

Спорттық бағдарламалау нұсқаулығы

Автор: Лааксонен Антти
Редактор: Итеғұлова Сая Ағанасқызы
Аудармашы: Талгаткызы Нурхаят

12 тамыз 2024 ж.

Мазмұны

Алғысөз	vi
Аудармашылардан алғысөз	vii
Қазақша аударма нұсқаға автордың пікірі	viii
Кітаптың редакторы	ix
I Негізгі әдіс-тәсілдер	1
1 Кіріспе	2
1.1 Бағдарламалау тілдері	2
1.2 Енгізу (input) мен шығару (output)	3
1.3 Сандармен жұмыс	5
1.4 Кодты ықшамдау	7
1.5 Математика	9
1.6 Контексттер және ресурстар	14
2 Уақытша күрделілігі	17
2.1 Есептеу ережелері	17
2.2 Күрделілік кластары	20
2.3 Тиімділікті бағалау	21
2.4 Ішжиымдардың ең жоғары қосындысы	22
3 Сұрыптау	25
3.1 Сұрыптау теориясы	25
3.2 C++ -тегі сұрыптау	29
3.3 Бинарлық ізденіс	31
4 Деректер құрылымдары	35
4.1 Динамикалық жиымдар	35
4.2 Жиын құрылымдары	37
4.3 Сөздік құрылымы	38
4.4 Итераторлар мен аралықтар	39
4.5 Басқа құрылымдар	41
4.6 Сұрыптаумен салыстыру	46

5	Толық ізденіс	48
5.1	Ішжиындар құру	48
5.2	Алмастырулар құрау	50
5.3	Қайта іздеу алгоритмі	51
5.4	Ізденісті ықшамдау	53
5.5	Ортада кездесу	56
6	Ашкөз алгоритмдер	58
6.1	Тиын жайлы есеп	58
6.2	Жоспарлау	59
6.3	Тапсырмалар және ақырғы мерзімдер	61
6.4	Қосындыны минималдау	62
6.5	Деректерді сығымдау	63
7	Динамикалық бағдарламалау	66
7.1	Тиын жайлы есеп	66
7.2	Ең ұзын өспелі іштізбек	71
7.3	Тордағы жолдар	72
7.4	Қоржын жайлы есептер	74
7.5	Түзету арақашықтығы	75
7.6	Тақташа төсеу жолдарын санау	77
8	Амортизацияланған талдау	79
8.1	Екі нұсқағыш әдісі	79
8.2	Ең жақын кішірек элементтер	81
8.3	Жылжымалы терезе минимумы	83
9	Аралық сұратымдар	85
9.1	Статикалық жиым сұратымдары	85
9.2	Бинарлы индекстелген дарақ	88
9.3	Кесінділер дарағы	91
9.4	Қосымша әдіс-тәсілдер	95
10	Биттік манипуляциялау	97
10.1	Биттік көрсетілім	97
10.2	Биттік операциялар	98
10.3	Жиындар көрсетілімі	100
10.4	Бит оңтайландырулары	102
10.5	Динамикалық бағдарламалау	104
II	Графтағы алгоритмдер	110
11	Граф негіздері	111
11.1	Граф терминологиясы	111
11.2	Графты көрсету жолдары	115

12	Графты аралау	119
12.1	Тереңдігі бойынша іздеу	119
12.2	Ені бойынша іздеу	121
12.3	Қолданыс аясы	123
13	Ең қысқа жолдар	126
13.1	Беллман-Форд алгоритмі	126
13.2	Дейкстра алгоритмі	129
13.3	Флойд-Уоршелл алгоритмі	132
14	Дарақтағы алгоритмдер	136
14.1	Дарақты аралап шығу	137
14.2	Диаметр	138
14.3	Барлық ең ұзын жолдар	140
14.4	Бинарлы дарақ	142
15	Қаңқалы дарақ	144
15.1	Краскал алгоритмі	145
15.2	Қиылыспайтын жиындар құрылымы	148
15.3	Прим алгоритмі	150
16	Бағытталған граф	153
16.1	Топологиялық сұрыптау	153
16.2	Динамикалық бағдарламалау	155
16.3	Мирасқорлар графы	158
16.4	Циклді анықтау	159
17	Берік байланыстылық	162
17.1	Косараджу алгоритмі	163
17.2	2SAT есебі	165
18	Дарақтағы сұратымдар	168
18.1	Бабалардың ізденісі	168
18.2	Ішдарақтар және жолдар	169
18.3	Жақын арадағы ата-тек	172
18.4	Оффлайн алгоритмдер	175
19	Жолдар және шынжырлар	179
19.1	Эйлер жолы	179
19.2	Гамильтон жолдары	183
19.3	Де Брёйін тізбегі	184
19.4	Ат сапары	185
20	Ағындар мен қималар	187
20.1	Форд–Фалкерсон алгоритмі	188
20.2	Қиылыспайтын жолдар	192
20.3	Максималды жұптасу	193
20.4	Жол бүркемелері	197

III	Күрделі тақырыптар	200
21	Сандар теориясы	201
21.1	Жай сандар және көбейткіштер	201
21.2	Модульдік арифметика	206
21.3	n жұп болған жағдайда $x^{n/2}$ мәні	206
21.4	Теңдеулерді шешу	208
21.5	Басқа нәтижелер	210
22	Комбинаторика	212
22.1	Биномдық коэффициент	213
22.2	Каталан сандары	215
22.3	Inclusion–exclusion principle	217
22.4	Бернсайд леммасы	219
22.5	Кели теоремасы	220
23	Матрицалар	222
23.1	Операциялар	222
23.2	Сызықтық рекуренттілік	225
23.3	Графтар мен матрицалар	227
24	Ықтималдылық	230
24.1	Есептеу	230
24.2	Оқиғалар	231
24.3	Кездейсоқ шамалар	234
24.4	Марковтық тізбе	236
24.5	Рандомизацияланған алгоритмдер	237
25	Ойындар теориясы	240
25.1	Ойын күйлері	240
25.2	Ним ойыны	242
25.3	Шпраг-Гранди теоремасы	243
26	Жолды өңдеу алгоритмдері	247
26.1	Жол терминологиясы	247
26.2	Префикс дарағы	248
26.3	Жол хэші	249
26.4	Z-алгоритм	252
27	Квадраттық түбір алгоритмдер	256
27.1	Алгоритмдерді біріктіру	257
27.2	Бүтін бөлінулер	259
27.3	Мо алгоритмі	260
28	Тағы да кесінділер дарағы туралы	262
28.1	Жалқау таратылу	263
28.2	Динамикалық дарақ	266
28.3	Деректер құрылымы	268

28.4	Екі өлшемділік	269
29	Геометрия	271
29.1	Комплекс сандар	272
29.2	Нүктелер мен сызықтар	274
29.3	Көпбұрыш ауданы	277
29.4	Арақашықтық функциялары	279
30	Сыпырма түзуге негізделген алгоритмдер	282
30.1	Қиылысу нүктелері	283
30.2	Ең жақын жұп есебі	284
	Әдебиеттер	285
	Пәндік сілтеме	290

АЛҒЫСӨЗ

Кітаптың мақсаты – өз оқырмандарын спорттық бағдарламалаудың тұжырымдарымен толық таныстырып шығу. Сіз бағдарламалаудың негіздерін бұған дейін де біледі деп бағамдаймыз, десек те, спорттық бағдарламалауда қандай да бір алдын ала даярлық қажет емес деп болжанады.

Кітап, әсіресе, алгоритмдерді үйренгісі келетін және Информатика бойынша халықаралық олимпиадаға (IOI) немесе Студенттердің бағдарламалау бойынша халықаралық конкурсына (ICPC) қатысқысы келетін білім алушыларға арналады. Сондай-ақ спорттық бағдарламалауға қызығушылық танытатын жалпы жұртшылыққа да пайдалы болмақ.

Бәсекеге қабілетті, жақсы спорттық бағдарламалаушы болу үшін көп уақыт қажет. Алайда мұны біраз дүниелерді үйрену үшін берілген мүмкіндік деп қабылдаған жөн. Егер уақытыңызды кітапты оқуға, есептерді шығаруға және конкурстарға қатысуға бөліп жүрсеңіз, алгоритмдер туралы жақсы жалпы түсінік алатыныңызға сенімді бола беріңіз.

Кітап үнемі жетілдіріліп, зерделеніп отырады. Кітап туралы пікіріңізді сіз әрқашан ahslaaks@cs.helsinki.fi мекенжайына жібере аласыз (ред. ескерту: қазақ тіліндегі аудармасы бойынша ұсыныстарыңызды, кітапшаның қазақша мазмұнына қатысты ойларыңызды crphb.kz@gmail.com мекенжайына жолдай аласыз).

Хельсинки, 2019 ж. тамыз
Лааксонен Антти

Аудармашылардан алғысөз

Аса Мейірімді, ерекше Рақымды Алланың атымен бастаймын! Барлық мақтау Алла тағалаға тән және пайғамбарымыз Мұхаммед Мұстафаға (Алланың оған сәлемі мен игілігі болсын), оның отбасына, сахабаларына Алла тағаланың рахым нұры жаусын!

Қолыңыздағы шағын кітап спорттық бағдарламалауға кіріспе ғана емес, мұнда республикалық олимпиадаларда кездесетін күрделі тақырыптар да қамтылып отыр. Игілігіңізге тұтыныңыз, қымбатты оқырман.

2024 ж. тамыз

Қазақша аударма нұсқаға автордың пікірі

It is great that my book has been translated into the Kazakh language, which makes the book accessible to more readers. I would like to thank the people who have created the translation. This year Kazakhstan hosts the ICPC World Finals, which I look forward to. I hope the book is useful for learners who are interested in competitive programming.

June 2024
Antti Laaksonen

Кітабымның қазақ тіліне аударылғаны қандай тамаша. Бұл оны көптеген оқырмандарға қолжетімді етеді. Мен осы аударманы жасаған адамдарға алғысымды білдіргім келеді. Биыл Қазақстанда ICPC әлемдік финалы өткелі отыр. Мен осынау атаулы күнді тағатсыздана күтудемін. Бұл кітап спорттық бағдарламалауға қызығушылық танытатындарға пайдалы болады деп үміттенемін.

2024 ж. маусым
Лааксонен Антти

Кітаптың редакторы

Итеғұлова Сая Ағанасқызы:

- Ақтөбе педагогикалық институтын 1995 жылы үздік аяқтады.
- Филология ғылымдарының кандидаты.
- ҚР Мәдениет және спорт министрінің Құрмет грамотасының иегері.
- 2023 жылдың қыркүйек айынан "Мемлекеттік тілді дамыту институты" ЖШС жоба орындаушысы.
- 2020-2022 жылдары "Шайсұлтан Шаяхметов атындағы "Тіл-Қазына" ұлттық ғылыми-практикалық орталығы" КеАҚ Терминология басқармасының жетекші ғылыми қызметкері.
- 2018-2020 жылдары Шайсұлтан Шаяхметов атындағы "Тіл-Қазына" ұлттық ғылыми-практикалық орталығының Қолданбалы лингвистика басқармасының басшысы.
- 2017-2018 жылдары ҚР Мәдениет және спорт министрлігі Туризм индустриясы комитеті Ішкі және келу туризмі басқармасының бас сарапшысы.
- "Қазақ мифологиялық эпосы" (Алматы: Ғылым, 2004) монографиясының, "Қазақтың мифтік кейіпкерлері" (Алматы: Полиграфкомбинат, 2016) ғылыми-танымдық еңбегінің, "Қазақ әдеби тілінің анықтамалығы" (Алматы, Полиграфкомбинат, 2016) еңбегінің авторы.

I БӨЛІМ.

Негізгі әдіс-тәсілдер

1-тарау. Кіріспе

Спорттық бағдарламалау екі тақырыпты қамтиды: (1) алгоритмдерді жобалау және (2) оларды жүзеге асыру.

Алгоритмдерді жобалау есеп шығару мен математикалық ойлаудан тұрады. Есеп шығару барысында креативті көзқарас пен талдау жасай алу қабілеті қажет. Есептің шешімі болатын алгоритм тиімді әрі дұрыс болуы тиіс, көбіне есептердің түпкі мақсаты оңтайлы алгоритмді анықтауға құрылады.

Теория – спорттық бағдарламалаушы үшін маңызды бөлім. Есептің шешімі – дұрыс пайдаланылған білім мен идеяның жемісі. Спорттық бағдарламалауда кездесетін кей әдістер алгоритмдерді тереңірек зерттеудің негізін қалыптастырады.

Жақсы бағдарлама жаза білу қабілеті алгоритмдерді жүзеге асыруға үлкен әсерін тигізеді. Спорттық бағдарламалауда есеп шешімі тест жиыны арқылы тексеріледі. Сондықтан алгоритм идеясы мен жазылуының дұрыс болуы шарт.

Бағдарламалау жарыстарындағы (контесттерде) ең жақсы код стиліне ”қарапайым да жинақы жазу” жатады. Уақыттың шетеулі болуына байланысты бағдарламаны тез жазған абзал. Инженерлік бағдарламалауға қарағанда бұл жерде код қысқа болады (әдетте максималды мөлшері бірнеше жүздеген жолдан тұрады), әрі жарыстан кейін ары қарай сүйемелдеуді қажет етпейді.

1.1 Бағдарламалау тілдері

Қазіргі таңда бағдарламалау жарыстарында (контесттерде) C++, Python, Java тілдері жиі қолданылады. Мысалы, Google Code Jam 2017 жарысында 3000 үздік қатысушылардың арасында 79 % C++, 16 % Python, 8 % Java [1] тілдерін қолданған. Қалған пайызы өзге тілдерді пайдаланған.

Көп адам C++ тілін спорттық бағдарламалау үшін ең қолайлы тіл деп есептейді және C++ барлық дерлік контесттер жүйесінде кездеседі. Бұл тілдің артықшылығы – тілдегі алгоритмдер мен деректер құрылымының үлкен жиынтығы болуында. Сондай-ақ C++ тілінде кодтың жылдамырақ жұмыс істеуі де бұл тілдің тиімділігін арттырады.

Бір жағынан бірнеше тілде жаза алудың өз тиімді тұстары да бар. Мысалы, есепте үлкен сандармен жұмыс жасау керек болса, Python ұтымды таңдау болмақ. Себебі оның ішіне үлкен сандармен жұмыс жасайтын операциялар кіріктірілген. Дегенмен контесттерде есептер барлық тілдерде шешім табуға болатындай етіп құрастырылатынын ұмытпағанымыз жөн.

Мысал ретінде берілген барлық кодтар C++ - те жазылып отыр және көбіне стандартты дерекханадағы алгоритмдер мен деректер құрылымдары қолданылады. Кітапта жазылған кодтар барлық заманауи контексттерде кездесетін C++11 стандартты нұсқасына сай келеді.

Егер сіз әлі C++ - пен таныс болмасаңыз, осы тілді меңгере бастауыңызға қолайлы сәт енді туды деп санауыңызға болады.

C++ код үлгісі

Жиі кездесетін C++ код үлгісі:

```
#include <bits/stdc++.h>

using namespace std;

int main() {
    // solution comes here
}
```

Кодтың ең жоғарғы бөлігінде жазылған `#include` жолы – g++ компиляторының ерекшелігі. Оның көмегімен барлық стандартты дерекханаларды пайдалана аламыз. `iostream`, `vector` және `algorithm` сияқты дерекханалардың әрқайсысын жеке-дара жазу қажеттілігі туындамайды, бірден автоматты түрде қолдана аламыз.

`using` жолы стандартты дерекхана кластары мен функциялары тікелей әрекет ете алатындығын көрсетеді. Онсыз `cout` орнына `std::cout` деп жазып отыруға тура келер еді.

Келесі команда арқылы код компиляциядан өтеді:

```
g++ -std=c++11 -O2 -Wall test.cpp -o test
```

Команда `test.cpp` кодынан `test` бинарлық файлын қайтарады. Компилятор C++11 стандартын қолданады (`-std=c++11`), кодты оңтайландырады (`-O2`) және туындауы мүмкін ескертулерге нұсқайды (`-Wall`).

1.2 Енгізу (input) мен шығару (output)

Көбіне контексттерде енгізу мен шығару үшін стандартты ағындарды пайдаланады. C++ үшін стандартты ағындар : `cin` енгізу үшін, `cout` шығару үшін. Сонымен қатар C функциялары `scanf` пен `printf` - ті пайдалануға болады.

Әдетте енгізу бос орын не жаңа жолдар арқылы ажыратылған сандар мен жолдардан (`string`) тұрады. Олар мысалдағыдай `cin` арқылы оқылады:

```
int a, b;
string x;
cin >> a >> b >> x;
```

Енгізуде әр элемент ең кем дегенде бос орынмен немесе жаңа жолмен бөлінгендіктен, әрдайым осы тәріздес кодтар жұмыс істейді.

Мысалы, жоғарыдағы код келтірілген мына екі енгізуді де қабылдай алады:

```
123 456 monkey
```

```
123    456  
monkey
```

cout ағымы осы жердегідей шығару үшін пайдаланылады:

```
int a = 123, b = 456;  
string x = "monkey";  
cout << a << " " << b << " " << x << "\n";
```

Дәл осы енгізу мен шығару бағдарламаның тиімді жұмыс жасауына кейде кедергі келтіреді. Ал төмендегі жолдар кедергіге қарсы әрекет етеді:

```
ios :: sync_with_stdio(0);  
cin . tie (0);
```

Тағы да ескеретін бір жайт бар. "\n" endl - ке қарағанда тезірек жұмыс жасайды, өйткені endl әрдайым буферді арылтып отырады.

C функциялары scanf пен printf C++ - тің стандартты ағымдарына балама ретінде қолданылады. C функциялары салыстырмалы түрде жылдамырақ жұмыс жасағанымен, қолданыста біраз қиындық тудырады. Төмендегі код екі сан енгізеді:

```
int a, b;  
scanf ("%d %d", &a, &b);
```

Ал келесі код керісінше екі сан шығарады:

```
int a = 123, b = 456;  
printf ("%d %d\n", a, b);
```

Кейде бағдарлама бір жолды толығымен, яғни бос орындармен бірге оқуы тиіс болады. Осындай кезде getline функциясын қолданамыз:

```
string s;  
getline (cin, s);
```

Егер енгізу саны нақты болмаса, келесі цикл пайдалы:

```
while (cin >> x) {  
    // code  
}
```

Бұл цикл енгізу деректері аяқталмайынша, стандартты енгізудегі барлық элементтерді қабылдай береді.

Кей контексттерде енгізу мен шығару файл ретінде беріледі. Мұндай жағдайда стандартты шығару мен енгізуді пайдаланып, төменде көрсетілген код жолдарын жазған абзал:

```
freopen("input.txt", "r", stdin);  
freopen("output.txt", "w", stdout);
```

Осыдан кейін бағдарлама енгізуді "input.txt" файлынан қабылдап, шығаруды "output.txt" файлында шығарады.

1.3 Сандармен жұмыс

Бүтін сандар

Бүтін сандардың ең жиі қолданылатын түрі – `int`. Ол – мәні $-2^{31} \dots 2^{31} - 1$ арасындағы (шамамен $-2 \cdot 10^9 \dots 2 \cdot 10^9$) бүтін сандарды қамтитын 32 биттік деректер типі. Егер `int` деректер типінен орасан үлкен/кіші сандарды сақтағыңыз келсе, 64 биттік `long long` деректер типін қолдана аласыз. Оның қамтитын ауқымы – $-2^{63} \dots 2^{63} - 1$ (шамамен $-9 \cdot 10^{18} \dots 9 \cdot 10^{18}$).

Төмендегі код `long long` типтік айнымалысын орнатады:

```
long long x = 123456789123456789LL;
```

LL суффиксі санның типі `long long` екенін көрсетеді.

`long long` қолданысындағы кең таралған қателердің бірі – `int` типін жаңылысып қолдану. Мысалы, төменгі кодта қате бар:

```
int a = 123456789;  
long long b = a*a;  
cout << b << "\n"; // -1757895751
```

b айнымалысы `long long` болғанымен, `a*a` өрнегіндегі айнымалылардың типі `int` болғандықтан, олардың көбейтіндісі де `int` болады. Сондықтан b айнымалысына қате нәтиже сақтайды. Қатені дұрыстау үшін a айнымалысының типін `long long` ауыстырсақ жеткілікті немесе өрнекті `(long long)a*a` деп өзгертсек те болады.

Контексттегі есептердің шешуіне әдетте `long long` жеткілікті. Дегенмен `g++` компиляторында 128 биттік типтік `__int128_t` деректер типін қолдануға болады. Оның қамтитын ауқымы – $-2^{127} \dots 2^{127} - 1$ (шамамен $-10^{38} \dots 10^{38}$). Бірақ, бұл деректер типі барлық контекст жүйелерінде қол жетімді емес.

Модульдік арифметика

Біз $x \bmod m$ деп x - ті m - ге бөлгендегі қалдықты белгілейміз. Мысалы, $17 \bmod 5 = 2$, себебі $17 = 3 \cdot 5 + 2$.

Кей есептердің жауабы тым үлкен сан болуы мүмкін. Сондай жағдайда есептің шартына сәйкес жауапты "модуль m " арқылы алуға болады. (мысалы, "модуль $10^9 + 7$ "). Мұндағы идея жауаптың үлкендігіне қарамастан `int` және `long long` типтерін қолданудың жеткілікті екендігіне негізделеді.

Қалдықтың маңызды қасиеті – қосу, азайту және көбейту операцияларын орындамастан бұрын қалдықты алдын-ала есептеу мүмкіндігі келесі формулалар арқылы көрінеді:

$$\begin{aligned}(a + b) \bmod m &= (a \bmod m + b \bmod m) \bmod m \\(a - b) \bmod m &= (a \bmod m - b \bmod m) \bmod m \\(a \cdot b) \bmod m &= (a \bmod m \cdot b \bmod m) \bmod m\end{aligned}$$

Осылайша біз әр операциядан кейін қалдық есептеу арқылы тым үлкен сан шықпауын қадағалаймыз.

Мысалы, $n!$ коды n факториалын m модулінде есептейді:

```
long long x = 1;
for (int i = 2; i <= n; i++) {
    x = (x*i)%m;
}
cout << x%m << "\n";
```

Әдетте қалдықтың $0 \dots m - 1$ аралығында болғанын қалаймыз. Алайда C++ және басқа тілдерде теріс сандардың қалдығы нөлге тең немесе теріс сандар болады. Қалдықтың теріс сан болмауын қадағалаудағы оңай жол – алдымен жай қалдықты есептеу, кейін жауап теріс болса, m - ді қосу:

```
x = x%m;
if (x < 0) x += m;
```

Дегенмен бұл жол кодта азайту болған жағдайда ғана қолданылады. Басқа жағдайларда теріс санды қалдыруға болады.

Қалқымалы нүктелі сандар

Әдетте спорттық бағдарламалауда кездесетін қалқымалы нүктелі сандар – деректер типі 64 биттік `double` мен `g++` компиляторындағы кеңейтілімі 80 биттік `long double`. Көп жағдайда `double` жеткілікті болғанымен, `long double` дәлірек келеді.

Талап етілетін жауап дәлдігі негізінен есеп шартында беріледі. Жауапты шығарудың оңай жолы – `printf` функциясын пайдаланып, нүктеден кейінгі сан мөлшерін форматтау жолында көрсету. Мысалы, төмендегі код x - ті нүктеден кейінгі 9 орынмен бірге шығарады:

```
printf("%.9f\n", x);
```

Қалқымалы нүктелі сандарды қолдану барысындағы қиындықтар кейбір сандардың дәлдігі мен ықшамдау кезінде туындайтын қателермен байланысты болып келеді. Мысалы, келесі код күтпеген нәтиже береді:


```
double x = 0.3*3+0.1;
printf("%.20f\n", x); // 0.99999999999999988898
```

Ықшамдау қателігі x - тің мәнін дұрыс жауабында дәл 1-ге тең екеніне қарамастан, 1 - ден сәл кішірек етіп көрсетеді.

Қалқымалы нүктелі сандарды `==` операторы арқылы салыстыру біраз қауіпті, өйткені дәлдік қатесінің әсерінен тең сандар дұрыс тексерілмеуі мүмкін. Оларды салыстырудың ұтымды жолы – екі санның айырмашылығы ϵ - ден көп болмаса, тең деп есептеу. Бұл жердегі ϵ –кішкентай сан.

Қолданыстағы сандарды осылай салыстыруға болады: ($\epsilon = 10^{-9}$):

```
if (abs(a-b) < 1e-9) {
    // a and b are equal
}
```

Қалқымалы нүктелі сандардың дәлдікпен сақталу мүмкіндігі болмаса да, бүтін сандарды белгілі мәндер аралығында жоғары дәлдікпен сақтай аламыз. Мысалы, `double` - ды қолданып, бүтін сандарды 2^{53} мәніне дейін сақтай аламыз.

1.4 Кодты ықшамдау

Кодты ықшамдап жазу – спорттық бағдарламалау үшін аса маңызды тәсіл. Себебі бағдарламаны барынша тез жазған абзал. Сондықтан спорттық бағдарламалаушылар әдетте деректер типтерін қысқа атаулармен алмастырады және кодтың өзге бөліктеріне де қысқа атаулар береді.

Тип атаулары

`typedef` командасын қолдану арқылы деректер типіне қысқа атау беруге болады. Мысалы, `long long` тым ұзақ, оны `ll` деп қысқарттық:

```
typedef long long ll ;
```

Төмендегі код:

```
long long a = 123456789;
long long b = 987654321;
cout << a*b << "\n";
```

Осылай қысқара алады:

```
ll a = 123456789;
ll b = 987654321;
cout << a*b << "\n";
```

typedef командасын өзге күрделірек типтер үшін де пайдалануға болады. Мысалы, төмендегі кодтағы vi атауы бүтін санды векторды, ал pi екі бүтін санды сақтауға арналған жұпты білдіреді.

```
typedef vector<int> vi;  
typedef pair<int,int> pi;
```

Макростар

Кодты ықшамдаудың тағы бір жолына макростарды қолдану жатады. Макрос компиляциялауға дейін белгілі бір жолдардың өзгеретінін білдіреді. C++ - та макростар #define кодсөзі арқылы анықталады.

Мысалы, төменде көрсетілген макростарды кодта қолдана аламыз:

```
#define F first  
#define S second  
#define PB push_back  
#define MP make_pair
```

Осыдан кейін кодты

```
v.push_back(make_pair(y1,x1));  
v.push_back(make_pair(y2,x2));  
int d = v[i].first + v[i].second;
```

осылай ықшамдай аламыз:

```
v.PB(MP(y1,x1));  
v.PB(MP(y2,x2));  
int d = v[i].F + v[i].S;
```

Сондай-ақ макростардың циклдер мен басқа да құрылымдарды қысқарта алатын параметрлері болуы мүмкін. Үлігі ретінде макростың бір нұсқасын берейік:

```
#define REP(i,a,b) for (int i = a; i <= b; i++)
```

Ол төмендегі кодты

```
for (int i = 1; i <= n; i++) {  
    search(i);  
}
```

осылай ықшамдай алады:

```
REP(i,1,n) {  
    search(i);  
}
```

Кейде макрос анықталуы қиын қателерді туындатуы мүмкін. Мысалы, санның квадратын есептейтін макросты қарастырайық:

```
#define SQ(a) a*a
```

Бұл макрос әрдайым күткеніміздей жұмыс жасамайды. Мына кодты қарастырайық:

```
cout << SQ(3+3) << "\n";
```

сәйкесінше:

```
cout << 3+3*3+3 << "\n"; // 15
```

Макростың дұрысырақ жазылған нұсқасы:

```
#define SQ(a) (a)*(a)
```

Ал енді төмендегі код:

```
cout << SQ(3+3) << "\n";
```

келесі өрнекке өзгереді

```
cout << (3+3)*(3+3) << "\n"; // 36
```

1.5 Математика

Математиканың спорттық бағдарламалаудағы рөлі орасан зор, сондықтан да математикалық ойлау қабілетінсіз бұл салада үздік болу мүмкін емес. Бұл бөлімде алдағы уақытта қажет болатын тұжырымдамалар мен формулаларды талдаймыз.

Қосынды формуласы

$$\sum_{x=1}^n x^k = 1^k + 2^k + 3^k + \dots + n^k,$$

формасындағы k бүтін сан болған жағдайда, қосындының формуласын $k+1$ дәрежесіндегі көпмүше арқылы келтіре аламыз. Мысалы, ¹,

$$\sum_{x=1}^n x = 1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

және

$$\sum_{x=1}^n x^2 = 1^2 + 2^2 + 3^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}.$$

¹ Осындай қосындылар үшін жалпы формула - Фаульхабер формуласы бар. Бірақ ол күрделі болғандықтан кітапта көрсетілмеді.

Арифметикалық прогрессия – әр екі көршілес тұрған санның айырмасы тұрақты санға тең болатын тізбек. Төмендегі тізбек –

$$3, 7, 11, 15$$

тұрақты саны 4 болатын арифметикалық прогрессия. Арифметикалық прогрессиялы тізбектің сомасын табатын формула:

$$\underbrace{a + \dots + b}_{n \text{ numbers}} = \frac{n(a + b)}{2}$$

a – ең алғашқы, ал b – ең соңғы сан болса, n – тізбектегі сандар мөлшері. Мысалы,

$$3 + 7 + 11 + 15 = \frac{4 \cdot (3 + 15)}{2} = 36.$$

тізбегінің тұрақты саны 4. Формулада n саннан тұратын тізбектегі әр сан орташа есеппен $(a + b)/2$ тең екенін негізге аламыз.

Геометриялық прогрессия – әр көршілес санның қатынасы тұрақты болатын сандар тізбегі.

Төмендегі тізбек:

$$3, 6, 12, 24$$

тұрақты саны 2-ге тең геометриялық прогрессия. Қосындысын есептеу формуласы:

$$a + ak + ak^2 + \dots + b = \frac{bk - a}{k - 1}$$

a – алғашқы сан, b – соңғы сан, ал k – екі көршілес санның қатынасы. Мысалы,

$$3 + 6 + 12 + 24 = \frac{24 \cdot 2 - 3}{2 - 1} = 45.$$

Формуланың шығу жолы:

$$S = a + ak + ak^2 + \dots + b.$$

болса, екі жақты k санына көбейтсек

$$kS = ak + ak^2 + ak^3 + \dots + bk,$$

өрнегі пайда болады. Теңдеуді шешу арқылы

$$kS - S = bk - a$$

формуласы қалады.

Геометриялық прогрессияның қосындысы үшін дербес жағдай формуласы:

$$1 + 2 + 4 + 8 + \dots + 2^{n-1} = 2^n - 1.$$

Гармоникалық қосынды –

$$\sum_{x=1}^n \frac{1}{x} = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}.$$

формасындағы қосынды. Гармоникалық қосындының үстіңгі шегі $\log_2(n) + 1$. Анығырақ айтсақ, әр $1/k$ өрнегіндегі k санын k -дан аспайтын және оған ең жақын болатын 2-нің дәрежесіне ауыстырамыз. Мысалы, $n = 6$ болған жағдайда қосындының өзгерісі:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} \leq 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}.$$

Жоғарғы шек $\log_2(n) + 1$ бөліктен тұрады $(1, 2 \cdot 1/2, 4 \cdot 1/4 \dots)$, және ең көп дегенде әр бөлік 1-ге тең болады.

Жиын теориясы

Жиын – элементтер топтамасы. Мысалы,

$$X = \{2, 4, 7\}$$

жиыны 2, 4 және 7 элементтерін қамтиды. \emptyset символы бос жиынды, ал $|S|$ S жиындардың өлшемін, яғни элементтер санын көрсетеді. Мысалға қайта оралсақ, $|X| = 3$.

S жиыны x элементін қамтитынын $x \in S$ белгісімен көрсетеміз, кері нұсқасы – $x \notin S$. Жоғарыда қарастырған жиынға қатысты мысал:

$$4 \in X \quad \text{және} \quad 5 \notin X.$$

Жиын операциялары арқылы жаңа жиындар құрастыра аламыз:

- $A \cap B$ қиылысуы A және B жиындарының екеуіне де тән элементтерден құралған. Мысалы, $A = \{1, 2, 5\}$, $B = \{2, 4\}$. Сәйкесінше, $A \cap B = \{2\}$.
- $A \cup B$ бірігуі – екі жиында да немесе кем дегенде екеуінің бірінде кездестірген элементтерді қамтиды. $A = \{3, 7\}$ және $B = \{2, 3, 8\}$ жиындары болса, олардың бірігуі үлгісі: $A \cup B = \{2, 3, 7, 8\}$.
- \bar{A} толықтауышы – A жиынында жоқ элементтер. Толықтауыштың түсіндірмесі болуы мүмкін барлық элементтерді қамтитын әмбебап жиынға тәуелді. Егер $A = \{1, 2, 5, 7\}$, ал әмбебап жиын $\{1, 2, \dots, 10\}$ болса, $\bar{A} = \{3, 4, 6, 8, 9, 10\}$.
- $A \setminus B = A \cap \bar{B}$ айырмашылығы – A жиынында болғанымен B жиынында жоқ элементтер. Бұл жерде B элементтері A элементтерінің қатарында жоқ екендігіне назар аудару керек: Егер $A = \{2, 3, 7, 8\}$ және $B = \{3, 5, 8\}$ болса, $A \setminus B = \{2, 7\}$.

A элементтері толықтай S жиынына кірсе, A S -тің ішжиыны, белгіленуі: $A \subset S$. S жиынында бос жиындарды санағанда әрдайым $2^{|S|}$ ішжиын бар. Мысалы, $\{2, 4, 7\}$ үшін ішжиындар:

$$\emptyset, \{2\}, \{4\}, \{7\}, \{2, 4\}, \{2, 7\}, \{4, 7\} \text{ және } \{2, 4, 7\}.$$

Жиі қолданылатын жиындар: \mathbb{N} (натурал сандар), \mathbb{Z} (бүтін сандар), \mathbb{Q} (рационал сандар) мен \mathbb{R} (нақты сандар). \mathbb{N} жиынын туындаған жағдайға байланысты екі түрде қарастыра аламыз: $\mathbb{N} = \{0, 1, 2, \dots\}$ немесе $\mathbb{N} = \{1, 2, 3, \dots\}$.

Сонымен қатар жиынды келесі форманы қолдану арқылы құрастыра аламыз:

$$\{f(n) : n \in S\},$$

$f(n)$ қандай да бір функцияны меңзейді. Жиын $f(n)$ формасындағы барлық мүмкін мәндерді қамтиды, n S элементі. Үлгідегі жиын барлық жұп сандарды қамтиды:

$$X = \{2n : n \in \mathbb{Z}\}$$

Логика

Логикалық өрнектің тек true (1) немесе false (0) мәндері бар. Ең маңызды логикалық операциялар: \neg (терістеу), \wedge (конъюнкция), \vee (дизъюнкция), \Rightarrow (импликация) мен \Leftrightarrow (эквиваленттеу). Төмендегі кестеде операторлар мағыналарымен көрсетілген:

A	B	$\neg A$	$\neg B$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0
1	0	0	1	0	1	0	0
1	1	0	0	1	1	1	1

$\neg A$ өрнегі – A - ға теріс мән. $A \wedge B$ өрнегі – егер A мен B екеуі де ақиқат болса, ақиқат, ал $A \vee B$ өрнегі – A мен B -ның бірі немесе екеуі де ақиқат болса, ақиқат. $A \Rightarrow B$ өрнегі – әрдайым A ақиқат және B да ақиқат болса, ақиқат. $A \Leftrightarrow B$ өрнегі – егер A мен B екеуі де ақиқат немесе жалған болса, ақиқат.

Предикат – ақиқат не жалған болуы параметрлерге байланысты өрнек. Предикаттар әдетте бас әріппен белгіленеді. Мысалы, $P(x)$ x саны жай болғанда ғана ақиқат деп белгілейік. Сәйкесінше, $P(7)$ – ақиқат, $P(8)$ – жалған.

Квантор логикалық өрнекті жиын элементтерімен байланыстырады. Ең маңызды кванторлар – \forall (жалпылық) және \exists (бар болу). Мысалы,

$$\forall x(\exists y(y < x))$$

жиындағы әр x элементі үшін өзінен кішірек болатын y элементі бар деген мағынаны білдіреді. Бұл бүтін сандар жиыны үшін ақиқат болса, натурал сандар жиыны үшін жалған.

Жоғарыда берілген түсініктемеге сәйкес көптеген логикалық тұжырымдарды өрнектей аламыз.

Үлгіде егер x 1-ден үлкен және жай сан болмаса,

$$\forall x((x > 1 \wedge \neg P(x)) \Rightarrow (\exists a(\exists b(a > 1 \wedge b > 1 \wedge x = ab))))$$

. демек екеуі де 1-ден үлкен, көбейтінділері x -ке тең a және b сандары бар. Тұжырымдама – бүтін сандар үшін ақиқат.

Функциялар

$\lfloor x \rfloor$ функциясы x санын бүтінге дейін төменге дөңгелектейді, ал $\lceil x \rceil$ x санын бүтінге дейін жоғарыға дөңгелектейді. Мысалы,

$$\lfloor 3/2 \rfloor = 1 \quad \text{және} \quad \lceil 3/2 \rceil = 2.$$

$\min(x_1, x_2, \dots, x_n)$ және $\max(x_1, x_2, \dots, x_n)$ функциялары x_1, x_2, \dots, x_n аралығындағы ең аз және ең көп сәйкес мәндерді қайтарады. Мысалы,

$$\min(1, 2, 3) = 1 \quad \text{және} \quad \max(1, 2, 3) = 3.$$

$n!$ факториалы төмендегі формула бойынша:

$$\prod_{x=1}^n x = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$$

немесе рекуррентті анықталады:

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

Фибоначчи сандары көп жағдайда туындайды. Оларды төмендегі рекуррентті қатынас арқылы анықтауға болады:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

Алғашқы Фибоначчи сандары:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

Фибоначчи сандарын анықтау үшін келесі тұйық формуланы қолдана аламыз. Ол кейде Бине формуласы деп те аталады:

$$f(n) = \frac{(1 + \sqrt{5})^n - (1 - \sqrt{5})^n}{2^n \sqrt{5}}.$$

Логарифмдер

x санының логарифмі $\log_k(x)$ ретінде белгіленеді, бұл жердегі k логарифм негізі. Анықтама бойынша, $k^a = x$ болған жағдайда ғана $\log_k(x) = a$.

Логарифмнің пайдалы қасиеті – $\log_k(x)$ арқылы x -тен 1-ге жету үшін k санына қанша мәрте бөлу қажеттігін есептеуінде. Мысалы, $\log_2(32) = 5$, себебі 2-ге 5 рет бөлеміз:

$$32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

Алгоритмдерге талдау жасауда логарифмдер жиі қолданылады, себебі көптеген тиімді алгоритмдер әр қадам сайын әлденені екі есеге қысқартады.

Осылайша, кей алгоритмдердің тиімділігіне логарифмдер арқылы баға бере аламыз.

Көбейтінді логарифмі

$$\log_k(ab) = \log_k(a) + \log_k(b),$$

сәйкесінше,

$$\log_k(x^n) = n \cdot \log_k(x).$$

Сондай-ақ, бөлінді логарифмі

$$\log_k\left(\frac{a}{b}\right) = \log_k(a) - \log_k(b).$$

Тағы бір пайдалы формула:

$$\log_u(x) = \frac{\log_k(x)}{\log_k(u)},$$

оның көмегімен кез-келген негіздегі логарифмді әлдеқандай белгілі негіз арқылы таба аламыз.

$\ln(x)$ негізі $e \approx 2.71828$ болатын x санының натурал логарифмін білдіреді. Логарифмдердің қасиеті: b негізіндегі x бүтін санының цифрлар мөлшерін $\lfloor \log_b(x) + 1 \rfloor$ арқылы біле аламыз. Мысалы, 123-тің негізі 2 болса, көрсетілімі – 1111011, сәйкесінше, $\lfloor \log_2(123) + 1 \rfloor = 7$ болады.

1.6 Контесттер және ресурстар

IOI

Информатикадан Халықаралық Олимпиада(IOI) – жоғарғы сынып оқушыларына арналған бағдарламалау контесті. Жарысқа әр елден 4 оқушыдан құралған команда қатыса алады. Әдетте шамамен 80 елден жиналған 300 қатысушы арасында өтеді.

IOI 5 сағатқа созылатын екі контесттен тұрады. Екі кезеңде де қатысушыларға 3 түрлі деңгейдегі алгоритмдік есептер ұсынылады. Олар ішесептерге бөлініп, әрбір ішесеп үшін нақты ұпайлар белгіленеді. Қатысушылар командаларға бөлінгенімен, әрқайсысы жеке дара контестке түседі.

IOI силлабусы [2] IOI есептерінің болжамды тақырыптарын реттейді. Кітапта IOI силлабусының түгелге дерлік тақырыптары қамтылған.

IOI қатысушылары ұлттық олимпиадалар арқылы іріктеледі. IOI-дан бұрын Балтық информатика олимпиадасы (BOI), Орталық Еуропа Информатикалық олимпиадасы (CEOI) және Азия-Тынық мұхиты Информатика Олимпиадасы (APIO) сияқты өңірлік контесттер ұйымдастырылады.

Кей елдерде болашақ IOI қатысушыларына арналған онлайн контесттер ұйымдастырылады. Мысалы, Croatian Open Competition in Informatics [3] және the USA Computing Olympiad [4]. Сондай-ақ көптеген Поляк контесттерінің есептері онлайн қолжетімді[5].

ICPC

Студенттердің Халықаралық Бағдарламалау Сайысы (ICPC) жыл сайын университет студенттеріне арнап өткізіледі. Әр командада 3 қатысушыдан болады. IOI-дан айырмашылығы – студенттер бірлесіп жұмыс жасайды, әр команда үшін бір компьютер беріледі.

ICPC бірнеше кезеңнен тұрады, үздік командалар Әлемдік Біріншілікке шақырылады. Мыңдаған қатысушылардың арасынан іріктеліп, финалға тек санаулысы өте алады¹, сондықтан финалға өтудің өзі үлкен жетістік деп саналады.

Әр ICPC контестінде 5 сағат ішінде командалар шамамен 10 алгоритмдік есеп шығаруы қажет. Есеп шешімі барлық тестілік жағдайды тиімді шешсе ғана қабылданады. Контест барысында командалар бір-бірінің нәтижелерін көре алады, бірақ соңғы сағаттағы нәтижелер мен соңғы жіберілген шешімдерді көру мүмкіндігі шектелген.

ICPC-да болжамды тақырыптар IOI-дағыдай көрсетілмейді. Десе де, ICPC терең білімді, әсіресе математикалық қабілеттілікті қажет етеді.

Онлайн контесттер

Жоғарыда аталған сайыстармен бірге көпшілікке арналған онлайн контесттерді де атап өтуге болады. Қазіргі таңдағы ең белсенді контесттер сайты – Codeforces, онда шамамен апта сайын контесттер ұйымдастырылып тұрады. Codeforces-та жаңа қатысушылар – Div2, ал тәжірибелі қатысушылар – Div1 деп аталатын екі дивизионға бөлінеді. Контесттер ұйымдастырылатын сайттардың қатарында AtCoder, CS Academy, HackerRank және Topcoder-ді де атап өтуге болады.

Кейбір компаниялар финалы офлайн болатын онлайн контесттерді ұйымдастырады. Мысалы: Facebook Hacker Cup, Google Code Jam мен Yandex.Algorithm. Әрине, компаниялар бұл контесттерді қызметкерлерді жинақтау үшін де өткізеді. Өйткені контестте жақсы нәтиже көрсету арқылы қатысушылар бағдарламалау қабілеттерін көрсетуге мүмкіндік алады.

Кітаптар

Спорттық бағдарламалау мен алгоритмдік есептерді шығаруға бағытталған бірнеше кітаптар тізімі (қолыңыздағы кітапты санамағанда) :

- S. S. Skiena and M. A. Revilla: Programming Challenges: The Programming Contest Training Manual [6]
- S. Halim and F. Halim: Competitive Programming 3: The New Lower Bound of Programming Contests [7]
- K. Diks et al.: Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions [8]

¹Қатысушылар саны жыл сайын өзгеріп отырады; 2017 жылғы қатысушылар саны – 133.

Алғашқы екі кітап жаңадан бастаушыларға арналса, соңғысы ілгері деңгейдегілерге арнап жазылған.

Спорттық бағдарламалауды зерделеуде пайдалы жалпы алгоритмдер жайлы кітаптар бар. Олардың ішіндегі кейбір танымалдары:

- T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein: Introduction to Algorithms [9]
- J. Kleinberg and É. Tardos: Algorithm Design [10]
- S. S. Skiena: The Algorithm Design Manual [11]

2-тарау. Уақытша күрделілігі

Алгоритмдердің тез жұмыс істеуінің спорттық бағдарламалауда маңызы зор. Әдетте есепті баяу шығаратын алгоритмді ойлап табу жеңіл-ақ. Ал сол есепті тез шығаратын алгоритмді ойлап табу қиынға соғады. Егер алгоритм өте баяу болса, ол есептен аз немесе нөл ұпай алады.

Алгоритмнің уақытша күрделілігі – берілген енгізу бойынша алгоритмнің қанша уақытта жұмыс жасайтындығының мөлшері. Уақытша күрделілігін енгізу бойынша функциямен көрсетеді және оны жазғанда $O(\dots)$ нотациясымен белгілейді. Алдағы уақытта бұған қайта ораламыз. Уақытша күрделілігін есептеу арқылы біз алгоритмді жазбастан бұрын оның есеп шығару үшін жылдамдығы жеткілікті дәрежеде екеніне көз жеткізе аламыз.

2.1 Есептеу ережелері

Алгоритмнің уақытша күрделілігі $O(\dots)$ түрінде белгіленеді, мұндағы көп нүкте қандай да бір функцияны білдіреді. Әдетте, n айнымалысы енгізудің өлшемін көрсетеді. Мысалы, егер енгізу сандар жиыны болса, n оның өлшемін көрсетеді. Егер жол болса, n оның ұзындығын көрсетеді.

Қайталымдар

Алгоритмнің баяу болуының әдеттегі себебі – енгізу бойынша өтетін кірістірілген циклдардың көп болуы. Егер k кірістірілген цикл болса, уақытша күрделілігі $O(n^k)$ -ға тең.

Мысалы, төмендегі кодтың уақытша күрделілігі – $O(n)$:

```
for (int i = 1; i <= n; i++) {  
    // code  
}
```

Ал келесі кодтың уақытша күрделілігі – $O(n^2)$:

```
for (int i = 1; i <= n; i++) {  
    for (int j = 1; j <= n; j++) {  
        // code  
    }  
}
```

Шаманың реті

Уақытша күрделілігі бізге циклдың ішінде алгоритмнің қанша рет жұмыс жасайтынын көрсетпейді. Негізінен ол бізге шаманың ретін ғана көрсетеді. Төмендегі мысалдарда алгоритмнің циклы $3n$, $n + 5$ және $\lceil n/2 \rceil$ итерация жасайтын болса да, олардың уақытша күрделілігі $O(n)$ болады.

```
for (int i = 1; i <= 3*n; i++) {  
    // code  
}
```

```
for (int i = 1; i <= n+5; i++) {  
    // code  
}
```

```
for (int i = 1; i <= n; i += 2) {  
    // code  
}
```

Ал мына мысалда уақытша күрделілігі $O(n^2)$ -не тең, себебі $1+2+3+\dots+n = \frac{n*(n+1)}{2}$ және $O(\frac{n^2}{2} + \frac{n+1}{2}) = O(n^2)$ (алдағы уақытта кеңірек тоқталамыз):

```
for (int i = 1; i <= n; i++) {  
    for (int j = i+1; j <= n; j++) {  
        // code  
    }  
}
```

Бөлімдер

Егер кодтағы уақытша күрделілігі бірнеше бөлімдерден тұратын болса, онда жалпы уақытша күрделілігі оның ең баяу бөліміне тең болады. Себебі кодтағы ең баяу бөлім әдетте кодтың тар өткеліне айналады. Егер сол баяу бөлімді тұрақты санға көбейтетін болсақ, оны өшіре салуға болады (яғни $O(3n) = O(n)$).

Мысалы, төмендегі код үш бөлімнен тұрады: $O(n)$, $O(n^2)$ және $O(n)$. Демек жалпы уақытша күрделілігі $O(n^2)$ -ге тең.

```
for (int i = 1; i <= n; i++) {  
    // code  
}  
for (int i = 1; i <= n; i++) {  
    for (int j = 1; j <= n; j++) {  
        // code  
    }  
}
```

```
for (int i = 1; i <= n; i++) {  
    // code  
}
```

Бірнеше айнымалылар

Кейде алгоритмнің уақытша күрделілігі бірнеше факторларға тәуелді болады. Сол себепті де уақытша күрделілігінде бірнеше айнымалылар кездеседі.

Төмендегі кодтың уақытша күрделілігі $O(nm)$ -ға тең:

```
for (int i = 1; i <= n; i++) {  
    for (int j = 1; j <= m; j++) {  
        // code  
    }  
}
```

Рекурсия

Рекурсиялық функцияның уақытша күрделілігі функцияның қанша рет шақырылғанына және бір шақырылымның уақытша күрделілігіне байланысты болады. Ал қорытынды уақытша күрделілігі олардың көбейтіндісіне тең келеді.

Мысалға келесі рекурсиялық функцияны қарастырайық:

```
void f(int n) {  
    if (n == 1) return;  
    f(n-1);  
}
```

$f(n)$ функциясын шақыру n рет сол функцияны орындайды және әрбір шақырудың уақытша күрделілігі $O(1)$ болады. Демек қорытынды уақытша күрделілігі $O(n)$ -ға тең.

Келесі мысалға назар аударайық:

```
void g(int n) {  
    if (n == 1) return;  
    g(n-1);  
    g(n-1);  
}
```

Бұл жерде әр функция шақырылуы $n \neq 1$ болған кезде функцияны 2 рет шақырады. Егер біз $g(n)$ функциясын шақырсақ, ол 2 $g(n-1)$ функциясын шақырады. Ал $g(n-1)$ болса, 4 $g(n-2)$ функциясын шақыратын болады. Дәл солай кете берсек, бізде төмендегідей кесте шығады:

Функция	неше рет шақырылды
$g(n)$	1
$g(n-1)$	2
$g(n-2)$	4
...	...
$g(1)$	2^{n-1}

Осыған байланысты біздің уақытша күрделілігіміз:

$$1 + 2 + 4 + \dots + 2^{n-1} = 2^n - 1 = O(2^n).$$

2.2 Күрделілік кластары

Төмендегі тізімде алгоритмдердің уақытша күрделіліктеріне сипаттама беріледі:

$O(1)$ – Алгоритмнің уақытпен жұмыс істеу кезеңі кіріс деректердің көлеміне байланысты болмайды. Бұның қарапайым мысалы ретінде сол бойынша жауапты есептеп шығаратын айқын формуланы алуға болады.

$O(\log n)$ – Логарифмдік алгоритмде кіріс деректердің көлемі әр қадам сайын әдетте екіге кеміп отырады. Жұмыс істеу уақыты кіріс деректердің көлеміне байланысты болады. Өйткені мұндай алгоритмге логарифмдік жұмыс уақыты тән. Себебі n -ді екіге бөле берсек, ақырында $\log n$ рет бөлуден кейін 1-ге жетеміз.

$O(\sqrt{n})$ – Уақытша күрделілігі $O(\sqrt{n})$ алгоритмдер $O(\log n)$ -нан баяуырақ, бірақ $O(n)$ -нен жылдамырақ болады. Квадратты түбірдің ерекше қасиеті – $\sqrt{n} = n/\sqrt{n}$ болуында. Демек n элементті $O(\sqrt{n})$ элементтер бойынша $O(\sqrt{n})$ бөлікке бөлшектей аламыз.

$O(n)$ – Сызықтық алгоритм кіріс деректерді тұрақты уақытта өтіп шығады. Бұл көбіне алгоритмнің ең жақсы ықтимал уақытша күрделілігі болып келеді. Себебі жауапты алу үшін әр элементке тым болмаса бір рет жүгіну керек болады.

$O(n \log n)$ – Алгоритмнің мұндай уақытша күрделілігі әдетте оның кіріс деректерді сұрыптайтын білдіреді. Себебі сұрыптау алгоритмдерінің ең тиімді уақытша күрделілігі $O(n \log n)$ -ге тең. Алгоритмде әр операция $O(\log n)$ уақыт алатын деректер құрылымы қолданылатын басқа мүмкіндіктер де бар.

$O(n^2)$ – Квадраттық алгоритм әдетте екі кірістірілген циклден тұрады. Кіріс элементтерінің барлық жұптарын өтіп шығу $O(n^2)$ уақытты алады.

$O(n^3)$ – Кубтық алгоритм әдетте үш кірістірілген циклден тұрады. Кіріс элементтердің барлық үштіктерін өтіп шығу $O(n^3)$ уақытты алады.

$O(2^n)$ – Алгоритмнің мұндай уақытша күрделілігі әдетте алгоритмнің кіріс деректер жиынының барлық ішжиынын өтіп шыққанына нұсқайды. Мысалы, $\{1, 2, 3\}$ элементтерінің барлық ішжиыны \emptyset , $\{1\}$, $\{2\}$, $\{3\}$, $\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$ және $\{1, 2, 3\}$ болады.

$O(n!)$ – Алгоритмнің мұндай уақытша күрделілігі әдетте алгоритмнің кіріс элементтердің барлық алмастыруларын өтіп шыққанына нұсқайды. Мысалы, $\{1, 2, 3\}$ элементтердің барлық алмастыруы $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$ және $(3, 2, 1)$ болады.

Уақытша күрделілігі ең көп дегенде $O(n^k)$ -не тең болатын алгоритм полиномдық алгоритм деп аталады. Мұндағы k тұрақты. $O(2^n)$ және $O(n!)$ алгоритмдерінен басқа барлық алгоритм полиномдық алгоритм болып есептеледі. Іс жүзінде мұндай алгоритмдер тиімді, себебі k көп жағдайда аз болады.

Бұл кітаптағы алгоритмдердің көбі - полиномдық алгоритмдер. Дегенмен кейбір маңызды есептерді полиномдық уақытта шығару мүмкін емес. Басқаша айтқанда, ондай есептерді ешкім тиімді шығара алмайды. Бұл есептердің жиынын NP-қиын есептер деп атайды және оларды шешетін полиномдық алгоритмдерді әзірше ешкім білмейді ¹.

2.3 Тиімділікті бағалау

Алгоритмнің уақытша күрделілігін есептей отырып, алгоритмді жазбас бұрын оның есеп үшін жеткілікті дәрежеде тиімді екенін тексеріп алуға болады. Бағалауды бастамас бұрын қазіргі заманғы компьютер бір секундта жүз миллиондаған операция орындай алатынын ескеру қажет.

Мысалы, бір есептің уақыт шегі 1 секунд, ал енгізудің өлшемі $n = 10^5$ дейік. Егер уақытша күрделілігі $O(n^2)$ болса, алгоритм $(10^5)^2 = 10^{10}$ операция жасайды. Бұл алгоритм ең кемінде 10 секунд алады. Есепті шешу үшін тым баяу болғандықтан, қолданысқа қолайлы емес екенін байқаймыз.

Екінші жағынан, берілген енгізудің өлшемі арқылы есепті шығаратын алгоритмнің уақытша күрделілігін болжай аламыз. Келесі кестеде уақыт шегі 1 секунд болатын есептерге көмектесетін алгоритмдердің болжамды бағалары берілген.

енгізу өлшемі	талап етілетін уақытша күрделілігі
$n \leq 10$	$O(n!)$
$n \leq 20$	$O(2^n)$
$n \leq 500$	$O(n^3)$
$n \leq 5000$	$O(n^2)$
$n \leq 10^6$	$O(n \log n)$ немесе $O(n)$
n тым үлкен	$O(1)$ немесе $O(\log n)$

¹Осы тақырып бойынша классикалық кітаптардың бірі – М.Р.Гэрей мен Д.С.Джонсонның Computers and Intractability: A Guide to the Theory of NP-Completeness атты кітабы [12].

Мысалы, егер енгізу өлшемі $n = 10^5$ болса, онда алгоритмнің уақытша күрделілігі $O(n)$ немесе $O(n \log n)$ болады деп күтілуде. Бұл мағлұмат алгоритмдерді жобалауға көмек береді, өйткені уақытша күрделілігі баяу алгоритмдерді қарастырған тиімсіз.

Уақытша күрделілігі кодтың тиімділігін бағалайтынын естен шығармау керек. Ол кодтың нақты неше операция жасайтынын көрсетпейді және ол тұрақты факторларды жасырады. Мысалы, $O(n)$ жасайтын алгоритм шын мәнінде $n/2$ немесе $5n$ операция жасауы мүмкін. Бұл алгоритмнің жұмыс жасау уақытына сөзсіз әсер етеді.

2.4 Ішжиымдардың ең жоғары қосындысы

Бір есептің шығарылу барысында уақытша күрделілігі әртүрлі бірнеше алгоритмдер кездеседі. Бұл бөлімде классикалық есепті $O(n^3)$ уақытта шығару жолын талқылаймыз. Дегенмен жақсырақ алгоритмді жобалау арқылы бұл есепті $O(n^2)$ немесе одан анағұрлым тиімдірек $O(n)$ уақытта шешуге болады.

Бізге өлшемі n болатын жиым берілген. Оның ішжиымдарының ең жоғары қосындысын табуымыз керек. Басқаша айтқанда жиымдағы көршілес болатын тізбектердің ең жоғары қосындысының мәнін табу¹ қажет. Егер жиымда теріс сандар болса, бұл есеп қызығырақ шығарылады. Мысалы, мына жиымда

-1	2	4	-3	5	2	-5	2
----	---	---	----	---	---	----	---

ішжиымдардың максималды қосындысы 10 болады:

-1	2	4	-3	5	2	-5	2
----	---	---	----	---	---	----	---

Бос ішжиымның қосындысы 0 деп санайық. Есеп шарты бойынша бос ішжиым алуға болатынын ескеруіміз керек. Демек ішжиымдардың ең жоғары қосындысы кем дегенде 0 болады.

1-алгоритм

Барлық ішжиымдарды қарап, олардың қосындысын бір айнымалыға сақтау арқылы бұл есепті ең оңай алгоритммен шығаруға болады. Осы алгоритмді төмендегі код іске асырады:

```
int best = 0;
for (int a = 0; a < n; a++) {
    for (int b = a; b < n; b++) {
        int sum = 0;
        for (int k = a; k <= b; k++) {
            sum += array[k];
        }
    }
}
```

¹Дж. Бэнтлейдің Programming Pearls [13] кітабы есепті танымал етті.


```

        best = max(best,sum);
    }
}
cout << best << "\n";

```

a мен b итераторлары ішжиымның бірінші және соңғы индекстері ретінде қарастырылады. Араларындағы элементтердің ең аз қосындысын sum айнымалысында, ал ең көп қосындыны $best$ айнымалысында сақтаймыз.

Бұл алгоритмнің уақытша күрделілігі $O(n^3)$, себебі алгоритмде үш кірістірілген цикл енгізуді өтіп шығады.

2-алгоритм

1-алгоритмнен бір циклді алып тастау арқылы оны тиімдірек жобалай аламыз. Ішжиымның соңғы индексі өзгерген сәтте ғана қосындыны санауға болады. Нәтижесінде код төмендегідей болып өзгереді:

```

int best = 0;
for (int a = 0; a < n; a++) {
    int sum = 0;
    for (int b = a; b < n; b++) {
        sum += array[b];
        best = max(best,sum);
    }
}
cout << best << "\n";

```

Бұл өзгерістен кейін уақытша күрделілігі $O(n^2)$ болады.

3-алгоритм

Бір қарағанда мүмкін еместей болып көрінгенімен бұл есепті $O(n)$ уақытта шығаруға болады ¹. Нақтырақ айтқанда, бір ғана цикл арқылы шығара аламыз. Идеясы әрбір индекс үшін сол жерден бітетін ең жоғары ішжиымның қосындысын есептеуге, ал кейін солардың максимумын табуға негізделеді.

k индексінде бітетін ең жоғары ішжиым есебін қарастырайық. Бұл жерде екі нұсқа бар:

1. Ең жоғары ішжиым индексі тек k болатын элементтен тұрады.
2. Ең жоғары ішжиым $k - 1$ индексінде аяқталатын ішкі жиымнан және одан кейінгі k индексіндегі элементтен тұрады.

Екінші жағдайда қосындысы ең жоғары ішжиымды іздейтіндіктен, $k - 1$ позициясында аяқталатын ішжиымның қосындысы да ең жоғары болуы керек. Демек бұл есептің тиімді шығарылу жолы солдан оңға қарай әр индекске сол жерден бітетін ең жоғары қосындыны есептеу болмақ.

¹[13]-еңбекте бұл сызықтық уақыт алгоритмінің Дж.Б.Каданеге қатыстылығы көрсетіледі. Оны кейде Кадане алгоритмі деп те атайды.

Бұл жағдайда төмендегі алгоритм коды тиімді:

```
int best = 0, sum = 0;
for (int k = 0; k < n; k++) {
    sum = max(array[k], sum+array[k]);
    best = max(best, sum);
}
cout << best << "\n";
```

Алгоритмді енгізу бойынша бір ғана цикл өтетіндіктен оның уақытша күрделілігі $O(n)$ болады және бұл ең тиімді алгоритм саналады.

Тиімділікті салыстыру

Енді аталған алгоритмдердің тиімділігін іс жүзінде байқап көрейік. Төмендегі кестеде үш алгоритмнің n әртүрлі мәндерге тең болғандағы заманауи компьютердегі жұмыс уақыты берілген.

Әр тест кездейсоқ мәндер арқылы құрастырылып, енгізуді оқу уақыты қарастырылмаған.

Енгізу өлшемі n	1-алгоритм	2-алгоритм	3-алгоритм
10^2	0.0 с	0.0 с	0.0 с
10^3	0.1 с	0.0 с	0.0 с
10^4	> 10.0 с	0.1 с	0.0 с
10^5	> 10.0 с	5.3 с	0.0 с
10^6	> 10.0 с	> 10.0 с	0.0 с
10^7	> 10.0 с	> 10.0 с	0.0 с

Бұл кесте егер енгізу өлшемі кішкентай болса, барлық алгоритм тиімді жұмыс істейтінін көрсетеді. Алайда енгізу өлшемі ұлғайған сайын, алгоритмдердің жұмыс уақытында үлкен айырмашылық пайда болатынын байқаймыз. $n = 10^4$ болған сәтте 1-алгоритм баяу болса, $n = 10^5$ болғанда 2-алгоритм баяулайды. Тек 3-алгоритм ғана үлкен енгізулерді бірден өңдеуге қабілетті.

3-тарау. Сұрыптау

Сұрыптау – алгоритмдерді жобалаудағы іргелі әдіс-тәсілдердің бірі. Ол көптеген тиімді алгоритмдерде ішбағдарлама ретінде қолданылады. Өйткені сұрыпталған енгізулермен есеп оңай шығарылады.

Мысалы, ”жиымда екі бірдей элемент бар ма?” деген сұраққа сұрыптау арқылы жауап берген ыңғайлы. Жиымда екі бірдей элемент болса, сұрыптаудан кейін олар бір-бірінің жанына орналасады. Осылайша олар оңай анықталады. Сонымен қатар ”ең жиі кездесетін элементті табу” есебін де ұқсас жолмен шеше аламыз.

Сұрыптау алгоритмдері өте көп, және оларды өзге алгоритмдерді құруда үлгі ретінде алуға болады. Тиімді жалпы сұрыптау алгоритмдері $O(n \log n)$ уақытта жұмыс жасайды. Сұрыптауды ішбағдарлама ретінде қолданатын алгоритмдердің уақытша күрделілігі де көбіне соған тең болып келеді.

3.1 Сұрыптау теориясы

Сұрыптаудағы негізгі есептердің бірі :

n элементтен тұратын жиым берілген, оны өсу реті бойынша жүйеге келтіруіміз қажет.

Мысалы, мына жиым

1	3	8	2	9	2	5	6
---	---	---	---	---	---	---	---

сұрыптаудан кейін төмендегідей болып өзгереді:

1	2	2	3	5	6	8	9
---	---	---	---	---	---	---	---

$O(n^2)$ алгоритмдері

Қарапайым сұрыптау алгоритмдері $O(n^2)$ уақытта жұмыс жасайды. Олар ықшам болады және әдетте екі циклден тұрады. $O(n^2)$ уақытта жұмыс істейтін сұрыптау алгоритмдерінің ішіндегі ең танымалы – көпіршікті сұрыптау. Ол n кезеңнен тұрады. Әр кезеңде алгоритм жиым ішімен өтіп шығады. Екі қатар келетін элемент бұрыс тұрса, оларды алмастырады.

Алгоритм төменде көрсетілген жолмен орындалады:

```

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n-1; j++) {
        if (array[j] > array[j+1]) {
            swap(array[j], array[j+1]);
        }
    }
}

```

1-кезеңнен кейін ең үлкен элемент өз орнына келеді, сәйкесінше k кезеңнен соң k үлкен сан нақты өз орнында болады. Осылайша, n кезеңде барлық жиымды сұрыптай аламыз.

Мысалы, жиымда

1	3	8	2	9	2	5	6
---	---	---	---	---	---	---	---

1-кезеңдегі алмасулар:

1	3	2	8	9	2	5	6
---	---	---	---	---	---	---	---



1	3	2	8	2	9	5	6
---	---	---	---	---	---	---	---



1	3	2	8	2	5	9	6
---	---	---	---	---	---	---	---



1	3	2	8	2	5	6	9
---	---	---	---	---	---	---	---



Инверсиялар

Көпіршікті сұрыптау қатар тұрған элементтерді алмастыру арқылы жүзеге асады. Осылайша ол үнемі кем дегенде $O(n^2)$ уақытта жұмыс істейді. Өйткені нашар жағдайда бізге $O(n^2)$ алмастырулар жасауға тура келер еді.

Сұрыптау алгоритмдерін талдауда инверсия, яғни элементтер жиымының ($array[a], array[b]$) жұбы үшін $a < b$ және $array[a] > array[b]$ орындалып, элементтердің бұрыс ретпен орналасуы пайдалы болмақ. Мысалы, келесі жиымда

1	2	2	6	3	5	9	8
---	---	---	---	---	---	---	---

үш инверсия бар: (6,3), (6,5) және (9,8). Инверсиялар саны сұрыптау үшін қаншалықты жұмыс істеу қажеттілігін анықтайды. Инверсияның болмауы жиымның сұрыпталғаны жайлы хабар береді. Ал егер, жиым кері реттілікте болса, инверсиялар санының болжамды ең үлкен мәні төмендегідей болмақ:

$$1 + 2 + \dots + (n - 1) = \frac{n(n - 1)}{2} = O(n^2)$$

Қате ретте орналасқан көршілес элементтер жұптарының алмасуы инверсиялар санын бірге азайтып отырады. Осылайша, алгоритм жоғарыда келтірілген әрекет арқылы жүзеге асса, әр алмасу көп дегенде 1 инверсияны жойып отырады, сол себепті алгоритмнің уақытша күрделілігі қалай болғанда да $O(n^2)$.

$O(n \log n)$ алгоритмдер

Жиымды $O(n \log n)$ уақыт ішінде тезірек сұрыптауды көршілес орналасқан элементтерді алмастырумен шектелмейтін алгоритм арқылы да жүзеге асыру мүмкіндігі бар. Осындай алгоритмдердің бірі рекурсияға негізделген біріктіру бойынша сұрыптау деп аталады¹.

Ол $\text{аггау}[a \dots b]$ ішжиымын төмендегідей сұрыптайды:

1. $a = b$ болған жағдайда, еш әрекет жасамаймыз, себебі ішжиым әлдеқашан сұрыпталған.
2. Центрде орналасқан элементтің позициясын анықтаймыз : $k = \lfloor (a + b)/2 \rfloor$.
3. Рекурсивті $\text{аггау}[a \dots k]$ ішжиымын сұрыптаймыз.
4. $\text{аггау}[k + 1 \dots b]$ ішжиымын да дәл осылай сұрыптаймыз.
5. Сұрыпталған $\text{аггау}[a \dots k]$ және $\text{аггау}[k + 1 \dots b]$ ішжиымдарын $\text{аггау}[a \dots b]$ біріктіреміз.

Бағдарламалауда бұл алгоритм тиімді саналады. Себебі ол әр қадам сайын ішжиым өлшемін екі есеге азайтып отырады. Рекурсия $O(\log n)$ деңгейден тұрады және әр деңгейді өңдеуге $O(n)$ уақыт кетеді. Ал $\text{аггау}[a \dots k]$ мен $\text{аггау}[k + 1 \dots b]$ ішжиымдарын біріктіру олар әлдеқашан сұрыпталғандықтан сызықты уақытты алады.

Мысалы, келесі жиымды сұрыптау жолын қарастырайық:

1	3	6	2	8	2	5	9
---	---	---	---	---	---	---	---

Жиымды екіге бөлейік:

1	3	6	2
8	2	5	9

Кейін ішжиымдар рекурсивті түрде сұрыпталады:

¹[14] мұндай сұрыптауды Дж. фон Нейман 1945 жылы ойлап тапты.

1	2	3	6
---	---	---	---

2	5	8	9
---	---	---	---

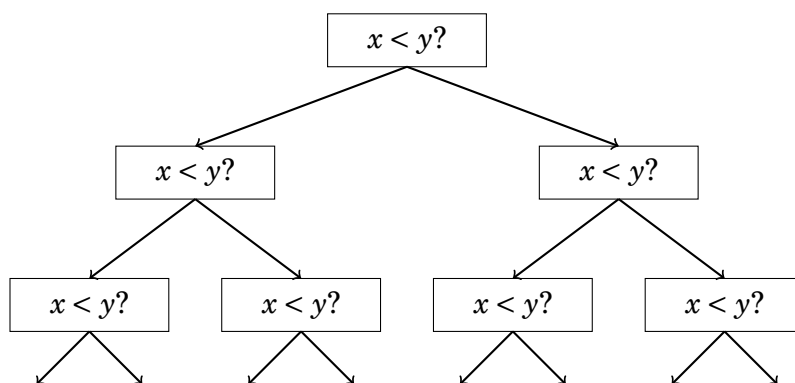
Соңында алгоритм барлық ішжиымдардан сұрыпталған жиым құрайды:

1	2	2	3	5	6	8	9
---	---	---	---	---	---	---	---

Сұрыптаудың төменгі шегі

” $O(n \log n)$ уақытынан жылдамырақ сұрыптау мүмкін бе?” – деген заңды сұрақ туындайды. Бұл сұраққа: ”Егер алмастыру алгоритмдерімен шектелсек, мүмкін емес” – деп жауап беруге болады.

Уақытша күрделілігінің төменгі шегін дәлелдеу үшін сұрыптауды әр салыстыру сайын жиым туралы молырақ ақпарат беретін үдеріс деп қарастырайық. Бұл үдерістен төмендегідей дарақ шығады:



Бұл жердегі ” $x < y?$ ” қандай да бір x пен y элементтерінің салыстырылуын білдіреді. Егер $x < y$ болса, үдеріс солға жалғасады. Ал кері жағдайда оңға қарай ауысады. Нәтижелер – жиымды сұрыптаудың $n!$ болжамды жолдары. Сол себепті, дарақтың биіктігі кем дегенде

$$\log_2(n!) = \log_2(1) + \log_2(2) + \dots + \log_2(n).$$

Қосындының төменгі шегін соңғы $n/2$ элементті таңдап, әрқайсының мәнін $\log_2(n/2)$ -ге өзгерту арқылы анықтай аламыз. Беретін бағасы

$$\log_2(n!) \geq (n/2) \cdot \log_2(n/2),$$

сондықтан дарақтың биіктігі мен сұрыптау алгоритмі ең нашар жағдайда жасайтын минималды жүріс саны $n \log n$ - ге тең.

Санамалы сұрыптау

$n \log n$ элементтерді салыстырмай, өзге ақпараттарды қолдана отырып сұрыптайтын алгоритмдер үшін төменгі шек бола алмайды. Осындай алгоритмдердің бірі – $O(n)$ уақытта жұмыс жасайтын санамалы сұрыптау, ол жиымдағы элементтер $0 \dots c$ аралығында деген болжамға негізделген және $c = O(n)$.

Біз индекстері бастапқы жиымның элементтерімен сәйкес келетін көмекші жиымды қолданатын боламыз. Алгоритм бастапқы жиымды айналып шығып, онда әр элементтің қанша рет кездесетінін есептейді.

Мысалы, мына жиымнан

1	3	6	9	9	3	5	9
---	---	---	---	---	---	---	---

келесі есептік жиым туындайды:

1	2	3	4	5	6	7	8	9
1	0	2	0	1	1	0	0	3

Есептік жиымның 3-индексінде 2 саны сақталған, себебі 3 элементі негізгі жиымда 2 рет кездеседі.

Аталған жиым $O(n)$ уақытта құрылады. Ал сұрыпталған жиым құрау $O(n)$ уақыт алады. Себебі әр элементтің қанша рет кездескенін есептік жиымнан біле аламыз. Сондықтан санамалы сұрыптаудың уақытша күрделілігі $O(n)$ -ге тең.

Тұрақты c жеткілікті деңгейде кіші болған жағдайда есептік жиым құру мүмкіндігі артады. Сондықтан санамалы сұрыптау өте тиімді алгоритм болмақ.

3.2 C++ -тегі сұрыптау

Контексте қолдан құрастырылған сұрыптау алгоритмін пайдалану сирек жағдайларда ғана жақсы идея деп саналады. Өйткені бағдарламалау тілдерінде кірістірілген жақсы тәсілдер баршылық. Мысалы, C++ стандартты дерекханасында `sort` функциясы бар, оның көмегімен жиымдар және басқа да деректер құрылымдарын оңай сұрыптай аламыз.

Дерекхананың сұрыптау функциясын пайдаланудың көптеген артықшылықтары бар. Біріншіден, ол уақытты үнемдейді. Себебі функция кодын жазуға уақыт кетпейді. Екіншіден, оның жүзеге асыру тәсілі тиімді әрі дұрыс. Өйткені қолдан құрастырылған алгоритмнің жақсырақ болуы екіталай.

Бұл бөлікте C++ `sort` функциясын қолдану барысын қарастырамыз. Келесі код векторды өсу ретімен сұрыптайды:

```
vector<int> v = {4,2,5,3,5,8,3};
sort(v.begin(),v.end());
```

Сұрыптаудан кейінгі вектор `[2,3,3,4,5,5,8]`. Әдепкі қалпы бойынша жұмыс істесе, өсу реті бойынша жүзеге асады, бірақ кері ретпен сұрыптауды төмендегіше орындай аламыз:

```
sort(v.rbegin(),v.rend());
```

Кәдімгі жиым осылай реттеледі:

```
int n = 7; // array size
int a[] = {4,2,5,3,5,8,3};
sort(a,a+n);
```

Ал s жолын сұрыптау төмендегідей орындалады:

```
string s = "monkey";
sort(s.begin(), s.end());
```

Жолды сұрыптау дегеніміз оның құрамындағы элементтердің реттелуін білдіреді. Мысалы, "monkey" жолы "ekmnoy" болып өзгереді.

Салыстыру операторлары

sort функциясы элементтерді сұрыптау кезінде дерек типіне байланысты салыстыру операторын талап етеді. Оператор екі элементтің орындарын анықтауда қажет.

Көптеген C++ дерек типтерінің кірістірілген салыстыру операторлары болады, сондықтан элементтер автоматты түрде сұрыптала алады. Мысалы, сандар мәндеріне байланысты, ал жолдар әліпбидегі реттілігіне байланысты сұрыпталады.

(pair) жұптары бірінші элементтері (first) бойынша сұрыпталады. Екі жұптың бірінші элементтері бірдей болса, екінші элементтері (second) бойынша реттеледі:

```
vector<pair<int,int>> v;
v.push_back({1,5});
v.push_back({2,3});
v.push_back({1,2});
sort(v.begin(), v.end());
```

Осыдан кейінгі жұптардың реттілігі: (1,2), (1,5) және (2,3).

(tuple) кортеждері ұқсас жолмен алғашқы ретте бірінші элемент бойынша, кейін екінші элемент бойынша, т.с.с. сұрыпталады ¹:

```
vector<tuple<int,int,int>> v;
v.push_back({2,1,4});
v.push_back({1,5,3});
v.push_back({2,1,3});
sort(v.begin(), v.end());
```

Кейін кортеждер реттілігі осылай өзгереді: (1,5,3), (2,1,3) және (2,1,4).

¹Кей ескі компиляторларда кортеж құрау үшін өрнек жақшалар орнына make_tuple функциясы қолданылуы қажет екенін ескерген жөн (мысалы, {2,1,4} орнына make_tuple(2,1,4)).

Пайдаланушы құрылымдары (struct)

Пайдаланушы құрылымдарында (struct) автоматты түрдегі салыстыру операторлары болмайды. Оператор құрылым (struct) ішінде параметрі сол типтегі элемент болатын `operator<` функциясы ретінде анықталуы қажет. Ол элемент параметрден кішірек болса, `true` арқылы, кері жағдайда `false` арқылы қайтару керек.

Мысалы, кезекті struct P x және y координатты нүктелерін қамтиды. Салыстыру операторы бірінші x координаты, ал кейін y координаты бойынша сұрыптайды.

```
struct P {  
    int x, y;  
    bool operator<(const P &p) {  
        if (x != p.x) return x < p.x;  
        else return y < p.y;  
    }  
};
```

Салыстыру функциялары

Салыстырудың сыртқы функциясын да солай анықтап, оның `sort` функциясын кері шақыру функциясы ретінде беруге болады. Мысалы, келесі `comp` салыстыру функциясы жолды бірінші кезекте ұзындығы бойынша, ал егер екі бірдей ұзындықтағы жол кездессе, әліпбидегі реттіліктері бойынша сұрыптайды:

```
bool comp(string a, string b) {  
    if (a.size() != b.size()) return a.size() < b.size();  
    return a < b;  
}
```

Жолдардан тұратын вектор осылай сұрыпталады:

```
sort(v.begin(), v.end(), comp);
```

3.3 Бинарлық ізденіс

Жиымдағы элементті іздеудің негізгі жолы – `for` қайталымы арқылы жиыммен өтіп шығу. Мысалы, келесі код жиымнан `x` элементін іздейді:

```
for (int i = 0; i < n; i++) {  
    if (array[i] == x) {  
        // x found at index i  
    }  
}
```

Тәсілдің уақытша күрделілігі $O(n)$, себебі ең нашар жағдайда жиымның барлық элементін қарап шығуға мәжбүрміз. Элементтер кез келген ретте орналасса, тәсіл біз үшін тиімді саналады. Себебі x -ті қайдан іздестіру керектігі туралы қосымша ақпарат ала алмаймыз.

Егер жиым сұрыпталған болса, жағдай басқаша қабылданады. Бұл жағдайда іздестіруді реттілік көмегімен тез жүзеге асыра аламыз. Бинарлық ізденіс алгоритмі сұрыпталған жиымнан элементті $O(\log n)$ уақытта тиімді таба алады.

Бірінші тәсіл

Алгоритмді жүзеге асырудың қарапайым жолы – сөздіктен сөз іздеуге ұқсас. Ізденісті алдымен жиымның барлық элементтерін қамтитын аралықтан бастаймыз. Кейін бірнеше қадам ішінде аралықтың өлшемін екі есеге азайтып отырамыз.

Әр қадам сайын ізденіс қарастырылып жатқан аралықтың орталық элементін тексереді. Егер орталық элемент іздестірілген элемент болса, ізденісті тоқтатамыз. Басқа жағдайда, ізденіс орталық элементтің мәніне байланысты оң жаққа немесе сол жаққа жалғасады.

Идеяның орындалу жолы:

```
int a = 0, b = n-1;
while (a <= b) {
    int k = (a+b)/2;
    if (array[k] == x) {
        // x found at index k
    }
    if (array[k] > x) b = k-1;
    else a = k+1;
}
```

Бұл жердегі қарастырылып жатқан аралық – $a \dots b$, бастапқы қалпы – $0 \dots n-1$ аралығы. Алгоритм әр қадамда ізденіс аралығын екі есеге кішірейтеді. Демек оның уақытша күрделілігі $O(\log n)$ болады.

Екінші тәсіл

Біз бинарлық ізденісті басқа тәсілмен де жүзеге асыра аламыз. Ол үшін жиым элементтерінен тиімді өтіп шығу керек. Мұндағы идея секіруге және нысанға жақындаған сайын жылдамдықты азайтуға негізделеді.

Ізденіс солдан оңға қарай жүреді, бастапқы секіріс – $n/2$. Әр қадамда секіру ұзындығы екі есеге қысқарады: алғашында $n/4$, кейін $n/8$, $n/16$, т.с.с., ақырғы ұзындық бір болғанға дейін жалғасады. Секірістердің соңында біз нысандағы элементті табамыз немесе оның жиымда жоқ екеніне көз жеткіземіз.

Төменде осы идея бойынша жазылған код берілген:

```
int k = 0;
for (int b = n/2; b >= 1; b /= 2) {
    while (k+b < n && array[k+b] <= x) k += b;
}
if (array[k] == x) {
    // x found at index k
}
```

Ізденіс барысында b айнымалысы секіріс ұзындығына жауап береді. алгоритмнің уақытша күрделілігі – $O(\log n)$. Себебі кодтағы while айнымалысы әр секіру ұзындығы үшін ең көп дегенде екі рет қолданылады.

C++ функциялары

C++ стандартты дерекханасы бинарлық ізденіске негізделген және логарифмдік уақытта жұмыс істейтін келесі функцияларды қамтиды:

- lower_bound жиымдағы мәні кемінде x болатын бірінші элементке нұсқайды.
- upper_bound жиымдағы мәні x -тен үлкен ең бірінші элементке нұсқайды.
- equal_range жоғарыдағы екеуін де қайтарады.

Функция жиым сұрыпталған деп қабылдайды. Егер қажет элемент табылмаса, нұсқағыш жиымның соңғы элементінен кейінгі элементке нұсқайды. Мысалы, төмендегі код жиымда x элементі бар-жоғын анықтайды:

```
auto k = lower_bound(array,array+n,x)-array;
if (k < n && array[k] == x) {
    // x found at index k
}
```

Ал келесі код x жиымда қанша рет кездесетінін есептейді:

```
auto a = lower_bound(array, array+n, x);
auto b = upper_bound(array, array+n, x);
cout << b-a << "\n";
```

equal_range көмегімен кодты ықшамдай аламыз:

```
auto r = equal_range(array, array+n, x);
cout << r.second-r.first << "\n";
```

Ең кіші шешімді табу

Бинарлық ізденістің маңызды қолданыс аясы – функцияның мәні өзгеретін позицияны анықтау. Есепке шешім бола алатын ең төмен k мәнін тапқымыз келеді делік. Бізде x жауап бола алса, true, басқа жағдайда false қайтаратын $ok(x)$ функциясы бар. Оған қоса, $ok(x)$ $x < k$ болғанда, false және $x \geq k$ болғанда, true қайтаратынын білеміз. Жағдаят көрінісі:

x	0	1	...	$k-1$	k	$k+1$...
$ok(x)$	false	false	...	false	true	true	...

k мәнін бинарлық ізденіс арқылы анықтай аламыз:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (!ok(x+b)) x += b;
}
int k = x+1;
```

$ok(x)$ false болатындай x -тің ең үлкен мәнін іздейміз. Осылайша, $k = x + 1$ $ok(k)$ true болатындай ең кіші мәнге тең. Бастапқы секіріс ұзындығы z жеткілікті деңгейде үлкен болуы қажет, мысалы біз алдын ала білетін қандай да бір $ok(z)$ true болатын мән.

Алгоритм ok функциясын $O(\log z)$ рет шақырады, сол себепті уақытша күрделілігі ok функциясына тәуелді. Мысалы, функция $O(n)$ уақытта жұмыс жасаса, жалпы уақытша күрделілігі – $O(n \log z)$.

Ең үлкен мәнді табу

Сонымен қатар бинарлық ізденіс мәні алдымен өсіп, кейін белгілі бір сәтте күрт азаятын функция үшін де пайдаланылады. Біздің тапсырмамыз төмендегідей k мәнін табу

- $x < k$ үшін $f(x) < f(x+1)$, ал
- $x \geq k$ үшін $f(x) > f(x+1)$.

Идеясы – $f(x) < f(x+1)$ болатын ең үлкен x -ті бинарлық ізденіс арқылы табу. Бұл $k = x + 1$ екенін білдіреді. Себебі $f(x+1) > f(x+2)$. Төмендегі код ізденісті жүзеге асырады:

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
    while (f(x+b) < f(x+b+1)) x += b;
}
int k = x+1;
```

Қарапайым бинарлық ізденістен айырмашылығы – функцияның қатар келетін мәндері бірдей болуына рұқсат етілмейді. Бұл жағдайда ізденісті қалай жалғастыру керектігін білу мүмкін емес.

4-тарау. Деректер құрылымдары

Деректер құрылымы – деректі компьютер жадында сақтау жолы. Деректер құрылымын есепке сай қолдана білудің маңызы зор. Себебі олардың әрқайсысының өз артықшылықтары мен кемшіліктері болады. Осыдан келіп, ”таңдалған деректер құрылымы үшін қандай операциялар тиімді?” - деген сұрақ туындайды.

Бұл бөлім C++ стандартты дерекханасының ең маңызды деректер құрылымдарымен таныстырады. Көп уақыт үнемдейтіндіктен стандартты дерекхананы қолдану қашанда жақсы идея саналады. Біз стандартты дерекханада жоқ күрделірек деректер құрылымдарын кейінірек қарастырамыз.

4.1 Динамикалық жиымдар

Динамикалық жиым – бағдарлама барысында өлшемі өзгере алатын жиым. C++-те кең қолданыстағы динамикалық жиым – кәдімгі жиым ретінде де пайдалануға болатын вектор құрылымы.

Төмендегі кодта бос вектор құрылып, оған 3 элемент қосылады:

```
vector<int> v;  
v.push_back(3); // [3]  
v.push_back(2); // [3,2]  
v.push_back(5); // [3,2,5]
```

Содан соң элементтерді кәдімгі жиымдағыдай ала аламыз:

```
cout << v[0] << "\n"; // 3  
cout << v[1] << "\n"; // 2  
cout << v[2] << "\n"; // 5
```

size функциясы вектордағы элементтер санын қайтарады. Төмендегі код вектор бойынша өтіп шығып, барлық элементтерін қамтиды:

```
for (int i = 0; i < v.size(); i++) {  
    cout << v[i] << "\n";  
}
```

Векторды өтіп шығудың қысқаша жолы:

```
for (auto x : v) {
```

```
    cout << x << "\n";
}
```

back функциясы вектордың соңғы элементін қайтарса, pop_back функциясы соңғы элементті өшіреді:

```
vector<int> v;
v.push_back(5);
v.push_back(2);
cout << v.back() << "\n"; // 2
v.pop_back();
cout << v.back() << "\n"; // 5
```

Келесі код арқылы бес элементтен тұратын вектор құрылады:

```
vector<int> v = {2,4,2,5,1};
```

Векторды құрудың тағы бір жолы – элементтер саны мен олардың әрқайсысы үшін бастапқы мәнді анықтау. Мысалы:

```
// size 10, initial value 0
vector<int> v(10);
```

```
// size 10, initial value 5
vector<int> v(10, 5);
```

Вектор орындалу барысында қарапайым жиымды қолданады. Егер вектордың өлшемі өсіп, жиым тым кішкентай болып қалса, жаңа жиым ашылып, барлық элементтер соған көшіріледі. Бұл процесс жиі болмайтындықтан, push_back орташа уақытша күрделілігі $O(1)$ -ге тең болады.

Жол (string) құрылымы да вектор сияқты қолданыла алатын динамикалық жиымға жатады. Сонымен қатар бұл жерде басқа деректер құрылымдарында жоқ арнайы синтаксис бар. Жолдар + символы арқылы біріктіріледі. substr(k, x) функциясы k позициясынан басталып, ұзындығы x болатын ішжолды қайтарады, ал find(t) функциясы алғаш кездескен t ішжолының позициясын қайтарады.

Келесі кодта кей жол операциялары көрсетілген:

```
string a = "hatti";
string b = a+a;
cout << b << "\n"; // hattihatti
b[5] = 'v';
cout << b << "\n"; // hattivatti
string c = b.substr(3,4);
cout << c << "\n"; // tiva
```

4.2 Жиын құрылымдары

Жиын (set) – элементтерді жинақтайтын деректер құрылымы. Ал жиындарда орындалатын негізгі операциялар – енгізу, іздеу мен өшіру.

C++ стандартты дерекханасында жиынның екі орындалу жолы бар. Олар: set – теңгерімделген бинарлы дараққа негізделген құрылым. Оның операциялары $O(\log n)$ уақытта жұмыс жасайды. Ал екіншісі – unordered_set, мұнда хэш қолданылады және оның операциялары орта есеппен $O(1)$ уақытта орындалады.

Қай орындалу жолын таңдау көбіне талғамға байланысты болып келеді. set құрылымының артықшылығы – элементтерді реттілігі бойынша сақтап, unordered_set-те жоқ функцияларды қамтуында. Дегенмен, unordered_set те кей жағдайларда әлдеқайда тиімді болуы мүмкін.

Келесі код бүтін сандарды сақтайтын жиын құрады және кейбір операцияларды орындайды. insert функциясы жиынға элементті қосады, count функциясы элементтің жиында қанша рет кезескенін анықтайды, ал erase функциясы элементті жиыннан өшіреді.

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << "\n"; // 1
cout << s.count(4) << "\n"; // 0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n"; // 0
cout << s.count(4) << "\n"; // 1
```

Жиын көбінесе вектор сияқты қолданыла алады, бірақ, элементтерге [] арқылы қол жеткізу мүмкін емес. Төмендегі код жиын құрайды, оның ішіндегі элементтер санын шығарады, кейін барлық элементтерді өтіп шығады:

```
set<int> s = {2,5,6,8};
cout << s.size() << "\n"; // 4
for (auto x : s) {
    cout << x << "\n";
}
```

Ішіндегі элементтердің барлығының өзгеше болуы жиынның маңызды қасиетіне жатады. Сол себепті count функциясы әрдайым 0 (элемент жиында жоқ) немесе 1 (элемент жиымда бар) мәндерін қайтарады және insert элемент жиында бар болған жағдайда оны ешқашан қайта қоспайды. Бұл келесі кодта көрсетілген:

```
set<int> s;
s.insert(5);
s.insert(5);
```

```
s.insert(5);  
cout << s.count(5) << "\n"; // 1
```

Сонымен қатар C++-те `multiset` және `unordered_multiset` құрылымдары бар. Олар `set` пен `unordered_set` сияқты жұмыс істейді, бірақ элементтің бірнеше данасын қамти алады. Мысалдағы кодта 5 саны `multiset`-ке қосылғандай үш рет кездеседі:

```
multiset<int> s;  
s.insert(5);  
s.insert(5);  
s.insert(5);  
cout << s.count(5) << "\n"; // 3
```

`erase` функциясы элементтің барлық кездескен сәттерін `multiset`-тен өшіреді:

```
s.erase(5);  
cout << s.count(5) << "\n"; // 0
```

Тек бір данасын ғана өшіру қажет болатын жағдай жиі кездеседі. Оны келесі түрде орындай аламыз:

```
s.erase(s.find(5));  
cout << s.count(5) << "\n"; // 2
```

4.3 Сөздік құрылымы

Сөздік (`map`) – кілт-мәнді жұптардан тұратын жиым. Қарапайым n өлшемді жиым үшін кілттер әрдайым жүйелі $0, 1, \dots, n-1$ аралығындағы бүтін сандар болса, сөздікте кілт ретінде кез келген дерек типін ала аламыз. Бұл ретте олардың жүйелелігі маңызды емес.

C++ стандартты дерекханасында жиынға ұқсас екі сөздік көрсетілімі бар. Олар: `map` – теңгерімделген бинарлы дараққа негізделген құрылым, элементтерге қол жеткізу $O(\log n)$ уақыт алады, ал екіншісі – `unordered_map`, ол хэшті қолданады және элементтерге қол жеткізу орташа есеппен $O(1)$ уақыт алады.

Келесі код кілті – жолдар, ал мәндері – бүтін сандар болатын сөздік құрады:

```
map<string,int> m;  
m["monkey"] = 4;  
m["banana"] = 3;  
m["harpsichord"] = 9;  
cout << m["banana"] << "\n"; // 3
```

Жолданған сауалда кілт болмаса, кілт автоматты түрде құрылып, әдепкі қалпы бойынша мәнге ие болады. Мысалы, келесі кодта `"aybabtu"` кілті 0

мәнімен қосылды.

```
map<string,int> m;  
cout << m["aybabbu"] << "\n"; // 0
```

count функциясы кілттің бар-жоғын тексереді:

```
if (m.count("aybabbu")) {  
    // key exists  
}
```

Төмендегі код сөздіктегі барлық кілттерді мәндерімен бірге шығарады:

```
for (auto x : m) {  
    cout << x.first << " " << x.second << "\n";  
}
```

4.4 Итераторлар мен аралықтар

C++ стандартты дерекханасындағы көптеген функциялар итераторлар көмегімен жұмыс жасайды. Итератор – деректер құрылымындағы элементке нұсқаушы.

Қолданыста ең жиі кездесетін begin мен end итераторлары деректер құрылымының барлық элементтерін қамтитын аралыққа нұсқайды. begin итераторы деректер құрылымының ең алғашқы элементіне нұсқаса, end итераторы ең соңғы элементтен кейінгі позицияға нұсқайды. Бұл ретте жағдай төмендегідей болмақ:

```
    { 3, 4, 6, 8, 12, 13, 14, 17 }  
      ↑                               ↑  
    s.begin()                       s.end()
```

Итераторлардың асимметриясына назар аударыңыз: s.begin() деректер құрылымының ішін меңзесе, s.end() оның сыртына нұсқайды. Сондықтан итераторлардың анықтаған аралығы – жартылай ашық.

Аралықтармен жұмыс

Итераторлар деректер құрылымындағы элементтер аралығын қажет ететін C++ стандартты дерекханасының функцияларында қолданылады. Әдетте біз деректер құрылымындағы барлық элементтерді өңдегіміз келеді. Сол себепті функцияда begin мен end итераторлары беріледі.

Мысалы, келесі кодта вектор sort функциясы арқылы сұрыпталады, кейін reverse функциясымен элементтер кері айналады, соңында random_shuffle функциясының көмегімен реттілікті араластырады.

```
sort(v.begin(), v.end());
reverse(v.begin(), v.end());
random_shuffle(v.begin(), v.end());
```

Бұл функциялар қарапайым жиыммен де қолданыла алады. Мына үлгідегі функцияларда итераторлар орнына жиымға нұсқаймыз:

```
sort(a, a+n);
reverse(a, a+n);
random_shuffle(a, a+n);
```

Жиын итераторлары

Итераторлар жиын элементтеріне жету үшін жиі қолданылады. Келесі код жиындағы ең кіші элементке нұсқайтын `it` итераторын құрайды:

```
set<int>::iterator it = s.begin();
```

Қысқартылған түрі:

```
auto it = s.begin();
```

Итератор нұсқаған элементке `*` символын пайдалану арқылы қол жеткізе аламыз. Мысалы, төмендегі код жиындағы алғашқы элементті шығарады:

```
auto it = s.begin();
cout << *it << "\n";
```

Итераторды келесі операторлар арқылы жылжыта аламыз: `++` (алдыға), `--` (артқа), яғни жиындағы дейінгі немесе келесі элементке жылжыту деген мағынада қолданылады.

Келесі код элементтерді өсу ретімен шығарады:

```
for (auto it = s.begin(); it != s.end(); it++) {
    cout << *it << "\n";
}
```

Ал мына код ең үлкен элементті шығарады:

```
auto it = s.end(); it--;
cout << *it << "\n";
```

`find(x)` функциясы мәні `x` болатын элементке нұсқайтын итераторды қайтарады. Дегенмен жиында `x` болмаған жағдайда итератор соңына нұсқайды.

```
auto it = s.find(x);
if (it == s.end()) {
    // x is not found
}
```

```
| }
```

`lower_bound(x)` функциясы жиындағы мәні ең кемінде x болатын кіші элементке нұсқаушы итератор қайтарса, `upper_bound(x)` мәні x -тен үлкенірек элементке нұсқайтын итератор қайтарады. Екі функцияда да, егер сондай элемент болмаса, қайтарылған мән `end` болмақ. Функциялар элементтер реттілігіне баса назар аудармайтын `unordered_set`-те қолданылмайды.

Мысалы, келесі код x -ке ең жақын элементті анықтайды:

```
auto it = s.lower_bound(x);
if (it == s.begin()) {
    cout << *it << "\n";
} else if (it == s.end()) {
    it--;
    cout << *it << "\n";
} else {
    int a = *it; it--;
    int b = *it;
    if (x-b < a-x) cout << b << "\n";
    else cout << a << "\n";
}
```

Код жиынның бос емес екенін жобалап, `it` итераторын қолданудың барлық мүмкіндіктерін тексеріп шығады. Алғашында итератор ең кіші мәні кем дегенде x болатын элементті көрсетеді. Егер `it` `begin`-ге тең болса, ол элемент жиындағы x -ке ең жақыны болмақ. Ал егер `it` `end`-ке тең болса, жиындағы ең үлкен сан x -ке ең жақыны болмақ. Екеуіне де сәйкес келмесе, x -ке ең жақын элемент не `it`, не оның алдыңғы жанындағы элемент болмақ.

4.5 Басқа құрылымдар

Битсет

Битсет (`bitset`) – әр мәні 0 немесе 1 болатын жиым. Мысалы, келесі код 10 элементтен тұратын битсет құрады:

```
bitset<10> s;
s[1] = 1;
s[3] = 1;
s[4] = 1;
s[7] = 1;
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

Битсет қолданудың артықшылығы – олардың қарапайым жиынмен салыстырғанда жадыны азырақ талап етуінде. Себебі әр элемент жадыда 1 бит қана алады. Мысалы, егер n бит `int` жиымында сақталса, жадыда $32n$ бит қолданылады, ал битсет тек n бит жадыны қажет етеді. Оған қоса, битсет

мәндерін бит операцияларының арқасында тиімді қолдана аламыз. Бұл өз кезегінде биттік жиындарда орындалатын алгоритмдерді оңтайландырады.

Төмендегі код битсет құраудың тағы бір жолын көрсетеді:

```
bitset<10> s(string("0010011010")); // from right to left
cout << s[4] << "\n"; // 1
cout << s[5] << "\n"; // 0
```

count функциясы битсеттегі бірліктер санын қайтарады:

```
bitset<10> s(string("0010011010"));
cout << s.count() << "\n"; // 4
```

Келесі кодта бит операцияларының қолданыс үлгілері көрсетілген:

```
bitset<10> a(string("0010110110"));
bitset<10> b(string("1011011000"));
cout << (a&b) << "\n"; // 0010010000
cout << (a|b) << "\n"; // 1011111110
cout << (a^b) << "\n"; // 1001101110
```

Екі жақты тізбек

Екі жақты тізбек (deque) – өлшемі екі жағынан да тиімді өзгертін динамикалық жиым. Вектор сияқты екі жақты тізбекте де push_back және pop_back функциялары орындалады. Сонымен қатар ол векторда жоқ push_front және pop_front функцияларын да қамтиды.

Үлгіде екі жақты тізбек қолданысы берілген:

```
deque<int> d;
d.push_back(5); // [5]
d.push_back(2); // [5,2]
d.push_front(3); // [3,5,2]
d.pop_back(); // [3,5]
d.pop_front(); // [5]
```

Ішкі құрылысы векторға қарағанда анағұрлым күрделірек, сол себепті екі жақты тізбек вектордан баяуырақ болмақ. Дегенмен екі ұшынан қосу және алу операциялары $O(1)$ уақыт алады.

Стек

Стек (stack) – екі $O(1)$ уақыт алатын операцияларды, яғни элементті жоғарыға қосу және жоғарыдан алу операцияларын орындауға болатын құрылым. Бұл жерде тек жоғарғы элемент қолжетімді болмақ.

Үлгідегі кодта стекті қолдану жолдары көрсетілген:

```
stack<int> s;
```

```
s.push(3);
s.push(2);
s.push(5);
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

Кезек

Кезек (queue) құрылымы да екі $O(1)$ уақыт алатын операцияларды, атап айтқанда элементті кезек соңына қосу және кезектегі алғашқы элементті алу операцияларын қамтиды. Мұнда кезектің соңғы және алғашқы элементтері ғана қолжетімді.

Үлгідегі кодта ол операцияларды қолдану жолдары көрсетілген:

```
queue<int> q;
q.push(3);
q.push(2);
q.push(5);
cout << q.front(); // 3
q.pop();
cout << q.front(); // 2
```

Басымдылық кезегі

Басымдылық кезегі (priority queue) элементтер жиынын жалғастырады және ол енгізу, кезек типіне байланысты ең көп немесе ең аз элементті шығару және жою сияқты операциялардан тұрады. Енгізу мен жою $O(\log n)$ уақыт алса, шығару $O(1)$ уақыт алады.

Реттелген жиын (set) басымдылық кезегінің (priority queue) барлық операцияларын тиімді өңдегенімен, басымдылық кезегін (priority queue) қолданудың айтарлықтай артықшылығы бар. Ол – тұрақты факторлардың аздығы. Басымдылық кезегі (priority queue) әдетте үйінді құрылымы арқылы орындалады және ол реттелген жиынды құрайтын теңгерімделген бинарлы дарақтан жеңілірек.

Әдепкі қалпы бойынша, C++ басымдылық кезегінде элементтер кему ретінде сұрыпталады және кезектен ең жоғары элементті тауып, оны жою мүмкіндігі бар. Бұл келесі кодта келтірілген:

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
```

```
cout << q.top() << "\n"; // 5  
q.pop();  
q.push(6);  
cout << q.top() << "\n"; // 6  
q.pop();
```

Егер ең кіші элементтерді тауып, өшіретін басымдылық кезегін құрастырғымыз келсе, оны төмендегідей ретпен орындай аламыз:

```
priority_queue<int,vector<int>,greater<int>>> q;
```

Саясатқа негізделген деректер құрылымдары

g++ компиляторы C++ стандартты дерекханасында жоқ, кей өзге деректер құрылымдарын да қолдайды. Мұндай құрылымдар саясатқа негізделген деректер құрылымдары деп аталады. Оларды пайдалану үшін кодқа төмендегі жолдарды қосуымыз қажет.

```
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
```

Осыдан кейін set-ке ұқсас, бірақ жиым сияқты индекстелетін indexed_set деректер құрылымы қолжетімді болады. Мұндағы int мәндері келесі түрде анықталады:

```
typedef tree<int,null_type,less<int>,rb_tree_tag,
            tree_order_statistics_node_update> indexed_set;
```

Ал енді жиынды төмендегідей етіп құрай аламыз:

```
indexed_set s;
s.insert(2);
s.insert(3);
s.insert(7);
s.insert(9);
```

Жиынның ерекшелігі – сұрыпталған жиымдағы элементтерге индекстер арқылы қол жеткізу мүмкіндігінде. find_by_order функциясы берілген позициядағы элементке итератор қайтарады:

```
auto x = s.find_by_order(2);
cout << *x << "\\n"; // 7
```

Ал order_of_key функциясы берілген элементтің позициясын қайтарады:

```
cout << s.order_of_key(7) << "\\n"; // 2
```

Егер элемент жиында болмаса, жиында болған жағдайдағы позициясын жобамен аламыз:

```
cout << s.order_of_key(6) << "\\n"; // 2
cout << s.order_of_key(8) << "\\n"; // 3
```

Екі функция да логарифмдік уақытта орындалады.

4.6 Сұрыптаумен салыстыру

Көп жағдайда есепті не деректер құрылымын, не сұрыптауды пайдалану арқылы шығара аламыз. Кейде бұл екі әдістің тиімділіктерінде үлкен айырмашылықтар байқалады және ол айырмашылықтар уақытша күрделілігіне байланысты туындауы мүмкін.

Мысал үшін n элементтен тұратын A және B тізімдері берілген есепті қарастырайық. Бізге екі тізімде де кездесетін элементтер санын анықтау тапсырылады. Мысалы,

$$A = [5, 2, 8, 9] \quad \text{және} \quad B = [3, 2, 9, 5],$$

тізімдері үшін жауап 3-ке тең. Себебі 2, 5 және 9 сандары екі тізімде де кездеседі.

Қарапайым шешімі: $O(n^2)$ уақытта барлық жұптарды тексеріп шығу. Бірақ біз келесі кезекте тиімдірек болатын алгоритмдерді қарастырамыз.

1-алгоритм

A тізіміндегі элементтерден тұратын жиын құраймыз, кейін B элементтерімен өтіп шығып, әр элементтің A тізімінде кездесуін тексереміз. Бұл тәсілдің тиімді болу себебі мынада: A элементтері жиында сақталған. set деректер құрылымын қолдану нәтижесінде уақытша күрделілігі $O(n \log n)$ болмақ.

2-алгоритм

Реттелген жиын қолдану міндетті емес болғандықтан set құрылымын unordered_set құрылымымен алмастыра аламыз. Бұл – алгоритмді тиімдірек етудің жеңіл жолы. Себебі бізге негізгі деректер құрылымын өзгерту ғана жеткілікті. Жаңа алгоритмнің уақытша күрделілігі – $O(n)$.

3-алгоритм

Деректер құрылымдарының орнына сұрыптауды қолдана аламыз. Біріншіден, A және B тізімдерін сұрыптаймыз. Кейін, екі тізімді де бір уақытта өтіп шығу арқылы бірдей элементтерді анықтаймыз. Сұрыптаудың уақытша күрделілігі $O(n \log n)$ -ге тең және алгоритмнің қалған бөлігі $O(n)$ уақыт алатындықтан қорытынды уақытша күрделілігі $O(n \log n)$ болмақ.

Тиімділікті салыстыру

Келесі кестеде жоғарыдағы алгоритмдердің қаншалықты тиімді екендігі n өзгеріп отыратын және тізім элементтері $1 \dots 10^9$ аралығындағы кез келген бүтін сандар болатын жағдайлар арқылы көрсетілген:

n	1-алгоритм	2-алгоритм	3-алгоритм
10^6	1.5 s	0.3 s	0.2 s
$2 \cdot 10^6$	3.7 s	0.8 s	0.3 s
$3 \cdot 10^6$	5.7 s	1.3 s	0.5 s
$4 \cdot 10^6$	7.7 s	1.7 s	0.7 s
$5 \cdot 10^6$	10.0 s	2.3 s	0.9 s

1 және 2-алгоритмдердің айырмашылығы жиын құрылымдарында ғана байқалады. Бұл есепте жасалған таңдаудың өңдеу уақыты үшін әсері орасан зор. Себебі 2-алгоритм 1-алгоритмге қарағанда 4-5 есе жылдамырақ.

Дегенмен ең тиімді алгоритм – сұрыптау қолданатын 3-алгоритм. Ол 2-алгоритммен салыстырғанда 2 есе аз уақыт жұмсайды. Қызығы, 1-алгоритм мен 3-алгоритмнің екеуінің де уақытша күрделілігі – $O(n \log n)$, бірақ бұған қарамастан, 3-алгоритм он есе жылдамырақ. Мұны келесі фактілер арқылы түсіндіре аламыз: сұрыптау – қарапайым процедура және ол 3-алгоритмнің басында 1 мәрте орындалады, ал алгоритмнің қалған бөлігі сызықты уақытта жұмыс жасайды. Басқа қырынан қарасақ, 1-алгоритм жұмыс барысында күрделі теңгерімделген бинарлы даракты сақтайды.

5-тарау. Толық ізденіс

Толық ізденіс – кез келген алгоритмдік есепті шығарудың жалпылама түрі. Негізгі идеясы толықтай іріктеуді қолдана отырып, барлық мүмкін жауаптарды қарастыруға, кейін есеп шартына байланысты ең үздік нәтижелерді іріктеу немесе шешімдер санын есептеуге бағытталады.

Толық ізденісті барлық мүмкін шешімдерді қарастыруға уақыт жеткілікті болғанда ғана тиімді әдіс деп есептеуге болады, себебі мұндай ізденіс әдетте оңай орындалады және әрдайым дұрыс жауапты көрсетеді. Егер толық ізденіс тым баяу болса, ашкөз алгоритмдер немесе динамикалық бағдарламалау сияқты өзге әдіс-тәсілдер қажет болуы мүмкін.

5.1 Ішжиындар құру

Бірінші қарастыратын есебіміз n элементтен тұратын жиынның барлық ішжиындарын құруға арналады. Мысалы, $\{0, 1, 2\}$ ішжиындары: \emptyset , $\{0\}$, $\{1\}$, $\{2\}$, $\{0, 1\}$, $\{0, 2\}$, $\{1, 2\}$ және $\{0, 1, 2\}$. Ішжиындарды құрудың негізгі екі әдісі бар. Олар: не рекурсивті ізденіс жүргізу, не бүтін сандардың биттік көрсетілімін қолдану.

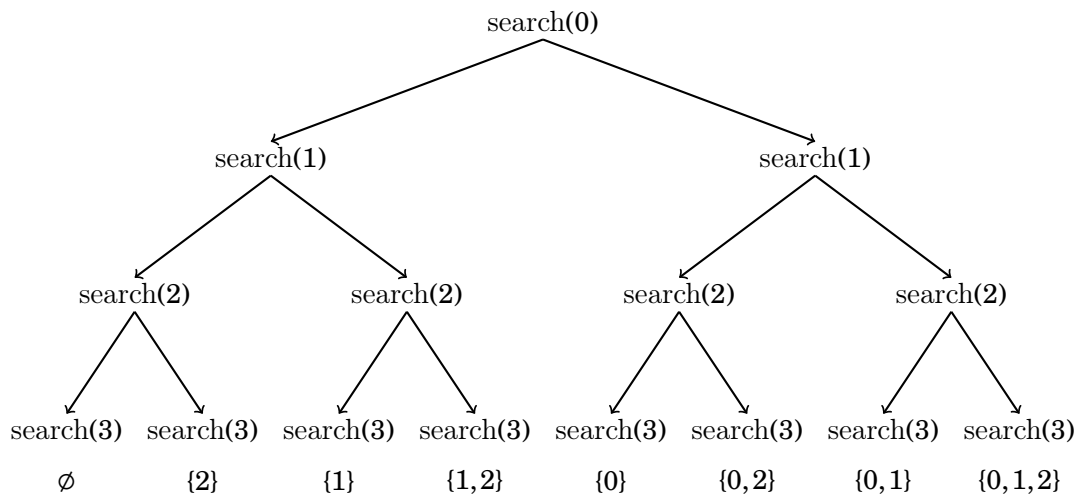
1-әдіс

Жиындағы барлық ішжиындарды өтіп шығудың ыңғайлы жолы – рекурсия. Келесі `search` функциясы $\{0, 1, \dots, n - 1\}$ жиынының ішжиындарын құрастырады. Функция әр ішжиынның элементтерін сақтайтын `subset` векторын қолдайды. Ізденіс функциясы 0 параметрімен шақырылғаннан кейін басталады.

```
void search(int k) {
    if (k == n) {
        // process subset
    } else {
        search(k+1);
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
    }
}
```

search функциясы k параметрімен шақырылғанда ішжиынға k элементін қосу, не қоспау туралы шешім қабылдайды, екі жағдайда да өзін $k + 1$ параметрімен қайта шақырады. Дегенмен, егер $k = n$ орындалса, функция барлық элементтер қаралғанын және ішжиын құрылғанын байқайды.

Төмендегі дарак $n = 3$ болған жағдайдағы шақыруларды көрсетеді. Біз әрдайым не сол тармақты (k ішжиымға кірмейді), не оң тармақты (k ішжиымға кіреді) таңдай аламыз.



2-әдіс

Ішжиындарды құраудың бүтін сандардың биттік көрсетіліміне негізделген жолы да бар. n элементтен тұратын жиынның әр ішжиыны n биттен тұратын тізбек ретінде көрсетіле алады, ол өз кезегінде $0 \dots 2^n - 1$ аралығындағы бүтін сан болады. Бит тізбегіндегі бірліктер қай элементтің ішжиынға кіретіндігін білдіреді.

Әдетте ең соңғы бит 0-элементке, ал соңғының алдындағы бит 1-элементке сәйкестендіріледі және солай жалғаса береді. Мысалы, 25 санының биттік көрсетілімі – 11001, осылайша біздің ішжиымымыз $\{0,3,4\}$ болмақшы.

Келесі код n элементтен тұратын жиынның ішжиындарымен өтіп шығады:

```

for (int b = 0; b < (1<<n); b++) {
    // process subset
}
  
```

Келесі код бит тізбегіне сәйкес элементтерден тұратын ішжиынды қалай табуға болатындағын көрсетеді. Әр ішжиынды өңдеу барысында код ішжиын элементтерін сақтайтын вектор құрайды.

```

for (int b = 0; b < (1<<n); b++) {
    vector<int> subset;
    for (int i = 0; i < n; i++) {
        if (b&(1<<i)) subset.push_back(i);
    }
}
  
```

```
|}
```

5.2 Алмастырулар құрау

Біз қарастыратын келесі мәселе – n элементтен тұратын жиынның барлық алмастыруларын құрау. Мысалы, $\{0, 1, 2\}$ алмастырулары – $(0, 1, 2)$, $(0, 2, 1)$, $(1, 0, 2)$, $(1, 2, 0)$, $(2, 0, 1)$ мен $(2, 1, 0)$ түрінде жалғасады. Мұндай алмастырулар да екі түрлі амалмен жасалады. Олар: рекурсияны қолдану немесе алмастыруларды итеративті өтіп шығу.

1-әдіс

Ішжиындар сияқты алмастырулар да рекурсивті құрала алады. Келесі search функциясы $\{0, 1, \dots, n-1\}$ жиынының алмастыруларымен өтіп шығады. Функция алмастыруды сақтайтын permutation векторын құрады, ізденіс параметрлерсіз шақырылған функция арқылы басталады.

```
void search() {
    if (permutation.size() == n) {
        // process permutation
    } else {
        for (int i = 0; i < n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permutation.push_back(i);
            search();
            chosen[i] = false;
            permutation.pop_back();
        }
    }
}
```

Әр шақырту permutation-ге жаңа элемент қосады. chosen жиымы қай элементтердің араластыруға әлдеқашан қосылғандығы туралы хабар береді. Егер permutation өлшемі жиын өлшемімен теңессе, алмастыру құралғандығын білдіреді.

2-әдіс

Алмастыруларды құраудың тағы бір жолына – $\{0, 1, \dots, n-1\}$ алмастыруымен басталып, келесі алмастыруды өсу реттілігінде құрастыратын функцияны бірнеше рет шақыру жатады. Бұл үшін C++ стандартты дерекханасындағы next_permutation функциясын қолдануға болады:

```
vector<int> permutation;
for (int i = 0; i < n; i++) {
```

```

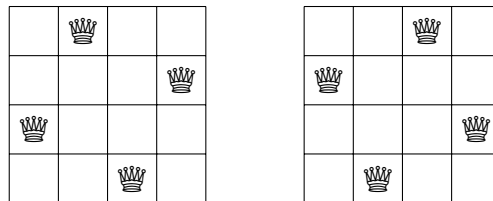
    permutation.push_back(i);
}
do {
    // process permutation
} while (next_permutation(permutation.begin(), permutation.end()));

```

5.3 Қайта іздеу алгоритмі

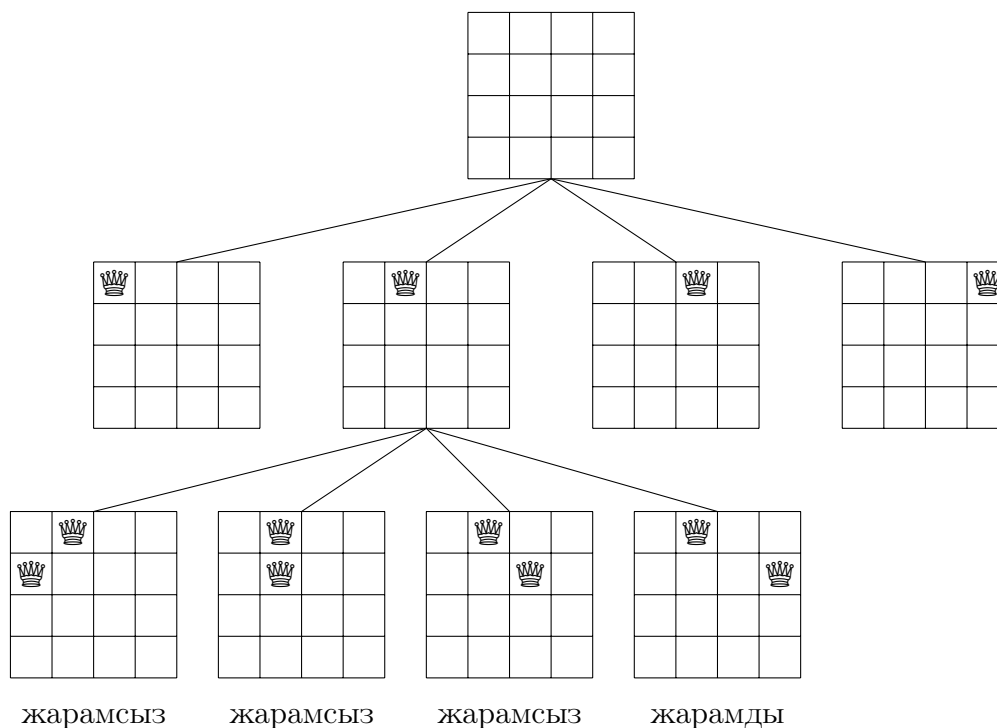
Қайта іздеу (backtracking) алгоритмі бос шешіммен басталып, бірте-бірте шешімді кеңейтеді. Ізденіс рекурсивті түрде шешімнің өзгеруі мүмкін барлық жолдарын қарастырады.

Үлгі ретінде n уәзірдің $n \times n$ шахмат тақтасында бір-біріне шабуыл жасамайтындай етіп орналастыру жолдарын табу туралы есепті қарастырайық. Мысалы, $n = 4$ болған жағдайда екі шешім ұсынылады:



Есепті қайта іздеу алгоритмінің көмегімен уәзірді тақтада қатар-қатар орналастыру арқылы шеше аламыз. Нақтырақ айтсақ, әр қатарға оған дейін тұрған басқа уәзірлерге шабуыл жасамайтындай етіп бір ғана уәзір орналастырылады. Шешім барлық n уәзір орналастырылғаннан кейін табылады.

Мысалы, төменде қайта іздеу алгоритмін қолдану арқылы $n = 4$ болған жағдайда табылған кей шешімдер көрсетілген:



Ең төменгі деңгейдегі алғашқы үш конфигурация жарамсыз, өйткені мұндай жағдайда уәзірлер бір-біріне шабуыл жасайды. Дегенмен төртінші конфигурация жарамды және тақтада тағы екі уәзірді орналастыру арқылы толық шешімге кеңейтіле алады. Қалған екі уәзірді орналастырудың бір ғана жолы бар.

Алгоритмнің жүзеге асырылу жолы:

```
void search(int y) {
    if (y == n) {
        count++;
        return;
    }
    for (int x = 0; x < n; x++) {
        if (column[x] || diag1[x+y] || diag2[x-y+n-1]) continue;
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 1;
        search(y+1);
        column[x] = diag1[x+y] = diag2[x-y+n-1] = 0;
    }
}
```

Ізденіс `search(0)` шақыруымен басталады. Тақтаның өлшемі – $n \times n$, шешімдер саны `count`-та сақталады.

Код тақтадағы бағандар мен қатарлар 0-ден $n - 1$ -ге дейін нөмірленген деп есептейді. `search` функциясы `y` параметрімен шақырылғанда уәзір `y` қатарына орналастырылады, кейін өзін `y + 1` параметрімен шақырады. Егер `y = n` болса, шешім табылғанын білдіреді және `count` айнымалысы бірге артады.

`column` жиымы уәзірі бар бағандарды, ал `diag1` және `diag2` жиымдары уәзірі бар диагональдарды сақтайды. Уәзірі бұрыннан бар диагональға немесе

бағанға тағы бір уәзір қоса алмаймыз. Мысалы, 4×4 өлшемді тақтаның диагональдары мен бағандары осылай нөмірленген:

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

column

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

diag1

3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

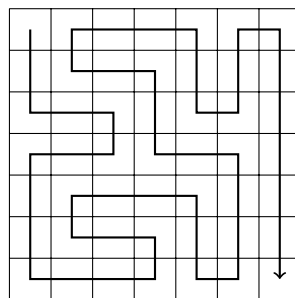
diag2

$q(n)$ n уәзірді $n \times n$ шахмат тақтасында орналастыру жолдарының саны деп белгілейік. Жоғарыдағы қайта іздеу алгоритміне сай келесі мысалды келтіре аламыз. $q(8) = 92$. n ұлғайған сайын ізденіс біртіндеп баяулайды, себебі шешімдер саны қарқынды өсіп отырады. Мысалы, $q(16) = 14772512$ мұны жоғарыда көрсетілген алгоритм көмегімен есептеу үшін заманауи компьютер бір минут жұмсайды ¹.

5.4 Ізденісті ықшамдау

Қайта іздеу алгоритмін ізденіс дарағын қысқарту арқылы жиі оңтайландыруға болады. Мұндағы басты идея – алгоритмге іздеу барысында түпкі нәтижеге қол жеткізудің мүмкін немесе мүмкін емес екендігін байқап, іздеуді одан ары жалғастыратын немесе кесіп тастай алатын "интеллект" қосу. Мұндай оңтайландырудың ізденісті тиімдірек етуге тигізетін септігі мол.

$n \times n$ тордың жоғарғы сол жақ бұрышынан төменгі оң жақ бұрышына дейінгі жолда әр шаршыдан бір рет қана жүріп өтетін қанша жол бар екенін іздейтін есепті қарастырайық. Мысалы, 7×7 торда осындай 111712 жол бар. Олардың бірі:



Күрделілік деңгейі біз қажет еткен деңгейге сай келгендіктен, біз 7×7 жағдайына тоқталамыз. Алдымен қарапайым қайта іздеу алгоритмінен бастаймыз, кейін ізденіс тоқтатылуы мүмкін жағдайларды байқай отырып, бірте-бірте оңтайландырамыз. Әр оңтайландырудан кейін орындалу уақытын өлшейміз және рекурсивті шақыруларды есептейміз. Осылайша әр оңтайландырудың ізденіс тиімділігіне тигізген септігін анық байқаймыз.

¹ $q(n)$ -нің үлкенірек мәндерін есептеуге мүмкіндік беретін тиімдірек алгоритм әлі табылған жоқ. Қазіргі үздік көрсеткіш – 2016 жылы есептелген $q(27) = 234907967154122528$ [15].

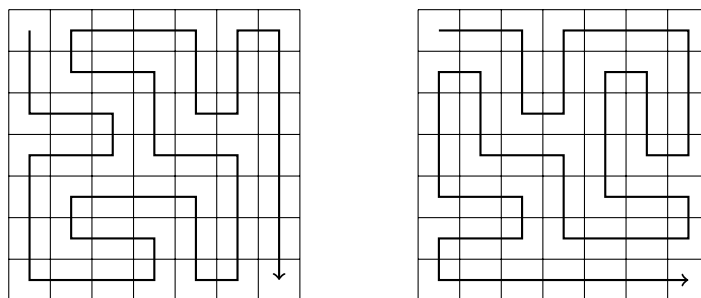
Негізгі алгоритм

Алгоритмнің алғашқы нұсқасы – белгілі бір нәтижеге жету үшін орындалатын әрекет ету тәртібін ешқандай оңтайландырусыз сипаттайтын нұсқаулар жиынтығы. Біз жай ғана қайта іздеу арқылы жоғарғы сол жақ бұрыштан төменгі оң жақ бұрышқа дейінгі барлық мүмкін жолдарды құрастырып, олардың санын есептейміз.

- іске асыру уақыты: 483 секунд
- рекурсия шақыруларының саны: 76 миллиард

1-оңтайландыру

Кез келген шешімде ең бірінші қадамды төменге, не оңға жасаймыз. Бірінші жүрістен кейін әрдайым тор диагоналы бойынша симметриялы екі жол шығады. Мысалы, келесі жолдар симметриялы:

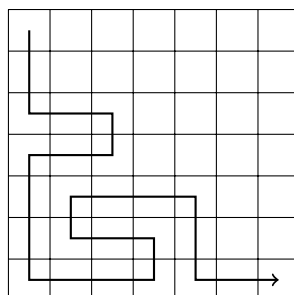


Осылайша әрдайым алдымен төмен (не оңға) бір қадам қозғалып, шешімдер санын екіге көбейтеміз.

- іске асыру уақыты: 244 секунд
- рекурсия шақыруларының саны: 38 миллиард

2-оңтайландыру

Егер тордағы барлық шаршыларды өтпестен оң жақ төменгі бұрышқа жетсек, бұл шешімнің толық болмайтындығы анық. Мұндай жол үлгісі төмендегідей болады:

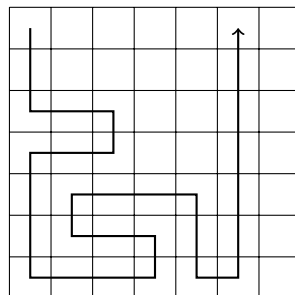


Осындай бақылауларды қолдана отырып, оң жақ төменгі бұрышқа тым ерте жеткен жағдайда ізденісті бірден тоқтатамыз.

- іске асыру уақыты: 119 секунд
- рекурсия шақыруларының саны: 20 миллиард

3-оңтайландыру

Жол шекараға жетіп, не солға, не оңға бағыт ала алса, тор әлі болмаған шаршыларды қамтитын екі бөлікке бөлінеді. Мысалы, келесі жағдайда жол оңға, не солға бағыт ала алады:

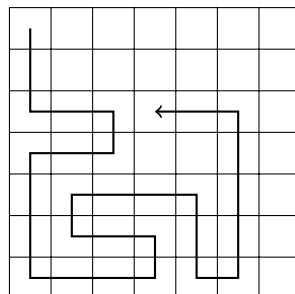


Бұл жағдайда біз барлық шаршыларға бара алмаймыз, сол себепті ізденісті ықшамдап аламыз. Бұл оңтайландыру аса пайдалы:

- іске асыру уақыты: 1.8 секунд
- рекурсия шақыруларының саны: 221 миллион

4-оңтайландыру

3-оңтайландырудың идеясын былайша жалпылай аламыз: егер жол жалғаса алмаса, бірақ сол не оңға бағыт ала алса, тор әлі болмаған шаршыларды қамтитын екі бөлікке бөлінеді. Мысалы, төмендегідей жол:



Енді барлық шаршыларға бара алмайтынымыз анық, сондықтан ізденісті тоқтатамыз. Мұндай оңтайландырудан кейін ізденіс өте тиімді болмақ:

- іске асыру уақыты: 0.6 секунд

- рекурсия шақыруларының саны: 69 миллион

Осы жерден алгоритмді оңтайландыруды тоқтатып, қандай нәтижеге қол жеткізгенімізді көрейік. Бастапқы алгоритмнің іске асыру уақыты 483 секунд болса, оңтайландырулардан кейінгі іске асыру уақыты – небәрі 0.6 секунд. Осылайша оңтайландырудың әсерінен алгоритмнің жылдамдығы шамамен 1000 есеге артады.

Бұл қайта іздеудегі әдепкі құбылыс, себебі ізденіс дарағы әдетте үлкен болып келеді және кішкене бақылаулардың өзі ізденісті тиімді ықшамдайды. Әсіресе алгоритмнің алғашқы қадамдарында, яки ізденіс дарағының басында орын алатын оңтайландырулар өте пайдалы болмақ.

5.5 Ортада кездесу

Ортада кездесу (meet in the middle) – ізденіс аясын екі тең бөлікке бөлу тәсілі. Мұнда екі бөлікке де дербес ізденіс жүргізіліп, соңында ізденіс нәтижелері қосылады.

Бұл тәсілді ізденіс нәтижелерін тиімді қосу мүмкіндігі болған жағдайда қолдана аламыз. Демек екі ізденіс бір үлкен ізденіске қарағанда аз уақытты талап етуі мүмкін. Әдетте ортада кездесу тәсілін қолдана отырып, 2^n факторын $2^{n/2}$ факторына айналдыра аламыз.

Үлгі ретінде n саннан тұратын тізім мен x саны берілген есепті қарастырайық. Тапсырма бойынша тізімдегі қандай да бір сандар қосындысынан x санын алуға болатынын анықтау керек. Мысалы, $[2, 4, 5, 9]$ тізбегі мен $x = 15$ саны берілген, $[2, 4, 9]$ сандарын таңдау арқылы $2 + 4 + 9 = 15$ ала аламыз. Бірақ $x = 10$ болған жағдайда дәл осы тізіммен қосындыға қол жеткізу мүмкін болмас еді.

Қарапайым алгоритм – барлық ішжиындармен өтіп шығып, қосындысы x болатынын тексеру. Мұндай алгоритмнің уақытша күрделілігі – $O(2^n)$, себебі $O(2^n)$ ішжиыны бар. Дегенмен ортада кездесу тәсілін қолдана отырып, тиімдірек $O(2^{n/2})$ уақытына¹ қол жеткізе аламыз. $O(2^n)$ мен $O(2^{n/2})$ әртүрлі уақытша күрделіліктері екендігін ескерген жөн, себебі $2^{n/2} \sqrt{2^n}$ тең.

Мұндағы идея – тізімді A және B тізімдеріне тең элемент санын қамтитындай етіп бөлу. Бірінші ізденіс A -ның барлық ішжиындарын құрайды және олардың қосындыларын S_A тізімінде сақтайды. Екінші ізденіс сәйкесінше S_B тізімін B -дан құрайды. Бұдан кейін S_A -дан бір элемент, S_B -дан бір элемент қана алу арқылы x санын құрау мүмкіндігін тексеру жеткілікті. Бұл бастапқы тізімдегі сандардан x қосындысын құрауға болатын жағдайда ғана мүмкін болмақ.

Мысалы, тізім – $[2, 4, 5, 9]$, ал $x = 15$. Алдымен екі тізімге бөліп аламыз: $A = [2, 4]$ және $B = [5, 9]$. Кейін $S_A = [0, 2, 4, 6]$ мен $S_B = [0, 5, 9, 14]$ тізімдерін құрамыз. Бұл жағдайда $x = 15$ -ті құрау мүмкін, себебі S_A 6 қосындысын қамтиды, ал S_B 9 мәнін қамтиды, сонда $6 + 9 = 15$ болады. Бұл $[2, 4, 9]$ шешіміне сәйкес келеді.

¹Бұл идеяны 1974 жылы Э.Горовиц пен С.Сахни таныстырған болатын[16].

Біз алгоритмді уақытша күрделілігі $O(2^{n/2})$ болатындай жүзеге асыра аламыз. Біріншіден сұрыпталған S_A және S_B тізімдерін құраймыз, ол $O(2^{n/2})$ уақытында бірігу тәсілін қолдану арқылы жүзеге асырылады. Бұдан кейін тізімдер сұрыпталғандықтан S_A мен S_B тізімдерінен x қосындысын құрауға болатындығын $O(2^{n/2})$ уақытында тексере аламыз.

6-тарау. Ашкөз алгоритмдер

Ашкөз алгоритм – есепті әрдайым дәл қазіргі уақытта тиімді болатын жауапты таңдау арқылы шығаратын алгоритм түрі. Ашкөз алгоритм жасаған шешімдерін қайтармайды, қарастырылған шешімдердің ең тиімдісінен жауап құрастырады. Сол себепті де ашкөз алгоритмдерді әдетте өте ұтымды таңдау деп есептейді.

Алайда ашкөз алгоритмді жобалаудың өзіндік қиындығы да бар. Ол – есепке әрдайым оңтайлы жауапты табатын ашкөз стратегияны құру. Локалды оңтайлы шешімдер глобалды оңтайлы болу керек. Ашкөз алгоритмнің дұрыс жұмыс жасайтынын дәлелдеу көбіне қиынға соғады.

6.1 Тиын жайлы есеп

Алдымен мысал арқылы қарастырайық: бізге тиындардың жиыны берілген, қосындысы n болатын тиындарды таңдау керек. Тиындардың мәндері: $\text{coins} = \{c_1, c_2, \dots, c_k\}$ және бір тиынды қалағанымызша алсақ болады. "Ең аз дегенде қанша тиын қажет?" - деген сұраққа жауап іздейміз.

Мысалы, $n = 520$ және тиындар

$$\{1, 2, 5, 10, 20, 50, 100, 200\}$$

болса, бізге кемінде 4 тиын алу керек. Бұл жердегі оңтайлы жауап – $200 + 200 + 100 + 20 (= 520)$ тиындарын алу.

Ашкөз алгоритм

Бұл есепті шығаратын қарапайым ашкөз алгоритм бізге қажет сома жиналғанға дейін әрдайым мүмкін болатын ең үлкен тиынды алып отырады. Алгоритм бұл мысалда қисынды жұмыс істейді, яғни басында біз құны 200-дік екі тиынды, одан кейін құны 100-дік бір тиынды аламыз, ал ең соңында құны 20-лық бір тиын алынады.

Егер тиындарды теңге деп қабылдасақ, онда алгоритміміз дұрыс жұмыс жасайды. Басқаша айтқанда, алгоритм әрдайым ең аз мөлшерде қолданылатын тиындарды алады. Алгоритмнің дұрыстығын былай көрсетуге болады:

Біріншіден, әр 1, 5, 10, 50 және 100 тиындары жауапта ең көп дегенде бір рет кездеседі. Егер жауапта бірдей тиын екі рет кездессе, оларды мәндес бір тиынмен ауыстырып, оңтайлы жауап алуға болады. Мысалы, егер жауапта $5 + 5$ тиындары болса, оларды 10 тиынмен ауыстыруға болады.

Дәл солай 2, 20 тиындары ең көп дегенде 2 рет кездеседі, себебі $2 + 2 + 2$ тиындарын $5 + 1$ тиындарымен, $20 + 20 + 20$ тиындарын $50 + 10$ тиындарымен ауыстырса болады. Оған қоса, оңтайлы жауапта $2 + 2 + 1$ және $20 + 20 + 10$ тиындары болмайды. Себебі біз оларды 5 және 50 тиындармен ауыстыра аламыз.

Осы бақылауларды қолдана отырып, біз әрбір x тиыны үшін тек x мәнінен кіші тиындарды пайдалану арқылы x қосындысын табу немесе кез келген үлкен соманы оңтайлы түрде құру мүмкін еместігін көрсете аламыз. Мысалы, $x = 100$ болса, кішкентай тиындарды қолдану арқылы шығатын ең үлкен және оңтайлы қосынды $50 + 20 + 20 + 5 + 2 + 2 = 99$ болады. Демек әрдайым үлкен тиынды алатын ашкөз алгоритм тиімді жауап береді.

Бұл мысал осындай қарапайым ашкөз алгоритмнің дұрыс жұмыс істейтінін дәлелдеу қиын екендігін көрсетеді.

Жалпы жағдай

Жалпы жағдайда тиын жиыны әртүрлі тиындардан тұруы мүмкін және ашкөз алгоритм міндетті түрде оңтайлы жауап бермейді.

Ашкөз алгоритмнің дұрыс жұмыс істемейтінін қарсы мысал келтіріп көрсетсек болады. Басқаша айтқанда, алгоритм қате жауап беретін мысал келтірсек болады. Мәселен, тиындар $\{1, 3, 4\}$ және қорытынды қосынды 6 болған жағдайды алайық. Алгоритм $4 + 1 + 1$ жауабын тапқанымен, оңтайлы жауап $3 + 3$ болады.

Есептің жалпы барысын шығаратын ашкөз алгоритмнің болу-болмауы белгісіз¹. Алайда, 7-тарауда, кейбір жағдайда бұл есептің жалпы түрін динамикалық бағдарламалау арқылы дұрыс шығаруға болатынын көреміз.

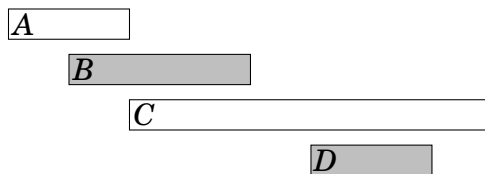
6.2 Жоспарлау

Көптеген жоспарлау есептерін ашкөз алгоритм арқылы шығаруға болады. Көрнекілік үшін классикалық есеп түрін келтірейік: Бізге n оқиғаның басталуы мен аяқталу уақыттары берілген. Мүмкіндігінше көп оқиғаларды қамтитын жоспарды табыңыз. Бір оқиғаны жартылай алуға болмайды және алған оқиғалар өзара қиылыспау керек. Мысалы, төмендегі оқиғаларды қарастырайық:

оқиға	басталу уақыты	аяқталу уақыты
<i>A</i>	1	3
<i>B</i>	2	5
<i>C</i>	3	9
<i>D</i>	6	8

Бұл жағдайда ең көп оқиғалар саны екіге тең. Үлгі ретінде *B* және *D* оқиғаларын алсақ болады:

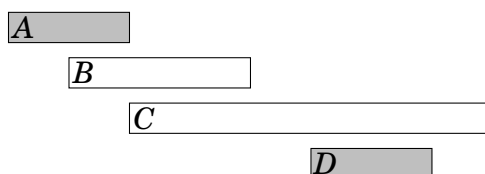
¹Дегенмен ашкөз алгоритмнің берілген тиындардың жиынында қалай жұмыс істейтінін полиномдық уақытта тексеруге болады [17].



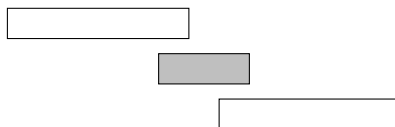
Бұл есепті шығаратын бірнеше ашкөз алгоритмдерді құрастыруға болады. Солардың қайсысы әрдайым дұрыс жауапты ұсына алмақ?

1-алгоритм

Бірінші идея – мүмкіндігінше қысқа оқиғаларды алу. Бұл жағдай үшін алгоритміміз келесі оқиғаларды таңдайды:



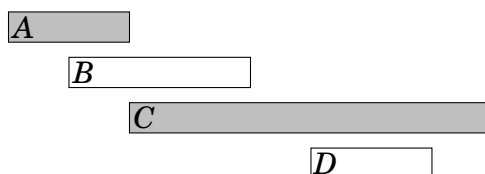
Бірақ, қысқа оқиғаларды алу әрдайым оңтайлы бола бермейді. Мысалы, келесі жағдайда алгоритм қате жауап шығарады:



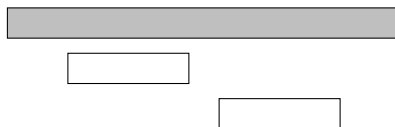
Егер қысқа оқиғаны таңдасақ, нәтижесінде бір ғана оқиғамен қаламыз. Алайда, тиімді жауап екі ұзын оқиға еді.

2-алгоритм

Келесі идея – әрдайым ертерек басталатын және алуға мүмкін оқиғаны таңдау. Бұл алгоритм төмендегідей жұмыс жасайды:



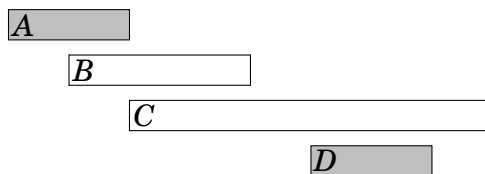
Алайда бұл алгоритмге қарсы үлгі таба аламыз. Мысалы, төмендегі жағдайда алгоритм тек бір оқиғаны алады:



Егер біз бірінші оқиғаны алатын болсақ, онда басқа оқиғаларды ала алмаймыз. Дегенмен бұл жерде басқа екі оқиғаны алған тиімдірек.

3-алгоритм

Үшінші идея – қолжетімді ең ерте аяқталатын оқиғаны алып отыру. Алгоритм төменде келтірілгендей жұмыс жасайтын болады:



Бұл алгоритм әрдайым оңтайлы жауап қайтарады. Өйткені мүмкіндігінше ерте аяқталатын оқиғаны таңдау әрқашан оңтайлы болмақ. Келесі оқиғаларды да осындай стратегиямен таңдаймыз, жарамды оқиғалар таусылғанша осылай жалғастыра береміз.

Бұл алгоритмді дәлелдеу үшін екі оқиғаның қайсысын таңдау тиімдірек болатынын қарастырайық. Екеуінен кешірек аяқталатынын таңдасақ, таңдау саны азаяды. Себебі ол басқа оқиғалардың алынбауына әсер етеді. Кеш аяқталатын оқиғаны алу ешқашан тиімді жауап бермейді. Сол үшін де жобалаған алгоритміміз дұрыс болмақ.

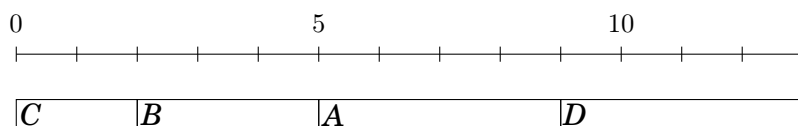
6.3 Тапсырмалар және ақырғы мерзімдер

Келесі есепті қарастырайық: Бізге n тапсырмалардың алатын уақыты және ақырғы мерзімдері берілген. Әрбір орындалған тапсырмаға $d - x$ ұпай аламыз, мұнда d – тапсырманың ақырғы мерзімі және x – тапсырманың орындалған уақыты. Тапсырмаларды қандай ретпен орындау арқылы ең көп ұпай аламыз?

Мысал:

тапсырма	алатын уақыты	ақырғы мерзім
<i>A</i>	4	2
<i>B</i>	3	5
<i>C</i>	2	7
<i>D</i>	4	5

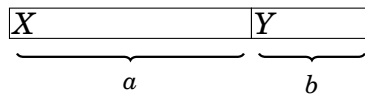
Бұл кездегі оңтайлы орындау ретіміз төмендегідей болады:



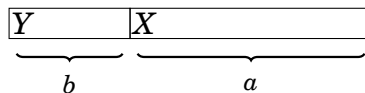
Бұл жауапта *C* тапсырмасы 5 ұпай, *B* тапсырмасы 0 ұпай, *A* тапсырмасы –7 ұпай және *D* тапсырмасы –8 ұпай береді. Қорытынды ұпай саны – –10.

Қанша таңғаларлық болса да, оңтайлы шешім ақырғы мерзімдерге тәуелді емес. Бұл жердегі дұрыс ашкөз стратегия – тапсырмаларды орындау уақытының өсуі бойынша атқару. Оны былай дәлелдей аламыз: Екі тапсырманы қарастырайық. Бірінші тапсырманың орындалу уақыты екінші тапсырманың орындалу уақытынан ұзағырақ. Егер олардың орындарын ауыстырсақ

жауапты тиімдірек ете аламыз. Осылайша біз орындалу уақыттары қысқа болатын тапсырмаларды бірінші орындау керектігін дәлелдейміз. Тағы бір мысал ретінде келесі кестені келтірейік:



Бұл жерде $a > b$, сол үшін олардың орындарын ауыстыру қажет:



X оқиғасы b ұпай кемітсе, Y оқиғасы a ұпай арттырды. Қорытынды ұпай саны $a - b > 0$ санына көбейеді. Оңтайлы шешімде қатарынан келетін әр екі тапсырма үшін алдымен орындалу уақыты тезірек тапсырма атқарылуы қажет. Сол себепті алгоритмді орындалу уақыты бойынша сұрыпталған реттілікпен атқарған жөн.

6.4 Қосындыны минималдау

Келесі есепте n сан a_1, a_2, \dots, a_n берілген. Бізге

$$|a_1 - x|^c + |a_2 - x|^c + \dots + |a_n - x|^c$$

қосындысы барынша аз болатын x санының мәнін табу керек. $c = 1$ және $c = 2$ жағдайларын ғана қарастырайық.

$c = 1$ жағдайы

Бұл жағдайда бізге

$$|a_1 - x| + |a_2 - x| + \dots + |a_n - x|$$

қосындысын барынша азайту керек. Мысалы, сандар $[1, 2, 9, 2, 6]$ болса, оңтайлы жауап $x = 2$. Нәтижесіндегі қосынды:

$$|1 - 2| + |2 - 2| + |9 - 2| + |2 - 2| + |6 - 2| = 12.$$

Жалпы x -тің тиімді мәні – сандардың медианасы (медиана – сандарды реттегеннен кейінгі ортаңғы сан). Мысалы, сандар $[1, 2, 9, 2, 6]$ болса, реттегеннен кейін $[1, 2, 2, 6, 9]$ және медиана 2-ге тең.

Медиана бұл жерде оңтайлы жауап, себебі x медианадан аз болса, x -ті арттырумен қосынды азаяды. Дәл солай x медианадан көп болса да x -ті азайту арқылы қосынды қайтадан азаяды. Демек оңтайлы жауап x -ті медианаға теңеу болып шығады. Егер n жұп сан болса, онда медиана екеу болады. Қайсысын алсаңыз да, жауап оңтайлы болады.

$c = 2$ жағдайы

Бұл жағдайда бізге

$$(a_1 - x)^2 + (a_2 - x)^2 + \dots + (a_n - x)^2$$

қосындысын барынша азайту керек. Мысалы, сандар $[1, 2, 9, 2, 6]$ болса, оңтайлы жауап $x = 4$ деп алайық. Ол бізге төмендегі қосындыны береді:

$$(1 - 4)^2 + (2 - 4)^2 + (9 - 4)^2 + (2 - 4)^2 + (6 - 4)^2 = 46.$$

Жалпы x -тің оңтайлы мәні – сандардың арифметикалық ортасы. Жоғарыда келтірген мысалдағы арифметикалық орта $(1 + 2 + 9 + 2 + 6)/5 = 4$ -ке тең. Осындай нәтижеге қосындыдағы жақшаларды ашу арқылы қол жеткізуге болады:

$$nx^2 - 2x(a_1 + a_2 + \dots + a_n) + (a_1^2 + a_2^2 + \dots + a_n^2)$$

Соңғы бөлімі x -ке тәуелді емес, сондықтан оны ескермеуге болады. Ал қалған бөлімдерін $nx^2 - 2xs$ функциясы ретінде қарастырамыз, мұндағы $s = a_1 + a_2 + \dots + a_n$. Бұл – жауаптары $x = 0$ және $x = 2s/n$ болатын үстіге ашылған парабола. Параболаның ең кіші мәні – жауаптарының арифметикалық ортасы $x = s/n$ -ге тең, мұндағы $s = a_1 + a_2 + \dots + a_n$.

6.5 Деректерді сығымдау

Бинарлық код жолдағы әрбір таңбаға биттерден тұратын код сөз белгілейді. Жолдағы әрбір таңбаны сәйкес келетін код сөзбен алмастырып, жолды сығымдауға болады. Мысалы, келесі бинарлық код A–D таңбаларына код сөз белгілейді:

таңба	код сөз
A	00
B	01
C	10
D	11

Бұл – тұрақты өлшемді код яғни, әрбір код сөздің өлшемі бірдей. Мысалы, біз AABACDACA жолын былай сығымдай аламыз:

000001001011001000

Осы код арқылы сығымдалған жолдың ұзындығы 18 бит болады. Бірақ біз жолдарды айнымалы өлшемді код арқылы тиімдірек сығымдай аламыз. Яғни код сөздердің өлшемі әртүрлі бола алады. Олай болса, көп кездесетін таңбаларға қысқа код сөзін, ал аз кездесетін таңбаларға ұзын код сөзін белгілеуімізге болады. Жоғарыда келтірілген жолға оңтайлы код төмендегідей болады:

таңба	код сөз
A	0
B	110
C	10
D	111

Оңтайлы код ең қысқа болатын сығымдалған жолды береді. Жоғарыдағы мысалда оңтайлы код арқылы сығымдалған жол:

001100101110100.

Осылайша 18 биттің орнына 15 бит қолдандық. Демек жақсы код арқылы сығымдалған жолда 3 бит сақтауға болады.

Код сөз басқа код сөздің префиксі болмауы – код сөзге қойылатын басты талап. Мысалы, кодта 10 және 1011 код сөздерінің болуына рұқсат етілмейді. Себебі сығымдалған жолдан түпкі жолды өндіруде қиындықтар туындауы мүмкін. Егер код сөз басқа код сөздің префиксі болса, түпкі жолды өндіруе алмаймыз. Мысалы, төменде жарамсыз код берілген:

таңба	код сөз
A	10
B	11
C	1011
D	111

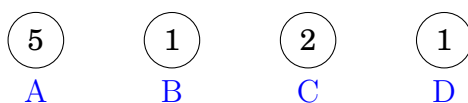
Кодтағы 1011 сығымдалған жолының түпкі жолы АВ жолы немесе С жолы екені нақты белгісіз.

Хаффман кодтауы

Хаффман кодтауы¹ – жолды сығымдау үшін оңтайлы кодты құрастыратын ашкөз алгоритм. Бұл алгоритм таңбалардың жиілігіне сүйеніп бинарлы дарақ құрайды. Әр таңбаның код сөзін дарақтың түбірінен бастап, жапырағына дейінгі жол арқылы тапсақ болады. Егер жүрісіміз солға болса, 0 битке, оңға болса, 1 битке сәйкес болады.

Бастапқыда жолдың әр таңбасын салмағы таңбаның жиілігіне тең төбе деп белгілейік. Кейін әр қадам сайын салмағы ең аз екі төбені алып, олардан салмағы осы екі төбе салмағының қосындысына тең жаңа төбе құраймыз. Бұл процесті барлық төбелер біріктірілгенше жалғастырамыз.

”ААВАСДАСА жолына Хаффман кодтауымен қандай оңтайлы код құрастыра аламыз?” - деген сұрақ туындайды. Басында бізде жолдағы 4 таңбаға сәйкес 4 төбе болады:

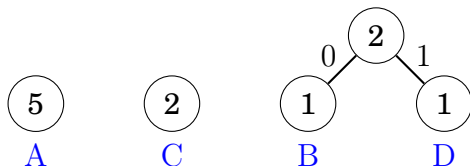


¹Д.А.Хаффман бұл әдісті университеттегі үй жұмысын орындап отырғанда тауып, 1952 жылы жариялаған. [18].

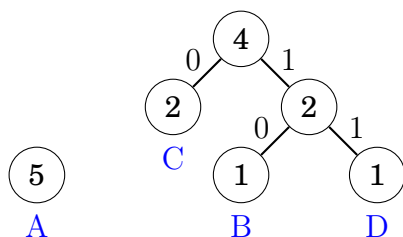
А таңбасын белгілейтін төбенің салмағы 5, себебі А таңбасы жолда 5 рет кездеседі. Басқа төбелердің салмақтары да дәл осылай есептеледі.

Бірінші қадамда салмақтары 1 болатын В және D сәйкес төбелерін біріктіріміз.

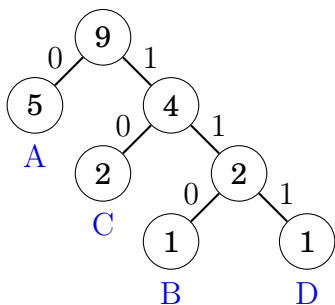
Нәтижесінде:



Бұдан соң салмақтары 2 болатын төбелерді біріктіріміз:



Ақырында соңғы екі төбені біріктіріміз:



Дарақтағы барлық төбелер біріктірілді. Демек кодымыз дайын. Дарақтан таңбаларға сәйкес код сөздерді біле аламыз. Олар:

таңба	код сөз
A	0
B	110
C	10
D	111

7-тарау. Динамикалық бағдарламалау

Динамикалық бағдарламалау – толық ізденістің дұрыстығы мен ашкөз алгоритмнің жылдамдығын біріктіретін әдіс. Егер есеп бірнеше қабаттасқан ішесептерге бөлінсе және әрқайсысы тәуелсіз шешілетін болса, онда динамикалық бағдарламалауды қолдануға болады.

Динамикалық бағдарламалауды екі кезде қолданады:

- Оңтайлы жауапты іздеу. Барынша аз немесе барынша үлкен жауапты тапқымыз келгенде қолданамыз.
- Шешімдердің санын есептеу. Мүмкін болатын шешімдердің жалпы санын есептегіміз келгенде қолданамыз.

Ең алдымен динамикалық бағдарламалау оңтайлы жауапты қалай іздейтінін көреміз, кейін дәл сол идеяны шешімдердің жалпы санын есептегенде қолданамыз.

Динамикалық бағдарламалауды түсіну – әр спорттық бағдарламалаушы үшін үлкен жетістік, айтулы қадам. Бірақ негізгі идеясы қарапайым болғанымен, динамикалық бағдарламалауды түрлі есептерде қолдану барысына келгенде қиындық туындайды. Бұл тарауда динамикалық бағдарламалаудың бастамасы болатын классикалық есептер жинағын ұсынамыз.

7.1 Тиын жайлы есеп

Алдымен 6-тарауда талқыланған есептен бастайық. Мысалы, бізге тиындардың жиыны берілген, қосындысы n болатын тиындарды таңдауымыз керек. Тиындардың мәндері: $\text{coins} = \{c_1, c_2, \dots, c_k\}$. "Ең аз дегенде қанша тиын қажет болады?" - деген сұраққа жауап іздейміз.

Осыған дейін біз бұл есепті әрдайым мүмкін болатын ең үлкен тиынды алатын ашкөз алгоритм арқылы шығардық. Алгоритм әңгіме қазақтың төл валютасы – тиындары туралы болған жағдайда жұмыс істейтіне көзіміз жетті. Бірақ, тұтастай алғанда, ол міндетті түрде тиімді жауап бере бермейді.

Енді бұл есепті барлық тиын жинағына тиімді жауап беретін динамикалық бағдарламалау арқылы шешу жолын қарастырайық. Динамикалық бағдарламалауда болуы мүмкін барлық қосындыларды қарап шығатын рекурсиялық функция негізге алынады. Бірақ, оның тиімділігі мемоизацияның қолданысында, яғни біз әр ішесепті бір рет ғана есептейміз.

Рекурсия анықтамасы

Динамикалық бағдарламалаудың идеясы – есепті кіші ішесептерге бөлу, оларды шығару, жауаптарын біріктіретін рекурсиялық формуланы анықтау. Тиын жайлы есепті қарапайым рекурсияға ауыстырсақ, шарты төмендегідей болады: ”Қосындысы x болатын қанша минималды тиын алуымыз қажет?” - деген сұраққа жауап іздейміз.

$\text{solve}(x)$ – қосындысы x - ке тең минималды тиындар санын қайтаратын функция. Функцияның мәндері тиындардың мәндеріне тәуелді. Мысалы, тиындар $\text{coins} = \{1, 3, 4\}$ болса, функцияның алғашқы мәндері төмендегідей болады:

$\text{solve}(0)$	$=$	0
$\text{solve}(1)$	$=$	1
$\text{solve}(2)$	$=$	2
$\text{solve}(3)$	$=$	1
$\text{solve}(4)$	$=$	1
$\text{solve}(5)$	$=$	2
$\text{solve}(6)$	$=$	2
$\text{solve}(7)$	$=$	2
$\text{solve}(8)$	$=$	2
$\text{solve}(9)$	$=$	3
$\text{solve}(10)$	$=$	3

Үлгіде, $\text{solve}(10) = 3$, себебі қосындысы 10 болатындай ең кемінде 3 тиын қажет. Оңтайлы жауап – $3 + 3 + 4 = 10$.

solve функциясының негізгі қасиеті – функцияның мәндерін кіші аргументтердің функциядағы мәндерінен есептеуі. Идеясы – қосынды үшін таңдайтын бірінші тиынға назар аудару. Мысалы, жоғарыдағы жағдайда бірінші тиын 1, 3 немесе 4 бола алады. Егер біз бірінші тиынды 1 деп алатын болсақ, бізге қосындысы 9 болатын минималды тиындар санын білу қажет. Ал ол – негізгі есептің ішесебі. Демек минималды тиындар санын санайтын төмендегідей рекурсиялық функция құрауымызға болады:

$$\begin{aligned}\text{solve}(x) = \min(\text{solve}(x - 1) + 1, \\ \text{solve}(x - 3) + 1, \\ \text{solve}(x - 4) + 1).\end{aligned}$$

Рекурсияның негізі $\text{solve}(0) = 0$, себебі нөл қосындысын құрау үшін тиын қажет емес. Мысалы,

$$\text{solve}(10) = \text{solve}(7) + 1 = \text{solve}(4) + 2 = \text{solve}(0) + 3 = 3.$$

Енді қосындысы x болатын минималды тиындардың санын есептейтін жалпы рекурсиялық функцияны анықтауға болады:

$$\text{solve}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{c \in \text{coins}} \text{solve}(x - c) + 1 & x > 0 \end{cases}$$

Біріншіден, егер $x < 0$ болса, онда мәні ∞ , өйткені теріс сан қосындысын құрау мүмкін емес. Келесі, егер $x = 0$ болса мәні 0 - ге тең, себебі нөл қосындысын құрау үшін тиын қажет емес. Соңында егер $x > 0$ болса, c айнымалысы қосындының бірінші тиын таңдауындағы барлық мүмкіндіктерінен өтіп шығады.

Есепті шығаратын рекурсиялық функция табылғаннан кейін кодты C++ тілінде жазуымызға болады (INF тұрақтысы шексіздікті білдіреді):

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    return best;
}
```

Дегенмен бұл функция тиімсіз, өйткені қосындыны құру тәсілдерінің экспоненциалды саны бар болуы мүмкін. Мемоизация арқылы функцияны оңтайлырақ ете аламыз.

Мемоизацияның қолданысы

Динамикалық бағдарламалаудың идеясы – мемоизация арқылы рекурсивтік функцияның мәндерін жылдам есептеу, яғни функцияның әр параметрінің мәнін бір рет есептеп, оның мәні қажет болған жағдайда жиымнан алу.

Есепте келесі жиымдарды қолданайық:

```
bool ready[N];
int value[N];
```

Мұндағы `ready[x]` `solve(x)` мәні саналғанын немесе саналмағанын көрсетсе, оның саналған жағдайдағы мәні `value[x]` - те сақталады. N тұрақтысы барлық мәндерді қамтитындай етіп алынған.

Енді осы функцияны төмендегідей етіп, тиімдірек жазуға болады:

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (ready[x]) return value[x];
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    value[x] = best;
    ready[x] = true;
    return best;
}
```

Функцияның $x < 0$ және $x = 0$ негізгі жағдайларын бұрынғыдай өңдейміз. Содан соң функция `ready[x]` арқылы `solve(x)` бұрын есептелгенін не есептелмегенін тексереді. Егер есептелген болса, бірден `value[x]` мәнін қайтаруға болады. Басқаша жағдайда функция рекурсия арқылы `solve(x)` мәнін есептеп, оның мәнін `value[x]`-ке сақтайды.

Әр x параметрі бір рет рекурсивті есептелгендіктен, функция тиімдірек жұмыс жасайды. `solve(x)` мәні `value[x]`-те сақталғаннан кейін x параметрімен функция тағы шақырылғанда, оны тиімді түрде алуға болады. уақытша күрделілігі – $O(nk)$, мұндағы n қалаған қосыныдының мәні және k тиындардың саны.

`value` жиымын цикл арқылы итеративті түрде құрастыруға болатынын ескеріңіз:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0) {
            value[x] = min(value[x], value[x-c]+1);
        }
    }
}
```

Спорттық бағдарламалаушылардың көп бөлігі өлшемі ықшам болып келетініне және тұрақты фактордың аздығына байланысты осы нұсқаны қолданғанды жөн көреді. Біз алдағы мысалдарда итеративті түрді де пайдаланатын боламыз. Дегенмен динамикалық бағдарламалаудың шешімдерін рекурсивті ойлау әдетте жеңіл екенін есте сақтағанымыз дұрыс.

Шешім құрастыру

Кейде бізден есептің оңтайлы жауабы мен оған жететін үлгіні табуды сұрайды. Мысалы, тиын жайлы есепте бірінші тиын қандай болатынын сақтайтын жаңа жиым ашуға болады:

```
int first [N];
```

Кейін алгоритмді төмендегідей өзгерте аламыз:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0 && value[x-c]+1 < value[x]) {
            value[x] = value[x-c]+1;
            first [x] = c;
        }
    }
}
```

```
}
}
```

Төмендегі код қосындысы n болатын оңтайлы тиындарды көрсетеді:

```
while (n > 0) {
    cout << first[n] << "\n";
    n -= first [n];
}
```

Шешімдердің жолын санау

Тиын жайлы есептің басқа түрін қарастырып көрейік. Бұл жерде "тиындарды қолданып x қосындысын қанша жолмен алуға болады?" - деген сұраққа жауап іздейміз. Мысалы тиындар $\text{coins} = \{1, 3, 4\}$ және $x = 5$ болса, 6 жол бар:

- $1 + 1 + 1 + 1 + 1$
- $3 + 1 + 1$
- $1 + 1 + 3$
- $1 + 4$
- $1 + 3 + 1$
- $4 + 1$

Есепті қайтадан рекурсия арқылы шығаруға да болады. $\text{solve}(x)$ функциясын x қосындысын құруға болатын жолдардың саны деп белгілейік. Мысалы, тиындар $\text{coins} = \{1, 3, 4\}$, $\text{solve}(5) = 6$ және рекурсивтік формула төмендегідей болса:

$$\begin{aligned} \text{solve}(x) = & \text{solve}(x - 1) + \\ & \text{solve}(x - 3) + \\ & \text{solve}(x - 4). \end{aligned}$$

Жалпы рекурсивтік функция келесі үлгідегідей болады:

$$\text{solve}(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum_{c \in \text{coins}} \text{solve}(x - c) & x > 0 \end{cases}$$

Егер $x < 0$ болса, онда мәні 0, себебі жауап жоқ. Егер $x = 0$ болса, онда мәні 1, себебі қосындысы 0 болатын тек бір ғана жол бар. Әйтпесе біз $\text{solve}(x - c)$ өрнегіндегі барлық мәндер қосындысын есептеп шығамыз, мұндағы c coins жиымының элементі.

Төмендегі код параметрі $0 \leq x \leq n$ болатын count мәндерінен құралған $\text{count}[x]$ жиымын құрастырады:

```
count[0] = 1;
for (int x = 1; x <= n; x++) {
    for (auto c : coins) {
        if (x - c >= 0) {
```



```

        count[x] += count[x-c];
    }
}

```

Әдетте жолдардың саны тым үлкен болғандықтан, оның нақты санын табу міндетті емес. Кей есепте оның m -ға бөлінгендегі қалдығын сұрайды (мысалы $m = 10^9 + 7$). Барлық есептеулер m модулі бойынша орындалатындай етіп кодты өзгерте аламыз.

Жоғарыдағы кодта төмендегі жолдан кейін

```
count[x] += count[x-c];
```

келесі жолды жазсақ жеткілікті

```
count[x] %= m;
```

Біз динамикалық бағдарламалаудың барлық негізгі тақырыптарын талдадық. Қолданыс аясы кең болғандықтан оның мүмкіндіктерін көрсететін бірнеше есепті қарастырамыз.

7.2 Ең ұзын өспелі іштізбек


Бірінші есебіміз – өлшемі n болатын жиымнан ең ұзын өспелі іштізбекті табу. Бұл – ұзындығы максималды болатын жиым элементтері солдан оңға қарай өсу ретімен орналасқан тізбек.

Мысалы, төмендегі жиым

0	1	2	3	4	5	6	7
6	2	5	1	7	4	8	3

ең ұзын өспелі іштізбектің өлшемі 4 болады:

0	1	2	3	4	5	6	7
6	2	5	1	7	4	8	3



k позициясына бітетін ең ұзын өспелі іштізбектің өлшемін $\text{length}(k)$ деп белгілейік. Осылайша $\text{length}(k)$ функциясының $0 \leq k \leq n - 1$ параметрлерінің мәндерін есептесек, жиымның ең ұзын өспелі іштізбегінің өлшемін таба аламыз. Мысалы, жоғарыда келтірілген жиымдағы функцияның мәндері мынадай болады:

```

length(0) = 1
length(1) = 1
length(2) = 2
length(3) = 1
length(4) = 3
length(5) = 2
length(6) = 4
length(7) = 2

```

Мысалы, $\text{length}(6) = 4$, себебі 6-позицияда бітетін ең ұзын өспелі іштізбек 4 элементтен тұрады.

$\text{length}(k)$ мәнін есептеу үшін $i < k$ позициясында $\text{array}[i] < \text{array}[k]$ шарты орындалуы және $\text{length}(i)$ мүмкіндігінше үлкен болуы қажет. Бұдан $\text{length}(k) = \text{length}(i) + 1$ болатынын байқаймыз. Бұл – ішжиымға $\text{array}[k]$ - ні қосудың тиімді жолы. Бірақ i табылмаса, $\text{length}(k) = 1$ болады, яғни іштізбектік тек $\text{array}[k]$ элементінен тұрады.

Функцияның әр мәні оның кіші мәндерінен есептелетіндіктен динамикалық бағдарламалауды қолдана аламыз. Төменде келтірілетін кодта функцияның мәндері `length` жиымында сақталады.

```

for (int k = 0; k < n; k++) {
    length[k] = 1;
    for (int i = 0; i < k; i++) {
        if (array[i] < array[k]) {
            length[k] = max(length[k], length[i]+1);
        }
    }
}

```

Кодтың уақытша күрделілігі – $O(n^2)$, өйткені ол екі кірістірілген циклден тұрады. Алайда дәл солай $O(n \log n)$ уақытта динамикалық бағдарламалаудың мәндерін анағұрлым тиімдірек есептеуімізге болады. Ал сіз қандай жолдарын білесіз?

7.3 Тордағы жолдар

Біздің келесі есебіміз $n \times n$ торының жоғарғы сол жақ бұрышынан төменгі оң жақ бұрышына дейін тек оңға және төменге қозғалу арқылы жол табуға арналады. Әр шаршыда бір сан жазылған, құрастырылған жолдағы сандардың қосындысы максималды болуы қажет.

Келесі сурет тордағы оңтайлы жолды көрсетеді:

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

Жолдағы сандардың қосындысы 67-ге тең және бұл жол жоғарғы сол жақ бұрыштан төменгі оң жақ бұрышқа апаратын ең үлкен қосынды бола алады.

Тордың жолдары мен бағаналары 1-ден n -ге дейін белгіленген, $value[y][x]$ - тің мәні (y, x) шаршысында жазылған сан. $sum(y, x)$ мәнін жоғарғы сол жақ бұрыштан (y, x) шаршысына дейінгі жолдағы максималды қосынды деп есептейік. Сәйкесінше, $sum(n, n)$ мәні жоғарғы сол жақ бұрыштан төменгі оң жақ бұрышқа дейінгі максималды қосындыны көрсетеді. Жоғарыда көрсетілген торда $sum(5, 5) = 67$.

Қосындыларды рекурсивті түрде былайша санай аламыз:

$$sum(y, x) = \max(sum(y, x - 1), sum(y - 1, x)) + value[y][x]$$

Рекурсивті формула (y, x) шаршысында аяқталатын жол $(y, x - 1)$ шаршысынан немесе $(y - 1, x)$ шаршысынан келетінін байқауға негізделген:

			↓	
		→		

Демек біз қосындыны арттыратын бағытты таңдаймыз. Егер $y = 0$ немесе $x = 0$ болса, $sum(y, x) = 0$ (себебі мұндай жол жоқ). Сондықтан формула $y = 1$ немесе $x = 1$ болған кезде жұмыс жасай береді.

sum функциясында екі параметр болғандықтан динамикалық бағдарламалау жиымы екі өлшемдік болады. Мысалы, төмендегі жиынды пайдалана отырып

```
int sum[N][N];
```

қосындыны келесі үлгідегідей етіп есептей аламыз:

```
for (int y = 1; y <= n; y++) {
    for (int x = 1; x <= n; x++) {
        sum[y][x] = max(sum[y][x-1], sum[y-1][x]) + value[y][x];
    }
}
```

Мұндағы алгоритмнің уақытша күрделілігі – $O(n^2)$.

7.4 Қоржын жайлы есептер

Қоржын туралы есептерге объектілер жиыны берілген және белгілі бір қасиеттері бар ішжиындарды табу талап етілетін есептер жатады. Әдетте оларды динамикалық бағдарламалау арқылы шығаруға болады.

Бұл бөлімде біз келесі есепті қарастырамыз: $[w_1, w_2, \dots, w_n]$ салмақтар тізімі берілген. Салмақтарды пайдаланып құруға болатын барлық қосындыларды анықтау қажет. Мысалы, салмақтар жиымы $[1, 3, 3, 5]$ болса, төмендігей қосындыларды құрастыруға болады:

0	1	2	3	4	5	6	7	8	9	10	11	12
X	X		X	X	X	X	X	X	X		X	X

Бұл жағдайда 2 мен 10-нан басқа $0 \dots 12$ арасындағы барлық қосындыларды құрастыра аламыз. Мысалы, 7 қосындысын $[1, 3, 3]$ салмақтарын таңдай отырып құрастырамыз.

Бұл есепті шығару үшін қосындыларды бастапқы k салмақтан құрастыратын ішесептерге назар аударайық. Егер біз алғашқы k салмақты қолданып, x қосындысын жинай алсақ, $\text{possible}(x, k) = \text{true}$ болады. Ал керісінше жағдайда $\text{possible}(x, k) = \text{false}$ деп белгілейміз. Функцияның мәндерін рекурсивті түрде есептеуге болады:

$$\text{possible}(x, k) = \text{possible}(x - w_k, k - 1) \vee \text{possible}(x, k - 1)$$

Келтірілген формула w_k салмағын қосындыда қолдану немесе қолданбауға негізделген. Егер w_k салмағын қолдансақ, есептің қалған бөлігі алғашқы $k - 1$ салмағын қолданып, қосындысы $x - w_k$ болатын салмақты табуға, ал керісінше оны қолданбасақ, есептің қалған бөлігі алғашқы $k - 1$ салмағын қолданып, қосындысы x болатын салмақты табуға арналады. Бұл ретте төмендегідей негізгі жағдайлар туындайды:

$$\text{possible}(x, 0) = \begin{cases} \text{true} & x = 0 \\ \text{false} & x \neq 0 \end{cases}$$

өйткені салмақтарды алмасақ, біз тек 0 қосындысын құрай аламыз.

Келесі кестеде $[1, 3, 3, 5]$ салмақтары үшін функцияның барлық мәндері көрсетілген ("X" таңбасы ақиқат мәнін көрсетеді):

$k \backslash x$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	X												
1	X	X											
2	X	X		X	X								
3	X	X		X	X		X	X					
4	X	X		X	X	X	X	X	X	X		X	X

Мәндерді есептеп болғаннан кейін, $\text{possible}(x, n)$ барлық салмақтар арқылы x қосындысын құрастыру мүмкіндігін көрсетеді.

W деп салмақтардың жалпы қосындысын белгілейік. Уақытша күрделілігі $O(nW)$ болатын динамикалық бағдарламалау арқылы құрастырылған шешім төмендегі рекурсивтік функцияға сәйкес келеді:

```
possible[0][0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = 0; x <= W; x++) {
        if (x-w[k] >= 0) possible[x][k] |= possible[x-w[k]][k-1];
        possible[x][k] |= possible[x][k-1];
    }
}
```

Дегенмен бір өлшемдік `possible[x]` жиымды ғана қолданатын тиімді код жазуға да болады. Бұл жерде `possible[x]` ішжиымдардан x қосындысын құруға болатынын немесе болмайтынын көрсетеді. Аздаған құлық жасап, әр жаңа салмақ үшін жиымды оңнан солға қарай жаңартып отыруға болады:

```
possible[0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = W; x >= 0; x--) {
        if (possible[x]) possible[x+w[k]] = true;
    }
}
```

Ұсынылған жалпы идеяны қоржын жайлы басқа да есептерде қолдануға болатынын ескеріңіз. Мысалы, заттардың салмағы мен құндылығы берілген болса, әр салмаққа максималды құндылық беретін ішжиынды тапсақ болады.

7.5 Түзету арақашықтығы

Түзету арақашықтығы немесе Левенштейн арақашықтығы¹ – бір жолды екінші жолға ауыстырудағы түзету операцияларының саны. Рұқсат етілген түзету операциялары:

- таңбаны енгізу (e.g. $ABC \rightarrow ABCA$)
- таңбаны өшіру (e.g. $ABC \rightarrow AC$)
- таңбаны өзгерту (e.g. $ABC \rightarrow ADC$)

Мысалы, LOVE пен MOVIE жолдарының түзету арақашықтығы 2-ге тең. Алдымен $LOVE \rightarrow MOVE$ (өзгерту) операциясын қолданамыз, содан соң $MOVE \rightarrow MOVIE$ (енгізу) операциясын қолданамыз. Бұдан аз операция қолдану арқылы бірінші жолды екінші жолға өзгерте алмаймыз. Бір ғана операция арқылы өзгерту мүмкін емес екендігі анық жайт.

Ұзындығы n болатын x және ұзындығы m болатын y жолдары берілген. Олардың арасындағы түзету арақашықтығын білгіміз келеді. Есепті шығару үшін $x[0 \dots a]$ және $y[0 \dots b]$ префикстерінің арасындағы түзету арақашықтығын $\text{distance}(a, b)$ функциясы түрінде белгілейік. Онда x пен y арасындағы түзету арақашықтығы $\text{distance}(n-1, m-1)$ тең.

¹Осы арақашықтық В. И. Левенштейн атымен аталған. Ол оны бинарлы кодтармен байланыстырып зерттеді. [19].

distance мәндерін төмендегіше есептейік:

$$\text{distance}(a, b) = \min(\text{distance}(a, b-1) + 1, \\ \text{distance}(a-1, b) + 1, \\ \text{distance}(a-1, b-1) + \text{cost}(a, b)).$$

Бұл жерде, егер $x[a] = y[b]$ болса, $\text{cost}(a, b) = 0$, әйтпесе $\text{cost}(a, b) = 1$. Формула x жолын өзгерту үшін төмендегі жолдарды қарастырады:

- $\text{distance}(a, b-1)$: x жолының соңына таңба енгізу
- $\text{distance}(a-1, b)$: x жолының соңғы таңбасын өшіру
- $\text{distance}(a-1, b-1)$: x жолының соңғы таңбасын сәйкестендіру немесе өзгерту

Алғашқы екі жағдайда бір түзету операциясы (енгізу немесе өшіру) қажет. Соңғы жағдайда $x[a] = y[b]$ болса, соңғы таңбаларын сәйкестіндіруге болады. Ал егер олар тең болмаса, 1 түзету операциясын (өзгерту) талап етеді.

Төмендегі кесте жоғарыда келтірілген мысалдың distance мәндерін көрсетеді:

	M O V I E					
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

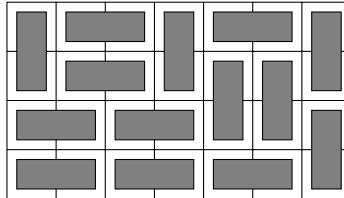
Төменгі оң жақтағы бұрышта LOVE мен MOVIE арасындағы түзету арақашықтығының 2 екенін көрсетеді. Сондай-ақ кестеде түзету операцияларының ең қысқа тізбегін құру жолы берілген. Түзету жолы:

	M O V I E					
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

LOVE пен MOVIE-дің соңғы таңбалары бірдей, сондықтан олардың түзету арақашықтығы LOV және MOVI арасындағы түзету арақашықтығымен бірдей болады. Бір түзету операциясымен MOVI-ден I таңбасын өшірсек жеткілікті. Демек түзету арақашықтығы LOV және MOV арасындағы түзету арақашықтығынан 1-ге үлкен. Ары қарай да дәл осылай жалғаса береді.

7.6 Тақташа төсеу жолдарын санау

Кейде динамикалық бағдарламалаудың шешімдегі күйлері жай тұрақты сандардан күрделірек болады. Мысал үшін келесі есепті қарастырайық: $n \times m$ торда өлшемі 1×2 мен 2×1 тақташаларын төсеу жолдарын табуымыз қажет. 4×7 торы үшін төсеудің жарамды жолын қарастырайық :



Жалпы жолдар саны – 781.

Есепті тордағы жолдардан кезек-кезек өтетін динамикалық бағдарламалау арқылы шығаруға болады. Жауаптағы әр жол $\{\sqcup, \sqsubset, \sqsupset, \sqsupset\}$ жиынындағы m таңбадан тұратын жол ретінде көрсетіле алады. Жоғарыдағы жауап төмендегі төрт жолға сәйкес келеді:

- $\sqsubset \sqsupset \sqsupset \sqsubset \sqsupset \sqsupset \sqsubset$
- $\sqsupset \sqsupset \sqsupset \sqsupset \sqsubset \sqsubset \sqsupset$
- $\sqsupset \sqsupset \sqsupset \sqsupset \sqsupset \sqsupset \sqsubset$
- $\sqsupset \sqsupset \sqsupset \sqsupset \sqsupset \sqsupset \sqsupset$

$\text{count}(k, x)$ деп k -жол x -ке сәйкес болатын тордағы $1 \dots k$ жолдары үшін төсеу санын анықтайық. Бұл жерде динамикалық бағдарламалауды қолдануға болады. Себебі жолдың күйі тек алдыңғы жолдың күйімен шектеледі

Егер 1-жолда \sqsupset таңбасы болмаса, n -жолда \sqsubset таңбасы болмаса және қатар жолдар үйлесімді болса, жауап жарамды болады. Мысалы, $\sqsupset \sqsupset \sqsupset \sqsupset \sqsupset \sqsupset \sqsupset$ $\sqsupset \sqsupset \sqsupset \sqsupset \sqsupset \sqsupset \sqsupset$ жолдары үйлесімді, ал $\sqsubset \sqsupset \sqsupset \sqsubset \sqsupset \sqsupset \sqsupset$, $\sqsupset \sqsupset \sqsupset \sqsupset \sqsupset \sqsupset \sqsupset$ жолдары үйлесімді емес.

Жол m таңбадан тұрғандықтан және әрбір таңбаға 4 нұсқа қоюға болатындықтан, жол түрлері санының ең көбі 4^m болады. Демек шешімнің уақытша күрделілігі $O(n4^{2m})$. Себебі әр жолдың $O(4^m)$ күйі бар және алдыңғы жолдың да $O(4^m)$ күйі бар. Іс барысында кіші жағы m болатындай етіп торды айналдырған дұрыс. Өйткені 4^{2m} факторы басым болады.

Егер жолдардың көрінісін ықшамдасақ, есепті тиімдірек шығаруға болады. Бұл жағдайда алдыңғы жолдың қай бағаналарында тік тақтайшаның жоғарғы шаршысы бар екенін білу жеткілікті. Яғни жолды тек \sqsubset және \sqsupset таңбалары арқылы көрсетсек болады, мұнда \sqsupset – \sqsupset , \sqsupset және \sqsupset таңбаларының комбинациясы бар. Осы көріністі қолдана отырып, әр жолдың 2^m түрі болуына сәйкес уақытша күрделілігі $O(n2^{2m})$ тең болады.

Сөз соңында бұл есепті төмендегі таңғажайып формула арқылы шығаруға да болатынын айта кеткіміз келеді: ¹:

$$\prod_{a=1}^{\lceil n/2 \rceil} \prod_{b=1}^{\lceil m/2 \rceil} 4 \cdot (\cos^2 \frac{\pi a}{n+1} + \cos^2 \frac{\pi b}{m+1})$$

Аталмыш формула аса тиімді. Ол төсеу жолдарын $O(nm)$ уақытта табады. Бірақ жауап нақты сандардың көбейтіндісінен тұратындықтан, олардың аралық нәтижелерін дәлдікпен сақтау қиынға соғады.

¹Бұл формуланы 1961 жылы өз бетінше жұмыс істеген екі зерттеу тобы ашты. [20, 21]

8-тарау. Амортизацияланған талдау

Алгоритмнің уақытша күрделілігі жай ғана алгоритм құрылымын зерттеу, алгоритмнің қандай циклдерді қамтитынын және олардың қанша мәрте орындалатынын анықтау арқылы оңай талдана алады. Дегенмен кейде мұндай қарапайым талдау алгоритмнің шын мәнінде қаншалықты тиімді жұмыс жасайтындығы туралы толық ақпарат бермейді.

Амортизацияланған талдауды уақыт күрделілігі түрленіп отыратын операцияларды қамтитын алгоритмдерді талдау үшін қолдануға болады. Талдаудың бұл түрінің негізгі идеясы алгоритмдегі операциялардың жұмыс жасау уақытын өз алдына бөлек бағалаудың орнына бірге бағалауға негізделген болатын.

8.1 Екі нұсқағыш әдісі

Екі нұсқағыш әдісінде жиым мәндерімен өтіп шығу үшін екі нұсқағыш пайдаланылады. Екі нұсқағыш та алгоритм тиімді болатындай тек бір бағытта қозғалады. Енді екі нұсқағыш әдісімен шығаруға болатын есептерді қарастырайық.

Ішжиым қосындысы

Бірінші мысал ретінде n оң бүтін саннан тұратын жиым мен x мақсатты сома берілген есепті қарастырайық. Біз қосындысы x болатын ішжиымды анықтағымыз келеді, болмаған жағдайда жоқ екені туралы хабарлаймыз.

Мысалы, төмендегі жиым

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

қосындысы 8 болатын ішжиымды қатиды:

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

Есеп шешімін екі нұсқағыш әдісімен $O(n)$ уақытта таба аламыз. Идеясы – нұсқағыштарды ішжиымның алғашқы және соңғы мәндеріне сілтеу. Әр жүрісте сол жақ нұсқағыш бір қадам оңға, ал оң жақ нұсқағыш нәтижесінде пайда болған ішжиым қосындысы x -тен аспайтындай позицияға дейін оңға жылжиды.

Егер қосынды x -ке тең болса, шешім табылды.

Үлгі ретінде төмендегі жиым мен мақсатты қосынды $x = 8$ болған жағдайды қарастырайық:

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

Бастапқы ішжиым қосындысы 6 болатын 1, 3 және 2 мәндерін қамтиды:

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

↑ ↑

Кейін сол жақ нұсқағыш бір қадам оңға жылжиды. Оң жақ нұсқағыш орнынан қозғалмайды, себебі қозғалған жағдайда ішжиым қосындысы x -тен асып кетер еді.

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

↑ ↑

Қайтадан сол жақ нұсқағыш бір қадам оңға жылжиды. Осы кезде оң жақтағы нұсқағыш оңға қарай 3 қадам жасайды. Ішжиым қосындысы $-2 + 5 + 1 = 8$, қосындысы x болатын ішжиым табылды.

1	3	2	5	1	1	2	3
---	---	---	---	---	---	---	---

 ↑ ↑

Алгоритмнің өңдеу уақыты оң жақ нұсқағыштың жасайтын қадам санына байланысты болады. Бір итерацияда жасалатын қадамның жоғарғы шегін білмейміз. Біз алгоритм барысында нұсқағыштың жалпылама $O(n)$ қадам жасайтындағын білеміз, себебі ол оңға ғана қозғалады.

Алгоритмдегі сол және оң нұсқағыштардың $O(n)$ қадам жасайтындығына байланысты алгоритм $O(n)$ уақытта жұмыс істейді.

2SUM есебі

Екі нұсқағыш тәсілімен шығарылатын тағы бір есептің, яғни 2SUM есебінің шығарылу барысына тоқталайық. n саннан тұратын жиым мен x мақсатты сомасы берілген, қосындылары x -ке тең екі жиым мәндерін анықтау немесе мүмкін еместігін хабарлау.

Есепті шығару үшін бірінші жиымды өсу реті бойынша сұрыптаймыз. Кейін екі нұсқағышты пайдаланып жиымды өтіп шығамыз. Сол жақ нұсқағыш бірінші мәннен бастап әр жүрісте бір қадам оңға жылжиды. Оң жақ нұсқағыш соңғы мәннен басталып сол және оң мәндердің қосындысы ең көп дегенде x болғанша солға жылжиды. Қосынды мәнінің x болуы шешімнің табылғанын білдіреді.

Мысалы, келесі жиым мен мақсатты сома $x = 12$ түрінде қарастырайық:

1	4	5	6	7	9	9	10
---	---	---	---	---	---	---	----

Нұсқағыштардың бастапқы позициялары көрсетілгендей. Мәндер қосындысы – $1 + 10 = 11$, бұл x -тен кіші.

1	4	5	6	7	9	9	10
↑							↑

Кейін сол жақ нұсқағыш бір қадам оңға жылжиды. Оң жақ нұсқағыш үш қадам солға жылжиды, осылайша пайда болған қосынды – $4 + 7 = 11$.

1	4	5	6	7	9	9	10
	↑			↑			

Бұдан соң сол жақ нұсқағыш қайтадан оңға бір қадам жасайды. Оң жақ нұсқағыш қозғалмайды, солайша $5 + 7 = 12$ шешімі табылды.

1	4	5	6	7	9	9	10
		↑		↑			

Алгоритмнің өңдеу уақыты – $O(n \log n)$, себебі бірінші жиым $O(n \log n)$ уақыт ішінде сұрыпталып, кейін екі нұсқағыш та $O(n)$ қадам жасайды.

Есепті $O(n \log n)$ ішінде бинарлы ізденіс арқылы да шығаруға болатындығын ескеріңіз. Бұл жағдайда біз жиымды өтіп шығып, әр мән үшін қосынды x -ке тең болатындай басқа мән іздейміз. Ол әрқайсысы $O(\log n)$ уақыт алатын n бинарлы ізденіс арқылы орындала алады.

Қосындысы x -ке тең болатын үш жиым мәндерін анықтауды көздейтін 3SUM есебі алдыңғы қарастырған есепке қарағанда қиынырақ. Жоғарыда қарастырылған алгоритмді қолдана отырып бұл есепті $O(n^2)$ уақытта шығара аламыз¹. Қалай шығарылатынын байқадыңыз ба?

8.2 Ең жақын кішірек элементтер

Амортизацияланған талдау деректер құрылымындағы операциялар санына баға беру үшін жиі қолданылады. Операциялар бірқалыпсыз, алгоритмнің белгілі бір бөлігінде жиірек орындалатындай болып таратылуы мүмкін. Дегенмен операциялар саны шектеулі болмақ.

Үлгі ретінде жиымдағы әр элемент үшін ең жақын кішірек элементті, яки ең бірінші болып сол элементке дейін кездесетін кішірек элементті анықтайтын есепті қарастырайық. Ондай элемент болмаған жағдайда алгоритм жоқ екенін хабарлауы қажет. Енді есепті стэк құрылымын пайдалану арқылы қалай тиімді шешуге болатынын көреміз.

¹Ұзақ уақыт бойы 3SUM есебін $O(n^2)$ уақытынан тез шешу мүмкін емес деп ойладық. Бірақ 2014 жылы бұлай еместігі[22] анықталды.

Біз жиым элементтерінің стегін сақтай отырып, жиымды солдан оңға қарай аралаймыз. Әр қадам сайын біз төбедегі элемент ағымдағы элементтен кіші болып шыққанға дейін немесе стек босап қалғанға дейін стэктен элементтерді өшіріп отырамыз. Осыдан кейін біз стэктің төбесіндегі элемент - ағымдағы ең жақын кіші элемент екенін немесе стэк бос болған жағдайда, ондай элементтің жоқтығын хабарлаймыз. Соңында ағымдағы элементті стэкке қосамыз.

Мысал ретінде келесі жиымды қарастырайық:

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

Бірінші 1, 3 және 4 сандары стэкке қосылады, себебі әр элемент алдыңғысынан үлкенірек. Сол себепті 4 үшін ең жақын кішірек элемент – 3, ал 3 үшін ең жақын кішірек элемент 2 болмақ.

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

1 → 3 → 4

Келесі элемент – 2. Ол стэктің жоғарысындағы екі элементтен кіші. Сол себепті 3 пен 4 элементтері стэктен өшіріліп, 2 стэкке қосылады. Оның ең жақын кішірек элементі – 1:

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

1 → 2

Кейін 5 элементі 2-ден үлкенірек болғандықтан осы стэкке қосылып, оның ең жақын кішірек элементі 2 болады:

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

1 → 2 → 5

Бұдан соң 5 элементі стэктен өшіріліп, стэкке 3 пен 4 элементтері қосылды:

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

1 → 2 → 3 → 4

Ақырында 1-ден өзге барлық элементтер өшіріліп, соңғы 2 элементі стэкке қосылады:

1	3	4	2	5	3	4	2
---	---	---	---	---	---	---	---

1 → 2

Алгоритмнің тиімділігі стэк операцияларының жалпы санына тәуелді. Егер алынған элемент стектің жоғарғы элементінен үлкен болса, ол стекке бірден тиімді қосылады. Бірақ кейде стек бірнеше үлкенірек элементтерді қамтуы мүмкін және оларды өшіру уақытты алады. Дегенмен әр элемент стекке бір рет қана қосылады және өшірілуі көп дегенде бір рет орын алады. Сондықтан әр элемент $O(1)$ стек операцияларын тудырып, алгоритм $O(n)$ уақытында орындалады.

8.3 Жылжымалы терезе минимумы

Жылжымалы терезе – константалы-өлшемді ішжиым, ол жиым бойымен солдан оңға қарай жылжиды. Әр терезе позициясында терезе ішіндегі элементтер туралы қандай да бір ақпаратты анықтағымыз келеді. Бұл бөлікте жылжымалы терезе минимумы есебіне назар аударамыз. Мұнда әр терезе ішіндегі ең кіші мәнді хабарлауымыз қажет.

Жылжымалы терезелердегі минимумдарды есептеу үшін ең жақын кішірек элементтерді табу барысындағы идеяны қолдануға болады. Бірақ бұл жолы бізге әр элементі алдыңғысынан үлкенірек, ал бірінші элементі әрдайым терезенің ішіндегі ең аз элементке сәйкес келетін кезек қажет болады. Әр терезе қозғалысы сайын кезек соңынан элементтерді алып отырамыз, бұл кезектің соңғы саны терезенің жаңа элементінен кіші болғанша немесе кезек босағанша жалғасады. Сонымен қатар егер кезектің алғашқы элементі терезеде болмаса, оны өшіреміз. Соңында терезенің жаңа элементін кезек соңына қосамыз.

Мысал ретінде келесі жиымды қарастырайық:

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

Жылжымалы терезе өлшемін 4 деп алайық. Терезенің бірінші позициясында ең кіші мән – 1:

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1 → 4 → 5

Кейін терезе бір қадам оңға жылжиды. Жаңа элемент 3 кезектегі 4 пен 5 элементтерінен кішірек, сондықтан 4 пен 5 элементтері кезектен өшіріліп, 3 қосылды. Ең кіші элемент әлі де 1 бола береді.

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1 → 3

Бұдан соң терезе тағы жылжиды, енді ең кіші элемент 1 терзеге кірмейді. Сондықтан ол кезектен өшіріліп, ең кіші мән 3-ке теңеседі. Сонымен қатар жаңа элемент 4 кезекке қосылады.

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

3 → 4

Келесі жаңа элемент 1 кезектегі барлық элементтерден кіші. Сол себепті барлық элементтер өшіріліп, кезек енді тек 1 элементін қамтиды:

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1

Ақырында терезе өзінің соңғы позициясына жетеді. 2 элементі кезекке қосылады, бірақ терезедегі ең кіші элемент 1 болып қала береді.

2	1	4	5	3	4	1	2
---	---	---	---	---	---	---	---

1 → 2

Жиымның әр элементі кезекке тек бір рет қосылғандықтан және кезектен көп дегенде бір-ақ рет өшірілгендіктен, алгоритм $O(n)$ уақытта жұмыс жасайды.

9-тарау. Аралық сұратымдар

Бұл тарауда аралық сұратымдарды тиімді өңдейтін деректер құрылымдары жайлы сөз қозғайтын боламыз. Әдетте аралық сұратымдар есепте ішжиымдарға негізделген мәндерді табуға арналады. Қарапайым аралық сұратымдар:

- $\text{sum}_q(a, b)$: $[a, b]$ аралығындағы сандардың қосындысын есептеу
- $\text{min}_q(a, b)$: $[a, b]$ аралығындағы сандардың минимумын есептеу
- $\text{max}_q(a, b)$: $[a, b]$ аралығындағы сандардың максимумын есептеу

Мысалы, төмендегі жиымдағы $[3, 6]$ арасын қарастырайық:

0	1	2	3	4	5	6	7
1	3	8	4	6	1	3	4

Қарастырылған жағдайда $\text{sum}_q(3, 6) = 14$, $\text{min}_q(3, 6) = 1$ және $\text{max}_q(3, 6) = 6$.

Бұл аралық сұратымды өңдеудің ең қарапайым жолы – аралықтың барлық элементтерін өтіп шығатын циклды қолдану. Мысалы, төмендегі функция қосынды сұратымдарын өңдейді:

```
int sum(int a, int b) {  
    int s = 0;  
    for (int i = a; i <= b; i++) {  
        s += array[i];  
    }  
    return s;  
}
```

Функция $O(n)$ уақытта жұмыс жасайды, мұндағы n – жиымның өлшемі. Демек q сұратымды $O(nq)$ уақытта өңдейміз. Егер n мен q екеуі де үлкен болса, бұл тәсіл біз үшін баяу. Бақытымызға орай, аралық сұратымдарды әлдеқайда тезірек өңдеу жолдары да бар.

9.1 Статикалық жиым сұратымдары

Алдымен жиым статикалық болатын (яғни жиымның мәндері сұратым араларында өзгермейтін) жағдайды қарастырайық. Бұл жағдайда барлық сұратымдарға жауап беретін статикалық деректер құрылымын құрыстыру жеткілікті.

Қосынды сұратымдары

Префиксті қосындылар жиымын құрыстыру арқылы біз статикалық жиымның қосынды сұратымдарын жеңіл есептей аламыз. Префиксті қосындылар жиымындағы әр мән негізгі жиымдағы алғашқы элементтен бастап сол позицияға дейінгі сандардың қосындысына тең болады, яғни k позициясындағы мән $\text{sum}_q(0, k)$ -ға тең. Префиксті қосындылар жиымы $O(n)$ уақытта құрастырылады.

Мысалы төмендегі жиымды қарастырайық:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

Оған сәйкес префиксті қосындылар жиымы:

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

Жиым $\text{sum}_q(0, k)$ -ның барлық мәндерін қамтығандықтан, біз қалаған $\text{sum}_q(a, b)$ мәнін $O(1)$ уақытта төмендегідей жолмен таба аламыз:

$$\text{sum}_q(a, b) = \text{sum}_q(0, b) - \text{sum}_q(0, a - 1)$$

$\text{sum}_q(0, -1) = 0$ деп санасақ, $a = 0$ болған жағдайда да формула жұмыс жасай береді.

Мысалы, $[3, 6]$ аралығын қарастырайық:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

Бұл жағдайда $\text{sum}_q(3, 6) = 8 + 6 + 1 + 4 = 19$. Қосындыны префиксті қосындылар жиымының екі мәні арқылы есептей аламыз:

0	1	2	3	4	5	6	7
1	4	8	16	22	23	27	29

Осылайша $\text{sum}_q(3, 6) = \text{sum}_q(0, 6) - \text{sum}_q(0, 2) = 27 - 8 = 19$.

Бұл идеяны әрі қарай көп өлшемдерге жалпылауға болады. Мысалы, екі өлшемді префиксті қосындылар жиымын құрастырып, төртбұрыш ішжиымдардың қосындыларын $O(1)$ уақытта таба аламыз. Жиымдағы әр қосынды жоғарғы сол жақ бұрыштан басталатын ішжиымға есептеледі.

Төмендегі сурет аталған идеяны бейнелейді:

		<i>D</i>				<i>C</i>			
		<i>B</i>				<i>A</i>			

Сұр түспен белгіленіп тұрған ішжиымның қосындысын

$$S(A) - S(B) - S(C) + S(D)$$

формуласы арқылы табуға болады, мұндағы $S(X)$ үстіңгі сол жақ бұрыштан басталып, X позициясында аяқталатын ішжиым мәндерінің қосындысы.

Минимум сұратымдары

Минимум сұратымдарын өңдеу қосынды сұратымдарды өңдеуден қиынырақ. Дегенмен $O(n \log n)$ уақыт ішінде алдын ала өңдеу арқылы кез келген минимум сұратымдарын $O(1)$ уақытта таба аламыз¹. Максимум мен минимум сұратымдары бірдей өңделетіндіктен, минимум сұратымдарын ғана қарастырайық.

Идеясы барлық $\min_q(a, b)$ мәндерін алдын ала есептеуге негізделеді. Мұндағы $b - a + 1$ (аралықтың ұзындығы) екiнiң дәрежесi. Мысалы, төмендегі жиым үшін

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

келесі мәндер есептеледі:

<i>a</i>	<i>b</i>	$\min_q(a, b)$	<i>a</i>	<i>b</i>	$\min_q(a, b)$	<i>a</i>	<i>b</i>	$\min_q(a, b)$
0	0	1	0	1	1	0	3	1
1	1	3	1	2	3	1	4	3
2	2	4	2	3	4	2	5	1
3	3	8	3	4	6	3	6	1
4	4	6	4	5	1	4	7	1
5	5	1	5	6	1	0	7	1
6	6	4	6	7	2			
7	7	2						

¹Бұл техника алғаш [23] көрсетілді және кейде сиретілген кесте әдісі деп аталады. Күрделі техникалар [24] арқылы алдын ала өңдеу уақыты $O(n)$ болатын алгоритмдер де бар, бірақ олар спорттық бағдарламалауда қажетсіз.

Алдын ала есептелген мәндердің саны $O(n \log n)$, себебі ұзындығы екінің дәрежесі болатын аралықтардың саны $O(\log n)$. Мәндер келесі рекурсиялық формула арқылы тиімді есептеледі:

$$\min_q(a, b) = \min(\min_q(a, a + w - 1), \min_q(a + w, b)),$$

мұндағы $b - a + 1$ екінің дәрежесі және $w = (b - a + 1)/2$. Барлық мәндерді есептеу $O(n \log n)$ уақыт алады.

Осыдан кейін кез келген $\min_q(a, b)$ мәнін екі алдын ала есептелген мән арқылы $O(1)$ уақытта есептей аламыз. k деп $b - a + 1$ -ден аспайтын екінің ең үлкен дәрежесін алайық. $\min_q(a, b)$ мәнін төмендегі формула арқылы есептеуге болады:

$$\min_q(a, b) = \min(\min_q(a, a + k - 1), \min_q(b - k + 1, b)).$$

Формулада $[a, b]$ аралығы ұзындықтары k болатын $[a, a + k - 1]$ және $[b - k + 1, b]$ аралықтарының бірігуі деп көрсетілген.

Мысалы, $[1, 6]$ аралығын қарастырайық:

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

Аралықтың ұзындығы – 6, ал 6-дан аспайтын екінің ең үлкен дәрежесі – 4. Ендеше, $[1, 6]$ аралығы – $[1, 4]$ аралығы мен $[3, 6]$ аралығының бірігуінен тұрады.

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

0	1	2	3	4	5	6	7
1	3	4	8	6	1	4	2

$\min_q(1, 4) = 3$ және $\min_q(3, 6) = 1$ болғандықтан, $\min_q(1, 6) = 1$ деп тұжырымдаймыз.

9.2 Бинарлы индекстелген дарақ

Бинарлы индекстелген дарақ немесе Фенвик дарағын ¹ префиксті қосындының динамикалық нұсқасы деп қарастыруға болады. Ол жиымға қатысты екі операцияны – аралық қосындыны табу мен мәнді өзгертуді $O(\log n)$ уақытта орындайды.

Бинарлы индекстелген дарақтың артықшылығы – аралық қосындылармен қатар жиымдағы мәндерді тиімді өзгертуінде. Бұл қасиет префиксті қосындылар жиымында жоқ. Әр жаңарту операциясы префиксті қосындылар жиымын басынан бастап қайта құрастыратындықтан, ол $O(n)$ уақыт алады.

¹Бинарлы индекстелген дарақты 1994 жылы П.М.Фенвик ұсынды [25].

Құрылымы

Құрылымның аты бинарлы индекстелген дарақ болғанымен, әдетте ол жиым ретінде көрсетіледі. Бұл бөлімде жиымды бірлік индекстелген деп қарастырамыз, себебі ол кодтың жазылу барысын жеңілдетеді.

$p(k)$ деп k -ні бөле алатын ең үлкен екінің дәрежесін белгілейік. Бинарлы индекстелген дарақты

$$\text{tree}[k] = \text{sum}_q(k - p(k) + 1, k)$$

болатындай tree жиымында сақтаймыз, яғни әр k позициясында берілген жиымның ұзындығы $p(k)$ болатын және k позициясында бітетін аралықтың қосындысын сақтаймыз. Мысалы, $p(6) = 2$ болғандықтан $\text{tree}[6] = \text{sum}_q(5, 6)$ мәнін сақтайды.

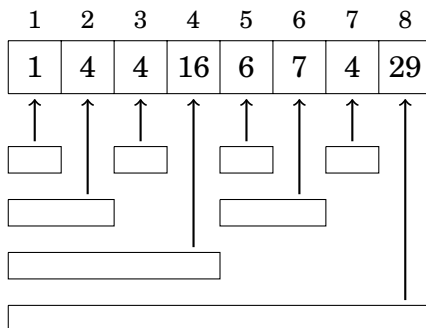
Өрнек ретінде төмендегі жиымды қарастырайық:

1	2	3	4	5	6	7	8
1	3	4	8	6	1	4	2

Осыған сәйкес бинарлы индекстелген дарақ:

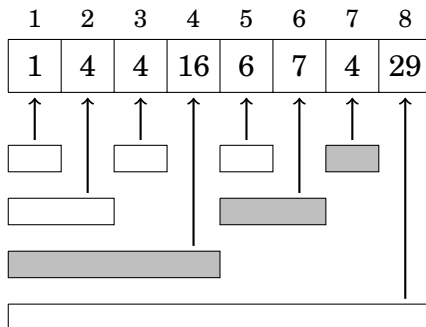
1	2	3	4	5	6	7	8
1	4	4	16	6	7	4	29

Төмендегі сурет бинарлы индекстелген дарақтың әр мәнінің әуелгі жиымдағы қандай аралыққа сәйкестігін анық көрсетеді:



Қарастырылған дарақ арқылы қалаған $\text{sum}_q(1, k)$ мәнін $O(\log n)$ уақытта есептей аламыз. Себебі $[1, k]$ аралығын әрдайым қосындылары дарақта сақталатын $O(\log n)$ аралыққа бөлуге болады.

Мысалы, $[1, 7]$ аралығы төмендегідей аралықтарға бөлінеді:



Демек біз қосындыны төменгідей жолмен есептей аламыз:

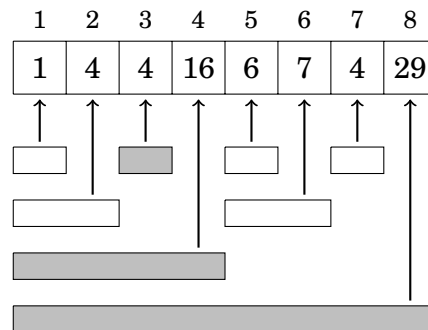
$$\text{sum}_q(1, 7) = \text{sum}_q(1, 4) + \text{sum}_q(5, 6) + \text{sum}_q(7, 7) = 16 + 7 + 4 = 27$$

$a > 1$ болатын $\text{sum}_q(a, b)$ мәнін есептеу үшін префиксті қосындылар жиымында қолданған амалды бұл жерде де қолдануға болады:

$$\text{sum}_q(a, b) = \text{sum}_q(1, b) - \text{sum}_q(1, a - 1).$$

$\text{sum}_q(1, b)$ және $\text{sum}_q(1, a - 1)$ мәндерін $O(\log n)$ уақытта есептей алғандықтан, қорытынды уақытша күрделілігі $O(\log n)$ болады.

Берілген жиымдағы мәnnің өзгеруіне байланысты бинарлы индекстелген дарақта да бірнеше мәндер өзгеруі тиіс. Мысалы, 3-позициядағы мән өзгерсе, төмендегі аралықтардың қосындылары да өзгереді:



Жиымдағы әр элемент бинарлы индекстелген дарақтағы $O(\log n)$ аралығының құрамында болғандықтан, дарақтағы $O(\log n)$ мәнін өзгерткен жеткілікті.

Кодтың жазылуы

Бинарлы индекстелген дарақ операцияларын биттік операциялар арқылы тиімді жүзеге асыра аламыз. Бұл жердегі маңызды факт – әр $p(k)$ мәнін

$$p(k) = k \& -k.$$

формуласы арқылы есептеу.

Төмендегі функция $\text{sum}_q(1, k)$ мәнін есептейді:

```
int sum(int k) {
    int s = 0;
    while (k >= 1) {
        s += tree[k];
        k -= k & -k;
    }
    return s;
}
```

Ал келесі функция жиымның k позициясындағы мәніне x -ті қосады (x оң немесе теріс сан бола алады):

```
void add(int k, int x) {
    while (k <= n) {
        tree[k] += x;
        k += k&-k;
    }
}
```

Екі функцияның уақытша күрделілігі – $O(\log n)$, өйткені функциялар бинарлы индекстелген дарақтағы $O(\log n)$ мәнді қарап, әр келесі позицияға өту үшін $O(1)$ уақыт жұмсайды.

9.3 Кесінділер дарағы

Кесінділер дарағы¹ — екі түрлі операцияны, атап айтқанда аралық сұратымдарды өңдеу және жиымдағы мәнді жаңарту операцияларын қолдайтын деректер құрылымы. Кесінділер дарағы қосындыны табу, минимум мен максимумды анықтау және басқа да сұратымдарға $O(\log n)$ уақытта жауап бере алады.

Бинарлы индекстелген дарақпен салыстырғанда кесінділер дарағының артықшылығы – жалпы деректер құрылымы болуында. Бинарлы индекстелген дарақ тек қосынды сұратымдарын қолдаса², кесінділер дарағы өзге сұратымдарды да қолдайды. Есесіне, кесінділер дарағы көп жадыны алады және оны кодта жазу сәл қиынырақ болады.

Құрылымы

Кесінділер дарағы – ең төменгі деңгейде орналасқан төбелері жиымның элементтеріне сәйкес келетін және басқа төбелері аралық сұратымдарды өңдеуге қажет ақпаратты сақтайтын бинарлы дарақ.

Бұл бөлімде біз жиымның өлшемі екінің дәрежесіне тең және жиым нөлдік индекстелген жағдайды қарастырамыз. Өйткені мұндай жиым үшін кесінділер дарағын құрастыру оңайырақ. Егер жиымның өлшемі екінің дәрежесі болмаса, онда біз әрдайым қосымша элементтер қоса аламыз.

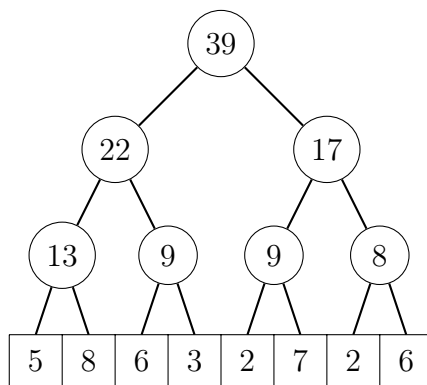
Алдымен қосынды аралық сұратымдарын қолдайтын кесінділер дарағын қарастырайық. Өрнек ретінде төмендегі жиымды алайық:

0	1	2	3	4	5	6	7
5	8	6	3	2	7	2	6

Ол үшін құрылған кесінділер дарағы:

¹Бөлімдегі төменнен жоғарыға қарай код жазылуы [26]-ге сәйкес келеді. Ұқсас құрылымдар 1970 жылдардың аяғында геометриялық есептерді шешу үшін қолданылған [27].

²Шын мәнінде екі бинарлы индекстелген дарақ арқылы минимум сұратымдарын қолдауға болады [28], бірақ бұл кесінділер дарағына қарағанда әлдеқайда қиынырақ.

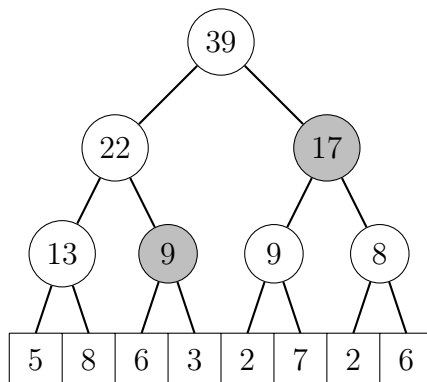


Дарақтағы әрбір ішкі төбенің өлшемі – екінің дәрежесі болатын жиымға пара-пар. Келтірілген дарақтағы ішкі төбелердің мәні – сәйкес келетін жиым элементтерінің қосындысы. Олар сол және оң жақтағы ұл төбелердің қосындысына тең болады.

$[a, b]$ аралығының кез келген мәндері дарақтағы төбелерде сақталған $O(\log n)$ аралықтарға бөлінеді екен. Мысалы, $[2, 7]$ аралығын қарастырайық:

0	1	2	3	4	5	6	7
5	8	6	3	2	7	2	6

Бұл жерде $\text{sum}_q(2, 7) = 6 + 3 + 2 + 7 + 2 + 6 = 26$. Біздің жағдайда төменде берілген дарақтағы екі төбе аралыққа сәйкес келеді:

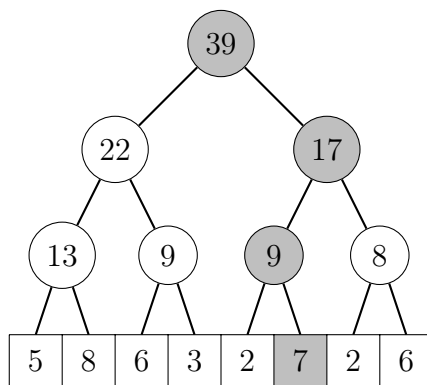


Осылайша, қосындыны есептеудің тағы бір жолы – $9 + 17 = 26$ болып шығады.

Қосындыны мүмкіндігінше ең жоғарғы төбелер арқылы есептейтін болсақ, дарақтағы әр деңгейде ең көбі екі төбе қажет болады. Демек жалпы төбелер саны – $O(\log n)$.

Жиымды жаңартқаннан кейін өзгерген мәнге тәуелді барлық төбелерді жаңартуымыз қажет. Мұны өзгерген жиым элементінен бастап ең жоғарғы төбеге дейінгі жолмен жүріп, жолдағы төбелерді өзгерту арқылы жүзеге асырамыз.

Төмендегі сурет 7 мәні өзгерсе, дарақтағы қандай төбелер өзгеретінін көрсетеді:

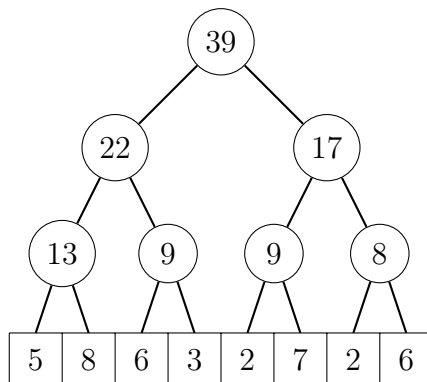


Төменнен жоғарыға дейінгі жол әрдайым $O(\log n)$ төбеден тұрады, сондықтан әр жаңарту дарақтағы $O(\log n)$ төбені жаңартады.

Кодтың жазылуы

Біз кесінділер дарағын $2n$ элементтен тұратын жиымда сақтайтын боламыз, мұндағы n берілген жиымның өлшемі және ол екінші дәрежесіне тең. Дарақтың төбелері жоғарыдан төменге қарай сақталады. Мұндағы $tree[1]$ –ң жоғарғы төбе, $tree[2]$ және $tree[3]$ – оның ұлдары болып жалғаса береді. Соңында $tree[n]$ мен $tree[2n - 1]$ аралығында орналасқан дарақтың төменгі деңгейіндегі мәндер – берілген жиым мәндеріне сәйкес келеді.

Мысалы, төмендегі кесінділер дарағы



төмендегі үлгіде сақталады:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	22	17	13	9	9	8	5	8	6	3	2	7	2	6

Осы көріністі қолдана отырып, $tree[k]$ -ның әкесі $tree[\lfloor k/2 \rfloor]$ болады, ал ұлдарына $tree[2k]$ мен $tree[2k + 1]$ жатады. Осылайша, төбенің позициясы жұп болса, онда ол сол жақ ұлы, ал тақ болса, ол оң жақ ұлы екенін көрсетеді.

Төмендегі функция $sum_q(a, b)$ мәнін есептейді:

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
```

```

while (a <= b) {
    if (a%2 == 1) s += tree[a++];
    if (b%2 == 0) s += tree[b--];
    a /= 2; b /= 2;
}
return s;
}

```

Функция басында $[a+n, b+n]$ аралықты ұстап тұрады. Сосын келесі қадамда аралық бір деңгей жоғары қозғалады және оған дейін үстіңгі аралыққа кірмейтін төбелердің мәні қосындыға қосылады.

Төмендегі функция k позициясындағы жиым мәнін x -ке арттырады:

```

void add(int k, int x) {
    k += n;
    tree[k] += x;
    for (k /= 2; k >= 1; k /= 2) {
        tree[k] = tree[2*k] + tree[2*k+1];
    }
}

```

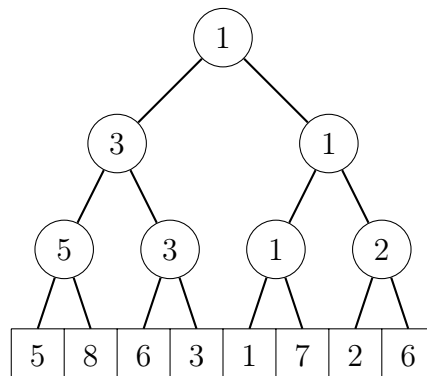
Алдымен функция дарақтың төменгі деңгейіндегі мәнді жаңартады. Кейін дарақтың жоғарғы төбесіне жеткенге дейін, барлық ішкі төбелердің мәндерін жаңартады.

Үстідегі екі функция $O(\log n)$ уақытта жұмыс істейді: n элементтен тұратын кесінділер дарағы $O(\log n)$ деңгейден тұрады және әр қадам сайын бір деңгей жоғары көтеріледі.

Басқа сұратымдар

Кесінділер дарағы аралықты екіге бөліп, жауапты бөлек есептеп, кейін оларды тиімді біріктіретін барлық аралық сұратымдарды қолдайды. Осы типтегі сұратымдар – минимум, максимум, ең үлкен ортақ бөлгіш және биттік операциялар – конъюнкция, дизъюнкция, жоюшы дизъюнкция (xor).

Мысалы, төмендегі кесінділер дарағы минимум сұратымдарды қолдайды:

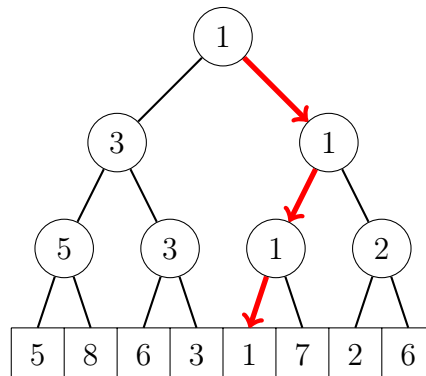


Бұл жағдайда дарақтың әр төбесі жиымның аралығына сәйкес ең кіші мәнді сақтайды. Дарақтағы ең жоғарғы төбе бүкіл жиымның ең кішкентай

мәнін сақтайды. Операцияларды бұрынғыдай орындауға болады, бірақ бұл жағдайда қосындылар орнына минимумдар есептеледі.

Кесінділер дарағының құрылымы жиымның элементтерін бинарлық ізденіс арқылы табуға жол береді. Мысалы, егер дарақ минимум сұратымдарын қолдаса, онда ең кішкентай элементтің позициясын $O(\log n)$ уақытта таба аламыз.

Мысалы, келтірілген дарақта ең кіші мәні 1 болатын элементті жоғарыдан төменге қарай жүру арқылы табуға болады:



9.4 Қосымша әдіс-тәсілдер

Индекстерді сығымдау

Жиымнан құрылған деректер құрылымының шектеуі элементтердің қатар орналасқан сандар арқылы индекстелуімен байланыстырылады. Үлкен индекстер қажет болған жағдайда қиындық туындайды. Мысалы, егер 10^9 индексі қолданғымыз келсе, сәйкесінше, жиым 10^9 элементтен тұруы шарт. Ол өз кезегінде тым көп жадыны талап етеді.

Алайда туындаған қиындықты сығымдау арқылы шешуге болады, яғни бастапқы индекстерді 1, 2, 3 сияқты индекстерге алмастырамыз. Мұны алгоритмге қажет барлық индекстерді алдын ала білген жағдайда ғана іске асыра аламыз.

Идеясы әр түпкі x индексі $c(x)$ -пен ауыстыруға негізделген. Мұндағы c – индекстерді сығымдайтын функция. Бұл жерде индекстер ретінің өзгермеуін талап еткеніміз жөн, яғни $a < b$ болса, $c(a) < c(b)$ болуы керек. Бұл индекстер сығымдалған болса да сұратымдарды ыңғайлы орындауға мүмкіндік береді.

Мысалы, түпкі индекстер 555, 10^9 және 8 болса, жаңа индекстер төмендегідей болады:

$$\begin{aligned} c(8) &= 1 \\ c(555) &= 2 \\ c(10^9) &= 3 \end{aligned}$$

Аралық жаңартулар

Осыған дейін біз аралық сұратымдарды қолдайтын және бір ғана мәнді өзгертетін деректер құрылымының кодын жаздық. Енді теріс жағдайды, яғни аралықты жаңартатын және бір ғана мәнді қарайтын сұратымдарды, $[a, b]$ аралығындағы барлық элементтерді x -ке арттыратын операцияны ғана қарастырайық.

Бөлімде аталған деректер құрылымын осы жағдайда да қолдануға болады. Ол үшін берілген жиымдағы қатар мәндер арасындағы айырмашылықтарды көрсететін айырма жиымын құрастырамыз. Осылайша, берілген жиым – айырма жиымның префиксті қосындылар жиымы болады. Мысалы, келесі жиымды қарастырайық:

0	1	2	3	4	5	6	7
3	3	1	1	1	5	2	2

Айырма жиымы төмендегідей болады:

0	1	2	3	4	5	6	7
3	0	-2	0	0	4	-3	0

Берілген жиымдағы 6-позициядағы 2 мәні айырма жиымындағы $3 - 2 + 4 - 3 = 2$ қосындысына сәйкес келеді.

Айырма жиымының артықшылығы – берілген жиымдағы аралықты жаңарту үшін өзінің екі ғана элементін өзгеруінде. Мысалы, берілген жиымдағы 1 және 4-позиция аралығын 5-ке арттыру үшін айырма жиымында 1-позициядағы санды 5-ке арттырып, 5-позициядағы санды 5-ке азайту жеткілікті. Нәтижесінде :

0	1	2	3	4	5	6	7
3	5	-2	0	0	-1	-3	0

Қорыта айтқанда $[a, b]$ аралығын x -ке арттыру үшін біз a позициясындағы санды x -ке арттырып, $b + 1$ позициядағы санды x -ке азайтамыз. Демек тек бір мәнді жаңарту және қосынды сұратымдарын өңдеу қажет. Ал ол үшін біз бинарлы индекстелген дарақ немесе кесінділер дарағын қолдана аламыз.

Сәл қиынырақ есепке аралық сұратымдарды және аралық жаңартуларды қолдау жатады. 28-тарауда біз мұның мүмкін екенін көреміз.

10-тарау. Биттік манипуляциялау

Компьютер бағдарламаларындағы барлық деректер биттер арқылы, яғни 0 және 1 сандарымен сақталған. Бұл тарауда бүтін сандардың биттік көрсетілімі мен биттік операциялар қолданысының үлгілерін талқылаймыз. Алгоритмдік бағдарламалауда биттік манипуляциялаудың көп жағдайда пайдалы екендігі байқалады.

10.1 Биттік көрсетілім

Бағдарламалауда n биттік саны n биттен тұратын бинарлы сан ретінде сақталады. Мысалы, C++ int типі – 32 биттік, бұл әр int саны 32 биттен тұратындығын білдіреді.

Төменде 43 int санының биттік көрсетілімі берілген:

[illegible]

Көрсетілімдегі биттер оңнан солға қарай индекстелген. $b_k \cdots b_2 b_1 b_0$ биттік көрсетілімін санға айналдыру үшін

$$b_k 2^k + \dots + b_2 2^2 + b_1 2^1 + b_0 2^0$$

формуласын қолдана аламыз. Мысалы:

$$1 \cdot 2^5 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 43.$$

Санның биттік көрсетілімі таңбалы немесе таңбасыз болып бөлінеді. Әдетте біз таңбалы көрсетілімді қолданамыз. Оның көмегімен оң және теріс таңбалы сандардың екі түрі де көрсетіле береді. Таңбалы n битті айнымалы кез келген -2^{n-1} және $2^{n-1}-1$ аралығындағы бүтін санды қамтиды. Мысалы, C++-тегі `int` типі – таңбалы тип, сондықтан `int` айнымалысы -2^{31} және $2^{31}-1$ аралығындағы кез келген бүтін санды қамти алады.

Таңбалы көрсетілімдегі бірінші бит – санның таңбасы (оң сандар үшін 0, теріс сандар үшін 1), ал қалған $n - 1$ бит – санның мөлшері. Қосымша кодтау деп сандағы биттердің барлығын кері аударып, кейін бірге арттыру арқылы қарама-қарсы санды табу операциясын айтамыз.

–43 int санының биттік көрсетілімін мысал ретінде келтірейік:

1111111111111111111111111010101.

Таңбасыз көрсетілімде тек қана оң сандар қолданыла алады, бірақ сандар мәнінің жоғарғы шегі биік болады. Таңбасыз n битті айнымалы 0 мен $2^n - 1$ аралығындағы кез келген бүтін санды қамти алады. Мысалы, C++-те unsigned int айнымалысы кез келген 0 мен $2^{32} - 1$ аралығындағы бүтін санды қамти алады.

Көрсетілімдер арасындағы байланыс: таңбалы $-x$ саны таңбасыз $2^n - x$ санына тең. Мысалы, төмендегі код таңбалы $x = -43$ санының таңбасыз $y = 2^{32} - 43$ санына тең екендігін көрсетеді:

```
int x = -43;
unsigned int y = x;
cout << x << "\n"; // -43
cout << y << "\n"; // 4294967253
```

Егер сан биттік көрсетілімнің жоғарғы шегінен асса, санның асатолуы орын алады. Таңбалы көрсетілімде $2^{n-1} - 1$ -ден кейінгі сан -2^{n-1} , ал таңбасыз көрсетілімде $2^n - 1$ -ден кейінгі сан 0 . Мысалға келесі кодты қарастырайық:

```
int x = 2147483647
cout << x << "\n"; // 2147483647
x++;
cout << x << "\n"; // -2147483648
```

Бастапқыда x -тің мәні $2^{31} - 1$ -ге тең. Бұл int айнымалысында сақтала алатын ең үлкен мән, сондықтан $2^{31} - 1$ -ден кейінгі мән -2^{31} болады.

10.2 Биттік операциялар

Конъюнкция (Биттік және)

$x \& y$ конъюнкциясы (биттік жәнәсі) x пен y -тің екеуі де бір бит болатын позициялардағы жазылуы бір биттік санды шығарады. Мысалы, $22 \& 26 = 18$, себебі

$$\begin{array}{r} 10110 \quad (22) \\ \& \quad 11010 \quad (26) \\ \hline = \quad 10010 \quad (18) \end{array}$$

Конъюнкцияны қолдану арқылы x санының жұп немесе тақ екенін тексере аламыз. Егер x жұп болса, $x \& 1 = 0$ -ге, ал егер x тақ болса, $x \& 1 = 1$ -ге тең.

Қорыта айтқанда, $x \& (2^k - 1) = 0$ болғанда ғана x 2^k -ге бөлінеді.

Дизъюнкция (Биттік немесе)

$x | y$ дизъюнкциясы (биттік немесесі) x пен y кем дегенде екеуінің бірінде бит болатын позицияларда бір бит жазылған санды құрайды. Мысалы, $22 | 26 = 30$, себебі

Санның жекелеген биттерін ұқсас идеяларды қолдану арқылы анықтауға болады. Мысалы, $x \mid (1 \ll k)$ формуласы x -тің k -битін бір деп белгілейді, $x \& \sim(1 \ll k)$ формуласы x -тің k -битін нөл деп белгілейді, ал $x \wedge (1 \ll k)$ формуласы x -тің k -битін терістейді.

$x \& (x - 1)$ формуласы x -тің соңғы бір битін нөл деп белгілейді, ал $x \& -x$ формуласы соңғысынан басқа барлық бір биттерін нөл деп белгілейді. $x \mid (x - 1)$ формуласы соңғы бір битінен кейінгі барлық биттерді терістейді. Сонымен қатар оң x саны $x \& (x - 1) = 0$ болған жағдайда екінің дәрежесі болатынын ескерген жөн.

Қосымша функциялар

g++ компиляторы биттерді санау үшін келесі функциялармен қамтамасыз етеді:

- `__builtin_clz(x)`: санның басындағы нөлдер саны
- `__builtin_ctz(x)`: санның соңындағы нөлдер саны
- `__builtin_popcount(x)`: сандағы бірлер саны
- `__builtin_parity(x)`: сандағы бірлер санының жұптығы (тақ немесе жұп)

Функциялар төмендегідей жолмен қолданылады:

```
int x = 5328; // 000000000000000000001010011010000
cout << __builtin_clz(x) << "\n"; // 19
cout << __builtin_ctz(x) << "\n"; // 4
cout << __builtin_popcount(x) << "\n"; // 5
cout << __builtin_parity(x) << "\n"; // 1
```

Жоғарыдағы функциялар `int` типіндегі сандарды ғана қолдайды. `long` `long` типіндегі сандармен жұмыс жасау үшін функциялардың атауларына `ll` суффиксін қосу керек.

10.3 Жиындар көрсетілімі

$\{0, 1, 2, \dots, n - 1\}$ жиынының әр ішжиыны n биттік бүтін сан ретінде көрсетіле алады, ондағы әр бірлік қай элемент жиынға жататынын анықтайды. Бұл – жиынды көрсетудің тиімді жолы, себебі әр элемент тек бір бит жадыны талап етеді және жиын операциялары бит операциялары сияқты орындала алады.

Мысалы, `int` 32-биттік тип болғандықтан `int` саны $\{0, 1, 2, \dots, 31\}$ жиынының кез келген ішжиынын көрсете алады. $\{1, 3, 4, 8\}$ жиынының биттік көрсетілімі $2^8 + 2^4 + 2^3 + 2^1 = 282$ санына сәйкес төмендегідей болады:

000000000000000000000000100011010.

Жиын орындалуы

Келесі код $\{0, 1, 2, \dots, 31\}$ ішжиынын қамти алатын `int x` айнымалысын жариялайды. Бұдан соң код 1, 3, 4 және 8 элементтерін жиынға қосып, жиынның өлшемін шығарады:

```
int x = 0;
x |= (1<<1);
x |= (1<<3);
x |= (1<<4);
x |= (1<<8);
cout << __builtin_popcount(x) << "\n"; // 4
```

Ал келесі код жиынға жататын барлық элементтерді шығарады:

```
for (int i = 0; i < 32; i++) {
    if (x & (1<<i)) cout << i << " ";
}
// output: 1 3 4 8
```

Жиын операциялары

Жиын операциялары бит операциялары сияқты төмендегі түрде орындалады:

	жиын синтаксы	бит синтаксы
қиылысу	$a \cap b$	$a \& b$
бірігу	$a \cup b$	$a b$
толықтыру	\bar{a}	$\sim a$
айырма	$a \setminus b$	$a \& (\sim b)$

Мысалы, келесі код алдымен $x = \{1, 3, 4, 8\}$ және $y = \{3, 6, 8, 9\}$ жиындарын құрайды, содан кейін ғана $z = x \cup y = \{1, 3, 4, 6, 8, 9\}$ жиынын құрайды:

```
int x = (1<<1)|(1<<3)|(1<<4)|(1<<8);
int y = (1<<3)|(1<<6)|(1<<8)|(1<<9);
int z = x|y;
cout << __builtin_popcount(z) << "\n"; // 6
```

Ішжиынды өтіп шығу

Келесі код $\{0, 1, \dots, n-1\}$ ішжиындарын өтіп шығады:

```
for (int b = 0; b < (1<<n); b++) {
    // process subset b
}
```

Төмендегі код дәл k элементі бар ішжиындарды өтіп шығады:

```
for (int b = 0; b < (1<<n); b++) {
    if (__builtin_popcount(b) == k) {
        // process subset b
    }
}
```

Келесі код x жиынының ішжиындарын өтіп шығады:

```
int b = 0;
do {
    // process subset b
} while (b=(b-x)&x);
```

10.4 Бит оңтайландырулары

Көптеген алгоритмдер бит операцияларын қолдану арқылы оңтайландырыла алады. Мұндай оңтайландырулар алгоритмнің уақытша күрделілігін өзгертпейді, бірақ олар кодтың нақты орындалу уақытына үлкен әсерін тигізе алады. Бұл бөлікте осындай мысалдарды қарастырамыз.

Хемминг арақашықтығы

$\text{hamming}(a, b)$ Хемминг арақашықтығы – ұзындықтары бірдей a және b жолдарының өзгешеленетін позициялар саны. Мысалы,

$$\text{hamming}(01101, 11001) = 2.$$

Келесі есепті қарастырайық: n бит жолдарының тізімі берілген, әрқайсының ұзындығы k , тізімдегі екі жол арасындағы минималды Хемминг арақашықтығын есептеңіз. Мысалы, $[00111, 01101, 11110]$ жауабы – 2, себебі

- $\text{hamming}(00111, 01101) = 2$,
- $\text{hamming}(00111, 11110) = 3$, және
- $\text{hamming}(01101, 11110) = 3$.

Есепті шығарудың оңай жолы – барлық жолдардың жұптарын қарастырып, олардың Хемминг арақашықтықтарын есептеу, бұл $O(n^2k)$ уақыт алады. Арақашықтықты есептеу үшін келесі функцияны қолдануға болады:

```
int hamming(string a, string b) {
    int d = 0;
    for (int i = 0; i < k; i++) {
        if (a[i] != b[i]) d++;
    }
    return d;
}
```


Егер k кіші болса, біз кодты бит жолдарын бүтін сандар ретінде сақтап, Хемминг арақашықтықтарын бит операцияларын қолданып шығару арқылы оңтайландыра аламыз. Егер $k \leq 32$ болса, көбіне жолдарды жай ғана `int` мәндері ретінде сақтап, келесі функциямен арақашықтықты есептей береміз:

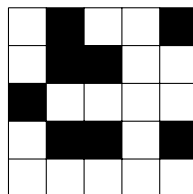
```
int hamming(int a, int b) {
    return __builtin_popcount(a^b);
}
```

Жоғарыдағы функцияда жоюшы дизъюнкция (xor) a мен b жолдары өзгешеленетін позицияларда бір бит болатындай биттік жол құрайды. Кейін биттер саны `__builtin_popcount` функциясы арқылы есептеледі.

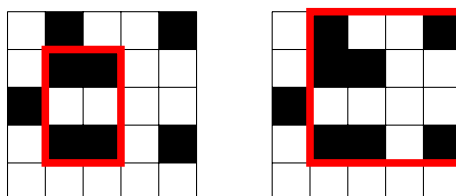
Орындалуларды салыстыру үшін біз ұзындығы 30 болатын 1000 кездейсоқ бит жолдарының тізімін құрдық. Бірінші тәсілді қолданғандағы ізденіс 13.5 секунд алса, бит оңтайландыруынан кейін ол бар болғаны 0.5 секундта орындалды. Осылайша биттік оңтайландырылған код бастапқы кодтан шамамен 30 есе тезірек жұмыс істейді.

Ішкі торларды есептеу

Басқа мысал ретінде келесі есепті қарастырайық: $n \times n$ тор берілген, әр ұяшығы қара (1) немесе ақ (0), барлық бұрыштары қара болатын ішкі торлар санын есептеңіз. Мысалы, мына тор



сондай екі ішкі тор қамтиды:



Есепті $O(n^3)$ уақытында келесі алгоритммен шығаруға болады. Барлық $O(n^2)$ қатарлар жұптарын өтіп шығып, әр (a, b) жұбы үшін екі қатарда да қара ұяшық болатындай бағандар санын $O(n)$ уақытта есептейміз. Келесі код `color[y][x]` – y қатары мен x бағанындағы түсті білдіреді деп қарастырады:

```
int count = 0;
for (int i = 0; i < n; i++) {
    if (color[a][i] == 1 && color[b][i] == 1) count++;
}
```

Кейін бұл бағандардан $\text{count}(\text{count} - 1)/2$ бұрыштары қара ішкі торлар санын есептейміз. Себебі кез келген екеуін таңдап, ішкі тор құруға болады.

Аталған алгоритмді оңтайландыру үшін торды бағандар блоктарына бөлеміз, блоктардың әрқайсысы N қатарлас келген бағаналардан тұрады. Кейін әр қатар N -биттік сандар тізімі ретінде сақталады, әр бит ұяшықтың түсін білдіреді. Енді N бағанды бір сәтте бит операцияларын қолдана отырып, өңдей аламыз. Келесі кодтағы $\text{color}[y][k]$ N түстерден тұратын блокты биттер түрінде ұсынады.

```
int count = 0;
for (int i = 0; i <= n/N; i++) {
    count += __builtin_popcount(color[a][i]&color[b][i]);
}
```

Нәтижесінде пайда болған алгоритм $O(n^3/N)$ уақытта жұмыс істейді.

2500×2500 өлшемді кездейсоқ тор құрастырып, бастапқы және биттік оңтайландырудан кейінгі орындалуларды салыстырдық. Бастапқы код 29.6 секундта орындалса, биттік оңтайландырудан кейінгі орындалу $N = 32$ (int сандарымен) 3.1 секундты, ал $N = 64$ (long long сандарымен) 1.7 секундты қажет етті.

10.5 Динамикалық бағдарламалау

Биттік операциялар күйлері элементтер ішжиындарын қамтитын динамикалық бағдарламалау алгоритмдерінің тиімді және қолайлы орындалуына септігін тигізеді, себебі мұндай күйлер бүтін сандар ретінде сақтала алады. Келесі қарастыратын мысалдарда биттік операциялар мен динамикалық бағдарламалау өзара біріктірілген.

Оңтайлы іріктеу

Алғашқы мысал ретінде келесі есепті қарастыруға болады. Бізге n күндегі k тауарлардың бағалары берілген, әр тауарды бірден сатып алғымыз келеді. Бірақ әр күні бір ғана тауар сатып ала аламыз. Минималды жалпы құн қанша екенін анықтауымыз керек. Үлгі ретінде келесі жағдайды қарастырайық ($k = 3$ және $n = 8$):

	0	1	2	3	4	5	6	7
тауар 0	6	9	5	2	8	9	1	6
тауар 1	8	2	6	2	7	5	7	2
тауар 2	5	3	9	7	3	5	1	4

Бұл жағдайдағы минималды жалпы құн – 5:

	0	1	2	3	4	5	6	7
тауар 0	6	9	5	2	8	9	1	6
тауар 1	8	2	6	2	7	5	7	2
тауар 2	5	3	9	7	3	5	1	4

$\text{price}[x][d]$ x тауарының d күніндегі бағасы деп белгілейік. Мысалы, жоғарыдағы жағдайда $\text{price}[2][3] = 7$. $\text{total}(S, d)$ S тауарлар ішжиынын d күні сатып алғандағы минималды жалпы құн деп белгілейік. Бұл функцияны қолдану арқылы есеп шешімі $\text{total}(\{0 \dots k-1\}, n-1)$ болмақ.

Алғашында $\text{total}(\emptyset, d) = 0$, себебі бос жиынды сатып алу бағасы жоқ, онымен қоса $\text{total}(\{x\}, 0) = \text{price}[x][0]$. Себебі бірінші күні тауарды сатып алудың жалғыз жолы бар. Кейін төмендегі рекурсияны қолдана аламыз:

$$\text{total}(S, d) = \min(\text{total}(S, d-1), \min_{x \in S}(\text{total}(S \setminus x, d-1) + \text{price}[x][d]))$$

Бұл d күні S -ке жататын қандай да бір x тауарын сатып алатынымызды не сатып алмайтынымызды білдіреді. Екінші жағдайда x -ті S -тен өшіріп, x бағасын жалпы құнға қосамыз.

Келесі қадам – динамикалық бағдарламалау көмегімен функция мәндерін есептеу. Функция мәндерін сақтау үшін келесі жиымды құрастырамыз:

```
int total[1<<K][N];
```

Мұндағы K мен N жарамды көлемде алынған, ал жиымдағы бірінші шама – ішжиынның биттік көрсетілімі.

Алдымен $d = 0$ болған жағдайларды өңдейміз:

```
for (int x = 0; x < k; x++) {
    total[1<<x][0] = price[x][0];
}
```

Кейін рекурсия төмендегідей кодқа ауысады:

```
for (int d = 1; d < n; d++) {
    for (int s = 0; s < (1<<k); s++) {
        total[s][d] = total[s][d-1];
        for (int x = 0; x < k; x++) {
            if (s & (1<<x)) {
                total[s][d] = min(total[s][d],
                                   total[s^(1<<x)][d-1] + price[x][d]);
            }
        }
    }
}
```

Алгоритмнің уақытша күрделілігі – $O(n2^k k)$.

Алмастырулардан ішжиындарға

Динамикалық бағдарламалауды қолданып, алмастырулар итерациясын ішжиындар итерациясына алмастыра аламыз¹. Мұндағы артықшылық алмастырулар саны – $n!$ ішжиындар саны – 2^n -нен анағұрлым үлкен болуында жатыр. Мысалы, егер $n = 20$ болса, $n! \approx 2.4 \cdot 10^{18}$, ал $2^n \approx 10^6$. Сондықтан кейбір n мәндері үшін алмастырулар емес, ішжиындарды өтіп шығу тиімдірек болмақ.

Мысал ретінде келесі есепті қарастырайық: Максималды жүк салмағы x болатын лифт пен салмақтары белгілі n адам бар. Олар төменгі қабаттан жоғарғы қабатқа көтерілгісі келеді. Егер адамдардың міну ретін оңтайландырса, ең азы қанша сапар жасалуы қажет?

Мысалы, $x = 10$, $n = 5$ деп алайық. Ал салмақтары төмендегідей болсын:

адам	салмақ
0	2
1	3
2	3
3	5
4	6

Бұл жағдайда минималды сапарлар саны – 2. Оңтайлы реттілік – $\{0, 2, 3, 1, 4\}$, ол адамдарды екі сапарға бөледі: бірінші $\{0, 2, 3\}$ (жалпы салмағы 10), ал кейін $\{1, 4\}$ (жалпы салмағы 9).

Есеп $O(n!n)$ уақыт ішінде барлық мүмкін болатын n адам алмастыруларын тексеру арқылы $O(n!n)$ уақытында оңай шешіледі. Бірақ $O(2^n n)$ уақытында орындалатын тиімдірек динамикалық бағдарламалау алгоритмін қолдана аламыз. Идеясы – әр адамдар ішжиыны үшін екі мәнді, атап айтқанда қажетті минималды сапар саны мен соңғы сапардағы адамдар тобының минималды салмағын сақтаймыз.

$\text{weight}[p]$ p адамның салмағын белгілейді десек, екі функцияны көрсетеміз. Олар: $\text{rides}(S)$ – S ішжиыны үшін минималды сапар саны, $\text{last}(S)$ – соңғы сапардағы минималды салмақ. Мысалы, жоғарыдағы жағдайда

$$\text{rides}(\{1, 3, 4\}) = 2 \quad \text{және} \quad \text{last}(\{1, 3, 4\}) = 5,$$

себебі оңтайлы сапарлар – $\{1, 4\}$ және $\{3\}$, сонымен қоса екінші сапардың салмағы – 5. Әрине, біздің ақырғы нысанымыз $\text{rides}(\{0 \dots n - 1\})$ мәнін есептеу.

Функциялардың мәндерін рекурсивті түрде есептеп, кейін динамикалық бағдарламалауды қолдануымызға болады. Мұндағы идея S -ке тиесілі барлық адамдарды өтіп шығып, лифтке ең соңғы болып кіретін p адамды оңтайлы таңдауға негізделеді. Әрбір осындай таңдау адамдардың ішжиынына арналған ішесептерге әкеледі. Егер $\text{last}(S \setminus p) + \text{weight}[p] \leq x$ болса, біз p -ді соңғы сапарға қосамыз. Әйтпесе бастапқыда тек қана p болатын жаңа сапарды кейінге сақтауға тура келеді.

Динамикалық бағдарламалауды орындау мақсатында әр S ішжиыны үшін $(\text{rides}(S), \text{last}(S))$ жұбын сақтайтын келесі жиымды жариялаймыз:

¹Бұл тәсілді 1962 жылы М.Хелд пен Р.М.Карп таныстырды[29].

```
pair<int,int> best[1<<N];
```

Бос топтың мәнін төмендегідей үлгіде меншіктейміз:

```
best[0] = {1,0};
```

Кейін жиымды келесідей тәртіппен толтырамыз:

```
for (int s = 1; s < (1<<n); s++) {  
    // initial value: n+1 rides are needed  
    best[s] = {n+1,0};  
    for (int p = 0; p < n; p++) {  
        if (s&(1<<p)) {  
            auto option = best[s^(1<<p)];  
            if (option.second+weight[p] <= x) {  
                // add p to an existing ride  
                option.second += weight[p];  
            } else {  
                // reserve a new ride for p  
                option.first++;  
                option.second = weight[p];  
            }  
            best[s] = min(best[s], option);  
        }  
    }  
}
```

Жоғарыда келтірілген циклдің $S_1 \subset S_2$ болатын кез келген S_1 және S_2 ішжиындары үшін S_1 S_2 -ден бірінші өңделетініне кепілдік беретініне назар аударғанымыз жөн. Осылайша динамикалық бағдарламалау мәндері дұрыс ретпен есептеледі.

Ішжиындар мәндерінің қосындысы

Бөлімнің соңғы есебіне келейік. $X = \{0 \dots n-1\}$ жиыны мен әр $S \subset X$ ішжиынына сәйкес $\text{value}[S]$ бүтін мәні берілген. Әр S -ке

$$\text{sum}(S) = \sum_{A \subset S} \text{value}[A],$$

қосындысын есептеу, яғни S ішжиындарының мәндер қосындысын есептеу тапсырма ретінде беріліп тұр.

Мысалы, $n = 3$ деп есептейік және келесі мәндер берілген делік:

- $\text{value}[\emptyset] = 3$
- $\text{value}[\{0\}] = 1$
- $\text{value}[\{1\}] = 4$
- $\text{value}[\{0,1\}] = 5$
- $\text{value}[\{2\}] = 5$
- $\text{value}[\{0,2\}] = 1$

- $\text{value}[\{1, 2\}] = 3$

- $\text{value}[\{0, 1, 2\}] = 3$

Ендеше бұл жағдайда:

$$\begin{aligned}\text{sum}(\{0, 2\}) &= \text{value}[\emptyset] + \text{value}[\{0\}] + \text{value}[\{2\}] + \text{value}[\{0, 2\}] \\ &= 3 + 1 + 5 + 1 = 10.\end{aligned}$$

Жалпы 2^n ішжиын болғандықтан, біз барлық ішжиындар жұптарынан өту арқылы есепті $O(2^{2n})$ уақытта шығара аламыз. Дегенмен динамикалық бағдарламалау арқылы бұл есепті $O(2^n n)$ уақытта шығаруға болады. Мұндағы идея S -тен өшірілуі мүмкін элементтері шектеулі болатын ішжиымдардың мәндер қосындысына баса назар аударуға бағытталады.

$\text{partial}(S, k)$ тек $0 \dots k$ элементтерін өшіруге болатын шектеуі бар S ішжиымдарының мәндер қосындысы деп белгілейік. Мысалы,

$$\text{partial}(\{0, 2\}, 1) = \text{value}[\{2\}] + \text{value}[\{0, 2\}],$$

себебі біз тек $0 \dots 1$ элементтерін өшіре аламыз. sum мәндерін partial мәндері арқылы есептеуімізге болады, себебі

$$\text{sum}(S) = \text{partial}(S, n - 1).$$

Функцияның негізгі жағдайлары –

$$\text{partial}(S, -1) = \text{value}[S],$$

өйткені бұл жағдайда S -тен ешқандай элемент өшірілмейді. Мұндай жағдайда біз келесі рекуренттік формулаларды қолдана аламыз:

$$\text{partial}(S, k) = \begin{cases} \text{partial}(S, k - 1) & k \notin S \\ \text{partial}(S, k - 1) + \text{partial}(S \setminus \{k\}, k - 1) & k \in S \end{cases}$$

Бұл жерде k элементіне назар аударғанымыз жөн. Егер $k \in S$ болса, бізде екі таңдау пайда болады. Біз k -ны S -те қалдырамыз немесе оны S -тен өшіреміз.

Қосындыларды есептеп шығаруды жүзеге асырудың ерекше ақылды тәсілі бар. Алдымен төмендегі жаңа жиымды жариялаймыз.

```
int sum[1<<N];
```

Бұл жиым әр ішжиынның қосындысын қамтиды. Жиым төмендегідей түрде инициалданған:

```
for (int s = 0; s < (1<<n); s++) {
    sum[s] = value[s];
}
```

Кейін жиымды келісі ретпен толтырамыз:

```
for (int k = 0; k < n; k++) {
    for (int s = 0; s < (1<<n); s++) {
        if (s & (1<<k)) sum[s] += sum[s ^ (1<<k)];
    }
}
```

Бұл код $k = 0 \dots n - 1$ үшін $\text{partial}(S, k)$ мәндерін sum жиымына есептейді. $\text{partial}(S, k)$ әрдайым $\text{partial}(S, k - 1)$ мәніне негізделгені үшін біз sum жиымын қайтадан қолдана аламыз. Ал ол өз кезегінде ыңғайлы және оңтайлы код жазуға мүмкіндік береді.

II БӨЛІМ.

Графтағы алгоритмдер

11-тарау. Граф негіздері

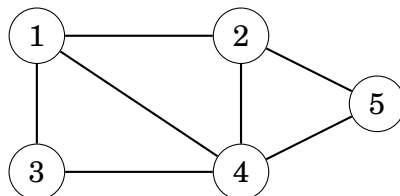
Бағдарламалаудың көптеген есептерін оларды граф түрінде моделдеу және графтағы сәйкес алгоритмдерді қолдану арқылы шығара аламыз. Графтың әдеттегі мысалы ретінде еліміздегі жолдар мен қалалар желісін алуға болады. Бірақ кей есепте граф жасырын түрде кездеседі және оны табу қиынға соғады.

Кітабымыздың осы бөлімінде біз граф алгоритмдерін талқылаймыз. Әсіресе спорттық бағдарламалауда пайдалы болатын алгоритмдерге баса назар аударамыз. Бұл тарауда графқа қатысты ұғымдарды қарастырамыз және графты көрсетудің түрлі жолдарын үйренеміз.

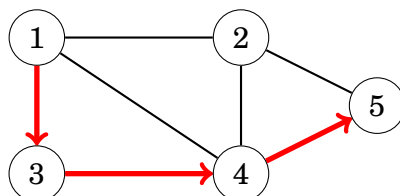
11.1 Граф терминологиясы

Граф төбелерден және қырлардан тұрады. Кітапта n деп графтағы төбелердің, ал m деп қырлардың саны белгіленген. Төбелер $1, 2, \dots, n$ деп нөмірленеді.

Мысалға, төменде 5 төбеден және 7 қырдан тұратын граф бейнеленген:



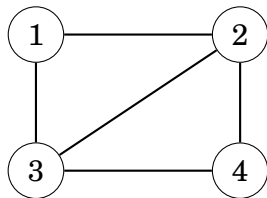
Жол – графтағы a төбесінен b төбесіне дейінгі қырлар тізбегі. Жолдағы қырлардың саны ұзындықты құрайды. Мысалы, жоғарыдағы графта $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ жолы бар. Жолдың ұзындығы – 3, ол 1-төбеден басталып, 5-төбеге дейін жетеді:



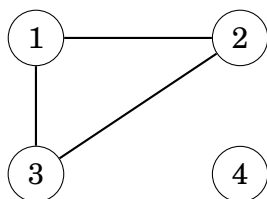
Жолдың бірінші және соңғы төбелері бірдей болса, оны цикл дейміз. Мысалға, жоғарыдағы графта $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$ циклы бар. Жолда бір төбе 1 реттен артық қайталанбаса, ол жай жол болып саналады.

Байланыстылық

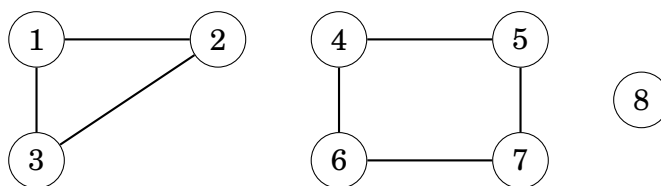
Кез келген екі төбенің арасында жол болса, граф өзара байланысты болады. Мысалы, төмендегі граф өзара байланысты:



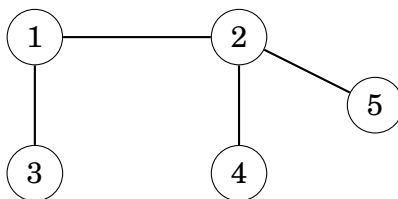
Ал төмендегі граф өзара байланысты емес. Себебі 4-төбеден басқа төбелерге баратын жол жоқ.



Графтың өзара байланысты бөліктері компонент деп аталады. Мысалы, төмендегі графта үш компонент бар: {1, 2, 3}, {4, 5, 6, 7} және {8}.

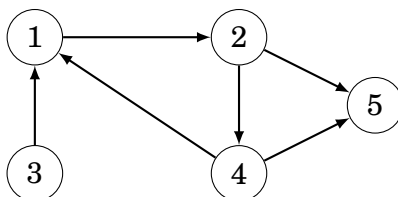


Дарақ деп n төбеден және $n - 1$ қырдан тұратын өзара байланысты графты айтамыз. Бұл графта кез-келген екі төбе арасында бірегей жол болады. Төмендегі графты дарақ деп санауымызға болады:



Қыр бағыты

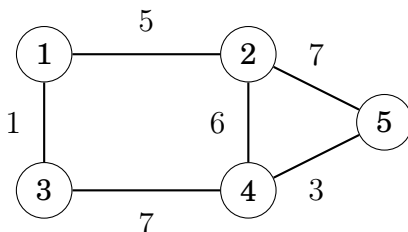
Егер қырлар бір ғана төбеге бағытталса, ол бағытталған граф болып саналады. Бағытталған граф үлгісі:



Жоғарыдағы графта $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$ жол бар. Бірақ, бұл графта 5-төбеден 3-төбеге жол жоқ.

Қыр салмағы

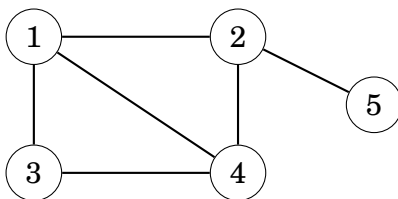
Салмақталған графтың әр қырында салмағы белгіленеді. Салмақ көбіне қырдың ұзындығы ретінде көрсетіледі. Төмендегі граф салмақталған графқа жатады:



Салмақталған графта жолдың ұзындығы жолдағы қырлардың салмақтарының қосындысына тең болады. Жоғарыдағы графта $1 \rightarrow 2 \rightarrow 5$ жолының ұзындығы 12-ге тең. Ал $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ жолының ұзындығы 11-ге тең. Осы жол 1-төбеден 5-төбеге дейін ең қысқа жол болып тұр.

Көршілер және дәреже

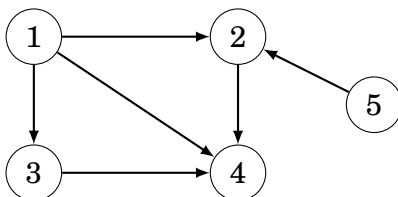
Егер екі төбе арасында қыр болса, олар өзара көршілер болады. Төбенің дәрежесі төбедегі көршілестердің санына тең келеді. Мысалы, үлгіде келтірілген графтың 2-төбесінде 1, 4 және 5 көрші төбелері бар. Сондықтан оның дәрежесі 3-ке тең болмақ.



Графта m қыр болса, графтағы дәрежелердің сомасы әрқашан $2m$ -ге тең болады. Себебі әр қыр екі көршілес тұрған төбелердің дәрежесін 1-ге көбейтеді. Демек дәрежелердің қосындысы әрдайым жұп сан болады.

Егер барлық төбелердің дәрежелері тең болса, ондай графты біртекті граф дейміз. Ал егер графтағы барлық төбелердің дәрежелері $(n - 1)$ -ге тең болса, ондай граф толық граф деп аталады. Яғни графтағы әр төбе өзінен басқа барлық төбелермен байланысты болады.

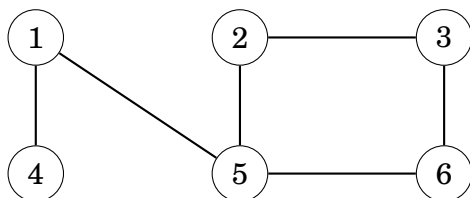
Бағытталған графтағы кірістің жарты дәрежесі деп төбеге бағытталған қырлардың санын белгілейміз. Ал шығыстың жарты дәрежесі деп төбеден бағыт алған қырлардың санын белгілейміз. Мысалға, төмендегі графтың 2-төбесіндегі кірісінің жарты дәрежесі 2-ге, ал шығысының жарты дәреже 1-ге тең.



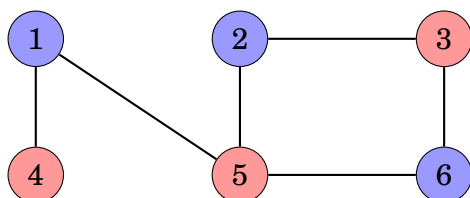
Бояулы

Бояулы графта әр төбеге бір бояу беріледі. Оның екі көршілес төбелері бірдей түске боялмайды.

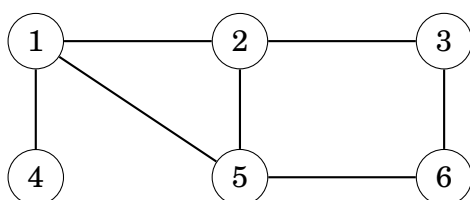
Егер граф екі түске боялатын болса, ол екі ұялы граф деп аталады. Граф екі ұялы болу үшін оның ішінде ұзындығы тақ сан болатын цикл болмауы тиіс. Мысалы, төменде екі ұялы граф берілген:



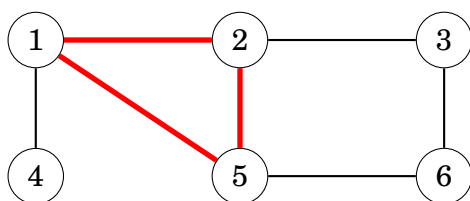
Себебі, оны төмендегідей етіп бояуға болады:



Ал келесі граф екі ұялы емес:

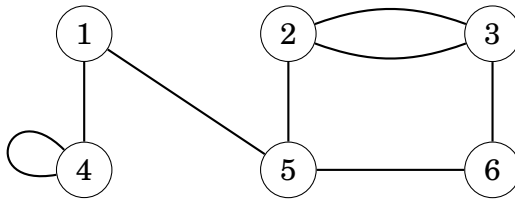


Өйткені 1, 2 және 5 төбелері құрайтын ұзындығы 3-ке тең циклды екі түске бояу мүмкін емес:



Қарапайым граф

Егер графта бір төбеден шығып дәл сол төбеге қайта бағытталатын қыр (ілмек) болмаса және екі төбе арасында бір ғана қыр болса, ондай графты қарапайым граф деп атаймыз. Көбінесе, біз графты елестеткенде оны жай граф түрінде қарастырамыз. Мысалы, төмендегі граф қарапайым графқа жатпайды.



11.2 Графты көрсету жолдары

Графты көрсетудің бірнеше жолы бар. Деректер құрылымын таңдау графтың өлшеміне және алгоритмнің өңдеу тәсіліне байланысты болып келеді. Біз графты көрсетудің 3 жалпы жолын қарастырамыз.

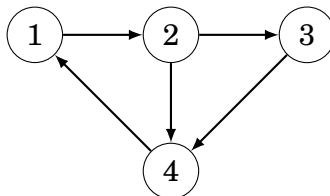
Сыбайластық тізім көрінісі

Бұл көріністе әр графтағы x төбесінің сыбайластық тізімі белгіленеді. Сыбайластық тізім – x төбесінен шыққан қырлар байланысқан төбелердің тізімі. Ол қолданыста ең жиі кездесетін көрсетілім түріне жатады. Оған қоса, көптеген алгоритмдер осы көрініспен тиімді жұмыс жасайды.

Сыбайластық тізімін сақтау үшін векторлар жиымы жарияланады:

```
vector<int> adj[N];
```

Барлық сыбайластық тізімдер сиятындай бір тұрақты сан N таңдалады. Мысалы, төмендегі графты



келесі ретпен сақтауға болады:

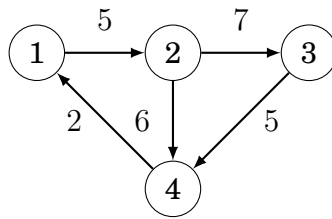
```
adj[1].push_back(2);
adj[2].push_back(3);
adj[2].push_back(4);
adj[3].push_back(4);
adj[4].push_back(1);
```

Егер граф бағытталмаған болса, оны да дәл солай сақтауға болады. Бірақ әрбір қырды екі бағыттан сақтайды.

Ал салмақталған графтың құрылымы аздап өзгереді:

```
vector<pair<int,int>> adj[N];
```

Бұл кезде a төбесінің сыбайластық тізімі (b, w) жұбын сақтайды, яғни бұл жерде a төбесінен басталып b төбесінде аяқталатын салмағы w -ге тең қыр бар. Үлгідегі графты



төмендегідей етіп сақтауға болады:

```

adj[1].push_back({2,5});
adj[2].push_back({3,7});
adj[2].push_back({4,6});
adj[3].push_back({4,5});
adj[4].push_back({1,2});

```

Сыбайластық тізімінің артықшылығы – берілген төбеден қыр арқылы бара алатын басқа төбелерді тиімді әрі тез таба алуымызда. Мысалы, келесі циклде біз s -төбесінен бағыт ала алатын барлық төбелерді өтіп шығамыз.

```

for (auto u : adj[s]) {
    // process node u
}

```

Сыбайластық матрица көрінісі

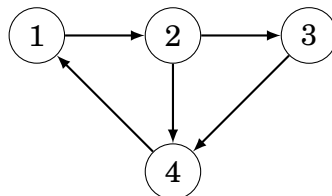
Сыбайластық матрицасы – графтың қай қырларды қамтитынын көрсететін екі өлшемді матрица. Біз сыбайластық матрицасы арқылы екі төбе арасында қырдың бар немесе жоқ екенін оңай анықтай аламыз. Матрицаны жиым ретінде сақтауға болады:

```

int adj[N][N];

```

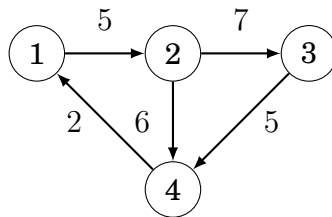
Бұл жерде $adj[a][b]$ графта a төбесінен басталып b төбесінде аяқталатын қырдың бар немесе жоқ екенін көрсетеді. Егер графта қыр бар болса, $adj[a][b] = 1$, әйтпесе $adj[a][b] = 0$. Мысалға, төмендегі графты



осылай көрсетуге болады:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

Егер граф салмақталған болса, сыбайластық матрицасының көрінісін толықтыруға болады. Матрица тек **1** және **0** сандарын сақтамай, қырдағы салмақты да сақтайды. Сәйкесінше, келесі графты



осы нұсқада көрсетуге болады:

	1	2	3	4
1	0	5	0	0
2	0	0	7	6
3	0	0	0	5
4	2	0	0	0

Бұл көріністің кемшілігі –матрицада n^2 элемент сақтай алуымызда және оның көп бөлігі **0**-ге тең болуында. Сондықтан егер граф үлкен болса, қарастырылған көрініс жарамсыз болмақ.

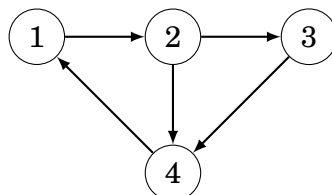
Қырлар тізімі көрінісі

Қырлар тізімінде графтағы барлық қырлар белгілі бір ретпен сақталады. Бұл көрініс алгоритм графтағы барлық қырларды өтіп шығатын болса және белгілі бір төбеден басталатын қырлардың тізімін қажет етпесе тиімді болады.

Қырлар тізімі векторда сақтала алады.

```
vector<pair<int,int>> edges;
```

Бұл жерде (a, b) жұбы a -төбесінен b -төбесіне дейін қырдың бар немесе жоқ екенін көрсетеді. Сондықтан бұл графты



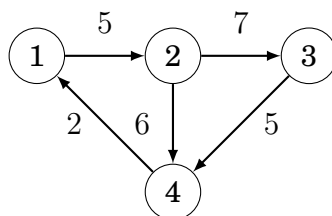
төмендегідей етіп көрсетуге болады:

```
edges.push_back({1,2});  
edges.push_back({2,3});  
edges.push_back({2,4});  
edges.push_back({3,4});  
edges.push_back({4,1});
```

Граф салмақталған болса, құрылымы осылай өзгере алады:

```
vector<tuple<int,int,int>> edges;
```

Вектордағы әр элемент (a, b, w) құрылымында болады. Бұл құрылымда әр қыр a төбесінен басталып b төбесінде аяқталады және салмағы w -ге тең болады. Мысалы, келесі графты



осы түрде көрсете аламыз ¹:

```
edges.push_back({1,2,5});  
edges.push_back({2,3,7});  
edges.push_back({2,4,6});  
edges.push_back({3,4,5});  
edges.push_back({4,1,2});
```

¹Кей ескі компиляторларда өрнек жақша орнына `make_tuple` функциясын қолдану керек. (Мысалы, `make_tuple(1,2,5)` орнына `{1,2,5}`).

12-тарау. Графты аралау

Бұл тарау екі негізгі граф алгоритмдері туралы болмақ. Біріншісі – тереңдігі бойынша іздеу, екіншісі – ені бойынша іздеу. Екі алгоритмде де алғашында графтағы бастапқы төбе беріледі және олар бастапқы төбеден жете алатын барлық төбелерге барып шығады. Алгоритмдердің айырмашылығы төбелерге келу реттілігіне байланысты туындайды.

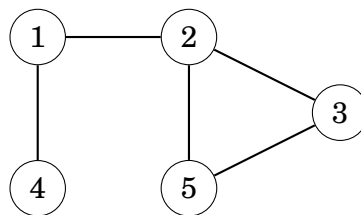
12.1 Тереңдігі бойынша іздеу

Тереңдігі бойынша іздеу (DFS) – қарапайым графты өтіп шығу әдісі. Алгоритм алғашқы төбеден бастап, графтағы қырларды қолдана отырып, басқа төбелерге жетеді.

Тереңдігі бойынша іздеу әрқашан бастапқы төбеден басталады және басқа бармаған төбелерге апаратын қыр болса, сол төбелерге тікелей жалғасады. Ал егер бармаған басқа төбеге апаратын жол болмаса, кейін қайтып, артынша бармаған басқа төбелерге қарай өтеді. Алгоритм әрбір төбеге тек бір рет қана бару үшін жеткен төбелерін сақтап отырады.

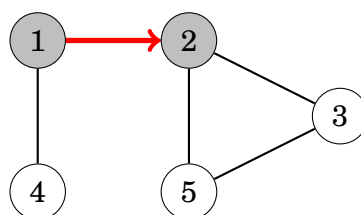
Мысал

Төменде берілген графты тереңдігі бойынша өтіп көрейік:

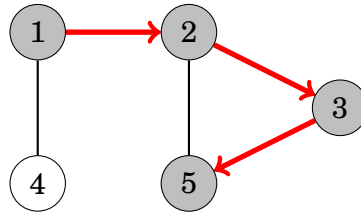


Біз ізденісті кез келген төбеден бастай аламыз. Келтіріліп отырған мысалда біз 1-төбеден бастаймыз.

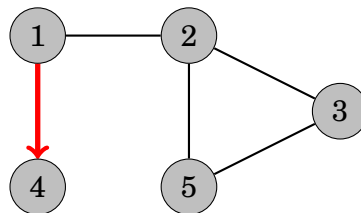
Бірінші, ізденіс 1-төбеден 2-төбеге жетеді:



Содан кейін 3 және 5-төбелерге барады:



5-төбенің көршілері – 2 және 3-төбелер. Бірақ біз бұл төбелерде әлдеқшан болғандықтан оларға бара алмай, артқы төбеге қайтамыз. Сондай-ақ 3, 2-төбелердің көршілеріне де бұған дейін барып қойғандықтан, біз 1-төбеге қайтамыз. Осылайша 4-төбеге де жетеміз:



Бұдан кейін ізденіс бітеді. Себебі біз барлық төбелерде болдық.

Төбелердің санын n , ал қырлардың санын m деп белгілесек, бұл алгоритмнің уақытша күрделілігі $O(n + m)$ -ге тең болады. Себебі алгоритм әр төбені және әр қырды бір рет өтіп шығады.

Кодтың жазылуы

Тереңдігі бойынша іздеу алгоритмін рекурсия арқылы жазу ең оңай әдіске жатады. Төмендегі dfs функциясында бастапқы төбені қабылдаймыз. Сол төбеден ізденісті бастаймыз. Бұл ізденіс функциясында біз сыбайластық тізімімен жұмыс жасаймыз:

```
vector<int> adj[N];
```

Сондай-ақ барған төбелерді жиымда сақтаймыз:

```
bool visited [N];
```

Алғашында жиымдағы әр элемент false болады. Ал s төбесіне келгенде $visited[s]$ элементі true болып өзгереді. Бұл функцияны төмендегідей етіп жазуға болады:

```
void dfs(int s) {  
    if (visited[s]) return;  
    visited[s] = true;  
    // process node s  
    for (auto u: adj[s]) {  
        dfs(u);  
    }  
}
```

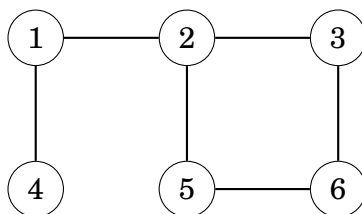
12.2 Ені бойынша іздеу

Ені бойынша іздеуде (BFS) графтың төбелеріне бастапқы төбеден арақашықтың өсу ретіне қарай барылады. Алгоритм арқылы бастапқы төбе мен графтағы кез келген төбе арасындағы арақашықтықты есептеу оңайға соғады. Бірақ кодтың жазылуы тереңдігі бойынша ізденістен қиынырақ болады.

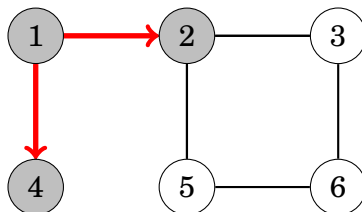
Ізденіс барысында төбелерді кезең-кезеңімен өтіп шығамыз. Алдымен бастапқы орнымыздан 1 қыр арақашықтығындағы барлық төбелерден өтеміз, кейін арақашықтығы 2 болатын төбелерге барамыз. Бұл үдеріс барлық төбелерді өтіп шыққанға дейін жалғаса береді.

Мысал

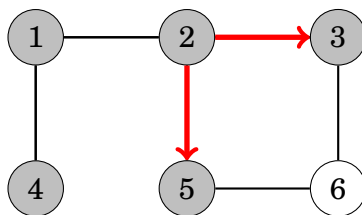
Мысалға төмендегі графты қарастырайық:



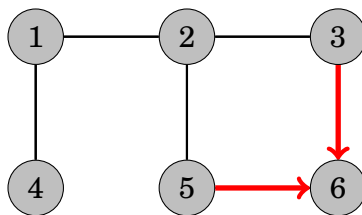
Ені бойынша ізденісті 1-төбеден бастаймыз. Алдымен біз 1-төбемен бір қыр арқылы жалғасып тұрған төбелерге барамыз:



Кейін 3 және 5-төбелерге жетеміз:



Ал ең соңында біз 6-төбеге жетеміз:



Енді біз бастапқы төбеден графтағы барлық төбелерге дейінгі арақашықтықты есептеп шығамыз. Арақашықтықтар төмендегі кестеде көрсетіледі:

төбе	арақашықтық
1	0
2	1
3	2
4	1
5	2
6	3

Тереңдігі бойынша ізденістегідей ені бойынша ізденістің уақытша күрделілігі $O(n + m)$. Мұндағы n деп төбелердің санын, ал m деп қырлардың санын белгілейміз.

Кодтың жазылуы

Ені бойынша ізденіс алгоритмін жазу тереңдік бойынша ізденіс алгоритмін жазудан қиынырақ. Себебі алгоритм графтың әр жеріндегі төбелер арқылы ретсіз жүреді. Көбіне бұл алгоритмді төбелердің тізімін сақтайтын кезек деректер құрылымы арқылы жазады. Әр қадам сайын кезектегі бірінші төбеге барамыз.

Осы ізденіс функциясында сыбайластық тізімін пайдаланамыз, сонымен қатар келесі деректер құрылымдарымен жұмыс жасаймыз:

```
queue<int> q;
bool visited[N];
int distance[N];
```

Кодтағы q – кезек деректер құрылымы, ол төбелердің тізімін сақтайды. q кезегі қашықтықтың өсу ретімен өңделетін төбелерді қамтиды. Кезектің басында алғашқы төбеге ең жақын, ал соңына қарай ең алыс төбелер орналасқан. Жаңа бір төбені тізімге қосақан сәтте әрқашан кезектің соңынан қосамыз, ал келесі баратын төбені тізімнің басынан аламыз. `visited` жиымы төбеде бұрын болғанымызды немесе болмағанымызды білдіреді. Ал `distance` жиымы бастапқы төбеден басқа төбелерге дейінгі арақашықтықты есептейді.

Кодты үлгідегідей етіп жазуға болады (x бастапқы төбе деп алсақ):

```
visited[x] = true;
distance[x] = 0;
q.push(x);
while (!q.empty()) {
    int s = q.front(); q.pop();
    // process node s
    for (auto u : adj[s]) {
        if (visited[u]) continue;
        visited[u] = true;
        distance[u] = distance[s] + 1;
```

```

    q.push(u);
  }
}

```

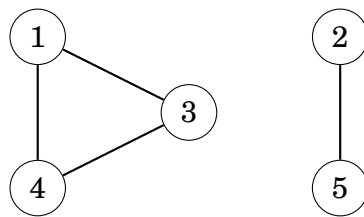
12.3 Қолданыс аясы

Атап өтілген алгоритмдер арқылы графтың көптеген қасиеттері туралы ақпарат алуға болады. Көп жағдайда екі ізденісті де пайдалана аламыз. Бірақ жазылу жолы оңайырақ болғандықтан, тереңдігі бойынша ізденіс жиі жасалады. Төмендегі қолданыстарда бағытталмаған граф келтірілген деп болжанады.

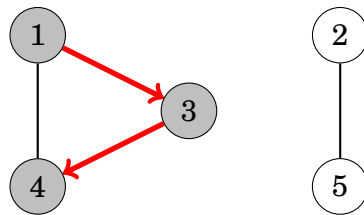
Графтың байланыстылығын тексеру

Егер кез келген екі төбе арасында жол болса, ол графтың байланыстылығын білдіреді. Демек біз графтың байланыстылығын кез келген бір төбені таңдап, одан басқа төбелерге жете алатындығымызды тексеру арқылы біле аламыз.

Мысалы, төмендегі графта



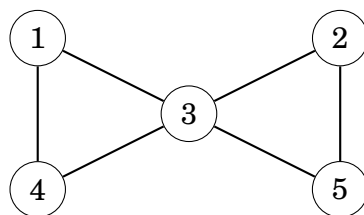
тереңдігі бойынша ізденісті 1-төбеден бастасақ, келесі төбелерге бара аламыз:



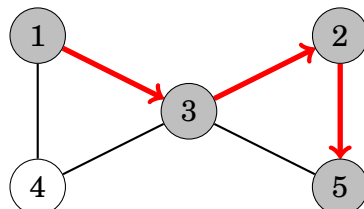
Жоғарыдағы мысалдан 1-ші төбеден басталған тереңдігі бойынша іздеу барлық төбелерге бара алмайтынын көреміз. Сондықтан графты өзара байланыссыз деп қорытынды жасауға болады. Дәл осылай графтың барлық компоненттерін таба аламыз. Ол үшін барлық төбелерді өтіп шығып, егер ағымдағы төбе бұған дейін табылған байланыстылық компонентінің бір де біреуіне жатпаса, тереңдігі бойынша жаңа іздеуді бастаймыз.

Циклдарды табу

Егер біз графтың тереңдігі бойынша ізденіс жасап келе жатып, сол сәтте тұрақтаған ағымдағы төбенің көршісі осыған дейін өтілгенін байқасақ, граф циклды қамтыды деп санаймыз. Мысалға, келесі граф екі циклды қамтиды:



Төменде циклдардың біреуінің табу жолы көрсетілген:



Мұндағы 2-төбеден 5-төбеге жеткен кезде 3-төбеге әлдеқашан барылғанын байқаймыз. Сондықтан бұл граф 3-төбеден өтетін циклды қамтиды. Мысалға, $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

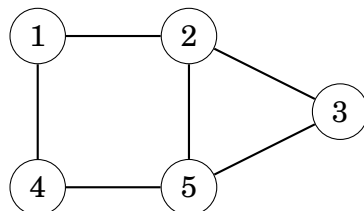
Графта цикл бар екенін басқа жолмен де тексеруге болады. Ол үшін біз жай ғана әр компоненттегі төбелер мен қырлардың санын есептейміз. Егер компонентте c төбе болса, әрі цикл болмаса, бұл компонентте (граф дарақ болуы үшін) дәл $c - 1$ қыр болуы керек. Егер де, c немесе одан да көп қыр болса, компонентте цикл бар екені сөзсіз анық.

Екі ұялыққа тексеріс

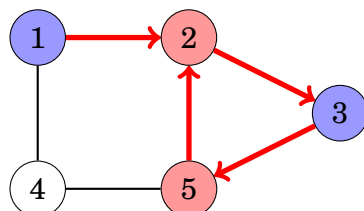
Егер графтағы төбелер екі көршісінің түстері бірдей болмайтындай етіп, екі түспен боялатын болса, граф екі ұялы болады. Графты айтылған алгоритмдер арқылы екі ұялылыққа оңай тексере аламыз.

Тексеру үшін бастапқы төбені көкке бояйық. Кейін оның көршілерін қызылға, ал қызылға боялған төбелердің көршілерін көкке, т.с.с. жалғастыра отырып бояймыз. Бояу барысында екі көршілес төбеде бірдей түс болса, граф екі ұялы бола алмайды. Ал графты толықтай бояу мүмкін болса, граф екі ұялы саналады.

Мысалға, төмендегі граф екі ұялы бола алмайды:



Себебі, 1-төбеден басталған тереңдігі бойынша ізденіс былай жұмыс жасайды:



Көріп отырғанымыздай үлгідегі граф екі ұялы емес. Себебі 2 және 5-төбелер бірдей қызыл түске боялған, оған қоса көршілес келген.

Аталған алгоритм әрқашан да дұрыс жұмыс жасайды. Себебі графты екі түске бояған кезде бастапқы төбе компоненттегі барлық басқа төбелердің түстерін анықтайды. Алғашқы төбенің қай түске, яғни қызылға немесе көкке боялғаны маңызды емес.

Басқа жағдайларда графты k түске бояу мүмкіндігін тексеру қиынға соғатынын ескергеніміз жөн. Тіпті $k = 3$ деп алсақ та, әзірге тиімді алгоритм жоқ екенін айта кеткіміз келеді. Аталған есеп NP-қиын есептердің қатарына жатады.

13-тарау. Ең қысқа жолдар

Графтағы екі төбенің арасындағы ең қысқа жолды табу – көптеген практикалық қолданысы бар маңызды есептердің бірі. Жол картасы мен барлық телімдердің ұзындығы белгілі болған жағдайда екі қала арасындағы төте маршрутты табу мәселесін оның айқын мысалы ретінде келтіруге болады. Салмақтанбаған графта жолдың ұзындығы жолдағы қырлардың санына тең болады. Осындай жағдайда ең қысқа жолды табу үшін жай ғана ені бойынша ізденіс алгоритмін қолдансақ болады. Алайда бұл тарауда біз салмақтанған графтарды қарастырамыз. Олармен жұмыс барысында қиынырақ алгоритмдерді қолдануға мәжбүр боламыз.

13.1 Беллман-Форд алгоритмі

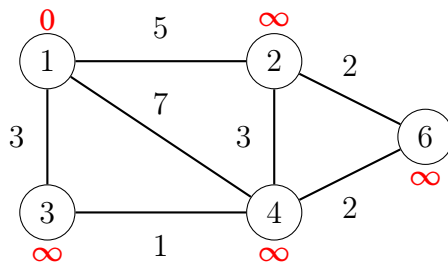
Беллман-Форд алгоритмі¹ бастапқы төбеден графтағы барлық төбелерге дейін ең қысқа жолдарды табады. Алгоритм графтың барлық түрлерімен жұмыс жасай алады. Бірақ графта ұзындығы 0-ден кем болатын цикл болмауы қажет. Графта ұзындығы минус болатын цикл болса, алгоритм оны байқайды.

Алгоритм бастапқы төбеден графтың барлық төбелеріне дейінгі қашықтықтарды қадағалайды. Алғашында арақашықтықты бастапқы төбеден 0-ге тең, ал басқалары үшін шексіздік деп белгілейміз. Кейін алгоритм төбелер арасындағы арақашықтықты мейлінше азайтатын қырларды іздейді. Егер де арақашықтықты қысқарту мүмкін болмаса, алгоритм аяқталады.

Мысал

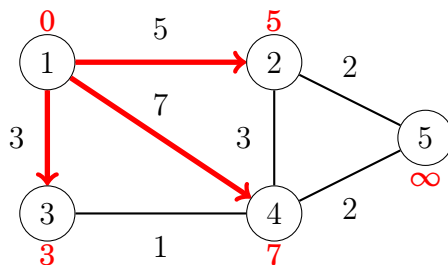
Келесі графта Беллман-Форд алгоритмі қалай жұмыс жасайтынын қарастырамыз:

¹Алгоритм атауы американдық ғалымдар Ричард Беллман мен Лестер Фордтың құрметіне қойылған. Форд бұл алгоритмді 1956 жылы басқа математикалық есепті зерттеу кезінде ойлап тауып, жарияласа, Беллман 1958 жылы ең қысқа жолды табудың нақты міндеті туралы мақаласында береді [30, 31].

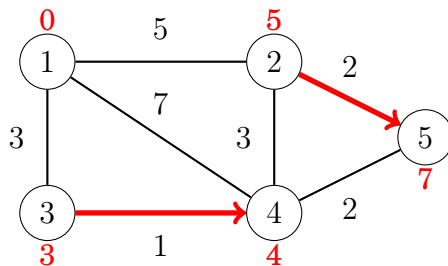


Біріншіден, графтың әр төбесіне арақашықтық тағайындалады. Бастапқы төбеге дейінгі арақашықтық **0**-ге тең, ал қалған барлық төбелерге дейінгісі – шексіз.

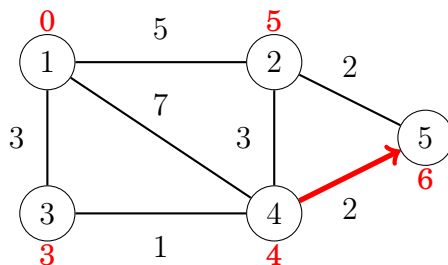
Алгоритм қашықтықты азайтатын қырларды іздейді. 1-төбенің барлық қырлары азайта алады:



Содан кейін $2 \rightarrow 5$ және $3 \rightarrow 4$ қырлары қашықтықты қысқартады:

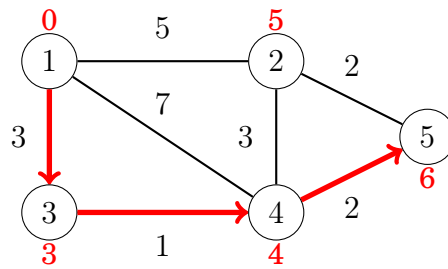


Төменде соңғы өзгеріс көрсетіледі:



Осыдан кейін ешбір қыр арақашықтықты азайта алмайды. Бұл арақашықтықтардың енді өзгермейтіндігін және біз бастапқы төбеден басқа төбелерге дейінгі арақашықтықтарды есептеп шыққанымызды білдіреді.

Мысалы, 1-төбеден 5-төбеге дейінгі арақашықтығы **6**-ға тең ең қысқа жол осылай белгіленген:



Кодтың жазылуы

Төменде берілген Беллман-Форд алгоритмінің коды x төбесінен графтың барлық төбелеріне дейінгі ең қысқа арақашықтықты анықтайды. Кодта (a, b, w) құрылымында сақталған edges қырлар тізбегі қолданылады, яғни a төбесінен b төбесіне дейін салмағы w -ге тең қыр бар.

Алгоритм қадамдарды $n - 1$ рет қайталайды. Әр қадамда графтағы барлық төбелерді өтіп шығып, арақашықтықты азайтуға тырысады. distance жиымы x төбесінен барлық төбелерге дейінгі қашықтықты сақтайды. Ал INF тұрақтысы шексіздікті білдіреді.

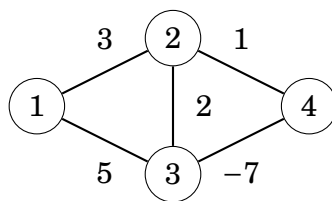
```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

Алгоритмнің уақытша күрделілігі – $O(nm)$. Себебі алгоритм $n - 1$ қадам жасайды және әр қадамда графтағы барлық m қырды өтіп шығады. Егер де графта ұзындығы минус цикл болмаса, $n - 1$ қадамнан кейін графтағы барлық арақашықтық ең тиімді болады. Себебі, әр ең қысқа жол $n - 1$ санынан көп қыр қамти алмайды.

Түпкілікті арақашықтық әдетте барлық $n - 1$ айналым жасалмастан бұрынақ белгілі болады. Сол себепті кезекті айналымда ешбір арақашықтық азаймаса, алгоритмді жай ғана тоқтата салуға болады.

Теріс цикл

Беллман-Форд алгоритмі арқылы графта теріс цикл (ұзындығы 0-ден кем болатын цикл) бар-жоғын тексеруге болады. Мысалға төмендегі графты алайық:



Бұл графта $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$ ұзындығы -4 болатын теріс цикл бар.

Бұл жағдайда мұндай циклды қамтитын кез келген жолдың ұзындығын шексіз азайтуға болады, сондықтан ең қысқа жол ұғымы мағынасын жоғалтады.

Графта теріс цикл бар-жоғын Беллман-Форд алгоритмін n қадамға созу арқылы тексеруге болады. Біз графтағы барлық төбелерге ең қысқа жолды табу үшін $n - 1$ қадам жеткілікті екенін білеміз. Бірақ граф теріс циклды қамтыса, алгоритм n қадамда да басқа қысқа жолды табады. Осылайша тағы бір қадам жасап, графты теріс циклға тексеруге болады. Бұл алгоритмнің бастапқы төбенің таңдалуына қарамастан, теріс циклды көрсететінін атап өтпекпіз.

SPFA алгоритмі

SPFA алгоритмі ("Shortest Path Faster Algorithm") [32] – Беллман-Форд алгоритмінің жылдамырақ нұсқасы. SPFA алгоритмі графтағы барлық қырларды өтпейді. Ол қарастырылатын қырларды ақылды түрде өтіп шығады.

Алгоритм арақашықтықтарды қысқарту үшін пайдаланылуы мүмкін төбелердің кезегін сақтайды. Алдымен кезекке бастапқы x төбесін енгізеді. Кейін кезектегі бірінші төбені алып, сол төбеден шығатын қырларды қарастырады. Егер $a \rightarrow b$ қыры арақашықтықты азайтса, b төбесі кезекке қосылады.

SPFA алгоритмінің тиімділігі графтың құрылымына тікелей байланысты: алгоритм көбіне тиімді, бірақ оның уақытша күрделілігі ең нашар дегенде $O(nm)$ болады және алгоритмді Беллман-Форд алгоритмі сияқты баяу ететін кіріс деректерін құра аламыз.

13.2 Дейкстра алгоритмі

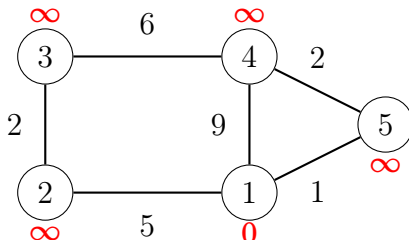
Дейкстра алгоритмі¹ де Беллман-Форд алгоритмі сияқты бастапқы төбеден графтың барлық төбелеріне дейінгі ең қысқа жолдарды табады. Дейкстра алгоритмінің артықшылығы – оның тиімділігінде және үлкен графтармен жұмыс жасай алуында. Дегенмен алгоритм жұмыс жасау үшін граф ішінде салмағы теріс болатын қыр болмауы керек.

Беллман-Форд алгоритмі сияқты Дейкстра алгоритмі төбелерге дейінгі арақашықтықты сақтайды және іздеу кезінде оларды біртіндеп азайтады. Графта теріс салмақты қыр болмағандықтан, Дейкстра алгоритмі әр қырмен тек бір рет жұмыс жасайды және осыған байланысты жылдам болып келеді.

¹Бұл алгоритмді 1959 жылы Е.В.Дейкстра жариялаған еді [33]; Дегенмен түпнұсқа еңбекте алгоритмді тиімді жүзеге асыру жолы жазылмаған болатын.

Мысал

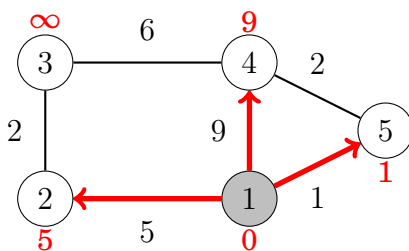
Келесі графта бастапқы төбесі 1 болатын Дейкстра алгоритмінің қалай жұмыс істейтінін қарастырамыз:



Беллман-Форд алгоритміндегідей қашықтық бастапқы төбе үшін 0-ге, ал қалған төбелер үшін шексіздікке тең.

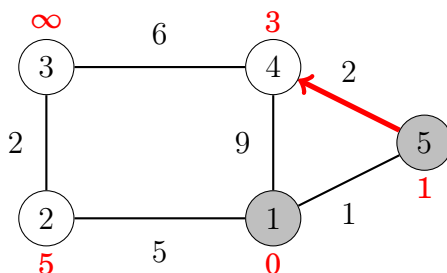
Әр қадам сайын Дейкстра алгоритмі қолданылмаған және қашықтығы ең аз төбені таңдайды. Ең алғашқы қарастырылатын төбе – қашықтығы 0-ге тең 1-төбе.

Төбе таңдалғанда, алгоритм содан шығып тұрған барлық қырларды өтіп шығып, басқа төбелерге қашықтықты азайтуын тексереді:

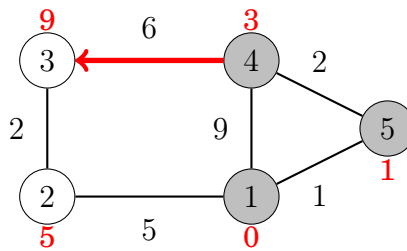


Осы жағдайда 1-төбеден шығып тұрған қырлар 2, 4 және 5 төбелеріне арақашықтықты қысқартты. Олардың арақашықтығы енді сәйкесінше 5, 9 және 1.

Келесі қарастырылатын төбе арақашықтығы 1-ге тең 5-төбе. Ол 4-төбенің арақашықтығын 9-дан 3-ке азайтты:

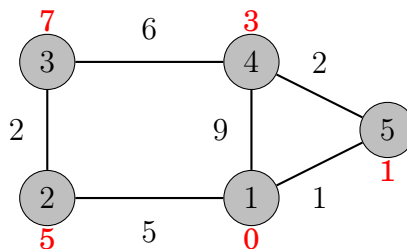


Ендігі қарастырылатын төбе – 4-төбе. Ол 3-төбеге дейінгі арақашықтықты 9-ға төмендетті:



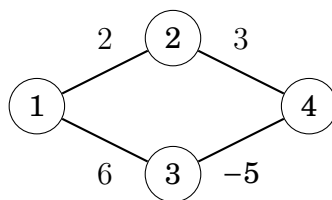
Бір төбе таңдалғанда оның арақашықтығының түпкі (яғни ең оңтайлы) болып табылуы Дейкстра алгоритмінің тамаша қасиетіне жатады. Мысалы, қазіргі сәтте 0, 1 және 3 арақашықтықтары 1, 5 және 4 төбелеріне түпкі болып саналады.

Алгоритм қалған екі төбемен жұмыс жасайды, түпкі арақашықтықтар төмендегідей болмақ:



Теріс қырлар

Дейкстра алгоритмінің тиімділігі графта теріс қыр жоқтығына негізделген. Егер теріс қыр болса алгоритм қате жауап беруі мүмкін. Мысалға, келесі графты қарастырайық:



1-төбеден 4-төбеге дейінгі ең қысқа жол $1 \rightarrow 3 \rightarrow 4$ -ке, ал оның ұзындығы 1-ге тең. Бірақ Дейкстра алгоритмі ең аз салмақтағы қырларға ілесе отырып, $1 \rightarrow 2 \rightarrow 4$ деген жолды табады. Алгоритм басқа жолда -5 -ке тең салмақ алдыңғы 6 -ға тең үлкен салмақтың орнын толтыратынын ескермейді.

Кодтың жазылуы

Жазылған келесі код бастапқы x төбесінен графтағы басқа төбелерге дейінгі ең қысқа арақашықты есептейді. Графтың қырлары сыбайластық тізімі арқылы сақталған. Егер a төбесінен b төбесіне салмағы w болатын қыр болса, $\text{adj}[a]$ векторы (b, w) жұбын сақтайды.

Дейкстра алгоритмінің тиімді үлгісі қолданылмаған ең аз қашықтықтағы төбені табудың тиімді жолдарын талап етеді. Осы ретте, төбелерді арақашықтығы бойынша реттейтін басымдылық кезегі жарамды деректер құрылымы саналады. Басымдылық кезегі арқылы келесі қарастырылатын төбені табу үшін логарифмдік уақыт жұмсалады.

Берілген кодта басымдылық кезегі $(-d, x)$ жұптарын сақтайды. Жұп қазіргі уақытта x -төбеге дейінгі қашықтық d екенін көрсетеді. distance жиымы әрбір төбеге дейінгі қашықтықты қамтиды және processed жиымы төбенің өңделгенін немесе өңделмегенін көрсетеді. Басында x -төбенің арақашықтығы 0-ге, ал басқа төбелер үшін ∞ тең.

```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
q.push({0,x});
while (!q.empty()) {
    int a = q.top().second; q.pop();
    if (processed[a]) continue;
    processed[a] = true;
    for (auto u : adj[a]) {
        int b = u.first, w = u.second;
        if (distance[a]+w < distance[b]) {
            distance[b] = distance[a]+w;
            q.push({-distance[b], b});
        }
    }
}
```

Басымдылық кезегінде арақашықтықты минус белгісімен сақтайтынымызды ескеруіміз керек. C++ ішіндегі басымдылық кезегі максималды элементтерді табатындықтан, ал бізге минималды элементтер қажет болғандықтан біз осылай сақтауға мәжбүрміз. Минус белгісін қою арқылы біз тікелей әдепкі басымдылық кезегімен жұмыс жасай аламыз¹.

Басымдылық кезегінде бір төбенің бірнеше данасы болуы мүмкін екендігін де ескеріңіз. Дегенмен ең аз қашықтықтағы данасы ғана өңделеді.

Жоғарыдағы кодтың уақытша күрделілігі – $O(n + m \log m)$. Себебі алгоритм графтағы барлық төбелерді өтіп шығады және әр қырдан көп дегенде бір арақашықтықты басымдылық кезегіне қосады.

13.3 Флойд-Уоршелл алгоритмі

Флойд-Уоршелл алгоритмі² ең қысқа жолдарды табу есебіне басқаша тәсіл ұсынады. Басқа алгоритмдерге қарағанда, ол төбелер арасындағы барлық қысқа жолдарды бір өту арқылы табатындығымен ерекшеленеді.

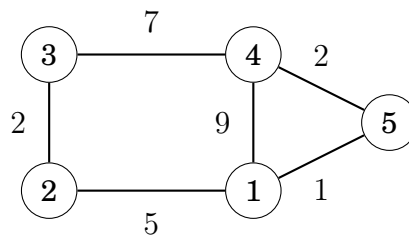
¹Әрине кезекті 4.5 тарауында сипатталғандай етіп жариялауға және оң қашықтықтарды пайдалануға болар еді, алайда бұл кезде код ұзақ болып кетеді.

²Алгоритм 1962 жылы оны дербес жариялаған Р.В.Флойд пен С.Уоршеллдің құрметіне аталған. [34, 35].

Алгоритм төбелер жұбының арақашықтығынан тұратын матрицаны қолдайды. Бастапқы сәтте матрица сыбайлас матрица негізінде меншіктейді. Содан кейін алгоритм бірнеше айналымдарды орындап, әр айналымда алдағы уақытта жолдың аралық төбесі болуы мүмкін жаңа төбені таңдайды. Сол төбені пайдалана отырып, қашықтықты азайтады.

Мысал

Төмендегі графта Флойд-Уоршелл алгоритмі қалай жұмыс істейтінін қарастырамыз:



Алғашында әр төбеден өзіне дейінгі арақашықтық 0-ге, ал егер a мен b төбелерінің арасында салмағы x болатын қыр болса, арақашықтық x -ке тең болады. Қалған арақашықтарды шексіздікке теңейміз.

Бұл граф үшін бастапқы жиым:

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

Алгоритм бірнеше дәйекті айналымдардан тұрады. Әр айналымда алгоритм бір төбені таңдап, төбелер арасындағы арақашықтықтарды таңдалған төбе арқылы азайтуға тырысады.

Алғашқы айналымда 1-төбе аралық төбе болады. 1-төбе оларды байланыстырып тұрғандықтан, 2 және 4-төбелер арасындағы арақашықтығы 14-ке, ал 2 мен 5-төбелер арасында 6-ға жаңарады.

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	14	6
3	∞	2	0	7	∞
4	9	14	7	0	2
5	1	6	∞	2	0

Екінші айналымда 2-төбе таңдалады. 1 мен 3-төбелер және 3 мен 5-төбелер арасында арақашықтық жаңарады.

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	14	6
3	7	2	0	7	8
4	9	14	7	0	2
5	1	6	8	2	0

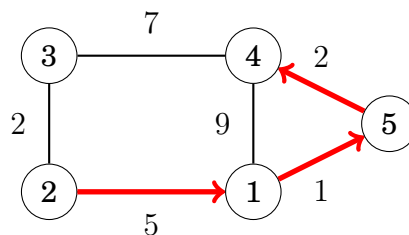
Үшінші айналымда, 3-төбе таңдалынады. Осы төбе 2 мен 4 арасында арақашықтықты азайтады.

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	9	6
3	7	2	0	7	8
4	9	9	7	0	2
5	1	6	8	2	0

Алгоритм барлық төбелерді қарап шыққанға дейін осылай жалғаса береді. Соңында жиым екі төбелер арасындағы минималды арақашықтарды қамтиды.

	1	2	3	4	5
1	0	5	7	3	1
2	5	0	2	8	6
3	7	2	0	7	8
4	3	8	7	0	2
5	1	6	8	2	0

Мысалы, осы жиым бізге 2 мен 4-төбелер арасында ең қысқа жолдың арақашықтығы 8 екенін көрсетеді. Ол келесі жолға сәйкес келеді:



Кодтың жазылуы

Флойд-Уоршалл алгоритмінің артықшылығы - кодтың оңай жазылуында. Келесі кодта $\text{distance}[a][b]$ a мен b арасындағы ең қысқа жолды сақтайтын жиым жарияланған. Алдымен distance жиымын adj сыбайластық матрицасы арқылы меншіктейміз.


```

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (i == j) distance[i][j] = 0;
        else if (adj[i][j]) distance[i][j] = adj[i][j];
        else distance[i][j] = INF;
    }
}

```

Содан кейін төмендегідей ретпен ең қысқа жолдарды табамыз:

```

for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            distance[i][j] = min(distance[i][j],
                                   distance[i][k]+distance[k][j]);
        }
    }
}

```

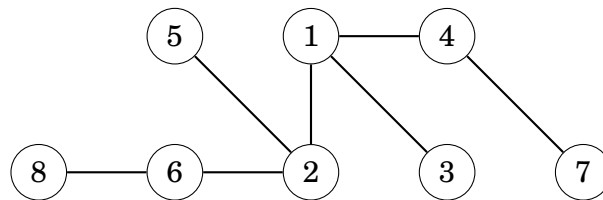
Аталған алгоритмнің уақытша күрделілігі $O(n^3)$. Себебі алгоритм төбелерден өтетін үш кірістірілген циклді қамтиды.

Флойд-Уоршалл алгоритмін жүзеге асыру өте қарапайым болғандықтан, оны тіпті графтағы бір ғана қысқа жолды табу қажеттілігі туындаған жағдайда да ұсынуға болады. Алайда оны кубтық уақытша күрделілігі қолайлы жағдайда шағын графтар үшін ғана қолдана аламыз.

14-тарау. Дарақтағы алгоритмдер

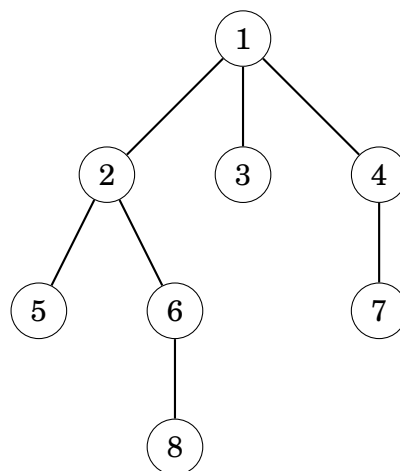
Дарақ – өзара байланысты, циклі жоқ граф. Дарақ n төбеден және $n - 1$ қырдан тұрады. Кез келген қырды алып тастаса, дарақ екі компонентке бөлінеді және кез келген жерге қыр қосса, дарақта цикл пайда болады. Оған қоса, дарақта әр екі төбенің арасында жалғыз жол болады.

Мысалы, төмендегі дарақ 8 төбеден және 7 қырдан тұрады:



Дарақтың жапырақтары – дәрежесі 1-ге тең (яғни жалғыз бір көршісі бар) төбелер. Мысалы, жоғарыда берілген дарақтың жапырақтары – 3, 5, 7 және 8-төбелер.

Түбірлі дарақта бір төбе дарақтың түбірі ретінде таңдалады, ал басқа төбелер оның астында орналасады. Үлгідегі графтың түбірі – 1-төбе:

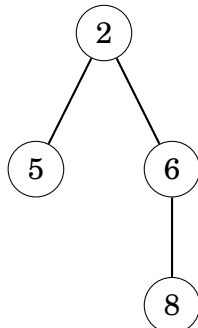


Түбірлі дарақтағы төбенің ұлдары – төмендегі көршілері, ал төбенің әкесі – үстіндегі көршісі. Түбірден басқа, әр төбенің бір әкесі болады. Мысалы үлгідегі дарақта 2-төбенің ұлдары – 5 және 6-төбелер, ал оның әкесі 1-төбе саналады.

Төбенің ішдарағы деп дарақтың сол төбеден және оның барша ұрпақтарынан тұратын бөлігін атаймыз. Басқаша айтқанда, v төбесінің ішдарағы u

төбесінің u -дан дарактың түбіріне дейінгі жолында міндетті түрде v кездесетін төбелерінен тұрады.

Мысалға, жоғарыдағы даракта 2-төбенің ішдарағы 2, 5, 6 және 8 төбелерден тұрады.



14.1 Даракты аралап шығу

Жалпы графты аралайтын алгоритмдерді даракты аралау үшін де қолдануға болады. Бірақ даракты аралайтын кодты жазу оңайырақ. Себебі даракта циклдер болмайды және төбеге бірнеше бағыттан келу мүмкін емес.

Даракты өтіп шығу үшін әдетте кез келген төбеден тереңдігі бойынша ізденіс жүргізіледі. Мысалы, төмендегі рекурсивті функцияны қолдануға болады:

```
void dfs(int s, int e) {  
    // process node s  
    for (auto u : adj[s]) {  
        if (u != e) dfs(u, s);  
    }  
}
```

Әлі бармаған төбелерді ажырату мақсатында қазіргі төбе s параметрінің қасына ертерек өтіп кеткен төбе e параметрін қосуымызға болады.

Келесі функцияның шақыруы ізденісті x төбесінен бастайды:

```
dfs(x, 0);
```

Бірінші шақырғанда $e = 0$. Өйткені бұл жерде ешқандай алдыңғы төбе жоқ және дарактың кез келген бағытына өтуге рұқсат етіледі.

Динамикалық бағдарламалау

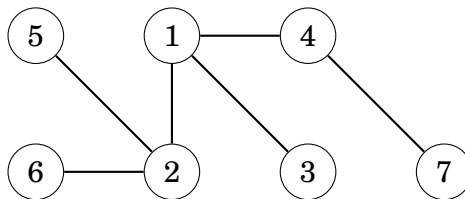
Даракты аралау барысында динамикалық бағдарламалауды ақпараттарды есептеу үшін қолдана аламыз. Мысалы, оның көмегімен түбірлі даракта $O(n)$ уақытта әр төбенің ішдарағындағы төбелердің санын немесе төбеден жапыраққа дейінгі ең ұзын жолдың ұзындығын есептеуге болады.

Үлгі ретінде келесі есепті қарастырамыз. Мысалы әр s төбесі үшін $\text{count}[s]$ мәнін сақтаймыз. Бұл мән бізге s төбесінің ішдарағындағы төбелер санын көрсетеді. Ішдарақ төбенің өзін және оның ұлдарының ішдарағындағы барлық төбелерді қамтиды. Осылайша біз төбелердің санын берілген код арқылы есептей аламыз:

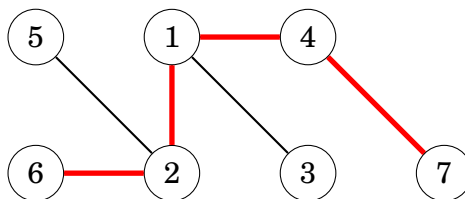
```
void dfs(int s, int e) {
    count[s] = 1;
    for (auto u : adj[s]) {
        if (u == e) continue;
        dfs(u, s);
        count[s] += count[u];
    }
}
```

14.2 Диаметр

Дарақтың диаметрі – дарақтағы екі төбенің арасындағы жолдың максималды ұзындығы. Мысалы, келесі дарақты қарастырайық:



Дарақтың диаметрі 4-ке тең, ол келесі жолға сәйкес келеді:



Ұзындықтары максималды бірнеше жол болуы мүмкін екенін ескерген дұрыс. Жоғарыдағы жолда 6-төбені 5-төбемен алмастырсақ, тағы бір ұзындығы 4-ке тең жолды табар едік.

Енді дарақтың диаметрін есептеу үшін уақытша күрделілігі $O(n)$ болатын екі алгоритмді қарастырамыз. Бірінші алгоритм динамикалық бағдарламалауға негізделген. Ал екінші алгоритм тереңдігі бойынша екі ізденісті қолданады.

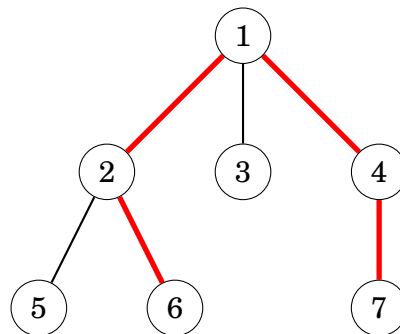
1-алгоритм

Көптеген дарақ есептерін шешуде мынадай жалпы тәсілді қолданамыз. Алдымен дарақтағы кез келген бір төбені түбір етіп таңдап аламыз. Кейін

есепті әр ішдарақ үшін бөлек шығара береміз. Біздің диаметрді есептеу үшін қолданатын бірінші алгоритміміз де осындай идеяны ұстанады.

Мынадай маңызды ескертуді жадымызда сақтағанымыз жөн: Түбірлі дарақта әр жолдың ең биік нүктесі – жолға жататын ең жоғарғы төбесі болады. Осылайша біз дарақтағы әр төбеге ең биік нүктесі сол болатындай ұзындығы максималды жолды есептей аламыз. Осындай барлық жолдардың біреуі дарақтың диаметріне тең болады.

Мысалы, төмендегі дарақта 1-төбе диаметрдің бойында жататын ең жоғарғы төбе болып тұр:



Біз әр x -төбеде екі санды сақтаймыз:

- $\text{toLeaf}(x)$: x -тен бастап кез келген жапыраққа дейінгі жолдың максималды ұзындығы
- $\text{maxLength}(x)$: ең жоғарғы нүктесі x болатын жолдың максималды ұзындығы.

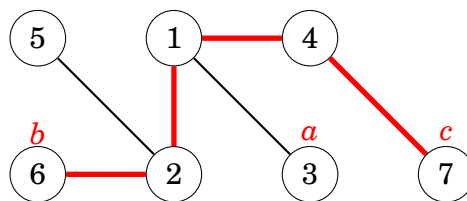
Мысалы, жоғарыдағы дарақта $1 \rightarrow 2 \rightarrow 6$ жолы болғандықтан $\text{toLeaf}(1) = 2$, және $6 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 7$ жолы болғандықтан $\text{maxLength}(1) = 4$. Осы жағдайда диаметр – $\text{maxLength}(1)$.

$O(n)$ уақыт ішінде барлық төбелер үшін осы мәндерді есептеуде динамикалық бағдарламалауды пайдалануға болады. Алдымен $\text{toLeaf}(x)$ мәнін есептеу үшін x төбесінің ұлдарынан өтіп шығамыз. Олардың ішінен $\text{toLeaf}(c)$ максималды мәні бар c төбесін таңдап, бұл мәнге 1-ді қосамыз. Содан соң $\text{maxLength}(x)$ мәнін есептеу үшін біз $\text{toLeaf}(a) + \text{toLeaf}(b)$ қосындысы максималды болатындай екі түрлі a және b ұл төбелерін таңдап, алынған мәнге 2-ні қосамыз.

2-алгоритм

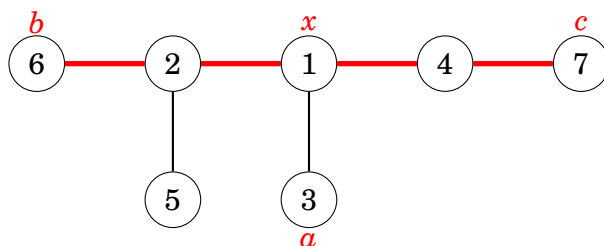
Дарақтың диаметрін есептеудің тағы бір тиімді әдісі екі тереңдігі бойынша ізденіске негізделеді. Алдымен біз дарақтағы кез келген бір a төбесін таңдаймыз және тереңдігі бойынша ізденіс арқылы a төбесінен ең алыс орналасқан b төбесін табамыз. Кейін біз екінші тереңдігі бойынша ізденіс арқылы b төбесінен ең алыс орналасқан c төбесін табамыз. Дарақтың диаметрі b мен c төбелерінің арақашықтығына тең болады.

Келесі граф үшін a , b , c төмендегідей нұсқада да кездесе алады:



"Бұл – талғампаз әдіс, бірақ неге ол дұрыс жұмыс істейді? деген сұраққа жауап іздейміз.

Ол дараққа басқаша қарауға көмектеседі. Оны көрсету үшін дарақтың диаметрін көлденең қойып, барлық басқа төбелер соған ілінетін түрін қарастырайық:

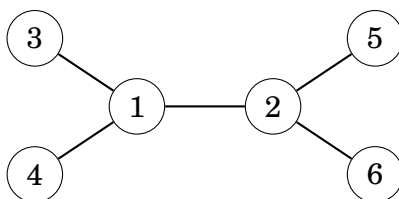


Мұнда a төбесінің дарақтағы диаметрге қосылған жері x төбесімен белгіленген. a төбесінен ең алыс орналасқан төбе – b төбесі, c төбесі немесе x төбесінен сондай немесе одан алыс арақашықтықта орналасқан қандай да бір басқа төбе. Осылайша бұл төбе диаметрге сәйкес келетін жолдың шеткі төбесіне жарамды таңдау бола алады.

14.3 Барлық ең ұзын жолдар

Келесі қарастыратын есебіміз – дарақтың әрбір төбесінен басталатын ұзындығы максималды жолды табу. Бұл есеп дарақтың диаметрін табудың жалпылама түріне ұқсас болғандықтан, оны $O(n)$ уақытта шығара аламыз.

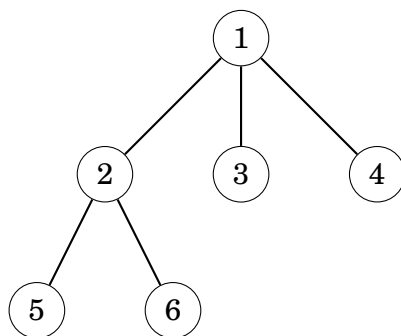
Мысалы, келесі дарақты қарастырайық:



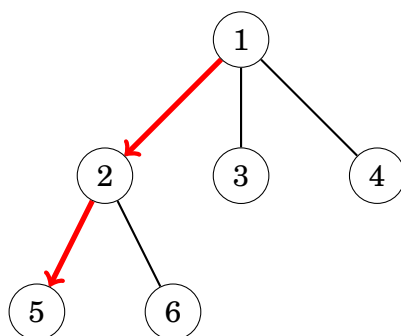
Біз $\text{maxLength}(x)$ мәнінде x төбесінен басталатын дарақтағы максималды жолдың ұзындығын сақтаймыз. Мысалы, жоғарыдағы дарақта $\text{maxLength}(4) = 3$, себебі, дарақта мынадай жол бар: $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$. Төменде барлық төбелердің maxLength мәндерін белгіледік:

x төбесі	1	2	3	4	5	6
$\text{maxLength}(x)$	2	2	3	3	3	3

Бұл есепте де бастапқыда бір төбені түбір ретінде таңдаймыз:

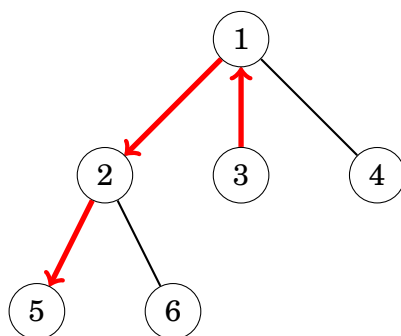


Есептің бірінші бөлігінде әр x төбесі үшін оның ұлы арқылы өтетін жолдың максималды ұзындығын табу керек. Мысалы, 1-төбеден ең ұзын жол 2-ұлынан өтеді:

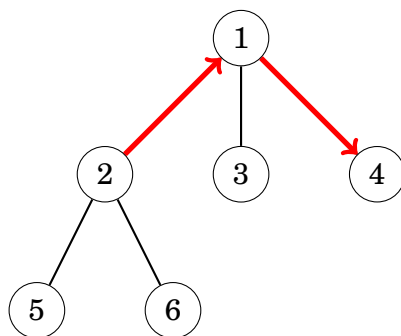


Бұл бөлікті $O(n)$ уақытта шығару оңай. Себебі алдында атап өткеніміздей динамикалық бағдарламалау әдісін қолдана аламыз.

Кейін есептің екінші бөлігінде әр x төбесіне әкесі p арқылы жалғанатын максималды ұзын жолды есептеу керек. Мысалы, 3-төбеге 1-төбеден келетін ең ұзын жол:



Бір қарағанда p төбесінен шығатын ең ұзын жолды таңдау керек сияқты көрінеді. Бірақ ол әрқашан жұмыс жасамайды. Себебі p -төбенің ең ұзын жолы x -төбеден өтуі мүмкін. Төменде осы жағдайға мысал келтіреміз:



Десек те екінші бөлікті $O(n)$ уақытта шығара аламыз. Бұл ретте әр төбе үшін екі максималды ұзындықты сақтау керек. Олар:

- $\text{maxLength}_1(x)$: x төбесінен шығатын максималды ұзын жолдың мәні
- $\text{maxLength}_2(x)$: x төбесінен шығатын бірінші бағыттан басқа максималды ұзын жолдың мәні

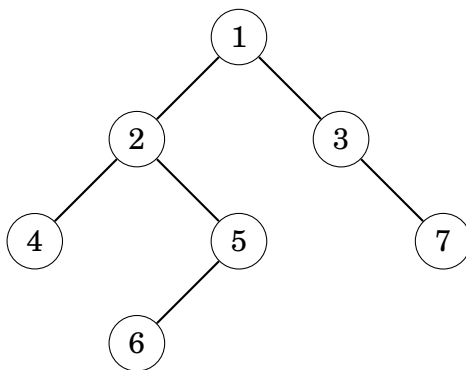
Мысалы, жоғарыдағы графта, $\text{maxLength}_1(1) = 2$ $1 \rightarrow 2 \rightarrow 5$ жолына сәйкес келсе, ал $\text{maxLength}_2(1) = 1$ $1 \rightarrow 3$ жолына сәйкес келеді.

Егер нәтижесінде $\text{maxLength}_1(p)$ мәніне сәйкес жол x төбесінен өтсе, максималды ұзындық $\text{maxLength}_2(p) + 1$ -ге тең болады. Басқа жағдайда $\text{maxLength}_1(p) + 1$ мәніне тең болмақ.

14.4 Бинарлы дарақ

Бинарлы дарақ – әр төбенің сол және оң жақ ішдарақтары болатын түбірлі дарақ. Төбенің сол немесе оң ішдарақтары болмауы да мүмкін. Осылайша бинарлы дарақтағы әр төбенің нөл, бір немесе екі ұлы болады.

Төмендегі дарақ бинарлы дараққа мысал бола алады:



Бинарлы дарақта төбелерді рекурсивті түрде үш түрлі әдіспен өтіп шығуға болады:

- кемімелі (pre-order): алдымен түбірден бастайды, сосын сол жақ ішдараққа өтіп, кейін оң жаққа барады.
- бірізді (in-order): алдымен сол жақ ішдараққа барады, ал содан кейін түбірге өтіп, соңында оң жақ ішдараққа жетеді.

- үдемелі (post-order): алдымен сол жақ ішдараққа барады, ал содан соң оң жақ ішдараққа өтіп, кейін түбірге келеді.

Берілген дарақтағы төбелердің тізімі кемімелі реттілік (pre-order) әдісімен [1,2,4,5,6,3,7] болса, бірізді реттілік (in-order) әдісінде [4,2,6,5,1,3,7], ал үдемелі реттілік (post-order) әдісінде [4,6,5,2,7,3,1] болады.

Егер біз дарақтың кемімелі (pre-order) және бірізді (in-order) реттіліктерінде өту тізімін білсек, сол тізімге қарап дарақты құрай аламыз. Мысалы, егер кемімелі (pre-order) реттілікте өту тізімі [1,2,4,5,6,3,7] және бірізді (in-order) реттілікте өту тізімі [4,2,6,5,1,3,7] болса, жоғарыдағы дарақты құрауға болады. Сондай-ақ үдемелі (post-order) және бірізді (in-order) реттілік тізімдері арқылы да ұқсас жолмен дарақтың құрылымын табуға болады.

Бірақ дарақтың кемімелі (pre-order) және үдемелі (post-order) реттілік тізімдерін білсек, жағдай басқаша болады. Бұл жағдайда сәйкес келуі мүмкін дарақ құрылымдары көбейеді. Мысалы, төмендегі дарақтарға қарайық:

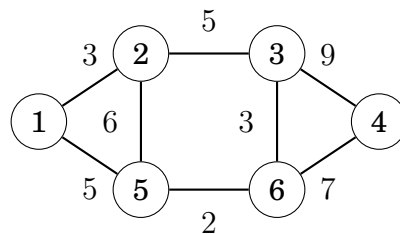


Екі дарақтың да кемімелі (pre-order) [1,2] және үдемелі (post-order) [2,1] реттілік тізімдері сәйкес. Дегенмен дарақтардың құрылымдары әртүрлі.

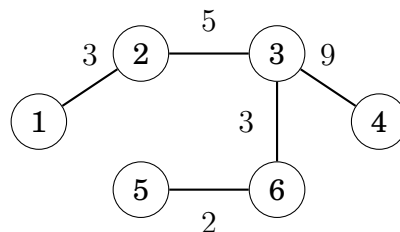
15-тарау. Қаңқалы дарақ

Қаңқалы дарақ графтың барлық төбелерінен және кез келген екі төбесі арқылы жол болатындай қырлар ішжиынынан тұрады. Олар да жалпы дарақ сияқты ациклді және байланысқан болады. Әдетте қаңқалы дарақты құрудың бірнеше тәсілі болады.

Үлгі ретінде төмендегі графты қарастырайық:

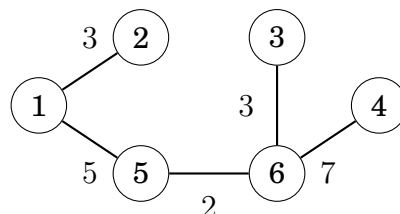


Графтың бір қаңқалы дарағы:

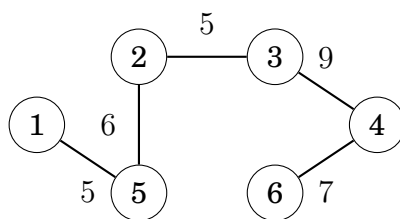


Қаңқалы дарақтың салмағы қыр салмақтарының қосындысына тең. Мысалы, жоғарыдағы қаңқалы дарақтың салмағы – $3 + 5 + 9 + 3 + 2 = 22$.

Минималды қаңқалы дарақ деп салмағы ең аз болатын қаңқалы дарақты айтамыз. Жоғарыдағы граф үшін минималды қаңқалы дарақтың салмағы 20-ға тең болады. Сондай дарақты төмендегідей етіп құрастыруға болады:



Ұқсас жағдайдағы максималды қаңқалы дарақты алайық. Максималды салмақтағы қаңқалы дарақ максималды қаңқалы дарақ деп аталады. Мысалдағы граф үшін максималды қаңқалы дарақтың салмағы 32-ге тең болады:



Графта бірнеше минималды және максималды қаңқалы дарақтың болуы мүмкін екенін ескергеніміз жөн. Сол себепті де дарақ бірегей бола алмайды.

Минималды және максималды қаңқалы дарақтарды құру үшін бірнеше ашкөз тәсілдерді қолдануға болады. Бұл бөлімде біз минималды және максималды қаңқалы дарақтарды құрастыруға көмектесетін екі алгоритмді талдаймыз. Екі алгоритм де графтың қырлары салмағы бойынша реттелген күйде жұмыс жасайды. Біз тек минималды қаңқалы дараққа тоқталамыз, себебі, жай ғана қырлардың ретін ауыстыру арқылы сол алгоритмдермен максималды қаңқалы дарақ құруға болады.

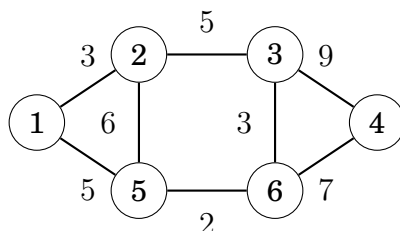
15.1 Краскал алгоритмі

Краскал алгоритміндегі¹ алғашқы қаңқалы дарақ ешқандай қыр қамтымай, тек графтың төбелерінен тұрады. Кейін алгоритм қырларды салмағы бойынша өсу ретімен өтіп шығу барысында әр қырды қосу немесе қоспау жағдайын тексереді. Егер қыр цикл тудырмаса, алгоритм қырды дараққа қосады.

Алгоритм дарақтың компоненттерін сақтап отырады. Басында графтың әр төбесі өзі құрап тұрған компонентке жатады. Кейін қыр жалғанған кезде екі компонент бір-бірімен қосылады. Нәтижесінде барлық төбелер бір компонентте болады және минималды қаңқалы дарақ құрылады.

Мысал

Краскал алгоритмі төмендегі графпен қалай жұмыс жасайтынын қарастырайық:



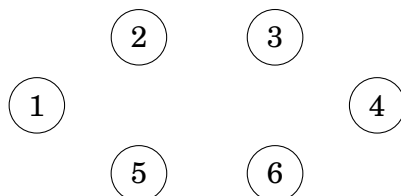
Алгоритмнің бірінші қадамы – қырларды өсу ретінде реттеу. Оның нәтижесі:

¹Алгоритмді 1956 жылы Дж.Б.Крускал жариялады [36].

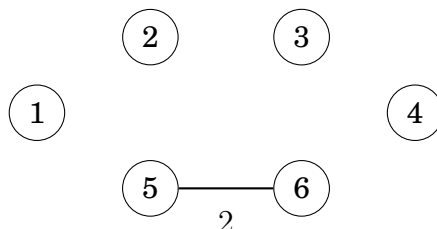
қыр	салмағы
5-6	2
1-2	3
3-6	3
1-5	5
2-3	5
2-5	6
4-6	7
3-4	9

Осыдан кейін алгоритм тізім арқылы өтіп, егер қыр екі бөлек компонентті біріктірсе, ол қырды дараққа қосады.

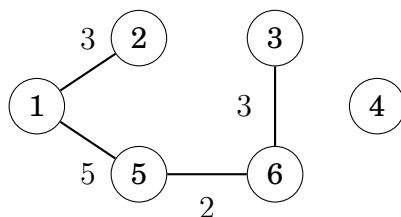
Бастапқыда әр төбе өзі құрап тұрған компоненттен тұрады:



Дараққа қосылатын бірінші қыр – 5-6. Осы {5} пен {6} қырлардың компоненттерін қосу арқылы біртұтас {5,6} компоненті құрылды.



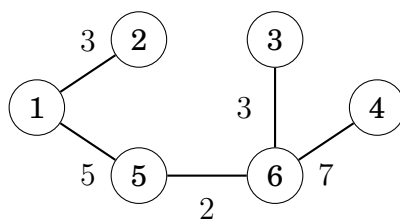
Содан кейін 1-2, 3-6 және 1-5 қырлары ұқсас әдіспен қосылады:



Бұл қадамдардан кейін компоненттердің көбі біріктіріледі де дарақта тек екі компонент қалады: {1,2,3,5,6} және {4}.

Тізімдегі келесі қыр – 2-3 қыры. Бірақ ол дараққа қосылмайды. Себебі 2 және 3 төбелері әлдеқашан компонентке біріктірілген. Дәл осы себеппен 2-5 қыры да дараққа қосылмайды.

Соңында 4–6 қыры дараққа қосылады:

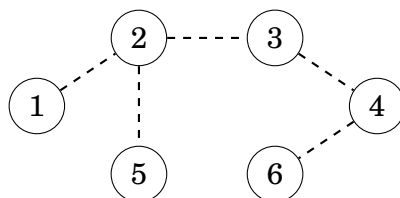


Бұдан кейін алгоритм ешқандай қыр қоспайды. Өйткені граф толық байланысып тұр. Соңғы құрастырылған граф – салмағы $2 + 3 + 3 + 5 + 7 = 20$ тең минималды қаңқалы дарақ.

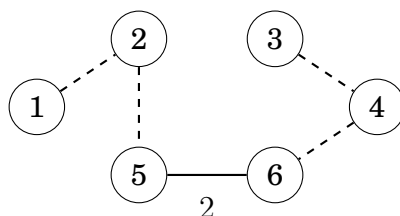
Жұмыс істеу себебі неде?

”Краскал алгоритмі қалай жұмыс істейді? Ашкөз стратегиясы неліктен минималды қаңқалы дарақты табуға кепілдік береді?” - деген жақсы сұрақ туындайды.

Графтың ең аз салмақты қыры қаңқалы дараққа қосылмаса не болатынын бақылап көрейік. Мысалы, төменде берілген графтың қаңқалы дарағы 5–6 қырын қамтымады деп есептейік. Біз дарақтың дәл құрылымын білмейміз. Бірақ ол қандай да бір қырларды қамтуы қажет. Дарақ келесідей болады делік:



Бірақ жоғарыда көрсетілген суреттегі қаңқалы дарақ минималды бола алмайды, өйткені бізге одан қандай да бір қырды алып тастауға және оны минималды салмағы 5–6 болатын қырмен ауыстыруға ешнәрсе кедергі келтірмейді. Осылайша салмағы азырақ қаңқалы дарақ құралады:



Сол себепті салмағы ең аз қаңқалы дарақты құру үшін дараққа ең аз салмақты қырларды қосу әрқашан оңтайлы. Ұқсас пайымдауды қолдана отырып, келесі минималды салмағы бар және т.с.с. қырды қосу қажет екенін көрсетуге болады. Сондықтан Краскал алгоритмі әрқашан минималды қаңқалы дарақты береді.

Кодтың жазылуы

Краскал алгоритмінің кодын жазған кезде қырлар тізімі көрінісін қолдану қолайлырақ. Алгоритмнің бірінші бөлігі тізімдегі қырларды $O(m \log m)$ уақытында реттейді. Осыдан кейін төменде көрсетілгендей алгоритмнің екінші бөлігінде ең аз қаңқалы дарақ құрылады:

```
for (...) {  
    if (!same(a,b)) unite(a,b);  
}
```

Цикл тізімдегі қырларды өтіп шығады және a мен b төбелерін қосатын әрбір $a-b$ қырларын өңдейді. Алгоритмді жүзеге асыру үшін екі функция қажет. Олар: a және b бір компонентте екенін анықтайтын `same` функциясы және a және b компоненттерін біріктіретін `unite` функциясы.

Әңгіме `same` және `unite` функцияларын қалай тиімді іске асыруға болады деген мәселеге барып тіреледі. Оның бір жолы – графты аралау арқылы `same` функциясын жүзеге асыру. Бұл кезде алгоритм a төбесінен b төбесіне өту мүмкіндігін тереңдік бойынша ізденіспен тексереді. Дегенмен мұндай функцияның уақытша күрделілігі – $O(n + m)$. Сондықтан алгоритм баяу болады. Өйткені `same` функциясы графтың әр қыры үшін шақырылады.

Біз есепті қиылыспайтын жиындар құрылымының – $O(\log n)$ уақытта екі функцияны да жүзеге асыруға мүмкіндік беретін деректер құрылымының көмегімен шешеміз. Осылайша, қырлар тізімін сұрыптап болғаннан кейінгі Краскал алгоритмінің уақытша күрделілігі $O(m \log n)$ болмақ.

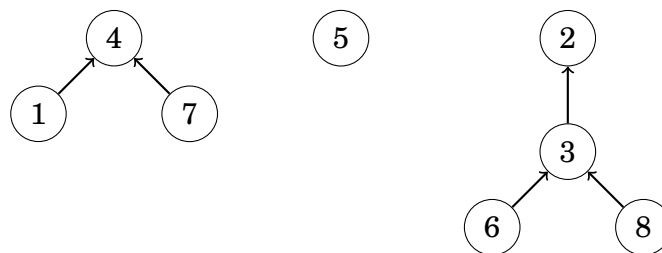
15.2 Қиылыспайтын жиындар құрылымы

Қиылыспайтын жиындар құрылымы (union-find structure) жиындар топтамасынан тұрады. Бұл жиындар екеуара қиылыспайды, яғни әр элемент бір жиынға ғана кіреді. Қиылыспайтын жиындар құрылымы уақытша күрделілігі $O(\log n)$ болатын екі амалды қолдайды. Олар: екі жиынды біріктіретін `unite` амалы және берілген элементті құрайтын жиынның басшысын табатын `find` амалы¹.

Құрылым

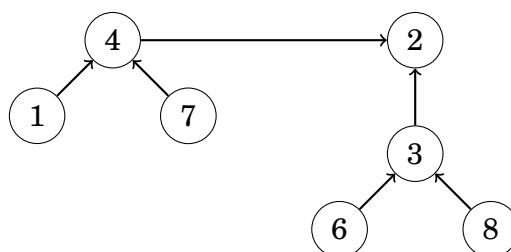
Қиылыспайтын жиындар құрылымындағы әр жиында бір элемент басшы болады және жиынның кез келген элементінен оның басшысына қарай жол түседі. Мысалы, $\{1,4,7\}$, $\{5\}$ және $\{2,3,6,8\}$ жиындарын төмендегі ретпен қарастырып көрейік:

¹Ұсынылған құрылымды 1971 жылы Дж. Д. Хопкрофт пен Дж. Д. Ульман енгізген болатын [37]. Ал 1975 жылы Р.Е.Тарьян бүгінде көптеген оқулықтарда талқыланып жүрген алгоритм құрылымының күрделі нұсқасын зерттеді [38].



Мұндағы 4, 5 және 2 – жиындардың басшылары болып саналады. Кез келген элементтің басшысын табу үшін сол элементтен басталатын жолмен өтіп шығу керек. Мысалы, 2-элемент 6-элементінің басшысы, себебі біз $6 \rightarrow 3 \rightarrow 2$ тізбегімен жүріп, 2-элементте тоқтадық. Басшылары бір болса ғана екі элемент бір жиынға жатады.

Бір жиынның басшысын екінші жиынның басшысына жалғау арқылы екі жиынды қосуға болады. Мысалы, $\{1, 4, 7\}$ және $\{2, 3, 6, 8\}$ жиындарын келесі ретпен қосуға болады:



Одан шығатын жиын $\{1, 2, 3, 4, 6, 7, 8\}$ элементтерін қамтиды. Кейін 2-элемент екі жиынның басшысы болады және бұрынғы басшы 4 енді 2-элементке нұсқайды.

Қиылыспайтын жиындар құрылымының тиімділігі жиындардың қалай біріктірілгеніне байланысты болады. Біз қарапайым стратегияны пайдалана аламыз, яғни әрқашан кіші жиынның басшысын үлкен жиынның басшысына қосамыз (ал егер жиындардың мөлшері бірдей болса, ерікті таңдау жасаймыз). Мұндай стратегияда кез келген жолдың ұзындығы $O(\log n)$ болады. Сондықтан біз сәйкес жолмен жүру арқылы кез келген элементтің басшысын оңтайлы таба аламыз.

Кодтың жазылуы

Қиылыспайтын жиындар құрылымының кодын жиымдар арқылы жазуға болады. Келесі кодта `link` жиымы тізбектегі әр элемент үшін келесі элементті, ал басшы болса өзін сақтайды. Сондай-ақ `size` жиымы әр басшы үшін сәйкес жиынының мөлшерін көрсетеді.

Басында, әр элемент жеке жиынға жатады:

```
for (int i = 1; i <= n; i++) link[i] = i;
for (int i = 1; i <= n; i++) size[i] = 1;
```

`find` функциясы x -элементтің басшысын тауып береді. Басшыны x элементінен басталатын жолдың бойымен жүру арқылы табуға болады.

```
int find(int x) {
    while (x != link[x]) x = link[x];
    return x;
}
```

same функциясы a мен b элементтері бір жиынға жататындығын тексереді. Мұны find функциясы арқылы оңай тексеруге болады:

```
bool same(int a, int b) {
    return find(a) == find(b);
}
```

unite функциясы a және b элементтерін қамтитын жиындарды біріктіреді (осы екі элемент әртүрлі жиында болуы қажет). Функция бірінші жиынның басшысын табады, кейін кіші жиынды үлкен жиынға қосады.

```
void unite(int a, int b) {
    a = find(a);
    b = find(b);
    if (size[a] < size[b]) swap(a,b);
    size[a] += size[b];
    link[b] = a;
}
```

Әр тізбектің ұзындығы $O(\log n)$ болса, find функциясының уақытша күрделілігі $O(\log n)$ -ге тең. Осы жағдайда same мен unite функциялары $O(\log n)$ уақытында жұмыс жасайды. unite функциясы кіші жиынды үлкен жиынға қосу арқылы жол тізбегін $O(\log n)$ ұзындығынан асырмауға тырысады.

15.3 Прим алгоритмі

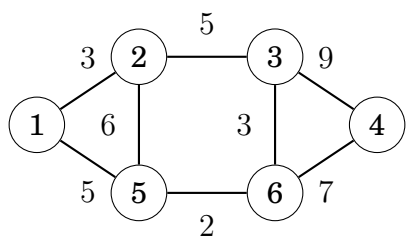
Прим алгоритмі¹ минималды қаңқалы дарақты табудың балама әдісіне жатады. Алдымен ол дараққа еркін төбені қосады. Ал келесі төбелерді қосу барысында минималды салмағы бар қырды таңдайды. Барлық төбелер қосылғаннан кейін, минималды қаңқалы дарақ құрылады.

Прим алгоритмі Дейкстра алгоритміне ұқсас болып келеді. Бірақ, олардың аздаған айырмашылығы бар: Дейкстра алгоритмі әрқашан бастапқы төбеден арақашықтығы ең аз болатындай қырды таңдаса, Прим алгоритмі қарапайым дараққа жаңа төбені қосатын ең аз қырды таңдайды.

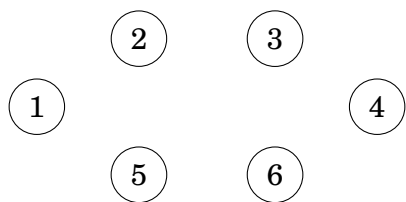
Мысал

Прим алгоритмі төмендегі графта қалай жұмыс істейтінін қарастырайық:

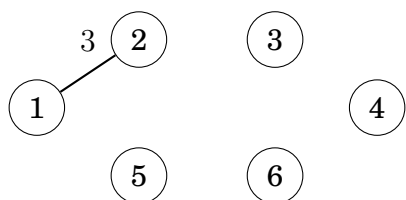
¹Алгоритм оны 1957 жылы жариялаған Р.С.Примнің атымен аталған [39]. Дегенмен дәл осындай алгоритмді 1930 жылы В.Ярник ашқан болатын.



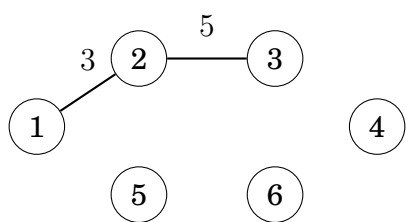
Бастапқыда төбелер арасында ешқандай қыр болмайды:



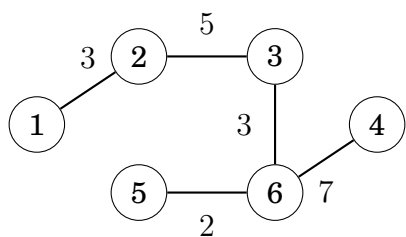
Еркін төбе ретінде 1-төбені таңдайық. Алдымен біз салмағы 3-ке тең қыр арқылы 2-төбені қосамыз:



Бұдан кейін салмағы 5-ке тең екі қыр қалады. Сол себепті біз дараққа не 3-төбені, не 5-төбені қосуға мүмкіндік аламыз. Алдымен 3-төбені қосып көрейік:



Осы процесс барлық төбелер дараққа қосылғанша жалғасады:



Кодтың жазылуы

Дейкстра алгоритмі сияқты Прим алгоритмін де басымдылық кезегін қолдана отырып, тиімді жазуға болады. Басымдылық кезегі қазіргі компонентке бір қырмен қосуға болатын барлық төбелерді сақтауы керек. Бұл жерде Дейкстра алгоритміндегідей төбелерді салмағының өсу ретімен сақтайды.

Прим алгоритмінің уақытша күрделілігі Дейкстра алгоритміндегідей $O(n + m \log m)$ болады. Көбінесе Примнің және Краскалдың алгоритмдері бірдей тиімді. Сондықтан қай алгоритмді таңдау мәселесі әр адамның қалауына байланысты болмақ. Дегенмен спорттық бағдарламалаушылардың көпшілігі Крускал алгоритмін пайдаланады.

16-тарау. Бағытталған граф

Осы тарауда біз бағытталған графтың екі түрін қарастырамыз:

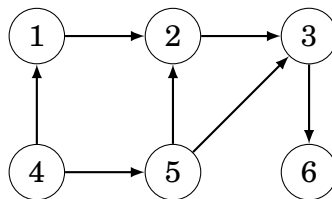
- Циклсіз граф: графта ешқандай цикл жоқ, яғни бір төбеден басталып, сол төбеден аяқталатын жол болмайды.
- Мирасқорлар графы: әр төбенің шығыстың жарты дәрежесі 1-ге тең, яғни әр төбенің бірегей мирасқор болады.

Екі жағдайда да графтардың ерекше қасиеттеріне негізделген тиімді алгоритмдерді қолдана аламыз.

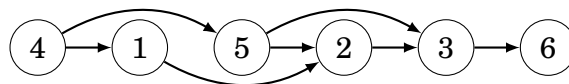
16.1 Топологиялық сұрыптау

Топологиялық сұрыптау – бағытталған графтағы төбелерді белгілі бір тәртіпке негіздеу арқылы реттеу. Бұл тәртіп бойынша егер a төбесінен b төбесіне жол болса, реттеу барысында a төбесі b төбесінен бұрын келеді.

Мысалы, төмендегі графтың



топологиялық сұрыпталған тізімі $[4, 1, 5, 2, 3, 6]$ осындай болады:



Циклсіз графты әрқашан топологиялық сұрыптауға болады. Ал егер графта цикл болса, оны топологиялық сұрыптай алмаймыз. Себебі, циклдің бір төбесі циклдегі басқа төбеден тізімде бұрын келе алмайды.

Тереңдік бойынша іздеу бағытталған графта циклдардың бар-жоғын тексеруге де, ал цикл болмаған жағдайда топологиялық сұрыптауды құруға да мүмкіндік береді екен.

Алгоритм

Алгоритмнің идеясы графтағы төбелерді бір-бірлеп өтуге, өту барысында төбенің бұрын қарастырылмағаны анықталса, сол төбеден тереңдігі бойынша ізденісті бастауға негізделеді. Іздеу кезінде төбелердің үш түрлі күйі болады. Олар:

- 0-күй: төбе әлі қарастырылмаған (ақ)
- 1-күй: төбе қарастыру барысында (сұрғылт)
- 2-күй: төбе қарастырылды, яғни толық өңделді (қара сұр)

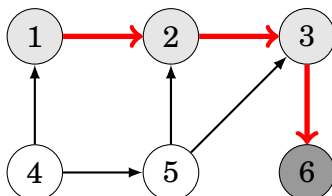
Басында әр төбе 0-күйде болады. Кейін ізденіс төбеге бірінші рет жеткен кезде, ол 1-күйге өтеді. Төбеден шығатын барлық төбелерін қарастырып болған кезде, ол 2-күйге ауысады.

Егер графта цикл болса, біз оны іздеу процесінде табамыз. Өйткені ерте ме, кеш пе, 1-күйдегі төбеге бір қадаламыз. Осы жағдайда топологиялық сұрыптау мүмкін болмай қалады.

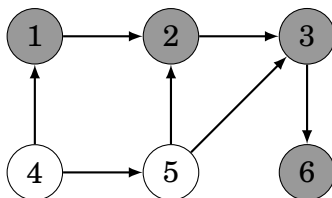
Егер графта цикл болмаса, біз графтағы төбелер 2-күйге ауысқан кезде әр төбені тізімге қоса отырып, топологиялық сұрыптаймыз.

1-мысал

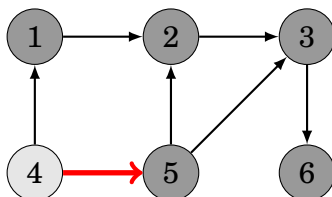
Осы мысалда графтағы 1-ден 6-ға дейінгі төбелерді өңдейміз:



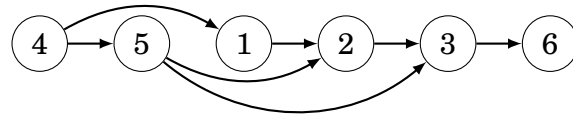
Алдымен 6-төбе өңделді, сондықтан оны тізімге қосамыз. Содан кейін 3, 2 және 1-төбелерді де тізімге қосамыз:



Қазіргі тізім – [6,3,2,1]. Келесі ізденіс 4-төбеден басталады:



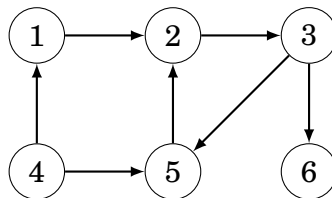
Түпкі тізім – [6,3,2,1,5,4]. Біз барлық төбелерді өтіп шықтық және оларды тізімге қостық. Енді топологиялық сұрыптау тізімін табу үшін соңғы тізімді кері төңкеру керек. Осылайша берілген графтың топологиялық сұрыптау тізімі – [4,5,1,2,3,6]:



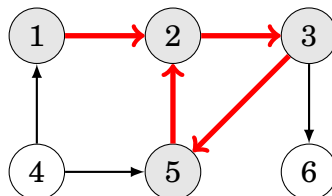
Топологиялық сұрыптаудың бірегей еместігін және графта бірнеше топологиялық сұрыптаулар болуы мүмкін екенін ескергеніміз жөн.

2-мысал

Енді графта цикл болғандықтан топологиялық сұрыптауды құра алмайтын мысалды қарастырайық:



Ізденіс келесі төбелерді қарастырады:



Ізденіс күйі 1-ге тең 2-төбеге жетеді. Бұл графтың циклды қамтитынын білдіреді. Бұл мысалда $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$ түріндегі цикл бар.

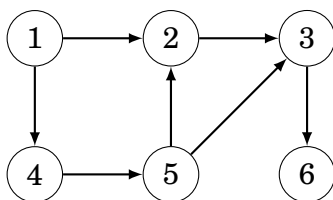
16.2 Динамикалық бағдарламалау

Егер бағытталған графта цикл болмаса, оған динамикалық бағдарламалау әдісін қолдануға болады. Мысалы, біз екі төбе арасындағы жолдарға қатысты келесі мәселелерді тиімді шеше аламыз:

- қанша әртүрлі жол бар?
- ең қысқа/ең ұзын жол қандай?
- жолдағы қырлардың минималды/максималды саны қандай?
- әр жолда қандай төбелер міндетті түрде кездеседі?

Жолдардың санын есептеу

Мысалы, 1-төбеден 6-төбеге дейінгі жолдардың санын есептейік:



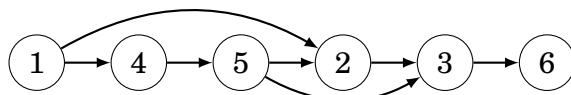
Төмендегідей үш жол бар:

- $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 6$
- $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$

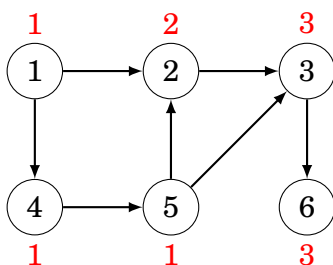
Біз 1-төбеден басталып x -төбеде аяқталатын жолдардың санын $\text{paths}(x)$ деп белгілейік. Ең басында $\text{paths}(1) = 1$ тең болады. Кейін $\text{paths}(x)$ жиымының басқа мәндерін есептеу үшін рекурсия қолдана аламыз.

$$\text{paths}(x) = \text{paths}(a_1) + \text{paths}(a_2) + \dots + \text{paths}(a_k)$$

бұл жерде a_1, a_2, \dots, a_k деп x -төбемен көршілес төбелер белгіленген. Граф ациклді болғандықтан $\text{paths}(x)$ жиымның мәндерін топологиялық сұрыптау тізімінің ретімен есептеп шығуға болады. Жоғарыдағы графтың топологиялық сұрыпталған тізімі:



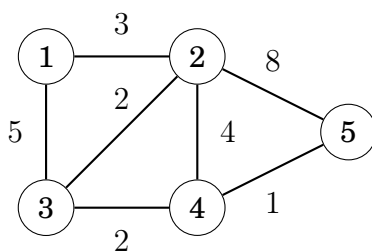
Сондықтан жолдардың саны төмендегідей болады:



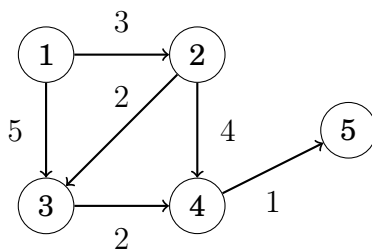
Мысалы, $\text{paths}(3)$ мәнін есептеу үшін біз $\text{paths}(2) + \text{paths}(5)$ формуласын қолдана аламыз. Себебі 2 және 5 төбелерінен 3-төбеге қыр бар. Мұндағы $\text{paths}(2) = 2$ және $\text{paths}(5) = 1$ тең болғандықтан, біз $\text{paths}(3) = 3$ мәнін осылай қорытындылаймыз.

Дейкстра алгоритмін кеңейту

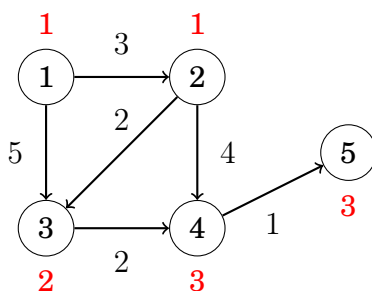
Бастапқы төбеден қалған төбелерге мейлінше қысқа жолмен жетудің мүмкін тәсілдерін көрсететін бағытталған, ациклді граф Дейкстра алгоритмінің жанама өніміне жатады. Бұл жаңа граф төбелерге бару үшін қолданылатын қырларды қалдырып, қалған қолданбайтын қырларды алып тастау арқылы құрылады. Яғни бастапқы төбеден түпнұсқа графтағы әр төбеге баратын ең қысқа жолдардың қырларын тауып, тек соларды қалдырады. Динамикалық бағдарламалауды сол жаңа графқа қолдануға болады. Мысалы, төмендегі графта



1-төбеден шығатын ең қысқа жолдар келесі қырларды қолданады:



Мысалы, біз динамикалық бағдарламалау арқылы 1-төбеден 5-төбеге дейінгі жолдардың санын таба аламыз:

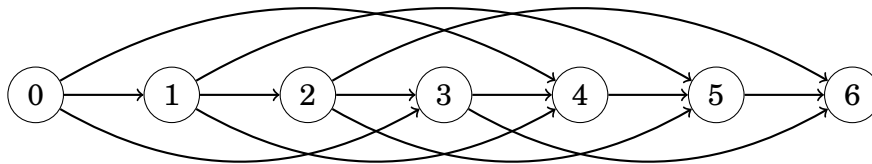


Есептерді граф тәріздес қарастыру

Шынында кез келген динамикалық бағдарламалау есебін бағытталған, ациклді граф тәрізді қарастыруға болады. Мұндай графта әрбір төбе динамикалық бағдарламалау мәніне сәйкес келеді және қырлар мәндердің бір-біріне тәуелділігін көрсетеді.

Мысалы, $\{c_1, c_2, \dots, c_k\}$ тиындарын қолдану арқылы қосындысы n болатын есепті қарастырайық. Бұл есепті шығару үшін граф қолдана аламыз. Графта

әр төбе тиындардың қосындысын белгілейді және қырлар қандай тиындар таңдау керек екенін белгілейді. Мысалы, $\{1, 3, 4\}$ тиындары бар және $n = 6$ тең болатын графтың көрінісі осындай болады:



Біз графтағы 0-төбеден n -төбеге дейінгі ең қысқа жолды табу арқылы n сомасын құрайтын минималды тиындар санын таба аламыз. Осы мысалда 0-ден 3-төбеге, содан соң 6-төбеге дейінгі жол ең қысқа жол болып табылады. Бұл жерде тек екі қыр қолданылған. Сонда 6 сомасын құру үшін минималды тиындар саны екі болмақ. Одан басқа, қарастырылған граф бізге 0-төбеден n -төбеге дейінгі жолдардың санын табу арқылы n суммасына жететін шешімдердің санын табуға көмектеседі.

16.3 Мирасқорлар графы

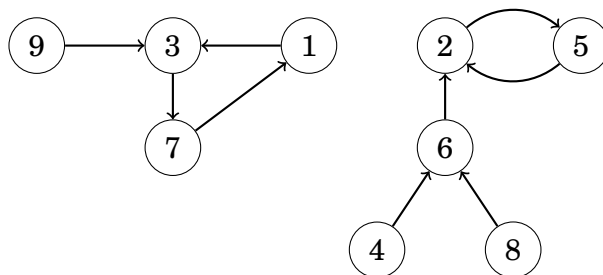
Тараудың қалған бөлігінде біз мирасқорлар графына (successor graph) тоқталамыз. Бұл графтарда әр төбенің шығыстың жарты дәрежесі дәрежесі 1-ге тең, яғни әр төбеден тек бір ғана қыр шығады. Мирасқорлар графы бір немесе бірнеше компоненттерден тұрады. Ал компоненттер бір циклды және оған бағытталған бірнеше жолдарды қамтиды.

Мирасқорлар графтарын кейде функционалды граф деп те атайды. Себебі кез келген мирасқорлар графы графтағы қырларды анықтайтын функцияға сәйкес келеді. Функцияның параметрі графтың төбесі болса, оның мәні мирасқор төбеге сәйкес келеді.

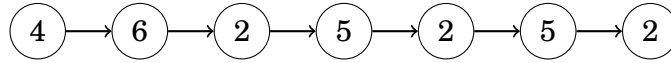
Мысалы, төмендегі функциямен

x	1	2	3	4	5	6	7	8	9
$\text{succ}(x)$	3	5	7	6	2	2	1	6	3

келесі графты құруға болады:



Мирасқорлар графындағы әр төбеде өзінің белгілі және бірегей мирасқоры болғандықтан, біз $\text{succ}(x, k)$ функциясын жария ете аламыз. Функция бізге x төбесінен бастап, алға k қадам жасау арқылы жететін төбені береді. Мысалы, жоғарыдағы графта $\text{succ}(4, 6) = 2$ тең. Себебі біз 2-төбеге 4-төбеден 6 қадам жасау арқылы жетеміз.



$\text{succ}(x, k)$ мәнін есептеудің қарапайым жолы x төбесінен басталып, k қадам алға жүру. Бұл $O(k)$ уақыт алады. Дегенмен алдын ала өңдеу арқылы $\text{succ}(x, k)$ -тың әр мәнін $O(\log k)$ уақытта табуға болады.

Бұл жердегі идея екінші k дәрежесі u - дан басым болмайтындай $\text{succ}(x, k)$ - ның барлық мәндерін алдын ала есептеуге негізделеді.

Келесі рекурсияны пайдалана отырып, мұны тиімді орындауға болады:

$$\text{succ}(x, k) = \begin{cases} \text{succ}(x) & k = 1 \\ \text{succ}(\text{succ}(x, k/2), k/2) & k > 1 \end{cases}$$

Мәндерді алдын ала өңдеу $O(n \log u)$ уақыт алады. Себебі әр төбеге $O(\log u)$ мән есептеледі. Жоғары графтағы бірінші мәндер:

x	1	2	3	4	5	6	7	8	9
$\text{succ}(x, 1)$	3	5	7	6	2	2	1	6	3
$\text{succ}(x, 2)$	7	2	1	2	5	5	3	2	7
$\text{succ}(x, 4)$	3	2	7	2	5	5	1	2	3
$\text{succ}(x, 8)$	7	2	1	2	5	5	3	2	7
...									

Осы алдын ала есептеуден кейін $\text{succ}(x, k)$ - ның кез келген мәнін k - ны екі дәрежелі қосындыларға жіктеу арқылы табуға болады. Мысалы, егер біз $\text{succ}(x, 11)$ мәнін тапқымыз келсе, алдымен 11 -ді $11 = 8 + 2 + 1$ түрінде жіктейміз. Осыны қолдана отырып,

$$\text{succ}(x, 11) = \text{succ}(\text{succ}(\text{succ}(x, 8), 2), 1).$$

Мысалға бұрынғы графта:

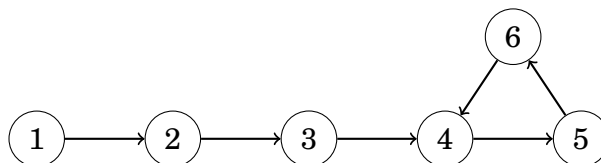
$$\text{succ}(4, 11) = \text{succ}(\text{succ}(\text{succ}(4, 8), 2), 1) = 5.$$

Осындай көрініс әрқашан да $O(\log k)$ қадамнан тұрады. Сондықтан $\text{succ}(x, k)$ мәнін есептеу $O(\log k)$ уақыт алады.

16.4 Циклді анықтау

Циклға апаратын жолдан ғана тұратын мирасқорлар графын қарастырайық. "Егер біз жүрісті бастапқы төбеден бастасақ, циклдегі бірінші төбе қандай болады және цикл қанша төбелерді қамтиды?" - деген сұрақ қоя аламыз.

Мысалы, төмендегі графта



біз жүрісті 1-төбеден бастаймыз. Циклге жататын бірінші төбе ол 4 және цикл 3 (4, 5 және 6) төбелерден тұрады.

Циклді табудың оңай жолына графты өтіп шығу және жеткен барлық төбелерді қадағалау жатады. Бір төбеге екінші рет жеткенде біз төбе циклдің бірінші төбесі екенін қорыта аламыз. Осы әдіс $O(n)$ уақытында жұмыс істейді және $O(n)$ жадысын қолданады.

Дегенмен циклді анықтауға арналған жақсырақ алгоритмдер де бар. Олардың уақытша тиімділігі $O(n)$ болып қалады, бірақ $O(1)$ жадысын ғана қолданады. Графта n үлкен болса, бұл алгоритмнің елеулі жақсарғанын байқатады. Әрі қарай біз осы қасиеттерге қол жеткізетін Флойд алгоритмін талқылаймыз.

Флойд алгоритмі

Флойд алгоритмі¹ графта a және b нұсқағыштары арқылы жылжиды. Екі нұсқағыш та жүрісті x төбесінен бастайды. Кейін әр қадамда a нұсқағышы бір қадам алдыға жылжиды, ал b екі қадам алдыға жылжиды. Осы үдеріс екі нұсқағыштар бір-бірімен кездескенге дейін жалғасады:

```
a = succ(x);
b = succ(succ(x));
while (a != b) {
    a = succ(a);
    b = succ(succ(b));
}
```

Осы сәтте a нұсқағышы k қадам жүреді, ал b нұсқағышы $2k$ қадам жүреді. Сондықтан циклдің ұзындығы k -ға бөлінеді. Енді циклге жататын бірінші төбені табу үшін a нұсқағышын x -төбеге қойып, қайтадан нұсқағыштарды кездескенге дейін біртіндеп жылжыту қажет.

```
a = x;
while (a != b) {
    a = succ(a);
    b = succ(b);
}
first = a;
```

Содан кейін циклдің ұзындығын төмендегідей етіп табуға болады:

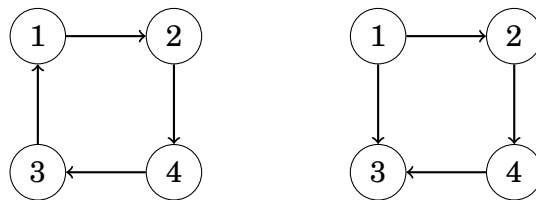
```
b = succ(a);
length = 1;
while (a != b) {
    b = succ(b);
    length++;
}
```

¹Алгоритм идеясын Р.В.Флойдка телиді [40], дегенмен алгоритмді шынымен Флойдтың ашқаны белгісіз.

17-тарау. Берік байланыстылық

Бағытталған графта қырлармен тек бір бағытта жүруге болады. Сондықтан граф байланысты болса да, кез келген екі төбе арасында жол болатынына кепілдік бермейді. Сондықтан байланыстықты талап етіп қана қоймайтын жаңа тұжырымдаманы анықтағанмыз жөн.

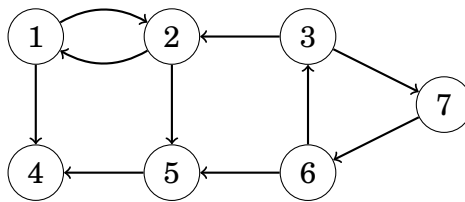
Егер әр төбеден графтағы басқа барлық төбелерге баратын жол болса, граф берік байланысты болады. Мысалы, төмендегі суреттің сол жағындағы граф – берік байланысты, ал оң жағындағы граф – берік байланысты емес.



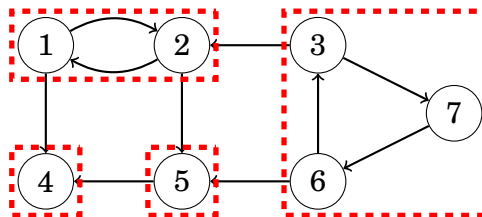
Оң жақтағы графтың берік байланысты еместігін 2-төбеден 1-төбеге баратын жолдың болмауына қарап байқай аламыз.

Графтың берік байланысты компоненттері графты мүмкіндігінше үлкен берік байланысты бөлшектерге бөледі. Берік байланысты компоненттер ациклді және бастапқы графтың терең құрылымын көрсететін компоненттерін құрайды.

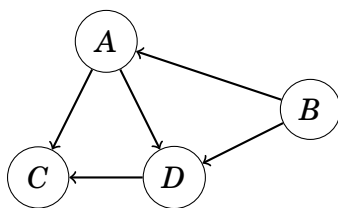
Мысалы, осы графтың



берік байланысты компоненттері келесідей:



Сәйкес компоненттер графы келесідей:



Компоненттер: $A = \{1, 2\}$, $B = \{3, 6, 7\}$, $C = \{4\}$ және $D = \{5\}$.

Компоненттерден тұратын граф ациклды, бағытталған болады. Сондықтан бұл графпен жұмыс істеу оңайырақ. Граф цикл қамтымағандықтан әрқашан да графты топологиялық сұрыптауға немесе 16-тарауда талқыланған динамикалық бағдарламалау әдістерін қолдануға болады.

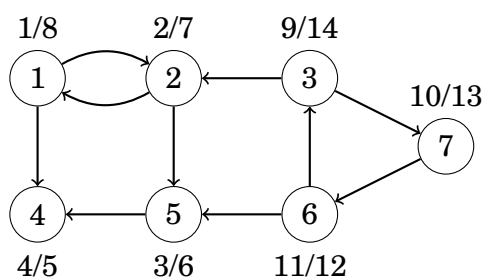
17.1 Косараджу алгоритмі

Косараджу алгоритмі¹ – бағытталған графтың берік байланысты компоненттерін табудың тиімді әдісі. Алгоритм тереңдігі бойынша екі ізденісті қолданады. Біріншісі арқылы графтың құрылымына сәйкес төбелердің тізімін құраса, екіншісі арқылы берік байланысты компоненттерді құрайды.

1-ізденіс

Косараджу алгоритмінің бірінші бөлімінде төбелердің тізімін тереңдігі бойынша ізденістің жүру ретімен құрайды. Алгоритм төбелерді біртіндеп өтіп, қандай да бір төбе әлі өңделмеген болса, сол төбеден тереңдігі бойынша ізденісті бастайды. Графтың әр төбесі тізімге өңделгеннен кейін қосылады.

Мысалдағы граф төбелерін осы ретпен өңдейді:



Бұл жерде x/y белгісі төбені өңдеу x уақытында басталып, y уақытында аяқталғанын білдіреді. Сәйкесінше, тізім келесідей болмақ:

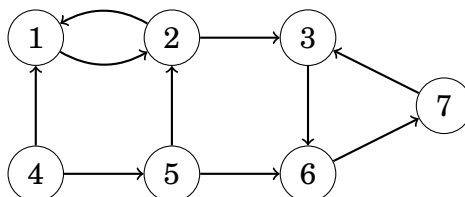
¹Бұл сілтемеге [41] бағынсақ, С.Р.Косараджу бұл алгоритмді 1978 жылы ойлап тапты, бірақ оны жарияламады. 1981 жылы дәл сол алгоритмді М.Шәрир қайта ашып, жариялады [42].

төбе	өңдеу уақыты
4	5
5	6
2	7
1	8
6	12
7	13
3	14

2-ізденіс

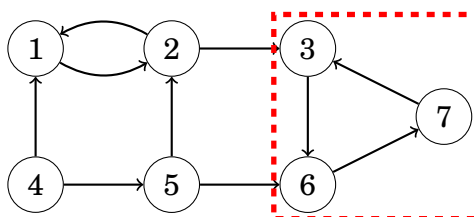
Косараджу алгоритмінің екінші бөлімінде графтың берік байланысты компоненттері құрылады. Алдымен алгоритм графтағы әр қырдың бағытын ауыстырады, яғни қыр бұрын a -төбесінен b -төбесіне бағытталса, енді b -төбесінен a -төбесіне қарай бағытталады. Бұл екінші іздеу кезінде берік байланысты компоненттерді әрқашан табатындығымызға толықтай кепілдік береді.

Мысалда қырларының бағытын ауыстырғаннан кейінгі граф берілген:



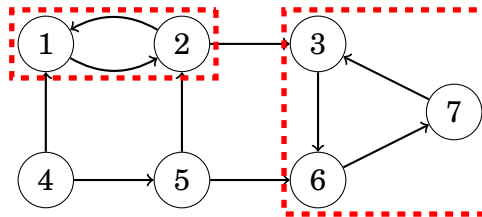
Содан кейін, алгоритм бірінші бөлімде құрылған төбелердің тізімін кері ретпен өтеді. Егер төбе компонентке жатпаса, алгоритм жаңа компонент құрып, тереңдігі бойынша ізденісті бастайды. Бұл ізденісте өтетін барлық төбелерді қазіргі құрылып жатқан компонентке қосады.

Мысалдағы графтың бірінші компоненті 3-төбеден басталады:

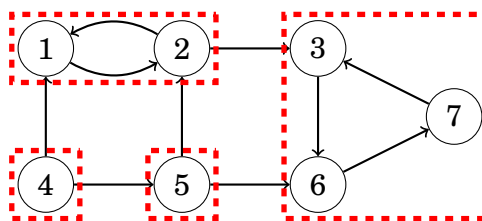


Барлық қырлардың бағытын ауыстырғандықтан компонент графтың басқа бөліктеріне "ағып кетпейді".

Тізімдегі келесі төбелерге 6 және 7-төбелер жатады. Бірақ олар бұрыннан компоненттің құрамында болғандықтан, келесі компонент 1-төбеден басталады:



Соңында алгоритм 5 және 4-төбелді өңдейді. Олар соңғы берік байланысты компонентті құрайды.



Алгоритмнің уақытша күрделілігі – $O(n + m)$. Себебі алгоритм тереңдігі бойынша екі ізденісті қамтиды.

17.2 2SAT есебі

Берік байланыстылықтың 2SAT есебіне де қатысы бар¹. Төмендегі есепте бізге логикалық формула беріледі:

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_m \vee b_m),$$

мұндағы a_i және b_i екеуі де логикалық айнымалы (x_1, x_2, \dots, x_n) немесе логикалық айнымалының терісі $(\neg x_1, \neg x_2, \dots, \neg x_n)$. "∧" және "∨" символдары "конъюнкция" (биттік және) және "дизъюнкция" (биттік немесе) логикалық операторларын белгілейді. Есепте берілген тапсырма – әр айнымалыға формула ақиқат(true) болатындай мән беру немесе бұл мүмкін емес екенін анықтау.

Мысалы, осы формулалар үшін

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

айнымалыларға төмендегідей мәндер берілсе, ақиқат болады :

¹Мұнда ұсынылған алгоритм осы жерде таныстырылған болатын [43]. Сондай-ақ қайта іздеу (backtracking) алгоритміне негізделген тағы бір танымал сызықтық уақыттағы күрделі алгоритм бар [44].

$$\begin{cases} x_1 = \text{false} \\ x_2 = \text{false} \\ x_3 = \text{true} \\ x_4 = \text{true} \end{cases}$$

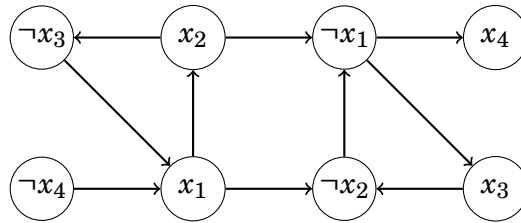
Дегенмен мына формулада

$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

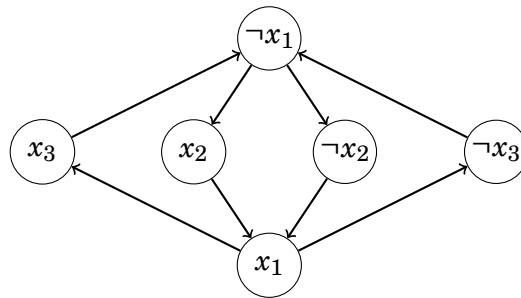
айнымалыларға қандай мән берсе де жалған болады. Мұның себебі x_1 мәнін қарама-қайшылықтарсыз таңдай алмауымызда жатыр. Егер x_1 жалған болса, x_2 және $\neg x_2$ де ақиқат болуы қажет еді, бірақ ол мүмкін емес. Сонымен қатар, егер x_1 ақиқат болса x_3 және $\neg x_3$ те ақиқат болуы қажет еді, бірақ бұл да мүмкін емес.

2SAT есебін граф сияқты қарастыруға болады. Бұл графта төбелер x_i және теріс $\neg x_i$ айнымалыларына сәйкес, ал қырлар айнымалылар арасындағы арақатынасты белгілейді. Әр $(a_i \vee b_i)$ жұбы екі қыр жасайды: $\neg a_i \rightarrow b_i$ және $\neg b_i \rightarrow a_i$. Бұл a_i орындалмаса, b_i орындалуы қажет немесе керісінше b_i орындалмаса, a_i орындалуы қажет деген арақатынасты білдіреді.

L_1 формуласының графы:



Ал L_2 формуласының графы:

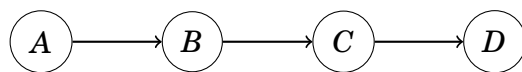


Графтың құрылымы айнымалыларға формуланы ақиқат ететіндей мән беруге болатынын немесе болмайтындығын көрсетеді. Егер әр x_i және $\neg x_i$ төбелері бөлек берік байланысты компоненттерге жатса ғана, формуланы ақиқат деуге болады. Егер бір берік байланыстылық компонентінде x_i және $\neg x_i$ қамтылса, граф x_i -төбеден $\neg x_i$ -төбеге және $\neg x_i$ -төбеден x_i -төбеге апаратын жолдардан тұрады. Сондықтан олар бір уақытта тең ақиқат болулары қажет, бірақ бұл мүмкін емес.

L_1 формуласының графында ешқандай x_i және $\neg x_i$ жұбы бір берік байланысты компонентке жатпайды. Сондықтан шешімі болады. Ал L_2 формуласының графында барлық төбелер бір берік байланысты компонентке жатады, сол себепті де шешімі жоқ.

Егер шешім бар болса, айнымалының мәндерін кері топологиялық сұрыптау ретімен компоненттер графының төбелерін өту арқылы табуға болады. Біз әрбір қадамда өңделмеген компонентке бағытталған қырды қамтымайтын компонентті қарастырамыз. Егер компоненттердегі айнымалыларға мән берілмесе, олардың мәндері компоненттегі мәндерге сәйкес анықталады, ал егер олардың мәндері әлдеқашан бар болса, олар өзгеріссіз қалады. Осы процесс әр айнымалыға мән тағайындалғанша жалғасады.

L_1 формуласының компонентті графы:



Бұл жердегі компоненттерге $A = \{\neg x_4\}$, $B = \{x_1, x_2, \neg x_3\}$, $C = \{\neg x_1, \neg x_2, x_3\}$ және $D = \{x_4\}$ жатады. Шешімді құрып жатқанда, алдымен D компонентін өңдейміз. Сол кезде x_4 айнымалысына ақиқат мәні беріледі. Кейін біз C компонентін өңдейміз. Бұл жерде x_1 және x_2 айнымалысы жалған болып, x_3 айнымалысы ақиқат болады. Барлық айнымалыға мән беріледі. Сондықтан қалған A және B компоненттері айнымалылардың мәндерін өзгертпейді.

Графтың арнайы құрылымы болғандықтан бұл әдістің тиімді болатындығын есте сақтағанымыз жөн, себебі егер x_i -төбесінен x_j -төбесіне және x_j -төбесінен $\neg x_j$ -төбесіне жол болса, мұндай жағдайда x_i төбесі ешқашан ақиқат бола алмайды. Себебі мұндай жағдайда $\neg x_j$ -төбесінен $\neg x_i$ -төбесіне де жол болады. Демек x_i және x_j жалған болады.

Бұдан қиынырақ есепке 3SAT есебі жатады. Бұл есепте формуланың әр бөлімі төмендегідей түрден тұрады: $(a_i \vee b_i \vee c_i)$. Есеп NP-қиын есептерге жататындықтан, шешуін тиімді табатын алгоритм әлі белгісіз болып тұр.

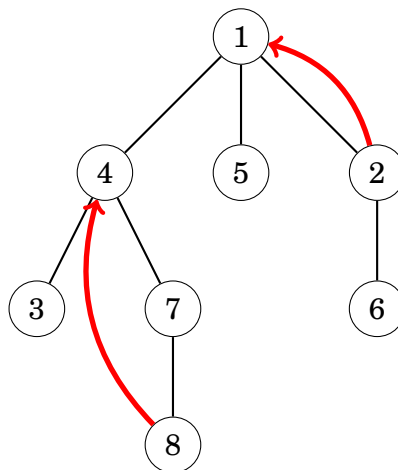
18-тарау. Дарақтағы сұратымдар

Бұл тарауда түбірлі дарақтағы жолдар мен ішдарақтардағы сұратымдарды өңдейтін әдістер талқыланады. Мысалы, оларға мынадай сұратымдар жатады:

- төбенің k -бабасы қай төбе?
- төбенің ішдарағындағы мәндер қосындысы неге тең?
- екі төбе арасындағы жол мәндерінің қосындысы неге тең?
- екі төбенің жақын арадағы ата-тегі қай төбе?

18.1 Бабалардың ізденісі

Түбірлі дарақта x төбесінің k -бабасы дегеніміз x -төбеден k қадам жоғарыға жүретін жолда кездесетін төбе. $\text{ancestor}(x, k)$ деп біз x -төбенің k -бабасын белгілейік (немесе сәйкес бабасы болмаған жағдайда 0 дейік). Мысалы, келесі дарақта $\text{ancestor}(2, 1) = 1$ және $\text{ancestor}(8, 2) = 4$.



Кез келген $\text{ancestor}(x, k)$ мәнін есептеудің оңай жолы – дарақта k рет үстіге қарай жүру. Дегенмен, бұл әдістің уақытша күрделілігі $O(k)$ болғандықтан, баяу саналады. Себебі n төбесі бар графта төбелер бір тізбекте орналасуы мүмкін. Солайша, уақытша күрделілігі $O(n)$ болып кетуі ықтимал.

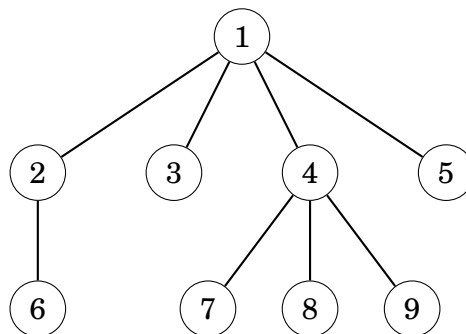
Бақытымызға орай, 16.3 тарауында талқыланған әдіс арқылы кез келген $\text{ancestor}(x, k)$ мәнін алдын ала өңдеуден кейін $O(\log k)$ уақыт ішінде табуға болады. Бұл жердегі идея k саны екінің дәрежесі болатындай барлық $\text{ancestor}(x, k)$ мәндерін есептеп шығуға негізделеді. Мысалы, жоғарыдағы дараққа келесі мәндер берілген:

x	1	2	3	4	5	6	7	8
$\text{ancestor}(x, 1)$	0	1	4	1	1	2	4	7
$\text{ancestor}(x, 2)$	0	0	1	0	0	1	1	4
$\text{ancestor}(x, 4)$	0	0	0	0	0	0	0	0
...								

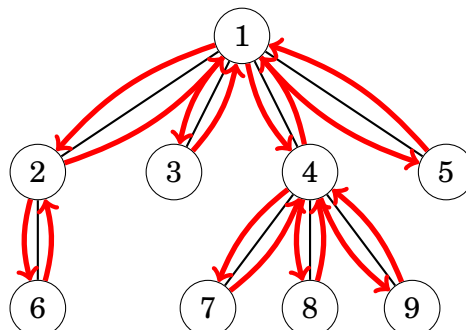
Әр төбе үшін $O(\log n)$ мән есептелгендіктен, алдын ала өңдеу $O(n \log n)$ уақыт алады. Бұдан соң $\text{ancestor}(x, k)$ -ның кез келген мәнін k -ның екі дәрежелі қосындыларына жіктеу арқылы табуға болады.

18.2 Ішдарақтар және жолдар

Дарақ айналымының жиымы түбірлі дарақтың төбелерін тереңдігі бойынша ізденіс арқылы жүру реттілігінде сақтайды. Мысалға төмендегі дарақты алайық:



тереңдігі бойынша ізденіс келесі ретпен жасалады:



Сол себепті дарақ айналымының жиымы төмендегідей болмақ:

1	2	6	3	4	7	8	9	5
---	---	---	---	---	---	---	---	---

Ішдарақ сұратымдары

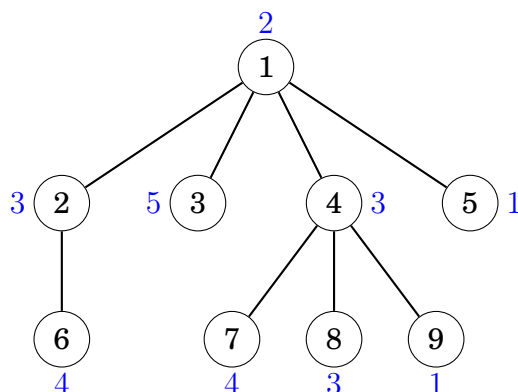
Дарақтың әр ішдарағы дарақ айналымының жиымындағы бір ішжиымға сәйкес келеді. Бұл ішжиымдағы бірінші элемент – ішдарақтың түбірі. Мысалы, келесі ішдарақ 4-төбе ішдарағының төбелерін қамтиды:

1	2	6	3	4	7	8	9	5
---	---	---	---	---	---	---	---	---

Осы дерек арқылы біз ішдараққа байланысты сұратымдарды тиімді өңдей аламыз. Мысалы, әр төбе үшін бір сан бекітілген есепті қарастырайық. Бұл есепте біз келесі сұратымдармен жұмыс жасаймыз:

- төбедегі санды басқа санмен жаңарту
- төбенің ішдарағындағы мәндер қосындысын есептеу

Көк сандармен төбелердің мәндері берілген дарақты қарастырайық. Мысалы, 4-төбе үшін ішдарағының қосындысы – $3 + 4 + 3 + 1 = 11$.



Бұл есепті шығару үшін біз әр төбе үшін оның идентификаторы, ішдарағының өлшемі мен оның мәні сақталатын дарақ айналымының жиымын құрамыз. Мысалы, жоғарыда келтірілген дарақтың жиымы келісідей болмақ:

төбе идентификаторы

ішдарақтың өлшемі

төбенің мәні

1	2	6	3	4	7	8	9	5
9	2	1	1	4	1	1	1	1
2	3	4	5	3	4	3	1	1

Бұл жиымды қолдана отырып, кез келген ішдарақтың қосындысын алдымен ішдарақтың өлшемін, кейін ішдараққа сәйкес төбелердің мәндерін табу арқылы есептей аламыз. Мысалы, 4-төбе ішдарағының қосындысын осылай табуға болады:

төбе идентификаторы

ішдарақтың өлшемі

төбенің мәні

1	2	6	3	4	7	8	9	5
9	2	1	1	4	1	1	1	1
2	3	4	5	3	4	3	1	1

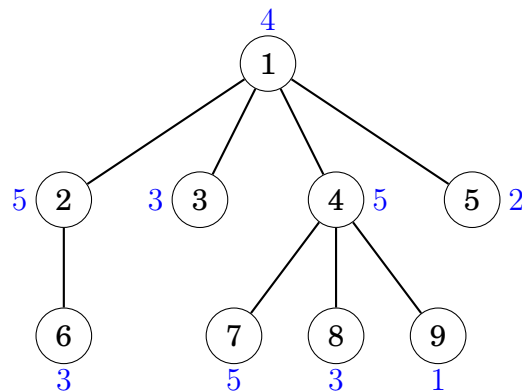
Сұратымдарға оңтайлы жауап беру үшін төбелердің мәнін бинарлы индекстелген дарақта немесе кесінділер дарағында сақтасақ, жеткілікті. Содан кейін біз мәндерді жаңартуды да, мәндердің қосындысын есептеуді де $O(\log n)$ уақытта жүзеге асыра аламыз.

Жол сұратымдары

Дарақ айналымының жиымын қолдану арқылы түбірден дарақтағы кез келген төбеге дейінгі жолдағы мәндердің қосындысын тиімді есептей аламыз. Келесі сұратымдарды қолдайтын есепті қарастырайық:

- төбедегі мәнді ауыстыру
- кез келген төбеден түбірге дейінгі мәндердің қосындысын есептеп шығу

Мысалы, төменде көрсетілген дарақтағы 7-төбеден түбірге дейінгі мәндердің қосындысы – $4 + 5 + 5 = 14$:



Біз бұл есепті бұрынғыдай шеше аламыз. Бірақ жиымның соңғы жолындағы мәндер енді түбірден сол төбеге дейінгі жол қосындысына тең болады. Мысалы, келесі жиым жоғарыда келтірілген дараққа сәйкес келеді:

төбе идентификаторы

ішдарақтың өлшемі

жол қосындысы

1	2	6	3	4	7	8	9	5
9	2	1	1	4	1	1	1	1
4	9	12	7	9	14	12	10	6

Егер төбедегі мән x -ке артса, ішдарақтағы барлық төбелердің қосындылары да x -ке артады. Мысалы, егер 4-төбенің мәнін 1-ге арттырсақ, жиым төмендегідей болып өзгереді:

төбе идентификаторы

ішдарақтың өлшемі

жол қосындысы

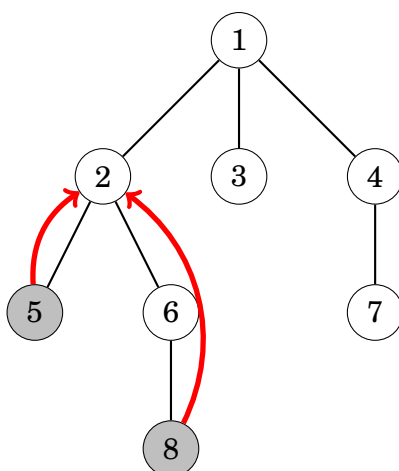
1	2	6	3	4	7	8	9	5
9	2	1	1	4	1	1	1	1
4	9	12	7	10	15	13	11	6

Екі операцияны да қолдау үшін біз диапазондағы барлық мәндерді арттыра отырып, бір мән алуымыз қажет. Мұны бинарлы индекстелген дарақты немесе кесінділер дарағын қолдану арқылы $O(\log n)$ уақыт ішінде іске асыра аламыз.

18.3 Жақын арадағы ата-тек

Екі төбенің жақын арадағы ата-тегі дегеніміз ішдарағы екі төбені де қамтитын ең төмен орналасқан төбе. Екі төбенің жақын арадағы ата-тегін тиімді табуға арналған есептер жиі кездеседі.

Мысалы, келесі дарақтағы 5 және 8 төбелерінің жақын арадағы ата-тегі – 2-төбе:



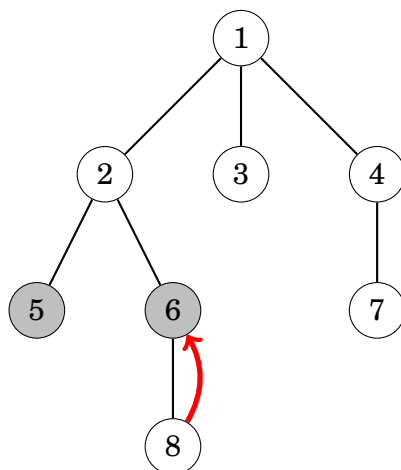
Енді біз жақын арадағы ата-текті табуға көмектесетін екі тиімді әдісті талқылаймыз.

1-әдіс

Біз бұған дейін дарақтағы кез келген төбенің k -бабасын табудың тиімді жолдарын қарастырдық. Осы әдісті қолдана отырып, жақын арадағы ата-текті табу есебін де екі бөлікке бөлуге болады:

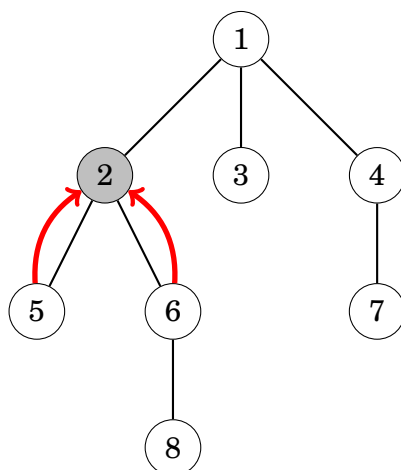
Есепті шығару барысында біз екі нұсқағышты қолданамыз. Бұл екі нұсқағыш басынан-ақ біз табуға тиісті жақын арадағы ата-текке қарай бағытталады. Алдымен нұсқағыштар бағытталған төбелердің бір деңгейде орналасқанын тексеріп аламыз. Егер олай болмаса, нұсқағыштардың біреуін жоғарыға жылжытамыз.

Үлгідегі дарақта біз 8-төбеде тұрған нұсқағышты бір деңгей жоғарыға көтеріміз. Осылайша ол 6-төбеге жетіп, екі нұсқағыш та бір деңгейде болады.



Кейін біз екі нұсқағыш бір төбеде кездесетіндей минималды қадам санын табамыз. Осы төбе жақын арадағы ата-тегі болады.

Мысалдағы графта нұсқағышты бір қадам жоғарыға жылжыту жеткілікті болып тұр. Сонда олар 2-төбеге нұсқайды. Бұл – жақын арадағы ата-тек:

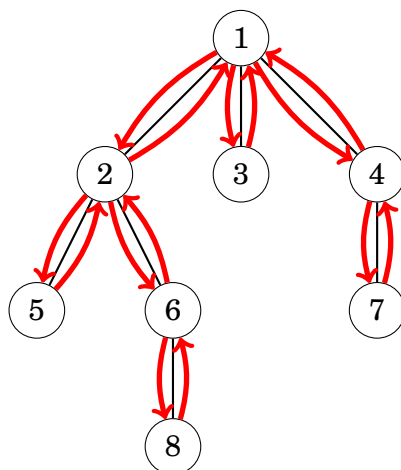


Алгоритмнің екі бөлігінің де алдын ала өңделген ақпаратты пайдаланып, $O(\log n)$ уақытында орындалуы мүмкін болғандықтан, жақын арадағы ата-тектерді $O(\log n)$ уақытында таба аламыз.

2-әдіс

Осы есепті басқа жолмен, атап айтқанда дарақ айналымының жиымын қолдану арқылы да ¹ шығаруға болады. Мұндағы идея тағы да тереңдігі бойынша ізденіс жасау арқылы төбелерді өтіп шығуға негізделеді.

¹Аталмыш жақын арадағы ата-текті табу алгоритмі осы жерде көрсетілген [23]. Бұл әдіс кейде (Euler tour technique) деп те аталады [45].



Дегенмен, біз осыған дейін қарастырған жиымнан басқа жиымды қолданамыз, төбені тек алғаш өткенде ғана емес, төбеден әр өткен сайын жиымға қосып отырамыз. Демек, k ұлдары бар төбе жиымда $k + 1$ рет кездеседі және жиымда жалпы $2n - 1$ төбе болады.

Біз жиымда екі мәнді сақтаймыз. Олар: төбенің идентификаторы және төбенің дарақтағы тереңдігі. Келесі жиым жоғарыдағы дараққа сәйкес келеді:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
төбе идентификаторы	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
тереңдігі	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Енді біз a және b төбелерінің жақын арадағы ата-тегін табу үшін жиымда a мен b арасындағы тереңдігі ең төменгі төбені ала аламыз. Мысалы, 5 және 8 төбелерінің жақын арадағы ата-тегін осы жолмен табуға болады:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
төбе идентификаторы	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
тереңдігі	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

↑

Бұл жерде 5-төбе 2-позицияда, 8-төбе 5-позицияда және 2-төбе 2...5 арасындағы 3-позицияда тереңдігі ең төменгі төбе болып отыр. Сондықтан 2-төбені 5-ші және 8-төбелердің жақын арадағы ата-тегі дей аламыз.

Демек екі төбенің жақын арадағы ата-тегін табу үшін аралықтың минималды сұратымын өңдеу жеткілікті болмақ. Жиым статикалық болғандықтан, біз бұл сұратымдарға $O(n \log n)$ уақыт алатын алдын ала өңдеуден кейін $O(1)$ уақытта да жауап бере аламыз.

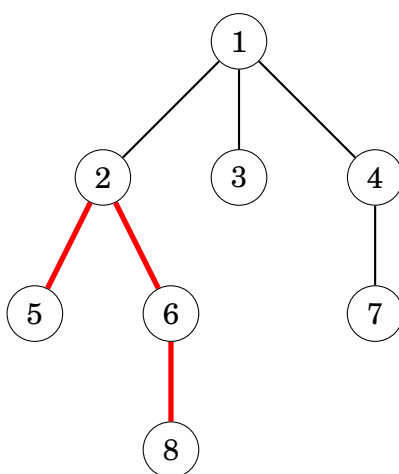
Төбелердің арақашықтығы

Екі төбенің арақашықтығы – төбелердің арасындағы жолдың ұзындығына тең. Төбелер арасындағы арақашықтықты есептеу есебін олардың жақын арадағы ата-тегін табу арқылы шешуге болады.

Алдымен біз кез келген бір төбені түбір ретінде аламыз. Содан кейін a мен b төбелерінің арақашықтығын төмендегі формуламен есептеп шығаруға болады:

$$\text{depth}(a) + \text{depth}(b) - 2 \cdot \text{depth}(c).$$

Мұндағы c – a мен b төбелерінің жақын арадағы ата-тегі және $\text{depth}(s)$ – s -төбесінің тереңдігін білдіреді. Мысалы, 5 пен 8 төбелерінің арақашықтығын қарастырайық:



5 және 8 төбелерінің жақын арадағы ата-тегі – 2-төбе. Төбелердің тереңдігі $\text{depth}(5) = 3$, $\text{depth}(8) = 4$ және $\text{depth}(2) = 2$ тең. Сондықтан 5 пен 8 төбелерінің арақашықтығы – $3 + 4 - 2 \cdot 2 = 3$.

18.4 Оффлайн алгоритмдер

Осыған дейін біз тек дарақ сұратымдарына арналған онлайн алгоритмдер туралы мысалдар келтірдік. Бұл алгоритмдер сұратымдардың бірінен соң бірін өңдей алады, яғни әрбір сұратымға келесі сұратымды өндегенге дейін жауап береді.

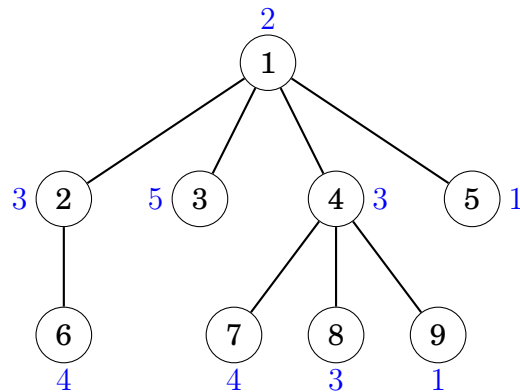
Дегенмен, көптеген есептер үшін сұратымдарға онлайн жауап алудың қажеті жоқ. Осы бөлімде біз оффлайн алгоритмдерге мән береміз. Бұл алгоритмдер сұратымдар жинағына кез келген ретпен жауап береді. Мұндай алгоритмді қолдану онлайн алгоритммен салыстырғанда көбінесе оңайырақ болады.

Деректер құрылымын біріктіру

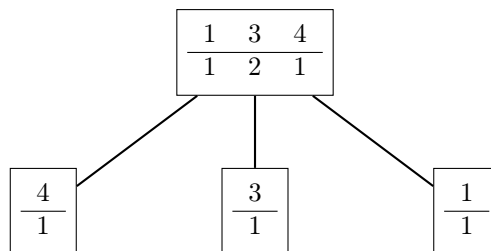
Оффлайн алгоритмді құрудың бір әдісі – дарақты тереңдігі бойынша өтіп, төбелерде деректер құрылымын сақтау. Әр s төбесі үшін оның ұлдарына

негізделген $d[s]$ деректер құрылымын құрамыз. Содан соң бұл деректер құрылымын қолдану арқылы біз s төбесіне байланысты барлық сұратымдарға жауап береміз.

Мысалға келесі есепті қарастырайық. Бізге әр төбеге бір мән тағайындалған дарақ беріліп, " s төбесінің ішдарағына жататын мәні x -ке тең төбелердің санын есептеу" сұратымдарына жауап беру тапсырылады. Мысалы, келесі дарақта 4-төбенің ішдарағында мәні 3-ке тең екі төбе бар.



Осы есепте сұратымдарға жауап беру үшін сөздік құрылымын қолдана аламыз. Мысалы, 4-төбе және оның ұлдарының сөздігі осылай болмақ (үстіндегі сан мәнді, ал астындағысы мәннің кездесетін санын белгілейді):



Егер осындай деректер құрылымын әр төбе үшін құрсак, біз барлық сұратымдарға оңай жауап бере аламыз. Себебі қандай да бір төбеге байланысты барлық сұратымдарға төбенің деректер құрылымын құрастырып болғаннан кейін бірден жауап беруге болады. Мысалы, жоғарыдағы 4-төбенің сөздік құрылымы бізге бұл төбенің ішдарағында мәні 3-ке тең 2 төбе бар екенін көрсетеді.

Дегенмен, барлық деректер құрылымын басынан бастап құрау тым баяу жүзеге асады. Оның орнына бастапқыда әр s төбесі үшін тек s төбесінің мәнін сақтайтын $d[s]$ деректер құрылымын құраймыз. Содан соң s -тің ұл төбелері арқылы өтіп, u төбесі s -тің ұлы болатын барлық $d[u]$ құрылымын және $d[s]$ -ті қосамыз.

Мысалы, жоғарыдағы дарақта 4-төбенің деректер құрылымы келесі деректер құрылымдарын біріктіру арқылы шығады:



Бұл жерде бірінші сөздік 4-төбенің басындағы деректер құрылымына және қалған үш сөздік 7, 8 және 9 төбелердің сөздіктеріне сәйкес.

s -төбесінде біріктіру келесі жолмен жүзеге асырылады: s төбесінің барлық ұлдар төбелерін өтіп шығып, әр u ұл төбесіне $d[u]$ мен $d[s]$ -ті қосамыз. Біз әрдайым $d[u]$ -дің ішіндегісін $d[s]$ -ке көшіреміз. Дегенмен $d[s]$ $d[u]$ -ден кіші болса, $d[s]$ пен $d[u]$ -дің ішіндегісі алдын ала алмастырылады. Нәтижесінде әрбір мән дарақты айналу процесінде тек $O(\log n)$ рет көшіріледі. Бұл алгоритмнің тиімділігіне кепілдік береді.

Екі a мен b деректер құрылымын тиімді ауыстыру үшін біз келесі кодты қолдана аламыз:

```
swap(a,b);
```

Егер a мен b C++ тілінің стандартты дерекханасының деректер құрылымы болса, жоғарыдағы код тұрақты уақытта жұмыс істейді.

Жақын арадағы ата-тек

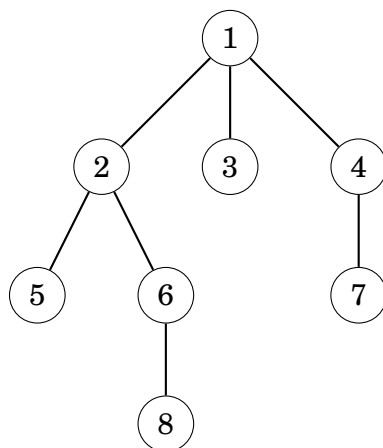
Сондай-ақ жақын арадағы ата-тектер сұратымдар жинағына жауап беретін оффлайн алгоритм бар¹. Бұл алгоритм қиылыспайтын жиындар құрылымына негізделген (15.2-тарауды еске түсіріңіз). Аталмыш алгоритмнің артықшылығы – бөлімде осыған дейін талқыланған алгоритмдерге қарағанда кодын жазу оңайырақ болуында.

Алгоритмге енгізу ретінде төбелер жұбының жиынтығы беріледі, ол өз кезегінде әр жұп үшін жақын арадағы ата-текті төбені тауып береді. Алгоритм дарақты тереңдігі бойынша өтіп шығады, оған қоса, төбелердің қиылыспайтын жиындарын сақтайды. Бастапқыда әр төбе жеке жиынға жатады. Әр жиын үшін сол жиынға жататын ең жоғарғы төбені сақтаймыз.

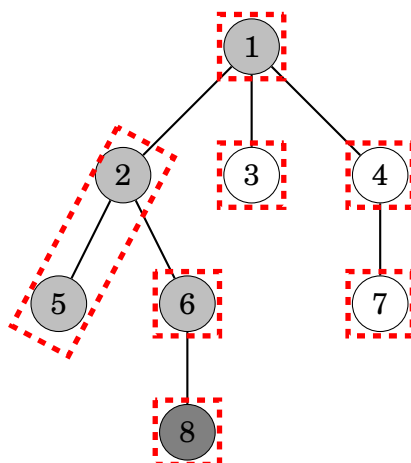
Алгоритм x төбесіне жеткенде x пен y төбелерінің жақын арадағы ата-тегін табуы тиіс барлық y төбелерінен өтіп шығады. Егер y төбесіне әлдеқашан барылған болса, алгоритм x және y мәндерінің жақын арадағы ата-тегі y жиынындағы ең жоғарғы төбе екенін хабарлайды. Кейін x төбені өңдеп болғаннан соң, алгоритм x жиынын және әкесінің жиынын біріктіреді.

Мысалы, (5,8) және (2,7) төбе жұптарының жақын арадағы ата-тектерін келесі дарақтан тапқымыз келеді:

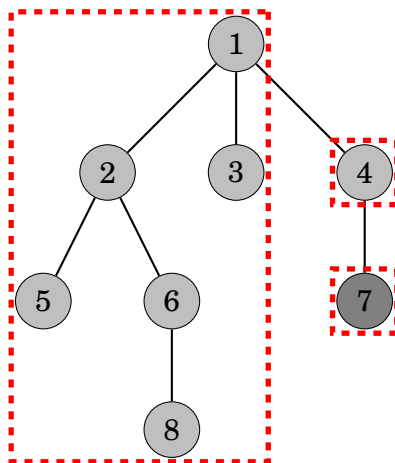
¹Бұл алгоритмді Р. Е. Таржан 1979 жылы жариялаған[46].



Төмендегі дарақтардың сұр төбелері барылған төбелерді білдіреді және төбелердің үзік сызықты топтары бір жиынға жатады. Алгоритм 8-төбеге жеткенде, ол 5-төбеге барғанын және оның жиынындағы ең жоғары төбе 2 екенін байқайды. Сондықтан 5 және 8 төбелерінің жақын арадағы ата-тегі 2-төбе болады.



Алгоритм кейінірек 7-төбеге барғанда, 2 мен 7 төбелерінің жақын арадағы ата-тегі болатын 1-төбені белгілейді.



19-тарау. Жолдар және шынжырлар

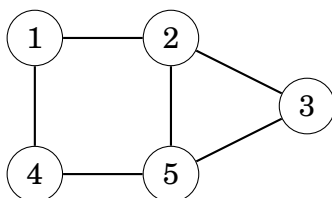
Бұл тарауда графтардағы жолдардың екі түріне назар аударылады:

- Эйлер жолы – әр қырдан дәл бір рет өтетін жол.
- Гамильтон жолы – әр төбеден дәл бір рет өтетін жол.

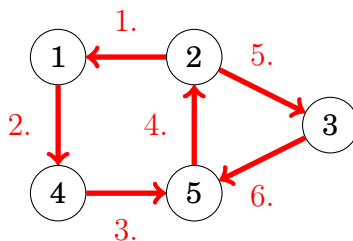
Эйлер және Гамильтон жолдары бір қарағанда ұқсас ұғымдар сияқты көрінгенімен, оларға байланысты есептер әртүрлі болып келеді. Графта Эйлер жолының бар-жоғын анықтайтын қарапайым ереже, сондай-ақ егер мұндай жол болған жағдайда, оны табудың тиімді алгоритмі бар. Ал Гамильтон жолының бар-жоғын тексеру NP-қиын есеп болып табылады және есепті шешу үшін қолданылатын тиімді алгоритм де әзірше белгісіз.

19.1 Эйлер жолы

Эйлер жолы¹ – графтағы әр қырдан дәл бір рет қана өтіп шығатын жол. Мысалы, төмендегі графта

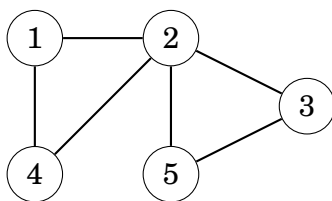


2-төбеден 5-төбеге баратын Эйлер жолы қамтылған:

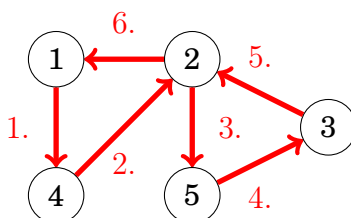


Эйлер шынжыры – бір төбеден басталатын және аяқталатын Эйлер жолы. Мысалы, төмендегі граф

¹Мұндай жолдарды Л. Эйлер 1736 жылы Кёнигсберг көпірі туралы танымал есебін шығару барысында зерттеген болатын. Осылайша граф теориясы пайда болған еді.



1-төбеден басталып, сол төбеде аяқталатын Эйлер шынжырын қамтиды:



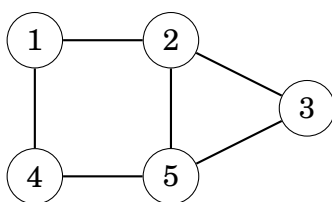
Графта бар болуына тексеріс

Графта Эйлер жолы мен шынжырының бар болуы төбелердің дәрежесіне тәуелді болып келеді. Эйлер жолы түзілу үшін келесі шарттылықтар сақталуы керек:

- Біріншіден, бағытталмаған графтағы барлық қырлар бір байланысқан компонентте жатуы керек;
- Екіншіден, әр төбенің дәрежесі жұп сан болуы немесе екі төбенің дәрежесі тақ сан, ал қалғандарының дәрежелері жұп сан болуы тиіс.

Бірінші жағдайда Эйлер жолы да, Эйлер шынжыры да бар деп саналады. Екінші жағдайда тақ дәрежелі төбелер Эйлер жолының басы немесе аяғы болады. Бұл жерде Эйлер жолы бар, бірақ ол Эйлер шынжыры бола алмайды.

Мысалы, төмендегі графта



1, 3 және 4 төбелерінің дәреже мәндері 2, ал 2 және 5 төбелерінің дәреже мәндері 3. Дәл екі төбенің дәрежелері тақ. Сондықтан 2 мен 5 төбелерінің арасында Эйлер жолы бар. Бірақ граф Эйлер шынжырын қамтымайды.

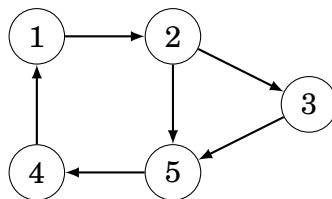
Бағытталған графта біз төбелердегі кірістің және шығыстың жарты дәрежелеріне мән береміз. Егер графтағы барлық қырлар бір байланысқан компонентте жатса және

- әр төбенің кірісінің жарты дәрежесі шығысының жарты дәрежесіне тең болса, немесе

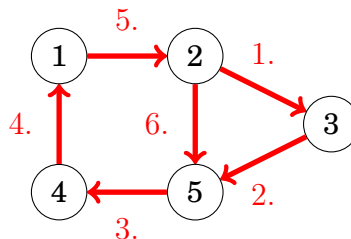
- бір төбенің кірісінің жарты дәрежесі шығысының жарты дәрежесінен 1-ге артық болса және басқа төбенің шығысының жарты дәрежесі кірісінің жарты дәрежесінен 1-ге артық болып, ал қалған төбелердің кірісінің және шығысының жарты дәрежелері тең болса, онда бағытталған графта Эйлер жолы бар деп есептейміз.

Бірінші жағдайда графта Эйлер жолы да, Эйлер шынжыры да бар. Екінші жағдайда графта шығысының жарты дәрежесі үлкенірек төбеден басталып, кірісінің жарты дәрежесі үлкенірек төбеде аяқталатын Эйлер жолы бар.

Мысалы, төмендегі графта



1, 3, және 4-төбелердің кірістерінің және шығыстарының жарты дәрежелері 1-ге тең, 2-төбенің кірісінің жарты дәрежесі – 1, ал шығысының жарты дәрежесі – 2, 5-төбенің кірісінің жарты дәрежесі – 2, ал шығысының жарты дәрежесі – 1. Демек, графта 2-төбеден 5-төбеге дейін Эйлер жолы бар:



Хирхольцер алгоритмі

Хирхольцер алгоритмі¹ – Эйлер шынжырын құрайтын тиімді әдіс. Алгоритм бірнеше бөлімнен тұрады, олардың әрқайсысы шынжырға жаңа қырларды қосады. Біз графта Эйлер жынжыры бар деп қарастырамыз, басқаша жағдайда Хирхольцер алгоритмі оны таба алмайды.

Алдымен алгоритм графтың кейбір қырларын (барлығын қамту міндетті емес) қамтитын шынжырды құрастырады. Ішшынжырларды біртіндеп қосу арқылы алгоритм шынжырды ұлғайта береді. Осы процесс барлық қырларды қосқанға дейін жалғасады.

Шынжырды кеңейту үшін біз шынжырдың ішінен шынжырға кірмеген шығыс қыры бар x төбесін табамыз. Содан соң тек қана әлі шынжырға кірмеген қырдан тұратын x төбесінен жаңа жол басталады. Ерте ме, кеш пе, бұл жол ішшынжыр басталған x төбесіне қайта оралады.

Егер граф тек Эйлер жолын қамтыса, біз осы графта Хирхольцер алгоритмін қолдана аламыз. Ол үшін біз графқа уақытша қосымша қыр қосып,

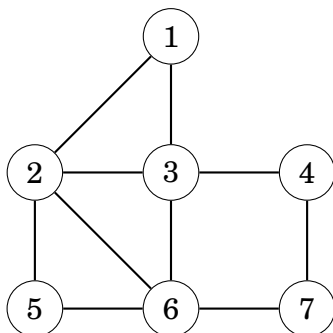
¹Алгоритм 1873 жылы Хирхольцер қайтыс болғаннан кейін жарияланды [47].

Хирхольцер алгоритмін жүргіземіз, ал шынжырды құрастырып болғаннан кейін сол қырды алып тастаймыз. Мысалы бағытталмаған графта біз екі тақ дәрежелі төбелердің арасына қосымша қыр қосамыз.

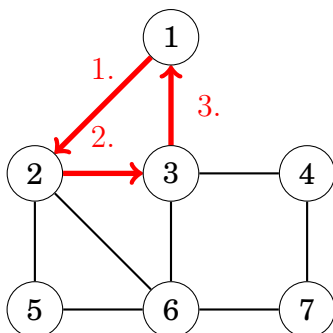
Әрі қарай Хирхольцер алгоритмі бағытталмаған граф үшін Эйлер шынжырын қалай құрастыратынын көреміз.

Мысал

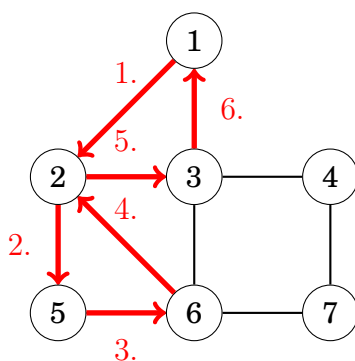
Келесі жолды қарастырайық:



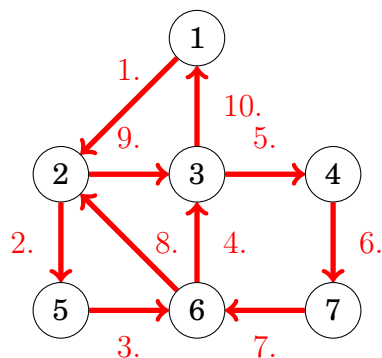
Алгоритм алдымен 1-төбеден басталатын шынжырды құрады делік. Шынжыр осындай болуы мүмкін: $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$



Содан кейін алгоритм $2 \rightarrow 5 \rightarrow 6 \rightarrow 2$ ішшынжырды бас шынжырға қосады:



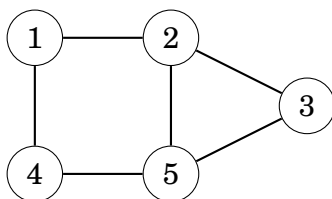
Ең соңында алгоритм $6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 6$ ішшынжырды бас шынжырға қосады:



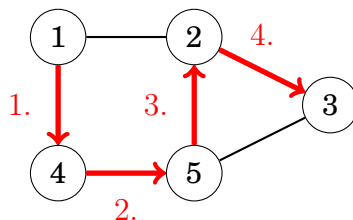
Көріп тұрғанымыздай, барлық қырларды шынжырға қосып шықтық. Осылайша Эйлер шынжырын құрастырдық.

19.2 Гамильтон жолдары

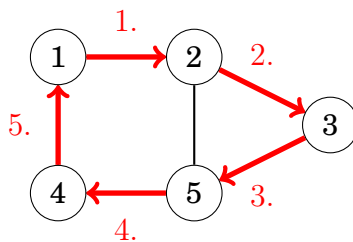
Гамильтон жолы – графтағы әр төбеге дәл бір рет баратын жол. Мысалы, осы граф



1-төбеден басталып 3-төбеде аяқталатын Гамильтон жолын қамтиды:



Егер Гамильтон жолы бір төбеден басталып, сол төбеден аяқталса, мұны Гамильтон шынжыры деп атаймыз. Жоғарыдағы граф 1-төбеден басталып, сол төбеде аяқталатын Гамильтон шынжырын қамтиды.



Графта бар болуына тексеріс

Графта Гамильтон жолы бар-жоғын тексеретін тиімді алгоритм жоқ және тапсырма NP-қиын есеп саналады. Дегенмен кейбір ерекше жағдайларда графта Гамильтон жолы бар екеніне сенімді бола аламыз.

Оны байқаудың қарапайым жолы манау: егер граф толық болса, яғни барлық төбелердің арасында қыр болса, онда Гамильтон жолы бар. Одан басқа төмендегі тәсілдерді де қолдануға болады:

- Дирак теоремасы бойынша, егер әр төбенің дәрежесі кем дегенде $n/2$ болса, графта Гамильтон жолы бар деп саналады.
- Оре теоремасы бойынша, егер әр көрші емес жұп дәрежелерінің қосындысы кем дегенде n болса, графта Гамильтон жолы бар деп саналады.

Егер графта көп қыр болса, олардың Гамильтон жолының бар екендігіне кепілдік беруі теоремалардың ортақ қасиетіне жатады. Бұл – ақылға қонымды жайт, өйткені графтың қырлары неғұрлым көп болса, Гамильтон жолын салу мүмкіндігі де соғұрлым көбейе түседі.

Құрылысы

Гамильтон жолының бар-жоғын тексерудің тиімді әдісі болмағандықтан, жолды тиімді құру әдісі де жоқ екені анық. Өйткені жолды салуға тырысу арқылы оның бар-жоғын оңай тексеруге де болар еді.

Гамильтон жолын іздеудің қарапайым жолы – Гамильтон жолын құратын барлық мүмкін жолдардан өтетін қайта іздеу алгоритмін (backtracking) пайдалану. Мұндай алгоритмнің уақытша күрделілігі кем дегенде $O(n!)$ құрайды, себебі n төбелерді $n!$ әртүрлі жолдармен өтуге болады.

Бұдан да анағұрлым тиімді шешім динамикалық бағдарламалауға негізделеді (10.5 тарауды еске түсіріңіз). Динамикалық бағдарламалаудың идеясы – $\text{possible}(S, x)$ функциясының мәндерін есептеу, бұл жердегі S – ішжиын, ал x – ішжиындағы төбелердің бірі. Функция S төбелеріне баратын және x төбесінен аяқталатын Гамильтон жолының бар-жоғын көрсетеді. Бұл шешімді $O(2^n n^2)$ уақытында жүзеге асыруға болады.

19.3 Де Брёйи тізбегі

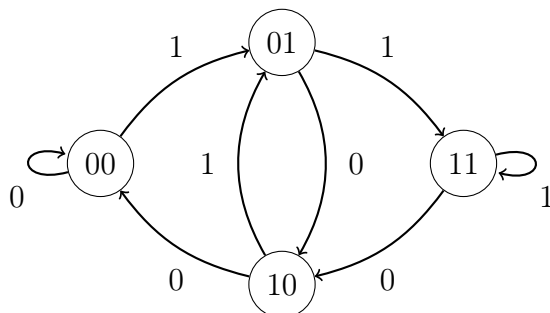
Де Брёйи тізбегі – ұзындығы n болатын әрбір жолды бір рет қамтитын, бекітілген k таңбалы әліпбиден тұратын жол.

Осы жолдың ұзындығы $k^n + n - 1$ формуласына тең болады. Мысалы, $n = 3$ және $k = 2$ болған жағдайдағы Де Брёйи тізбегі:

0001011100.

Осы жолдың ішжолдары – үш биттің барлық комбинациялары: 000, 001, 010, 011, 100, 101, 110 және 111.

Әрбір Де Брөйн тізбегі графтағы Эйлер жолына сәйкес келеді. Бұл жердегі идея әр төбесі ұзындығы $n - 1$ таңбадан тұратын жолды қамтитын және әрбір қыр жолға бір таңба қосатын графты құрастыруға негізделеді. Келесі граф негізгі идеяға сәйкес келеді:



Графтағы Эйлер жолы ұзындығы n болатын барлық жолдарды қамтиды. Ол бастапқы төбенің таңбаларынан және қырлардың барлық таңбаларынан тұрады. Бастапқы төбеде $n - 1$ таңба, ал қырларда k^n таңба бар. Сонда жолдың ұзындығы $k^n + n - 1$ болады.

19.4 Ат сапары

Шахмат атының бағдарғысы – ат жүрісінің $n \times n$ шахмат тақтасындағы шахмат ережелеріне сәйкес әр шаршыға тура бір рет барылатын тізбегі. Егер ат соңында бастапқы төбеге қайтып келсе, аттың сапарын жабық сапар дейміз. Ал басқа жағдайда ашық сапар деп айтамыз.

Мысалы, 5×5 тақтада ашық ат сапары бар:

1	4	11	16	25
12	17	2	5	10
3	20	7	24	15
18	13	22	9	6
21	8	19	14	23

Аттың сапары графтағы Гамильтон жолына сәйкес құрылған. Графтағы төбелер тақтадағы шаршыларға сәйкес келеді. Ал ат бір төбеден екінші төбеге шахмат заңдары бойынша бара алса ғана, сол екі төбе арасында жол бар болмақ.

Аттың сапарын құрудың қарапайым жолы – қайта іздеу алгоритмін қолдану. Аттың толық жолды тезірек табуы үшін бағдар беруге тырысатын эвристикалық тәсілдерді қолану арқылы да ізденісті одан да тиімді етуге болады.

Уарнсдорф ережесі

Уарнсдорф ережесі – аттың сапарын табатын қарапайым және тиімді эвристика.¹ Осы ережені қолдана отырып, ат сапарын үлкен тақтайда да тиімді құрастыруға болады. Мұндағы идея атты әрдайым әлі жүрілмеген шаршылардың минималды санына баруға мүмкіндік беретін шаршыға қоюға негізделеді.

Мысалы, төмендегі жағдайда атты қозғауға болатын бес шаршы бар ($a \dots e$ шаршалары):

1				a
		2		
b				e
	c		d	

Бұл жағдайда Уарнсдорф ережесі атты a -шаршысына қозғайды. Себебі бұл жүрісті таңдағаннан кейін, тек бір ғана қозғалыс жасау мүмкіндігі қалады. Ал басқаша таңдау жасағанда, ат үш рет жүріс жасау мүмкіндігін беретін шаршыға түсер еді.

¹Эвристика Уарнсдорфтың кітабында [48] 1823 жылы жазылған. Сондай-ақ ат сапарын табудың полиномдық алгоритмдері де бар [49], бірақ олар қиынырақ.

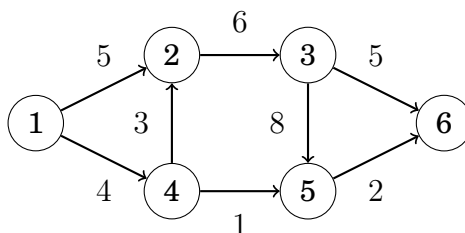
20-тарау. Ағындар мен қималар

Бұл тарауда төмендегі екі есепті қарастыратын боламыз:

- Максималды ағынды табу, яғни бір төбеден екінші төбеге максималды қанша ағын жібере алатынымызды анықтау.
- Минималды қиманы табу, яғни графта салмағы минималды болатын қай қырлар жиындысы екі төбенің арасын ажырататынын анықтау.

Аталған екі есепте де бағытталған, салмақталған граф пен екі арнаулы төбе берілген. Мұндағы бастау – кіріс қырлары жоқ төбе, саға — шығыс қырлары жоқ төбе.

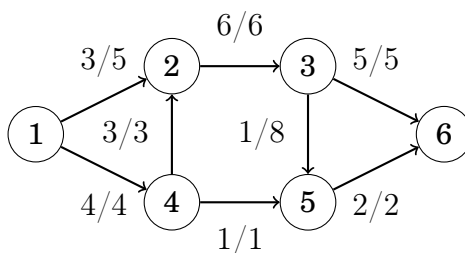
Өрнек ретінде бастауы 1-төбе, ал сағасы 6-төбе болатын төмендегі графты қолданамыз:



Максималды ағын

Ең максималды ағын есебіндегі бізге берілетін тапсырама – бастаудан сағаға ең максималды ағынды жіберу. Әр қырдағы салмақ – сол қыр арқылы қанша ағын өте алатынын шектеуші сыйымдылық. Әр аралық төбенің кіріс және шығыс ағындары бірдей болуы қажет.

Өрнектегі графтың максималды ағыны – 7. Ал төмендегі сурет ағынды қалай бағыттау керектігін көрсетеді:

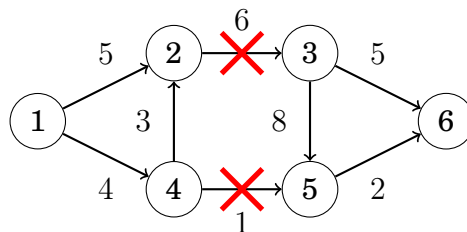


v/k нотациясы – k бірлік сыйымдылығы бар қырдан v бірлік ағын бағытталғанын көрсетеді. Ағынның өлшемі – 7, себебі бастау $3 + 4$ бірлік ағын жіберіп, саға $5 + 2$ ағынды қабылдайды. Бұл жағдайда ағын максималды екенін түсіну оңай, өйткені сағаға жетелейтін барлық қырлардың сыйымдылығы – 7.

Минималды қима

Минималды қима есебіндегі бізге берілетін тапсырма графта бастаудан сағаға жол болмайтындай етіп қырлар жиындысын өшіруге және өшірілген қырлардың салмағы минималды болуына арналады.

Өрнек графындағы қиманың минималды өлшемі – 7. $2 \rightarrow 3$ пен $4 \rightarrow 5$ қырларын өшіру жеткілікті:



Қырларды өшіргеннен кейін бастаудан сағаға дейін ешқандай жол болмайды. Қиманың өлшемі – 7, ал өшірілген қырлардың салмақтары – 6 мен 1. Жалпы салмағы 7-ден кем болатын өшіруге келетін қырлар жиындысы графта болмағандықтан қима минималды деп саналады.

Жоғарыдағы өрнекте максималды ағын мен минималды қиманың бірдей болуы жай сәйкестік емес. Шын мәнінде олар әрқашан тең болады. Сондықтан бұл ұғымдар әртүрлі болып көрінгенімен, өте тығыз байланысты.

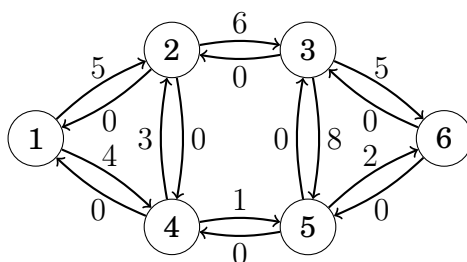
Алдағы уақытта біз максималды ағын мен минималды қиманы табатын Форд-Фалкерсон алгоритмін талқылайтын боламыз. Алгоритм олардың не себепті тең екенін түсінуімізге көмектеседі.

20.1 Форд–Фалкерсон алгоритмі

Форд–Фалкерсон алгоритмі [50] графтағы максималды ағынды табады. Алгоритм бос ағыннан басталады, әр қадамда бастаудан сағаға дейін көбірек ағын өндіретін жолды іздейді. Ақырында алгоритм ағынды үлкейте алмаса, максималды ағын табылады.

Алгоритм әр негізгі қырға оған кері бағытта қыр болатын графтың арнайы көрінісін қолданады. Әр қырдың салмағы тағы қанша ағынның өте алатынын бейнелейді. Алгоритмнің басында негізгі қырлардың салмақтары сол қырлардың сыйымдылықтарына тең болса, кері қырлардың салмақтары нөлге тең болады.

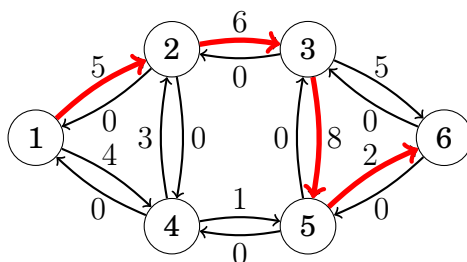
Өрнек графының жаңа көрінісі төмендегідей болады:



Алгоритм сипаттамасы

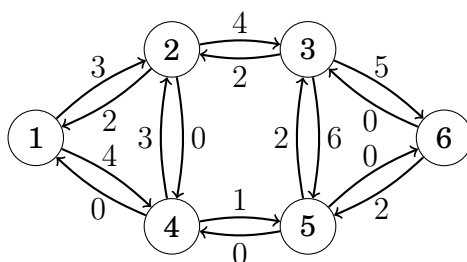
Форд-Фалкерсон алгоритмі бірнеше кезеңдерден тұрады. Әр кезеңде алгоритм барлық қырлардың салмақтары оң болатын бастаудан сағаға жол іздейді. Егер мұндай бірнеше жол болса, кез келгенін таңдай аламыз.

Мысалы, төмендегідей жолды таңдайық:



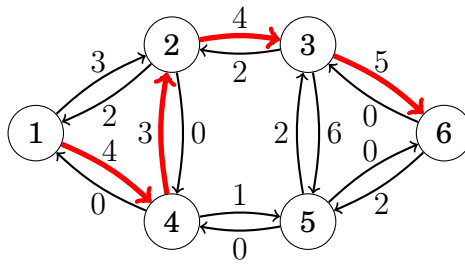
Жолды таңдағаннан кейін ағын x бірлікке артады, мұнда x – жолдағы ең кіші салмақ. Оған қоса жолдағы қырлардың салмақтарын x -ке азайтамыз және кері қырлардың салмақтарын x -ке арттырамыз.

Жоғарыдағы жолда қырлардың салмақтары – 5, 6, 8 және 2. Ең кіші салмақ – 2, жаңа граф төмендегідей:



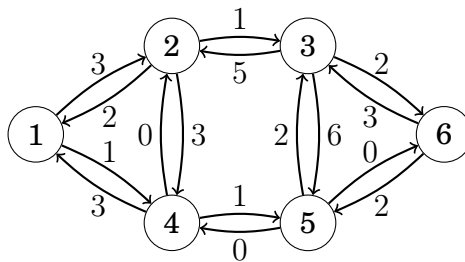
Идеясы ағынды арттыру арқылы болашақта қырлардан өте алатын ағынды азайтуға бағытталады. Екінші жағынан егер ағынды басқа жолмен өткізген оңтайлырақ болса, өткізген ағынды графтағы кері қырлар арқылы болдырмауға болады.

Алгоритм бастаудан сағаға қырлардың салмақтары оң болатын жол болғанша ағынды арттырады. Мысалдағы келесі жолымыз төмендегідей болады:

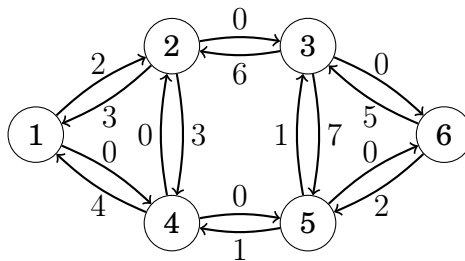


Осы жолдағы ең кіші қырдың салмағы – 3, демек жол ағынды 3-ке арттырады және осы жолды өңдегеннен кейін қорытынды ағын 5 болады.

Жаңа граф төмендегідей:



Максималды ағынға жетуіміз үшін әлі де 2 кезең өтуіміз қажет. Мысалы, біз $1 \rightarrow 2 \rightarrow 3 \rightarrow 6$ және $1 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 6$ жолдарын таңдай аламыз. Екеуі де ағынды 1-ге арттырады және қорытынды граф төмендегідей болады:



Бұдан кейін ағынды арттыру мүмкін емес, өйткені бастаудан сағаға дейінгі қырлардың салмағы оң болатын жол жоқ. Демек алгоритм тоқтайды және максималды ағын 7-ге тең болады.

Жолдар іздеу

Форд-Фалкерсон алгоритмі ағынды арттыратын қандай жолдарды алу керектігін атап өтпейді. Кез келген жағдайда алгоритм ерте ме, кеш пе, бір тоқтайды және максималды ағынды дұрыс табады. Бірақ алгоритмнің тиімділігі жолдардың қалай таңдалғанына тәуелді болады.

Жолдарды табудың қарапайым тәсілі – тереңдігі бойынша ізденісті (DFS) қолдану. Әдетте бұл жақсы жұмыс істейді, бірақ ең нашар жағдайда әр жол ағынды 1-ге арттыратындықтан, мұндай алгоритм баяу өтеді. Бақытымызға орай, төмендегі әдістердің бірін қолдану арқылы бұл жағдайдан құтыла аламыз:

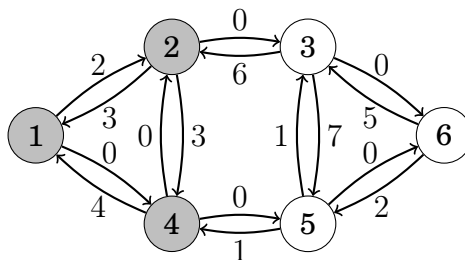
Эдмондс–Карп алгоритмі [51] жолдағы қырлардың саны ең аз болатын жолды таңдап отырады. Мұны тереңдігі бойынша ізденістің (DFS) орнына ені бойынша ізденіс (BFS) жасау арқылы орындауымызға болады. Сол арқылы ағынның жылдам өсу кепілдігін дәлелдей аламыз және алгоритмнің уақытша күрделілігі $O(m^2n)$ болмақ.

Ағынды масштабтау алгоритмі [52] тереңдігі бойынша ізденіс (DFS) арқылы жолдағы қырлардың салмағы кем дегенде шекті мән болатын жолдарды іздейді. Бастапқыда шекті мән үлкен сан, мысалы барлық қырлардың салмақтар қосындысы болуы мүмкін. Жол табылмаған жағдайда шекті мәнді әрдайым 2-ге бөлеміз. Алгоритмнің уақытша күрделілігі — $O(m^2 \log c)$, мұндағы c – бастапқы шекті мән.

Тәжірибеде ағынды масштабтау алгоритмін жазу жеңілірек, яғни жолдарды табу үшін тереңдігі бойынша ізденісті (DFS) қолдансақ жеткілікті. Екі алгоритм де әдетте бағдарламалау контексттерінде кездесетін есептер үшін жеткілікті деңгейде тиімді болады.

Минималды қима

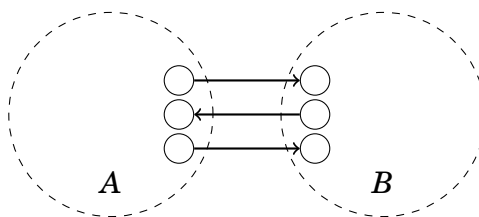
Форд–Фалкерсон алгоритмі максималды ағынды тапқаннан кейін, минималды қиманы да анықтайды. **A** деп бастаудан салмақтары оң қырлар арқылы жетуге болатын төбелер жиындысын белгілейік. Өрнектегі графта **A** 1, 2, және 4 төбелерінен тұрады:



Енді минималды қима **A**-дағы төбеден басталып, **A**-ның сыртындағы төбеде аяқталатын және максималды ағында сыйымдылығы толық қолданылған қырлардан тұрады. Жоғарыдағы графта сондай қырлар – $2 \rightarrow 3$ және $4 \rightarrow 5$, олар $6 + 1 = 7$ минималды қимасын береді.

”Неге алгоритмнен құрылған ағын максималды және қима минималды болады?” - деген сұрақ туындауы мүмкін. Оған мынадай түсіндірме береміз: графта кез келген қиманың салмағынан үлкен ағын болуы мүмкін емес. Демек ағын мен қиманың мәндері тең болса, онда ағын максималды, ал қима минималды болмақ.

Бастау **A**-да, саға **B**-да болатын және екі жиынның араларында қырлары бар қималарды қарастырайық:



Қиманың өлшемі – A мен B араларынан өтетін қырлардың қосындысы. Бұл графтағы ағынның жоғарғы шекті мәні, себебі ағын A мен B арасынан өтуі қажет. Осылайша максималды ағынның өлшемі кез келген қиманың өлшемінен кіші немесе тең болады.

Басқа жағынан қарасақ, Форд-Фалкерсон алгоритмі бір қиманың өлшеміне тең ағынды табады. Осылайша ағын максималды және қима минималды болуы керек.

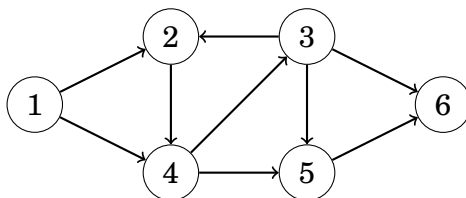
20.2 Қиылыспайтын жолдар

Граф есептерінің көбісін максималды ағын есебіне келтіріп шығаруымызға болады. Сондай есептердің біріне мысал келтірейік. Бізге бастау мен саға-сы бар бағытталған граф берілген. Тапсырма бойынша бастаудан сағаға апаратын қиылыспайтын жолдардың максималды санын есептеуіміз керек.

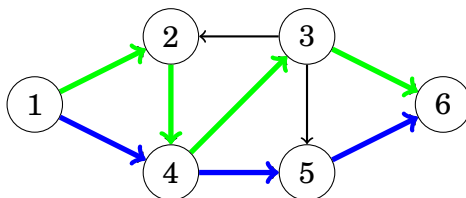
Қырлары қиылыспайтын жолдар

Алдымен бастаудан сағаға дейінгі қырлары қиылыспайтын жолдардың максималды санын есептейтін боламыз, яғни әр қыр ең көбі бір жолда ғана кездесетін жолдар жиынын құрастыруымыз керек.

Мысалға төмендегі графты қарастырайық:



Графтағы қырлары қиылыспайтын жолдардың максималды саны 2-ге тең. Біз $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 6$ және $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$ жолдарын таңдай аламыз:

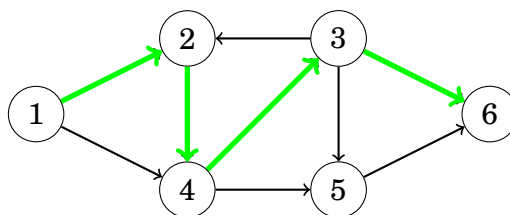


Егер граф қырларының сыйымдылығын бір деп алсақ, қырлары қиылыспайтын жолдардың максималды саны максималды ағынға тең болады. Максималды ағын құрастырылғаннан кейін бастаудан сағаға апаратын жолдарды ашкөз алгоритм арқылы қарастырып, қырлары қиылыспайтын жолдарды таба аламыз.

Төбелері қиылыспайтын жолдар

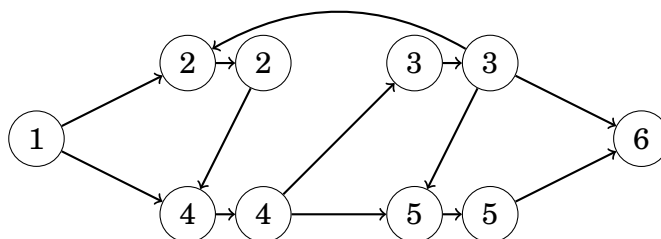
Келесі есепте бастаудан сағаға дейін төбелері қиылыспайтын жолдардың максималды санын есептеп көрейік. Есепте әр төбе бастау мен сағадан басқа жолда ең көбі бір рет кездеседі. Төбелері қиылыспайтын жолдардың саны қырлары қиылыспайтын жолдар санынан аз болуы мүмкін.

Мысалы, төмендегі графта төбелері қиылыспайтын жолдардың максималды саны – 1:

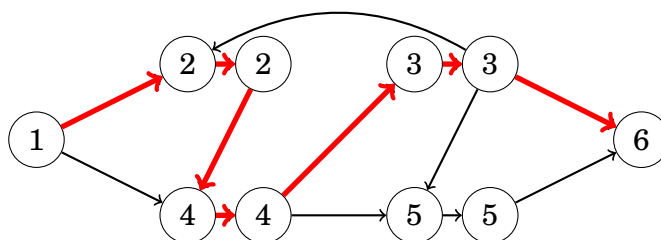


Бұл есепті де максималды ағын есебіне келтірсек болады. Әр төбе жолда ең көбі бір рет бола алғандықтан төбеден өтетін ағынды шектеуіміз қажет. Бұл үшін стандартты әдісті қолданып, әр төбені екі төбеге бөлеміз. Бірінші төбе бастапқы төбеден кіріс қырларды қамтыса, екінші төбе бастапқы төбеден шығыс қырларды қамтиды және бірінші төбеден екінші төбеге жаңа қыр түседі.

Графымыз төмендегідей ретпен өзгереді:



Графтағы максималды ағын:



Осылайша бастаудан сағаға дейінгі төбелері қиылыспайтын жолдардың максималды саны 1 болады.

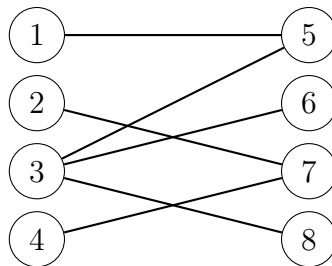
20.3 Максималды жұптасу

Максималды жұптасу есебінде тапсырма ретінде бағытталмаған графта қырлар арқылы қосылып, әр төбе ең көбі бір жұпта болатын максималды төбелер жұптарының санын табу беріледі.

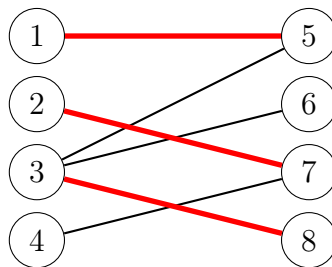
Жалпы графтарда максималды жұптасуды іздейтін полиномдық алгоритмдер бар[53], бірақ ондай алгоритмдер күрделі және бағдарламау контекстерінде сирек кездеседі. Дегенмен екі ұялы графта максималды жұптасуды табу оңайырақ, өйткені біз оны максималды ағын есебіне келтіре аламыз.

Максималды жұптасуды іздеу

Екі ұялы графтағы төбелерді қырлары сол топтан оң топқа бағытталатындай етіп, әрдайым екі топқа бөле аламыз. Мысалы, төмендегі графтағы екі топ – $\{1,2,3,4\}$ және $\{5,6,7,8\}$.

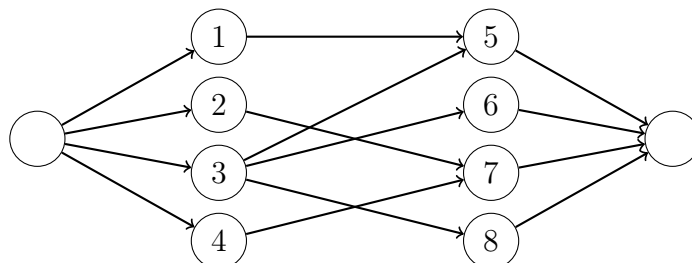


Графтың максималды жұптасуының өлшемі – 3:

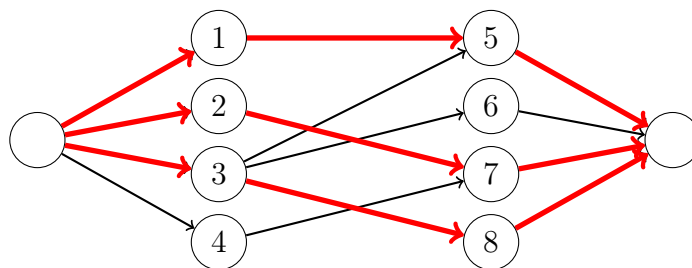


Максималды жұптасу есебін бастау мен саға төбелерін қосу арқылы максималды ағын есебіне келтіре аламыз. Оған қоса, бастаудан сол жақ топтағы барлық төбелерге қырлар қосып, сағаға оң жақ топтағы барлық төбелерден қырлар қосамыз. Кейін осы графтағы максималды ағынның өлшемі берілген графтағы максималды жұптасу өлшеміне тең болады.

Мысалы, жаңа өзгерген графымыз төмендегідей болады:



Графтың максималды ағыны келесідей болмақ:

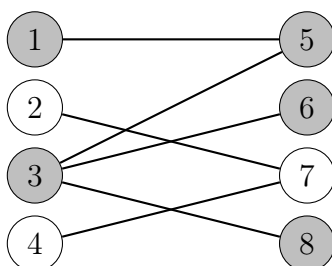


Хол теоремасы

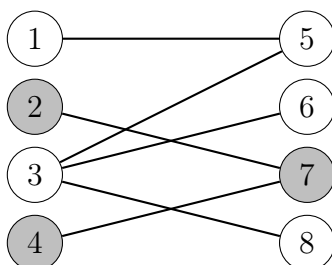
Хол теоремасы арқылы екі ұялы графта барлық сол немесе оң жақ топтағы төбелерді қамтитын жұптасудың бар-жоғын тексеруге болады. Егер сол мен оң жақ топтағы төбелердің сандары бірдей болса, Хол теоремасы графтағы барлық төбелерді қамтитын мінсіз жұптасу құрылысының мүмкін екендігін хабарлайды.

Мысалы, барлық сол жақ топтағы төбелерді қамтитын жұптасуды тапқымыз келеді делік. X деп кез келген сол жақ топтағы төбелер жиындысы және $f(X)$ деп олардың көршілерін белгілейік. Хол теоремасы бойынша әр X -ке $|X| \leq |f(X)|$ шарты орындалған жағдайда ғана барлық сол жақ топтағы төбелерді қамтитын жұптасу жүзеге асады.

Хол теоремасын граф мысалында қарайық. Басында $X = \{1, 3\}$ деп алсақ және сәйкесінше $f(X) = \{5, 6, 8\}$ болса:



Хол теоремасының шарты сақталды. Себебі $|X| = 2$ және $|f(X)| = 3$. Келесі $X = \{2, 4\}$ деп алайық және сәйкесінше $f(X) = \{7\}$ болсын:



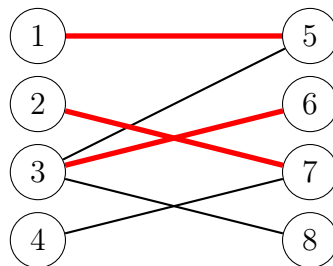
Осы жағдайда $|X| = 2$ және $|f(X)| = 1$, яғни Хол теоремасының шарты орындалмады. Бұл граф үшін мінсіз жұптасуды құру мүмкін емес дегенді білдіреді. Тұжырым таңқаларлық емес, өйткені осыған дейін де осы графтың максималды ағыны 4 емес, 3 екенін білген болатынбыз.

Егер Хол теоремасының шарты орындалмаса, X жиыны ондай жұптасу неге бола алмайтынына түсіндірме береді. X -те $f(X)$ -тен төбелер көп болғандықтан, X -тің барлық төбелеріне жұп табылмайды. Мысалы, жоғарыдағы графта 2-төбе мен 4-төбе екеуі де 7-төбемен байланысуы керек, бірақ ол мүмкін емес.

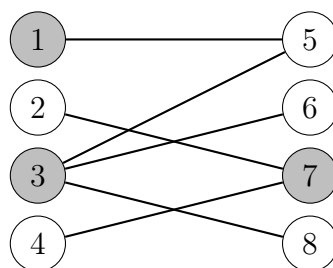
Кёниг теоремасы

Графтың минималды төбелер бүркемесі – графтағы қырлардың әр шетінің кем дегенде біреуі жиында болатын минималды төбелер жиыны. Жалпы графтың минималды төбелер бүркемесін табу NP-қиын есептер қатарынан саналады. Бірақ егер граф екі ұялы болса, Кёниг теоремасы минималды төбелер бүркемесінің өлшемі максималды жұптасу өлшемімен тең деп тұжырымдайды. Осылайша біз минималды төбелер бүркемесін максималды ағын алгоритмі арқылы таба аламыз.

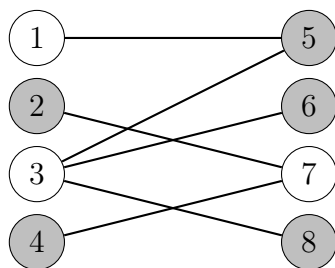
Төмендегі максималды жұптасу өлшемі 3-ке тең графты қарастырайық:



Кёниг теоремасы минималды төбелер бүркемесінің өлшемі де 3 екенін айтады. Сондай бүркеме төмендегідей ретпен құрыстырылады:



Минималды төбелер бүркемесіне кірмейтін төбелер максималды тәуелсіз жиынды құрайды. Бұл – оған тиесілі екі төбенің ешқайсысы қырлармен байланыспайтын, төбелердің мүмкін болатын максималды жиыны. Қайталап өтейік, жалпы графтарда максималды тәуелсіз жиынды табу NP-қиын есеп саналады, бірақ екі ұялы графтарда есепті Кёниг теоремасы арқылы тиімдірек шешуге болады. Мысалдағы графта максималды тәуелсіз жиын төмендегідей болады:

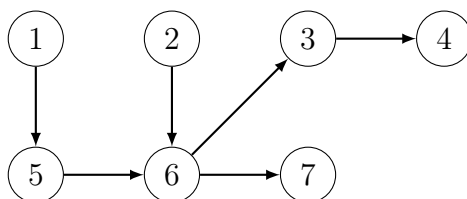


20.4 Жол бүркемелері

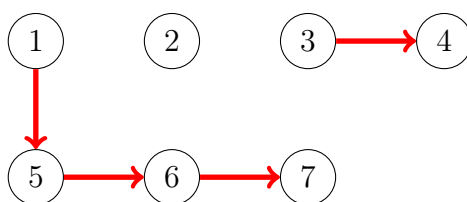
Жол бүркемесі – графтың әрбір төбесі кем дегенде бір жолға жататын жолдар жиынтығы. Бағытталған, циклсіз графта минималды жол бүркеме есебін басқа графтағы максималды ағын табу есебіне келтіре аламыз.

Төбелері қиылыспайтын жол бүркемесі

Төбелері қиылыспайтын жол бүркемесінде әр төбе тек бір жолға жатады. Үлгі ретінде төмендегі графты қарастырайық:



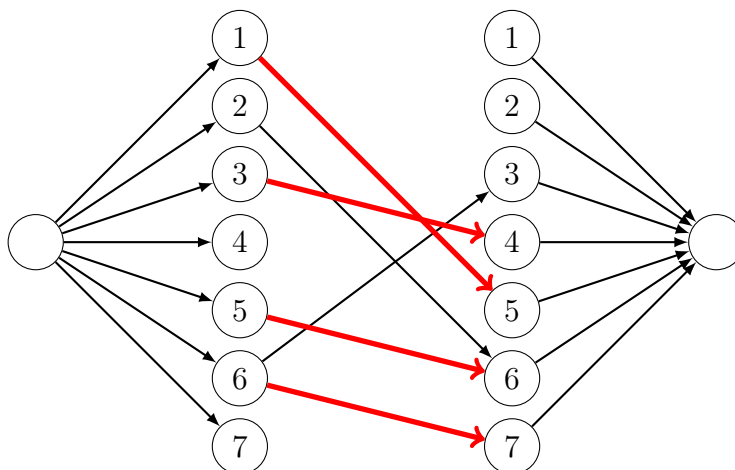
Графтағы минималды төбелері қиылыспайтын жол бүркемесі 3 жолдан тұрады. Мысалы, төмендегі үш жолды таңдасақ болады:



Жолдардың біреуі тек 2-төбеден тұратынына назар аударыңыз, яғни жолда ешқандай қыр болмауы мүмкін.

Минималды төбелері қиылыспайтын жол бүркемесін берілген графтың әр төбесін оң және сол төбелермен бейнелейтін жұптасу графын құрастырып табуға болады. Егер бастапқы графта сол жақ төбеден оң жақ төбеге қарай қыр болса, жұптасу графында қыр сол жақ және оң жақ төбелердің арасынан өтеді. Оған қоса жұптасу графы бастау мен сағаны қамтиды. Бастаудан барлық сол жақ төбелерге қырлар өтеді және сағаға барлық оң жақ төбелерден қырлар өтеді.

Жаңа графтағы максималды жұптасу берілген графтағы минималды төбелері қиылыспайтын жол бүркемесіне сәйкес болмақ. Мысалы, жоғарыдағы графтың жұптасу графының максималды жұптасуының өлшемі 4-ке тең:

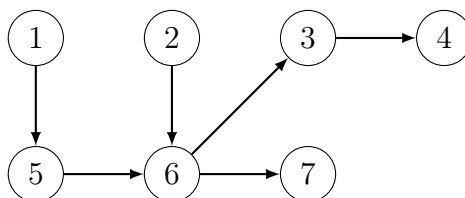


Жұптасу графындағы максималды жұптасудың әр қыры берілген граф-тағы төбелері қиылыспайтын жол бүркемесінің қырларына сәйкес келеді. Демек минималды төбелері қиылыспайтын жол бүркемесінің өлшемі $n - c$ болады, мұндағы n берілген графтағы төбелер саны болса, c максималды жұптасудың өлшемі болмақ.

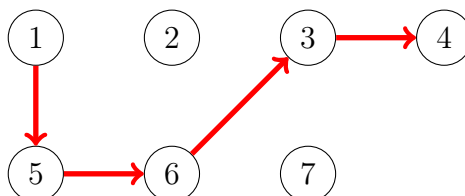
Жалпы жол бүркемесі

Жалпы жол бүркемесі – төбелер бірден көп жолдарда жата алатын жол бүркемесі. Минималды жалпы жол бүркемесі минималды төбелері қиылыспайтын жол бүркемесінен кіші болуы мүмкін, өйткені төбе жолдарда бірнеше мәрте қолданылуы мүмкін.

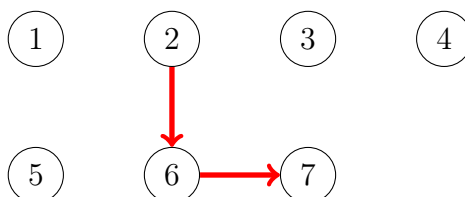
Келесі графты қарастырайық:



Графтың минималды жалпы жол бүркемесі екі жолдан тұрады. Мысалы, бірінші жол төмендегідей болуы мүмкін:

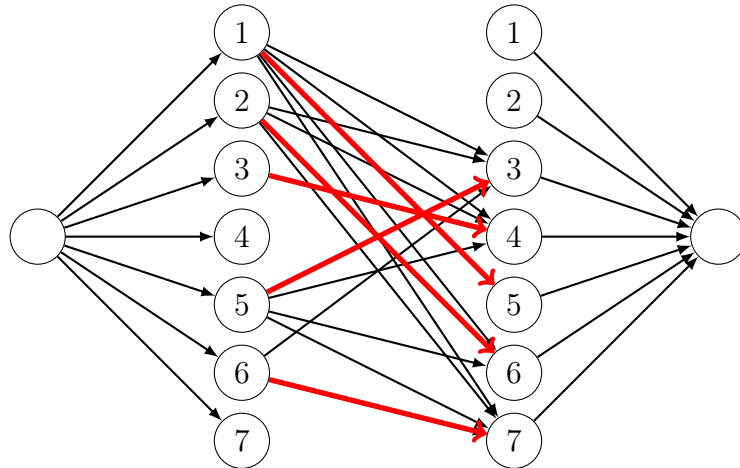


Ал екінші жол осындай болуы мүмкін:



Минималды жалпы жол бүркемесін төбелері қиылыспайтын минималды жол бүркемесі сияқты табуға болады. Ол үшін бастапқы графта a -дан b -ға (бәлкім бірнеше төбелер арқылы өтетін) жол болған жағдайда $a \rightarrow b$ қыры болатындай етіп, жұптасқан графқа тағы бірнеше қырлар қоссақ жеткілікі.

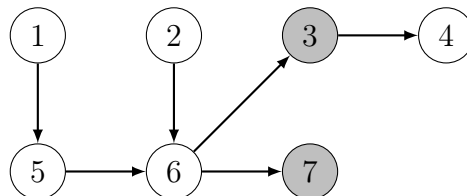
Жоғарыдағы графтың жұптасу графы төмендегідей болады:



Дилуорс теоремасы

Антитізбек – бір төбеден екінші төбеге қырлар арқылы жол болмайтын төбелер жиындысы. Дилуорс теоремасы бағытталған циклсіз графтың минималды жалпы жол бүркемесінің өлшемі максималды антитізбектің өлшеміне сәйкес деп мәлімдейді.

Мысалы төмендегі графта 3-төбе мен 7-төбе антитізбекті құрайды:



3 төбеден тұратын антитізбек мүлдем құрастырылмайтын болғандықтан, мысалдығы тізбек максималды антитізбекке жатады. Осы графтың минималды жалпы жол бүркемесінің өлшемі 2 жолдан тұратынын осыған білген едік.

III БӨЛІМ.

Күрделі тақырыптар

21-тарау. Сандар теориясы

Сандар теориясы – математика ғылымының бүтін сандарды зерттейтін тармағы. Сандар теориясы қызықты да күрделі сала. Себебі бүтін сандарға байланысты көптеген есептер бар және олар бір қарағанда қарапайым болып көрінгенімен, шешу барысында өте қиын екені байқалады.

Үлгі ретінде келесі теңдеуді қарастырайық:

$$x^3 + y^3 + z^3 = 33$$

Теңдеуді қанағаттандыратын x , y және z үш нақты санын табу оңай. Мысалы, біз келесі мәндерді таңдай аламыз:

$$\begin{aligned}x &= 3, \\y &= \sqrt[3]{3}, \\z &= \sqrt[3]{3}.\end{aligned}$$

Дегенмен: "теңдікті қанағаттандыратын x , y және z сандары бар ма" [54] - деген сұрақ сандар теориясындағы ашық мәселе болып отыр.

Тарауда сандар теориясындағы негізгі ұғымдар мен алгоритмдерге тоқталамыз. Егер басқаша көрсетілмесе, біз үшін тарау бойы барлық сандар бүтін сандар болады.

21.1 Жай сандар және көбейткіштер

Егер a саны b санын қалдықсыз бөлсе, a санын b санының көбейткіші немесе бөлгіші дейміз. Егер a саны b санының көбейткіші болса, $a \mid b$ түрінде, ал керісінше болса, $a \nmid b$ түрінде жазамыз. Мысалы, 24 санының көбейткіштері: 1, 2, 3, 4, 6, 8, 12 және 24.

Егер $n > 1$ санының жалғыз көбейткіштері 1 және n болса, ол – жай сан. Мысалы, 7, 19 және 41 – жай сандар. Бірақ 35 жай сан емес, себебі $5 \cdot 7 = 35$. Әр $n > 1$ санын жай көбейткіштерге бірегей жіктеуге болады:

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},$$

бұл жерде p_1, p_2, \dots, p_k – бірегей жай сандар, ал $\alpha_1, \alpha_2, \dots, \alpha_k$ – оң сандар. Мысалы, 84 санын жай сандарға келесі жолмен жіктейміз:

$$84 = 2^2 \cdot 3^1 \cdot 7^1.$$

n санының көбейткіштер саны:

$$\tau(n) = \prod_{i=1}^k (\alpha_i + 1),$$

себебі әр p_i жай саны көбейткіште $\alpha_i + 1$ ретке дейін кездесе алады. Мысалы, 84 санының көбейткіштер санын осылай есептейміз: $\tau(84) = 3 \cdot 2 \cdot 2 = 12$. Оның көбейткіштері: 1, 2, 3, 4, 6, 7, 12, 14, 21, 28, 42 және 84.

n санының көбейткіштер қосындысы осы формулаға тең:

$$\sigma(n) = \prod_{i=1}^k (1 + p_i + \dots + p_i^{\alpha_i}) = \prod_{i=1}^k \frac{p_i^{\alpha_i+1} - 1}{p_i - 1},$$

бұл жердегі екінші формула геометриялық прогрессия формуласына негізделген. Мысалы, 84 санының көбейткіштерінің қосындысы:

$$\sigma(84) = \frac{2^3 - 1}{2 - 1} \cdot \frac{3^2 - 1}{3 - 1} \cdot \frac{7^2 - 1}{7 - 1} = 7 \cdot 4 \cdot 8 = 224.$$

n санының көбейткіштер көбейтіндісінің формуласы –

$$\mu(n) = n^{\tau(n)/2}.$$

Себебі көбейтіндісі n болатын $\tau(n)/2$ көбейткіштер жұптарын құра аламыз. Мысалы, 84 санының көбейткіштері $1 \cdot 84$, $2 \cdot 42$, $3 \cdot 28$ және т. б. жұптарын құрайды, осылайша көбейткіштердің көбейтіндісі $\mu(84) = 84^6 = 351298031616$ тең.

Егер $n = \sigma(n) - n$ тең болса, яғни 1-ден $n - 1$ -ге дейінгі көбейткіштердің қосындысы n санына тең болса, n санын кемел сан дейміз. Мысалы, 28 кемел сан, себебі $28 = 1 + 2 + 4 + 7 + 14$.

Жай сандар мөлшері

Жай сандар мөлшері шексіз екенін көрсету оңай. Егер жай сандар шекті болса, $P = \{p_1, p_2, \dots, p_n\}$ барлық жай сандарды қамтитын жиын құрайтын едік. Мысалы, $p_1 = 2$, $p_2 = 3$, $p_3 = 5$ және дәл солай жалғаса береді. Дегенмен P жиынын қолданып біз жаңадан

$$p_1 p_2 \cdots p_n + 1$$

жай санын құрай аламыз. Ал ол сан P жиындағы барлық сандардан үлкен, бұл – қарама-қайшылық, сондықтан жай сандар мөлшері шексіз болмақ.

Жай сандар тығыздығы

Жай сандар тығыздығы – сандар арасындағы жай сандардың жиілігін көрсетеді. $\pi(n)$ деп 1-ден n -ге дейінгі жай сандардың санын көрсетеді. Мысалы, $\pi(10) = 4$, себебі 1 мен 10 арасында 4 жай сан бар. Олар: 2, 3, 5 және 7.

Келесіні көрсетуге болды:

$$\pi(n) \approx \frac{n}{\ln n}.$$

Бұл жай сандардың жиі кездесетінін байқатады. Мысалы, 1 және 10^6 арасындағы жай сандар мөлшері $\pi(10^6) = 78498$ тең және $10^6 / \ln 10^6 \approx 72382$.

Гипотеза

Жай сандарға қатысты көптеген гипотезалар бар. Көп адамдар гипотезаларды шындық деп санайды, бірақ оны ешкім әлі дәлелдей алмады. Мысалы, бізге белгілі келесі гипотезалар бар:

- Голдбах гипотезасы: әр $n > 2$ жұп бүтін саны a және b жай сандардың қосындысына тең.
- Егіз сандар гипотезасы: p және $p + 2$ жай сан болатындай $\{p, p + 2\}$ түріндегі жұптардың шексіз саны бар.
- Лежандр гипотезасы: n саны оң бүтін сан болса, n^2 және $(n + 1)^2$ арасында әрқашанда жай сан болады.

Негізгі алгоритмдер

Егер n саны жай сан болмаса, оны $a \cdot b$ көбейтіндісі ретінде көрсетуге болады. Бұл жерде $a \leq \sqrt{n}$ немесе $b \leq \sqrt{n}$, демек бұл сан 2 мен $\lfloor \sqrt{n} \rfloor$ арасында көбейткішті қамтиды. Осы бақылау арқылы біз санды жай санға $O(\sqrt{n})$ уақытта тексере аламыз. Одан басқа, санды жай сандарға да $O(\sqrt{n})$ уақытта жіктей аламыз.

Келесі prime функциясы n саны жай сан екендігіне тексереді. Функция n санын 2 мен $\lfloor \sqrt{n} \rfloor$ арасындағы барлық сандарға бөліп көреді. Соның арасындағы еш сан бөлмесе, онда n саны жай сан болады.

```
bool prime(int n) {
    if (n < 2) return false;
    for (int x = 2; x*x <= n; x++) {
        if (n%x == 0) return false;
    }
    return true;
}
```

Келесі factors функциясы n санын жай сандарға жіктеген кездегі жай сандарды қамтитын векторды құрайды. Функция n санды оның жай сан болатын көбейткіштеріне бөледі және соларды векторға қосады. Егер қалған n саны 2 мен $\lfloor \sqrt{n} \rfloor$ арасындағы көбейткіштерді қамтымаса, бұл процесс аяқталады. Егер $n > 1$, ол жай сан және соңғы көбейткіш болады.

```
vector<int> factors(int n) {
    vector<int> f;
    for (int x = 2; x*x <= n; x++) {
```

```

        while (n%x == 0) {
            f.push_back(x);
            n /= x;
        }
    }
    if (n > 1) f.push_back(n);
    return f;
}

```

Әрбір жай көбейткіштің векторда санды қанша бөлсе, сонша рет пайда болатынын ескергеніміз жөн. Мысалға, $24 = 2^3 \cdot 3$ сондықтан функцияның вектордағы элементтері $[2, 2, 2, 3]$ болады.

Эратосфен елегі

Эратосфен елегі дегеніміз $2 \dots n$ арасындағы сан жай сан екендігін тиімді тексеруге көмектесетін жиымды құрайтын алдын ала өңдеу алгоритмі.

Алгоритм $2, 3, \dots, n$ позициялары қолданылатын жиым елегін құрайды.

$\text{sieve}[k] = 0$ деген мән k саны жай сан екенін білдіреді, ал керісінше, $\text{sieve}[k] \neq 0$ мәні сан жай сан емес екенін білдіріп, осы санның бір жай көбейткіші $\text{sieve}[k]$ екенін тауып береді.

Алгоритм $2 \dots n$ арасындағы сандарды біртіндеп жүріп шығады. x жай саны табылған кезде, алгоритм x санының еселіктерін жай сан емес деп белгілейді. Себебі, оларды x саны бөледі.

Мысалға, егер $n = 20$, жиым осындай болады:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	2	0	3	0	2	3	5	0	3	0	7	5	2	0	3	0	5

Эратосфен елегінің коды келесі ретпен жазылады. Код sieve жиымының әрбір элементін бастапқыда нөлге тең деп есептейді.

```

for (int x = 2; x <= n; x++) {
    if (sieve[x]) continue;
    for (int u = 2*x; u <= n; u += x) {
        sieve[u] = x;
    }
}

```

Алгоритмнің ішкі қайталымы әр x мәніне n/x рет орындалады. Сондықтан алгоритмнің орындалу уақытының жоғарғы шегі гармоникалық қосынды деп саналады:

$$\sum_{x=2}^n n/x = n/2 + n/3 + n/4 + \dots + n/n = O(n \log n).$$

Ішкі қайталым тек x саны жай болғанда ғана орындалатындықтан, бұл алгоритм шынымен де тиімдірек саналады. Алгоритмнің орындалу уақыты тек $O(n \log \log n)$ болатындықтан, бұл алгоритм күрделілігі жағынан $O(n)$ -ға өте жақын дей аламыз.

Евклид алгоритмі

a мен b сандардың ең үлкен ортақ бөлгіші – $\gcd(a, b)$, ол – a мен b -ны бөлетін ең үлкен сан және a мен b сандардың ең кіші ортақ еселігі – $\text{lcm}(a, b)$, ол – a және b -ға бөлінетін ең кіші сан. Мысалы, $\gcd(24, 36) = 12$ және $\text{lcm}(24, 36) = 72$.

Ең үлкен ортақ бөлгіш пен ең кіші ортақ еселік келесі ретпен байланысады:

$$\text{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$$

Эвклид алгоритмі¹ екі санның ең үлкен ортақ бөлгішін табудың тиімді әдісін ұсынады. Алгоритм келесі формулаға негізделген:

$$\gcd(a, b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & b \neq 0 \end{cases}$$

Мысалы,

$$\gcd(24, 36) = \gcd(36, 24) = \gcd(24, 12) = \gcd(12, 0) = 12.$$

Алгоритмді осылай жазуға болады:

```
int gcd(int a, int b) {  
    if (b == 0) return a;  
    return gcd(b, a%b);  
}
```

Евклид алгоритмінің $O(\log n)$ ($n = \min(a, b)$) уақытта жұмыс істейтінін көрсетуге болады. Егер a және b қатар Фибоначчи сандары болса, алгоритм үшін ең нашар жағдай туындайды. Мысалы,

$$\gcd(13, 8) = \gcd(8, 5) = \gcd(5, 3) = \gcd(3, 2) = \gcd(2, 1) = \gcd(1, 0) = 1.$$

Эйлер функциясы

Егер $\gcd(a, b) = 1$ болса, a және b сандары өзара жай сандар болып есептеледі. Эйлер функциясы $\varphi(n)$ n санына 1 мен n арасындағы өзара жай сандардың мөлшерін қайтарады. Мысалы, $\varphi(12) = 4$. Себебі 12 санына 1, 5, 7 және 11 өзара жай сан болады.

Келесі формулада n санын жай сандарға жіктеу арқылы $\varphi(n)$ мәнін табуға болады:

$$\varphi(n) = \prod_{i=1}^k p_i^{\alpha_i - 1} (p_i - 1).$$

Мысалы, $\varphi(12) = 2^1 \cdot (2 - 1) \cdot 3^0 \cdot (3 - 1) = 4$. Бұл жерде мынаны ескеру керек, егер n жай сан болса, $\varphi(n) = n - 1$ болады.

¹Евклид –біздің эрамызға дейінгі 300-жылдары өмір сүрген грек математигі.

21.2 Модульдік арифметика

Модульдік арифметикада сандардың жиындары тек $0, 1, 2, \dots, m-1$ сандары ғана қолданылатындай болып шектеледі, мұндағы m - тұрақты сан.

Әр x санын $x \bmod m$ саны ретінде көрсетуге болады. Ол – x санының m санға бөлгендегі қалдығы. Мысалы, егер $m = 17$ десек, 75 санын $75 \bmod 17 = 7$ ретінде қарауға болады.

Біз көбіне есептеулерді жасамас бұрын қалдықтарды аламыз. Атап айтқанда, келесі формулалар қолданылады:

$$\begin{aligned}(x + y) \bmod m &= (x \bmod m + y \bmod m) \bmod m \\(x - y) \bmod m &= (x \bmod m - y \bmod m) \bmod m \\(x \cdot y) \bmod m &= (x \bmod m \cdot y \bmod m) \bmod m \\x^n \bmod m &= (x \bmod m)^n \bmod m\end{aligned}$$

Модульдік дәрежеге шығару

Көбіне $x^n \bmod m$ мәнін тиімді есептеу қажет болып жатады. Оны $O(\log n)$ уақытта рекурсия арқылы табуға болады:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ жұп сан} \\ x^{n-1} \cdot x & n \text{ тақ сан} \end{cases}$$

21.3 n жұп болған жағдайда $x^{n/2}$ мәні

тек бір рет есептелетіні маңызды. Бұл алгоритмнің уақытша күрделілігі $O(\log n)$ болатынына кепілдік береді, себебі n жұп болған кезде ол әрқашан екі есе азаяды.

Келесі функция $x^n \bmod m$ мәнін есептейді:

```
int modpow(int x, int n, int m) {
    if (n == 0) return 1%m;
    long long u = modpow(x, n/2, m);
    u = (u*u)%m;
    if (n%2 == 1) u = (u*x)%m;
    return u;
}
```

Ферма теоремасы және Эйлер теоремасы

Ферма теоремасы m саны жай сан болса, x және m сандары өзара жай болса,

$$x^{m-1} \bmod m = 1$$

болады деген формуланы тұжырымдайды. Бұдан келесі формуланы алуға болады:

$$x^k \bmod m = x^{k \bmod (m-1)} \bmod m.$$

Жалпы жағдайда, Эйлер теоремасы x және m өзара жай болса,

$$x^{\varphi(m)} \bmod m = 1$$

деп тұжырымдайды. Ферма теоремасы Эйлер теоремасынан шығады. Өйткені егер m жай сан болса, онда $\varphi(m) = m - 1$ болар еді.

Модуль бойынша кері сан

$$xx^{-1} \bmod m = 1$$

теңдеуі шығатындай x модуліндегі m санының кері саны x^{-1} болады. Мысалы, егер $x = 6$ және $m = 17$ болса, $x^{-1} = 3$ тең, себебі $6 \cdot 3 \bmod 17 = 1$.

Осылай модуль бойынша кері сандарды алу арқылы біз сандарды m модулі бойынша бөле аламыз. Сондықтан x бөлуі x^{-1} көбейтуіне балама болады.

Мысалы, $36/6 \bmod 17$ мәнін есептеу үшін біз $2 \cdot 3 \bmod 17$ формуласын қолдана аламыз, себебі $36 \bmod 17 = 2$ және $6^{-1} \bmod 17 = 3$.

Дегенмен әр санның модуль бойынша кері саны бола бермейді. Мысалы, егер $x = 2$ және $m = 4$ болса, алгоритм

$$xx^{-1} \bmod m = 1$$

теңдеуін шығаруға келмейді. Себебі, екінің барлық көбейтінділері жұп сан болады және оның қалдығы $m = 4$ болғанда ешқашан бір бола алмайды. $x^{-1} \bmod m$ мәнін x және m өзара жай болатын кезде ғана есептеуге болады.

Егер модуль бойынша кері сан бар болса, оны осы формула арқылы табуға болады:

$$x^{-1} = x^{\varphi(m)-1}.$$

Егер m жай сан болса, формула осындай болады:

$$x^{-1} = x^{m-2}.$$

Мысалы,

$$6^{-1} \bmod 17 = 6^{17-2} \bmod 17 = 3.$$

Бұл формула модульдік дәрежеге шығару алгоритмін пайдаланып, модуль бойынша кері мәндерді тиімді есептеуге мүмкіндік береді. Формула Эйлер теоремасы арқылы шығарылуы мүмкін. Біріншіден, модуль бойынша кері сан келесі теңдеуді қанағаттандыруы керек:

$$xx^{-1} \bmod m = 1.$$

Екіншіден, Эйлер теоремасы бойынша,

$$x^{\varphi(m)} \bmod m = xx^{\varphi(m)-1} \bmod m = 1,$$

сонда, x^{-1} және $x^{\varphi(m)-1}$ тең болады.

Компьютердегі арифметика

Бағдарламалауда таңбасыз бүтін сандар 2^k модулімен көрсетіледі, мұндағы k саны – деректер типіндегі биттердің саны, яғни сан асып кетсе, ол 2^k модулі бойынша алынады.

Мысалы, C++ тілінде unsigned int типіндегі сандар 2^{32} модулі бойынша көрсетіледі. Келесі код мәні 123456789 болатын unsigned int типіндегі айнымалыны жариялайды. Содан кейін мәнді өзіне көбейтеді және төмендегідей соңғы нәтижеге қол жеткізеді: $123456789^2 \bmod 2^{32} = 2537071545$.

```
unsigned int x = 123456789;
cout << x*x << "\n"; // 2537071545
```

21.4 Теңдеулерді шешу

Диофант теңдеулері

Диофант теңдеулері деп

$$ax + by = c$$

түріндегі теңдеулерді айтамыз. Мұндағы a , b мен c тұрақты сандар. Біз x пен y мәндерін табуымыз қажет. Теңдеудегі әр сан бүтін сан болу керек. Мысалы, $5x + 2y = 11$ теңдеудің бір шешімінде $x = 3$ және $y = -2$ мәндері шығады.

Евклид алгоритмін қолдану арқылы Диофант теңдеулерін тиімді шеше аламыз. Евклид алгоритмін келесі теңдеуді қанағаттандыратын x және y сандарын анықтайтындай етіп кеңейтуге болады:

$$ax + by = \gcd(a, b)$$

Егер c саны $\gcd(a, b)$ санына бөлінетін болса, Диофант теңдеуін шешуге болады. Ал басқа кезде шеше алмаймыз.

Мысал ретінде мына теңдеуді қанағаттандыратын x және y сандарын табайық:

$$39x + 15y = 12$$

$\gcd(39, 15) = 3$ және $3 \mid 12$ болғандықтан, теңдеуді шеше аламыз. Евклид алгоритмі 39 бен 15 сандарының ең үлкен ортақ бөлгішін есептегенде, функцияны келесі тізбекте шақырады:

$$\gcd(39, 15) = \gcd(15, 9) = \gcd(9, 6) = \gcd(6, 3) = \gcd(3, 0) = 3$$

Бұл дегеніміз келесі теңдеулерге сәйкес:

$$\begin{aligned} 39 - 2 \cdot 15 &= 9 \\ 15 - 1 \cdot 9 &= 6 \\ 9 - 1 \cdot 6 &= 3 \end{aligned}$$

Осы теңдеулерді пайдалана отырып, біз мынаны шығара аламыз:

$$39 \cdot 2 + 15 \cdot (-5) = 3$$

Содан соң оны 4 санына көбейту арқылы келесі нәтижені аламыз:

$$39 \cdot 8 + 15 \cdot (-20) = 12,$$

осылай теңдеудің шешімі $x = 8$ және $y = -20$ мәндері екенін таптық.

Диофант теңдеуінің шешімі бірегей емес, өйткені егер бір шешімді білсек, біз шешімдердің шексіз санын құра аламыз. Егер (x, y) жұбы шешім болса, сонда k саны кез-келген бүтін сан болатын

$$\left(x + \frac{kb}{\gcd(a, b)}, y - \frac{ka}{\gcd(a, b)}\right)$$

жұптары да шешім бола алады.

Қалдықтар туралы қытай теоремасы

Қалдықтар туралы қытай теоремасы төмендегі теңдеулер тобын шешеді:

$$\begin{aligned} x &= a_1 \bmod m_1 \\ x &= a_2 \bmod m_2 \\ &\dots \\ x &= a_n \bmod m_n \end{aligned}$$

бұл жердегі барлық m_1, m_2, \dots, m_n жұптары өзара жай.

m модуліндегі x кері санын x_m^{-1} деп белгілейік және

$$X_k = \frac{m_1 m_2 \cdots m_n}{m_k}.$$

Бұл нотацияны пайдаланып, теңдеудің шешімі төмендегідей болады:

$$x = a_1 X_1 X_{1m_1}^{-1} + a_2 X_2 X_{2m_2}^{-1} + \cdots + a_n X_n X_{nm_n}^{-1}.$$

Осы шешімде, әр $k = 1, 2, \dots, n$ мәндеріне

$$a_k X_k X_{km_k}^{-1} \bmod m_k = a_k,$$

себебі

$$X_k X_{km_k}^{-1} \bmod m_k = 1.$$

Қосындыдағы барлық басқа мүшелер m_k -ға бөлінетіндіктен, олардың қалдыққа әсері жоқ, және $x \bmod m_k = a_k$.

Мысалы, төмендегі есептің

$$\begin{aligned} x &= 3 \bmod 5 \\ x &= 4 \bmod 7 \\ x &= 2 \bmod 3 \end{aligned}$$

шешімі осындай болады:

$$3 \cdot 21 \cdot 1 + 4 \cdot 15 \cdot 1 + 2 \cdot 35 \cdot 2 = 263.$$

Біз бір x шешімін тапқаннан кейін басқа шешімдердің шексіз санын жасай аламыз. Себебі

$$x + m_1 m_2 \cdots m_n$$

түріндегі барлық сандар шешім бола алады.

21.5 Басқа нәтижелер

Лагранж теоремасы

Лагранж теоремасы әрбір натурал санды төрт квадраттың қосындысы ретінде көрсетуге болатынын, яғни $a^2 + b^2 + c^2 + d^2$ түрінде келетінін білдіреді. Мысалы, 123 санын $8^2 + 5^2 + 5^2 + 3^2$ деген сандардың қосындысы ретінде қарастыруға болады.

Цекендорф теоремасы

Цекендорф теоремасы әрбір оң бүтін санның Фибоначчи сандарының қосындысы ретінде бірегей көрінісі болатынын дәлелдейді. Бірақ екі Фибоначчи саны тең емес және олар қатар орналаспаған болу керек. Мысалы, 74 санын $55 + 13 + 5 + 1$ Фибоначчи сандардың қосынды ретінде қарауға болады.

Пифагор үштіктері

Пифагор үштігі дегеніміз $a^2 + b^2 = c^2$ түріндегі Пифагор теоремасына сай келетін (a, b, c) үштігі. Бұл қабырғаларының ұзындығы a , b және c болатын тікбұрышты үшбұрыш бар екенін білдіреді. Мысалы, $(3, 4, 5)$ деген Пифагор үштігі.

Егер (a, b, c) Пифагор үштігі болса, барлық $k > 1$ сандарына (ka, kb, kc) түріндегі үштіктер де Пифагор үштігі болып саналады. Егер a , b және c өзара жай болса, Пифагор үштігі қарапайым болады және барлық Пифагор үштіктерін қарапайым үштікті k -ға көбейтіп құрастыруға болады.

Эвклид формуласын барлық Пифагор үштіктерін табу үшін қолдануға болады. Әр сондай үштік $0 < m < n$, n және m өзара жай және кем дегенде екеуінен біреуі жұп сан деп алсақ,

$$(n^2 - m^2, 2nm, n^2 + m^2)$$

түрінде көрсетуге болады. Мысалы, $m = 1$ және $n = 2$ болғанда, формула ең кіші Пифагор үштігін анықтайды:

$$(2^2 - 1^2, 2 \cdot 2 \cdot 1, 2^2 + 1^2) = (3, 4, 5).$$

Уилсон теоремасы

Уилсон теоремасы бойынша егер

$$(n - 1)! \bmod n = n - 1$$

теңдеуі орындалса ғана n саны жай сан болады деп тұжырымдалады.

Мысалы, 11 деген жай сан, өйткені

$$10! \bmod 11 = 10$$

Ал 12 жай сан емес, себебі

$$11! \bmod 12 = 0 \neq 11.$$

Демек Уильсон теоремасын санның қандай екенін анықтау үшін қолдануға болады. Бірақ тәжірибеде теореманы үлкен мәндерге қолдануға болмайтыны байқалады, себебі n үлкен болғанда $(n - 1)!$ мәндерін есептеу қиынға соғады.

22-тарау. Комбинаторика

Комбинаторика объектілер комбинацияларын санау әдістерін зерттейді. Әдетте ол әр комбинацияны бөлек өндірмей, комбинацияларды тиімді санау мақсатын көздейді.

Өрнек ретінде n бүтін санын оң бүтін сандардың қосындысы ретінде қанша жолмен көрсетуге болатын анықтайтын есептерді қарастырайық. Мысалы 4 санының 8 көрінісі бар:

- | | |
|-------------|---------|
| • $1+1+1+1$ | • $2+2$ |
| • $1+1+2$ | • $3+1$ |
| • $1+2+1$ | • $1+3$ |
| • $2+1+1$ | • 4 |

Комбинаторлық есептерді әдетте рекурсивті функция арқылы шешуге болады. $f(n)$ деп n санының көріністер санын беретін функцияны белгілейік. Жоғарыдағы мысалға сәйкес $f(4) = 8$ болмақ. Функцияның мәндері рекурсивті түрде төмендегідей есептеледі:

$$f(n) = \begin{cases} 1 & n = 0 \\ f(0) + f(1) + \dots + f(n-1) & n > 0 \end{cases}$$

Функцияның негізгі жағдайы $f(0) = 1$, өйткені бос қосынды 0 санының көрінісі. Егер $n > 0$ болса, қосындының бірінші саны болатын барлық жолдарды қарастырамыз. Егер бірінші сан k болса, онда қосындының қалған бөлігін $f(n-k)$ түрінде көрсетуге болады. Демек $k < n$ болатын $f(n-k)$ формасындағы барлық мәндерінің қосындысын санаймыз.

Функцияның алғашқы мәндері:

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 1 \\ f(2) &= 2 \\ f(3) &= 4 \\ f(4) &= 8 \end{aligned}$$

Кейде рекурсивті формуланы аналитикалық шешіммен көрсетуге болады. Мысалы, үстідегі есепте

$$f(n) = 2^{n-1}.$$

Бұл шешімде біз $n-1$ позицияларына $+$ таңбасын қоя аламыз. Біз әр іш-жиымды осы фактіге негіздей отырып есептеу керекпіз. Ал ол 2^{n-1} болады.

22.1 Биномдық коэффициент

Биномдық коэффициент $\binom{n}{k}$ n элемент жиында k элементінен тұратын қанша ішжиын бар екенін көрсетеді. Мысалы $\binom{5}{3} = 10$, себебі $\{1, 2, 3, 4, 5\}$ жиынында 3 элементтен тұратын 10 ішжиын бар:

$\{1, 2, 3\}, \{1, 2, 4\}, \{1, 2, 5\}, \{1, 3, 4\}, \{1, 3, 5\}, \{1, 4, 5\}, \{2, 3, 4\}, \{2, 3, 5\}, \{2, 4, 5\}, \{3, 4, 5\}$

1-формула

Биномдық коэффициент рекурсивті түрде келесідей ретпен есептеледі:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Идеясы жиындағы x элементін бекітуге негізделеді. Егер x ішжиында болатын болса, онда бізге $n-1$ элементтен $k-1$ элемент таңдау керек болады. Ал егер x элементі ішжиында болмайтын болса, онда бізге $n-1$ элементтен k элемент таңдау керек болады.

Рекурсияның негізгі жағдайлары –

$$\binom{n}{0} = \binom{n}{n} = 1,$$

өйткені бос ішжиынды және барлық элементтерді қамтитын ішжиынды тек қана осындай жолмен құрастыруға болады.

2-формула

Төменде биномдық коэффициенттерді есептеудің басқаша жолы келтірілген:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

n элементте $n!$ алмастыру болады. Біз барлық алмастырулардан өтіп шығып, алмастырудың алдыңғы k элементтерін ішжиынға қосамыз. Ішжиынның ішіндегі және сыртындағы сандардың тәртібі маңызды емес болғандықтан, нәтижені $k!$ және $(n-k)!$ бөлеміз.

Қасиеттері

Биномдық коэффициенттерде

$$\binom{n}{k} = \binom{n}{n-k},$$

қасиеті орындалады, себебі n элементтерді екі ішжиынға бөлген кезде біріншісі k элементін қамтыса, екіншісі $n-k$ элементін қамтиды.

Биномдық коэффициенттердің қосындысы –

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{n} = 2^n.$$

Олардың ”биномдық коэффициент” аталу себебін $(a + b)$ биноминалын n -ші дәрежеге шығарғанда көре аламыз:

$$(a + b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \dots + \binom{n}{n-1}a^1b^{n-1} + \binom{n}{n}a^0b^n.$$

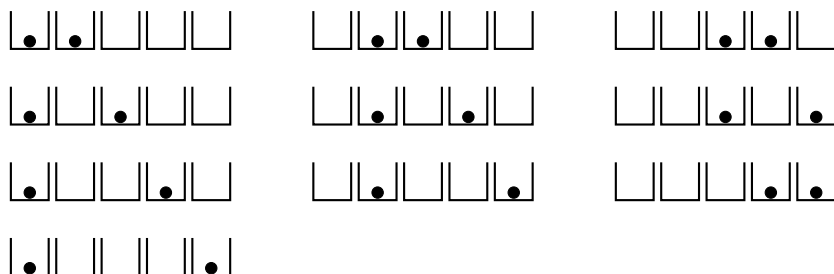
Биномдық коэффициенттер Паскаль үшбұрышында да кездеседі. Паскаль үшбұрышындағы мәндер үстіңгі екі мәндерінің қосындысынан құрылады:

$$\begin{array}{ccccccc} & & & 1 & & & \\ & & 1 & & 1 & & \\ & 1 & & 2 & & 1 & \\ 1 & & 3 & & 3 & & 1 \\ 1 & 4 & 6 & 4 & 1 & & \\ \dots & \dots & \dots & \dots & \dots & & \end{array}$$

Қораптар мен доптар

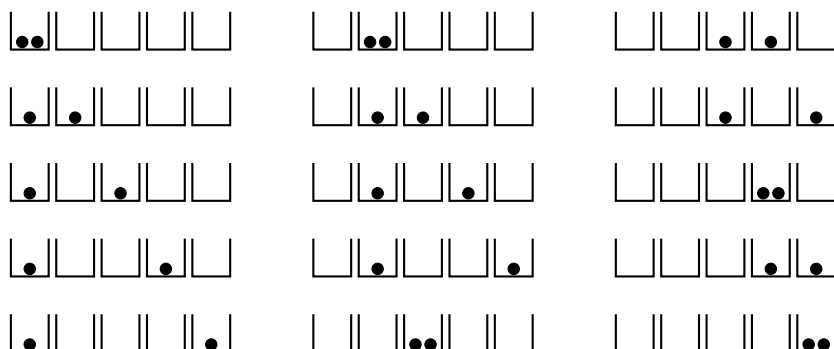
”Қораптар мен доптар” — n қораптарға k доптарды салу жолдарын есептейтін пайдалы модель. Үш түрлі жағдайды қарастырайық:

1-жағдай: Әр қорапта ең көбі 1 доп бола алады. Мысалы, егер $n = 5$ және $k = 2$ болғанда, 10 түрлі комбинация түзіледі:



Бұл жағдайда жауап биномдық коэффициентке $\binom{n}{k}$ тең.

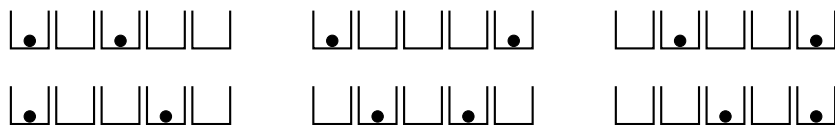
2-жағдай: Бір қорапта бірнеше доп бола алады. Мысалы $n = 5$ және $k = 2$ болғанда, доптарды қорапқа салудың 15 түрлі жолы шығады:



Доптарды қораптарға салуды "о" мен "→" таңбаларынан тұратын жол арқылы көрсетуге болады. Басында біз бірінші қорапта тұрамыз. "о" таңбасы допты қазіргі қорапқа салғанымызды білдіреді. "→" таңбасы келесі қорапқа көшкенімізді білдіреді.

Осы нотацияны қолдана отыра, әр жауап k мәрте "о" таңбадан және $n - 1$ мәрте "→" таңбадан тұратын жол түрінде болады. Мысалы үстіңгі оң жақтағы жауап "→ → о → о →" жолына сәйкес келеді. Осылайша жауаптардың саны $-\binom{k+n-1}{k}$ болады.

3-жағдай: Әр қорапта ең көбі бір доп бола алады және екі қатар келетін қораптарда доп болмау тиіс. Мысалы егер $n = 5$ және $k = 2$ болса, 6 жауап бола алады:



Бұл жағдайда басында k доптар қораптарға қойылды делік және екі қатар қораптардың арасында бос қорап болсын. Ендігі тапсырмамыз – қалған бос қораптардың позициясын таңдау. Ондай $n - 2k + 1$ қорап бар және оларға сәйкес $k + 1$ позициялары бар. Осылайша 2-жағдайдағы формула арқылы жауап $\binom{n-k+1}{n-2k+1}$ болады.

Мультиномдық коэффициент

Мультиномдық коэффициент

$$\binom{n}{k_1, k_2, \dots, k_m} = \frac{n!}{k_1! k_2! \dots k_m!},$$

n элементтерді k_1, k_2, \dots, k_m ішжиымдарға бөлу жолдарына тең, мұнда $k_1 + k_2 + \dots + k_m = n$. Мультиномдық коэффициенттер биномдық коэффициенттердің жалпылауы екенін көре аламыз. Егер $m = 2$ болса, жоғарыдағы формула биномдық коэффициентке сәйкес келеді.

22.2 Каталан сандары

Каталан сандары C_n n сол жақшалардан және n оң жақшалардан тұратын дұрыс жақшалар тізбектерінің санына тең.

Мысалы $C_3 = 5$, себебі үш оң және үш сол жақшалардан тұратын тізбектерді төмендегідей құрастыра аламыз:

- $()()()$
- $((()))$
- $()(())$
- $((())())$
- $((()()))$

Жақшалар тізбегі

”Жалпы дұрыс жақшалар тізбегі деген не?” - деген сұрақ туындайды. Төмендегі ережелер дұрыс жақшалар тізбегін дәл анықтап береді:

- Бос жақшалар тізбегі дұрыс болады.
- Егер A тізбегі дұрыс болса, онда (A) тізбегі де дұрыс болады.
- Егер A мен B тізбектері дұрыс болса, онда AB тізбегі де дұрыс болады.

Оны басқаша былай сипаттауға болады: Егер біз тізбектің кез келген префиксін алатын болсақ, оның сол жақшалар саны оң жақшалардың санынан көп болмауы тиіс. Оған қоса толық тізбекте оң жақшалардың саны сол жақшалардың санына тең болуы керек.

1-формула

Каталан сандарын келесі формула арқылы есептеуге болады:

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}.$$

Қосынды тізбекті екі дұрыс тізбек болатындай және бірінші тізбекте жақшалардың саны минималды болатындай етіп бөлетін жолдар арқылы өтеді. Әр i -ға бірінші тізбек $i+1$ жұп жақшаларды қамтиды және тізбектердің саны келесі мәндердің көбейтіндісіне тең болады:

- C_i : сыртқы жақшаларды есептемей, бірінші бөліктің жақшаларын пайдалану арқылы құрылған тізбектер саны
- C_{n-i-1} : екінші бөліктің жақшаларын пайдалану арқылы құрылған тізбектер саны

Негізгі жағдай $C_0 = 1$, себебі бос жақшалар тізбегі нөл жақшадан тұратын тізбекке сәйкес келеді.

2-формула

Каталан сандары биномдық коэффициенттер арқылы да есептеледі:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

Формулаға төмендегідей түсіндірме береміз:

n сол жақша мен n оң жақшадан тұратын жақшалар тізбегін (міндетті түрде дұрыс емес) құрастыруға жалпы $\binom{2n}{n}$ жол бар. Енді солардың ішінен дұрыс еместерін санайық.

Егер жақшалар тізбегі дұрыс емес болса, ол оң жақшалары саны сол жақшалар санынан артық префиксті қамтиды. Сондай префикске кіретін

жақшаларды керісінше өзгерту қажет. Мысалы $(())()$ тізбегі $()$ префиксін қамтиды және өзгерткеннен кейін тізбек $)(())($ болады.

Нәтижедегі тізбек $n + 1$ сол жақшалар мен $n - 1$ оң жақшалардан тұрады. Сондай тізбектердің саны $-\binom{2n}{n+1}$, яғни бұл дұрыс емес жақшалар тізбектердің санына тең дегенді білдіреді. Демек дұрыс жақшалары бар тізбектердің саны келесі формула арқылы есептелінеді:

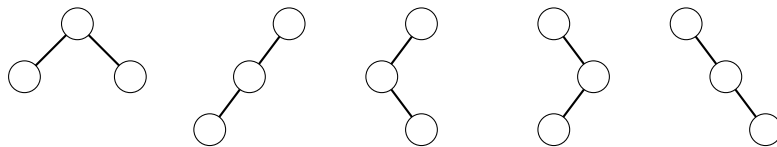
$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1} \binom{2n}{n} = \frac{1}{n+1} \binom{2n}{n}.$$

Дарақтарды санау

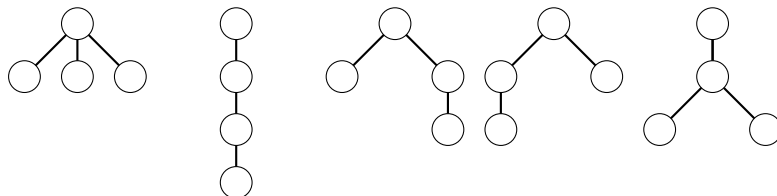
Каталан сандарының төмендегі дарақтарға да байланысы болады:

- n төбелі C_n бинарлы дарақтар бар
- n төбелі C_{n-1} түбірлі дарақтар бар

Мысалы $C_3 = 5$ кезінде төмендегідей бинарлы дарақтар



және төмендегідей түбірлі дарақтар болады

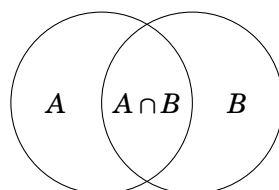


22.3 Inclusion–exclusion principle

Inclusion–exclusion principle – жиындар қиылысуының өлшемі берілген кездегі жиындар біріктіруінің өлшемін есептеуге арналған техника. Төмендегі формула аталмыш техникаға қарапайым мысал бола алады:

$$|A \cup B| = |A| + |B| - |A \cap B|,$$

мұнда A мен B жиындар және $|X|$ деп X -тің өлшемін белгілейміз. Формула төмендегідей көрініс алады:

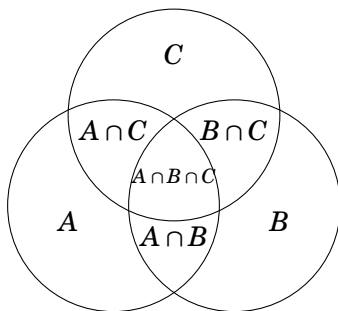


Біздің мақсатымыз – кем дегенде бір шеңберге кіретін бөліктің ауданына қатысты $A \cup B$ біріктіруін есептеу. Сурет $A \cup B$ ауданын есептеу үшін алдымен A мен B -ның ауданының қосындысын тауып, $A \cap B$ ауданын азайтса болатынын көрсетеді.

Дәл осындай идеяны жиындардың саны көп болған кезде де қолдануға болады. Егер үш жиын болса, inclusion–exclusion формуласы

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

және оған сәйкес сурет төмендегідей болады:



Жалпы жағдайда $X_1 \cup X_2 \cup \dots \cup X_n$ біріктіруінің өлшемін X_1, X_2, \dots, X_n жиындардың барлық қиылысуларынан өтіп санасақ болады. Егер жиындардың саны тақ болса, онда қиылысуын жауапқа қосатын боламыз. Басқаша жағдайда қиылысуын жауаптан шегереміз.

Осыған ұқсас формуланы жиындардың біріктірулерінің өлшемдері арқылы жиындардың қиылысуының өлшемін табуға қолдана алатынымызды ескеру керек. Мысалы

$$|A \cap B| = |A| + |B| - |A \cup B|$$

және

$$|A \cap B \cap C| = |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C| + |A \cup B \cup C|.$$

Ретсіздіктер

Өрнек ретінде $\{1, 2, \dots, n\}$ элементтерінің ретсіздіктерін, яғни ешқандай элемент өз орында тұрмайтын алмастыруларды санайық. Мысалы, $n = 3$ болғанда, екі ретсіздік кездеседі. олар: $(2, 3, 1)$ және $(3, 1, 2)$.

Бұл есептің шығару жолдарының бірі – inclusion–exclusion әдісін қолдану. X_k деп k элементі k -позициясында тұратын алмастыруларды белгілейік. Мысалы $n = 3$ болған кезде төмендегідей жиындар құрылады:

$$\begin{aligned} X_1 &= \{(1, 2, 3), (1, 3, 2)\} \\ X_2 &= \{(1, 2, 3), (3, 2, 1)\} \\ X_3 &= \{(1, 2, 3), (2, 1, 3)\} \end{aligned}$$

Осы жиындарды қолданып отырғандағы ретсіздіктер саны –

$$n! - |X_1 \cup X_2 \cup \dots \cup X_n|,$$

демек бізге жиындардың біріктіруінің өлшемін табу жеткілікті. Inclusion–exclusion әдісін пайдалансақ, бұл есепті қиылысулардың өлшемін табу есебіне келтіре аламыз. Ал ол болса, тиімді түрде есептелінеді. Мысалы $n = 3$ болған кезде $|X_1 \cup X_2 \cup X_3|$ -нің өлшемі

$$\begin{aligned} & |X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3| \\ &= 2 + 2 + 2 - 1 - 1 - 1 + 1 \\ &= 4, \end{aligned}$$

демек жауаптардың саны $- 3! - 4 = 2$.

Есепті inclusion–exclusion-сіз де шығаруға болады екен. $f(n)$ деп $\{1, 2, \dots, n\}$ элементтерінің ретсіздік санын белгілейік. Келесі рекурсивті формула арқылы оны есептеуге болады:

$$f(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ (n-1)(f(n-2) + f(n-1)) & n > 2 \end{cases}$$

Формуланы 1-элементтің ретсіздікті қалай өзгертетінін қарастыру арқылы дәлелдесек болады. 1-элементтің орынында тұратын x элементін $n-1$ жолмен таңдасақ болады. Әр жағдайдың екі нұсқасы болады:

1-нұсқа: x элементі 1-элементпен орын алмасады. Осыдан кейін $n-2$ элементтен тұратын ретсіздіктер санын санау ғана қалады.

2-нұсқа: x элементін 1-элементтен басқа элементпен ауыстырамыз. Енді бізге $n-1$ элементтен тұратын ретсіздіктерді құрастыру қажет болады. Өйткені біз x элементін 1-элементпен ауыстыра алмаймыз және қалған басқа элементтер де өзгеруі қажет болады.

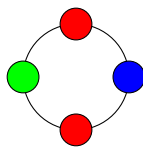
22.4 Бернсайд леммасы

Бернсайд леммасын әр симметрикалық комбинацияларды тек бір рет санау арқылы комбинацияларды есептеуге қолдануға болады. Бернсайд леммасы комбинациялар саны

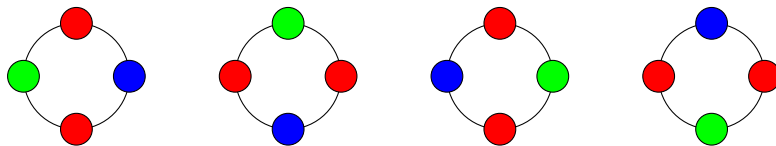
$$\sum_{k=1}^n \frac{c(k)}{n}$$

екенін тұжырымдайды, мұндағы n – комбинацияның орындарын ауыстыру жолдарының саны, ал $c(k)$ – k -нші ауыстыру жолын қолданғандағы өзгермеген комбинациялар саны.

Өрнек ретінде n тастан тұратын және әр таста m түс бола алатын алқалардың санын есептейік. Екі алқа, егер оларды айналдырғаннан кейін ұқсас бола берсе, симметриялы болады. Мысалы төмендегі алқаға



келесі симметриялы алқалар болады:



Алқа позициясын өзгеруінің n жолы бар: $0, 1, \dots, n-1$ рет сағат бағыты бойынша айналдыруға болады. 0 айналдырудан кейін m^n алқалар өзгермейді. 1 айналдырудан кейін тастарының түстері бірдей болатын m алқалар ғана өзгермейді.

Жалпылай айтқанда, айналдыру саны k болса,

$$m^{\gcd(k,n)}$$

алқалар өзгермейді, мұндағы $\gcd(k,n)$ – k мен n -нің ең үлкен ортақ бөлгіші. Өйткені ұзындығы $\gcd(k,n)$ болатын алқаның бөліктері орындарын ауыстырады. Осылайша Бернсайд леммасы бойынша алқалардың жалпы саны –

$$\sum_{i=0}^{n-1} \frac{m^{\gcd(i,n)}}{n}.$$

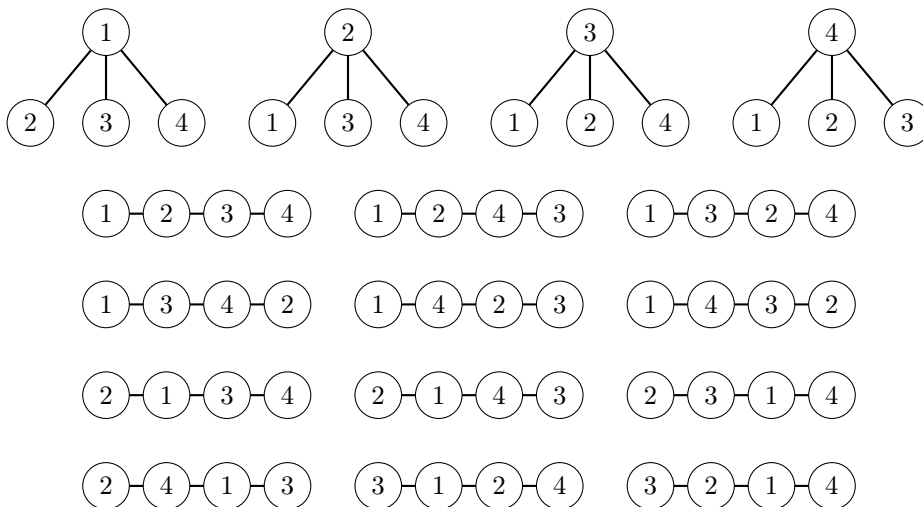
Мысалы, егер алқаның ұзындығы 4 және тастардың түстерінің саны 3 болса, алқалардың саны –

$$\frac{3^4 + 3 + 3^2 + 3}{4} = 24.$$

22.5 Кели теоремасы

Кели теоремасы n төбеде тұратын және сандармен белгіленген n^{n-2} дарақтың бар екенін тұжырымдайды. Төбелер $1, 2, \dots, n$ сандарымен белгіленеді. Егер құрылымдары немесе белгіленген сандары бірдей болмаса, екі дарақ бірдей емес деп есептеледі.

Мысалы $n = 4$ болған кезде, сандармен белгіленген $4^{4-2} = 16$ дарақ болады:

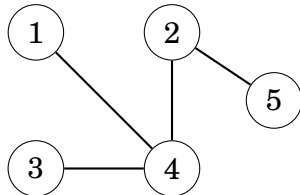


Кели теоремасын дәлелдеу үшін Прюфер кодтарын қолдануға болады.

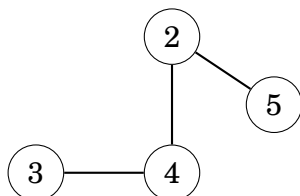
Прюфер кодтары

Прюфер коды – $n - 2$ сандардан тұратын белгіленген дарақты сипаттайтын тізбек. Код дарақтан $n - 2$ жапырақтарды өшіру үдерісі арқылы құрастырылады. Әр қадамда белгісі ең кіші жапырақ өшіріледі және оның жалғыз көршісінің белгісі кодқа жазылады.

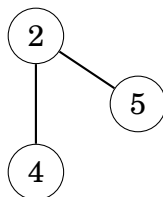
Мысалы келесі графтың Прюфер кодын есептейік:



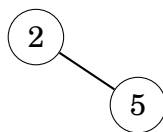
Алдымен біз 1-төбені өшіреміз, сосын 4-төбені кодқа қосамыз:



Кейін 3-төбені өшіреміз және 4-төбені кодқа қосамыз:



Соңында 4-төбені өшіріп, 2-төбені кодқа қосамыз:



Осылайша дарақтың Прюфер коды $[4, 4, 2]$ -ке тең болады.

Кез келген дараққа Прюфер кодын құрастыруға болады және ең маңыздысы, Прюфер кодымен бастапқы дарақты қайта құра аламыз. Демек n төбеден тұратын белгіленген дарақтардың саны n^{n-2} -ге, яғни өлшемі n болатын Прюфер кодтарының санына тең болады.

23-тарау. Матрицалар

Матрица бағдарламалаудағы екі өлшемді жиымға сәйкес математикалық тұжырымдама. Мысалы:

$$A = \begin{bmatrix} 6 & 13 & 7 & 4 \\ 7 & 0 & 8 & 2 \\ 9 & 5 & 4 & 18 \end{bmatrix}$$

– 3×4 өлшемді матрица, яғни 3 жолы және 4 бағанасы бар. $[i, j]$ нотациясы i -жолда және j -бағанда тұрған элементке сілтейді. Мысалы жоғарыдағы матрицада $A[2, 3] = 8$ және $A[3, 1] = 9$.

Матрицаның ерекше жағдайы – вектор. Ол – өлшемі $n \times 1$ болатын матрица. Мысалы:

$$V = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

– үш элементті қамтитын вектор.

A матрицаның аударуы A^T -ны A матрицасының жолдарымен бағаналарын ауыстыру арқылы алынады, яғни $A^T[i, j] = A[j, i]$:

$$A^T = \begin{bmatrix} 6 & 7 & 9 \\ 13 & 0 & 5 \\ 7 & 8 & 4 \\ 4 & 2 & 18 \end{bmatrix}$$

Егер жолдар саны мен бағана саны бірдей болса, ондай матрица шаршы матрица деп аталады. Мысалы төменде шаршы матрица берілген:

$$S = \begin{bmatrix} 3 & 12 & 4 \\ 5 & 9 & 15 \\ 0 & 2 & 4 \end{bmatrix}$$

23.1 Операциялар

A мен B матрицаларының қосындысы $A + B$ анықталған. Егер матрицалардың өлшемдері бірдей болса, нәтижесінде әр элементі сәйкес A және B элементтерінің қосындысын беретін матрица пайда болады.

Мысалы

$$\begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 7 \\ 11 & 10 & 5 \end{bmatrix}.$$

A матрицасын x мәніне көбейту деген сөз A әр элементінің x -ке көбейтілетінін білдіреді.

Мысалы

$$2 \cdot \begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 2 \cdot 6 & 2 \cdot 1 & 2 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 9 & 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 8 \\ 6 & 18 & 4 \end{bmatrix}.$$

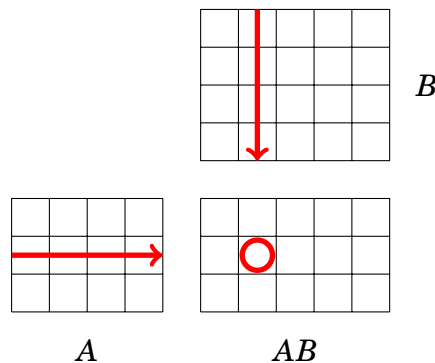
Матрицаларды көбейту

Егер A -ның өлшемі $a \times n$ және B -ның өлшемі $n \times b$ (яғни A -ның ені B -ның биіктігіне тең) болса, A мен B матрицаларының көбейтіндісі AB анықталған болады. Төменде көбейтіндісі

$$AB[i, j] = \sum_{k=1}^n A[i, k] \cdot B[k, j].$$

формуласы арқылы есептелген $a \times b$ өлшемді матрица беріледі.

Яғни AB -ның әр элементі төмендегі суреттегідей A мен B элементтері көбейтінділерінің қосындысына тең болады:



Мысалы:

$$\begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 6 \\ 2 & 9 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 4 \cdot 2 & 1 \cdot 6 + 4 \cdot 9 \\ 3 \cdot 1 + 9 \cdot 2 & 3 \cdot 6 + 9 \cdot 9 \\ 8 \cdot 1 + 6 \cdot 2 & 8 \cdot 6 + 6 \cdot 9 \end{bmatrix} = \begin{bmatrix} 9 & 42 \\ 21 & 99 \\ 20 & 102 \end{bmatrix}.$$

Матрицадағы көбейту — ассоциативті, яғни $A(BC) = (AB)C$, бірақ коммутативті емес, яғни әдетте $AB = BA$ болмайды.

Бірлік матрица — диагоналдағы элементтері бірге тең және қалған элементтері нөлге тең шаршы матрица. Мысалы, төменде 3×3 өлшемді бірлік матрица келтірілген:

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

Бірлік матрицаға көбейтілген матрица өзгермейді. Мысалы:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \quad \text{және} \quad \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix}.$$

Қарапайым алгоритм арқылы екі $n \times n$ матрицаларының көбейтіндісін $O(n^3)$ уақытта есептеуге болады. Матрицалардың көбейтіндісін есептейтін

одан да тиімді алгоритмдер бар¹, бірақ оларға теориялық тұрғыдан қызығушылық танытқанымызбен, жарыстарда қолдана бермейміз.

Матрица дәрежесі

Егер A матрицасы шаршы матрица болса, онда A матрицасының дәрежесі A^k анықталған болады.

Оның анықтамасы матрицаларды көбейтуге негізделеді:

$$A^k = \underbrace{A \cdot A \cdot A \cdots A}_{k \text{ рет}}$$

Мысалы

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^3 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 48 & 165 \\ 33 & 114 \end{bmatrix}.$$

Оған қоса, A^0 бірлік матрицаға тең болады. Мысалы

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

A^k матрицасын 21.2-тарауда айтылған алгоритм арқылы $O(n^3 \log k)$ уақыт ішінде тиімді есептеуімізге болады. Мысалы:

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^8 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4 \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4.$$

Анықтауыш

Егер A шаршы матрица болса, A матрицасының анықтауышы $\det(A)$ анықталған болады. A -ның өлшемі 1×1 болса, онда $\det(A) = A[1, 1]$. Одан үлкен матрицаның анықтауышы рекурсивті формула арқылы есептелінеді.

$$\det(A) = \sum_{j=1}^n A[1, j] C[1, j],$$

мұндағы $C[i, j] - [i, j]$ позициясының алгебралық толықтауышы. Алгебралық толықтауыш төмендегі формула арқылы есептелінеді:

$$C[i, j] = (-1)^{i+j} \det(M[i, j]),$$

мұндағы $M[i, j]$ — i -жолы мен j -бағанасы өшірілген A матрицасы. Алгебралық толықтауыштың $(-1)^{i+j}$ коэффициентіне байланысты анықтауыштар аралата келе, оң және теріс болып бөлінеді. Мысалы

$$\det\left(\begin{bmatrix} 3 & 4 \\ 1 & 6 \end{bmatrix}\right) = 3 \cdot 6 - 4 \cdot 1 = 14$$

¹Сондай алгоритмдердің алғашқысын 1969 жылы Штрассен ойлап тапты [55]. Қазір алгоритм оның атымен аталады. Оның уақытша күрделілігі — $O(n^{2.80735})$; бүгінге дейін ең тиімді алгоритм $O(n^{2.37286})$ уақытта жұмыс істейді [56].

және

$$\det \begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix} = 2 \cdot \det \begin{pmatrix} 1 & 6 \\ 2 & 4 \end{pmatrix} - 4 \cdot \det \begin{pmatrix} 5 & 6 \\ 7 & 4 \end{pmatrix} + 3 \cdot \det \begin{pmatrix} 5 & 1 \\ 7 & 2 \end{pmatrix} = 81.$$

A -ның анықтаушы матрицаның кері матрицасы A^{-1} бар-жоғын айтады. Кері матрица дегеніміз $A \cdot A^{-1} = I$ орындалатын A^{-1} матрицасы, мұндағы I бірлік матрицасы. Егер $\det(A) \neq 0$ болса ғана, A^{-1} кері матрицасы бар болады және оны төмендегі формула бойынша есептей аламыз:

$$A^{-1}[i, j] = \frac{C[j, i]}{\det(A)}.$$

Мысалы:

$$\underbrace{\begin{pmatrix} 2 & 4 & 3 \\ 5 & 1 & 6 \\ 7 & 2 & 4 \end{pmatrix}}_A \cdot \underbrace{\frac{1}{81} \begin{pmatrix} -8 & -10 & 21 \\ 22 & -13 & 3 \\ 3 & 24 & -18 \end{pmatrix}}_{A^{-1}} = \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_I.$$

23.2 Сызықтық рекуренттілік

Сызықтық рекуренттілік – бастапқы мәндері $f(0), f(1), \dots, f(k-1)$ болатын және одан үлкен мәндері

$$f(n) = c_1 f(n-1) + c_2 f(n-2) + \dots + c_k f(n-k),$$

рекурсивті формула арқылы есептелетін функция, мұндағы c_1, c_2, \dots, c_k тұрақты коэффициенттер.

Динамикалық бағдарламалау арқылы $f(n)$ -нің кез келген мәнін $O(kn)$ уақытта $f(0), f(1), \dots, f(n)$ мәндерін біртіндеп есептеу арқылы табуға болады. Бірақ k кішкентай болса, $f(n)$ -ді матрица операциялары арқылы $O(k^3 \log n)$ уақытта әлдеқайда тиімдірек есептей аламыз.

Фибоначчи сандары

Сызықтық рекуренттілікке ең қарапайым үлгі Фибоначчи сандарын анықтайтын функция болмақ:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

Бұл жағдайда $k = 2$ және $c_1 = c_2 = 1$.

Фибоначчи сандарын тиімді есептеу үшін Фибоначчи формуласын өлшемі 2×2 болатын және келесі ара қатынас орындалатын шаршы матрица X арқылы көрсетейік:

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

Осылайша $f(i)$ және $f(i+1)$ мәндері X -ке "енгізу" ретінде берілген және X солар арқылы $f(i+1)$ және $f(i+2)$ мәндерін есептеп береді. X матрицасының мәні төмендегідей болады:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

Мысалы:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}.$$

Осылайша $f(n)$ -ді келесі формула арқылы есептеуімізге болады:

$$\begin{bmatrix} f(n) \\ f(n+1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

X^n мәні $O(\log n)$ уақытта есептелінеді, демек $f(n)$ -нің мәні де $O(\log n)$ уақытта есептелінеді.

Жалпы жағдай

Жалпы сызықтық рекуренттілік $f(n)$ функциясын қарастырайық. Біздің мақсатымыз қайтадан төмендегі орындалатын X матрицасын құру

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}.$$

Ондай матрицаның түрі осындай болады

$$X = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & c_{k-3} & \cdots & c_1 \end{bmatrix}.$$

Бастапқы $k-1$ жолда бір элемент бірге тең және қалған элементтер нөлге тең болады. Бұл жолдар $f(i)$ -ді $f(i+1)$ -мен, $f(i+1)$ -ді $f(i+2)$ -мен, т.с.с. ауыстырады. Соңғы жол болса, жаңа $f(i+k)$ мәнін есептеу үшін рекуренттіліктің коэффициенттерін қамтиды.

Енді $f(n)$ мәнін $O(k^3 \log n)$ уақытта төмендегі формула арқылы есептеуімізге болады:

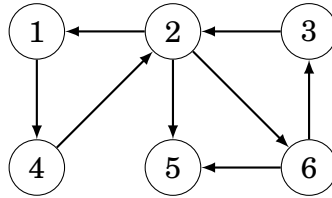
$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

23.3 Графтар мен матрицалар

Жолдар санау

Графтың сыбайластық матрицасының дәрежесі қызық қасиетке ие болады. Егер V салмақталмаған графтың сыбайластық матрицасы болса, V^n матрицасы n қыр арқылы өтетін төбелер арасындағы жолдар санын қамтиды.

Келесідегі графқа



сыбайластық матрица

$$V = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

болады. Төмендегі матрица

$$V^4 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

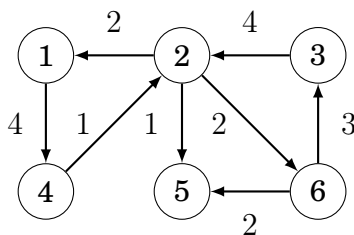
4 төбе арқылы өтетін төбелер арасындағы жолдар санын қамтиды. Мысалы, $V^4[2,5] = 2$, себебі 2-төбе мен 5-төбе арасында 4 қырдан өтетін 2 жол бар: $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$ және $2 \rightarrow 6 \rightarrow 3 \rightarrow 2 \rightarrow 5$.

Ең қысқа жол

Дәл жаңағы идеяны салмақталған графта қолдансақ, n қыр арқылы өтетін төбелер арасындағы минималды жолдың ұзындығын есептей аламыз. Оны

есептеу үшін матрицалардың көбейтуін жолдардың санын санамайтын, бірақ жолдардың ұзындығын минималдайтындай жаңа түрде анықтауымыз қажет.

Өрнек ретінде төмендегі графты қарастырайық:



Сыбайластық матрицасын құрастырайық. Егер қыр болмаса, ∞ деп анықтайық, әйтпесе соған сәйкес мәнімен анықтайық. Матрица –

$$V = \begin{bmatrix} \infty & \infty & \infty & 4 & \infty & \infty \\ 2 & \infty & \infty & \infty & 1 & 2 \\ \infty & 4 & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & \infty & 2 & \infty \end{bmatrix}.$$

Матрицаларды көбейту үшін төмендегі формуланың орнына

$$AB[i, j] = \sum_{k=1}^n A[i, k] \cdot B[k, j]$$

осы формуланы қолданамыз:

$$AB[i, j] = \min_{k=1}^n A[i, k] + B[k, j],$$

демек біз қосындының орнына минимумды және элементтердің көбейтіндісінің орнына қосындысын есептейміз. Осы өзгерістен кейін матрицаның дәрежесі графтың ең қысқа жолдарына сәйкес келеді.

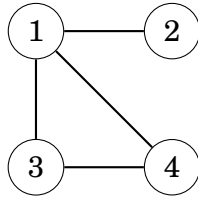
Мысалы:

$$V^4 = \begin{bmatrix} \infty & \infty & 10 & 11 & 9 & \infty \\ 9 & \infty & \infty & \infty & 8 & 9 \\ \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 12 & 13 & 11 & \infty \end{bmatrix},$$

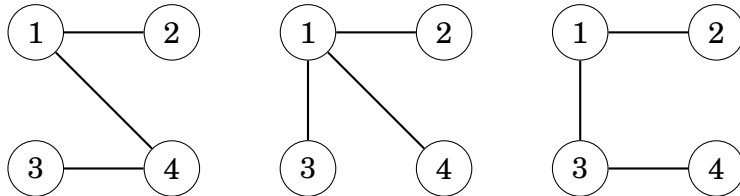
болғандықтан, 2-төбеден 5-төбеге дейін 4 қыр арқылы өтетін жолдың ең қысқа ұзындығы 8 болатынын қорытындылай аламыз. Ол жол – $2 \rightarrow 1 \rightarrow 4 \rightarrow 2 \rightarrow 5$.

Кирхгоф теоремасы

Кирхгоф теоремасы арнайы матрицаның анықтауышы арқылы графтың қаңқалы дарақтарының санын табуға мүмкіндік береді. Мысалы, төмендегі графта



үш қаңқалы дарақ бар:



Қаңқалы дарақтардың санын санау үшін Лаплас матрицасын L құрайық. Матрицада $L[i, i]$ i -төбенің дәрежесіне тең және егер i мен j арасында қыр болса, $L[i, j] = -1$, әйтпесе $L[i, j] = 0$ тең. Жоғарыдағы графқа сәйкес Лаплас матрицасы төмендегідей болады

$$L = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 2 & -1 \\ -1 & 0 & -1 & 2 \end{bmatrix}$$

Егер L матрицасының кез келген жолы мен бағанасын өшіріп, оның анықтауышын қарасақ, ол қаңқалы дарақтардың санына тең екенін көреміз.

Мысалы егер бірінші жол мен бірінші бағананы өшірсек, нәтижесінде

$$\det \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} = 3.$$

L матрицасының қандай да болсын жолы мен бағанасын өшірсек, анықтауышы әрдайым бірдей болады.

22.5-тарауындағы Кэли формуласы Кирхгоф теоремасының дербес жағдайы екенін көруімізге болады, себебі n төбелі толық граф төмендегідей болмақ:

$$\det \begin{pmatrix} n-1 & -1 & \cdots & -1 \\ -1 & n-1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ -1 & -1 & \cdots & n-1 \end{pmatrix} = n^{n-2}.$$

24-тарау. Ықтималдылық

Ықтималдылық – оқиғаның қаншалықты мүмкін болатынын көрсететін 0 мен 1 арасындағы нақты сан. Егер оқиға белгілі болатын болса, онда ықтималдылығы 1, ал оқиға мүмкін емес болса, онда ықтималдылығы 0 болады. Оқиғаның ықтималдылығы $P(\dots)$ түрінде белгілінеді, мұндағы үш нүкте оқиғаны көрсетеді.

Мысалы, ойын сүйегін лақтыратын болсақ, нәтижесі 1 мен 6 арасындағы бүтін сан және әр нәтиженің ықтималдылығы $1/6$ болады. Өрнек ретінде төмендегі ықтималдылықтарды есептей аламыз:

- $P(\text{"нәтиже 4 болады"}) = 1/6$
- $P(\text{"нәтиже 6 болмайды"}) = 5/6$
- $P(\text{"нәтиже жұп сан"}) = 1/2$

24.1 Есептеу

Оқиғаның ықтималдылығын есептеу үшін комбинаториканы қолдануымызға болады немесе оқиғаны тудыратын үдерісті симуляциялай аламыз. Өрнек ретінде араластырылған карта колодасынан мәні бірдей үш картаны алып шығу ықтималдылығын есептейік. (мысалы $\spadesuit 8$, $\clubsuit 8$ және $\diamondsuit 8$).

1-әдіс

Ықтималдылықты есептеу үшін келесі формуланы қолдануымызға болады:

$$\frac{\text{қажет нәтижелер саны}}{\text{жалпы нәтижелер саны}}.$$

Бұл есепте қажет нәтижелер – карталардың мәндері бірдей болған жағдай. Ондайдың $13\binom{4}{3}$ жолы бар: картаның мәніне 13 мүмкіндік бар және әр санға 4 түстен 3 түсті алу үшін $\binom{4}{3}$ жол бар.

Жалпы нәтижелердің саны – $\binom{52}{3}$: біз 52 карта ішінен 3 карта аламыз. Осылайша оқиғаның ықтималдылығы

$$\frac{13\binom{4}{3}}{\binom{52}{3}} = \frac{1}{425}.$$

2-әдіс

Ықтималдылықты есептеудің басқа жолы – оқиғаны тудыратын үдерісті симуляциялау. Бұл мысалда біз үш картаны алуымыз керек, демек үдеріс 3 қадамнан тұрады. Біз үдерістің әрбір қадамы сәтті болуын талап етеміз.

Ешқандай шектеу қойылмағандықтан, бірінші картаны алу әрдайым сәтті болады. Екінші картаны алу $3/51$ ықтималдылықпен сәтті болады, өйткені бізде 51 карта қалады және 3-үі бірінші картадағыдай мәнге ие болады. Дәл солай үшінші қадам $2/50$ ықтималдылықпен сәтті болады.

Толық үдерістің сәтті болуының ықтималдылығы

$$1 \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{1}{425}.$$

24.2 Оқиғалар

Ықтималдылық теориясында оқиғаны жиын ретінде көрсетуге болады

$$A \subset X,$$

мұнда X барлық нәтижелерді қамтиды және A – нәтижелердің ішжиыны. Мысалы, ойын сүйегін лақтырған кезде, нәтижелер төмендегідей болады:

$$X = \{1, 2, 3, 4, 5, 6\}.$$

Мысалы, ”нәтиже жұп сан” деген оқиғаға сәйкес жиын

$$A = \{2, 4, 6\}.$$

Әр x нәтижеге $p(x)$ ықтималдылығын белгілейік. Онда A оқиғасының ықтималдылығы $P(A)$ нәтижелердің ықтималдылықтарының қосындысы ретінде төмендегі формула арқылы есептеуге болады:

$$P(A) = \sum_{x \in A} p(x).$$

Мысалы, ойын сүйегін лақтырған кезде, әр x нәтижеге $p(x) = 1/6$ сәйкес келеді, демек ”нәтиже жұп сан” оқиғасының ықтималдылығы

$$p(2) + p(4) + p(6) = 1/2.$$

.

X -тің барлық нәтижелерінің ықтималдылықтарының қосындысы 1 болу керек, яғни $P(X) = 1$.

Ықтималдылық теориясындағы оқиғалар жиын болғандықтан, біз оған қарапайым жиын операцияларын қолдана аламыз:

- \bar{A} деп A -ға толықтыру оқиғасын белгілейді. Мысалы ойын сүйегін лақтырған кезде, $A = \{2, 4, 6\}$ оқиғасына толықтыру оқиғасы $\bar{A} = \{1, 3, 5\}$ болады.

- $A \cup B$ деп A мен B оқиғаларының біріктіруін белгілейді. Мысалы $A = \{2, 5\}$ және $B = \{4, 5, 6\}$ оқиғаларының біріктіруі – $A \cup B = \{2, 4, 5, 6\}$.
- $A \cap B$ деп A мен B оқиғаларының қиылысуын белгілейді. Мысалы $A = \{2, 5\}$ және $B = \{4, 5, 6\}$ оқиғаларының қиылысуы — $A \cap B = \{5\}$.

Толықтыру

\bar{A} толықтырудың ықтималдылығы келесі формула арқылы есептеледі:

$$P(\bar{A}) = 1 - P(A).$$

Берілген есепті толықтыруларды пайдаланып қарама-қарсы есепті шешу тәсілі арқылы оңай шешуге болады. Мысалы, 10 рет ойын сүйегін лақтырған кезде 6 санын кем дегенде бір мәрте көруіміздің ықтималдылығы:

$$1 - (5/6)^{10}.$$

Бұл жердегі $5/6$ – бір лақтырыстың нәтижесі 6 болмауының, ал $(5/6)^{10}$ – 10 лақтырыстың ешқайсысы 6 болмауының ықтималдылығы. Осының толықтыруы – берілген есептің жауабы.

Біріктіру

$A \cup B$ біріктіруінің ықтималдылығын төмендегі формула арқылы есептеуге болады:

$$P(A \cup B) = P(A) + P(B) - P(A \cap B).$$

Мысалы ойын сүйегін лақтырған кезде,

$$A = \text{”нәтиже жұп сан”}$$

оқиғасы мен

$$B = \text{”нәтиже 4-тен кем”}$$

оқиғаларының біріктіруі —

$$A \cup B = \text{”нәтиже жұп сан немесе 4-тен кем”},$$

және оның ықтималдылығы –

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) = 1/2 + 1/2 - 1/6 = 5/6.$$

Егер A мен B оқиғалары қиылыспайтын, яғни $A \cap B$ бос болса, $A \cup B$ оқиғасының ықтималдылығы –

$$P(A \cup B) = P(A) + P(B).$$

Шартты ықтималдылық

Шартты ықтималдылық

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

– B орын алғанда A оқиғасының ықтималдылығы. A –ның ықтималдылығын есептегенде, біз тек B –ға тиесілі нәтижелерді қарастырамыз.

Алдыңғы жиындарды қолдансақ,

$$P(A|B) = 1/3,$$

себебі B –ның нәтижелері $\{1, 2, 3\}$, және олардың біреуі ғана жұп. Бұл егер нәтиже $1 \dots 3$ арасында болса, нәтиженің жұп болуының ықтималдылығын білдіреді.

Қиылысу

Шартты ықтималдылықты қолданып, $A \cap B$ қиылысуының ықтималдылығын келесі формула арқылы есептеуге болады:

$$P(A \cap B) = P(A)P(B|A).$$

Егер

$$P(A|B) = P(A) \quad \text{және} \quad P(B|A) = P(B),$$

болса, A мен B оқиғалары тәуелсіз болады. Бұл B –ның орын алуы A –ның ықтималдылығына және A –ның орын алуы B –ның ықтималдылығына еш әсер бермейтінін білдіреді. Бұл жағдайдағы қиылысудың ықтималдылығы –

$$P(A \cap B) = P(A)P(B).$$

Мысалы, колодадан картаны алғанда,

$$A = \text{”картаның түсі жүрек”}$$

және

$$B = \text{”картаның мәні 4”}$$

екенін көреміз. Демек

$$A \cap B = \text{”картаның мәні 4 және түсі жүрек”}$$

оқиғасы

$$P(A \cap B) = P(A)P(B) = 1/4 \cdot 1/13 = 1/52$$

ықтималдылықпен орын алады.

24.3 Кездейсоқ шамалар

Кездейсоқ шама – кездейсоқ үдерістен құрылған мән. Мысалы, екі ойын сүйегін лақтырған кезде, келесі кездейсоқ шаманы анықтауға болады:

$$X = \text{”нәтижелердің қосындысы”}.$$

Егер нәтижелер $[4, 6]$ болса (яғни бірінші лақтырыстың нәтижесі төрт және екінші лақтырыстың нәтижесі алты), онда X -тің мәні 10 болады.

$P(X = x)$ деп X кездейсоқ шаманың x -ке тең болуының ықтималдығы деп белгілейміз. Мысалы, екі ойын сүйегін лақтырған кезде, $P(X = 10) = 3/36$, өйткені жалпы нәтижелердің саны 36 және қосындысы 10 болатын 3 нәтиже бар: $[4, 6]$, $[5, 5]$ және $[6, 4]$.

Математикалық күтім

Математикалық күтім $E[X]$ – X кездейсоқ айнымалының орташа мәні. Математикалық күтімді қосындылар арқылы есептеуге болады:

$$\sum_x P(X = x)x,$$

мұндағы x барлық X -тің мәндерінен өтіп шығады.

Мысалы, ойын сүйегін лақтырған кездегі нәтиженің математикалық күтімі

$$1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 4 + 1/6 \cdot 5 + 1/6 \cdot 6 = 7/2.$$

Математикалық күтімнің пайдалы қасиеті – сызықтық. Ол $E[X_1 + X_2 + \dots + X_n]$ қосындысы әрдайым $E[X_1] + E[X_2] + \dots + E[X_n]$ қосындысына тең дегенді білдіреді. Бұл формула егер кездейсоқ шамалар өзара тәуелді болса да орындалады.

Мысалы екі ойын сүйегін лақтырған кезде, математикалық күтім –

$$E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7.$$

Енді басқа есепті қарастырайық: n доп n қорапқа кездейсоқ салынған. Біздің тапсырмамыз – бос қораптар санының математикалық күтімін есептеу. Әр доп кез келген қорапқа бірдей ықтималдылықпен салына алады. Мысалы, егер $n = 2$ болса, нәтижелер төмендегідей болуы мүмкін:



Бұл жағдайда қораптың бос болуының математикалық күтімі –

$$\frac{0 + 0 + 1 + 1}{4} = \frac{1}{2}.$$

Жалпы жағдайда қораптың бос болуының ықтималдылығы –

$$\left(\frac{n-1}{n}\right)^n,$$

себебі оған ешқандай доп салынбауы керек. Осылайша, сызықтық қасиетті қолданған кездегі бос қораптардың санының математикалық күтімі –

$$n \cdot \left(\frac{n-1}{n} \right)^n.$$

Үлестірім

X кездейсоқ шаманың үлестірімі X -те бола алатын барлық мәндерінің ықтималдылығын көрсетеді. Үлестірім $P(X = x)$ мәндерінен тұрады. Мысалы, төменде екі ойын сүйегін лақтырған кездегі нәтижелердің қосындысының үлестірімі берілген:

x	2	3	4	5	6	7	8	9	10	11	12
$P(X = x)$	1/36	2/36	3/36	4/36	5/36	6/36	5/36	4/36	3/36	2/36	1/36

Бірқалыпты үлестірімде X кездейсоқ шамасында n мүмкін болатын $a, a+1, \dots, b$ мәндері бар және әр мәнінің ықтималдылығы $1/n$. Мысалы ойын сүйегін лақтырған кезде $a = 1$, $b = 6$ және әр x мәніне $P(X = x) = 1/6$ орындалады.

Бірқалыпты үлестірімдегі X мәнінің математикалық күтімі —

$$E[X] = \frac{a+b}{2}.$$

Биномдық үлестірімде n сынақ жасалады және әр сынақтың сәтті болуының ықтималдылығы — p . X кездейсоқ шамасы неше сынақ сәтті өткенін санайды және x мәнінің ықтималдылығы —

$$P(X = x) = p^x(1-p)^{n-x} \binom{n}{x},$$

мұндағы p^x және $(1-p)^{n-x}$ сәтті және сәтсіз сынақтарға сәйкес келсе, $\binom{n}{x}$ сынақтарды қанша жолмен жасауға болатынын көрсетеді.

Мысалы, ойын сүйегін он рет лақтырған кезде, 6 санын 3 рет көруіміздің ықтималдылығы — $(1/6)^3(5/6)^7 \binom{10}{3}$.

Биномдық үлестірімде X мәнінің математикалық күтімі

$$E[X] = pn.$$

Геометриялық үлестірімде сынақтың сәтті өтуінің ықтималдылығы — p және сынақтарды бірінші сәтті сынаққа дейін жалғастырамыз. X кездейсоқ шамасы қанша сынақ керектігін есептейді және x мәнінің ықтималдылығы

$$P(X = x) = (1-p)^{x-1}p,$$

мұндағы $(1-p)^{x-1}$ сәтсіз сынақтарға сәйкес келсе, p бірінші сәтті сынаққа сәйкес келеді.

Мысалы, ойын сүйегін алты санын көргенше лақтырамыз десек, лақтырыс санының 4 болуының ықтималдылығы $(5/6)^3 1/6$.

Геометриялық үлестірімде X мәнінің математикалық күтімі —

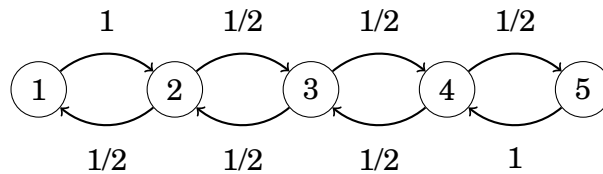
$$E[X] = \frac{1}{p}.$$

24.4 Марковтық тізбе

Марковтық тізбе — күйлер мен олардың ауысуларын қамтитын кездейсоқ үдеріс. Ауысулар бір күйден басқа күйлерге көшу ықтималдылығын көрсетеді. Марковтық тізбекті төбелері күй, қырлары ауысу болатын граф ретінде көрсетуге болады.

Өрнек ретінде мына есепті қарастырайық: n қабаттан тұратын ғимараттың біз бірінші қабатында тұрмыз. Әр қадам сайын біз кездейсоқ түрде бір қабат үстіге көтерілеміз немесе бір қабат астыға түсеміз. Бірақ бірінші қабатта біз әрдайым үстіге көтерілеміз және n -қабатта әрдайым төменге түсеміз. k қадамнан кейін m -қабатта болуымыздың ықтималдылығы қандай?

Бұл есепте, ғимараттың әр қабаты Марковтық тізбедегі әр күйге сәйкес келеді. Мысалы, егер $n = 5$ болса, граф төмендегідей болады:



Марковтық тізбенің ықтималдылық үлестірімі — $[p_1, p_2, \dots, p_n]$ векторы, мұндағы p_k — қазіргі күйіміз k болуының ықтималдылығы. $p_1 + p_2 + \dots + p_n = 1$ формуласы әрдайым орындалады.

Жоғарыдағы жағдайда бастапқы үлестірім — $[1, 0, 0, 0, 0]$, себебі біз бірінші қабаттан бастаймыз. Одан кейінгі үлестірім $[0, 1, 0, 0, 0]$, себебі біз бірінші қабаттан тек екінші қабатқа көше аламыз. Кейін біз төменге түсеміз не жоғарыға көтерілеміз, демек келесі үлестірім — $[1/2, 0, 1/2, 0, 0]$ және дәл солай кете береді.

Марковтық тізбедегі қадамдарды симуляциялаудың тиімді жолы — динамикалық бағдарламалауды қолдану. Динамикалық бағдарламалаудың идеясы — ықтималдылық үлестірімін сақтай отырып, әр қадамда барлық мүмкін болатын жолдарды қарастыру. Осы әдісті қолдана отырып, m қадамдық кездуді $O(n^2 m)$ уақытта симуляциялауымызға болады.

Марковтық тізбенің ауысуларын ықтималдылық үлестірімді жаңартатын матрица ретінде көрсетуге болады. Аталған жағдайға сәйкес төмендегідей матрица құрылады:

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix}.$$

Ықтималдылық үлестірімін осы матрицаға көбейткенде, бір қадам өткеннен кейінгі ықтималдылық үлестірімді алатын боламыз. Мысалы $[1, 0, 0, 0, 0]$ ықтималдылық үлестірімнен $[0, 1, 0, 0, 0]$ ықтималдылық үлестірімге келісі ретпен қозғалатын боламыз:

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

Матрицаның дәрежесін тиімді есептейтін болсақ, m қадамнан кейінгі үлестірімді $O(n^3 \log m)$ уақытта есептей аламыз.

24.5 Рандомизацияланған алгоритмдер

Кейде ықтималдылыққа байланысы жоқ есептерді шығару үшін кездейсоқтықты қолдануға болады. Рандомизацияланған алгоритм деп кездейсоқтыққа негізделген алгоритмді атаймыз.

Монте Карло алгоритмі — кейде қате жауап беретін рандомизацияланған алгоритм. Алгоритм пайдалы болу үшін жауаптың қате шығу ықтималдылығы төмен болуы тиіс.

Лас Вегас алгоритмі — орындалу уақыты әртүрлі болатын және әрдайым дұрыс жауап беретін алгоритм. Бұл жердегі мақсатымыз – жоғары ықтималдылықпен тиімді алгоритмді жобалау.

Біз төменде кездейсоқтықпен шығарылатын үш есепті қарастырамыз.

Реттік статистика

Жиымның k -реттік статистикасы – жиымды сұрыптағаннан кейін k -позицияда тұратын элемент. Кез келген реттік статистиканы $O(n \log n)$ уақытта жеңіл түрде есептеуге болады: басында жиымды сұрыптаймыз, кейін k -элементті аламыз. Бірақ бір элементті табу үшін барлық жиымды сұрыптау шыныменде қажет пе?

Реттік статистиканы жиымды сұрыптамай, рандомизацияланған алгоритм арқылы да табуға болады екен. Мұндай алгоритмдердің біріне Лас Вегас алгоритмі типіне кіретін quickselect алгоритмі жатады¹. Оның орындалу уақыты әдетте $O(n)$, бірақ ең жаман жағдайда $O(n^2)$ болады.

Алгоритм жиымның x кездейсоқ элементін алады, сосын x -тен кіші элементтерді жиымның сол жағына жылжытады, ал қалған барлық элементтерді жиымның оң жағына жылжытады. Егер жиымда n элемент бар болса, бұл $O(n)$ уақыт алады. Сол жағында a элемент және оң жағында b элемент бар делік. Егер $a = k$ болса, x элементі k -реттік статистика болады. Егер $a > k$ болса, біз k -реттік статистиканы рекурсивті түрде сол жақтан іздейміз. Ал егер $a < k$ болса, біз r -реттік статистиканы рекурсивті түрде оң жақтан іздейтін боламыз, мұндағы $r = k - a$. Ізденіс солай элемент табылғанға дейін қайталана береді.

¹1961 жылы Ч. А. Р. Хоар орташа есеппен тиімді екі алгоритмді, жиымдарды сұрыптайтын quicksort [57] және реттік статистиканы табатын quickselect [58] алгоритмдерін жариялады.

x элементі кездейсоқ түрде алынған кезде жиымның өлшемі әр қадам сайын екіге бөлінеді. Сондықтан k -реттік статистиканы табудың уақытша күрделілігі шамамен

$$n + n/2 + n/4 + n/8 + \dots < 2n = O(n).$$

Алгоритм ең жаман жағдайда $O(n^2)$ уақытты қажет етеді, өйткені x саны жиымдағы ең кіші не ең үлкен сан болып таңдалынуы мүмкін және ол кезде $O(n)$ қадам жасау керек болады. Бірақ ондай жағдайдың болуының ықтималдылығы өте төмен. Сондықтан бұл тәжірибеде мүлдем кездеспейді.

Матрицалардың көбейтіндісін тексеру

Келесі есебіміз – өлшемдері $n \times n$ болатын A , B және C матрицаларға $AB = C$ орындалатынын тексеру. Әрине бұл есепті шығару үшін AB көбейтіндісін қайтадан (қарапайым алгоритм арқылы $O(n^3)$ уақытта) есептесек болады, бірақ жауапты тексеру қайтадан есептеуден әлдеқайда оңайырақ болуы керек деген үміт бар.

Есепті уақытша күрделілігі $O(n^2)$ болатын Монте Карло алгоритмі¹ арқылы шығаруға болады екен. Алгоритмнің идеясы – қарапайым. Басында n кездейсоқ элементтерден тұратын X аламыз, сосын ABX және CX матрицаларын есептейміз. Егер $ABX = CX$ орындалса, $AB = C$ деп хабарлаймыз, әйтпесе $AB \neq C$ деп хабарлаймыз.

Алгоритмнің уақытша күрделілігі – $O(n^2)$, себебі ABX және CX матрицалары $O(n^2)$ уақытта есептелінеді. ABX матрицасын $A(BX)$ көрінісі арқылы тиімдірек есептеуімізге болады. Осылайша екі $n \times n$ және $n \times 1$ матрицаларын көбейту ғана қажет болады.

Алгоритмнің кемшілігі – $AB = C$ болуын кішкентай ықтималдылықпен қате баяндауында. Мысалы

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \neq \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix},$$

бірақ

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix}.$$

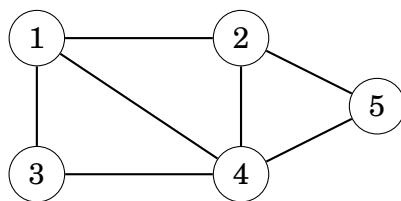
Дегенмен тәжірибеде алгоритмнің қате орындау ықтималдылығы төмен және сол ықтималдылықты бірнеше кездейсоқ X векторлармен тексеру арқылы одан сайын төмендете аламыз.

Графты бояу

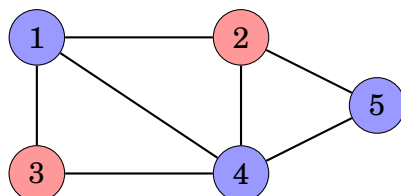
n төбелер мен m қырлардан тұратын граф берілген. Біздің тапсырмамыз – ең кемінде $m/2$ қырлардың шеттері әртүрлі түске ие болатындай етіп төбелерді бояу жолын табу.

Мысалы, төмендегі графта

¹Р. М. Фрейвалдс бұл алгоритмді 1977 жылы жариялады [59], оны кейде Фрейвалдс алгоритмі деп те атайды.



жарамды бояулардың бірі осындай болмақ:



Жоғарыдағы граф 7 қырдан тұрады және 5-еуінің шеттері әртүрлі түске ие, демек бұл бояу жарамды деп саналады.

Есепті жарамды бояуды тапқанша дейін кездейсоқ түрде бояйтын Лас Вегас алгоритмі арқылы шығаруға болады. Кездейсоқ бояуда әр төбенің түсі $1/2$ ықтималдылықпен тәуелсіз түрде таңдалады.

Кездейсоқ бояуда бір қырдың шеттері бірдей емес түске ие болуының ықтималдылығы – $1/2$. Демек қырлардың шеттері бірдей емес түске ие болуының математикалық күтімі – $m/2$. Кездейсоқ бояу жарамды деп күтілгендіктен, іс жүзінде жарамды бояуды тез табатын боламыз.

25-тарау. Ойындар теориясы

Бұл тарауда кездейсоқ элементтерді қамтымайтын, екі ойыншы ойнайтын ойындарды қарастыратын боламыз. Біздің мақсатымыз – қарсыласымыз қандай да бір жүріс жасағанда әрдайым жеңіске жетуді қамтамасыз ететін стратегияны табу немесе ондай стратегия жоқ екенін хабарлау.

Мұндай ойындарға арналған жалпы стратегия бар екені белгілі болып отыр және ойындарды ним теориясы арқылы талдауымызға болады. Алдымен ойыншылар таяқтар үйіндісінен таяқтарды алып отыратын ойынды талдаймыз, содан кейін стратегияны басқа ойындарға жалпылайтын боламыз.

25.1 Ойын күйлері

Басында n таяқтардан тұратын үйінді берілген ойынды қарастырайық. A және B ойыншылары кезек бойынша жүреді және A ойыншысы бірінші бастайды. Әр жүрісте ойыншы үйіндіден 1, 2 немесе 3 таяқ алу қажет. Ең соңғы таяқты алған ойыншы ойынды жеңеді.

Мысалы, егер $n = 10$ болса, ойын келесідей жалғаса алады:

- A ойыншысы 2 таяқты алады (8 таяқ қалды).
- B ойыншысы 3 таяқты алады (5 таяқ қалды).
- A ойыншысы 1 таяқты алады (4 таяқ қалды).
- B ойыншысы 2 таяқты алады (2 таяқ қалды).
- A ойыншысы 2 таяқты алады және ойынды жеңеді.

Ойын $0, 1, 2, \dots, n$ күйлерден тұрады, ол қалған таяқтар санына сәйкес келеді.

Ұтыс және ұтылыс күйлері

Ұтыс күйі – егер ойыншы оңтайлы түрде ойнайтын болса, әрдайым ұтатын күй. Ал ұтылыс күйі – егер қарсыласы оңтайлы түрде ойнайтын болса, әрдайым ұтылатын күй. Ойынның әр күйін ұтыс немесе ұтылыс күйлеріне топтастырсақ болады.

Жоғарыдағы ойында 0 күйі – анық ұтылыс күйі. Өйткені ойыншы ешқандай жүріс жасай алмайды. 1, 2 және 3 күйлері – ұтыс күйлер. Өйткені 1, 2 немесе 3 таяқты алсақ, ойынды ұтатын боламыз. 4 күйі – ұтылыс күй, себебі кез келген жүріс қарсылас үшін ұтыс күйге жетелейді.

Жалпы айтқанда егер қазіргі күйден ұтылыс күйіне апаратын жүріс болса, онда қазіргі күй ұтыс күйі болады. Әйтпесе қазіргі күй ұтылыс күйі болады. Осы бақылауды қолдана отыра ойынның барлық күйлерін топтастыруға болады.

Жоғарыдағы ойынның $0 \dots 15$ күйлері келесі ретпен топтастырылады (W ұтыс күйін белгілейді және L ұтылыс күйін белгілейді):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	W	W	W	L	W	W	W	L	W	W	W	L	W	W	W

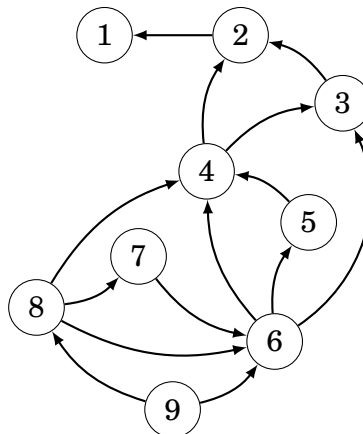
Бұл ойынды оңай түрде талдауға болады: егер k 4-ке бөлінсе, k күйі ұтылыс күйі болады, басқа жағдайда, ол ұтыс күйі болмақ. Бұл ойынды ұтудың оңай жолы – әрдайым таяқтардың саны 4-ке бөлінетін күйге апаратын жүріс жасау. Ақырында таяқтар қалмайды және қарсылас жеңіледі.

Бұл стратегия әрине біздің жүрісіміздегі таяқтар санының 4-ке бөлінбеуін талап етеді. Егер бөлінсе, біз ештеңе істей алмаймыз. Егер қарсыласымыз оңтайлы ойнайтын болса, әрдайым жеңеді.

Күйлер графы

Енді басқа таяқтар ойынын қарастырайық. Бұл ойынның шартына сәйкес әр k күйде k -дан кіші және k x -ке бөлінугі тиіс кез келген таяқтарды алуымызға болады. Мысалы 8 күйінде 1, 2 немесе 4 таяқтарды алсақ болады, бірақ 7 күйінде тек 1 таяқты алуға болады.

Келесі сурет ойынның $1 \dots 9$ күйлерін төбелері күй болатын және қырлары жүріс болатын күйлер графы ретінде көрсетеді:



Ойынның ақырғы күйі әрдайым 1, және ол ұтылыс күйі, себебі біз ешқандай жарамды жүріс жасай алмаймыз. $1 \dots 9$ күйлері төмендегідей топтастырылады:

1	2	3	4	5	6	7	8	9
L	W	L	W	L	W	L	W	L

Бір қызығы, бұл ойында жұп нөмірлі күйлер ұтыс күйлері болады, ал тақ нөмірлі күйлер ұтылыс күйлері болады.

25.2 Ним ойыны

Ним ойыны – ойындар теориясындағы маңызы зор қарапайым ойын. Сондықтан басқа ойындарды осы стратегия арқылы ойнауымызға болады. Алдымен ним ойынын қарастырамыз, одан әрі стратегияны басқа ойындарға жалпылаймыз.

Бізде n үйінділер бар және әр үйіндіде біршама таяқтар бар. Ойыншылар кезек бойынша жүреді және әр жүрісте ойыншы таяғы бар үйіндіден қалаған таяқтарын алады. Ең соңғы таяқты алған ойыншы жеңімпаз атанады.

Нимдағы күйлердің формасы – $[x_1, x_2, \dots, x_n]$, мұндағы x_k – k -үйіндідегі таяқтар саны. Мысалы $[10, 12, 5]$ күйі 10, 12 және 5 таяқтардан тұратын 3 үйіндіні белгілейді. $[0, 0, \dots, 0]$ күйі ұтылыс күйі болады. Өйткені бұл күйде ешқандай таяқ алынбайды және ол ақырғы күй болады.

Талдау

Нимнің әр күйін ним қосындысы $s = x_1 \oplus x_2 \oplus \dots \oplus x_n$ арқылы жеңіл түрде топтастыра аламыз, мұндағы \oplus – хог операциясы¹. Ним қосындысы 0 болатын күйлер ұтылыс күйі болады және одан басқа күйлер ұтыс күйлер болады. Мысалы $[10, 12, 5]$ -тің ним қосындысы $10 \oplus 12 \oplus 5 = 3$, демек бұл күй ұтыс күйі.

Бірақ ним қосындысының ним ойынымен қандай байланысы бар? Бұны ним күйі өзгергенде ним қосындысының қалай өзгеретінін көрсету арқылы түсіндіруге болады.

Ұтылыс күйлері: Ақырғы күй $[0, 0, \dots, 0]$ ұтылыс күйі болады және оның ним қосындысы күткеніміздей-ақ 0-ге тең. Басқа ұтылыс күйлерінде кез келген жүріс ұтыс күйіне апарады, себебі бір x_k мәні өзгергенде, ним қосындысы да өзгереді және осы жүрістен кейін ним қосындысы 0-ге тең болмайды.

Ұтыс күйлері: Егер $x_k \oplus s < x_k$ болатын k үйіндісі бар болса, ұтылыс күйге өтіп кетуіміз мүмкін. Бұл жағдайда k үйіндісінде $x_k \oplus s$ таяқ қалатындай етіп таяқтарды ала аламыз. Ал ол жағдай ұтылыс күйіне апарады. s санының ең сол жақтағы 1 тұратын биттің позициясында 1 тұратын x_k саны бар үйінді әрдайым кездеседі.

Өрнек ретінде $[10, 12, 5]$ күйін қарастырайық. Бұл күй – ұтыс күйі, себебі оның ним қосындысы 3-ке тең. Демек бұл жерде ұтылыс күйіне апаратын жүріс бар. Оны тауып көрейік.

Күйдің ним қосындысы төмендегідей болады:

10		1010
12		1100
5		0101
3		0011

Бұл жағдайда 10 таяғы бар үйінді — ним қосындысының ең сол жақтағы 1 тұратын биттің позициясында 1 тұратын жалғыз үйінді:

¹Ч.Л.Боутон нимнің оңтайлы стратегиясын 1901 жылы жариялады [60].

10	1010
12	1100
5	0101
3	0011

Үйіндінің жаңа өлшемі $10 \oplus 3 = 9$ болуы қажет, демек 1 таяқты ғана алуымыз керек. Осыдан кейін күйіміз $[9, 12, 5]$ болады, ал ол – ұтылыс күйі:

9	1001
12	1100
5	0101
0	0000

Мизер ойыны

Мизер ойынының шарты жоғарыдағы ойындарға қарама-қайшы. Бұл жерде ең соңғы таяқты алған ойыншы ұтылады. Мизер ним ойынын да қарапайым ним ойыны сияқты оңтайлы ойнауға болады екен.

Ойынның идеясы мизер ойынын қарапайым ойын сияқты бастап, ойынның соңында стратегияны өзгертуге негізделеді. Бірқатар жүрістен кейін әр үйіндіде ең көбі бір таяқ қалған кезде жаңа стратегияны қолдана бастаймыз.

Қарапайым ойында бір таяқтары бар үйінділер саны жұп болатындай жүрісті таңдауымыз керек болатын. Ал мизер ойынында бір таяқтары бар үйінділер саны тақ болатындай жүрісті таңдауымыз керек.

Ойын барысында стратегия өзгеретін күй орын алатындықтан және ол күй – ұтыс күйі болғандықтан, осы стратегия тиімді болады. Себебі бірден көп таяғы бар үйінді тек бір рет кездеседі, демек оның ним қосындысы 0-ге тең бола алмайды.

25.3 Шпраг-Гранди теоремасы

Шпраг-Гранди теоремасы¹ ним ойында қолданылған стратегияны келесі талаптарды орындайтын барлық ойындарға жалпылайды:

- Екі ойыншы кезектесіп жүреді.
- Ойын күйлерден тұрады және күйдегі мүмкін жүрістер кімнің жүргеніне тәуелді болмайды.
- Ойыншының жүру мүмкіндігі қалмағанда ойын аяқталады.
- Ойынның ерте ме, кеш пе бір аяқталатынына кепіл беріледі.
- Ойыншылардың күйлер мен мүмкін болатын жүрістерден толық хабары бар және ойында кездейсоқтық жоқ.

Идеясы ойынның әр күйіне ним үйіндісіндегі таяқтар санына сәйкес Гранди санын есептеуге негізделеді. Барлық күйлердің Гранди санын білген соң, ним ойыны сияқты ойынды ойнай аламыз.

¹Теореманы П.Шпраг [61] және П.М.Гранди [62] бір-бірінен тәуелсіз ашты.

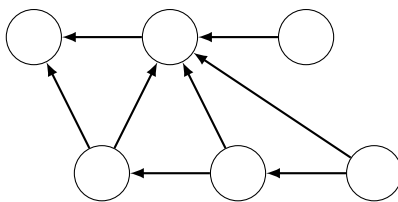
Гранди сандары

Ойын күйінің Гранди саны —

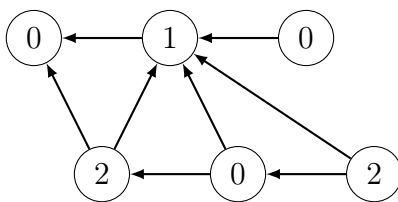
$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

мұндағы g_1, g_2, \dots, g_n — қазіргі күйден жүріс жасауға болатын күйлердің Гранди сандары, ал mex функциясы жиында жоқ ең минималды оң санды береді. Мысалы $\text{mex}(\{0, 1, 3\}) = 2$. Егер күйде жүріс болмаса, онда оның Гранди саны 0 болады, себебі $\text{mex}(\emptyset) = 0$.

Мысалы, төмендегі күйлер графының



Гранди сандары келесідей:

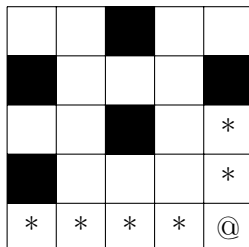


Ұтылыс күйінің Гранди саны — 0, ал ұтыс күйінің Гранди саны — оң сан.

Күйдегі Гранди саны ним үйіндісіндегі таяқтар санына сәйкес келеді. Егер Гранди саны 0 болса, біз тек Гранди саны оң болатын күйлерге бара аламыз. Ал егер Гранди саны $x > 0$ болса, онда біз Гранди саны $0, 1, \dots, x - 1$ болатын барлық күйлерге бара аламыз.

Мысал ретінде ойыншылар лабиринттегі фигураны жылжытатын ойынды қарастырайық. Лабиринттегі әр шаршы не еденді, не қабырғаны білдіреді. Әр жүріс сайын ойыншы фигураны бірнеше қадам солға немесе үстіге жылжытуы керек. Соңғы жүрісті жасаған ойыншы жеңіске жетеді.

Келесі сурет ойынның күйін көрсетеді, мұнда @ таңбасы фигураның орналасуын белгілейді және * таңбасы жүріс жасай алатын торларды белгілейді.



Ойынның күйлері — лабиринт еденіндегі барлық шаршылар. Жоғарыда келтірілген лабиринттегі Гранди сандары келесідей:

0	1		0	1
	0	1	2	
0	2		1	0
	3	0	4	1
0	4	1	3	2

Лабиринттегі әр күй ним ойындағы үйіндіге сәйкес келеді. Мысалы, төменгі оң жақтағы шаршының Гранди саны 2, демек бұл – ұтыс күйі. Ал фигураны 4 қадам солға немесе 2 қадам үстіге жылжытсақ, ұтылыс күйіне жетеміз.

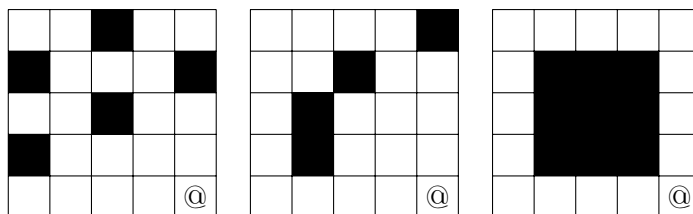
Ним ойынына қарағанда қазіргі күйдің Гранди санынан үлкен күйге өтуге болатынын ескергеніміз жөн. Бірақ қарсылас сондай жүрістің күшін жоятын жүрісті әрдайым таңдай алады. Демек ұтылыс күйінен қашу мүмкін емес.

Ішойындар

Ойын ішойындардан тұрады делік. Әр жүрісте ойыншы бірінші ішойынды, содан кейін сол ішойындағы жүрісті таңдайды. Ойыншылардың ешқайсысы ешбір ішойында жүріс жасай алмайтындай болған кезде ойын аяқталады.

Бұл жағдайда ойынның Гранди саны ішойындардың Гранди сандарының ним қосындысына тең болады. Ойынды ішойындардың барлық Гранди сандарын, содан кейін олардың ним қосындысын есептеу арқылы қарапайым ним ойыны сияқты ойнауға болады.

Мысалы 3 лабиринттен тұратын ойынды қарастырайық. Бұл ойында әр жүрісте ойыншы бір лабиринтті таңдап, соның ішіндегі фигураны жылжытады. Ойынның бастапқы күйі келесідей делік:



Лабиринттердің Гранди сандары келесідей:

0	1		0	1
	0	1	2	
0	2		1	0
	3	0	4	1
0	4	1	3	2

0	1	2	3	
1	0		0	1
2		0	1	2
3		1	2	0
4	0	2	5	3

0	1	2	3	4
1				0
2				1
3				2
4	0	1	2	3

Әуелгі күйде Гранди сандарының ним қосындысы $2 \oplus 3 \oplus 3 = 2$, демек бірінші ойыншы жеңеді. Тиімді жүрістердің бірі – бірінші лабиринтте 2 қадам фигураны алдығы жылжыту. Ол $0 \oplus 3 \oplus 3 = 0$ ним қосындысын береді.

Гранди ойыны

Кейде ойындағы жүріс ойынды өзара тәуелсіз бірнеше ішойындарға бөледі. Ондай кездегі ойынның Гранди саны

$$\text{mex}(\{g_1, g_2, \dots, g_n\}),$$

мұндағы n мүмкін болатын жүрістер саны және

$$g_k = a_{k,1} \oplus a_{k,2} \oplus \dots \oplus a_{k,m},$$

мұндағы k жүрісі Гранди сандары $a_{k,1}, a_{k,2}, \dots, a_{k,m}$ болатын ішойындарды өндіреді.

Сондай ойындардың мысалы ретінде Гранди ойынын келтіруімізге болады. Басында бізге n таяқтан тұратын бір үйінді берілген делік. Әр жүрісте ойыншы үйіндіні таңдап, бос емес және өлшемдері әртүрлі екі үйіндіге бөледі. Соңғы болып жүрген ойыншы жеңеді.

$f(n)$ деп n таяқтан тұратын үйіндідегі Гранди санын белгілейік. Гранди санын үйіндіні екі үйіндіге бөлудің барлық жолдарын қолдану арқылы есептеуімізге болады. Мысалы $n = 8$ болған кезде, бөлу мүмкіндіктері $1 + 7$, $2 + 6$ және $3 + 5$ болады. Демек

$$f(8) = \text{mex}(\{f(1) \oplus f(7), f(2) \oplus f(6), f(3) \oplus f(5)\}).$$

Осы ойында $f(n)$ мәні $f(1), \dots, f(n-1)$ мәндеріне негізделеді. Функцияның негізгі жағдайлары $f(1) = f(2) = 0$, себебі 1 және 2 таяқтан тұратын үйінділерді бөлу мүмкін емес. Алғашқы Гранди сандары:

$$\begin{aligned} f(1) &= 0 \\ f(2) &= 0 \\ f(3) &= 1 \\ f(4) &= 0 \\ f(5) &= 2 \\ f(6) &= 1 \\ f(7) &= 0 \\ f(8) &= 2 \end{aligned}$$

Гранди саны $n = 8$ үшін 2-ге тең, демек ойынды ұтуға болады. Ұтыс жүріске $1 + 7$ үйінділерін құру арқылы қол жеткіземіз. Себебі $f(1) \oplus f(7) = 0$ болады.

26-тарау. Жолды өңдеу алгоритмдері

Бұл тарауда жолдарды тиімді өңдейтін алгоритмдер қарастырылады. Жолға байланысты көптеген мәселелерді $O(n^2)$ уақытта шешуге болады, бірақ $O(n)$ немесе $O(n \log n)$ уақытта жұмыс істейтін алгоритмдерді табу қиынырақ болады.

Мысалы жолды өңдейтін негізгі есептердің бірі – үлгімен салғастыру есебі. Бізге ұзындығы n болатын жол берілген және ұзындығы m болатын үлгі берілген. Біздің тапсырмамыз – берілген жолда қанша рет үлгі кездесетінін табу. Мысалы егер үлгі ABC болса, ол ABABCSBABC жолында 2 мәрте кездеседі.

Үлгімен салғастыру есебі жолдың барлық позицияларын үлгімен басталуына тексеретін толық ізденіс алгоритмі арқылы $O(nm)$ уақытта жеңіл шығарылады. Дегенмен бұл тарауда тек $O(n + m)$ уақыт қажет ететін тиімдірек алгоритмдер бар екенін көреміз.

26.1 Жол терминологиясы

Бүкіл тарау бойынша жолды нөлмен индекстейтін боламыз. Демек ұзындығы n болатын жол $s[0], s[1], \dots, s[n-1]$ таңбалардан тұрады. Жолда кездесе алатын таңбалардың жиынтығы әліпби деп аталады. Мысалы $\{A, B, \dots, Z\}$ әліпбиі ағылшын бас әріптерінен тұрады.

Ішжол – жолдың қатар келетін әріптер тізбегі. Біз $s[a \dots b]$ нотаиясын a -дан басталатын b -дан бітетін ішжолға қолданатын боламыз. Ұзындығы n болатын жолда $n(n+1)/2$ ішжол бар. Мысалы, ABCD жолының ішжолдары – A, B, C, D, AB, BC, CD, ABC, BCD және ABCD.

Іштізбек деп тізбектегі кейбір элементтерді алып тастай отырып, бірақ қалған элементтердің орналасу тәртібін өзгертпестен жасауға болатын тізбекті айтамыз. Ұзындығы n болатын жолда $2^n - 1$ іштізбек бар. Мысалы, ABCD жолдың іштізбектері – A, B, C, D, AB, AC, AD, BC, BD, CD, ABC, ABD, ACD, BCD және ABCD.

Префикс – жолдың басынан басталатын ішжол, ал суффикс – жолдың соңында аяқталатын ішжол. Мысалы, ABCD жолының префикстері – A, AB, ABC және ABCD. Ал суффикстері – D, CD, BCD және ABCD.

Айналым жолдың басындағы таңбаларды бір-бірден соңына жылжыту (немесе керісінше) арқылы өндіріледі. Мысалы, ABCD жолының айналымдары – ABCD, BCDA, CDAB және DABC.

Период – өзін қайталау арқылы жолды құрастыруға болатын жолдың

префиксі. Соңғы қайталау толық болмауы мүмкін, яғни соңғы қайталау периодтың префиксы болуы мүмкін. Мысалы, ABCABCA жолының ең қысқа периоды ABC.

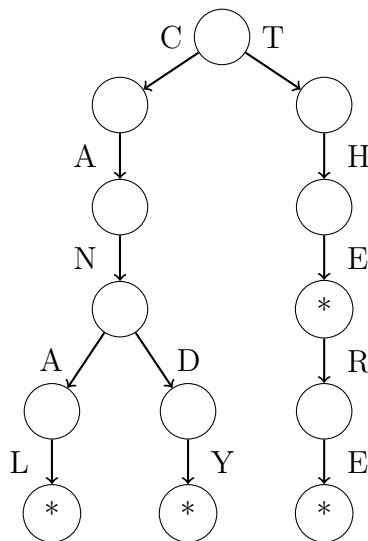
Шек – жолдың префиксі және суффиксі болатын жол. Мысалы, ABACABA жолының шектері – A, ABA және ABACABA.

Қарапайым тілмен айтқанда, лексикографиялық реттілік – сөздердің сөздіктерде берілу реттілігі. Ресми анықтамасы мынадай: егер $p_i < q_i$ болатын i позициясы болса және барлығы үшін $j < i$, $p_j = q_j$ болса, онда p жолы лексикографиялық тұрғыдан q жолынан кіші болады. Егер мұндай i позициясы болмаса, p ұзындығы q ұзындығынан кем болған жағдайда p лексикографиялық тұрғыдан q ұзындығынан кіші болады. Мысалы, $abdc < abe$ және $abc < abcd$, мұндағы p ұзындығы q -дан лексикографиялық тұрғыдан кіші болса, $p < q$ деп жазамыз.

26.2 Префикс дарағы

Префикс дарағы – жолдардың жиынын қолдайтын түбірлі дарақ. Жиынның әрбір жолы түбірден басталатын таңбалар тізбегі ретінде сақталады. Егер екі жолдың ортақ префиксі болса, олар дарақта да ортақ тізбекке ие болады.

Мысалға келесі префикс дарағын қарастырайық:



Бұл дарақ {CANAL, CANDY, THE, THERE} жиынына сәйкес. Төбедегі * таңбасы жиынның бір жолы осы төбеде аяқталатынын көрсетеді. Ондай таңба қажет-ақ, өйткені бір жол екінші жолдың префиксі болуы мүмкін. Мысалы, жоғарыдағы префикс дарағында THE жолы THERE жолының префиксі болады.

Ұзындығы n болатын жолдың дарақта бар-жоғын $O(n)$ уақытта тексере аламыз, өйткені біз түбір төбеден басталатын тізбек бойынша ере аламыз. Сондай-ақ ұзындығы n болатын жолды басында тізбек бойынша ере отырып, кейін қажеттілігіне қарай дараққа жаңа төбелерді тіркей отырып, $O(n)$ уақытта дараққа қоса аламыз.

Префикс дарағын қолдана отырып, берілген жолдың жиымға жататын ең ұзын префиксін таба аламыз. Сондай-ақ, әр төбеде қосымша ақпаратты сақтай отырып, біз жиымға жататын және префиксі берілген жол болатын жолдар санын есептей аламыз.

Префикс дарағын келесі жиым ретінде сақтауға болады:

```
int trie[N][A];
```

мұндағы N – төбелердің максималды саны (жиындағы барлық жолдардың жиынтық ұзындығы) және A – әліпбидің ұзындығы. Префикс дарағындағы төбелер $0, 1, 2, \dots$ сандарымен нөмірленген. Түбірдің нөмірі – 0 және $\text{trie}[s][c]$ – s төбеден c таңбасы арқылы өткен кездегі тізбектегі келесі төбе.

26.3 Жол хәші

Жол хәші арқылы екі жолдардың тең екенін оңтайлы тексеруге болады.¹ Идеясы жолдардың таңбаларын жекелеп тексерудің орнына жолдардың хәштерін тексеруге негізделеді.

Хәшті есептеу

Жолдың хәші – жолдың таңбалары арқылы есептелген сан. Егер жолдар бірдей болса, олардың хәштері де бірдей болады. Демек бұл жолдардың хәші арқылы жолдарды салыстыруға мүмкіндік береді.

Әдетте жолдың хәшін көпмүшелік хәш арқылы есептейді. Ондай кезде ұзындығы n болатын жолдың хәші

$$(s[0]A^{n-1} + s[1]A^{n-2} + \dots + s[n-1]A^0) \bmod B,$$

мұндағы $s[0], s[1], \dots, s[n-1]$ – s жолының таңбалар коды, A және B – алдын ала таңдалған тұрақтылар.

Мысалы, ALLEY жолы таңбаларының коды келесідей:

A	L	L	E	Y
65	76	76	69	89

Демек егер $A = 3$ және $B = 97$ болса, ALLEY жолының хәші –

$$(65 \cdot 3^4 + 76 \cdot 3^3 + 76 \cdot 3^2 + 69 \cdot 3^1 + 89 \cdot 3^0) \bmod 97 = 52.$$

Алдын ала өңдеу

Көпмүшелік хәш арқылы s жолының кез келген i шжолының хәшін $O(n)$ уақыт алдын ала өңдеуден кейін $O(1)$ уақытта есептей аламыз. Идеясы $h[k]$ $s[0 \dots k]$ префиксінің хәшінен тұратын h жиымын құруға негізделеді.

¹Бұл әдіс Рабин-Карп үлгімен салғастыру алгоритмінің арқасында танымал болды [63].

Жиынның мәндерін рекурсивті түрде келесі түрде есептеуімізге болады:

$$\begin{aligned}h[0] &= s[0] \\h[k] &= (h[k-1]A + s[k]) \bmod B\end{aligned}$$

Оған қоса, $p[k] = A^k \bmod B$ болатындай p жиынын құрастырамыз:

$$\begin{aligned}p[0] &= 1 \\p[k] &= (p[k-1]A) \bmod B.\end{aligned}$$

Сол екі жиынды құрастыру $O(n)$ уақыт алады. Одан кейін кез келген $s[a \dots b]$ ішжолдың хэші $O(1)$ уақытта келесі формула арқылы есептеледі

$$(h[b] - h[a-1]p[b-a+1]) \bmod B$$

мұндағы $a > 0$. Ал егер $a = 0$ болса, хэші жай $h[b]$ болады.

Қолданысы

Хэштер арқылы жолдарды тиімді салыстыруға болады. Жеке таңбаларды салыстырудың орнына, олардың хэштерін салыстырсақ жеткілікті. Егер хэштердің мәні бірдей болса, жолдардың да бірдей болуы мүмкін, ал егер хэштердің мәні әртүрлі болса, онда жолдар сөзсіз бірдей емес болады.

Хэш қолдана отыра, біз әдетте оңтайлы толық ізденіс жасай аламыз. Үлгі ретінде салғастыру есебін қарастырайық: Бізге s жолы мен p үлгісі берілген. s -та p үлгісі кездесетін барлық позицияларды табу керек. Толық ізденіс алгоритмі p кездесе алатын барлық позициялар арқылы өтіп, жолдардың таңбаларын жекелеп салыстырады. Ондай алгоритмнің уақытша күрделілігі – $O(n^2)$.

Толық ізденіс алгоритмін хэштерді қолдана отырып, тиімдірек қыла аламыз, өйткені алгоритм жолдардың ішжолдарын салыстырады. Хэштарды қолдану арқылы әр салыстыру $O(1)$ уақыт алады, себебі ішжолдардың хэштері ғана салыстырылады. Бұл уақытша күрделілігі $O(n)$ болатын алгоритмге әкеледі. Ал ол – бұл есепті шығару үшін ең жақсы уақытша күрделілігі.

Хэштер мен бинарлық ізденісті бірге қолдана отырып, екі жолдың лексикографиялық ретін логарифмдік уақытта анықтауға болады. Ол үшін бинарлық ізденіс арқылы жолдардың ортақ префиксінің ұзындығын табамыз. Сосын сол префикстен кейінгі таңбаны қараймыз, себебі ол реттілікті анықтайды

Қақтығыстар мен параметрлер

Хэштерді салыстырғанда туындайтын айқын қауіп – қақтығыс. Ол – екі бірдей емес жолдардың бірдей хэштерге ие болуы. Мұндай жағдайда хэштердің мәніне сүйенетін алгоритм жолдар бірдей деп тұжырымдауы мүмкін. Ал шын мәнінде жолдар бірдей емес болады, сөйтіп алгоритм қате шешім шығарады.

Мүмкін болатын жолдардың саны мүмкін болатын хэштердің санынан үлкен болуына байланысты қақтығыстар әрдайым орын алып тұрады. Дегенмен

A мен B тұрақтылары мұқият таңдалса, қақтығыстың болу ықтималдылығы төмендейді. Әдетте A мен B -ны 10^9 -на жақын үлкен кездейсоқ тұрақты етіп алады. Мысалы:

$$A = 911382323$$

$$B = 972663749$$

Осындай тұрақтыларды қолдана отырып, long long типін хэштарды есептеу үшін пайдалануға болады. Себебі AB мен BB көбейтінділері long long типіне сияды. Хештің шамамен 10^9 әртүрлі мәнін алу үшін осы жеткілікті ме?

Хэштер қолданыла алатын келесі 3 жағдайды қарастырайық:

1-жағдай: x және y жолдары өзара салыстырылады. Егер хэштердің барлық мәндері бірдей ықтималдылыққа ие болса, қақтығыс болуының ықтималдылығы $1/B$ тең.

2-жағдай: x жолы y_1, y_2, \dots, y_n жолдарымен салыстырылады. Бір немесе одан көп қақтығыстардың болуының ықтималдылығы

$$1 - (1 - \frac{1}{B})^n.$$

3-жағдай: x_1, x_2, \dots, x_n жолдардың барлық жұптары бір-бірімен салыстырылады. Бір немесе одан көп қақтығыстардың болуының ықтималдылығы

$$1 - \frac{B \cdot (B-1) \cdot (B-2) \cdots (B-n+1)}{B^n}.$$

Келесі кесте $n = 10^6$ және B мәні өзгерген кездегі қақтығыстардың болуының ықтималдығын көрсетеді:

тұрақты B	1-жағдай	2-жағдай	3-жағдай
10^3	0.001000	1.000000	1.000000
10^6	0.000001	0.632121	1.000000
10^9	0.000000	0.001000	1.000000
10^{12}	0.000000	0.000000	0.393469
10^{15}	0.000000	0.000000	0.000500
10^{18}	0.000000	0.000000	0.000001

Кесте 1-жағдайда $B \approx 10^9$ болған кезде қақтығыстың болуының ықтималдылығы болмашы екенін көрсетеді. 2-жағдайда қақтығыс болуы мүмкін, бірақ оның ықтималдылығы төмен. Бірақ 3-жағдайда $B \approx 10^9$ болған кезде қақтығыс әрқашан дерлік болады.

3-жағдайдағы құбылыс туған күн парадоксы деп аталады: егер бөлмеде n адам болса, онда екі адамда бірдей туған күн болуының ықтималдылығы n кішкентай болса да үлкен. Хэштерде сәйкесінше егер барлық хэштер өзара салыстырылса, екі хэштердің бірдей болуының ықтималдылығы үлкен.

Қақтығыс болуының ықтималдылығын бірнеше әртүрлі параметрлер қолданатын хэштер арқылы төмендетсек болады. Барлық хэштерде қақтығыс болуының ықтималдығы екіталай. Мысалы, параметрі $B \approx 10^9$ болатын

екі хэш параметрі $B \approx 10^{18}$ болатын бір хэшке сәйкес. Ол болса, қақтығыс ықтималдылығын төмендетеді.

Кейбір адамдар $B = 2^{32}$ және $B = 2^{64}$ тұрақтыларын қолданады. 32 және 64-бит бүтін сандардың операциялары 2^{32} және 2^{64} модулімен есептелгендіктен ыңғайлы болады. Дегенмен бұл жақсы шешім емес, себебі 2^x формасында болған тұрақтыларға әрдайым қақтығыс жасайтын енгізу [64] құрауымызға болады.

26.4 Z-алгоритм

Ұзындығы n болатын s жолының Z -жиымы z әрбір $k = 0, 1, \dots, n-1$ үшін k позицияда басталатын, s жолының префиксі болатын ең ұзын ішжолдың ұзындығын қамтиды. Демек $z[k] = p$ теңдігі $s[0 \dots p-1]$ ішжолы мен $s[k \dots k+p-1]$ ішжолы тең екенін хабарлайды. Көптеген жолдарға байланысты есептер Z -жиымы арқылы тиімді шығарыла алады.

Мысалы, АСВАССАСВАСВАСДА жолының Z -жиымы келесідей:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1

Бұл жағдайда, $z[6] = 5$, себебі ұзындығы 5 болатын АСВАС ішжолы s жолының префиксі болады, бірақ ұзындығы 6 болатын АСВАСВ ішжолы s жолының префиксі бола алмайды.

Алгоритм

Бұдан әрі біз Z -жиымын $O(n)$ уақытта құрастыратын Z -алгоритм¹ деп аталатын алгоритмді қарастырамыз. Алгоритм Z -жиымның мәндерін оған дейін сақталған Z -жиымның мәндерін қолдана отырып, ішжолдардың таңбаларын жеке салыстыра келе, солдан оңға қарай есептейді.

Z -жиымның мәндерін тиімді есептеу үшін алгоритм $s[x \dots y]$ ішжолы s жолының префиксі болатын және y ең максималды болатын $[x, y]$ аралығын қолдайды. $s[0 \dots y-x]$ және $s[x \dots y]$ тең болғандықтан, оны $x+1, x+2, \dots, y$ позицияларының Z -мәнін есептеуге қолдана аламыз.

Әр k позициясына біз алдымен $z[k-x]$ мәнін тексереміз. Егер $k+z[k-x] < y$ болса, онда $z[k] = z[k-x]$ екенін білеміз. Бірақ $k+z[k-x] \geq y$ болса, $s[0 \dots y-k]$ ішжолы $s[k \dots y]$ ішжолына тең болады және $z[k]$ мәнін анықтау үшін ішжолдардың жеке таңбаларын салыстыру керек. Дегенмен біз салыстыруды $y-k+1$ және $y+1$ позицияларында бастайтын болғандықтан, алгоритм $O(n)$ уақыт жұмыс істейді.

Мысалы, келесі Z -жиымын құрастырайық:

¹ Z -алгоритмді [65] мақаласында үлгімен салғыстырудың танымал қарапайым сызықтық әдісі ретінде ұсынылды, бастапқы идея [66] тиесілі.


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

$z[6] = 5$ есептеп болғаннан кейін қазіргі $[x, y]$ аралығы $[6, 10]$ тең болады:

						x		y							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	?	?	?	?	?	?	?	?	?


Енді келесі Z-жиымның мәндерін тиімді есептей аламыз, себебі $s[0...4]$ мен $s[6...10]$ тең екенін білеміз. Біріншіден, $z[1] = z[2] = 0$ болғандықтан, $z[7] = z[8] = 0$ екенін де бірден білеміз:

						x		y							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?



Кейін $z[3] = 2$ болғандықтан, $z[9] \geq 2$ екенін білеміз:

						x		y							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?



Дегенмен 10-позициядан кейінгі жол жайында еш ақпарат болмағандықтан, ішжолдардың таңбаларын жеке салыстыруға тура келеді:

						x		y							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
–	0	0	2	0	0	5	0	0	?	?	?	?	?	?	?

$z[9] = 7$ екені белгілі болған соң, жаңа $[x, y]$ аралығы $[9, 15]$ аралығына тең болды:

									x							y
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A	
–	0	0	2	0	0	5	0	0	7	?	?	?	?	?	?	

Одан кейін барлық Z-жиынның мәндерін Z-жиымда оған дейін сақталған ақпараттар арқылы анықтауымызға болады:

									x							y
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A	
–	0	0	2	0	0	5	0	0	7	0	0	2	0	0	1	

Қолданысы

Хэшті немесе Z-алгоритмін қолдану әдетте талғамға байланысты болады. Хэшке қарағанда Z-алгоритмі әрдайым тиімді жұмыс істейді және қақтығыс болу ықтималдығы жоқ. Дегенмен Z-алгоритмін жазу қиынырақ және кейбір есептерді тек хэш арқылы шығаруға болады.

Мысал ретінде үлгі салғыстыру есебін қайтадан қарастырайық. Есепте p үлгісінің қайда кездесетінін табу керек. Біз бұл есепті осыған дейін хэш арқылы тиімді шығардық, дегенмен Z-алгоритмі есепті шешудің басқа жолын ұсынады.

Жол өңдеудегі әдеттегі идея арнайы таңбамен ажыратылған бірнеше жолдардан тұратын жолды құрастыруға негізделеді. Бұл есепте $p\#s$ жолын құрастыра аламыз, мұнда p мен s жолдарда кездеспейтін $\#$ арнайы таңбасы арқылы ажыратылған. $p\#s$ жолының Z-жиымы p -ның s -те қай позицияларда кездесетінін хабарлайды: олар – мәні p ұзындығы болатын позициялар.

Мысалы, егер $s = \text{НАТТИВАТТИ}$ және $p = \text{АТТ}$ болса, Z-жиымы келесідей болады:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	T	T	#	H	A	T	T	I	V	A	T	T	I
–	0	0	0	0	3	0	0	0	0	3	0	0	0

5 және 10-позициядағы элементтердің мәндері – 3, демек АТТ үлгісі НАТТИВАТТИ жолындағы сәйкес позицияларда кездеседі.

Z-жиымды құрастырып, оның мәндерінен өтіп шығу жеткілікті болғандықтан нәтижелік алгоритмде сызықтық уақытша күрделігі сақталады.

Коды

Төменде Z-жиымына сәйкес векторды қайтаратын Z-алгоритмінің қысқа коды берілген:

```
vector<int> z(string s) {  
    int n = s.size ();  
    vector<int> z(n);  
    int x = 0, y = 0;  
    for (int i = 1; i < n; i++) {  
        z[i] = max(0,min(z[i-x],y-i+1));  
        while (i+z[i] < n && s[z[i]] == s[i+z[i]]) {  
            x = i; y = i+z[i]; z[i]++;  
        }  
    }  
    return z;  
}
```

27-тарау. Квадраттық түбір алгоритмдер

Квадраттық түбір алгоритмі – уақытша күрделілігінде квадрат түбірі бар алгоритм. $O(\sqrt{n})$ күрделілігі $O(n)$ -нен тез болғанымен, $O(\log n)$ -нен баяу болуына байланысты оны ”кемшілікті логарифм” ретінде де қарастыруға болады. Қандай жағдай болмасын көптеген түбір алгоритмдер қолданыста жылдам, ыңғайлы және тиімді болып келеді.

Мысалға төмендегі есепті қарастырайық: берілген позициядағы элементті өзгерту және берілген аралық бойынша қосындыны есептеу операцияларын қолдайтын деректер құрылымын құруымыз керек. Осыған дейін біз бұл есепті екі операцияны $O(\log n)$ уақытта қолдайтын бинарлы индекстелген дарақ немесе кесінділер дарағы арқылы шығарған едік. Ал енді $O(1)$ уақытта элементтерді өзгертуге және $O(\sqrt{n})$ уақытта қосындыларды есептеуге мүмкіндік беретін квадраттық түбір құрылымын пайдаланып, басқа жолмен шешеміз.

Идеясы жиымды өлшемі \sqrt{n} болатын блоктарға бөлу және әр блок үшін ішіндегі элементтер қосындысын сақтауға негізделеді. Мысалы, 16 элементтен тұратын жиым төмендегідей 4 элементтік блоктарға бөлінеді:

21				17				20				13			
5	8	6	3	2	7	2	6	7	1	7	5	6	2	3	2

Аталған құрылымда жиым элементтерін өзгерту жеңіл. Өйткені әр жаңартудан кейін 1 ғана блоктың қосындысы өзгереді, ал ол өз кезегінде $O(1)$ уақытта орындалады. Мысалы, төмендегі суретте элементтің мәні және оған сәйкес блок қалай өзгеретіні көрсетілген:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Аралықтағы қосындыны есептеу үшін, дара элементтер мен олардың арасындағы бүтін блоктардан тұратындай етіп, аралықты үшке бөлеміз:

21				15				20				13			
5	8	6	3	2	5	2	6	7	1	7	5	6	2	3	2

Дара элементтердің саны да $O(\sqrt{n})$, блоктардың саны да $O(\sqrt{n})$ болғандықтан, қосынды сұратымы $O(\sqrt{n})$ уақыт алады. Блоктың өлшемі – \sqrt{n} , өйткені ол екі затты теңгерімдейді: жиым \sqrt{n} блоктан, ал әр блок \sqrt{n} элементтен тұрады.

Тәжірибеде параметр ретінде дәл \sqrt{n} мәнін қолдану шарт емес, оны k мен n/k арқылы алмастыра аламыз, мұндағы k \sqrt{n} -ге тең емес. Параметр тиімділігі есеп пен енгізуге байланысты. Мысалы, егер алгоритм блоктарды жиі қарап, дара элементтерге сирек жүгінетін болса, жиымды $k < \sqrt{n}$ блоктарға, ал әр блокты $n/k > \sqrt{n}$ бөлген тиімді.

27.1 Алгоритмдерді біріктіру

Бұл бөлімде біз екі алгоритмді біріктіруге негізделген екі квадрат түбір алгоритмін талқылаймыз. Қай жағдайда да алгоритмдердің біреуін ғана қолдана отырып, есепті $O(n^2)$ уақытта шығаруға болады. Бірақ, алгоритмдерді біріктіру арқылы уақытша күрделілігі $O(n\sqrt{n})$ -ге қысқарады.

Жағдайларды қарастыру

Бізге n ұяшығы бар екі өлшемдік тор берілген. Әр ұяшықта әріп белгіленген. Бізге арақашықтығы ең аз болатын бірдей әріпті екі ұяшықты табу тапсырылады. (x_1, y_1) және (x_2, y_2) ұяшықтарының қашықтығы – $|x_1 - x_2| + |y_1 - y_2|$. Мысалы, төмендегі торды қарастырайық:

A	F	B	A
C	E	G	E
B	D	A	F
A	C	B	D

Бұл жағдайда минималды қашықтық 2-ге тең – екі 'E' әріптерінің арасы.

Есепті әр әріпті жеке қарастыру арқылы шығаруға болады. Бұл тәсілді қолдансақ, есебіміз тұрақты c әрпі бар екі ұяшық арасындағы ең аз қашықтықты есептейтін жаңа есепке ауысады. Екі алгоритмді қарастырайық:

1-алгоритм: барлық c әрпі бар ұяшық жұптарын өтіп, сол ұяшықтар арасындағы минимум қашықтықты есептеу. Бұл $O(k^2)$ уақыт алады, мұндағы k – c әрпі бар ұяшықтар саны.

2-алгоритм: барлық c әрпі бар ұяшықтардан бір уақытта жүргізілетін ені бойынша іздеу (BFS) алгоритмін бастаймыз. c әрпі бар ұяшықтардың арасындағы минимум қашықтықты $O(n)$ уақытта есептейтін болады.

Есепті шешудің бір жолы – алгоритмдердің біреуін тандап, барлық әріптерге қолдану. Егер 1-алгоритмді қолдансақ, оның орындалу уақыты $O(n^2)$: барлық ұяшықтарда бірдей әріп болуы мүмкін және сол жағдайда $k = n$. Егер 2-алгоритмді қолдансақ, оның да орындалу уақыты $O(n^2)$, себебі әр ұяшықта әртүрлі әріп болуы мүмкін және бұл жағдайда n ізденіс жасауы қажет.

Дегенмен біз екі алгоритмді біріктіре аламыз және әр әріптің санына қарай әртүрлі алгоритм қолдануға болады. c әрпі k мәрте қайталанды делік. Егер $k \leq \sqrt{n}$ болса, 1-алгоритмді қолданамыз, ал $k > \sqrt{n}$ болса, 2-алгоритмді қолданамыз. Осы жолды ұстансақ, қорытынды уақытша күрделілігі $O(n\sqrt{n})$ болады.

Алдымен c әрпі үшін 1-алгоритмді қолданамыз делік. c әрпі ең көбі \sqrt{n} кездескендіктен, біз әр ұяшықты басқа c әріптермен $O(\sqrt{n})$ рет салыстырамыз. Осылайша сондай барлық ұяшықтарды өңдеуге $O(n\sqrt{n})$ уақыт кетеді. Ал енді c әрпі үшін 2-алгоритмді қолданамыз делік. c әрпі ең көбі \sqrt{n} кездескендіктен, барлық сондай әріптерді өңдеу үшін де $O(n\sqrt{n})$ уақыт қажет болады.

Дестелік өңдеу

Келесі есебіміз n ұяшығы бар екі өлшемдік торды қарастырады. Бастапқыда бір ұяшықтан басқасының барлығы ақ болады. Біз $n-1$ операция орындаймыз. Әр операцияда берілген ақ ұяшықтан қара ұяшыққа минимум қашықтықты есептеп, ақ ұяшықты қараға бояймыз.

Мысалы, төмендегі операцияларды қарастырайық:

		*	

Әуелде $*$ арқылы белгіленген ақ ұяшықтан қара ұяшыққа минимум қашықтықты есептейміз. Минимум қашықтық – 2, яғни екі қадам солға жүріп, қара ұяшыққа жете аламыз. Кейін ақ ұяшықты қараға бояймыз:

Төмендегі екі алгоритмді қарастырайық:

1-алгоритм: Ені бойынша іздеу (BFS) арқылы әр ақ ұяшық үшін ең жақын орналасқан қара ұяшықты анықтаймыз. Бұл $O(n)$ уақыт алады және әр ізденістен кейін әр ақ ұяшықтан қара ұяшыққа дейінгі қашықтықты $O(1)$ уақытта айта аламыз.

2-алгоритм: Қараға боялған ұяшықтарды тізімде сақтаймыз, әр операция сайын тізімді өтіп шығып, соңында жаңа ұяшық қосамыз. Операция $O(k)$ уақыт алады, мұндағы k – тізімнің өлшемі.

Операцияларды әрқайсысы $O(\sqrt{n})$ операциядан тұратын $O(\sqrt{n})$ дестелерге бөлу арқылы жоғарыдағы екі алгоритмді біріктіреміз. Әр дестенің алдында біз 1-алгоритмді жүргіземіз. Кейін дестедегі операцияларды өңдеу

үшін 2-алгоритмді қолданамыз. Дестелер арасында 2-алгоритмнің тізімін тазартып отырамыз. Әр операцияда қара ұяшыққа дейінгі минималды қашықтық – 1-алгоритмнен есептелген қашықтық немесе 2-алгоритмнен есептелген қашықтық болады.

Қорытынды алгоритм $O(n\sqrt{n})$ уақытта жұмыс істейді. Біріншіден, 1-алгоритм $O(\sqrt{n})$ мәрте орындалады және әр ізденіс $O(n)$ уақыт алады. Екіншіден, дестенің ішінде 2-алгоритмді қолданғанда, тізім $O(\sqrt{n})$ элементті қамтиды (себебі біз дестелер арасында тізімді тазартамыз). Демек әр операция $O(\sqrt{n})$ уақыт алады.

27.2 Бүтін бөлінулер

Кейбір квадраттық түбір алгоритмдер мынадай бақылауға негізделеді: егер n бүтін саны бүтін қосылғыштармен көрсетілсе, қосылғыштардың саны ең көбі $O(\sqrt{n})$ әртүрлі саннан тұрады. Себебі қосылғыштардың саны көп болатын және әртүрлі қосылғыштардан тұратын қосындыны құрау үшін ең кішкентай сандарды алуымыз қажет.

Егер сандарды $1, 2, \dots, k$ деп алсақ, қорытынды қосынды

$$\frac{k(k+1)}{2}.$$

болады. Демек максималды әртүрлі қосылғыштардың саны – $k = O(\sqrt{n})$. Кейін бұл бақылауға сүйеніп, 2 есептің шығару жолын талқылаймыз.

Қоржын

Бізге қосындысы n болатын бүтін салмақтар тізімі берілген. Салмақтардың ішжиындарынан құралатын барлық қосындыларды табуымыз керек. Мысалы, егер салмақтар $\{1, 3, 3\}$ болса, төмендегідей қосындылардың болуы мүмкін:

- 0 (бос жиын)
- 1
- 3
- $1 + 3 = 4$
- $3 + 3 = 6$
- $1 + 3 + 3 = 7$

Қарапайым қоржын тәсілін (7.4-тарауын қараңыз) қолдансақ, есеп былай шығарылады: $\text{possible}(x, k)$ функциясының мәні алғашқы k салмақты пайдаланып x қосындысын жинаса, 1-ге тең, ал басқа жағдайда 0-ге тең болады. Салмақтардың қосындысы n болғандықтан, ең көбі n салмақ бола алады және функцияның барлық мәндері динамикалық бағдарламалау арқылы $O(n^2)$ уақытта есептеледі.

Бірақ, әртүрлі салмақтардың саны ең көбі $O(\sqrt{n})$ болатындықтан, бұл алгоритмді жылдамырақ жаза аламыз. Осылайша бірдей салмақтарды топтастырып қарастырамыз. Әр топты $O(n)$ уақытта өңдей аламыз, демек ақырғы уақытша күрделілігі – $O(n\sqrt{n})$.

Идеясы осы уақытқа дейін өңделген топтарды пайдаланып, құруға болатын салмақтарды сақтайтын жиым қолдануға негізделеді. Жиым n элементтен тұрады. Егер k қосындысын құрай алса, k -элемент 1-ге, ал кері жағдайда 0-ге тең. Салмақтардың тобын өңдеу үшін жиымды солдан оңға қарай өтіп, осы топ пен алдыңғы топ арқылы құрауға болатын жаңа салмақтарды сақтаймыз.

Жол құрылысы

Ұзындығы n болатын s жолы берілген және жалпы ұзындығы m болатын D жолдар жиыны берілген. D жолдарының конкатенациясы арқылы s жолын қанша жолмен құрауға болатынын қарастырайық.

Мысалы, егер $s = \text{ABAB}$ және $D = \{A, B, AB\}$ болса, онда 4 жол бар:

- $A + B + A + B$
- $AB + A + B$
- $A + B + AB$
- $AB + AB$

Бұл есепті динамикалық бағдарламалау арқылы шығаруға болады: $\text{count}(k)$ деп D -дағы жолдар арқылы $s[0 \dots k]$ префиксін құрайтын жолдар санын белгілейік. Енді $\text{count}(n - 1)$ есеп жауабы бола алады және бұл есепті $O(n^2)$ уақытта префикс дарағы арқылы шығара аламыз.

Алайда жол хеші мен D -да ең көбі $O(\sqrt{m})$ әртүрлі жолдар ұзындығы болатыны туралы бақылаудың арқасында жылдамырақ шешуге болады. Алдымен D -дағы барлық жолдардың хештері бар H жиынын құрастырамыз. Кейін $\text{count}(k)$ мәнін есептеу үшін D -да болуы мүмкін жол ұзындығы p мәндерін қарастырамыз және $s[k - p + 1 \dots k]$ хеш мәнін есептеп, оның H -та бар-жоғын тексереміз. Ұзындықтары әртүрлі жолдардың ең көп саны $O(\sqrt{m})$ болғандықтан, қорытынды уақытша күрделілігі $O(n\sqrt{m})$ болады.

27.3 Мо алгоритмі

Мо алгоритмі¹ статикалық (яғни сұратымдардың арасында жиым өзгермейді) жиымда аралық сұратымдарды өңдеуді талап ететін көптеген есептерде қолданылады. Әр сұратымда бізге $[a, b]$ аралығы берілген және a мен b аралығындағы жиым элементтеріне негізделген мәнді есептеу қажет. Статикалық жиым болғандықтан сұратымдарды кез келген ретте өңдеуге болады. Мо алгоритмі сұратымдарды алгоритмнің жылдам болуына кепілдік беретіндей арнайы ретпен өңдейді.

¹[67]–ге сәйкес бұл алгоритм қытайлық спорттық бағдарламалаушы Мо Таоның атымен аталған. Алайда бұл техника әдебиетте ертерек пайда болды [68].

Мо алгоритмі жиымның белсенді аралығын сақтайды және белсенді аралыққа қатысты сұратымға әрқашан жауап береді. Алгоритм сұратымдарды бір-бірден өңдейді және элементтерді қосып, не өшіріп отырып, белсенді аралықтың шеткі нүктелерін жылжытады. Уақытша күрделілігі – $O(n\sqrt{n}f(n))$, мұндағы жиым n элементтен тұрады, n сұратым бар және элементті қосу немесе өшіру $O(f(n))$ уақыт алады.

Мо алгоритмінің айласы – сұратымдарды өңдеу реті. Жиым $k = O(\sqrt{n})$ элементтен тұратын блоктарға бөлінеді. Егер

- $\lfloor a_1/k \rfloor < \lfloor a_2/k \rfloor$ немесе
- $\lfloor a_1/k \rfloor = \lfloor a_2/k \rfloor$ және $b_1 < b_2$

шарты орындалса, $[a_1, b_1]$ сұратымы $[a_2, b_2]$ сұратымнан ертерек өңделеді.

Осылайша, сол жақ шеткі нүктесі бір блокта тұрған барлық сұратымдар оң жақ шеткі нүктелеріне қарай сұрыпталған ретте бірінен кейін бірі өңделеді. Осы рет бойынша алгоритм тек $O(n\sqrt{n})$ операция орындайды. Сол жақ шеткі нүкте $O(n)$ рет $O(\sqrt{n})$ қадамға өзгереді, ал оң жақ шеткі нүкте $O(\sqrt{n})$ рет $O(n)$ қадамға өзгереді. Демек алгоритм барысында екі шеткі нүктелер жалпылай $O(n\sqrt{n})$ қадам жасайды.

Өрнек

Өрнек ретінде жиым бойынша бірнеше сұратым берілді делік. Әр сұратымда қанша әртүрлі сан бар екенін табуымыз керек.

Мо алгоритмінде сұратымдар әрдайым бірдей сұрыпталған, бірақ есепке қарай сұратымға жауап қалай сақталанатыны белгілі болады. Бұл есепте count жиымын белгілейік. $\text{count}[x]$ деп x саны белсенді аралықта қанша рет кездесетінін көрсетеді.

Бір сұратымнан екінші сұратымға көшкенде, белсенді аралық өзгереді. Мысалы, белсенді аралық төмендегідей,

4	2	5	4	2	4	3	3	4
---	---	---	---	---	---	---	---	---

ал келесі аралық көрсетілгендей болса,

4	2	5	4	2	4	3	3	4
---	---	---	---	---	---	---	---	---

келесі үш қадам жасалу керек: сол жақ шеткі нүкте бір қадам оңға жылжиды, ал оң жақ шеткі нүкте екі қадам оңға жылжиды.

Әрбір қадамнан кейін count жиымы жаңаруы тиіс. x санын қосқаннан кейін, біз $\text{count}[x]$ мәнін 1-ге арттырып, кейін $\text{count}[x] = 1$ болса, осы сұратымға байланысты жауапты 1-ге арттыруымыз қажет. Дәл солай, x санын өшіргеннен кейін, біз $\text{count}[x]$ мәнін 1-ге азайтып, ал одан кейін $\text{count}[x] = 0$ болса, осы сұратымға байланысты жауапты 1-ге азайтуымыз қажет.

Бұл есепте әрбір қадамды орындау үшін $O(1)$ уақыт керек. Демек алгоритмнің қорытынды уақытша күрделілігі – $O(n\sqrt{n})$.

28-тарау. Тағы да кесінділер дарағы туралы

Кесінділер дарағы – түрлі-түрлі есептерді шығару үшін қолданылатын икемді деректер құрылымы. Дегенмен кесінділер дарағына байланысты кейбір тақырыптарды әлі өтпедік. Енді сол күрделі тақырыптарды өтетін уақыт келді.

Осыған дейін біз кесінділер дарағының операцияларын дарақтың түбінен басына қарай көтерілу арқылы жүзеге асырдық. Мысалы, аралық қосындысын төмендегідей есептедік (9.3-тарау):

```
int sum(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
    return s;
}
```

Бірақ күрделі кесінділер дарақтарында әдетте операцияларды керісінше, яғни жоғарыдан төменге қарай орындау қажет. Осы тәсілді қолдансақ, функция төмендегідей болады:

```
int sum(int a, int b, int k, int x, int y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return tree[k];
    int d = (x+y)/2;
    return sum(a,b,2*k,x,d) + sum(a,b,2*k+1,d+1,y);
}
```

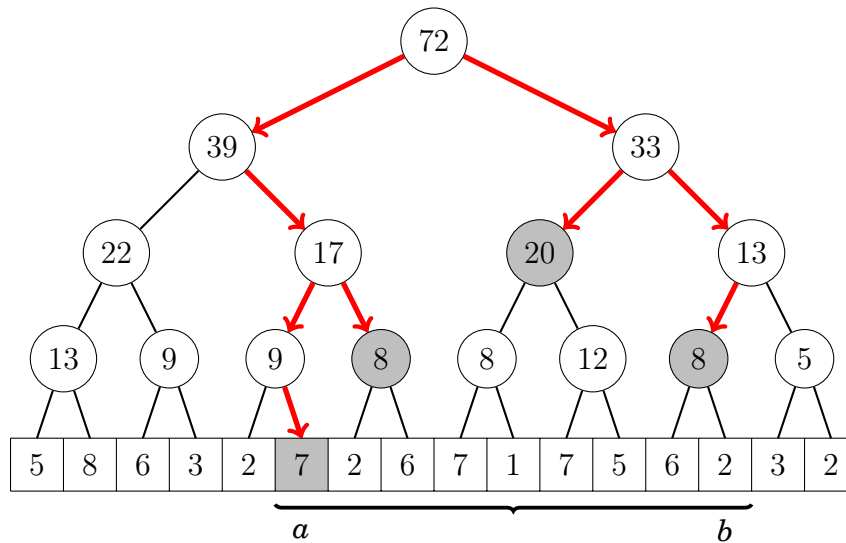
Енді кез келген $\text{sum}_q(a, b)$ мәнін $[a, b]$ аралығындағы жиым мәндерінің қосындысы) келесідей есептейтін боламыз:

```
int s = sum(a, b, 1, 0, n-1);
```

k параметрі дарақтағы қазіргі позицияны көрсетеді. Басында k 1-ге тең, себебі біз дарақтың түбірінен бастаймыз. $[x, y]$ аралығы k төбесіне сәйкес

және басында $[0, n - 1]$ аралығына тең. Қосындыны есептеген кезде, егер $[x, y]$ аралығы $[a, b]$ аралығының сыртында болса, қосынды 0-ге тең. Ал егер $[x, y]$ аралығы толықтай $[a, b]$ аралығына кірсе, қосындыны дарақтан алсақ болады. Егер $[x, y]$ аралығының кейбір бөлігі $[a, b]$ аралығына кірсе, ізденіс рекурсивті түрде $[x, y]$ аралығының оң мен сол жартысына жалғасады. Сол жартысы – $[x, d]$ және оң жартысы – $[d + 1, y]$, мұндағы $d = \lfloor \frac{x+y}{2} \rfloor$.

Келесі сурет $\text{sum}_q(a, b)$ мәнін есептеу кезінде ізденістің қалай жүретінін көрсетеді. Сұр төбелер рекурсия тоқтағанын көрсетеді және қосындыны дарақтың сол төбесінен алуымызға болады.



Бұл код та операцияларды өңдеу үшін $O(\log n)$ уақыт алады, өйткені барлық өтіп шығатын төбелердің саны – $O(\log n)$.

28.1 Жалқау таратылу

Жалқау таратылу арқылы аралық жаңартулар мен аралық сұратымдарды $O(\log n)$ уақытта қолдайтын кесінділер дарағын құрастыруға болады. Идеясы жаңарту және сұратым операцияларын жоғарыдан төменге қарай өңдеуге және жаңартуларды керек кезінде ғана жалқау түрде төменге таратуға негізделеді.

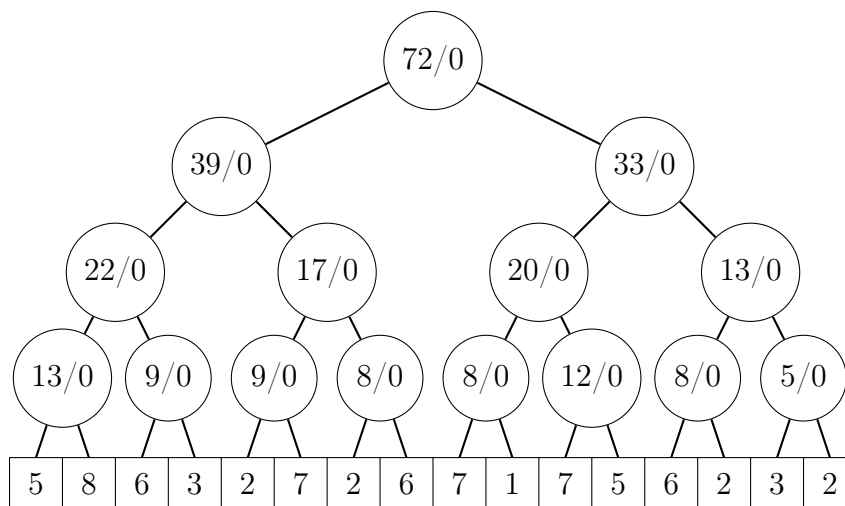
Жалқау кесінділер дарағындағы төбе екі түрлі ақпаратты сақтайды. Біріншіден, қарапайым кесінділер дарағы сияқты әр төбе сәйкес ішжиымның қосындысын немесе ішжиымға қатысты басқа мәнді сақтайды. Екіншіден, төбе ұлдарына таратылмаған жалқау жаңартуларға қатысты ақпаратты сақтауы мүмкін.

Аралық сұратымдардың екі түрі бар. Аралықтағы барлық жиым мәндері бір мәнге арттырылады немесе жиым мәндеріне бір мән меншіктеледі. Екі операцияны да ұқсас идея арқылы жүзеге асыруға болады, тіпті екі операцияны бір уақытта қолдайтын дарақты да құруға болады.

Жалқау кесінділер дарағы

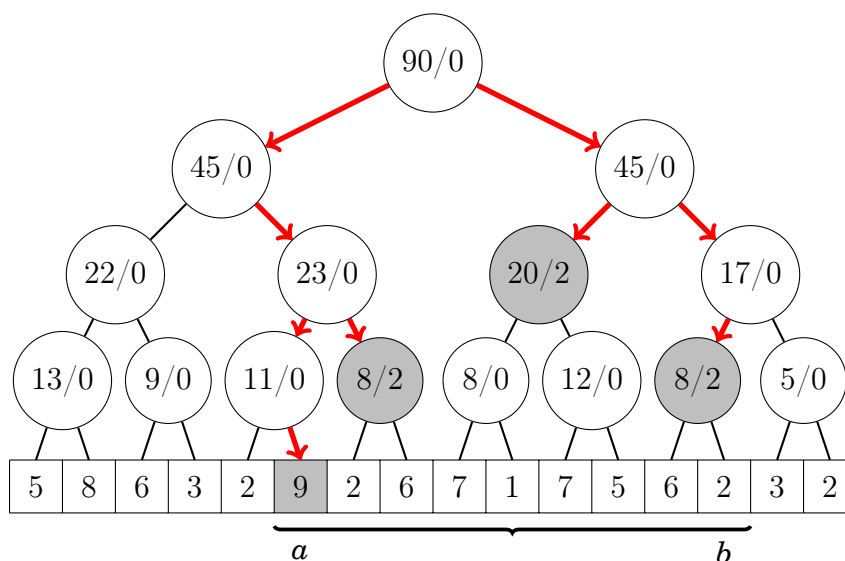
$[a, b]$ аралығын тұрақты санға арттыратын және $[a, b]$ аралығындағы мәндерінің қосындысын табатын кесінділер дарағын мысал ретінде қарастырайық.

Біз әр төбеде екі мән s/z болатындай дарақты құрастырамыз. Мұндағы s аралықтағы мәндердің қосындысын белгілесе, z жалқау жаңарту мәнін белгілейді, яғни сол аралықтағы мәндердің барлығы z -ке арттырылуы тиіс. Келесі дарақта барлық төбелерде $z = 0$, яғни әзірге жалқау жаңартулар жоқ.



$[a, b]$ аралығындағы элементтер u -ге арттырылған кезде біз түбірден жапырақтарға қарай жүріп, дарақтағы төбелерді келесі ретпен өзгерте аламыз. Егер төбенің $[x, y]$ аралығы толықтай $[a, b]$ аралығына кірсе, төбенің z мәнін u -ге арттырамыз және тоқтаймыз. Ал егер $[x, y]$ аралығының кейбір бөлігі $[a, b]$ аралығына кірсе, онда төбенің s мәнін hu мәніне арттырамыз, мұндағы h – $[a, b]$ мен $[x, y]$ аралықтарының қиылысу өлшемі, және дарақ бойынша өтуімізді рекурсивті түрде жалғастырамыз.

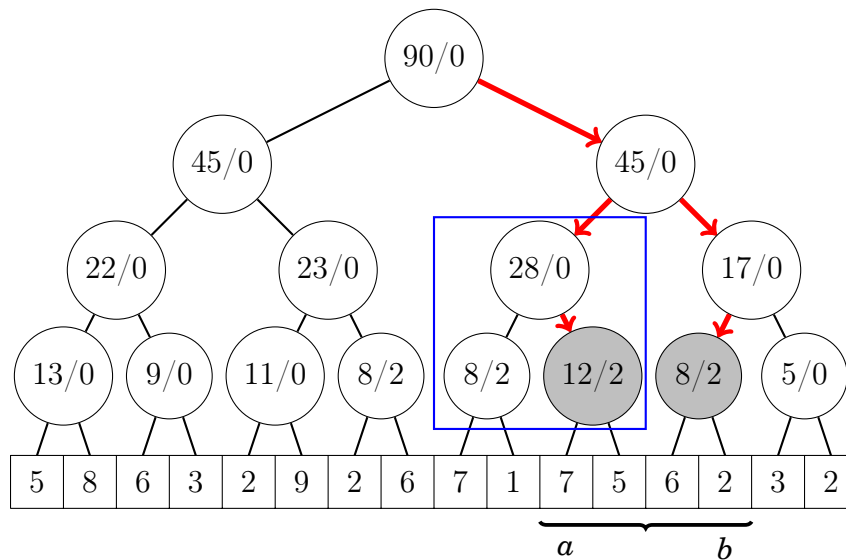
Мысалы, келесі суретте $[a, b]$ аралықтағы элементтерді 2-ге арттырғаннан кейінгі дарақ бейнеленген:



Сондай-ақ біз $[a, b]$ аралығындағы элементтердің қосындысын дарақты жоғарыдан төменге қарай өту арқылы есептейміз. Егер төбенің $[x, y]$ аралығы толықтай $[a, b]$ аралығына кірсе, біз қосындыға төбенің s мәнін қосамыз. Әйтпесе ізденісті рекурсивті түрде дарақтың төменіне қарай жалғастырамыз.

Жаңартуда да, сұратымда да әрдайым төбені өңдеуден бұрын жалқау жаңартудың мәні төбенің ұлдарына таратылады. Демек жаңартулар керек болған жағдайда ғана төменге қарай таратылады, ал бұл операциялардың оңтайлы болуына кепілдік береді.

Келесі сурет $\text{sum}_a(a, b)$ мәнін есептеген кезде дарақ қалай өзгеретінін көрсетеді. Төртбұрыш жалқау жаңартулардың төменге таратылуына байланысты өзгерген төбелерді көрсетеді.



Кейде жалқау жаңартуларды біріктіру қажет болады. Ондай жағдай жалқау жаңартуы бар төбеге жаңа жалқау жаңарту келген кезде туындайды. Қосындыны есептеген кезде жалқау жаңартуларды қосу оңай, яғни z_1 мен z_2 жаңартулардың біріктіруі $z_1 + z_2$ жаңартуына сәйкес келеді.

Көпмүшелі жаңартулар

Жалқау жаңартуларды

$$p(u) = t_k u^k + t_{k-1} u^{k-1} + \dots + t_0$$

формасындағы көпмүшелі аралық жаңартуларды қолдайтындай етіп жалпылай аламыз.

Бұл жағдайда $[a, b]$ аралығындағы i -элементтің жаңартуы $p(i - a)$ тең болады. Мысалы $p(u) = u + 1$ көпмүшесін $[a, b]$ аралығына қосқан кезде a позициядағы элементті 1-ге арттырады, $a + 1$ позициядағы элементті 2-ге арттырады және солай кете береді.

Көпмүшелі жаңартуларды қолдау үшін әр төбеде $k + 2$ мән сақтайтын боламыз, мұндағы k көпмүшенің дәрежесіне тең. Олар – аралықтағы элементтердің қосындысының мәні s және жалқау жаңартуға сәйкес z_0, z_1, \dots, z_k көпмүшенің коэффициенттері.

Онда $[x, y]$ аралығындағы элементтерінің қосындысы

$$s + \sum_{u=0}^{y-x} z_k u^k + z_{k-1} u^{k-1} + \dots + z_0$$

болады.

Жоғарыдағы қосындыны қосындылар формуласы арқылы оңтайлы есептеуге болады. Мысалы z_0 мүшесі $(y - x + 1)z_0$ қосындысына сәйкес, ал $z_1 u$ мүшесі

$$z_1(0 + 1 + \dots + y - x) = z_1 \frac{(y - x)(y - x + 1)}{2}$$

қосындысына сәйкес.

Дарақта жаңарту таратылған кезде $p(u)$ индекстері өзгереді, себебі әр $[x, y]$ аралығына мәндер $u = 0, 1, \dots, y - x$ кезге есептелінеді. Дегенмен бұл қиыншылық емес, себебі $p'(u) = p(u + h)$ — $p(u)$ -ға тең дәрежелі көпмүше. Мысалы, егер $p(u) = t_2 u^2 + t_1 u - t_0$, онда

$$p'(u) = t_2(u + h)^2 + t_1(u + h) - t_0 = t_2 u^2 + (2ht_2 + t_1)u + t_2 h^2 + t_1 h - t_0.$$

28.2 Динамикалық дарақ

Қарапайым кесінділер дарағы статикалық болып келеді, яғни әр төбе жиымда тұрақты орын алады және дарақ жадыда тұрақты өлшем алады. Динамикалық кесінділер дарағында алгоритм барысында жүгінетін төбелерге ғана жады бөлінеді. Бұл жадыны айтарлықтай үнемдей алады.

Динамикалық дарақтағы төбелерді келесі құрылым (struct) ретінде көрсетуге болады:

```
struct node {
    int value;
    int x, y;
    node *left, *right;
    node(int v, int x, int y) : value(v), x(x), y(y) {}
};
```

Бұл жерде value — төбенің мәні, $[x, y]$ — сәйкес аралық, ал left пен right сол мен оң ішдарақтарға нұсқайды.

Бұдан кейін төбелер келесі түрде құрылады:

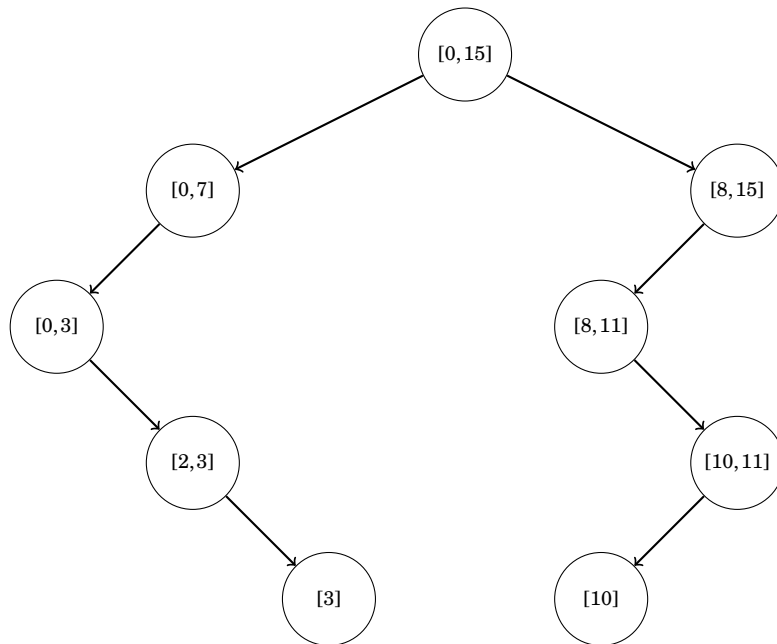
```
// create new node
node *x = new node(0, 0, 15);
// change value
x->value = 5;
```

Сиретілген кесінділер дарағы

Динамикалық кесінділер дарағы оның артындағы жиым сиретілген, яғни индекстерінің ауқымы $[0, n - 1]$ өте үлкен, бірақ элементтерінің көпшілігі

нөлдер болған жағдайда ерекше пайдалы болмақ. Қарапайым кесінділер дарағын сақтау үшін $O(n)$ көлемі бар жады қажет болса, динамикалық кесінділер дарағына тек $O(k \log n)$ жады қолданылады, мұндағы k – орындалған операциялар саны.

Сиретілген кесінділер дарағында бастапқыда мәні 0-ге тең бір ғана $[0, n-1]$ төбе болады. Ол жиымдағы барлық элементтердің мәні нөлге тең дегенді білдіреді. Жаңартулардан кейін жаңа төбелер дараққа динамикалық түрде қосылады. Мысалы, егер $n = 16$ және 3 пен 10-позициядағы элементтер өзгертілсе, дарақ келесі төбелерді қамтиды:



Түбір төбеден жапырақ төбеге дейінгі жол $O(\log n)$ төбеден тұрады. Сондықтан да әр операция дараққа көп дегенде $O(\log n)$ жаңа төбелер қосады. Осылайша k операциядан кейін, дарақ ең көбі $O(k \log n)$ төбелерден тұрады.

Мына жайтты ескергеніміз жөн: жаңартылатын барлық элементтерді алгоритмнің басында білетін болсақ, бізге динамикалық кесінділер дарағы керек болмас еді. Өйткені біз индекстерді сығымдау арқылы қарапайым кесінділер дарағын (9.4-тарау) қолданар едік. Алайда, егер индекстер алгоритмнің жұмысы барысында туындаса, бұл мүмкін болмайды.

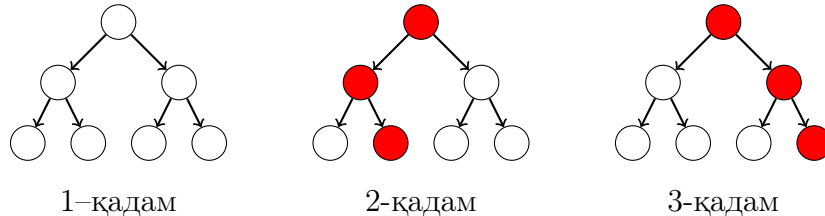
Ұзақ сақталушы кесінділер дарағы

Динамикалық іске асыру арқылы жаңартулар тарихын сақтайтын ұзақ сақталушы кесінділер дарағын да құрыстыруымызға болады. Ол арқылы біз алгоритмнің барысында дарақта болған барлық нұсқаларға оңтайлы жүгіне аламыз.

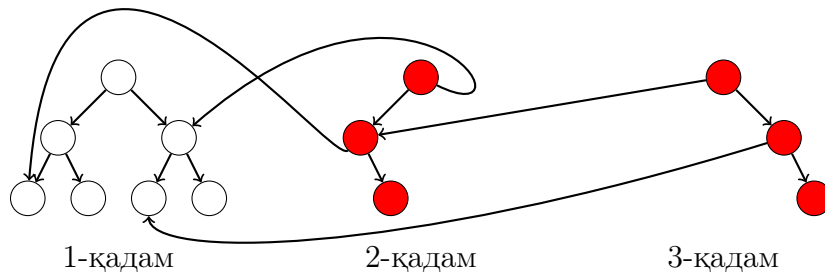
Модификациялау тарихы қолжетімді болса, дарақтың кез келген алдыңғы версияларына қарапайым кесінділер дарағы сияқты сұратымдар жасауымызға болады. Өйткені әр дарақтың толық құрылымы сақталып тұрады.

Сондай-ақ біз бұрынғы дарақтардың негізінде жаңа дарақтар құрып, оларды дербес өзгерте аламыз.

Қызыл төбелері өзгеретін, ал басқа төбелері өзгеріссіз қалатын келесі жаңартулар тізбегін қарастырайық:



Әр жаңартудан кейін төбелердің көбі өзгеріссіз қалады, сондықтан модификация тарихын сақтаудың ықшам тәсілі дарақтың әр тарихи версиясын жаңа төбелер мен алдыңғы дарақ ішдарақтарының комбинациялары түрінде ұсынудан тұрады. Бұл мысалда жаңартулар тарихы келесі ретпен сақталады:



Әр алдыңғы дарақтың құрылымын сәйкес түбірден бастап, нұсқағыштарға ілесе отырып, қалпына келтіруімізге болады. Әр операция дараққа жаңа $O(\log n)$ төбе қосатындықтан, дарақтың барлық жаңарту тарихын сақтай аламыз.

28.3 Деректер құрылымы

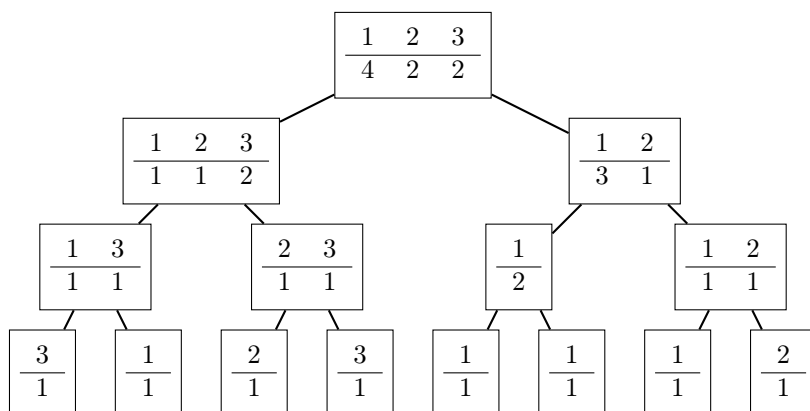
Кесінділер дарағындағы төбелер жекелеген мәндердің орнына сәйкес аралық бойынша ақпараттан тұратын деректер құрылымын да сақтай алады. Ондай дарақта операциялар $O(f(n)\log n)$ уақыт алады, мұндағы $f(n)$ – операция барысында бір төбені өңдеуге кететін уақыты.

Үлгі ретінде ” $[a, b]$ аралықта x элементі қанша рет кездеседі?” - деген сұратымдарды қолдайтын кесінділер дарағын қарастырайық. Мысалы, 1 элементі келесі аралықта 3 рет кездеседі:

3	1	2	3	1	1	1	2
---	---	---	---	---	---	---	---

Осындай сұратымдарды қолдау үшін әр төбеде кез келген x саны сәйкес аралықта қанша рет кездесетінін айтып беретін деректер құрылымы бар кесінділер дарағын құрастырамыз. Осы дарақты қолдана отырып, аралыққа кіретін төбелердің жауаптарын біріктіру арқылы сұратымның жауабын алуымызға болады.

Мысалы, келесі кесінділер дарағы үстідегі жиымға сәйкес келеді:



Дарақты әр төбеде тар құрылымы болатындай етіп құрастыра аламыз. Ондай жағдайда әр төбені өңдеуге $O(\log n)$ уақыт жұмсалады, демек сұратымның жалпы уақытша күрделілігі $O(\log^2 n)$ болады. Дарақ $O(n \log n)$ жадыны қолданады, өйткені дарақта $O(\log n)$ деңгей, ал әр деңгейде $O(n)$ элемент бар.

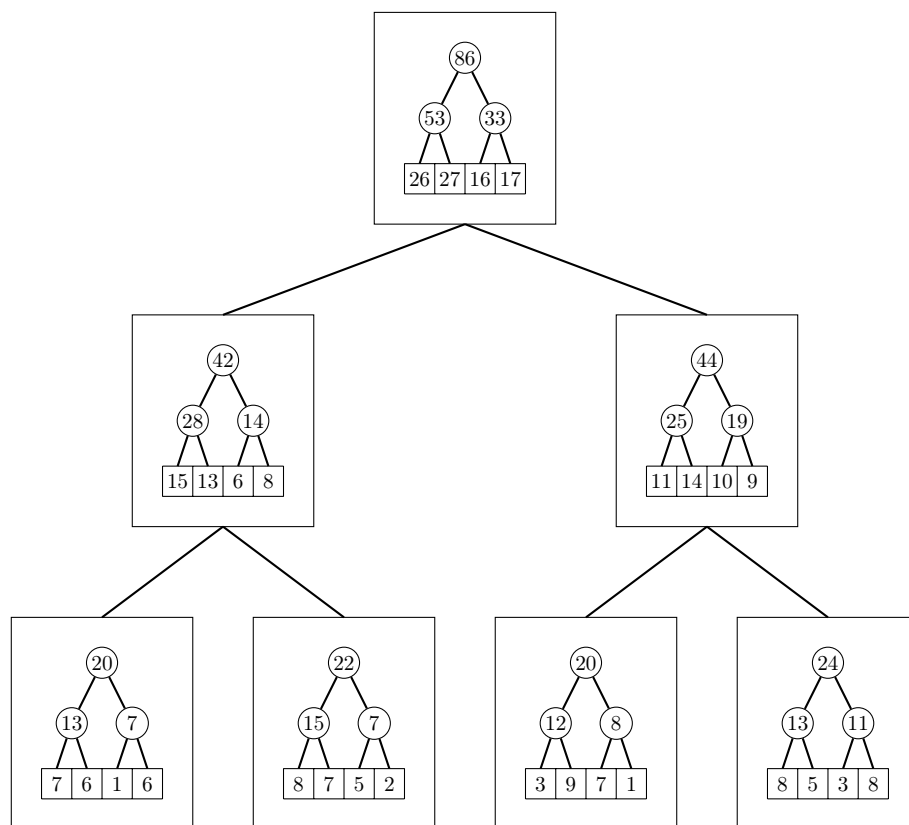
28.4 Екі өлшемділік

Екі өлшемді кесінділер дарағы екі өлшемді жиымның тіктөртбұрышты іш-жиымдарына қатысты сұратымдарды қолдайды. Ондай дарақты кірістірілген кесінділер дарағы арқылы жүзеге асыра аламыз. Үлкен дарақ жиымның жолдарына сәйкес келеді және әр төбе бағанға сәйкес кішкентай дарақты қамтиды.

Мысалы:

7	6	1	6
8	7	5	2
3	9	7	1
8	5	3	8

жиымның кез келген ішжиымын төмендегі кесінділер дарағы арқылы есептеуімізге болады:

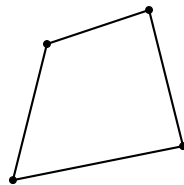


Екі өлшемді кесінділер дарағының операциялары $O(\log^2 n)$ уақыт алады. Себебі үлкен дарақ және әр кішкентай дарақ $O(\log n)$ деңгейлерден тұрады. Әр кіші дарақ $O(n)$ мәндерден тұратындықтан, дараққа $O(n^2)$ жады керек болады.

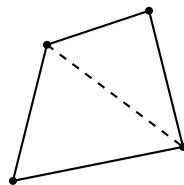
29-тарау. Геометрия

Геометрияға байланысты есептерде есепті ыңғайлы шығаратын және дербес жағдайлар аз болатын кодты жазу қиындық туғызады.

Үлгі ретінде төртбұрыштың (4 нүктесі бар көпбұрыш) нүктелері берілген есепті қарастырайық. Бізге оның ауданын табу қажет. Мысалы, төмендегідей төртбұрышты алайық:



Бұл есепті шығару жолдарының бірі – төртбұрышты қарама-қарсы нүктені қосатын екі сызық арқылы екі үшбұрышқа бөлу.

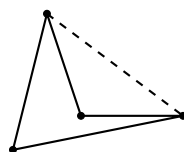


Содан кейін бізге үшбұрыштардың ауданын табу жеткілікті. Үшбұрыштардың ауданын Герон формуласы арқылы табуымызға болады:

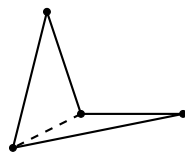
$$\sqrt{s(s-a)(s-b)(s-c)},$$

мұнда a , b , c – үшбұрыштың қабырғалары және $s = (a + b + c)/2$.

Бұл – есепті шығарудағы мүмкін болатын жолдардың бірі. Алайда бізді: ”төртбұрышты үшбұрыштарға қалай бөлеміз?” - деген мәселе ойландыруы керек. Өйткені біз кей жағдайларда қарама-қарсы жатқан екі еркін нүктелерді жай ғана қоса салмайды екенбіз. Мысалы төмендегі жағдайда бөлетін сызық төртбұрыштың сыртында орналасып тұр:



Дегенмен, сызықпен бөлудің басқа да жолдары бар:



Адам қай сызықтың дұрыс екенін оңай таба алады, ал компьютер үшін оны табу қиындау.

Дегенмен бұл есепті бағдарламалаушыға ыңғайлы болатындай етіп шешуімізге болады екен. Атап айтқанда, келесі жалпы формула

$$x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + x_3y_4 - x_4y_3 + x_4y_1 - x_1y_4,$$

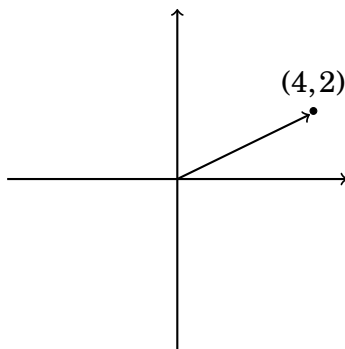
нүктелері (x_1, y_1) , (x_2, y_2) , (x_3, y_3) және (x_4, y_4) болатын төртбұрыштың ауданын есептеп береді.

Формуланы код түрінде жазу оңай, өйткені бұл жерде ешқандай дербес жағдай жоқ, содай-ақ біз бұл формуланы барлық көпбұрыштарға жалпылай аламыз.

29.1 Комплекс сандар

Комплекс сан – $x + yi$ формасындағы сан, мұндағы $i = \sqrt{-1}$ – жорамал сан. Комплекс санның геометриялық интерпретациясы – (x, y) нүктесі немесе координат басынан (x, y) нүктесіне дейінгі вектор.

Мысалы $4 + 2i$ келесі нүкте мен векторға сәйкес келеді:



Геометриялық есептерді шешуде C++-тің complex класы өте тиімді. Класты қолданып нүктелерді және векторларды комплекс сандар ретінде көрсете аламыз, ал класс геометрияда қолданылатын пайдалы тәсілдерден тұрады.

Келесі кодтағы C – координатаның типі және P – нүкте немесе вектордың типі. Оған қоса код x пен y координаталарға сілтейтін X және Y макростарын анықтайды.

```
typedef long long C;
typedef complex<C> P;
#define X real()
#define Y imag()
```

Мысалы келесі код $p = (4, 2)$ нүктесін анықтайды және оның x пен y координатасын шығарады:

```
P p = {4,2};  
cout << p.X << " " << p.Y << "\n"; // 4 2
```

Келесі код $v = (3, 1)$ және $u = (2, 2)$ векторларын анықтайды және олардың $s = v + u$ қосындысын есептейді.

```
P v = {3,1};  
P u = {2,2};  
P s = v+u;  
cout << s.X << " " << s.Y << "\n"; // 5 3
```

Іс жүзінде координаттардың қолайлы типіне әдетте `long long` (бүтін сандар) немесе `long double` (нақты сандар) жатады. Бүтін сандардың есептеулері дәл шығатындықтан, мүмкіндігінше бүтін сандарды қолдану ұсынылады. Егер нақты сандар керек болса, сандарды салыстырғанда дәлдік қателерді ескеру қажет. a мен b нақты сандардың тең екенін тексерудің сенімді жолы – оларды $|a - b| < \epsilon$ арқылы салыстыру, мұндағы ϵ өте кішкентай сан (мысалы, $\epsilon = 10^{-9}$).

Функциялар

Келесі мысалдардағы координаталардың типі – `long double`.

`abs(v)` функциясы $v = (x, y)$ вектордың ұзындығын $|v| \sqrt{x^2 + y^2}$ формуласы арқылы есептейді. Функция (x_1, y_1) және (x_2, y_2) нүктелер арасындағы ұзындықты есептеуге де қолданылады, өйткені олардың арасындағы ұзындық $(x_2 - x_1, y_2 - y_1)$ векторының ұзындығына тең.

Келесі код $(4, 2)$ және $(3, -1)$ нүктелер арасындағы ұзындықты есептейді:

```
P a = {4,2};  
P b = {3,-1};  
cout << abs(b-a) << "\n"; // 3.16228
```

`arg(v)` функциясы $v = (x, y)$ вектордың x осіне қатысты бұрышты есептейді. Функция бұрышты радианда береді (r радиан $180r/\pi$ градусқа тең). Оңға бағытталған вектордың бұрышы 0, бұрыш сағат тілі бағытымен кемиді және сағат тіліне қарсы бағытта арттырады.

`polar(s, a)` функциясы ұзындығы s болатын және a бұрышына бағыттайтын векторды құрастырады. Векторды a бұрышына бұру үшін ұзындығы 1 және бұрышы a болатын векторға көбейту қажет.

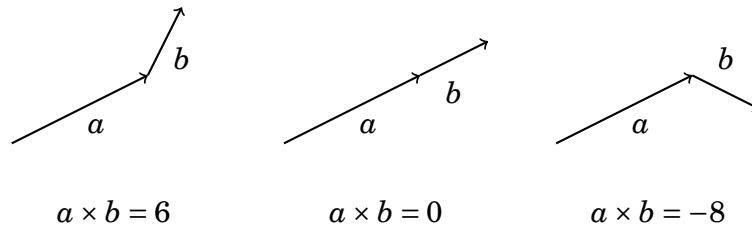
Келесі код $(4, 2)$ векторының бұрышын есептейді, сосын оны $1/2$ радиан сағат тіліне қарсы бұрады, содан соң бұрышты қайтадан есептейді:

```
P v = {4,2};  
cout << arg(v) << "\n"; // 0.463648  
v *= polar(1.0, 0.5);  
cout << arg(v) << "\n"; // 0.963648
```

29.2 Нүктелер мен сызықтар

$a = (x_1, y_1)$ және $b = (x_2, y_2)$ векторларының векторлық көбейтіндісі $a \times b$ $x_1 y_2 - x_2 y_1$ формуласы арқылы есептеледі. Векторлық көбейтінді b векторының a векторынан кейін қойылғанда солға қарай бұрылатынын (оң мән), бұрылмайтынын (нөл) немесе оңға қарай бұрылатынын (теріс мән) көрсетеді.

Келесі сурет жоғарыда аталып өткен жағдайларды көрсетеді:



Мысалы, бірінші жағдайда $a = (4, 2)$ және $b = (1, 2)$. Келесі код комплекс класы арқылы векторлық көбейтіндіні есептейді:

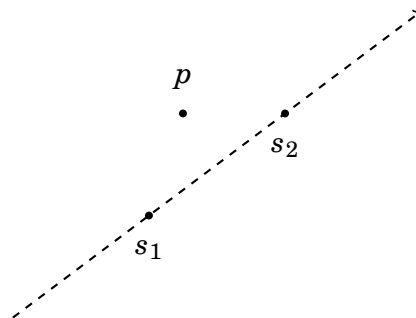
```
P a = {4,2};
P b = {1,2};
C p = (conj(a)*b).Y; // 6
```

Келесі код векторлық көбейтіндіні дұрыс есептейді. Өйткені `conj` функциясы вектордың y координатасының таңбасын өзгертеді және $(x_1, -y_1)$ мен (x_2, y_2) векторлар көбейтіндісінің y координатасы $x_1 y_2 - x_2 y_1$ тең болады.

Нүктенің орналасуы

Векторлық көбейтінді арқылы нүкте түзу сызықтың оң не сол жағында тұрғанын тексеруімізге болады. Түзу сызық s_1 және s_2 нүктелерінен өтеді делік. Сызық s_1 -ден s_2 -ге бағытталған, ал p – берілген нүкте.

Мысалы, келесі суретте p нүктесі сызықтың оң жағында тұр:

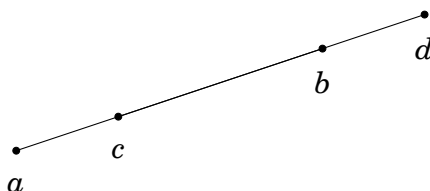


$(p - s_1) \times (p - s_2)$ векторлық көбейтіндісі p нүктенің орналасуын білдіреді. Егер векторлық көбейтінді оң сан болса, p нүктесі сызықтың сол жағында орналасады, егер векторлық көбейтінді теріс сан болса, p нүктесі сызықтың оң жағында орналасады. Ал егер векторлық көбейтінді нөл болса, s_1 , s_2 және p нүктелері бір сызықта орналасады.

Кесінділердің қиылысуы

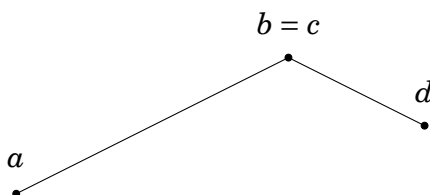
Келесі тапсырмада ab және cd кесінділердің қиылысуын тексеру беріледі. Төмендегідей мүмкін болатын жағдайлар бар:

1-жағдай: Кесінділер бір сызықта орналасқан және кей бөлігі бірінің үстіне бірі орналасқан. Бұл жағдайдағы қиылысу нүктелерінің саны – шексіз. Мысалы, келесі суреттегі c мен b нүктелерінің арасындағы барлық нүктелер қиылысу нүктелері болады.



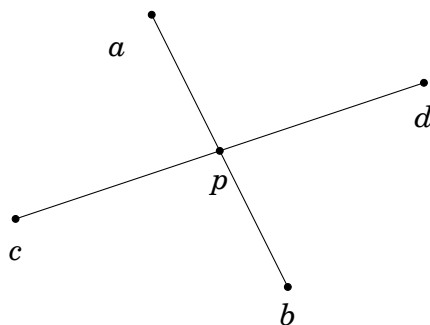
Бұл жағдайда векторлық көбейтіндіні барлық нүктелердің бір сызықта орналасқанын тексеру үшін қолдануымызға болады. Осыдан кейін нүктелерді сұрыптап, кесінділердің бірінің үстіне бірі орналасқанын тексере аламыз.

2-жағдай: Тек бір ғана қиылысу нүктесі бар және ол – кесінділердің шеткі нүктесі. Мысалы, келесі суреттегі қиылысу нүктесі – $b = c$:



Бұл жағдайды жеңіл түрде тексере аламыз. Ондай қиылысудың төрт түрлі мүмкіндігі бар: $a = c$, $a = d$, $b = c$ және $b = d$

3-жағдай: Тек бір ғана қиылысу нүктесі бар және ол кесінділердің шеткі нүктелері емес. Келесі суреттегі p нүктесі – қиылысу нүктесі:



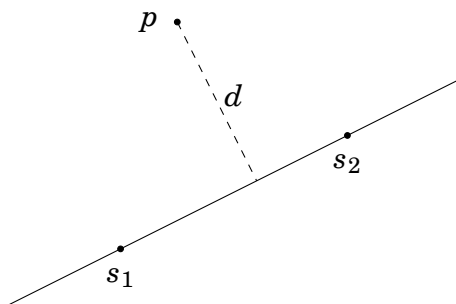
Осы жағдайда егер c мен d нүктелері a мен b -дан жүргізілген сызықтың әртүрлі жағында орналасса, ал a мен b нүктелері c мен d -дан жүргізілген сызықтың әртүрлі жағында орналасса, кесінділер қиылысады. Оны тексеру үшін векторлық көбейтіндіні қолдана аламыз.

Нүктеден сызыққа дейінгі қашықтық

Векторлық көбейтіндінің тағы да бір қасиеті бар. Ол – үшбұрыштың ауданын

$$\frac{|(a - c) \times (b - c)|}{2}$$

формуласы арқылы есептей алуымыз. Мұндағы a , b және c – үшбұрыштың нүктелері. Осы қасиетті қолдана отырып, нүктеден сызыққа дейінгі ең қысқа қашықтықты есептей аламыз. Мысалы, төмендегі суретте d – p нүктесі мен s_1 және s_2 нүктелері арқылы анықталған сызық арасындағы ең қысқа қашықтық.

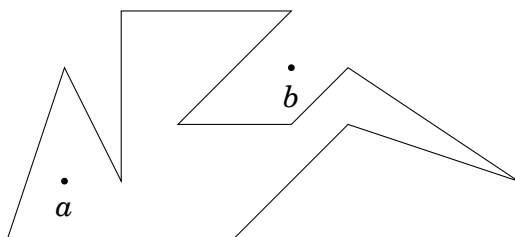


Нүктелері s_1 , s_2 және p болатын үшбұрыштың ауданын екі жолмен есептей аламыз: $\frac{1}{2}|s_2 - s_1|d$ немесе $\frac{1}{2}((s_1 - p) \times (s_2 - p))$. Демек ең қысқа қашықтық

$$d = \frac{(s_1 - p) \times (s_2 - p)}{|s_2 - s_1|}.$$

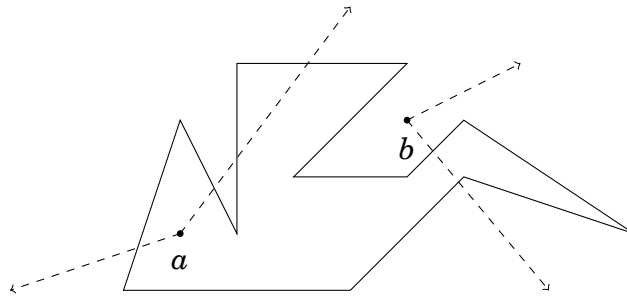
Көпбұрыштың ішіндегі нүкте

Келесі қарастырылатын тапсырма – нүктенің көпбұрыштың ішінде не сыртында орналасқанын тексеру. Мысалы, келесі суреттегі a нүктесі көпбұрыштың ішінде, ал b нүктесі көпбұрыштың сыртында орналасқан.



Есепті шығарудың ыңғайлы жолы – нүктеден басталатын кездейсоқ бағытта сәуле жүргізіп, көпбұрыштың қанша қабырғаларымен қиылысатынын есептеу. Егер қиылысу саны тақ болса, нүкте көпбұрыштың ішінде қалады, ал егер қиылысу саны жұп болса, нүкте көпбұрыштың сыртында болады.

Үлгі ретінде келесідей сәулелерді жүргізуге болады:



a -дан басталатын сәулелер көпбұрыштың қабырғаларымен 1 және 3 рет қиылысады, демек a көпбұрыштың ішінде орналасқан. Сәйкесінше b -дан басталатын сәулелер көпбұрыштың қабырғаларымен 0 және 2 рет қиылысады, демек b көпбұрыштың сыртында орналасып тұр.

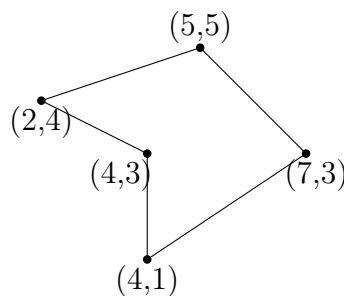
29.3 Көпбұрыш ауданы

Көпбұрыштың ауданын есептеудің төмендегідей жалпы формуласы бар (ол кейде Гаусстың аудан есептеу формуласы деп аталады):

$$\frac{1}{2} \left| \sum_{i=1}^{n-1} (p_i \times p_{i+1}) \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|,$$

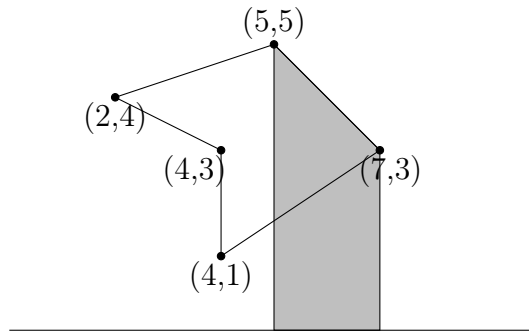
Көпбұрыштың төбелері $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, ..., $p_n = (x_n, y_n)$ деп берілген. Мұндағы қатар келетін p_i мен p_{i+1} төбелері – көпбұрыштың көршілес төбелері, ал бірінші мен соңғы төбе бірдей (яғни $p_1 = p_n$) болады.

Мысалы төмендегі көпбұрыштың ауданы



$$\frac{|(2 \cdot 5 - 5 \cdot 4) + (5 \cdot 3 - 7 \cdot 5) + (7 \cdot 1 - 4 \cdot 3) + (4 \cdot 3 - 4 \cdot 1) + (4 \cdot 4 - 2 \cdot 3)|}{2} = 17/2.$$

Формуланың идеясы бір қабырғасы көпбұрыштың қабырғасы болатын және екінші қабырғасы $y = 0$ сызықта болатын трапецияларды өтіп шығуға негізделеді. Мысалы:



осы трапецияның ауданы

$$(x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2},$$

мұндағы көпбұрыштың төбелері – p_i және p_{i+1} . Егер $x_{i+1} > x_i$ болса, аудан оң болады. Ал егер $x_{i+1} < x_i$ болса, аудан теріс болады.

Көпбұрыштың ауданы осындай барлық трапециялар аудандарының қосындысына тең, ал ол –

$$\left| \sum_{i=1}^{n-1} (x_{i+1} - x_i) \frac{y_i + y_{i+1}}{2} \right| = \frac{1}{2} \left| \sum_{i=1}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) \right|.$$

Ескерту: бұл жерде қосындының абсолюттік мәні алынған, себебі көпбұрыштардың төбелерін сағат тілі бағытымен не сағат тіліне қарсы бағытта өтуімізге байланысты қосындының мәні не оң, не теріс болады.

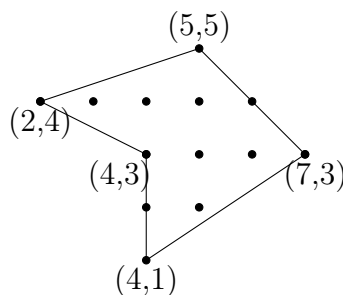
Пик теоремасы

Пик теоремасы көпбұрыштың төбелері бүтін координаталар болған жағдайда көпбұрыштың ауданын табудың жаңа жолын көрсетеді. Пик теоремасына сәйкес көпбұрыштың ауданы

$$a + b/2 - 1,$$

мұндағы a – көпбұрыштың ішінде орналасқан бүтін нүктелер саны, b – көпбұрыштың қабырғасында орналасқан бүтін нүктелер саны.

Мысалы осы көпбұрыштың ауданы



$$6 + 7/2 - 1 = 17/2 \text{ болады.}$$

29.4 Арақашықтық функциялары

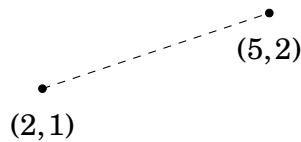
Арақашықтық функциясы екі нүкте арасындағы арақашықты белгілейді. Әдеттегі арақашықтық функциясы – Евклидтік арақашықтық, ондағы (x_1, y_1) мен (x_2, y_2) нүктелерінің арақашықтығы –

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Кей есептерде Манхэттендік арақашықтық та қолданылады, ондағы (x_1, y_1) мен (x_2, y_2) нүктелерінің арақашықтығы –

$$|x_1 - x_2| + |y_1 - y_2|.$$

Мысалы, келесі суретті қарастырайық:



Евклидтік арақашықтық



Манхэттендік арақашықтық

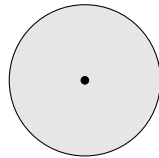
Нүктелердің Евклидтік арақашықтығы –

$$\sqrt{(5 - 2)^2 + (2 - 1)^2} = \sqrt{10}$$

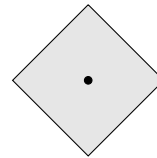
және Манхэттендік арақашықтығы –

$$|5 - 2| + |2 - 1| = 4.$$

Келесі суретте Евклидтік және Манхэттендік арақашықтықты қолдану арқылы орталық нүктеден бірлік қашықтықтағы аудан көрсетіледі:



Евклидтік арақашықтық

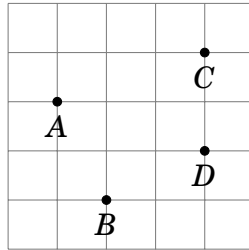


Манхэттендік арақашықтық

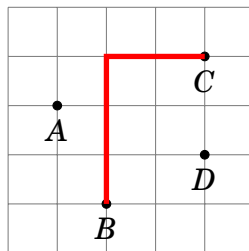
Координаталарды түрлендіру

Кейбір есептерді Евклидтік арақашықтықты қолданудың орнына Манхэттендік арақашықтықты қолданып шығарған оңайырақ. Үлгі ретінде келесі есепті қарастырайық: Координаттық жазықтықта n нүктелер берілген, екі нүкте арасындағы Манхэттендік арақашықтығы ең үлкен болатын арақашықты табыңыз.

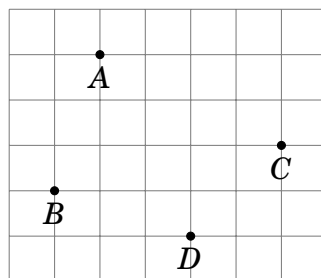
Мысалы келесі нүктелердің жиынын қарастырайық:



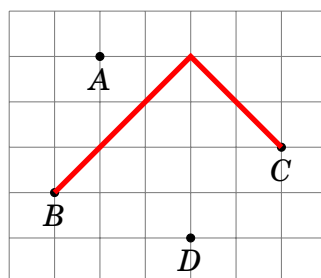
B мен C арасындағы максималды Манхэттендік арақашықтық 5-ті құрайды:



Манхэттендік арақашықтықпен жұмыс барысында (x, y) координатасын $(x + y, y - x)$ координатасына түрлендіруді қолданған тиімді болмақ. Сонда барлық координаталар 45 градусқа өзгереді. Мысалы, жоғарыдағы нүктелерді түрлендіруден кейін нәтиже төмендегідей болады:



және максималды арақашықтық келесідей болмақ:



Мысалға, түрленуі $p'_1 = (x'_1, y'_1)$, $p'_2 = (x'_2, y'_2)$ болатын $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$ нүктелерін қарастырайық. p_1 , p_2 нүктелерінің Манхэттендік арақашықтығын екі жолмен көрсете аламыз:

$$|x_1 - x_2| + |y_1 - y_2| = \max(|x'_1 - x'_2|, |y'_1 - y'_2|)$$

Мысалы, егер $p_1 = (1, 0)$ және $p_2 = (3, 3)$ болса, олардың түрлендірілген координаттары $p'_1 = (1, -1)$ және $p'_2 = (6, 0)$ болады. Олардың Манхэттендік арақашықтығы –

$$|1 - 3| + |0 - 3| = \max(|1 - 6|, |-1 - 0|) = 5.$$

Түрлендірілген координаталар Манхэттендік арақашықтықпен жұмыс істеу барысын жеңілдетіп, x пен y координаттарын жеке-жеке қарастыруға мүмкіндік береді. Манхэттендік арақашықтықты максималдау үшін

$$\max(|x'_1 - x'_2|, |y'_1 - y'_2|)$$

мәнін максималдайтын екі түрлендірілген нүктелерді табу керек. Ал оны табу оңай, өйткені түрлендіру координаттардың айырмашылығы не көлденеңінен, не тігінен максималды болуы қажет.

30-тарау. Сыпырма түзуге негізделген алгоритмдер

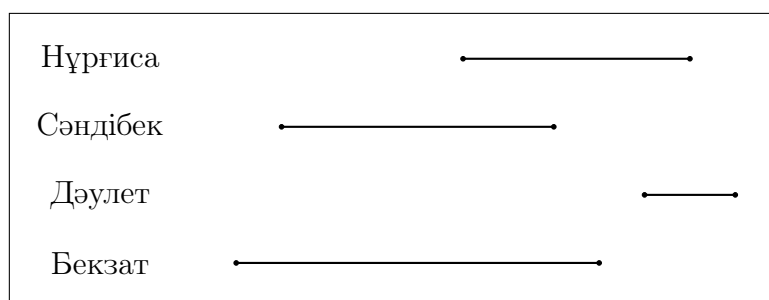
Көптеген геометриялық есептерді сыпырма түзуге негізделген алгоритмдер арқылы шығаруға болады. Ондай алгоритмдердің идеясы – есепті жазықтықтағы нүктелерге сәйкес оқиғалар жиынтығы ретінде көрсету. Кейін оқиғалар сәйкес x немесе y координаталары бойынша өсу ретімен өңделеді.

Үлгі ретінде келесі есепті қарастырайық: Бір мекемеде n жұмысшы еңбек етеді. Біз әр жұмысшының белгілі бір күндегі келу және кету уақыттарын білеміз. Біздің тапсырмамыз – нақтылы бір уақытта кеңсе ішінде болатын жұмысшылардың максималды санын табу.

Есепті әрбір қызметкерге олардың келу және кету уақытына сәйкес екі оқиға белгілеу арқылы шығаруға болады. Оқиғаларды сұрыптағаннан кейін, оларды рет бойынша өтіп, кеңседегі адам санын қадағалаймыз. Мысалы, төмендегі кесте

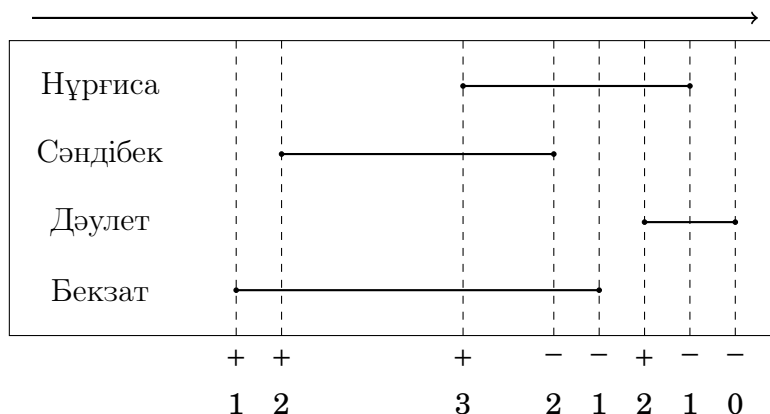
қызметкер	келу уақыты	кету уақыты
Нұрғиса	10	15
Сәндібек	6	12
Дәулет	14	16
Бекзат	5	13

келесі оқиғаларға сәйкес келеді:



Біз оқиғалар бойынша солдан оңға қарай өтіп, есептегіш айнымалыны қолданамыз. Жаңа жұмысшы келгенде, есептегіш айнымалыны әрдайым бірге арттырамыз. Ал жұмысшы кеткенде, есептегіш айнымалыны бірге азайтамыз. Алгоритм барысындағы есептегіш айнымалының максималды мәні есептің жауабы болады.

Мысалдағы оқиғалар келесі ретпен өңделеді:

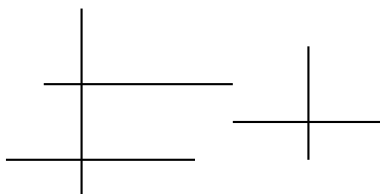


+ пен – таңбалары есептегіш айнымалының не артқанын, не кемігенін көрсетеді, ал есептегіш айнымалының мәні астында жазылған. Нұрғисаның келуі мен Сәндібектің кетуі арасындағы есептегіш айнымалының максималды мәні 3-ке тең.

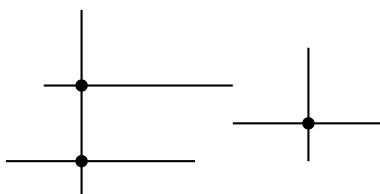
Алгоритмнің уақытша күрделілігі – $O(n \log n)$, себебі оқиғаларды сұрыптау $O(n \log n)$ уақыт алады, ал алгоритмнің қалған бөлігі $O(n)$ уақыт алады.

30.1 Қиылысу нүктелері

n горизонталды немесе вертикалды кесінділер жинағы берілген. Қиылысу нүктелерінің жалпы санын табатын есепті қарастырайық. Мысалы, кесінділер төмендегідей орналасқан кезде



3 қиылысу нүктелері пайда болады:



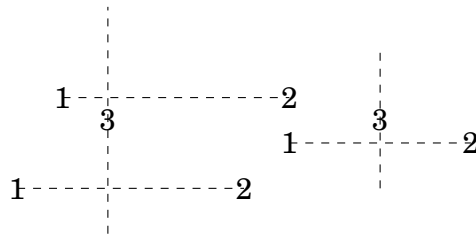
Бұл есепті $O(n^2)$ уақытта шығарған жеңіл, өйткені біз барлық кесінділердің жұптары бойынша өтіп, олардың қиылысуын тексере аламыз. Дегенмен бұл есепті сыпырма түзуіне негізделген алгоритм мен аралық сұратым жасайтын деректер құрылымы арқылы $O(n \log n)$ уақытта тиімдірек шығара аламыз.

Мұндағы идея кесінділердің шеткі нүктелерін солдан оңға қарай өңдеуге және 3 түрлі оқиғаларға назар аударуға негізделеді. Олар:

- (1) көлденең кесінді басталады
- (2) көлденең кесінді аяқталады

(3) тік кесінді

Келесі оқиғалар жоғарыдағы мысалға сәйкес келеді:



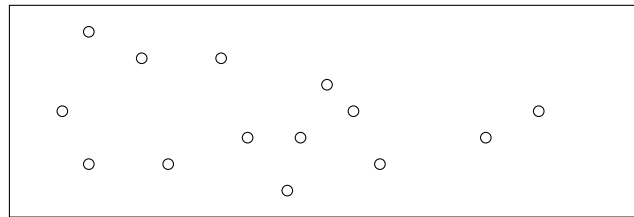
Оқиғалар бойынша солдан оңға қарай өтеміз және белсенді көлденең кесіндісі бар y координаталардың жинағын сақтайтын деректер құрылымын қолданамыз. 1-оқиғада біз y координатасын жиымға қосамыз, ал 2-оқиғада y координатасын жиымнан өшіреміз.

Қиылысу нүктелері 3-оқиғада анықталады. y_1 мен y_2 нүктелері арасында тік кесінді келген кезде, y координатасы y_1 мен y_2 арасындағы белсенді көлденең кесінділердің санын есептейміз және оны жалпы қиылысу нүктелерінің санына қосамыз.

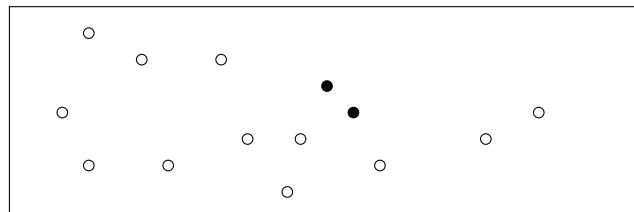
Көлденең кесінділердің y координаталарын сақтау үшін бинарлы индекстелген дарақ немесе кесінділер дарағын (бәлкім индекстерді сығымдау арқылы) пайдалануға болады. Сондай деректер құрылымы қолданылған кезде, әр оқиғаны өңдеу $O(\log n)$ уақыт алады, демек алгоритмнің қорытынды уақытша күрделілігі – $O(n \log n)$.

30.2 Ең жақын жұп есебі

n нүктелер жиынтығы берілген. Біздің келесі тапсырмамыз – Евклидік арақашықтығы ең аз болатын екі нүктені табу. Мысалы, егер нүктелер төмендегідей орналасса,



келесі нүктелерді табуымыз қажет:



Бұл — сыпырма түзуіне негізделген алгоритм¹

Әдебиеттер

- [1] Google Code Jam Statistics (2017), <https://www.go-hero.net/jam/17>
- [2] The International Olympiad in Informatics Syllabus, <https://people.ksp.sk/~misof/ioi-syllabus/>
- [3] Croatian Open Competition in Informatics, <http://hsin.hr/coci/>
- [4] USA Computing Olympiad, <http://www.usaco.org/>
- [5] SZKOpul, <https://szkopul.edu.pl/>
- [6] S. S. Skiena and M. A. Revilla. Programming Challenges: The Programming Contest Training Manual, Springer, 2003.
- [7] S. Halim and F. Halim. Competitive Programming 3: The New Lower Bound of Programming Contests, 2013.
- [8] K. Diks et al. Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions, University of Warsaw, 2012.
- [9] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. Introduction to Algorithms, MIT Press, 2009 (3rd edition).
- [10] J. Kleinberg and É. Tardos. Algorithm Design, Pearson, 2005.
- [11] S. S. Skiena. The Algorithm Design Manual, Springer, 2008 (2nd edition).
- [12] M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman and Company, 1979.
- [13] J. Bentley. Programming Pearls. Addison-Wesley, 1999 (2nd edition).
- [14] D. E. Knuth. The Art of Computer Programming. Volume 3: Sorting and Searching, Addison–Wesley, 1998 (2nd edition).
- [15] 27-Queens Puzzle: Massively Parallel Enumeration and Solution Counting. <https://github.com/preusser/q27>
- [16] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. Journal of the ACM, 21(2):277–292, 1974.

- [17] D. Pearson. A polynomial-time algorithm for the change-making problem. *Operations Research Letters*, 33(3):231–234, 2005.
- [18] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [19] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.
- [20] P. W. Kasteleyn. The statistics of dimers on a lattice: I. The number of dimer arrangements on a quadratic lattice. *Physica*, 27(12):1209–1225, 1961.
- [21] H. N. V. Temperley and M. E. Fisher. Dimer problem in statistical mechanics – an exact result. *Philosophical Magazine*, 6(68):1061–1063, 1961.
- [22] A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. In *Proceedings of the 55th Annual Symposium on Foundations of Computer Science*, 621–630, 2014.
- [23] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, 88–94, 2000.
- [24] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Annual Symposium on Combinatorial Pattern Matching*, 36–48, 2006.
- [25] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.
- [26] P. Stańczyk. *Algorytmika praktyczna w konkursach Informatycznych*, MSc thesis, University of Warsaw, 2006.
- [27] J. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, C-29(7):571–577, 1980.
- [28] M. Dima and R. Ceterchi. Efficient range minimum queries using binary indexed trees. *Olympiad in Informatics*, 9(1):39–44, 2015.
- [29] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.
- [30] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
- [31] L. R. Ford. *Network flow theory*. RAND Corporation, Santa Monica, California, 1956.
- [32] D. Fanding. A faster algorithm for shortest-path – SPFA. *Journal of Southwest Jiaotong University*, 2, 1994.

- [33] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- [34] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [35] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.
- [36] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.
- [37] J. E. Hopcroft and J. D. Ullman. A linear list merging algorithm. Technical report, Cornell University, 1971.
- [38] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.
- [39] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.
- [40] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, Addison–Wesley, 1998 (3rd edition).
- [41] A. V. Aho, J. E. Hopcroft and J. Ullman. *Data Structures and Algorithms*, Addison–Wesley, 1983.
- [42] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.
- [43] B. Aspvall, M. F. Plass and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.
- [44] S. Even, A. Itai and A. Shamir. On the complexity of time table and multi-commodity flow problems. *16th Annual Symposium on Foundations of Computer Science*, 184–193, 1975.
- [45] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, 12–20, 1984.
- [46] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.
- [47] C. Hierholzer and C. Wiener. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6(1), 30–32, 1873.

- [48] H. C. von Warnsdorf. Des Rösselsprunges einfachste und allgemeinste Lösung. Schmalkalden, 1823.
- [49] I. Parberry. An efficient algorithm for the Knight's tour problem. *Discrete Applied Mathematics*, 73(3):251–260, 1997.
- [50] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.
- [51] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.
- [52] R. K. Ahuja and J. B. Orlin. Distance directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics*, 38(3):413–430, 1991.
- [53] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.
- [54] M. Beck, E. Pine, W. Tarrat and K. Y. Jensen. New integer representations as the sum of three cubes. *Mathematics of Computation*, 76(259):1683–1690, 2007.
- [55] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.
- [56] F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, 296–303, 2014.
- [57] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.
- [58] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, 1961.
- [59] R. Freivalds. Probabilistic machines can use less running time. In *IFIP congress*, 839–842, 1977.
- [60] C. L. Bouton. Nim, a game with a complete mathematical theory. *Annals of Mathematics*, 3(1/4):35–39, 1901.
- [61] R. Sprague. Über mathematische Kampfspiele. *Tohoku Mathematical Journal*, 41:438–444, 1935.
- [62] P. M. Grundy. Mathematics and games. *Eureka*, 2(5):6–8, 1939.
- [63] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

- [64] J. Pachocki and J. Radoszewski. Where to use and how not to use polynomial string hashing. *Olympiads in Informatics*, 7(1):90–100, 2013.
- [65] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
- [66] M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.
- [67] Codeforces: On "Mo's algorithm", <http://codeforces.com/blog/entry/20032>
- [68] C. Kent, G. M. Landau and M. Ziv-Ukelson. On the complexity of sparse exon assembly. *Journal of Computational Biology*, 13(5):1013–1027, 2006.
- [69] M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, 151–162, 1975.
- [70] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.

Пәндік сілтеме

- 2SAT есебі, 165
- 2SUM есебі, 80
- 3SAT есебі, 167
- 3SUM есебі, 81

- complex, 272
- сыбайластық матрицасы, 116
- сыбайластық тізімі, 115

- inclusion–exclusion principle, 217

- mex функциясы, 244

- next_permutation, 50
- NP-қиын есептер, 21

- programming language, 2

- queue, 43
- quickselect, 237
- quicksort, 237

- random_shuffle, 39
- reverse, 39

- set, 37
- sort, 29, 39
- SPFA алгоритм, 129
- string, 36

- typedef, 7

- Z-алгоритм, 252
- Z-жиым, 252

- Байланысты граф, 112, 123
- Беллман-Форд алгоритмі, 126
- Бернсайд леммасы, 219
- Бине формуласы, 13
- Биномдық үлестірім, 235
- Гамильтон жолы, 183
- Гамильтон шынжыры, 183
- Гармоникалық қосынды, 10
- Гаусстың аудан есептеу формуласы, 277
- Геометриялық прогрессия, 10
- Геометриялық үлестірім, 235
- Герон формуласы, 271
- Голдбах гипотезасы, 203
- Гранди ойыны, 246
- Гранди саны, 244
- Де Брёйн тізбегі, 184
- Дейкстра алгоритмі, 129, 157
- Дилуорс теоремасы, 199
- Диофант теңдеуі, 208
- Дирак теоремасы, 184
- Евклид алгоритмі, 205
- Евклид формуласы, 210
- Евклидтік арақашықтық, 279
- Екі нұсқағыш әдісі, 79
- Енгізу (input) мен шығару (output), 3
- Жол бүркемесі, 197
- Каталан сандары, 215
- Кели теоремасы, 220
- Кирхгоф теоремасы, 228
- Компонент, 112
- Косараджу алгоритмі, 163
- Краскал алгоритмі, 145
- Кёниг теоремасы, 196
- Лагранж теоремасы, 210
- Лаплас матрицасы, 229
- Лас Вегас алгоритмі, 237
- Левенштейн арақашықтығы, 75
- Лежандр гипотезасы, 203
- Максималды ағын, 187
- Манхэттендік арақашықтық, 279
- Марковтық тізбе, 236

Мо алгоритмі, 260
 Монте Карло алгоритмі, 237
 Оре теоремасы, 184
 Паскаль үшбұрышы, 214
 Пик теоремасы, 278
 Пифагор үштіктері, 210
 Предикат, 12
 Прим алгоритмі, 150
 Прюфер кодтары, 221
 Салмақталған граф, 113
 Уарнсдорф ережесі, 186
 Уилсон теоремасы, 210
 Уәзір есебі, 51
 Факториал, 13
 Фаульхабер формуласы, 9
 Фенвик дарағы, 88
 Ферма теоремасы, 206
 Фибоначчи сандары, 13, 210, 225
 Флойд алгоритмі, 160
 Флойд-Уоршелл алгоритмі, 132
 Форд-Фалкерсон алгоритмі, 188
 Хаффман кодтауы, 64
 Хемминг арақашықтығы, 102
 Хирхольцер алгоритмі, 181
 Хол теоремасы, 195
 Цекендорф теоремасы, 210
 Шпраг-Гранди теоремасы, 243
 Эдмондс-Карп алгоритмі, 190
 Эйлер жолы, 179
 Эйлер теоремасы, 206
 Эйлер функциясы, 205
 Эйлер шынжыры, 179
 Эйлер өту әдісі, 173
 Эратосфен елегі, 204
 айналым, 247
 айырма жиымы, 96
 айырмашылық, 11
 алгебралық толықтауыш, 224
 алмастыру, 50
 амортизацияланған талдау, 79
 антитізбек, 199
 анықтауыш, 224
 аралық сұратымдар, 85
 арифметикалық прогрессия, 10
 ат сапары, 185
 аудару, 222
 ашкөз алгоритмдер, 58
 ағын, 187
 ағынды масштабтау алгоритмі, 191
 баба, 168
 басымдылық кезегі, 43
 бағытталған граф, 112
 берік байланысты граф, 162
 берік байланысты компонент, 162
 бинарлы дарақ, 142
 бинарлы индекстелген дарақ, 88
 бинарлық код, 63
 бинарлық ізденіс, 31
 биномдық коэффициент, 213
 битсет, 41
 биттік көрсетілім, 97
 биттік терістеу, 99
 биттік ығысу, 99
 бояу, 238
 бояулы, 114
 бірлік матрица, 223
 біртекті граф, 113
 бірігу, 11
 бірізді, 142
 біріктіру бойынша сұрыптау, 27
 бірқалыпты үлестірім, 235
 бүтін сандар, 5
 бөлгіш, 201
 бөлінгіш, 201
 вектор, 35, 222, 272
 векторлық көбейтінді, 274
 гармоникалық қосынды, 204
 геометрия, 271
 граф, 111
 граф компоненті, 162
 дарақ, 112, 136
 дарақ айналымының жиымы, 169
 дарақтағы сұратымдар, 168
 деректер құрылымы, 35
 деректерді сығымдау, 63
 диаметр, 138
 дизъюнкция, 12, 98
 динамикалық бағдарламалау, 66
 динамикалық жиым, 35
 динамикалық кесінділер дарағы,
 266
 дәреже, 113

егіз сандар, 203
 екі жақты тізбек, 42
 екі ұялы граф, 114, 124
 екі өлшемді кесінділер дарағы, 269
 ені бойынша іздеу, 121
 ең жақын жұп, 284
 ең жақын кішірек элементтер, 81
 ең кіші ортақ еселік, 205
 ең қысқа жол, 126
 ең үлкен ортақ бөлгіш, 205
 ең ұзын өспелі іштізбек, 71
 жай көбейткіштерге жіктеу, 201
 жай сан, 201
 жалқау кесінділер дарағы, 263
 жалқау таратылу, 263
 жапырақ, 136
 жақшалар тізбегі, 216
 жақын арадағы ата-тек, 172
 жиын, 11
 жиын теориясы, 11
 жол, 111, 247
 жол хеші, 249
 жоюшы дизъюнкция, 99
 жылжымалы терезе, 83
 жылжымалы терезе минимумы, 83
 жұп, 30
 жұптасу, 193
 импликация, 12
 инверсиялар, 26
 индекстерді сығымдау, 95
 итератор, 39
 квадратты түбір алгоритмдер, 20
 квадраттық түбір алгоритм, 256
 квантор, 12
 квардаттық алгоритм, 20
 кездейсоқ шамалар, 234
 кемел сан, 202
 кемімелі, 142
 кері матрица, 225
 кесінділер дарағы, 91, 262
 кесінділердің қиылысуы, 275
 кеңейтілген Эвклид алгоритмі, 208
 код сөз, 63
 комбинаторика, 212
 комплекс сан, 272
 конъюнкция, 12
 конъюнкция, 98
 кортеж, 30
 кубтық алгоритм, 20
 кірістің жарты дәрежесі, 113
 күрделілік кластары, 20
 көбейткіш, 201
 көпмүшелік хэш, 249
 көпіршікті сұрыптау, 25
 көрші, 113
 лексикографикалық рет, 248
 логарифмдік алгоритм, 20
 логарифм, 13
 логика, 12
 макрос, 8
 максималды жұптасу, 193
 максималды тәуелсіз жиын, 196
 максималды қаңқалы дарақ, 144
 максимум сұратымы, 85
 математикалық күтім, 234
 матрица, 222
 матрица дәрежесі, 224
 матрица көбейтіндісі, 238
 матрицаларды көбейту, 223
 мемоизация, 68
 мизер ойыны, 243
 минималды төбелер бүркемесі, 196
 минималды қаңқалы дарақ, 144
 минималды қима, 188, 191
 минимум сұратымы, 85
 мирасқорлар графы, 158
 модуль бойынша кері сан, 207
 модульдік арифметика, 5, 206
 мультиномдық коэффициент, 215
 мінсіз жұптасу, 195
 натурал логарифм, 14
 ним ойыны, 242
 ним қосындысы, 242
 нүкте, 272
 ортада кездесу, 56
 период, 247
 полиномдық алгоритм, 21
 префикс, 247
 префикс дарағы, 248
 префиксті қосындылар жиымы, 86
 рандомизацияланған алгоритм, 237
 ретсіздіктер, 218

реттік статистика, 237
 салыстыру операторлары, 30
 салыстыру функциясы, 31
 санамалы сұрыптау, 28
 сандар теориясы, 201
 сиретілген кесте, 87
 сиретілген кесінділер дарағы, 266
 стек, 42
 суффикс, 247
 сызықтық рекурренттілік, 225
 сызықтық уақыт алгоритм, 20
 сыпырма түзу, 282
 сұрыптау, 25
 сөздік, 38
 тереңдігі бойынша іздеу, 119
 теріс цикл, 128
 терістеу, 12
 толық граф, 113
 толықтауыш, 11
 топологиялық сұрыптау, 153
 туған күн парадоксы, 251
 түбір, 136
 түбірлі дарақ, 136
 түзету арақашықтығы, 75
 тұрақты алгоритм, 20
 тұрақты фактор, 22
 тәуелсіз жиын, 196
 тәуелсіздік, 233
 төбе, 111
 төбелер бүркемесі, 196
 уақытша күрделілігі, 17
 функционалды граф, 158
 хэш, 249
 цикл, 111, 123, 153, 159
 циклді анықтау, 159
 шартты ықтималдылық, 233
 шаршы матрица, 222
 шек, 248
 шығыстың жарты дәрежесі, 113
 ықтималдылық, 230
 эвристика, 186
 эквиваленттеу, 12
 ішдарақ, 136
 ішжиымдардың ең жоғары қосындысы, 22
 ішжиын, 11, 48
 ішжол, 247
 іштізбек, 247
 қайта іздеу алгоритмі, 51
 қалдық, 5
 қалдықтар туралы қытай теоремасы, 209
 қалқымалы нүктелі сандар, 6
 қарапайым граф, 114
 қашықтық функциясы, 279
 қақтығыс, 250
 қаңқалы дарақ, 144, 228
 қима, 188
 қиылыспайтын жиындар құрылымы, 148
 қиылысу, 11
 қиылысу нүктесі, 283
 қоржын, 74
 қосынды сұратымы, 85
 қыр, 111
 қырлар тізімі, 117
 үдемелі, 142
 үйінді, 43
 үлгімен салғастыру, 247
 үлестірім, 235
 ұзақ сақталушы кесінділер дарағы, 267
 ұл, 136
 ұтылыс күйі, 240
 ұтыс күйі, 240
 әке, 136
 әліпби, 247
 әмбебап жиын, 11
 өзара жай, 205