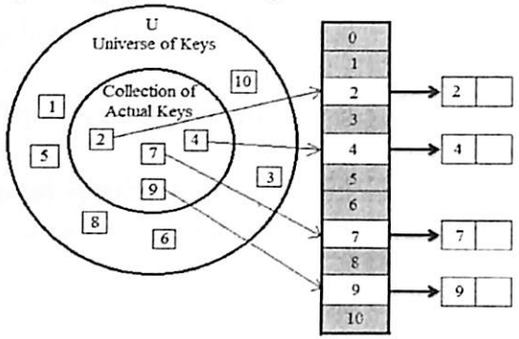



*7.7-rasm. Xeshlash yordamida massiv indekslarini yaratish.*

Massiv indekslarini yaratish jarayoni qiymatlarni massiv indeksiga aylantirish uchun ishlatiladigan ***xesh funksiyasi***dan foydalanadi. Bunday yozuvlar saqlanadigan massiv ***xesh-jadval*** deb nomlanadi. Oddiy hayotiy misol, biz lug'atdan so'zni qidirib, qiymat va uning indeksi yordamida ta'rifi yoki ma'nosini aniqlaymiz. Haydovchilik guvohnomasi raqamlari va sug'urta kartasi raqamlari hech qachon o'zgarmaydigan ma'lumotlardan, ya'ni tug'ilgan sana, ism va hokazolardan xeshlash yordamida yaratiladi.

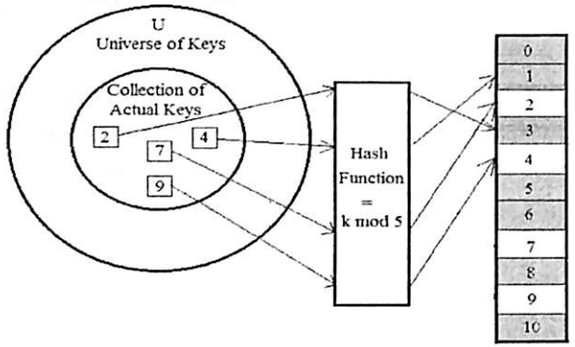
Xeshlash va to'g'ridan-to'g'ri adreslash o'rtasidagi farq. To'g'ridan-to'g'ri adreslashda qiymatni 7.8-rasmda ko'rsatilganidek, qiymatning qiymati bilan bir xil manzilda saqlanadi. Biroq, xeshlashda, 7.9-rasmda ko'rsatilganidek, qiymatning manzili *xesh funksiyasi* deb nomlanuvchi matematik funksiya yordamida aniqlanadi. Xesh funksiyasi qiymatning manzilini aniqlash uchun qiymatlar bilan ishlaydi. To'g'ridan-to'g'ri adreslash qiymatning xotira bo'ylab tasodifiy taqsimlanishiga olib

kelishi mumkin va shuning uchun ba'zida xeshlash bilan solishtirganda adreslashda ko'proq bo'sh joyni sarflashga olib keladi.




*7.8-rasm. To'g'ridan-to'g'ri adreslash usuli yordamida qiymatlarni joylashtirish (Qiymatlar fazosi va Haqiqiy qiymatlar to'plami)*

**Xesh-jadval** - bu samarali qidiruv usullaridan birini, ya'ni xeshlashni qo'llabquvvatlaydigan ma'lumotlar tuzilmasi hisoblanadi. Xesh-jadval - bu qiymat deb ataladigan maxsus indeks orqali ma'lumotlarga kirish mumkin bo'lgan massiv. Xeshjadvalda qiymatlar xesh funksiyasi yordamida massiv indekslari bilan taqqoslanadi. Xesh-funksiya - bu funksiya yoki matematik formula bo'lib, u qiymatga qo'llanilganda, xesh jadvalidagi qiymatni topish uchun indeks sifatida ishlatiladigan butun sonni hosil qiladi. Shunday qilib, xesh-jadvalda saqlangan qiymatni hesh funksiyasi yordamida $O(1)$ vaqtida qidirish mumkin. Xesh jadvalining asosiy g'oyasi massivning qiymatlari va indekslari o'rtasida to'g'ridan-to'g'ri moslikni o'rnatishdir.






*7.9-rasm.* Xeshlash yordamida Xesh-jadval bilan qiymatlarni moslashtirish

***Xesh funksiyasi*** matematik formula bo'lib, u qiymatga qo'llanilganda, xesh jadvalidagi qiymatni topish uchun indeks sifatida ishlatiladigan butun sonni hosil qiladi. Xesh funksiyasining xususiyatlari. Xesh funksiyalarining to'rtta asosiy xususiyati mavjud:

- xesh funksiyasi barcha kiritilgan ma'lumotlarni ishlatadi.
- xesh funksiyasi turli xil xesh qiymatlarini yaratishi kerak.
- xesh qiymati to'liq xeshlangan ma'lumotlar bilan aniqlanadi.
- xesh funksiyasi qiymatlarni butun xesh jadvali bo'ylab bir xilda taqsimlashi kerak.

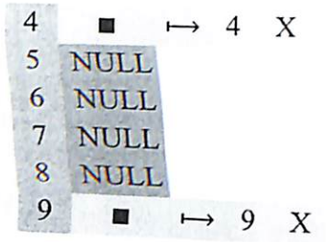
### 7.4.2. Xesh funksiyalarining turlari

**Bo'lish usuli.** Bo'lish usulida $k$ sonni $m$ ga bo'lganda qoldiqni olib, $k$ qiymati $m$ o‘lchamli xesh jadvalning mos yacheykalarning biriga joylashtiriladi. Oddiy qilib aytganda, bu usul butun sonni, aytaylik, $x$ ni $m$ ga bo'ladi, so'ngra olingan qoldiqdan foydalanadi. Bu xeshlashning eng oddiy usuli. Xesh funksiyasi quyidagicha ifodalanadi:

$$h(k) = k \bmod m$$

addres qiymat jadval o‘lchami

Misol uchun, agar $m = 5$ va qiymat $k = 10$ bo'lsa, $h(k) = 2$ bo‘ladi. Shunday qilib, bo'lish usuli juda tez ishlaydi, chunki u faqat bitta bo'lish operatsiyasini talab qiladi. Garchi bu usul $m$ ning istalgan qiymati uchun yaxshi bo'lsa-da, agar $m$ juft son bo'lsa, $k$ ning qiymati juft bo'lganda $h(k)$ juft bo'ladi va $k$ ning qiymati toq bo'lganda $h(k)$ toq bo'ladi. Shuning uchun, agar juft va toq sonlar deyarli teng bo'lsa, unda hech qanday muammo bo'lmaydi. Ammo agar juft qiymatlar soni ko'proq bo'lsa, bo'linish usuli yaxshi emas, chunki u qiymatlarni xesh jadvalida bir xilda taqsimlamaydi. Bundan tashqari, $m$ ning ma'lum qiymatlaridan qochamiz, ya'ni $m$ ning qiymati 2 ning darajasi bo'lmasligi kerak, chunki $h(k) =$

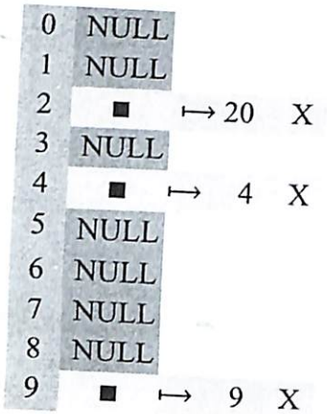



3-qadam: 20 ni jadvalga kiritish:

$$h(20) = 20\ mod\ 10$$

$$h(20) = 2$$

Demak 2-joy uchun bog'langan ro'yxatni yaratamiz va qiymat element 20 unda saqlanadi.

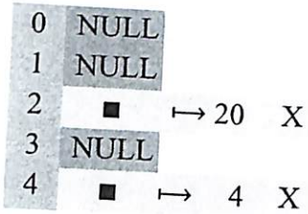



4-qadam: 35 ni jadvalga kiritish

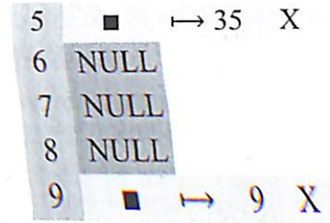
$$h(35) = 35\ mod\ 10\ h(35)$$

$$= 5$$

Demak 5-joy uchun bog'langan ro'yxatni yaratamiz va qiymat element 35 unda saqlanadi.





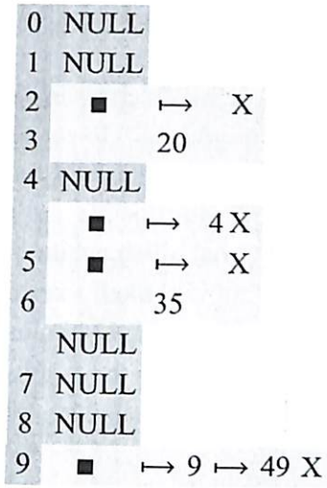




5-qadam: 49 ni jadvalga joylashtirish

$$h(49) = 49\ mod\ 10\ h(49)$$

$$= 9$$

Demak, 9-joyning bog'langan ro'yxatining oxiriga 49-ni qo'shamiz.




*Zanjirlash usulning afzalliklari va kamchiliklari.* Ushbu usulning asosiy *afzalligi* shundaki, u to‘qnashuv muammosini to'liq hal qiladi. Xesh-jadvalda saqlanishi kerak bo'lgan qiymat elementlar xesh jadvalidagi joylar sonidan ko’p bo'lsa ham, u samarali bo'lib qoladi. Biroq, qiymat elementlar sonining ko'payishi bilan ushbu usulning samaradorligi pasayishi aniq.

Ushbu usulning *kamchiligi* qiymat elementlar bog'langan ro'yxatda saqlanganligi sababli saqlash joyini ortiqcha sarflash hisoblanadi; qo'shimcha ravishda, kirish uchun har bir element uchun ko'rsatkichlar talab qilinadi, bu esa o'z navbatida ko'proq joy egallaydi.

***Ochiq adreslash usuli*** (Open Addressing Method). Ochiq adreslash usulida barcha elementlar xesh-jadvalning o'zida saqlanadi. Ushbu usulda ko'rsatgichlarni taqdim etishning hojati yo'q, bu usulning eng katta afzalligi. Bu usul to'qnashuv sodir bo'lganida yangi ma'lumotni qo'yish uchun boshqa manzilni qidirishni nazarda tutadi. Ya'ni, ma'lumot xesh jadvalida dastlabki hisoblangan joyga sig'magan taqdirda, boshqa bo'sh manzillar izlanadi va ma'lumot shu yerga joylashtiriladi. Open Addressing Method quyidagicha amalga oshiriladi:

1) Xesh funktsiyasi ma'lumot uchun xesh qiymatni hisoblaydi va shu xesh qiymatga muvofiq manzil (joy) aniqlanadi.

2) Agar bu manzil band bo'lsa (to'qnashuv), unda ma'lumot uchun yangi bo'sh manzil qidiriladi.

3) Yangilangan manzil izlash bir necha turli algoritmlar orqali amalga oshirilishi mumkin: chiziqli qidirish (linear probing), kvadratik qidirish (quadratic probing) yoki ikkilik xeshlash (double hashing).

Bu usul ma'lumotlar joylashtirilishida to'qnashuvlar bilan samarali kurashish va ularni tezkor hal qilish uchun qo'llaniladi.

**Chiziqli zondlash** (Linear Probing) **usuli** - xeshlashda to'qnashuv muammosini hal qilishning eng oddiy usuli. Ushbu usulda, agar qiymat $h(k)$ xesh funksiyasi tomonidan yaratilgan joyda allaqachon saqlangan bo'lsa, vaziyatni quyidagi xesh funksiyasi bilan hal qilish mumkin:

$$h'(k) = (h(k) + i)\ mod\ m$$

Bu yerda, $h(k) = k\ mod\ m$, $i = 0,1,2, \dots, (m-1)$.

Keling, ushbu usulniing ishlashini tekshirib ko'ramiz. Berilgan $k$ qiymat uchun avval $(h(k) + 0)\ mod\ m$ tomonidan hosil qilingan joylashuv tekshiriladi, chunki birinchi marta $i = 0$. Agar yaratilgan joy bo'sh bo'lsa, u holda qiymat unda saqlanadi. Aks holda, ikkinchi band qilish hesh funksiyasi $(h(k) + 1)\ mod\ m$ tomonidan berilgan $i = 1$ uchun hosil bo'ladi. Xuddi shunday, agar yaratilgan joy bo'sh bo'lsa, u holda qiymat unda saqlanadi; aks holda, $(h(k) + 2)\ mod\ m$, $(h(k) + 3)\ mod\ m$ va hokazo kabi bo'sh joy topilguncha keyingi zondlar hosil qilib boriladi.



**2-misol**. Berilgan $k = 13, 25,14, 35$ qiymatlarni chiziqli band qilish yordamida $m = 5$ o'lchamdagi xesh jadvaliga joylashtiring.

Xesh jadval dastlab quyidagicha bo'ladi:

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| NULL | NULL | NULL | NULL | NULL |

1-qadam: $i = 0$, 13 qiymatni joylashtirish

$$h'(k) = (k\ mod\ m + i)\ mod\ m$$
$$h'(13) = (13\%5 + 0)\%5$$
$$h'(13) = (3 + 0)\%5$$
$$h'(13) = 3\%5 = 3$$

$T[3]$ yacheyka bo'sh bo'lgani uchun, ushbu manziliga 13 qo'shiladi.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| NULL | NULL | NULL | 13 | NULL |

2-qadam: $i = 0$, 25 qiymatni joylashtirish

$$h'(25) = (25\%5 + 0)\%5$$
$$h'(25) = (0 + 0)\%5$$
$$h'(13) = 0\%5 = 0$$

$T[0]$ yacheyka bo'sh bo'lgani uchun, ushbu manziliga 25 qo'shiladi.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 25 | NULL | NULL | 13 | NULL |

3-qadam: $i = 0$, 14 qiymatni joylashtirish

$$h'(14) = (14\%5 + 0)\%5$$
$$h'(14) = (4 + 0)\%5$$
$$h'(14) = 4\%5 = 4$$

$T[4]$ yacheyka bo'sh bo'lgani uchun, ushbu manziliga 14 qo'shiladi.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 25 | NULL | NULL | 13 | 14 |

4-qadam: $i = 0$, 35 qiymatni joylashtirish

$$h'(35) = (35\%5 + 0)\%5$$
$$h'(35) = (0 + 0)\%5$$

$$h'(35) = 0\%5 = 0$$

$T[0]$ yacheyka oldindan band qilinganligi uchun, $i = 1$ holat uchun xesh qiymatni hisoblaymiz:

$$h'(35) = (35\%5 + 1)\%5$$
$$h'(35) = (0 + 1)\%5$$
$$h'(35) = 1\%5 = 1$$

$T[1]$ bo'sh, ushbu joyga 35 ni qo'shamiz. Natijada, xesh jadvalning oxirgi ko'rinishi quyidagicha bo'ladi:

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 25 | 35 | NULL | 13 | 14 |

Ushbu chiziqli band qilish usuli yordamida to'qnashuvni bartaraf etish dasturi:

```
// xesh_jadval_chiziqli_zondlash.cpp :
#include<iostream>
#include<conio.h>
#include<stdlib.
h> #define
SIZE 10 using
namespace std;
int arr[SIZE];

void insertion(int, int[]);
void lprob(int k, int
arr[SIZE]); void
display(int arr[SIZE]); int
main() {    int i, k, tanlash;
//clrscr();
     for (i = 0; i < SIZE; i++) {
            arr[i] = NULL;
      while (1) {
cout << "\n MENU"; cout << "\n
1. Qiymatni kiritish "; cout <<
"\n 2. Qiymatni qidirish "; cout
```



```
<< "\n 3. Qiymatlarni chiqarish ";
cout << "\n 4. Chiqish ";
cout << "\n\nQaysi amalni bajarasiz (1, 2, 3,4?)
:";    cin >> tanlash;
     switch (tanlash) {
case 1: cout<<"\nQiymat qiymatini kiriting:";
      cin >> k;
    if (k != -1) insertion(k, arr);
cout<<"\nMuvaffaqiyatli qo'shildi!!!";
      break;
      case 2: cout << "\nQidirish uchun qiymat qiymatini
kiriting : ";      cin >> k;  lprob(k, arr);     break;      case
3: display(arr);  break;      case 4: exit(0);
default:  cout  <<  "Noto'g'ri  tanlov";
      exit(0);
}
}
}
}
void insertion(int k, int
arr[SIZE]){      int position;
position = k % SIZE;          while
(arr[position] != NULL){
position = ++position % SIZE;
}
arr[position] = k;
}
void lprob(int k, int
arr[SIZE]) {      int
position;   position = k %
SIZE;
  while ((arr[position] != k) && (arr[position] != NULL)) {
    position = ++position % SIZE;
      }
```

```
    if (arr[position] != NULL)
   cout << "\nMuvaffaqiyatli qidiruv : " << position;
     else
    cout<<"\nMuvaffaqiyatsiz qidiruv";
}
void display(int arr[SIZE]) {
     int i;
     cout << "\n Qiymatlar ro'yxati :
\n"; for (i = 0; i < SIZE; i++)
          cout << "\t" << arr[i];
}
```

Dastur ishi natijasi:

```
MENU
1. Qiymatni kiritish
2. Qiymatni qidirish
3. Qiymatlarni chiqarish
4. Chiqish
Qaysi amalni bajarasiz (1, 2, 3,4?) :1
Qiymat qiymatini kiriting : 25
Muvaffaqiyatli qo'shildi!!!
 MENU
1. Qiymatni kiritish
2. Qiymatni qidirish
3. Qiymatlarni chiqarish
4. Chiqish
Qaysi amalni bajarasiz (1, 2, 3,4?) :2
Qidirish uchun qiymat qiymatini kiriting : 25
Muvaffaqiyatli qidiruv, indexsi : 5
 MENU
1. Qiymatni kiritish
2. Qiymatni qidirish
3. Qiymatlarni chiqarish
4. Chiqish
```



```
Qaysi amalni bajarasiz (1, 2, 3,4?) :3
Qiymatlar ro'yxati :
    0    0    0    0    0    25    0    0    0    0
 MENU
1. Qiymatni kiritish
2. Qiymatni qidirish
3. Qiymatlarni chiqarish
4. Chiqish
```

*Chiziqli zondlashning afzalliklari va kamchiliklari. Afzalliklari*:

1. Tashkil qilish soddaligi: Chiziqli zondlashning logikasi juda sodda, u faqat keyingi manzillarni bosqichma-bosqich tekshirib boradi. Buning natijasida uni amalga oshirish oson va oddiy.

2. Xotira samaradorligi: Har bir ma'lumot shu jadvalning o'zida saqlanadi, ya'ni qo'shimcha yordamchi tuzilmalar talab qilinmaydi, bu hotiradan samarali foydalanish imkonini beradi.

3. To'g'ridan to'g'ri yozish va o'qish: To'qnashuvni hal qilish uchun yangi ma'lumotni qo'shish va unga murojaat qilish bir manzildan boshlab ketma-ketlikda amalga oshiriladi, bu ba'zi hollarda qidirishni tezlashtirishi mumkin.

*Kamchiliklari*:

1. Birlashish muammosi (clustering): Chiziqli zondlashning eng katta kamchiligi shundaki, yaqin manzillarda to'qnashuvlar ko'payishi mumkin va birlashuv hodisasi yuzaga keladi. Agar bir necha qator ma'lumotlar bir xil joylarda to'qnashsa, ularning ketma-ketligi hosil bo'ladi va bu keyingi qidirish va kiritish jarayonlarini sekinlashtiradi.

2. To'qnashuvni hal qilish samaradorligi pasayishi: Katta hajmdagi ma'lumotlar bilan ishlaganda to'qnashuvlar soni ortishi mumkin. Bu esa har bir yangi qidiriladigan yoki qo'shiladigan element uchun katta vaqt talab qiladi.

3. Zondlash zanglashi: Jadvalda bo'sh manzillar kamayganda, chiziqli zondlash uchun to'qnashuvlarni hal qilish vaqti ortgani uchun

zondlash jarayoni juda sekinlashishi mumkin. Bu muammo katta zondlash zanglashini keltirib chiqaradi va usulni samarasiz qiladi.

4. Yarim to'lgan jadvalda ham muammolar: Xesh jadvali yarim to'lgan bo'lsa ham, to'qnashuvlar sababli ma'lumotlarni kiritish va qidirish jarayonlari sekinlashishi mumkin. Buning sababi manzillarni bosqichma-bosqich qidirish zaruratidir.

Chiziqli zondlash (linear probing) afzalliklari va kamchiliklariga qarab, u oddiy va samarali usul hisoblanadi, lekin jadvaldagi to'qnashuvlar va jadval hajmining o'lchamiga bog'liq holda uning samaradorligi pasayishi mumkin.

**Kvadrat zondlash** - xeshlashda kollizuya muammosini hal qilishning yana bir usuli. Ushbu usulda, agar qiymat $h(k)$ xesh funksiyasi tomonidan yaratilgan joyda oldindan qiymat mavjud bo'lsa, vaziyatni quyidagi xesh funksiyasi bilan hal qilish mumkin:

$$h'(k) = (h(k) + c_1 i + c_2 i^2) mod\ m$$

Bu yerda, $h(k) = k\ mod\ m$; $c_1$, $c_2$ - 0 ga teng bo'lmagan konstantalar, $i = 0,1,2 \dots, (m-1)$ - zondlashdagi qadamlar.

Kvadrat zondlash (quadratic probing) - xeshlash jadvallarida to'qnashuvni (collision) hal qilish usullaridan biri bo'lib, u chiziqli zondlashdan farqli o'laroq, manzillar ketma-ket emas, balki kvadrat formula asosida aniqlanadi. Bu usul to'qnashuv yuz berganda, ma'lumotni qo'yish uchun keyingi bo'sh manzilni aniqlashda kvadratli funksiyalardan foydalanadi.



Misol: $k = 25,13,14,35$ qiymatlari berilgan, ushbu qiymatlarni $m = 5$ bo'lgan xesh jadvalga joylashtiring. Kvadratli zondlash usulidan foydalaning, bunda: $c_1 = 1$ va $c_2 = 3$ bo'lsin. Xesh jadvalning dastlabki holati:

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| NULL | NULL | NULL | NULL | NULL |

1-qadam: $i = 0$, $c_1 = 1$ va $c_2 = 3$, jadvalga joylashtiriladigan birinchi qiymat=25.

$$h'(k) = (k\ mod\ m + c_1 \cdot i + c_2 \cdot i^2)\ mod\ m$$

$$h'(25) = (25\%5 + 1 \cdot 0 + 3 \cdot 0^2)\%5$$

$$h'(25) = (0 + 0)\%5$$

$$h'(25) = 0\%5 = 0$$

$T[0]$ bo'sh bo'lgani uchun 25 qiymati 0 indeksga joylashadi

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 25 | NULL | NULL | NULL | NULL |

2-qadam: $i = 0$, $c_1 = 1$ va $c_2 = 3$ , jadvalga joylashtiriladigan keyingi qiymat=13

$$h'(13) = (13\%5 + 1 \cdot 0 + 3 \cdot 0^2)$$

$$\%5\ \ h'(13) = (3 + 0)\%5$$

$$h'(13) = 3\%5 = 3$$

$T[3]$ bo'sh bo'lganligi uchun 13 qiymat 3-indeksga joylashtiriladi.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 25 | NULL | NULL | 13 | NULL |

3-qadam:$i = 0$, $c_1 = 1$ va $c_2 = 3$ , jadvalga joylashtiriladigan keyingi qiymat=14

$$(14) = (14\%5 + 1 \cdot 0 + 3 \cdot 0^2)$$

$$h' \qquad h'(14) = (4 + 0)\%5 \qquad \%5$$

$$h'(14) = 4\%5 = 4$$

$T[4]$ bo'sh bo'lganligi uchun 14 qiymat 4-indeksga joylashtiriladi.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 25 | NULL | NULL | 13 | 14 |

4-qadam: $i = 0$, $c_1 = 1$ va $c_2 = 3$, jadvalga joylashtiriladigan keyingi qiymat=35

$$(35) = (35\%5 + 1 \cdot 0 + 3 \quad h' \quad \cdot 0^2)\%5$$
$$h'(35) = (0 + 0)\%5$$
$$h'(35) = 0\%5 = 0$$

$T[0]$ bo'sh emas, shuning uchun kvadratli zondalashning keyingi qadamiga bo'sh joyni qidirishga o'tamiz, ya'ni $i = 1$ holatida xesh qiymatni qayta hisoblaymiz. $i = 1$

$$h'(35) = (35\%5 + 1 \cdot 1 + 3 \cdot 1^2)\%5$$
$$h'(35) = (0 + 1 + 3)\%5$$
$$h'(35) = 4\%5 = 4$$

Bu qiymat uchun ham $T[4]$ bo'sh emas, shuning uchun $i = 2$ holat uchun qayta hisoblashni bajaramiz.

$$h'(35) = (35\%5 + 1 \cdot 2 + 3 \cdot 2^2)\%5$$
$$h'(35) = (0 + 2 + 12)\%5$$
$$h'(35) = 14\%5 = 4$$

Bu qiymat uchun ham $T[4]$ bo'sh emas, shuning uchun $i = 3$ holat uchun qayta hisoblashni bajaramiz:

$$h'(35) = (35\%5 + 1 \cdot 3 + 3 \cdot 3^2)\%5$$
$$h'(35) = (0 + 3 + 27)\%5$$
$$h'(35) = 30\%5 = 0$$

Bu qiymat uchun ham $T[0]$ bo'sh emas, shuning uchun $i = 4$ holat uchun qayta hisoblashni bajaramiz:

$$h'(35) = (35\%5 + 1 \cdot 4 + 3 \cdot 4^2)\%5$$
$$h'(35) = (0 + 4 + 48)\%5$$
$$h'(35) = 52\%5 = 2$$

$T[2]$ bo'sh, 35 qiymati 2-indeksga joylashtiriladi. Natijada barcha qiymatlarning xesh qiymatlari to'liq hisoblab chiqildi va xesh jadvalga joylashtirildi:

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 25 | NULL | 35 | 13 | 14 |

Kvadratli zondlash usuli uchun dastur kodi.

// hash_kvadrat_zondlash.cpp :



```
#include
<iostream>
#include <vector>
using namespace
std; class
HashTable {
private:
vector<int> table;
int tableSize;
  int c1, c2;
  // Xesh funksiyasi: (k%m+c1*i+c2*i^2)%m
  int hashFunction(int key, int i) {
    return (key % tableSize + c1 * i + c2 * i * i) % tableSize;
  }

public:
  HashTable(int size, int c1, int c2) : tableSize(size), c1(c1),
c2(c2) {      table.resize(size, -1);
  }
  // Qiymatni joylashtirish funktsiyasi
void insert(int key) {
    int i = 0;
int index;
    // To'qnashuv holatida kvadrat zondlashdan
foydalanish      do {
      index = hashFunction(key, i);
      if (table[index] == -1) { // Joy bo'sh bo'lsa, qiymatni
joylashtiramiz          table[index] = key;
        cout<<key<<"qiymat      "<<index<<"      indeksga
joylashtirildi"<<endl;          return;
      }
i++;
    } while (i < tableSize);
```

```
        cout << "Jadval to'la. "<< key << " qiymat joylashtirilmadi"
  << endl;
    }
    // Xesh jadvalni ko'rsatish funksiyasi
  void display() {
        for (int i = 0; i < tableSize; i++) {
          if (table[i] != -1)
            cout << i << " --> " << table[i] <<
  endl;          else
            cout << i << " --> " << "Bo'sh" << endl;
        }
    } }; int main()
  {    int tableSize
  = 5;    int c1 =
  1;    int c2 = 3;
    HashTable hashTable(tableSize, c1, c2);
    // Qiymatlarni
  joylashtirish
  hashTable.insert(25);
  hashTable.insert(13);
  hashTable.insert(14);
  hashTable.insert(35);    //
  Xesh jadvalni ko'rsatish
  hashTable.display();
   return 0; }
```

Natija:

```
25 qiymat 0 indeksga joylashtirildi
13 qiymat 3 indeksga joylashtirildi
14 qiymat 4 indeksga joylashtirildi
35 qiymat 2 indeksga joylashtirildi
0 --> 25
1 --> Bo'sh
2 --> 35
3 --> 13
```



```
4 --> 14
```

*Kvadrat zondlashning afzalliklari va kamchiliklari.* Yuqorida aytib o'tilganidek, kvadratik zondlashning eng katta afzalliklaridan biri shundaki, u birlamchi klasterlash hodisasini yo'q qiladi. Shunga qaramay, ushbu usulning asosiy kamchiliklaridan biri shundaki, ketma-ket zondlar ketma-ketligi faqat xesh jadvalining ma'lum bir qismini qamrab olishi mumkin va bu qism juda kichik bo'lishi mumkin. Shuning uchun, agar bunday vaziyat yuzaga kelsa, jadval to'liq bo'lmaganiga qaramay, biz uchun hash jadvalida bo'sh joyni topish qiyin bo'ladi. Demak, kvadratik zondlash ikkilamchi klasterlash deb nomlanuvchi muammoga duch keladi. Ushbu usulda, xesh jadvali to'la bo'lganda, bir nechta to'qnashuvlar ehtimoli ortadi. Ushbu turdagi vaziyatni qo‘sh xeshlash orqali bartaraf etish mumkin.

***Qo‘sh xeshlash usuli.*** Qo‘sh xeshlash ochiq manzillashning eng yaxshi usullaridan biri. Nomidan ko'rinib turibdiki, bu usul bitta xesh funksiyasidan ko'ra ikkita xesh funksiyasidan foydalanadi. Xesh funksiyasi quyidagicha berilgan: $h'(k) = (h_1(k) + i * h_2(k)) mod\ m$,

bu yerda $h_1(k) = k\ mod\ m$ va $h_2(k) = k\ mod\ m'$ ikkita xesh funksiyasi, $m$ - xesh jadvalining o'lchami, $m'$ - $m$ dan kichik $(m-1)$ yoki $(m-2)$ bo'lishi mumkin), $i$ - 0 dan $(m-1)$ gacha o'zgarib turadigan zond raqami.

Berilgan $k$ qiymat uchun avval $h_1(k)\ mod\ m$ tomonidan hosil qilingan joylashuv tekshiriladi, chunki birinchi marta $i = 0$. Agar yaratilgan joy bo'sh bo'lsa, u holda qiymat unda saqlanadi. Aks holda, keyingi zondlar oldingi joydan $h_2(k)\ mod\ m$ jadvalda joylashgan joylarni hosil qiladi. Shuningdek, jadval ikkinchi xesh funksiyasi tomonidan yaratilgan qiymatga qarab har bir zond bilan farq qilishi mumkin, ya'ni $h_2(k)\ mod\ m$. Natijada, qo‘sh xeshlashning ishlashi bir xil xeshlashning "ideal" sxemasini bajarishga juda yaqin.

**3-misol.** Berilgan $k = 71,29,38,61,100$ qiymatlari, bu qiymatlarni qo‘sh xeshlash yordamida $m = 5$ o‘lchamdagi xesh jadvaliga solishtiring.

$h_1 =$

($k \ mod \ 5$) va $h_2 = (k \ mod \ 4)$ bo'lsin. Dastlab, xesh jadvali quyidagicha berilgan:

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| NULL | NULL | NULL | NULL | NULL |

1-qadam: $i = 0$, qiymat=71

$$h'^{(k)} = (h_1(k) + i * h_2(k)) mod \ m$$

$h'(k) = (k \ mod \ m + (i * k \ mod \ m')) mod \ m \ h'(71) = (71\%5 + (0 * 71\%4))\%5 \ h'(71) = (1 + (0 * 3))\%5 \ h'(71) = 1\%5 = 1$

$T[1]$ bo'sh, 71 ni joylashtiramiz.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| NULL | 71 | NULL | NULL | NULL |

2-qadam: $i = 0$, qiymat= 29

$$h'(29) = (29\%5 + (0 * 29\%4))\%5$$
$$h'(29) = (4 + (0 * 1))\%5$$
$$h'(29) = 4\%5 = 4$$

$T[4]$ bo'sh, 29 ni joylashtiramiz.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| NULL | 71 | NULL | NULL | 29 |

3-qadam: $i = 0$, qiymat= 38

$h'(38) = (38\%5 + (0 * 38\%4))\%5 \ h'(38) = (3 + (0 * 2))\%5 \ h'(38) = 3\%5 = 3$

$T[3]$ bo'sh, 38 ni joylashtiramiz.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| NULL | 71 | NULL | 38 | 29 |

4-qadam: $i = 0$, qiymat= 61

$h'(61) = (61\%5 + (0 * 61\%4))\%5 \ h'(61) = (1 + (0 * 1))\%5 \ h'(61) = 1\%5 = 1$



$T[1]$ bo'sh emas, zondalsha qadami $i = 1$ uchun hisoblashni davom ettiramiz:

$$h'(61) = (61\%5 + (1 * 61\%4))\%5$$
$$h'(61) = (1 + (1 * 1))\%5$$
$$h'(61) = (1 + 1)\%5$$
$$h'(61) = 2\%5 = 2$$

$T[2]$ bo'sh, 61 ni joylashtiramiz

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| NULL | 71 | 61 | 38 | 29 |

5-qadam: $i = 0$, qiymat = 100,

$h'(100) = (100\%5 + (0 * 100\%4))\%5 \ h'(100) = (0 + (0 * 0))\%5 \ h'(100) = 0\%5 = 0$

$T[0]$ bo'sh, 100 ni joylashtiramiz. Barcha qiymatlar xesh-jadvalga joylashdi:

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 100 | 71 | 61 | 38 | 29 |

*Qo'sh xeshlashning afzalliklari va kamchiliklari*. Qo'sh xeshlash usuli birlamchi va ikkilamchi klasterlashning barcha muammolaridan xoli. Bundan tashqari, takroriy to'qnashuvlarni kamaytiradi.

***Xulosa***. Xesh-jadval - bu qiymat deb ataladigan maxsus indeks orqali ma'lumotlarga kirish mumkin bo'lgan massiv. Xesh-jadvalda qiymatlar xesh funksiyasi orqali massiv indekslari bilan taqqoslanadi.

Xesh-funksiya matematik formula bo'lib, qiymatga qo'llanganda butun sonni hosil qiladi va u xesh jadvalidagi qiymatni topish uchun indeks sifatida ishlatiladi.

Sonli qiymatlardan foydalanadigan turli xil xesh funksiyalari mavjud bo'lib, eng ommabop usullari: bo'lish usuli, o'rta kvadrat usuli va bo'laklash (folding) usuli.

Bo'lish usulida $k$ ning $m$ ga bo'lingan qolgan qismini olish yo'li bilan $k$ qiymat $m$ ta yacheykalardan biriga joylashtiriladi. Bo'lish usulining asosiy kamchiligi shundaki, ko'plab ketma-ket qiymatlar mos ravishda

ketma-ket xesh qiymatlariga mos keladi, bu ketma-ket qatorlar joylashishini anglatadi va shuning uchun ishlashga ta'sir qiladi.

O'rta kvadrat usulida biz berilgan qiymatning kvadratini hisoblaymiz. Sonni kvadratidagi o'rta raqamlar ajratib olinadi.

Bo'laklash usulida biz qiymatni bo'laklarga ajratamiz, shunda har bir bo'lak oxirgisidan tashqari bir xil sondagi raqamlarga ega bo'ladi, boshqa qismlarga qaraganda kamroq raqamlar bo'lishi mumkin. Endi bu alohida qismlar qo'shiladi. Shunday qilib, hash qiymati hosil bo'ladi.

*To'qnashuv* - bu xesh-funksiya ikki xil qiymatni xesh jadvalidagi bitta/bir xil joyga joylashtirganda yuzaga keladigan holat. To'qnashuvni hal qilish usullari xeshlashda to'qnashuv muammosini bartaraf etish uchun ishlatiladi. To'qnashuvlarni hal qilish uchun ikkita mashhur usul qo'llaniladi, ular zanjirlash usuli va ochiq adreslash usuli. Zanjirlash usulida bir xil xesh-manzilga ega bo'lgan elementlar zanjiri saqlanadi. Bu yerda xesh-jadvallar ko'rsatkichlar massivi kabi ishlaydi. Xesh-jadvaldagi har bir joy o'sha joyga xeshlangan barcha asosiy elementlarni o'z ichiga olgan bog'langan ro'yxatga ko'rsatgichni saqlaydi. Ushbu usulning kamchiligi asosiy elementlar bog'langan ro'yxatda saqlanganligi sababli saqlash joyini ortiqcha sarflashga olib keladi; qo'shimcha ravishda, kirish uchun har bir element uchun ko'rsatkichlar talab qilinadi, bu esa o'z navbatida ko'proq joy sarflaydi.

Ochiq adreslash usulida barcha elementlar xesh-jadvalning o'zida saqlanadi. Ushbu usulda ko'rsatgichlarni taqdim etishning hojati yo'q, bu usulning eng katta afzalligi. To'qnashuv sodir bo'lgandan so'ng, ochiq manzil zond ketma-ketligidan foydalangan holda yangi joylarni hisoblaydi va keyingi element yoki keyingi yozuv o'sha joyda saqlanadi.

*Zondash* - bu xesh jadvalidagi xotira joylarini tekshirish jarayoni. *Chiziqli zondlash* - xeshlashda to'qnashuv muammosini hal qilishning eng oddiy usuli. Bu usulda, agar qiymat $h(k)$ xesh-funksiyasi tomonidan yaratilgan joyda allaqachon saqlangan bo'lsa, vaziyatni quyidagi xesh-funksiya yordamida hal qilish mumkin: $h'(k) = (h(k) + i)\ mod\ m$

*Kvadrat zondlash* - xeshlashda to'qnashuv muammosini hal qilishning yana bir usuli. Bu usulda agar qiymat $h(k)$ xesh-funksiyasi



tomonidan yaratilgan joyda allaqachon saqlangan bo'lsa, vaziyatni quyidagi xesh funksiyasi yordamida hal qilish mumkin:

$$h'(k) = (h(k) + c_1 * i + c_2 * i^2)\ mod\ m$$

Qo'sh xeshlash ochiq manzillashning eng yaxshi usullaridan biridir. Nomidan ko'rinib turibdiki, bu usul bitta xesh funksiyasidan ko'ra ikkita xesh funksiyasidan foydalanadi. Xesh funksiyasi quyidagicha berilgan:

$$h'(k) = (h_1(k) + i * h_2(k))\ mod\ m$$ 1-

savol: Xeshlash atamasini tushuntirib bering.

Javob: Xeshlash - bu qiymatlar uchun xesh-jadvalda mos (tegishli) joylarni aniqlash jarayoni. Bu massiv yoki xesh jadvalidagi qiymatlarni qidirishning eng samarali usuli.

2-savol: 50 ta xotira yacheykasining xesh-jadvalini hisobga olgan holda, bo'lish usuli yordamida 20 va 75 qiymatlarining xesh qiymatlarini hisoblang.

Javob. $m = 50, k1 = 10, k2 = 75$ xesh qiymatlari quyidagicha hisoblanadi: $h(10) = 10\%50 = 10$

$$h(75) = 75\%50 = 25$$

3-savol: 100 ta xotira yacheykasidan iborat xesh-jadvalni hisobga olgan holda, o'rta kvadrat usuli yordamida 2045 va 1357 qiymatlarning xesh qiymatlarini hisoblang.

Javob: Demak masalaning qo'yilishidan ma'lumki, indekslari 0 dan 99 gacha bo'lgan 100 ta xotira yacheykasi mavjud. Qiymatlarni moslashtirish uchun faqat ikkita raqam olamiz. Demak, $r$ ning qiymati 2 ga teng.

$$k = 2045, k^2 = 4182025, h(2045) = 20$$

$$k = 1357, k^2 = 1841449, h(1357) = 14$$

Eslatma: Uchinchi va to'rtinchi raqamlar o'ngdan boshlash uchun tanlangan.

4-savol: 100 ta xotira yacheykasidan iborat xesh-jadvalni hisobga olgan holda, bo'laklarga bo'lish usuli yordamida 2486 va 179 qiymatlarining xesh qiymatlarini hisoblang.

Javob: Indekslar 0 dan 99 gacha bo'lgan 100 ta xotira yacheykasi mavjud.

Demak, qiymatning har bir qismida ikkita raqam bo'lishi kerak.

$$h(2486) = 24 + 86 = 110$$

$$h(2486) = 10$$

$$h(179) = 17 + 9 = 26$$

$$h(179) = 26$$

### 7.5. Mustaqil ishlash uchun savollar va mashqlar

#### 7.5.1. Qidiruv algoritmlari

**1.** Ketma-ket qidiruv tomonidan amalga oshirilgan taqqoslashlar sonini aniqlang:

*a.* eng yomon holatda.

*b.* o'rtacha holatda, agar muvaffaqiyatli qidiruv ehtimoli $p$ ($0 \le p \le 1$) bo'lsa.

**2.** Ketma-ket qidirish orqali amalga oshirilgan asosiy taqqoslashlarning o'rtacha soni quyidagi formula bilan ifodalanadi:

$$C_o'_{rta}(n) = \frac{p(n+1)}{2} + n(1-p),$$

bu yerda $p$ - muvaffaqiyatli qidiruv ehtimoli. Berilgan $n$ uchun $p$ ning ( $0 \le p \le 1$) shunday qiymatlarini aniqlangki, bu formula $C_o'_{rta}(n)$ ning maksimal va minimal qiymatlarini bersin.

**3.** Qurilma sinovlari. Kompaniya o'zining $n$ qavatli bosh ofisining qaysi eng yuqori qavatidan qurilmani tushirib yuborsa sinmasligini aniqlashni xohlaydi. Kompaniyada sinov o'tkazish uchun ikkita bir xil qurilma mavjud. Agar ulardan biri sinsa, uni ta'mirlash imkonsiz bo'ladi va sinov qolgan qurilma bilan yakunlanishi kerak. Ushbu muammoni hal qilish uchun eng samarali algoritm tuzing.

**4.** MULKIM qism matnini quyidagi matndan qidirishda algoritm (chiziqli) tomonidan amalga oshirilgan belgilar taqqoslashlari sonini aniqlang:

HAYOTDA_TEZLIKNI_OSHIRISHDAN_KO'RA_MUHIMRO
Q_NARS
ALAR_BOR



Matnning uzunligi - 56 ta belgi - qidiruv boshlanishidan oldin ma'lum deb hisoblang.

**5.** Mingta noldan iborat ikkilik (binar) matndagi quyidagi belgilarning har birini qidirishda algoritm qancha taqqoslash (muvaffaqiyatli va muvaffaqiyatsiz) amalga oshiradi?

*a.* 00001; *b.* 10000; *c.* 01010.

**6.** To'g'ridan-to'g'ri satrlarni taqqoslash algoritmi uchun eng yomon holat hisoblanadigan $n$ uzunlikdagi matn va $m$ uzunlikdagi qism-satr misolini keltiring. Bunday kirish uchun aynan nechta belgi taqqoslanadi?

**7.** Satrlarni taqqoslash masalasini yechishda qism-satr va matn belgilarini chapdan o'ngga emas, o'ngdan chapga taqqoslashning biror afzalligi bormi?

**8.** Berilgan matndagi *A* harfi bilan boshlanib, *B* harfi bilan tugaydigan qism satrlar sonini hisoblash masalasini ko'rib chiqing. Masalan, CABAAXBYA matnida bunday qism satrlar 4 ta.

**9.** So'z topish. AQShda keng tarqalgan ermak bo'lgan "so'z topish" (yoki "so'z qidirish") jumboqlari o'yinchidan berilgan so'zlar to'plamining har birini yagona harflar bilan to'ldirilgan kvadrat jadvaldan topishni talab qiladi. So'z gorizontal (chapdan o'ngga yoki o'ngdan chapga), vertikal (yuqoridan pastga yoki pastdan yuqoriga), yoki 45 darajali diagonal bo'ylab (to'rt yo'nalishning istalganida) jadvalning ketma-ket yondosh kataklaridan o'qilishi mumkin. U jadval chegaralarini aylanib o'tishi mumkin, ammo bir xil yo'nalishda, zigzag qilmay o'qilishi shart. Jadvalning bir katagi turli so'zlarda ishlatilishi mumkin, lekin bitta so'zda ayni katak faqat bir marta qo'llanilishi mumkin. Ushbu jumboqni yechish uchun kompyuter dasturi yarating.

**10.** Jangovar kema o'yini. Kompyuterda "Harbiy kema" o'yinini o'ynash uchun to'liq qidirish usulidan foydalangan holda dastur tuzing. O'yin qoidalari quyidagicha: O'yinda ikkita raqib ishtirok etadi (bu holda, inson o'yinchi va kompyuter). O'yin ikkita bir xil taxtada (10x10 katakli jadval) o'ynaladi, unda har bir raqib o'z kemalarini raqibga ko'rinmaydigan tarzda joylashtiradi. Har bir o'yinchida beshta kema bo'lib, ular taxtada ma'lum miqdordagi katakni egallaydi: esminets (ikki

katak), suv osti kemasi (uch katak), kreyser (uch katak), linkor (to'rt katak) va aviataşuvchi (besh katak). Har bir kema gorizontal yoki vertikal joylashtiriladi, bunda ikki kema bir-biriga tegmasligi kerak. O'yin raqiblar navbatma-navbat bir-birlarining kemalariga "o't ochish" bilan davom etadi. Har bir zarbaning natijasi zarba yoki xato sifatida ko'rsatiladi. Agar tegsa, o'yinchi yana o't ochishni davom ettiradi va toki tegmaguncha o'yinni davom ettiradi. Maqsad - raqib muvaffaqiyatga erishishidan oldin uning barcha kemalarini cho'ktirish. Kemani cho'ktirish uchun u egallagan barcha kataklarga tegish kerak.

**11.** Tayoqni kesish. Uzunligi $n$ dyuym bo'lgan cho'pni $n$ ta 1 dyuymli bo'laklarga bo'lish kerak. Agar tayoqchaning bir nechta bo'lagini bir vaqtning o'zida kesish mumkin bo'lsa, bu vazifani eng kam kesish bilan bajaradigan algoritmni tuzing. Shuningdek, eng kam kesish sonining formulasini keltiring.

**12.** $\lfloor \log_2 n \rfloor$ ni hisoblashning ikki marta kamaytirish algoritmini tuzing va uning vaqt bo'yicha samaradorligini aniqlang.

**13.** ***a.*** Quyidagi massivda kalitni qidirishda binar qidiruv usulida amalga oshirilgan eng ko'p kalit taqqoslashlar soni qancha? {3, 14, 27, 31, 39, 42, 55, 70, 74, 81, 85, 93, 98}

***b.*** Ikkilik qidiruv usuli bilan qidirilganda eng ko'p kalitlarni taqqoslashni talab
qiladigan ushbu massivning barcha kalitlarini sanab o'ting.

***c.*** Ushbu massivdagi muvaffaqiyatli qidiruvda binar qidiruv orqali amalga oshirilgan asosiy taqqoslashlarning o'rtacha sonini toping. Har bir kalit bir xil ehtimollik bilan qidiriladi deb faraz qilaylik.

***d.*** Ushbu massivdagi muvaffaqiyatsiz qidiruvda binar qidiruv orqali amalga oshirilgan kalitli taqqoslashlarning o'rtacha sonini toping. Faraz qilaylik, massiv elementlari hosil qilgan 14 ta intervalning har birida kalitlarni izlash bir xil ehtimolli bo'lsin.

**14.** Bir million elementdan iborat tartiblangan massivda o'rtacha muvaffaqiyatli qidiruv binar qidiruv va ketma-ket qidiruv orqali amalga oshirilsa, necha barobar tezroq bo'lishini baholang.

**15.** Ketma-ket qidiruvning vaqt bo'yicha samaradorligi ro'yxatning massiv ko'rinishida yoki bog'langan ro'yxat ko'rinishida amalga



oshirilishiga bog'liq emas. Bu binar qidiruv bo'yicha saralangan ro'yxatni qidirish uchun ham to'g'rimi?

**16.** Rasmni taxmin qilish. Mashhur muammoni hal qilish vazifasining bir varianti odamlarga 42 ta rasmdan iborat massivni taqdim etishni o'z ichiga oladi - har biri oltita rasmdan iborat yettita qator - va ulardan "ha" yoki "yo'q" deb javob berish mumkin bo'lgan savollarni berish orqali maqsadli rasmni aniqlashni so'rashadi. Bundan tashqari, odamlardan iloji boricha kamroq savollar bilan rasmni aniqlash talab qilinadi. Ushbu masala uchun eng samarali algoritmni taklif qiling va kerak bo'lishi mumkin bo'lgan eng ko'p savollar sonini ko'rsating.

**17.** Uchlik qidiruv - $A[0\,..\,n-1]$ tartiblangan massivda quyidagi qidiruv algoritmini ko'rib chiqing. Agar $n = 1$ bo'lsa, qidiruv kaliti $K$ ni massivning bitta elementi bilan taqqoslash kifoya, aks holda, $K$ ni $A[\lfloor n/3 \rfloor]$ bilan taqqoslash orqali rekursiv qidirish, agar $K$ katta bo'lsa, uni $A[\lfloor 2n/3 \rfloor]$ bilan taqqoslash orqali qidiruvni massivning qaysi uchdan birida davom ettirish kerakligini aniqlash mumkin.

***a.*** Ushbu algoritm qanday loyihalash texnikasiga asoslangan?

***b.*** Eng yomon holatda asosiy taqqoslashlar soni uchun takrorlanishni
o'rnating. $n = 3^k$ deb oling.

***c.*** $n = 3^k$ uchun recurrent munosabatni o'rnating.

***d.*** Ushbu algoritmning samaradorligini binar qidiruv bilan solishtiring.

**18.** $A[0\,..\,n-2]$ massiv o'sish tartibida 1 dan $n$ gacha bo'lgan $n-1$ ta butun sonni o'z ichiga oladi. (Shunday qilib, bu oraliqda bitta butun son yo'q.) Yetishmayotgan butun sonni topish va uning vaqt samaradorligini ko'rsatish uchun eng samarali algoritmni ishlab chiqing.

**19.** Interpolyatsion izlashga asos bo'lgan formulani keltirib chiqaring.

**20.** Interpolyatsion qidiruv uchun eng yomon kirish holatiga misol keltiring va algoritm eng yomon holatda chiziqli ekanligini ko'rsating.

**21.** ***a.*** Binar qidiruv daraxtida eng katta kalitni topish algoritmini tasvirlang.

Algoritmingizni o'zgaruvchan o'lchamni kamaytirish algoritmi sifatida tasniflash mumkinmi?

***b***. Eng yomon holatda algoritmingizning vaqt samaradorligi sinfi qanday?

**22.** ***a***. Binar qidiruv daraxtidan kalitni o'chirish algoritmini tasvirlang. Ushbu algoritmni o'zgaruvchan o'lchamni kamaytirish algoritmi sifatida tasniflash mumkinmi?

***b***. Eng yomon holatda algoritmingizning vaqt samaradorligi sinfi qanday?

### 7.5.2. Xesh jadva va xesh-funksiyalar

**1.** Xesh-jadvallar nima?

**2.** Xeshlash nima? Uning ba'zi amaliy qo'llanish **sohalarini** keltiring.

**3.** Xesh-funksiyaga ta'rif bering, shuningdek, xesh-funksiyaning turli xususiyatlarini tushuntiring.

**4.** Xeshlashdagi to'qnashuv nima va uni qanday hal qilish mumkin?

**5.** Xesh funksiyalarning turlarini misollar bilan tushuntirib bering.

**6.** Xeshlashda to'qnashuv **(kolliziya)**larni bartaraf etish usullarini muhokama qiling.

**7.** Xeshlashda klasterlash nima? Klasterlashning qanday turlari mavjud?

**8.** Qo'sh xeshlash deganda nimani tushunasiz?

**9.** Quyidagi atamalarga ta'rif bering: Kvadrat zondlash, Chiziqli zondlash.

**10.** Xeshlashda zanjirlash usuli nima va u to'qnashuvlarni hal qilishda qanday yordam berishi mumkin?

**11.** O'lchami 10 bo'lgan xesh-jadvalga chiziqli zondlash yordamida 12, 45, 67, 122, 78 va 34 qiymatlarni kiriting.

**12.** O'lchami 9 ga teng bo'lgan xesh-jadvalga qo'sh xeshlashdan foydalanib, 4, 17, 30, 55, 90, 11, 54, 77 qiymatlarni joylashtiring. $h1 = k \bmod 9$ va $h2 =$



$k \bmod 6$.

**13.** O'lchami 11 ga teng bo'lgan xesh-jadvalga kvadratik zondlash yordamida 10, 45, 56, 97, 123 va 1 qiymatlarni kiriting.

**14.** To'qnashuvlarni hal qilishda ochiq adreslash usulidan qanday foydalanish mumkin?

**15.** Chiziqli zondlash va kvadratik zondlash yordamida xesh-jadvaldan elementni olish uchun C++ funksiyasini yozing.

**16.** Quyidagi to'qnashuvlarni hal qilish usullaridan qaysi biri klasterlash hodisasidan xoli?

a) Chiziqli zondlash.

**b)**Kvadrat zondlash

**c)**Qo'sh xeshlash

**d)**Bularning hech biri

**17.** Xotira o'rnini tekshirish jarayoni _____ deb ataladi.

**a)**Zondlash

**b)**Xeshlash

**c)**Zanjirlash

**d)**Manzillash

**18.** To'qnashuvlarni hal qilish usuli sifatida zanjirli xesh-jadval quyidagiga aylanadi:

a) Daraxt

**b)**Graf

**c)**Massiv

**d)**Bog'langan ro'yxat

**19.** Zondlash usullaridan qaysi biri birlamchi klasterlash muammosidan aziyat chekadi?

**a)**Kvadrat zondlash

**b)**Chiziqli zondlash

**c)**Qo'sh xeshlash

**d)**Bularning hammasi

**20.** $h(k) = k \bmod 6$ xesh-funksiyasi berilgan bo'lsa, 16, 20, 45, 68 qiymatlar ketma-ketligini ochiq manzillash yordamida saqlash uchun to'qnashuvlar soni qancha?

a) 1. b) 3 ta c) 2 ta d) 5 ta.

**21.** Xesh-jadvalda $k$ qiymatli element _____ da saqlanadi. a) $k$

**b)** $h(k2)$

**c)** $h(k)$

**d)** $log\ h(k)$

**22.** Yaxshi xesh-funksiya to‘qnashuv muammosini bartaraf etadi. **a) to‘g‘ri**

**b)** yolg‘on

**c)** Fikr bildirish imkonsiz

**23.** O‘lchami 7 bo‘lgan xesh-**jadval** va $h(k) = k\ mod\ 7$ xesh-funksiya berilgan bo‘lsa, quyidagi qiymatlarni kiritish uchun chiziqli zondlash bilan to‘qnashuvlar soni qancha: 29, 36, 16 va 30?

a) 1 b) 2 ta c) 3 ta d) 4 ta.

**24.** _____ - bu qiymatlarni xesh-jadvalning tegishli joylariga moslashtirish jarayonidir.

a) Zondlash

b) Xeshlash

c) To‘qnashuv

d) Manzillash

**25.** Xesh-jadvalda bo‘sh joy bo‘lmaganda, _____ sodir bo‘ladi.

**a)**Pastki oqim

**b)**Toshib ketmoq

**c)**To‘qnashuv

**d)**Yuqoridagilarning hech biri



## Foydalanilgan adabiyotlar

1. Anany Levitin. Introduction to The Design&Analysis of Algorithms. 3rd
Edition. 2012, Pearson Education, Inc., publishing as Addison-Wesley. ISBN10:0-
13-231681-1, ISBN13:978-0-13-231681-1

2. N. Karumanchi. Data Structures and Algorithms Made Easy: Data structures and Algorithmic Puzzles, Second edition, 2015.

3. M.A.Weiss, Data Structures & Algorithm Analysis in C++, 4th edition, 2014.

4. Aho A.V., Hopkroft J.E., Ullman J.E. /Ахо А., Хопкрофт Дж., Ульман Дж. - Data Structures and Algorithms / Структуры данных и алгоритмы, 2003.

5. Bucknall J. / Бакнелл Дж. - The Tomes of Delphi Algorithms and Data Structures / Фундаментальные алгоритмы и структуры данных в Delphi, 2003.

6. Boynazarov I.M., Xujayarov I.Sh. Linux operatsion tizimi. O‘quv qo‘llanma. -T.: “Fan va texnologiyalar nashriyot-matbaa uyi”, 2022. -332 b. Xujayarov I.Sh.

7. Boynazarov I.M., Qarshiyev A.B., Babajanov B.B., Bobonazarov A. Algoritmlarni loyihalash. O‘quv qo‘llanma. -T.: “Fan va texnologiyalar nashriyotmatbaa uyi”, 2024.-164 b.

8. Boynazarov I.M. Sonli usullar va chiziqli dasturlash. O‘quv qo‘llanma. -T.: “Fan va texnologiyalar nashriyot-matbaa uyi”, 2023. -332 b.

9. Boynazarov I.M., Toirov Sh.A. Ma’lumotlar tuzilmasi va algoritmlar. O‘quv qo‘llanma. -T.: “Fan va texnologiyalar nashriyot-matbaa uyi”, 2023. -432 b.

10. Boynazarov I.M., Xujayarov I.Sh., Qutratov R.B. Operatsion tizimlar. Darslik. -T.: “Fan va texnologiyalar nashriyot-matbaa uyi”, 2024. -432 b.

11. Bo‘ronova, G.Y. — "Ma’lumotlar tuzilmasi va algoritmlar tahlili" (2024).
https://uniwork.buxdu.uz/resurs/13321_2_C443F96EFBF0FC5EB974DE3A8D5FB7B3029C94F6.pdf?utm_source=chatgpt.com

12. Turdialiyev, S.M. maqolasi: "Algoritmlarni ishlab chiqish usullaridan foydalanish" (2023).
13. Sedgewick, R., Wayne, K. Algorithms (4th ed., Addison-Wesley, 2011).
14. Skiena, S. S. – The Algorithm Design Manual (Springer, 3rd ed., 2020).
15. Goodrich, M.T., Tamassia, R., Goldwasser, M.H. – Data Structures and Algorithms in Python (Wiley, 2013).
16. Кормен Т., Лейзерсон Ч., Ривест Р., Штейн К. Алгоритмы: построение и анализ (3-е изд., Вильямс, 2016).



***QAYDLAR UCHUN***

**Mamajanov R.Ya., Boynazarov I.M.**

# ALGORITMLARNI LOYIHALASHTIRISH VA TAHLIL QILISH

***O'quv qo'llanma***

Muharrir: X. Tahirov
Texnik muharrir: S. Meliquziyeva
Musahhih: M. Yunusova
Sahifalovchi: D. Toshboltayev

Nashr. Lits No 2244. 25.08.2020 y.
Bosishga ruxsat etildi 27.12.2025 y.
Bichimi 60x84 1/16. Ofset qog'ozi.
"Times New Roman" garniturasi. Hisob-nashr tabog'i. 16,75.
Adadi 100 dona. Buyurtma No 22.

«ZEBO PRINTS» MCHJ bosmaxonasida chop etildi.
Manzil: Toshkent sh., Yashnobod tumani, 22-harbiy shaharcha.
+998 (94) 673-66-56, +998 (97) 017-01-01

ISBN 978-9910-14-011-2