Състезателно програмиране за 6 клас

Петър Петров, Марин Шаламанов

10 февруари 2021 г.

Съдържание

0.1 Въведение

Неща, които трябва да споменем тук:

- Целта на книгата е прадстави на състезателите от D група, всички теми и материал, които трябва да знаят за да се преставят отлично на регионалните и националните състезания по информатика.
- Темите съдържат много задачи. Когато се стигне до задача е препоръчително читателя да спре да чете текста и да се опита да реши задачата сам. Чак след това да чете нататък. Често решаването на тези задачи е необходимо за успешното разбиране на материала.
- Може да се използва от ученици самостоятелно и от учители в редовно часове или школи.
- Линк към система, където са качени задачите.

Глава 1

Стил на писане

1.1 Основни правила

При писането на код, както на всяко други нещо, има добри и лоши практики. Разбира се нормално е да има разлика, ако пишем код по време на състезание и при работа в голям екип. Много хора започват да се учат като състезатели и се научават, че техния код не е нужно да следва правила. Впоследствие започват работа и разбират, че трябва да се отучат от този навик, и че всеки сериозен екип има серия от правила за спазване. Не правете тази грешка, не отричайте правилата! Истината разбира се не е черна или бяла, тя е някъде по средата. Има правила, които по-време на състезание ни помагат да структурираме по-добре нашите мисли и най-важното - по-лесно да намираме грешките, които допускаме.
В следващите секции описваме основните практики, които смятаме че са полезни да се спазват от състезателите.

1.2 Подреждане на кода

1.2.1 Индентация

Всеки път когато влезем в блок отместването на кода отляво(индентацията) се увеличава с един таб или четири интервала. Когато блока завърши, отместването се връща на предишното ниво. Под блок разбираме тяло на функция, `if`, `else`, `for`, `do` и `while`.

1.2.2 Една команда на ред

На всеки ред трябва да има най-много една команда.

1.2.3 Къдрави скоби

Използват се винаги с `if`, `else`, `for`, `do` и `while`, дори когато тялото е празно или има една команда. Отварящата скоба се слага в края на реда към който се отнася. Затварящата скоба е сама на ред със същото отстояние като това на реда на отварящата скоба.
Единственото изключение на това правило, което ще препоръчаме е ако в тялото

на `if` има една команда и тя е ключова дума - `continue`, `break`, `return`, да пропуснем скобите и да напишем командата на същия ред. Препоръчително е този ред да е последван от празен ред, за да е съвсем разпознаваемо допуснатото изключение, което също така ще отбележим, че променя и нормалната последователност на изпълнение на кода. Ето така изглежда единствения случай на `if` без къдрави скоби:

```
if (условие) continue/break/return;
```

1.2.4 if, if-else, if-else-if-else

Ето така трябва да изглеждат if-else структурите:

```
if (условие) {
    команди;
}

if (условие) {
    команди;
} else {
    команди;
}

if (условие) {
    команди;
} else if (условие) {
    команди;
} else {
    команди;
}
```

1.2.5 for

Ето така трябва да изглежда цикъла for:

```
for (инициализация; условие; промяна) {
    команди;
}
```

1.2.6 while

Ето така трябва да изглежда цикъла while:

```
while (условие) {
    команди;
}
```

1.2.7 do-while

Ето така трябва да изглежда цикъла do-while:

```
do {
    команди;
} while (условие);
```

1.3 Променливи

1.3.1 Предназначение на променлива

Основната идея е една променлива - едно предназначение. Ако искаме да сменим предназначението създаваме отделна променлива.

1.3.2 Именуване

Имената на променливите трябва да ни подсказват за тяхното предназначение. Същевременно е желателно да са кратки, за да не отнемат много време за писане. Компромиса яснота-краткост го оставяме на вас.
Все пак най-често се използват еднобуквени имена. Те не са описателни, за това е добре да изградите ваша конвенция за стандартни значения на еднобуквените имена. Например ако имаме редица от числа, броя е n, масива е a. Ако имаме низ използваме s. Брой заявки пазим в q. Управляваща променлива във for - i, j.

1.3.3 Глобални или локални

Всички променливи по правило ги правим локални. Все пак може да направим променлива глобална, ако планираме да я използваме в различни функции и не искаме да я подаваме като параметър. Обикновено масивите ги правим глобални и като бонус получаваме начални стойности.

1.3.4 Създаване

Създаваме локални променливи близо до мястото, където ги използваме за първи път.

```
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        int d = countDivisors(a[i][j]);
        br[d]++;
    }
}
```

В случая променливата `d` се използва във вътрешния цикъл. Няма нужда да я създаваме във външния цикъл или преди това.

1.3.5 Начални стойности

На локалните променливи даваме начални стойности в момента на създаването. Или ги четем от входа в близките няколко реда. При създаване на локални масиви използвайте кудрави скоби за начални стойности по подразбиране - `int a[5] = {}`.

1.3.6 Скриване

В обхвата на дадена променлива никога не деклариране друга променлива със същото име. Така например ако имаме глобална променлива никъде в кода не трябва да създаваме променлива с такова име.

1.4 Функции

Функциите са незаменими в няколко отношения:

- Разделят кода на отделни логически елементи.
- Позволяват да преизползваме код.
- Позволяват да тестваме кода на части.

1.4.1 Кудрави скоби

Важат правилата като за другите блокове - отварящата е в края на реда, където започва функцията. Затварящата скоба е сама на ред със същото отстояние като това на реда на отварящата скоба, като обикновено това отстояние е 0.

1.4.2 Връщана стойност

Всяка функция, която не е `void` трябва да връща резултат. Без значение кои редове на функцията ще се изпълнят, задължително накрая трябва да се стига до `return`. В противен случай може да се получат неочаквани резултати и това е проблем, които може да е труден за откриване.

1.4.3 Частни случай

Понякога за конкретни стойности на параметрите на функцията трябва да се върне конкретен резултат и това не е част от основната логика. В такъв случай е добре тези проверки да са в началото на функцията. Така основната логика няма да е вътре в условие и няма да е отместена излишно.

```
bool isPrime(int n) {
    if (n < 2) {
        return false;
    }

    ...
}
```

В примера при проверка за просто число, ако числото е по-малко от 2 това е частен случай, който не се покрива от нашата логика. За това още в началото го проверяваме и връщаме резултат. След това съществената част от кода не се налага да бъде отместена.

Глава 2

Тънкости при вход и изход

2.1 Защо ни е това?!

Обикновено в началото на една програма четем входни данни и в края на програмата изкарваме резултат. В повечето задачи това се прави лесно и не представлява проблем. Обаче има изключения и понякога се налага да знаем някои специфики, за да може да прочетем входа или да форматираме изхода. Защо е важно винаги да можем да прочитаме входа? Отговорът на този въпрос трябва да е лесен! Дори с перфектно решение, вероятно няма да вземем точки, което е все едно да се откажем от тази задача. Да загубим точки, от цяла една задача, понеже не знаем нещо свързано с четенето на вход, е нещо което не трябва да допускаме.

2.2 `cin`

Нещото което отлично знаем, е че може да прочетем едно число, един символ или един низ по следния начин:

```
cin >> n;
```

Трябва да знаете, че `cin` използва интервалите и новите редове като разделител за входните данни. Дори при четене на символи, `cin` пропуска интервалите и новите редове.
Какво всъщност се случва при четене със `cin`?
Четенето не винаги започва точно от мястото до което сме стигнали. Ще прескочим всички интервали и нови редове преди да започнем да четем.
След като знаем от къде започва четенето сега ни интересува какво точно ще прочетем. Това зависи от типа на променливата която искаме да прочетем.
Цяло число - ако четем цяло число трябва в началото да сме стигнали до цифра. Като тръгнем от тази цифра ще прочетем всички последващи цифри. Ще спрем на първия символ който не е цифра - може да е интервал, нов ред, точка, тире и всеки друг символ.
Дробно число - ако имаме поредица от цифри, която не завършва с точка, ще бъде същото все едно четем цяло число. Ако има цифри, последвани от точка, последвани от други цифри, ще прочетем всичко преди края на втората серия от цифри.
Символ - ще прочетем един символ.
Низ - ще прочетем всички символи преди следващия интервал или до края на реда.

Каквото и да прочетем спираме веднага след последния прочетен символ, от където ще започне следващото четене. Важно е да се отбележи, че след четене винаги оставаме на реда на който се намира прочетеното нещо.
Примери:

четем	начало	края	прочетено
цяло число	|123.456.7	123|.456.7	123
цяло число	| 15-33	15|-33	15
дробно число	|66.7.8	66.7|.8	66.7
дробно число	| 1 8	1| 8	1
символ	| a9good	a|9good	a
символ	|.-..	.|-..	.
низ	|abc123 456	abc123| 456	abcd123
низ	| .df0-dfs d	.df0-dfs| d	.df0-dfs

Да разгледаме как може да прочетем следният вход: "10 hello c". В началото курсорът е на първия символ - "|10 hello c"и ние искаме да прочетем цяло число. Прочитаме "10"и курсорът се премества на интервала след това което сме прочели - "10| hello c". Сега искаме да прочетем низ. Понеже в момента сме на интервал, то той се прескача и четенето започва от "h"и свършва преди следващия интервал. Прочели сме "hello"и курсорът е на интервала - "10 hello| c". Сега искаме да прочетем символ и понеже пак има интервали, даже два, ги прескачаме. Прочитаме символа "c"и курсора отива след символа, където или е краят на реда ила на целия вход - "10 hello c|". Ако има още редове и продължим да четем, ще прескочим на следващия ред и ще продължим по същия начин.

2.3 getline

Да разгледаме следните задачи:

Задача 2.1. Дадени са три символа, две малки латински букви и един интервал, в разбъркан ред. Намерете на коя позиция е интервалът. Например “a b” → “2”.

Има няколко начина да се справим с проблема с интервалите. Това което ще правим е да четем входа ред по ред. Когато четем цял ред, получаваме един низ, състоящ се от всички символи на реда. Това става по следния начин:

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

Това което прави "getline"е да прочете всички символи от мястото до което сме стигнали с четенето до края на реда и да ги върне като низ. След това веднага отиваме на следващия ред.

Решението на задача 2.1. би изглеждало така:

```
#include <iostream>
using namespace std;

int main() {
  string line;
```

```
  getline(cin, line);

  if (line[0] == ' ') {
    cout << 1 << endl;
  } else if (line[1] == ' ') {
    cout << 2 << endl;
  } else {
    cout << 3 << endl;
  }

  return 0;
}
```

Задача 2.2. Дадени са два реда символи - малки латински букви и интервали. Намерете колко броя букви има на първия ред.

Задача 2.3. Ако имаме един низ (*string s*), то каква е разликата между *cin » s* и *getline(cin, s)*?

2.4 while(cin)

Задача 2.4. На един ред са дадени няколко цели числа разделени с по един интервал. Намерете броя на дадените числа.

Обикновено броят на числата е даден в условието или го четем от входа. Затова входа е нестандартен, но въпреки това решението е лесно.
Може да прочетем целия ред в един низ и да преброим интервалите. Числата ще са с едно повече от интервалите.

Задача 2.5. На отделни редове са дадени няколко цели числа. Намерете броя на дадените числа.

Тази задача много прилича на предходната, но няма как да стане по същия начин. За да прочетем всички числа от входа, трябва да разберем какво означава край на входа. Всъщност, входните данни представляват един текстов файл. Когато тестваме вашите програми им казваме от кой файл да четат входните данни. Съответно край на входа означава край на файла.
Когато пишем една програма, ние работим с конзолата и нямаме файл с входни данни. За да кажем на нашата програма, че сме свършили с писането на входа в конзолата, използваме специалната клавишна комбинация `CTRL + Z`. Ако натиснем едновременно тези два клавиша в конзолата се появява специалният символ `^Z`, който за програмата означава край на входа. След като се появи специалният символ трябва да натиснем `ENTER`, за да го прочете програмата. Ако въведем нещо след специалният символ програмата няма да го прочете.
Да погледнем как изглеждат нещата от програмна гледна точка. Числа четем със *cin » n*. Тази операция връща резултат от булев тип, дали сме успели да прочетем число или не. Може да си запишем резултата *bool readSuccess = cin » n* или направо да го използваме за условие за проверка *if (cin » n)*.

Задача 2.6. На входа са дадени едно или две числа. Намерете колко са дадените числа.

Тъй като винаги има поне едно число, то него ще го прочетем нормално. След това може да използваме *if (cin » n)*. Ако условието е изпълнено, значи са дадени две числа, иначе числото е едно.

```
#include <iostream>
using namespace std;

int main() {
  int n;
  cin >> n;
  if (cin >> n) {
    cout << 2 << endl;
  } else {
    cout << 1 << endl;
  }

  return 0;
}
```

Пуснете програма. Пробвайте да въведете едно или две числа и специалният символ. Да се върнем на задачата за намиране на броя числа дадени на входа. Решението е с един цикъл. Всеки път когато успеем да прочетем число влизаме в цикъла и увеличаваме брояча на числата. Когато числата свършат излизаме от цикъла и изкарваме резултата.

```
#include <iostream>
using namespace std;

int main() {
  int n, br = 0;
  while (cin >> n) {
    br = br + 1;
  }
  cout << br << endl;

  return 0;
}
```

Важно е да отбележим, че по подобен начин може да четем ред по ред до края на входа - *while(getline(cin, s))*.

2.5 cin + getline

Задача 2.7. На първия ред са дадени две числа, на втория малки латински букви и интервали. Търсим сбора на двете числа, умножен по броя на интервалите на втория ред.

Проблемът в тази задача пак идва от интервалите. За да може да ги преброим трябва да прочетем втория ред с *getline*. Ако по същия начин прочетем първия ред

ще трябва допълнително да се занимаваме да отделим числата. За това най-лесно изглежда да се опитаме, да прочетем първият ред със *cin* и вторият с *getline*.

Нека да видим как би изглеждала тази комбинация със *cin* и *getline* за входа:

```
1 2
a b d fsafasd
```

В началото сме на единицата и четем число със *cin»a*. Прочели сме числото едно и вече сме на интервала между двете числа. Прочитаме второто число със *cin»b*. Прескачаме интервала преди числото, прочитаме числото 2 и оставаме в края на същия ред. Ето къде сме:

```
1 2|
a b d fsafasd
```

Останали сме на края на първия ред, а с *getline* четенето започва от текущия ред. Трябва някак си да минем в началото на втория ред. Въпросът е как да стигнем до следната сутиация:

```
 1 2
|a b d fsafasd
```

Отговорът е, че трябва да използваме *getline*. Ще повторим, че *getline* чете от текущото място до края на реда и след това отива на следващия ред. Всъщност ние ще прочетем един празен низ, но това не е от значение, важното е че ще отидем на следващия ред. След което с още един *getline*, ще прочетем и него.
Четенето на входа в задачата изглежда по следния начин:

```
int a, b;
cin >> a >> b;
string s;
getline(cin, s);
getline(cin, s);
```

Останалата част от програмата я оставяме за упражнение.

2.6 getline версия 2

Задача 2.8. На един ред са дадени три думи състоящи се от малки латински букви, разделени с точки. Намерете дължината на най-дългата дума.

Ясно е, че трябва да намерим дължините на всяка една от думите.
Един вариант за решение е да прочетем целия ред и спрямо положенията на точките да намерим дължините на трите думи.
Тук обаче ще използваме една нова опция на *getline* - *getline(cin, низ, символ)*. Този *getline* ще започне да чете от текущата позиция и ще прочете всичко докато не срещне символ който ние изберем. Самия символ няма да го прочете, но ще го прескочи и следващото четене ще започне след този символ.

```
#include <bits/stdc++.h>
using namespace std;

int main() {
  string s1, s2, s3;
  getline(cin, s1, '.');
  getline(cin, s2, '.');
  getline(cin, s3);
  cout << max(s1.size(), max(s2.size(), s3.size())) << endl;

  return 0;
}
```

Първите два низа ги прочитаме с новият вариант на *getline* докато стигнет точка. Третият низ го четем до края на реда.

2.7 Вход++

Задача 2.9. Дадени са един милион цели числа. Намерете сумата им.

На пръв поглед решението е очевидно. На състезание обаче е възможно решението да не влиза във времевия лимит. Това идва поради факта, че входните данни са много и четенето им ще отнеме време. За да забързаме четенето използваме един трик, който на този етап не е нужно да знаете какво точно означава. За сметка на това обаче трябва да го научите на изуст. Трикът се състои в това да добавим два реда в началото на главната функция на програмата.

```
#include <iostream>
using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(NULL);

  // code

  return 0;
}
```

Тези два реда е задължително да добавяме във всяка програма където има голям вход (над сто хиляди числа). Силно препоръчително е всяка една програма да започва с тях.

Задача 2.10. Напишете програма която да извежда числата от 1 до 10000 разделени с интервали. Напишете я веднъж използвайки трика и веднъж без него. Пуснете я няколко пъти и сравнете времената за изпълнение.

2.8 Извеждане на специални символи

Задача 2.11. Отпечатайте на екрана символите на емотиконка на пингвина. Ако не ги знаете те са - <(").

Когато отпечатваме низ, той е заобиколен от кавички. Ако в самия низ има кавички, трябва да може да ги различим от тези обграждащи низа. Това става като сложим обратна наклонена черта преди кавичките, които искаме да отпечатаме - \". Когато използваме апостроф за символи имаме подобен проблем, ако символа е апостроф. Решението е подобно - \'. И другият специален символ, за който трябва и в двата случая да се внимава е самата обратна наклонена черта. За да отпечатаме една такава, трябва да сложим две - \\.

```
#include <iostream>
using namespace std;

int main() {
  cout << "<(\")" << endl;

  return 0;
}
```

2.9 Форматиране на дробни числа

Задача 2.12. Колко знака след точката ще изведе *cout « 1.0/4* и колко ще изведе *cout « 1.0/3*?

Когато извеждаме дробни числа в конзолата има две важни неща:
Първото е колко най-много символа след точката искаме да се извеждат. То се указва със *cout « setprecision(n)*, където n е максималният брой символи след точката. Второто нещо е свързано с това колко точно символа се извеждат. По подразбиране, ако числото се събира в по-малко цифри след точката, ще го изведем без да използваме всички разрешени позиции. Например максималният брой цифри е 6 и искаме да изведем 1.25, то ще изведем 1.25. Другият вариант, е ако искаме числото да се изведе с точно толкова позиции, колкото сме задали в *setprecision*. Това става със *cout « fixed*. В примера с 6 позиции и числото 1.25 ще се изведе 1.250000.

Задача 2.13. Дадени са две положителни цели числа. Отпечатайте тяхното частно с точност до два знака след точката.

Започваме със *cout « a/b « endl*. Тук проблемът е, че имаме целочислено деление и резултатът съответно е цяло число. За да получим дробен резултат може да продължим с умножение с дробно число *cout « 1.0*a/b « endl*. Сега обаче не е ясно какво точно ще се изпише и колко знака след точката ще имаме. Понеже имаме фиксиран брой знаци и този брой е равен на 2 трябва да използваме е *cout « fixed « setprecision(2) « 1.0*a/b « endl*.

2.10 Задачи

Задача 2.14 (част от Пролетен Турнир 2017, Е група, Животинска аритметика)**.** Даден е един ред на който има малки латински букви и интервали в някакъв ред. Изпишете първата, третата и т.н букви.

вход	изход
pa	p
biso	bs
p a pa	pp

Задача 2.15 (Пролетен Турнир 2017, Е група, Избори)**.**

Задача 2.16 (част от Пролетен Турнир 2016, Е група, Почивни дни)**.** На един ред е дадена дата във формат ден.месец.година. Дните и месеците могат да съдържат водеща нула, но това не е задължително. На един разделени с интервал изпишете същата дата във формат месец.ден.година с и без водещи нули.

вход	изход
10.06.2016	10.06.2016 10.6.2016
20.6.2016	20.06.2016 20.6.2016
1.3.2020	01.03.2020 1.3.2020

Глава 3

Вектор

3.1 Дефиниция

Дефиниция 3.1 (Вектор)**.** Вектор е специален тип данни, които позволява да работим с много променливи от един и същи тип, подобно на масив. Основната разлика е, че докато при масива броя елементи е фиксиран, то при вектора, той може да се променя - може да добавяме и махаме елементи. Вектора е един вид динамичен масив.

За да използваме вектор трябва да добавим неговата библиотека:

```
#include<vector>
```

Разбира се, ако вече сме добавили библиотека, която го включва, например `bits/stdc++.h`, това не е необходимо.

3.2 Създаване

3.2.1 С начален брой елементи

Може да декларираме вектор подобно на масив. За целта трябва да зададем тип данни, име на вектора и начален брой елементи.

```
vector<тип> име(начален_брой_елементи);
```

Без значение дали вектора е глобален или локален, неговите елементи винаги приемат начални стойности. За целочислени типове това е 0, за char - символ с код 0, за bool - false и за string - празен низ.

Задача 3.1. Как ще декларираме вектор, които е подобен на следния масив: `int a[1000]`.

Решение.

```
vector<int> a(1000);
```

3.2.2 С начални стойности

Може да създадем вектор с различи начални стойности от тези по подразбиране. Така всички елементи в началото ще са равни на избраната от нас стойност. След броя елементи със запетая, добавяме началната стойност.

```
vector<тип> име(начален_брой_елементи, начална_стойност);
```

Задача 3.2. Как ще декларираме вектор с 1000 символа, които в началото са буквата a.

Решение.

```
vector<char> a(1000, 'a');
```

3.2.3 Празен вектор

Понеже във вектора може да се добавят и махат елементи, може да създадем празен вектор - без елементи.

```
vector<тип> име;
```

3.3 Достъп до елементите

3.3.1 Брой елементи

Брой на елементите, които в момента са във вектор, може да се достъпи чрез функцията `size`:

```
v.size();  // броя на елементите във вектора v
```

3.3.2 Елементи

Във всеки момент може да използваме всеки от елементите по същия начин както в масив. Всеки елемент има позиция от 0 до `v.size()-1`. Така самите елементи са: `v[0], v[1], ..., v[v.size()-1]`.

Задача 3.3. Създали сме вектор с 10 елемента - `vector<int> v(10)`. Как ще въведем стойностите от конзолата?

Решение. Самите елементи се използват като елементите на масив. Така, че тази задача има същото решение като за масив.

```
for (int i = 0; i < 10; i++) {
    cin >> v[i];
}
```

3.3.3 Последен елемент

Понеже често се налага да използваме последния елемент, то за него има още един начин за използване. Използването на `v.back()` е еквивалентно на `v[v.size()-1]`.

3.4 Добавяне и махане в края

От всички операции свързани с добавяне и махане, най-бързите и най-често използваните са тези за добавяне и махане на елементи в неговия край.

3.4.1 Добавяне в края

За да добавим елемент в края на вектор, ще използваме вградената функция `push_back`. Така добавянето в края става чрез нея:

```
v.push_back(елемент); // добавя подадения елемент след последния
```

При изпълнението на тази команда броя на елементите във вектора се увеличава с един и стойността на последния става равна на това, което сме подали в скобите.

Задача 3.4. Създали сме празев вектор - `vector<int> v`. Как ще въведем стойностите на 10 елемента от конзолата?

Решение. Всеки елемент може да го добавяме в края:

```
for (int i = 0; i < 10; i++) {
    int x; cin >> x;
    v.push_back(x);
}
```

3.4.2 Махане на последния елемент

За да премахнем последния елемент, ще използваме вградената функция `pop_back`. Така махането от края става чрез нея:

```
v.pop_back(елемент); // премахва последния елемент
```

При изпълнението на тази команда броя на елементите във вектора намалява с един.

Задача 3.5. Даден е вектор с елементи - `vector<int> v`, който е запълнен със стойности. Искаме да махнем всички елементи отзад, така че последния елемент да стане не по-голям от x.

Решение.

```
while (v.size() > 0 && v.back() > x) {
    v.pop_back();
}
```

3.5 Сортиране

Ако искаме да подредим елементите в един вектор по големина може да използваме вградената функция за сортиране. Така елементите ще се наредят в нарастващ ред, спрямо стандартната им наредба. За целта, ако имаме `vector<тип> v`, трябва да извикваме следната функция:

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

Глава 4

Стринг

4.1 Дефиниция

Дефиниция 4.1 (Стринг)**.** Стринг(`string`) е специален тип данни за работа с низове. Той е подобен на `vector<char>`, но добавя допълнителни функции, които са полезни за работа.

За да използваме стринг трябва да добавим неговата библиотека:

```
#include<string>
```

Разбира се, ако вече сме добавили библиотека, която го включва, например `bits/stdc++.h`, това не е необходимо.

4.1.1 Създаване

Имаме няколко възможности за създаване:

```
string име;
string име(начален_низ);
string име(начален_брой_елементи, начален_символ);
```

4.1.2 Вход и изход

Един стринг s може да го четем или пишем в конзолата по стандартния начин.

```
cin >> s;
cout << s;
```

Трябва да внимаваме, че при `cin`, не може да четем низ, в който има интервали. За целта трябва да използваме `getline`.

4.1.3 Присвояване

Ако присвояваме един низ на друг, то първоначалния се копира символ по символ. Трябва да знаем, че това не е бърза операция за дълги низове.

```
s = another_string;
```

4.1.4 Събиране

Може да събираме низове - резултата е нов низ в който двата низа са долепени един до друг. Низовете не познават числа, дори да имаме два низа като числа, резултата пак ще е долепване един до друг.

```
s = стринг + стринг;
```

4.2 Основни вградени функции

Нека имаме `string s`. Ще разгледаме основните функции, които може да използваме. Относно тяхната бързина основното правило е следното: бавни са тези операции, при които се добавят или махат елементи, които не са в края на низа. Помислете например как бихме премахнали първия елемент от масив - трябва да преместим всички останали елементи с една позиция напред. И съответно какво ще ни коства да премахнем последния елемент от масив.

4.2.1 Вградени функции, които работят бързо

Това са стандартните функции, които често ще използваме без много да се замисляме за бързината.

```
s.size() // връща броя символи в низа
s.push_back(символ) // добавя символа в края на низа
s.pop_back() // премахва последния символ от низа
s.append(стринг) // добавя подадения стринг в края на низа
```

4.2.2 Вградени функции, които работят бавно

Това са функции, които може да работят бавно. Ще ги използваме само в случай, че не може да измислим нещо по-добро.

```
s.insert(позиция, стринг) // вмъква стринга на дадената позиция
s.erase(позиция) // премахва всички символи от дадената позиция до края
s.erase(позиция, брой_символи) // премахва дадения брой символи от дадената позиция
```

4.3 Стринг във функция

При използването на стринг във функция имам няколко варианта.

4.3.1 Глобални променливи

Ако стринговете, които ни трябват са глобални променливи, то няма нужда да ги подаваме като параметри.

4.3.2 Параметър с копиране

Ако функцията приема параметър по следния начин: `f(string s)`, то при извикването на функцията се създава нов низ и оригиналния се копира в новия. Така промяната на низа във функцията няма да промени оригиналния. Обаче цената на копирането може да е висока, ако оригиналния низ има много символи. Ще предаваме низове по този начин само в крайна необходимост.

4.3.3 Параметър с псевдоним

Ако функцията приема параметър по следния начин: `f(string& s)`, то във функцията се подава оригиналния низ. Ако променим s във функцията, ще променим и оригинални стринг. Понеже преизползваме низа, то не се налага той да се създава наново и съответно нямаме забавянето, както при копирането.

4.3.4 Параметър с псевдоним, който не се променя

Ако функцията не променя оригиналния стринг, то задължително подаваме параметъра по следния начин: `f(const string& s)`. Основната разлика с предния вариант е, че думата `const` ни защитава. Ако функцията не трябва да променя s, но по погрешка опитаме да го направим, програмата няма да се компилира.

4.4 Число към низ и низ към число

Задача 4.1. Дадено е число n. Напишете функция, която го превръща в низ.

Решение. За целта трябва да преминем през цифрите на числото една по една и да ги добавяме към резултатния низ. Обхождането на цифрите от ляво на дясно е трудно, за това ще го правим отдясно наляво както сме свикнали. Сега понеже обхождаме наобратно може да добавяме цифрите в началото на низа. Това обаче е много бавна операция и ще да я избегнем. За целта ще изберем друг подход - ще добавяме в края на низа и преди да върнем резултат ще обърнем низа наобратно.

```
string toString(int n) {
    if (n == 0) return "0";
    string s;
    while (n > 0) {
        s.push_back(n%10+'0');
        n /= 10;
    }
    reverse(s.begin(), s.end());
    return s;
}
```

Задача 4.2. Дадено е число n и символ c, който е цифра. Напишете функция, която добавя символа в края на числото.

Решение. Тази задача се решава на две стъпки. Първо искаме да добавим 0 като последна цифра на n. Това става като умножим n по десет. Сега трябва да сменим нулата с цифрата c. Понеже това е последната цифра на числото е достатъчно да добавим $c -' 0'$.

```
int append(int n, char c) {
    return 10*n+c-'0';
}
```

Задача 4.3. Даден е стринг s, съставен от цифри. Напишете функция, която го превръща в число.

Решение. Таза задача има две решения. По-стандартното е да умножим цифрата на единиците по едно, тази на десетиците по десет и т.н. и да съберем получените числа. Така $\overline{abc} = 1c + 10b + 100a$. За целта ни трябва една допълнителна променлива да пазим текущата степен на десет. Това решение е по-стандартно, но ние няма да го показваме, както и използваме. По-доброто решение на тази задача е да използваме идеята за добавяне на цифра отзад на число. Така може постепенно да добавяме първата, втората и т.н. към края на текущото число. Така $\overline{a} = 10.0 + a$, $\overline{ab} = 10\overline{a} + b = 10.(10.0 + a) + b$ и $\overline{abc} = 10\overline{ab} + c = 10(10.(10.0 + a) + b) + c$. Така във всеки момент, за да добавим нова цифра отдясно умножаваме текущия резултат по 10 и добавяме цифрата.

```
int toInt(const string& s) {
    int n = 0;
    for (int i = 0; i < s.size(); i++) {
        n = 10*n+s[i]-'0';
    }
    return n;
}
```

4.5 Поднизове

Задача 4.4. Дадени е низа s. Премахнете първите p символа на s.

Решение. Един вариант е за премахването да викаме функцията `erase`. Но махането е ефикасно само за последния елемент. Във всеки друг случай, то е бавно и ще се опитваме да го избягваме. Втори вариант е да създадем нов низ, които включва елементите след първите p. Това и ще направим:

```
string removeFirstElements(const string& s) {
    string t;
    for (int i = p; i < s.size(); i++) {
        t.push_back(s[i]);
    }
    return t;
}
```

Задача 4.5. Дадени са два низа s и t. Намерете най-дългия низ, с които започват едновременно и двата низа.

Решение. Обхождаме отляво надясно и докато съвпадат съответните символи добавяме към резултата. Важно е да внимаваме, да не излезем от границите на низовете.

```
string maxCommonPrefix(const string& s, const string& t) {
    string w;
    for (int i = 0; i < min(s.size(), t.size()); i++) {
        if (s[i] != t[i]) break;
        w.push_back(s[i]);
    }
    return w;
}
```

Задача 4.6. Дадени са два низа s и t. След премахването на първите p символа от s, намерете най-дългия низ, с които започват едновременно и двата низа.

Решение. Един вариант е да създадем нов низ, слез премахване на първите p елемента. Обикновено вместо модифицирането на низа е по-добре да работим с индексите, като при обхожданията внимаваме за кой низ кой индекс трябва да гледаме. Да пробваме да гледаме индексите. След премахването на p елемента от s ефективно първи ще стане елемента $s[p]$. Така трябва да сравняваме $s[p]$ с $t[0]$, $s[p+1]$ с $t[1]$, и т.н.

```
string maxPrefix(const string& s, const string& t, int p) {
    string ans;
    for (int i = 0; p+i < s.size() && i < t.size(); i++) {
        if (s[p+i] != t[i]) break;
        ans.push_back(t[i]);
    }
    return ans;
}
```

Задача 4.7. Дадени са два низа s и t. Проверете дали t се съдържа като последователност в s.

Решение. Трябва да проверим за всеки индекс от s дали започва последователност, която съвпада с t. Ще използваме помощна функция, на която ще подаваме позиция от s и ще проверяваме дали с начало тази позиция има съвпадение с t. В началото ще проверяваме дали от началната позиция на s има поне $t.size()$ символа. Ако започваме от позиция p, то последната позиция от s ще бъде $p + t.size() - 1$.

```
bool hasMatch(const string& s, const string& t, int p) {
    if (p+t.size()-1 >= s.size()) return false;
    for (int i = 0; i < t.size(); i++) {
        if (s[p+i] != t[i]) return false;
    }
    return true;
}
```

Задача 4.8. Дадени са два низа s и t. Поставяме s над t и ги подравняваме отляво. След това отместваме t с p позиции надясно. Намерете на колко места имаме символи един под друг, които са еднакви.

Решение. Трябва да нагласим индексите. Ако i е позиция на символ в s, то позицията в t след изместването ще бъде $i - p$. Един вариант е да обходим всички позиции в s - $0 \leq i < s.size()$ и да проверяваме дали имаме валидна позиция в t - $0 \leq i-p < t.size()$. Друг вариант е да пресметнем предварително най-малката и най-голямата възможна стойност на i, които изпълняват и двете условия. Ще напишем второто като $p \leq i < t.size()+p$. И така получаваме $p \leq i < min(s.size(), t.size()+p)$. Тази сметка може да ни спести малко итерации.

```
int calcCommon(const string& s, const string& t, int p) {
    int ans = 0;
    for (int i = p; i < min(s.size(), t.size()+p); i++) {
        if (s[i] == t[p-i]) {
            ans++;
        }
    }
    return ans;
}
```

Задача 4.9. Дадени са два низа s и t. Може ли да изтрием произволен брой символи от s, така че да получим t.

Решение.

```
bool canErase(const string& s, const string& t) {
    int tIndex = 0;
    for (int i = 0; i < s.size(); i++) {
        if (s[i] == t[tIndex]) {
            tIndex++;
            if (tIndex == t.size()) return true;
        }
    }
    return false;
}
```

4.6 Палиндроми

Дефиниция 4.2 (Палиндром)**.** Палиндром е низ, които е огледален - чете се еднакво от ляво надясно и отдясно наляво. Например думите "потоп"и "капак"са палиндроми.

Задача 4.10. Проверете дали даден низ s е палиндром.

Решение. Обхождаме първата половина от символите и за всеки символ i сравняваме с огледалния му - $s.size() - i - 1$.

```
bool isPalindrome(const string& s) {
    for (int i = 0; i < s.size()/2; i++) {
        if (s[i] != s[s.size()-i-1]) return false;
    }
    return true;
}
```

В предното решение като знаем позицията i, трябваше да сметнем коя е огледалната му. Въпреки, че в случая това не е проблем, понякога е неприятно да правим тези сметки. Това много лесно може да се избегне ако пазим отделни променливи за лявата и дясната позиция. Знаем, че 0 и $s.size()-1$ са огледални и може да започнем с тях. На всяка стъпка лявата позиция ще се увеличава с едно, а дясната ще намалява. Това има смисъл докато не се разминат двете променливи. Ето как би изглеждало решението без връзката между позициите.

```
bool isPalindrome(const string& s) {
    for (int l = 0, r = s.size()-1; l < r; l++, r--) {
        if (s[l] != s[r]) return false;
    }
    return true;
}
```

Задача 4.11. Проверете дали даден низ s е палиндром, след като се премахнат интервалите от него.

Решение. Най-лесното решение е да създадем нов низ, в който няма интервали. Това може да стане като обходим символите на оригиналния низ и всеки, който не е интервал го добавяме към резултата. Разбира се по-добро решение ще е да работим само с индексите. Това е по-сложна задачка, но си струва да и се обърне внимание. Сега ако имаме един символ не е ясно къде е огледалния му заради интервалите. За това се налага да използваме отделни променливи за позициите на текущия символ и неговия огледален. Може да стартираме с решението на предната задача с двете позиции. Единствено сега трябва ако стъпим на интервал да го прескочим. Така в началото на цикъла ще проверяваме отделно за лявата и дясната позиция - докато са в границите на низа и са интервали ще ги променяме. Понеже след това пак може да се разминат отново ще проверяваме, че не са.

```
bool isPalindrome(const string& s) {
    for (int l = 0, r = s.size()-1; l < r; l++, r--) {
        while (l < s.size() && s[l] == ' ') {
            l++;
        }
        while (r >= 0 && s[r] == ' ') {
            r--;
        }
        if (l >= r) break;
        if (s[l] != s[r]) return false;
    }
    return true;
}
```

Задача 4.12. Даден е низ s. Може ли с добавяне на точно p символа вдясно да получим палиндром.

Решение. Резултатния низ ще има $s.size() + p$ символа. Сега за всеки символ от първата половина $s[i]$ ще искаме да е равен на $s[s.size() + p - i - 1]$. Разбира се ако някои от двата символа излиза от границите на s, значи ние го избираме и може да смятаме, че ще е същия като огледалния. Така трябва да обходим всяко $i : 0 \leq i < (s.size() + p)/2$ и да сравняваме $s[i]$ с $s[s.size() + p - i - 1]$, като това има смисъл когато и двете позиции са част от първоначалния низ. Може да проверяваме дали това е изпълнени вътре в цикъла, но може да сме умни и да подобрим условията на самия цикъл. Искаме $i < s.size()$ и $s.size() + p - i - 1 < s.size()$. Последното е еквивалентно на $p - 1 < i$, което е $p \leq i$. Така стигаме до следния код:

```
bool canMakePalindrome(const string& s, int p) {
    for (int i = p; i < min(s.size(), (s.size()+p)/2); i++) {
        if (s[i] != s[s.size()+p-i-1]) return false;
    }
    return true;
}
```

4.7 Задачи

Задача 4.13 (Есенен Турнир 2014, D група, Подниз).

Задача 4.14 (Есенен Турнир 2011, D група, Симетрична редица).

Задача 4.15 (НОИ, Общински кръг 2016, D група, Уравнение).

Задача 4.16 (Есенен Турнир 2019, D група, Отгатни цифрата).

Задача 4.17 (Зимен Турнир 2017, D група, Подобни думи).

Задача 4.18 (НОИ, Областен кръг 2015, C група, Пакети от думи).

Глава 5

Разделяне на последователности

5.1 Дефиниция

Дефиниция 5.1 (Последователност)**.** Разглеждаме редицата - $a_0, a_1, ..., a_{n-1}$. Всяка група от няколко съседни елемента(един, няколко или всички) наричаме последователност. Една последователност най-често се определя от позицията на първия и последния елемент в редицата. Така двойката (i, j) определя последователността $a_i, a_{i+1}, ..., a_j$. Друг стандартен вариант за определяне е позицията на първия елемент и броя елементи в последователността.

Задача 5.1. Кои са всички последователности на редицата $3, 5, 6, 1$?

Решение. За изброяване на последователностите ще разгледаме два подхода. Първият и по-стандартен е да разгледаме всички последователности с начало първия елемент, след това всички с начало втория елемент и т.н. Вторият вариант е да разгледаме всички с дължина едно, след това всички с дължина две и т.н.
Да разгледаме първия подход. Започваме с последователностите с първия елемент - $\{3\}, \{3, 5\}, \{3, 5, 6\}, \{3, 5, 6, 1\}$. След това тези с начало втория - $\{5\}, \{5, 6\}, \{5, 6, 1\}$. Тези с начало третия - $\{6\}, \{6, 1\}$ и накрая тази с начало последния - $\{1\}$.
При втория подход започваме с четири последователности с по един елемент - $\{3\}, \{5\}, \{6\}, \{1\}$. Тези с по два елемента са 3 - $\{3, 5\}, \{5, 6\}, \{6, 1\}$. Две с по три елемента - $\{3, 5, 6\}, \{5, 6, 1\}$ и една с четири елемента - $\{3, 5, 6, 1\}$.

Задача 5.2. Колко на брой са всички последователности на редица с n елемента?

Решение. За да ги преброим може да използваме всеки от двата подхода в предишната задача. Ние ще покажем решението чрез първия подход. С начало първия елемент имаме n последователности - всеки елемент е край на последователност с първия. При начало втория елемент само първия не може да е край - имаме $n - 1$ варианта. И т.н с всеки следващ елемент вариантите намаляват един по един, докато стигнем до един вариант с начало последния елемент. Така общия брой варианти е $n + (n - 1) + (n - 2) + ... + 1 = n(n + 1)/2$.

Дефиниция 5.2 (Разделяне на последователности)**.** Разделянето на една редица на една или повече последователности, при което всеки елемент участва в точно една последователност наричаме разделяне на последователности. Друг начин да си го представим е слагането на прегради - между някои съседни елементи слагаме прегради, така елементите преди първата преграда са първата последователност, тези между първата и втората преграда - втората последователност и т.н.

Задача 5.3. Разглеждаме редицата $3, 5, 5, 3, 1, 1, 1$. Разделете я на възможно най-малко последователности, така че всички числа от една последователност да са равни.

Решение. Може да разделим на четири последователностите - $3|5, 5|3|1, 1, 1$.

Задача 5.4. Разглеждаме редицата $3, 5, 4, 1, 2, 3$. Разделете я на възможно най-малко последователности, така че всички числа от една последователност да са в нарастващ ред.

Решение. Може да разделим на най-малко три последователностите - $3, 5|4|1, 2, 3$.

5.2 Определяне на преградите

Една преграда може да я определим от двойка съседни елементи. В тази секция ще разгледаме как проверяваме дали два съседни елемента са част от една последователност или между тях има преграда. Когато проверяваме дали между два елемента има прегради е удобно да използваме функция. В тази глава ще пишем функцията `bool shouldSplit(int i, int j)`, която като параметри приема индексите на два елемента и връща като резултат дали има преграда между тях. Реално винаги ще имаме, че $i + 1$ е равно на j, но въпреки това за удобство, ще подаваме и двете променливи.

Задача 5.5. Разглеждаме редицата $a_0, ..., a_{n-1}$. Разделяме я на възможно най-малко последователности, така че всички числа от една последователност да са равни. Намерете между кои двойки индекси ще има прегради.

Решение. Разглеждаме всички съседни двойки индекси, това са - $\{0, 1\}, \{1, 2\}, ..., \{n-2, n-1\}$. Най-удобно за целта е цикъл, който обхожда възможностите за втория индекс в двойката - $1, 2, ..., n-1$. Сега остава за две числа да проверим дали между тях има преграда. Ако те са равни ще бъдат част от една последователност. Значи преграда между ще има само ако са различни.

```
// проверява дали между елементите a[i] и a[j] има преграда
bool shouldSplit(int i, int j) {
    return a[i] != a[j];
}

// печата двойките съседни индекси, между които има прегради
void printSplits() {
    for (int i = 1; i < n; i++) {
        if (shouldSplit(i-1, i)) {
            cout << "{" << i-1 << "," << i << "}" << endl;
        }
    }
}
```

Задача 5.6. Разглеждаме редицата $a_0, ..., a_{n-1}$. Разделяме я на възможно най-малко последователности, така че всички числа от една последователност да са в строго нарастващ ред. Намерете между кои двойки индекси ще има прегради.

Решение. Две числа са в една последователност ако първото е по-малко от второто. Така функцията за преграда ще проверява обратното.

```
bool shouldSplit(int i, int j) {
    return a[i] >= a[j];
}
```

Ще отбележим, че ако е по-лесно да проверим дали двата елемента са от една последователност може да го направим и да връщаме отрицанието:

```
bool shouldSplit(int i, int j) {
    bool inSameSplit = a[i] < a[j];
    return !inSameSplit;
}
```

Задача 5.7. Разглеждаме редицата $a_0, ..., a_{n-1}$, съставено от нули и единици. Разделяме я на последователности от вида 0...01, нули и накрая единица. Намерете между кои двойки индекси ще има прегради.

Решение. В този случай не са необходими два елемента за да намерим край на последователност. Всяка единица е край на такава. В случая между две числа има преграда ако първото е единица.

```
bool shouldSplit(int i, int j) {
    return a[i] == 1;
}
```

Задача 5.8. Разглеждаме редицата $a_0, ..., a_{n-1}$, където всеки елемент е буква или цифра. Разделяме я на възможно най-малко последователности, така че всички елементи от една последователност да са или само букви, или само цифри. Намерете между кои двойки индекси ще има прегради.

Решение. Два елемента са в различна последователност, когато единия е цифра, другия - буква. Или казано по-друг начин - единия е цифра, другия не е цифра. Това и ще проверим.

```
bool shouldSplit(int i, int j) {
    return isdigit(a[i]) != isdigit(a[j]);
}
```

Задача 5.9. Разглеждаме n кутии с размери (a_i, b_i, c_i). Разделяме ги на възможно най-малко последователности, така че всяка кутия от една последователност се побира в предишната от тази последователност, ако има такава. Намерете между кои двойки индекси ще има прегради.

Решение. Две кутии са в една последователност ако втората влиза в първата. За такъв тип проверки, най-добре да въведем наредба в трите числа за една кутия. Или още при четенето на входа, или във функцията за проверка трябва да подсигурим, че $a_i \leq b_i \leq c_i$. При тази наредба проверката дали едната кутия влиза в другата е лесна. Проверяваме дали всяко от трите числа за първата кутия е по-голямо или равно на съответното такова за втората. Кода за наредбата оставяме на вас:

```
bool shouldSplit(int i, int j) {
    // подсигуряваме, че a[i] <= b[i] <= c[i]
    // подсигуряваме, че a[j] <= b[j] <= c[j]

    bool inSameSplit = (a[i] >= a[j]) && (b[i] >= b[j]) && (c[i] >= c[j]);
    return !inSameSplit;
}
```

5.3 Определяне на начало или край

Задача 5.10. Разглеждаме редицата $a_0, ..., a_{n-1}$. Разделяме я на възможно най-малко последователности, така че всички числа от една последователност да са равни. Намерете всички индекси, които са начало на последователности.

Решение. Един индекс i е начало на последователност, ако преди него завършва последователност. Т.е. между елементите с индекси $i-1$ и i има преграда. За всяко i ще проверяваме дали `shouldSplit(i-1,i)` връща `true`. Обаче има един важен момент - първият елемент, този с индекс 0 винаги е начало на последователност и за него е изключително важно да не викаме проверката с предишния понеже при $i = 0$, $i - 1$ ще излезе от границите на масива. Така първият елемент винаги е начало на последователност, а за останалите проверяваме функцията `shouldSplit(i-1,i)`.

```
void printBeginnings() {
    cout << 0 << endl;
    for (int i = 1; i < n; i++) {
        if (shouldSplit(i-1, i)) {
            cout << i << endl;
        }
    }
}
```

В това решение имаме `cout` на две отделни места. Обикновено в задачите трябва да правим нещо по-сложно с началата и това да го правим на две отделни места е лоша практика. Отделно това решение няма да работи за празна редица, но да кажем, че това е доста частен случай. Ще променим програмата, така че да има `cout` на едно място.
Очевидно е, че писането в цикъла трябва да остане. Варианта е да преместим първия `cout` в цикъла. Така i ще трябва да започва от 0 и в `if`-а да добавим една допълнителна проверка дали i е 0. Задължително първо трябва да проверяваме i и после `shouldSplit`, понеже в противен случай пак ще имаме проблема с излизането от масива:

```
void printBeginnings() {
    for (int i = 0; i < n; i++) {
        if (i == 0 || shouldSplit(i-1, i)) {
            cout << i << endl;
        }
    }
}
```

Задача 5.11. Разглеждаме редицата $a_0, ..., a_{n-1}$. Разделяме я на възможно най-малко последователности, така че всички числа от една последователност да са равни. Намерете всички индекси, които са краища на последователности.

Решение. Решението е доста сходно на предното. За всеки елемент i проверяваме дали между i и $i+1$ има преграда, като сега трябва да внимаваме да не излезем от масива при проверката за последния елемент. Ще използваме същия трик:

```
void printEndings() {
    for (int i = 0; i < n; i++) {
        if (i == n-1 || shouldSplit(i, i+1)) {
            cout << i << endl;
        }
    }
}
```

Задача 5.12. Разглеждаме редицата $a_0, ..., a_{n-1}$. Разделяме я на възможно най-малко последователности, така че всички числа от една последователност да са равни. Намерете броя на тези последователности.

Решение. За да преброим последователностите е достатъчно да преброим началата или краищата. Ще използваме, че броя последователности е равен на броя начала.

```
int countSplits() {
    int ans = 0;
    for (int i = 0; i < n; i++) {
        if (i == 0 || shouldSplit(i-1, i)) {
            ans++;
        }
    }
    return ans;
}
```

В тази задача, ако редицата със сигурност има поне един елемент, решението което гледа $i = 0$ извън цикъла е по-удобно. Може да променим ans да е 1 в началото и цикъла, да започва от 1.

5.4 Определяне на начало и край

Задача 5.13. Разглеждаме редицата $a_0, ..., a_{n-1}$. Разделяме я на възможно най-малко последователности, така че всички числа от една последователност да са равни. Намерете началото и края на всяка последователност.

Решение. Тази задача е основна за темата. Ще разгледаме няколко решения, като последното смятаме за най-добро. Първият вариант е най-близък до това което правихме. Видяхме как се проверява за всеки индекс дали е начало или край. Може да пазим два `vector`-а за началата и краищата, да обходим всички елементи и когато се налага до добавяме във векторите. За проверка за начало и край ще използваме постигнатото в предишната секция:

```
void printSplits() {
    vector<int> beginnings;
    vector<int> endings;
    for (int i = 0; i < n; i++) {
        if (i == 0 || shouldSplit(i-1, i)) {
            beginnings.push_back(i);
        }
        if (i == n-1 || shouldSplit(i, i+1)) {
            endings.push_back(i);
        }
    }

    for (int i = 0; i < beginnings.size(); i++) {
        cout << beginnings[i] << " " << endings[i] << endl;
    }
}
```

За втория вариант ще използваме, че няма нужда от началата и краищата. Достатъчно е да имаме едно от двете. Нека например запазим само краищата на последователности. Ако това са например индексите - $3, 4, 6$, то е ясно че самите последователности ще бъдат - $(0, 3), (4, 4), (5, 6)$. Ако пак пазим краищата в `endings`, то два съседни края `endings[i-1]` и `endings[i]` дават последователността - `(endings[i-1]+1, endings[i])`. Ако разгледаме отново примера - $3, 4, 6$, то първите два елемента определят последователността $(4, 4)$, последните два - $(5, 6)$. Първата последователност липсва. За да не гледаме като частен случай ще добавим елемент в началото на вектора `endings`, така че този елемент и първия край да дават първата последователност. Ако елемента, който добавяме е x, то първата последователност ще има начало $x + 1$, за това искаме $x + 1$ да е 0, т.е. $x = -1$.

```
void printSplits() {
    vector<int> endings;
    endings.push_back(-1);
    for (int i = 0; i < n; i++) {
        if (i == n-1 || shouldSplit(i, i+1)) {
            endings.push_back(i);
        }
    }
```

```
    for (int i = 1; i < endings.size(); i++) {
        cout << endings[i-1]+1 << " " << endings[i] << endl;
    }
}
```

За следващия вариант ще извеждаме последователност, в момента в който намерим край на такава. За целта ще стартираме с кода за намиране на краища. Когато намерим край, това което липсва е началото. За целта в една променлива ще пазим началото на текущата последователност, която ще я променяме всеки път когато видим ново начало.

```
void printSplits() {
    int lastBeginning;
    for (int i = 0; i < n; i++) {
        if (i == 0 || shouldSplit(i-1, i)) {
            lastBeginning = i;
        }
        if (i == n-1 || shouldSplit(i, i+1)) {
            cout << lastBeginning << " " << i << endl;
        }
    }
}
```

Като за последно ще подобрим малко този вариант. Ще отбележим, че първото начало е с индекс 0. При намиране на край i, знаем че началото на следващата последователност ще е $i + 1$, така че може да променяме `lastBeginning`, когато намерим край.

```
void printSplits() {
    int lastBeginning = 0;
    for (int i = 0; i < n; i++) {
        if (i == n-1 || shouldSplit(i, i+1)) {
            cout << lastBeginning << " " << i << endl;

            lastBeginning = i+1;
        }
    }
}
```

Този код е доста универсален и може да решава голяма част от задачите, в които по един или друг начин искаме да сложим прегради между елементите. В момента на `cout`-а имаме началото и края на поредната последователно и може да го заменим с това което ще ни трябва - намиране на дължината, намиране на сумата на елементите и т.н.

Задача 5.14. Разглеждаме редицата $a_0, ..., a_{n-1}$. За дадено число x разделяме редицата на последователности със сума x. Дадено е, че такова разделяне съществува. Намерете началото и края на всяка последователност.

Решение. При тази задача не може да определим дали има преграда като гледаме два съседни елемента. Трябва да пазим сумата от началото на последователността и в момента, в който достигнем сума x това означава, че сме стигнали край. Освен начало на последователността, ще пазим и сумата до момента. Всеки път към тази сума добавяме текущия елемент. Когато достигнем край нулираме сумата.

```
void printSplits() {
    int lastBeginning = 0;
    int currentSum = 0;
    for (int i = 0; i < n; i++) {
        currentSum += a[i];
        if (currentSum == x) {
            cout << lastBeginning << " " << i << endl;

            lastBeginning = i+1;
            currentSum = 0;
        }
    }
}
```

Задача 5.15. Разглеждаме редицата $a_0, ..., a_{n-1}$. За дадено число x разделяме редицата на най-малко последователности, така че сумата на всяка последователност е не по-голяма от x. Дадено е че такова разделяне съществува. Намерете началото и края на всяка последователност.

Решение. От една страна трябва да пазим сумата до момента, от друга трябва да проверим дали може да добавим следващия елемент към текущата последователност. Между два елемента ще има преграда ако `currentSum+a[j] > x`. Въпреки, че не е нужно отново ще напишем функция `shouldSplit`, като допълнително ще подаваме текущата сума.

```
bool shouldSplit(int i, int j, int currentSum) {
    return currentSum+a[j] > x;
}

void printSplits() {
    int lastBeginning = 0;
    int currentSum = 0;
    for (int i = 0; i < n; i++) {
        currentSum += a[i];
        if (i == n-1 || shouldSplit(i, j, currentSum)) {
            cout << lastBeginning << " " << i << endl;

            lastBeginning = i+1;
            currentSum = 0;
        }
    }
}
```

Задача 5.16. Разглеждаме редицата $a_0, ..., a_{n-1}$. За дадено число x разделяме редицата на най-малко последователности, така че разликата между всеки два елемента в една последователност е не по-голяма от x. Намерете началото и края на всяка последователност.

Решение. Най-голямата разлика ще бъде тази между най-големия и най-малкия елемент. Сега за всяко последователност трябва да пазим тези два елемента. И отделно трябва да проверим дали с добавянето на нов елемент разликата няма да стане голяма. За проверката може при добавянето на елемент да смятаме новите минимум и максимум или да разгледаме случаите дали новия елемент е по-малък от минимума или по-голям от максимума.

```
bool shouldSplit(int i, int j, int minimum, int maximum) {
    int newMinimum = min(minimum, a[j]);
    int newMaximum = max(maximum, a[j]);
    return newMaximum-newMinimum > x;
}

void printSplits() {
    int lastBeginning = 0;
    int minimum = 1000;
    int maximum = 0;
    for (int i = 0; i < n; i++) {
        minimum = min(minimum, a[i]);
        maximum = max(maximum, a[i]);
        if (i == n-1 || shouldSplit(i, j, minimum, maximum)) {
            cout << lastBeginning << " " << i << endl;

            lastBeginning = i+1;
            minimum = 1000;
            maximum = 0;
        }
    }
}
```

Задача 5.17. Разглеждаме редицата $a_0, ..., a_{n-1}$. Намерете дължината на най-дългата последователност от равни числа.

Решение. Ако имаме последователност с начало и край (i, j), то броя елементи е равен на $j - i + 1$. Остава да минем през всички последователности с равни числа и да намерим най-голямото $j - i + 1$.

```
int findMaxLength() {
    int ans = 0;
    int lastBeginning = 0;
    for (int i = 0; i < n; i++) {
        if (i == n-1 || shouldSplit(i, i+1)) {
```

```
            ans = max(ans, i-lastBeginning+1);

            lastBeginning = i+1;
        }
    }
    return ans;
}
```

Задача 5.18. Разглеждаме редицата $a_0, ..., a_{n-1}$. Намерете началото и край на най-дългата последователност от равни числа. Ако има няколко такива намерете първата.

Решение.

```
pair<int, int> findMaxLength() {
    pair<int, int> ans = {0, -1};
    int lastBeginning = 0;
    for (int i = 0; i < n; i++) {
        if (i == n-1 || shouldSplit(i, i+1)) {
            if (i-lastBeginning > ans.second-ans.first) {
                ans = {lastBeginning, i};
            }

            lastBeginning = i+1;
        }
    }
    return ans;
}
```

5.5 Задачи

Задача 5.19 (Есенен Турнир 2008, Е група, Шоколад).

Задача 5.20 (Софийски пролетен турнир 2019, Е група, Съгласни).

Задача 5.21 (Зимен Турнир 2012, Е група, GPS).

Задача 5.22 (НОИ, Областен кръг 2017, Е група, Картончета).

Задача 5.23 (НОИ, Национален кръг 2014, Е група, Символ).

Задача 5.24 (Зимен Турнир 2014, Е група, Спирка).

Задача 5.25 (Зимен Турнир 2014, Е група, Камера).

Задача 5.26 (НОИ, Областен кръг 2013, Е група, Свиване).

Задача 5.27 (НОИ, Областен кръг 2016, Е група, Кодиране).

Задача 5.28 (НОИ, Общински кръг 2016, C група, Тетрис).

Задача 5.29 (Пролетен Турнир 2008, D група, Синоптици).

Задача 5.30 (НОИ, Областен кръг 2010, D група, Тетрис).

Задача 5.31 (НОИ, Областен кръг 2011, D група, Ненамаляваща редица).

Задача 5.32 (НОИ, Областен кръг 2012, D група, Цветни топчета).

Глава 6

Сортиране

6.1 Най-малък елемент

Задача 6.1. Даден е масив `int a[n]`. Намерете най-малкият елемент.

Решение. За целта ни трябва една допълнителна променлива, където да пазим най-малкият елемент - `int minElement`. За начална стойност имаме два варианта. Или да сложим число което е по-голямо от всички в масива(`int minElement = 1000000000`) или да вземем първото число от масива(`int minElement = a[0]`). Винаги когато сме сигурни че имаме поне един елемент в масива ще ползваме втория вариант.
Сега идеята е да обходим всички елементи на масива един по един като винаги в `minElement` ще пазим най-малкият намерен елемент до момента. Т.е. `minElement` ще го променяме всеки път когато текущия елемент е по-малък (`if(a[i] < minElement)`).

```
int minElement = a[0];
for (int i = 1; i < n; i++) {
    if (a[i] < minElement) {
        minElement = a[i];
    }
}
```

Може да ползваме функцията (`min(a, b)`), която връща като резултат по-малкият от два елемента.

```
int minElement = a[0];
for (int i = 1; i < n; i++) {
    minElement = min(minElement, a[i]);
}
```

6.2 Индекс на най-малък елемент

Задача 6.2. Даден е масив `int a[n]`. Намерете индекса на най-малкият елемент.

Решение. За да намерим индекса ни трябва една допълнителна променлива, която да се променя заедно с (`minElement`).

```
int minIndex = 0;
int minElement = a[0];
for (int i = 1; i < n; i++) {
    if (a[i] < minElement) {
        minIndex = i;
        minElement = a[i];
    }
}
```

Сега обаче забелязваме връзката `minElement = a[minIndex]` и съответно е излишно да пазим `minElement`.

```
int minIndex = 0;
for (int i = 1; i < n; i++) {
    if (a[i] < a[minIndex]) {
        minIndex = i;
    }
}
```

6.3 Втори най-малък елемент

Задача 6.3. Даден е масив `int a[n]`. Намерете втория най-малкият елемент.

Решение. Задачата има различни начини за решение. Да разгледаме няколко варианта за решение с две обхождания, като при първото обхождане на елементите ще намерим най-малкият елемент. Една опция е при второто обхождане просто да го пропуснем. Това може да стане ако при първото обхождане сме запомнили къде е най-малкият и като го стигнем направо да продължим напред. Втори вариант е да сменим най-малкият с някакво голямо число и да повторим обхождането за намиране на най-малък елемент. Трети вариант е да разменим най-малкият с първия и да започнем второто обхождане от втория елемент. За имплементацията на всеки един вариант при първото обхождане всъщност трябва да намерим не най-малкият елемент, а неговият индекс. Трите варианта оставят различен масив накрая. Само след първият вариант масивът не се променя. При втория вариант губим едното число от масива понеже го сменяме с някакво голямо число. При третия вариант запазваме същите числа в масива, но най-малкото отива най-отпред. Ще разгледаме и четвърти вариант, при който ще направим едно обхождане. За целта във всеки момент ще пазим двата най-малко елемента до момента - $min1$ - най-малкият и $min2$ - вторият най-малък. Ако новият елемент $a[i]$ е по-малък от $min2$, то той ще заеме мястото на $min2$. Остава възможността $a[i]$ да е по-малък о и от $min1$. Ако това е така, ще трябва да разменим $min1$ и $min2$.

```
int min1 = 1000000;
int min2 = 1000000;
for (int i = 0; i < n; i++) {
    if (a[i] < min2) {
        min2 = a[i];
```

```
    }
    if (min2 < min1) {
        swap(min1, min2);
    }
}
```

6.4 Сортиране на елементи

Задача 6.4. Даден е масив `int a[n]`. Подредете числата в нарастващ ред.

След трите алгоритъма с две обхождания за намиране на втория по-големина елемент, помислете може ли някой от тях да ни помогне за сортирането.
С първата идея ще стане много трудно и със сигурност е най-неподходящ от трите. Вторият вариант би свършил работа ако подреждаме числата в допълнителен масив. Намираме най-малкото добавяме го в началото на допълнителния масив и го променяме на голямо число в първия масив. Намираме следващото най-малко в първия масив добавяме го като втори елемент на допълнителния масив и го променяме на голямо число в първия масив. Повтаряме това действие толкова пъти колкото са дадените елементи и в допълнителния масив ще ги имаме подредени по големина.
Третият вариант обаче не само ще свърши работа, но и няма да се нуждае от ползването на допълнително памет.

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

6.5 Сортиране чрез вмъкване

Задача 6.5. Дадена е редица от n числа - $a_0, ..., a_{n-1}$. Първите $n-1$ числа са подредени по-големина, в нарастващ ред. Вмъкнете последното число a_{n-1} на правилното място, така че всички числа да са сортирани.

Решение. Нека $a_{n-1} = x$. Може да сравним x с предишния елемент - a_{n-2} и ако е по-малък да ги разменим. След това ще сравним x с a_{n-3} и ще ги разменим ако се налага. Ще спрем, когато няма нужда от размяна с предишния или сме стигнали началото на редицата. Ще обхождаме по възможните предишни елементи. В момента, в който няма нужда от размяна ще спираме, в противен случай разменяме и продължаваме.

```
void insert(int n) {
    for (int i = n-2; i >= 0; i--) {
        if (a[i+1] >= a[i]) break;
        swap(a[i], a[i+1]);
    }
}
```

Една добра оптимизация е да забележим, че при всяко викане на `swap` разменяме поредното число с x. Няма нужда x да го прекопираме всеки път. За целта може в началото да си го запазим, вместо `swap` да преместваме стойността, която не е x и накрая като свършим да сложим x на мястото му.

Задача 6.6. Дадена е редица от n числа - $a_0, ..., a_{n-1}$. Променете предната програма, така че да сортирате редицата, като вмъквате всеки един елемент.

Решение. Ще приложим горния алгоритъм за всеки елемент. В резултат на това в началото на всяка стъпка имаме, че числата до $i-1$-вото са подредени, след края, числата до i-тото са подредени.

```
void insertionSort() {
    for (int i = 0; i < n; i++) {
      for (int j = i-1; j >= 0; j--) {
          if (a[j+1] >= a[j]) break;
          swap(a[j], a[j+1]);
      }
  }
}
```

Една оптимизация, която не си заслужава споменаването, е да започваме вмъкването от втория елемент, понеже първия е сортиран.
Друга по-добра оптимизация е споменатата в предната задача.

6.6 Метод на мехурчето

Задача 6.7. Дадена е редица от n числа - $a_0, ..., a_{n-1}$. Какво може да кажем за редицата след изпълнението на следния код:

```
void bubble() {
    for (int i = 1; i < n; i++) {
        if (a[i-1] > a[i]) {
            swap(a[i-1], a[i]);
        }
    }
}
```

Какво ще стане ако изпълним кода още един път?

Решение. Ако имаме число след което има по-малко те ще си разменят местата. След което по-голямото ще го сравним със следващото. Така едно голямо число ще ходи надясно докато следващото число е по-малко. При това положение какво ще се

случи с най-голямото число? То винаги ще ходи надясно и така накрая ще достигне последната позиция. След изпълнението на този код най-голямото число изплува най-накрая.
При повторно изпълнение на кода е ред на втория най-голям елемент да изплува. Така след двете изпълнения ще имаме, че последния елемент е най-големия, а предпоследния втори по-големина.

Задача 6.8. Колко пъти трябва да изпълним горния код за да сортираме числата?

Решение. При всяко изпълнение ще имаме, че един елемент в края застава на мястото си. Така при n извиквания ще имаме, че всичките елементи са на местата си. Всъщност няма нужда от последното извикване, но това е дребно подробност. Може да направим една по-добра оптимизация. След като знам, че на всяка стъпка последните няколко числа са вече сортирани може да си спестим сравненията до края на редицата.

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

6.7 Вградена функция за сортиране

Сортирането се ползва толкова често, че няма как в c++ да пропуснат да ни предоставят помощна функция за него.
Ако имаме масив a с n елемента, то може да го сортираме в нарастващ ред по следния начин:

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

Ако искаме да сортираме в намаляващ ред пак може да използваме функцията *sort* по следния начин:

```
sort(a, a+n, greater<int>());
```

Съответно вместо *int* трябва да напишем типа данни които ползваме.
Ако няма причина за друго винаги когато искаме да сортираме масив ще използваме този начин. Функцията sort също така работи доста по-бързо от нашия метод по-горе.
За да подредим символите на един string s, също може да използваме функицята sort, но по-малко по-различен начин:

```
sort(s.begin(), s.end());
```

За да подредим буквите в низа в обратен ред ползваме

```
sort(s.begin(), s.end(), greater<char>());
```

6.8 Pair

Задача 6.9. Имаме n човека. За всеки е дадено името му(низ) и височината в сантиметри(цяло число). Изпишете имената на хората в нарастващ ред на височините им. При равна височина първо да е името по-напред в азбуката.

Възможно е да използваме един масив от низове за имената и един от цели числа за височините. Трябва да подредим по нарастващ ред на височините. Ако подредим само масива с числата ще изгубим връзка между височините и имената. Трябва някак си подреждането да става едновременно. Един вариант е да ползваме нашият код за подреждане и всеки път когато разменяме две височини да разменяме съответните имена, за да ги държим в синхрон. Проблема е че нашият начин на подреждане е бавен. Обаче с вградената функция не знаем кога се разменят елементите. Така да ползваме нашият код е по-добре от нищо, но пак не е достатъчно.
Искаме да "вържем"височината на човека с неговото име. Когато разменяме нещо да разменяме и двете. Това "завързване"може да стане с един нов тип данни в c++, направен специално за това. Това е /textitpair. Благодарение на него свързваме два от простите типове в един по-сложен тип. Когато създаваме този свързващ тип трябва да знаем какви типове ще свързваме. В нашия случай това са име(string) и височина(int).

```
pair<string, int> p;
```

Сега искаме да дадем стойности на низа и цялото число. Това става чрез

```
p = make_pair("pesho", 172);
```

И за да се обръщаме към двете стойности има значение кой тип сме дали първи и кой втори. Да ги четем или променяме, използваме

```
p.first
p.second
```

В нашия случай *p.first* ще бъде низа и *p.second* - цялото число.
Сега идва голямата полза от сдвоените в pair неща. Може да сортираме масив от pair елементи. По подразбиране сортирането сравнява първите елементи на двойките. Ако те са равни, тогава сравнява вторите. Трябва единтсвено да сменим имената и височините, като първи и втори елемент, понеже първото нещо по което искаме да сортираме е височината. Ето как ще изглежда кода:

```
#include<bits/stdc++.h>
using namespace std;

pair<int, string> p[100000];
int main() {
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    string name;
    int height;
    cin >> name >> height;
```

```
    p[i] = make_pair(height, name);
  }
  sort(p, p+n);
  for (int i = 0; i < n; i++) {
    cout << p[i].second << endl;
  }

  return 0;
}
```

6.9 Задачи

Задача 6.10 (Есенен Турнир 2015, D група, Магазин).

Задача 6.11 (част от Пролетен Турнир 2017, E група, Животинска аритметика). Даден е един ред на който има малки латински букви и интервали в някакъв ред. Изпишете първата, третата и т.н букви.

вход	изход
pa	p
biso	bs
p a pa	pp

Задача 6.12 (Пролетен Турнир 2017, E група, Избори).

Задача 6.13 (част от Пролетен Турнир 2016, E група, Почивни дни). На един ред е дадена дата във формат ден.месец.година. Дните и месеците могат да съдържат водеща нула, но това не е задължително. На един разделени с интервал изпишете същата дата във формат месец.ден.година с и без водещи нули.

вход	изход
10.06.2016	10.06.2016 10.6.2016
20.6.2016	20.06.2016 20.6.2016
1.3.2020	01.03.2020 1.3.2020

Глава 7

Броене

7.1 Броене

Задача 7.1. Намерете броя на нулите в редицата $a_0, ..., a_{n-1}$.

Решение. Решението е лесно - създаваме една променлива за броя br, обхождаме всички елементи на масива, като за всеки равен на нула увеличаваме br.

```
int countZeros() {
    int br = 0;
    for (int i = 0; i < n; i++) {
        if (a[i] == 0) {
            br++;
        }
    }
    return br;
}
```

Задача 7.2. Намерете броя на нулите и единиците в редицата $a_0, ..., a_{n-1}$.

Решение. Задачата не е много по-различна. Може да използваме две променливи - една за броя на нулите (`br0`) и една за броя на единиците (`br1`).

Задача 7.3. Намерете броя срещания на всяко число между нула и сто в редицата $a_0, ..., a_{n-1}$.

Решение. Сега ще ни трябват много повече променливи - `br0, br1, ..., br100`. Разбира се в такъв случай ще използваме масив - `int br[101]`. Така броя нули ще пазим в `br[0]`, броя единици в `br[1]` и т.н. Така за да отбележим срещането на числото `a[i]` ще увеличаваме стойността на `br[a[i]]`.

```
void countAndPrint() {
    for (int i = 0; i < n; i++) {
        br[a[i]]++;
    }
    for (int i = 0; i <= 100; i++) {
```

```
        cout << br[i] << endl;
    }
}
```

7.2 Сортиране чрез броене

Задача 7.4. Дадени са n неотрицателни числа - $a_0, ..., a_{n-1}$. Как може да ги подредим по големина чрез броене?

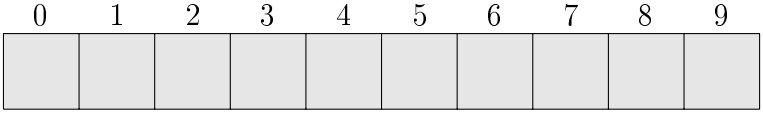
Решение. Може да създадем един масив `br` и да го запълним така, че стойността на i-ия елемент да е равна на броя срещания на числото i в редицата. След това ще минем през всички възможни числа и ще видим колко нули имаме, колко единици и т.н.

```
void countingSort() {
    for (int i = 0; i < n; i++) {
        br[a[i]]++;
    }
    int index = 0;
    for (int i = 0; i <= MAXA; i++) {
        for (int j = 0; j < br[i]; j++) {
            a[index] = i;
            index++;
        }
    }
}
```

7.3 Броене на големи числа

Задача 7.5. Дадени са $q \leq 10^6$ заявки. На всяка или се добавя ново число или се пита за наличието на друго. Напишете бързо решение, ако числата са в интервала $[0, 10^18]$.

Ако числата са малки знаем решението - ще използваме масив за броене. Проблемът в случая е че няма как да създадем масив с толкова много елементи. Това поставя интересен въпрос, можем ли да броим големи числа с не толкова много елементи? Да помислим как би могло да стане това.
Да разгледаме един по-малък пример. Как с масив от десет елемента(десет места за пъхане на числа), ще пазим информация за числа между 0 и 50?



За числата между 0 и 9 е логично да пазим информация в съответстващите им индекси.

7.4 Двойки суми

Задача 7.6. Дадена е редица от n числа. По колко начина може да изберем два елемента със сума s?

Може да обходим всички възможни двойки и да видим колко от тях имат сума s.

```
int countPairs(int[] a, int n, int s) {
    int br = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            if (a[i]+a[j] == s) br++;
        }
    }
    return br;
}
```

Разбира се това решение ще има квадратна сложност и трябва да измислим по-добро. Основната разлика ще дойде от проверката, която правихме преди малко - $a[i] + a[j] == s$. Гледаме двойките елементи и проверяваме дали сумата е s, което води до много проверки, които не са изпълнени. Вместо да проверяваме всички възможни стойности за $a[j]$, то може да използваме, че $a[j] = s - a[i]$. Сега знаем колко трябва да е $a[j]$ и остава да проверим колко числа са равни на $a[j]$.

```
int countPairs(int[] a, int n, int s) {
    unordered_map<int, int> br;
    int ans = 0;
    for (int i = 0; i < n; i++) {
        ans += br[s-a[i]];
        br[a[i]]++;
    }
    return ans;
}
```

7.5 Редици определени от първия елемент

Задача 7.7. Дадена е редица от n числа. Колко най-малко елемента от редицата трябва да променим, така че да получим редица от последователни нарастващи числа.

Новата редицата се определя от всеки един неин елемент. Така ако първото число е x, то редицата ще е $x, x+1, x+2, ..., x+n-1$. Ако имаме x, може да сравним двете редици и да преброим колко елемента трябва да променим в първата. Така ако имаме всички възможни стойности на x, за всяко от тях ще намерим броя необходими промени и ще решим задачата. За целта обаче ще имаме толкова обхождания на редиците, колкото са възможните стойности за x.
За да забързаме нещата ни трябва още нещо. За сега знаем, че втората редица е напълно определена от първия си елемент x. Числото a_i във финалната редица ще е

равно на $x+i$. Съществото нещо, което може да забележим е че всеки елемент от първата редица a_i няма да има нужда от промяна при само един възможен вариант за x. Така a_i няма да се промени само ако $a_i = x + i$, т.е. $x = a_i - i$. Сега остава да намерим тази стойност на x, за която най-голям брой елементи няма да се променят. Това е най-често срещаното число от всички $a_i - i$.

Сега вече е видно, че за $x = 3$, ще имаме 3 елемента, които не се променят. Ако $x = 5$, ще са 2, за $x = -2$, ще е един и за всички други стойности на x ще трябва да променим всички елементи.

```
int minChanges(int[] a, int n) {
    unordered_map<int, int> br;
    int changes = n;
    for (int i = 0; i < n; i++) {
        int x = a[i]-i;
        br[x]++;
        changes = min(changes, n-br[x]);
    }
    return changes;
}
```

7.6 Задачи

Задача 7.8 (НОИ - Областен кръг 2009, D1, Камъчета).

Задача 7.9 (НОИ - Общински кръг 2018, C1, Балони).

Задача 7.10 (НОИ - Общински кръг 2012, C1, Топчета).

Задача 7.11 (НОИ - Общински кръг 2018, D3, Редица).

Задача 7.12 (Зимен турнир 2014, D3, Ученици).

Задача 7.13 (НОИ - Национален кръг 2019, D4, Аритметична прогресия).

Задача 7.14 (НОИ - Общински кръг 2014, C1, Суми).

Задача 7.15 (НОИ - Национален кръг 2016, D4, Везни).

Глава 8

Умно обхождане

8.1 Обхождане

Задача 8.1. Дадена е редицата $a_0, a_1, ..., a_{n-1}$. Намерете най-голямата разлика между две числа a_i и a_j, където $i < j$.

Решение.

Задача 8.2. Дадена е редицата $a_0, a_1, ..., a_{n-1}$. Намерете най-голямата разлика между две числа a_i и a_j, където $i < j$.

Решение. Лесното решение е да разгледаме всички такива двойки числа. То ще има квадратна сложност. Да видим как ще изглежда задачата, ако фиксираме по-малкото число. Ако то е a_j, то трябва да намерим най голямата разлика от $a_0 - a_j, ..., a_{j-1} - a_j$. Очевидно тя ще се получава за най-голямото от възможните умаляеми. Така търсената разлика е $max(a_0, ..., a_{j-1}) - a_j$. Така за всяка a_j трябва да намерим най-голямото от числата преди него.

Задача 8.3. Дадена е редицата $a_0, a_1, ..., a_{n-1}$, която е пермутация на числата от 1 до n. За колко индекса i, числата $a_0, ..., a_i$ са пермутация на числата от 1 до $i + 1$.

Решение. Един вариант за решение е с броене да пазим кои числа са минали и за всяко i да проверяваме дали имаме числата от 1 до $i + 1$. Проверката обаче изисква доста обхождане и решението ще има квадратна сложност. Друга опция е да използваме сортиране чрез вмъкване и така на всяка стъпка i числата да момента да са подредени. Ако числата до момента са подредени, за да са пермутация, последното трябва да е $i+1$. Но и това решение ще има квадратна сложност. Да разгледаме елементите $a_0, ..., a_i$. За да да пермутация, те трябва да са числата $1, ..., i+1$ в разбъркан ред. Да видим кога това няма да е изпълнени - във всеки случай когато имаме, че някое от числата е по-голямо от $i + 1$. Така забелязваме, че числата са пермутация тогава и само тогава, когато $max(a_0, ..., a_i) = i + 1$.

```
int countPermutations() {
    int ans = 0;
    int currentMax = 0;
    for (int i = 0; i < n; i++) {
        currentMax = max(currentMax, a[i]);
```

```
        if (currentMax == i+1) {
            ans++;
        }
    }
    return ans;
}
```

Задача 8.4. Дадена е редицата $a_0, a_1, ..., a_{n-1}$, която е пермутация на числата от 1 до n. За всеки индекс i, подреждаме числата $a_0, ..., a_i$ в нарастващ ред. За колко индекса числата са последователни след подредбата?

Решение. Искаме да проверим дали числата $a_0, a_1, ..., a_i$ след сортиране са от вида $k, k+1, ..., k+i$. Проблем е, че в случая не знаем k. Всъщност това може лесно да го решим. Очевидно k е най-малкото от числата, значи $k = min(a_0, ..., a_i)$. След като имаме k задачата вече е много сходна с предишната. За да са последователни числата, трябва най-голямото число да е $k + i$.

```
int countConsecutives() {
    int ans = 0;
    int currentMin = 1000000;
    int currentMax = 0;
    for (int i = 0; i < n; i++) {
        currentMax = min(currentMin, a[i]);
        currentMax = max(currentMax, a[i]);
        if (currentMax == currentMin+i) {
            ans++;
        }
    }
    return ans;
}
```

Задача 8.5. Дадени са n цели числа $a_0, a_1, ..., a_{n-1}$. Намерете най-голямата сума от последователни числа.

Решение. За лесни решения има няколко варианта. Най-бавното е да разгледаме всички такива последователности, като за всяка от тях намерим сумата на числата. За намирането на сумата на дадена последователност може да използваме префиксни суми, което ще забърза нещата.
Сега да разгледаме по-различна идея. Искаме да обходим последователно числата и за всяко число a_i ще намерим най-голямата сума, която завършва в това число. Нека m_i е най-голямата сума на поредица числа, която завършва в a_i. Ще разгледаме два основни варианта, в зависимост от това дали числото a_{i-1} участва или не в най-голямата сума завършваща на a_i. Ако числото a_{i-1} не участва, то няма как да участват други числа и m_i ще е равно на a_i. Ако числото a_{i-1} участва вместо да гледаме предишните числа, може да използваме вече намерената най-голяма сума m_{i-1}. Така m_i ще бъде равно на $m_{i-1}+a_i$. Окончателно за m_i ще изберем по-голямата стойност от двете възможности - $m_i = max(a_i, m_{i-1} + a_i) = a_i + max(0, m_{i-1})$.

Задача 8.6. Дадени е редица с n цели числа $a_0, a_1, ..., a_{n-1}$. За всяко число a_i намерете най-дългата последователност от строго нарастващи числа с край i-тото число?

Решение. Наивното решение е за всяко число a_i да започнем да обхождаме наляво, докато числата намаляват.
По-умния вариант е да използваме връзка между различните отговори. Нека m_i е дължината на най-дългата последователност от строго нарастващи числа с край i-тото число. Когато търсим m_i имаме два варианта - или числото a_{i-1} ще участва в търсената последователност или не. Ако не участва, то значи отговорът ще е 1. Ако участва обаче може да използваме най-дългата последователност с последен елемент a_{i-1} и така отговорът ще е $m_{i-1}+1$. Дали a_{i-1} участва или не зависи от това дали е изпълнено $a_{i-1} < a_i$.

8.2 Обхождане отзад напред

Задача 8.7. Дадени е редица с n цели числа $a_0, a_1, ..., a_{n-1}$. За всяко число a_i намерете най-дългата последователност от строго намаляващи числа с начало i-тото число?

Задача 8.8. Разглеждаме редицата $a_0, a_1, ..., a_{n-1}$ от положителни числа. Нека m_i е най-голямото от числата $a_i, ..., a_{n-1}$, т.е. $m_i = max(a_i, ..., a_{n-1})$. Намерете числата $m_0, m_1, ..., m_{n-1}$.

Решение. Тривиалното решение е за всяко m_i да решим задачата поотделно. Така за всяко i ще обходим числата $a_i, ..., a_{n-1}$ и ще намерим най-голямото. Това решение има квадратна сложност. За да подобрим бързината трябва да намерим някаква връзка между отговорите за отделните числа. Да разгледаме как се получават отговорите за два съседни елемента m_i и m_{i+1}. Имаме, че $m_i = max(a_i, a_{i+1}, ..., a_{n-1})$ и $m_{i+1} = max(a_{i+1}, ..., a_{n-1})$. За да сметнем m_i разглеждаме всички числа, които влизат в отговора за m_{i+1} и самото a_i. Така имаме връзката, че $m_i = max(m_{i+1}, a_i)$. Така ако сме сметнали отговора m_{i+1} може много бързо да сметнем m_i. За целта може да обхождаме отзад напред. Остава първоначално да сложим $m_{n-1} = a_{n-1}$, за да избегнем излизане от границите на масива. За всеки елемент смятаме отговора с една операция, което води до линейна сложност на решението.

```
void solve() {
    m[n-1] = a[n-1];
    for (int i = n-2; i >= 0; i--) {
        m[i] = max(m[i+1], a[i]);
    }
}
```

Задача 8.9. Разглеждаме редицата $a_0, a_1, ..., a_{n-1}$ от положителни числа. Намерете за колко елемента всички следващи са по-малки, т.е. броя елементи a_i, за които всички числа $a_{i+1}, ..., a_n$ са по-малки от a_i.

Решение. Тривиално решение е ясно. Да видим какво означава всички елементи след a_i да са по-малки - $a_i < a_{i+1}, ..., a_i < a_{n-1}$. Главния проблем идва от това, че

трябва да сравним a_i с много числа, което всъщност това не е необходимо. Достатъчно е да сравним a_i с най-голямото от числата след него. Ако $a_i \geq max(a_{i+1}, ..., a_{n-1})$, то очевидно a_i ще отговаря на условието. Така стигаме до нещо подобно на предната задача. Сега обаче, няма да използваме допълнителен масив, а само една променлива - `currentMax`. На всяка стъпка i тя ще е равна на $max(a_{i+1}, ..., a_{n-1})$.

```
int solve() {
    int ans = 0;
    int currentMax = 0;
    for (int i = n-1; i >= 0; i--) {
        if (a[i] > currentMax) {
            ans++;
        }
        currentMax = max(currentMax, a[i]);
    }
    return ans;
}
```

8.3 Обхождане в двете посоки

Задача 8.10. Дадена е редица от n числа - $a_0, a_1, ..., a_{n-1}$. Намерете дължината на най-дългата последователност от елементи, в която стойностите първо са строго растящи, а после строго намаляващи.

Решение.

Задача 8.11. Дадена е редица от n числа - $a_0, a_1, ..., a_{n-1}$. Намерете дължината на най-дългата последователност от елементи, която може стане строго нарастваща с промяната на най-много един елемент.

Решение.

Задача 8.12. По колко начина може да разделим редица от цели числа на три последователности с равни суми.

Решение.

8.4 Задачи

Задача 8.13 (НОИ - Областен кръг 2016, D2, Лидери).

Задача 8.14 (НОИ - Общински кръг 2018, C1, Балони).

Задача 8.15 (НОИ - Общински кръг 2012, C1, Топчета).

Задача 8.16 (НОИ - Общински кръг 2018, D3, Редица).

Задача 8.17 (Зимен турнир 2014, D3, Ученици).

Задача 8.18 (НОИ - Национален кръг 2019, D4, Аритметична прогресия).

Задача 8.19 (НОИ - Общински кръг 2014, C1, Суми).

Задача 8.20 (НОИ - Национален кръг 2016, D4, Везни).

Глава 9

Префиксни суми

9.1 Загрявка

Задача 9.1. Намерете броя числа в интервала $[l, r]$, които се делят на 7.
Ограничения: $1 \leq l \leq r \leq 10^{18}$.

Решение. Може да обходим и проверим всички числа в интервала, но ще е доста бавно. Може да го забързаме като намерим първото число кратно на 7 и после прескачаме през 7, но пак може да се наложи да проверим много числа.
Да разгледаме друг подход. Знаем, че всяко седмо число е кратно на седем. От там идва идеята да вземем броя числа в интервала и да го разделим на 7. Идеята е много добра, но не съвсем вярна - има интервали с по равен брой числа, но различен брой кратни на 7, например интервалите $[6, 13]$ и $[7, 14]$. Това, че всяко седмо число е кратно на 7, ще ни даде правилно решение, ако броя на числата в интервала е кратен на 7. Може да използваме това, като проверим последните няколко числа поотделно, и да намалим интервала така, че той да стане с кратен на 7 брой числа и да използваме формула за останалия интервал. Например в интервала $[20, 99]$ има 80 числа. За да получим кратен на 7 брой може да извадим последните 3 от интервала и да ги проверим отделно. Така ще имаме интервал със 77 числа - $[20, 96]$ и отделно ще разгледаме 97, 98 и 99. В $[20, 96]$ има $77/7 = 11$ кратни на 7, 98 също е кратно на 7 и така общия отговор за интервала $[20, 99]$ ще е 12.
Друг подобен вариант е да намерим първото и последното число кратни на 7 и да използваме формула свързана с тях. Ако първото и последното число кратни на 7 са a и b, то броят на всички е $(b - a)/7 + 1$.
Последните два варианта решават задачата, но има доста случаи за които трябва да се внимава. Също ако търсим кратни не на 7, а на нещо доста по-голямо ще трябва по-умно да намираме последното число в интервала кратно на даденото.
Целта на тази глава е да ви накара винаги когато видите нещо, което се търси в произволен интервал, да пробвате да го разбиете на два интервала, които започват от едно място, например 0 или 1. В тази задача в сила е следната важна връзка - броя числа кратни на 7 в $[l, r]$ е равен на броя числа кратни на 7 в $[1, r]$ минус броя числа кратни на 7 в $[1, l - 1]$. Иначе казано, ако преброим числата кратни на 7 от 1 до r, ще сме броили и тези по-малки от l, за това трябва да ги извадим. Важно е да отбележим, че интервала, който вадим е до $l - 1$, понеже самото l, не трябва го вадим.
Остава да видим как да сметнем броя числа кратни на 7 в интервала $[1, n]$. Тук вече

доста по-лесно може да съобразим, че отговора е точно $n/7$.
Следва имплементация на решението:

```
long long solve(long long l, long long r) {
    return r/7-(l-1)/7;
}
```

Задача 9.2. Намерете броя числа в интервала $[l, r]$, които се делят на d.
Ограничения: $1 \leq l \leq r \leq 10^{18}, 1 \leq d \leq 10^{18}$.

Решение. Идеята е същата, само сменяме 7 с d.

```
long long solve(long long l, long long r) {
    return r/d-(l-1)/d;
}
```

9.2 Префиксни суми

В тази глава всички редици с n елемента ще ги пазим в масиви с $n+1$ елемента. Ако масива е a, то елемента a_0 ще бъде помощен и винаги ще има стойност нула. Елементите представляващи редицата ще са $a_1, ..., a_n$.

Дефиниция 9.1 (Префиксна сума)**.** Да разгледаме редицата $a_1, a_2, ..., a_n$. Всяка сума от вида $p_i = a_1 + ... + a_i$, която включва първите няколко последователни числа се нарича префиксна сума.

Задача 9.3. Дадена е редица с n числа - $a_1, a_2, ..., a_n$ и q заявки. За всяка заявка е дадено едно число k и трябва да намерите сумата $a_1 + a_2 + ... + a_k$.
Ограничения: $1 \leq n \leq 10^6, 1 \leq q \leq 10^6, 0 \leq a_i \leq 10^6, 1 \leq k \leq n$.

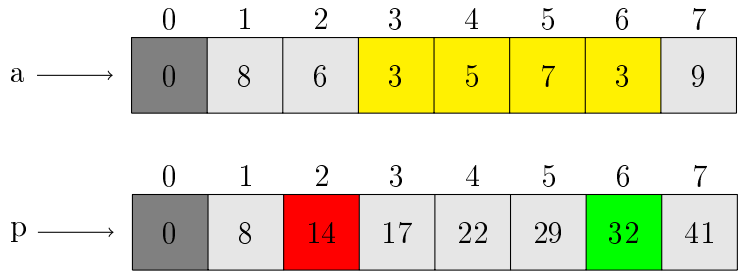
Решение. Първо ще отбележим, че сумата на числата може да стане голяма и за това ще я пазим в променлива от тип *long long*.
Задачата има очевидно решение - за всяка заявка обхождаме всички числа $a_1, a_2, ..., a_k$ и ги събираме. Това решение обаче не е за максимален резултат, понеже е бавно. За всяка заявка може да са необходими близо до n събирания, като умножим по q заявки, получаваме nq операции. Това е доста голямо число при дадените ограничения. Понеже числата в заявките са ограничени до n, то ние може предварително да пресметнем всички отговори. В масива p ще пазим тези суми, т.е. $p_i = a_1 + a_2 + ... + a_i$, p_i е сумата на всички числа до i-тото. Ако имаме този масив на всяка заявка k, отговорът ще е p_k.
Сега трябва да запълним масива p. Отново имаме бавен вариант като той изисква за всяко i да съберем всички числа, участващи в p_i.
Много лесно обаче може да забързаме нещата. Да разгледаме сумата $p_i = a_1 + a_2 + ... + a_{i-1} + a_i$. Знаем, че $p_{i-1} = a_1 + a_2 + ... + a_{i-1}$ и може да заместим. Така стигаме до $p_i = p_{i-1} + a_i$, което е идеално за нас. Иначе казано сумата на първите i числа е равна на сумата на първите $i-1$ числа плюс i-тото число. Така с едно обхождане на масива може да пресметнем всички суми от началото.

Тук е момента да отбележим, че при така намерената формула $p_i = p_{i-1} + a_i$ имаме $p_1 = p_0 + a_1$. Това p_0, което използваме при смятането на първата префиксна сума е една от причините да имаме нулев елемент със стойност нула. Ако нямахме този първи елемент първата префиксна сума щеше да е $p_0 = p_{-1} + a_0$ и щяхме да излизаме от масива.

```
void solve(int n, int q) {
    for (int i = 1; i <= n; i++) {
        p[i] = p[i-1]+a[i];
    }
    for (int i = 0; i < q; i++) {
        int k;
        cin >> k;
        cout << p[k] << endl;
    }
}
```

Задача 9.4. Дадена е редица с n числа - $a_1, a_2, ..., a_n$ и q заявки. За всяка заявка са дадени две числа l и r, и трябва да намерите сумата $a_l + a_{l+1} + ... + a_r$.
Ограничения: $1 \leq n \leq 10^6, 1 \leq q \leq 10^6, 0 \leq a_i \leq 10^6, 1 \leq l \leq r \leq 10^6$.

Решение. Сумата на числата отново трябва да е от тип *long long*.
По подобие на предната задача има бавно решение при което за всяка заявка обхождаме всички числа $a_l, a_{l+1}, ..., a_r$ и ги събираме.
Идеята за подобрение идва от това, че работим с интервали. Прилагаме първото правило при решаване на задачи с интервали $[a, b]$ - да проверим дали може да решим задачата като разлика на два интервала с общо начало.
Да разгледаме редицата $a = \{8, 6, 3, 5, 7, 3, 9\}$ и да намерим сумата от 3ия до 6ия елемент, $a_3 + a_4 + a_5 + a_6$. Да видим как може да използваме префиксните суми p, където $p_i = a_1 + ... + a_i$.



Лесно може да се усетим, че $a_3 + a_4 + a_5 + a_6 = (a_1 + a_2 + a_3 + a_4 + a_5 + a_6) - (a_1 + a_2) = p_6 - p_2$. Може и да го видим - сумата от числата в жълтите квадратчета е равна на разликата от числата в зеленото и червеното квадратче.
Може да обобщим, че $a_l + ... + a_r = p_r - p_{l-1}$, т.е. сумата на числата $a_l + a_{l+1} + ... + a_r$ е равна на сумата на всички числа от началото до r минус сумата на всички числа преди l.

```
void solve(int n, int q) {
    for (int i = 1; i <= n; i++) {
```

```
        p[i] = p[i-1]+a[i];
    }
    for (int i = 0; i < q; i++) {
        int l, r;
        cin >> l >> r;
        cout << p[r]-p[l-1] << endl;
    }
}
```

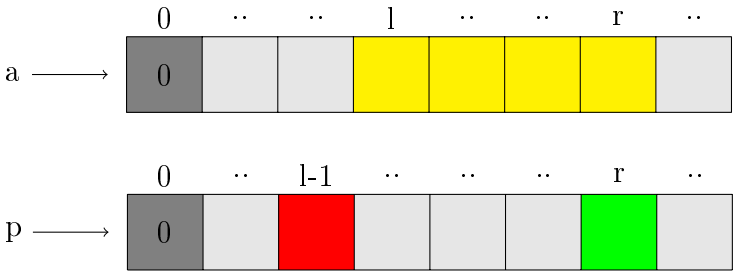
Задача 9.5. Дадена е последователност от n букви - $c_1, c_2, ..., c_n$ и q заявки. Всяка буква е a, b или c. За всяка заявка са дадени две числа l и r, и трябва да намерите най-често срещаната буква в интервала $[l, r]$. Ако има няколко такива букви изведете най-предната в азбуката.
Ограничения: $1 \leq n \leq 10^6, 1 \leq q \leq 10^6, 1 \leq l \leq r \leq 10^6$.

Решение. Може да използваме идея подобна на префиксните суми и за всяка буква поотделно да броим колко пъти се среща в първите i символа. Нека pa_i е броят срещания на буквата a в първите i символа. Ако c_i е буквата a, то $pa_i = pa_{i-1} + 1$. В противен случай $pa_i = pa_{i-1}$. Аналогично правим и за другите букви.
В интервала $[l, r]$ буквата a се среща $pa_r - pa_{l-1}$ пъти. За всяка заявка с префиксните масиви знаем коя буква колко пъти се среща и намираме най-често срещаната такава.

```
void solve(int n, int q) {
    for (int i = 1; i <= n; i++) {
        pa[i] = pa[i-1];
        pb[i] = pb[i-1];
        pc[i] = pc[i-1];
        if (c[i] == 'a') {
            pa[i]++;
        } else if (c[i] == 'b') {
            pb[i]++;
        } else {
            pc[i]++;
        }
    }
    for (int i = 0; i < q; i++) {
        int l, r;
        cin >> l >> r;
        pair<int, char> ans = {pa[r]-pa[l-1], 'a'};
        if (pb[r]-pb[l-1] > ans.first) {
            ans = {pb[r]-pb[l-1], 'b'};
        }
        if (pc[r]-pc[l-1] > ans.first) {
            ans = {pc[r]-pc[l-1], 'c'};
        }
        cout << ans.second << endl;
    }
}
```

Задача 9.6. Дадена е редица с n числа - $a_1, a_2, ..., a_n$. Намерете дали съществува последователност от числа със сума равна на нула.
Ограничения: $1 \leq n \leq 10^6, 0 \leq a_i \leq 10^6$.

Решение. Бавните решения включват проверка за всяка последователност, като префиксните суми ще помогнат бързо да проверяваме дали сумата в даден интервал е нула.
Да помислим как може да използваме префиксните суми $p_i = a_1 + ... + a_i$ по ефикасно в тази задача. Нека да си представим, че съществува последователност $a_l + ... + a_r = 0$.



Знаем, че $a_l + ... + a_r = p_r - p_{l-1}$ и понеже разглеждаме последователност със сума нула, то $0 = p_r - p_{l-1}$, т.е. $p_{l-1} = p_r$. Това което получихме е, че ако има последователност с нулева сума, то има две равни префиксни суми. Лесно може да проверим, че и обратното е вярно. Т.е. нулева сума ще има тогава и само тогава, когато има поне две равни префиксни суми.
Възможно е да получим, че нашия специален елемент $p_0 = p_r$. Това ще се случи ако имаме нулева сума от първия елемент до някъде, което означава, че трябва да имаме предвид и p_0. Т.е. като проверяваме за равни префиксни суми, трябва да имаме гледаме и p_0.
Сега остава да промерим дали има две равни префиксни суми. За целта може да ги сортираме и след това да проверим дали има две съседни равни числа.

```
bool zeroSum(int n) {
    for (int i = 1; i <= n; i++) {
        p[i] = p[i-1]+a[i];
    }

    sort(p, p+n+1);

    bool hasZeroSum = false;
    for (int i = 1; i <= n; i++) {
        if (p[i] == p[i-1]) {
            hasZeroSum = true;
            break;
        }
    }
    return hasZeroSum;
}
```

9.3 Суфиксни суми

Дефиниция 9.2 (Суфиксна сума)**.** Да разгледаме редицата $a_1, a_2, ..., a_n$. Всяка сума от вида $s_i = a_i + a_{i+1} + ... + a_n$, която включва последните няколко последователни числа се нарича суфиксна сума.

Задача 9.7. Дадена е редица с n числа - $a_1, a_2, ..., a_n$ и q заявки. За всяка заявка е дадено едно число k и трябва да намерите сумата $a_k + a_{k+1} + ... + a_n$.
Ограничения: $1 \leq n \leq 10^6, 1 \leq q \leq 10^6, 0 \leq a_i \leq 10^6, 1 \leq k \leq n$.

Решение. Тази задача е много подобна на задачата за префиксните сума. Нека $s_i = a_i + ... + a_n$. Връзката която може да съобразим е, че $s_i = s_{i+1} + a_i$.
Понеже да сметнем s_i ни трябва s_{i+1}, то трябва да попълваме масива s в обратен ред.
За първия суфикс имаме $s_n = s_{n+1} + a_i$. Така както при префиксите използвахме нулевия елемент за служебен, тук ще трябва да имаме един елемент след последния. За улеснение най-добре винаги да имаме два служебни - един в началото и едни в края.

```
void solve(int n, int q) {
    for (int i = n; i >= 1; i--) {
        s[i] = s[i+1]+a[i];
    }
    for (int i = 0; i < q; i++) {
        int k;
        cin >> k;
        cout << s[k] << endl;
    }
}
```

Задача 9.8. Дадена е редица с n цифри - $a_1, a_2, ..., a_n$. За всяка позиция i намерете последната цифра на произведението на всички числа без i-тото, т.е за всяко i, намерете $(a_1 * ... * a_{i-1} * a_{i+1} * ... * a_n)\%10$.
Ограничения: $1 \leq n \leq 10^6, 0 \leq a_i \leq 9$.

Решение. Първо ще отбележим, че при произведение резултата бързо ще надвиши *int* и *long long*. Понеже ни трябва последната цифра ще пазим винаги само нея. Дори след като прочетем първоначалните числа ще ги сменим с последната им цифри, понеже другите не оказват влияние.
За варианта да пазим произведението на всички числа и да делим на всяко ще ни трябват дълги числа и въпреки това пак ще е бавно.
Друго решение е за всяко число да умножим всички останали, което очевидно е бавно.
Понеже имаме интервали логично е да се запитаме дали може да използваме префиксни суми, като в тази задача те ще се префиксни произведения. Нека p_i е последната цифра от произведението на първите i числа - $p_i = (a_1 * ... * a_i)\%10$. Като заместим в това, което търсим получаваме $(a_1 * ... * a_{i-1} * a_{i+1} * ... * a_n)\%10 = (p_{i-1} * a_{i+1} * ... * a_n)\%10$. Имаме произведението на всички числа преди a_i, но сега ни трябва и произведението на всички числа след a_i. Но всъщност това са суфиксните произведения. Така

ако пресметнем и тях в масива s, бързо ще може да кажем, че последната цифра на произведението на всички числа без i-тото е $(p_{i-1} * s_{i+1})\%10$.
Понеже имаме произведение двата служебни елемента в началото и в края трябва да ги сложим да са равни на 1.

```
void solve(int n) {
    p[0] = 1;
    for (int i = 1; i <= n; i++) {
        p[i] = (p[i-1]*a[i])%10;
    }
    s[n+1] = 1;
    for (int i = n; i >= 1; i--) {
        s[i] = (s[i+1]*a[i])%10;
    }
    for (int i = 1; i <= n; i++) {
        cout << (p[i-1]*s[i+1])%10 << endl;
    }
}
```

9.4 Задачи

Задача 9.9. Дадена е редица с n числа - $a_1, a_2, ..., a_n$. Намерете номерата на всички елементи, за които сумата на числата преди тях е равна на сумата на числата след тях, т.е всички числа i, за които $a_1 + ... + a_{i-1} = a_{i+1} + ... + a_n$. Може да считаме, че сумата на числата преди първия и след последния е равна на нула.
Ограничения: $1 \leq n \leq 10^6, 1 \leq q \leq 10^6, -10^6 \leq a_i \leq 10^6, 1 \leq k \leq n$.

Задача 9.10. Летен Турнир 2018, E група, E3.Редица

Задача 9.11. НОИ 3 2017, E група, E3. Редица от правоъгълници

Задача 9.12. НОИ 3 2015, D група, D4. Пропуснат множител

Задача 9.13. НОИ 2 2013, C група, C3. Думи

Задача 9.14. НОИ 3 2019, D група, D3. Поздрави

Задача 9.15. НОИ 2 2020, C група, C2. Диапазон

Задача 9.16. Втора контрола за младежкия национален 2018, C група, CK4. Баланс

Глава 10

Груба сила

10.1 Дефиниция

Дефиниция 10.1 (Груба сила)**.** Това е подход за решаване на задачи, при който се генерират всички възможни кандидати за решение и от тези които отговарят на изискванията се намира най-подходящия. По-често ще срещате термина на английски - Brute force.

За решаването на една задача чрез груба сила трябва да преминем през няколко стъпки:

1. Да открием всички възможности - например всички начини да поставим n топки в k кутии.
2. Да генерираме всички тези възможности.
3. Да проверим кои възможности отговарят на условието на задачата - може да сме сложили в някоя кутия повече топки от позволеното.
4. Да проверим коя от отговарящите възможности дава най-доброто решение.

10.2 Груба сила с двойки и тройки

Задача 10.1. Изведете на екрана всички наредени двойки с числа в интервала от $[0; n-1]$, без повторения на едно число. Понеже двойките са наредени, ако изведете (a, b), трябва отделно да извеждате (b, a).

Решение. Най-стандартния вариант да минем през всички двойки е с два вложени цикъла. В първия фиксираме първия елемент от двойката и за него във втория цикъл обхождаме всички възможни втори елементи. Така при фиксиран първи елемент обхождаме всички двойки - $(i, 0), (i, 1), ..., (i, n-1)$.
Решението е супер само, че в този вариант при j равно на i ще изведем двойка с повторение на число. Понякога това може да е целта, но не и сега. За да не го изведем ще направим една допълнителна проверка за това.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (i == j) continue;
```

```
        cout << "(" << i << "," << j << ")" << endl;
    }
}
```

Задача 10.2. Изведете на екрана всички ненаредени двойки с числа в интервала $[0; n-1]$, без повторения на едно число. Понеже двойките са ненаредени, ако изведете (a, b), не трябва отделно да извеждате (b, a).

Решение. Разликата с предната задача е, че тук не трябва да извеждаме една и съща двойка числа два пъти.
Един стандартен подход е да въведем наредба в двойката. Да вземем например двойките $(0, 1)$ и $(1, 0)$. От тези две двойки се интересуваме само от едната. В единия случай първото число в двойката е по-малко, в другия е по-голямо. Така може да гледаме само двойките където първото число е по-малко от второто. Това ще ни гарантира, че няма да имаме повторения. Така пак ще обходим всички възможни двойки, но ще извеждаме само тези, които трябва.

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (i >= j) continue;
        cout << "(" << i << "," << j << ")" << endl;
    }
}
```

Има и втори вариант, в които може да спестим обхождането на двойки, които не ни интересуват. Като отново ще използваме идеята за наредба в двойката, но по-умно. В първия цикъл фиксираме първия елемент от двойката i и търсим всички двойки, където втория елемент е по-голям. Ние знаем кои са тези елементи - $i+1, ..., n-1$. Така ако j започва от $i+1$, а не от 0, всъщност ще обходим само това което искаме.

```
for (int i = 0; i < n; i++) {
    for (int j = i+1; j < n; j++) {
        cout << "(" << i << "," << j << ")" << endl;
    }
}
```

Задача 10.3. Изведете на екрана всички ненаредени тройки с числа в интервала от $[0; n-1]$, като числата могат да се повтарят.

Решение. Това, че едно число може да се повтаря, не променя много нещата. Отново ключово за решението е въвеждане на наредба в тройката. Ще извеждаме (a, b, c) само когато $a \leq b \leq c$.
Първият вариант е да обхождаме всички тройки и да пропускаме ненужните:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
            if (i > j || j > k) continue;
```

```
            cout << "(" << i << "," << j << "," << k << ")" << endl;
        }
    }
}
```

Вторият вариант е да обхождаме само това което ни трябва:

```
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
        for (int k = j; k < n; k++) {
            cout << "(" << i << "," << j << "," << k << ")" << endl;
        }
    }
}
```

Задача 10.4. Дадена е редица от числа - `int a[n]`. Проверете съществува ли последователност от 1 или повече съседни числа със сума s.

Решение. Грубият подход е да намерим сумите на всички последователности и да проверим дали някоя е равна на s. Една последователност се определя от позициите на първия и последния елемент в нея. Така една двойка (i, j), където $i \leq j$ определя последователността $a[i], ..., a[j]$.

```
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {
    // проверяваме дали сумата на числата a[i]+...+a[j] е равна на s
    }
}
```

Задача 10.5. Дадена е редица от числа - `int a[n]`. Проверете дали всички числа в редицата са различни.

Решение. За да проверим дали всички са различни, грубият подход е да сравним всеки две числа - за всяка ненаредена двойка (i, j), проверяваме дали $a[i] \neq a[j]$.

```
bool areElementsUnique() {
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            if (a[i] == a[j]) {
                return false;
            }
        }
    }
    return true;
}
```

Задача 10.6. Дадена е редица от числа - `int a[n]`. Проверете дали има три числа от редицата, така че едното да е сума на другите две.

Решение. Може да обходим всички ненаредени тройки числа. И да проверяваме дали едно от тях е равно на сумата на другите две.

```
bool isOneSumOfTwo() {
    for (int i = 0; i < n; i++) {
      for (int j = i+1; j < n; j++) {
          for (int k = j+1; k < n; k++) {
              if (a[i]+a[j] == a[k] || a[i]+a[k] == a[j] || a[j]+a[k] == a[i]) {
                  return true;
              }
          }
      }
    }
    return false;
}
```

10.3 Груба сила с комбинации

Задача 10.7. Изведете на екрана всички редици с по 4 числа, съставени само от нули и единици.

Решение. Ще използваме четири вложени цикъла. Първият ще определя първото число, вторият - второто и т.н. Така управляващата променлива във всеки цикъл ще приема стойностите 0 и 1.

```
for (int i1 = 0; i1 < 2; i1++) {
    for (int i2 = 0; i2 < 2; i2++) {
        for (int i3 = 0; i3 < 2; i3++) {
            for (int i4 = 0; i4 < 2; i4++) {
                cout << i1 << " " << i2 << " " << i3 << " " << i4 << endl;
            }
        }
    }
}
```

Задача 10.8. Изведете на екрана всички редици с по 4 числа, съставени от числата от 0 до 3, които може да се повтарят.

Задача 10.9. Дадени са две големи раници. Трябва да сложим 4 предмета в тях, така че разликата в теглото на раниците да е минимална. Намерете тази разлика.

Решение. Първата стъпка е да открием всички възможности. За всеки предмет имаме два варианта - да отиде в първата или във втората раница.
Втората стъпка е да генерираме тези възможности. Ще номерираме двете раници с

1 и 2. Сега искаме на всеки предмет да съпоставим число 1 или 2 - съответстващо на раницата, в която ще отиде. Едно поставяне на предметите в раниците се определя от четири числа, които са 1 или 2. Например комбинацията $[2, 1, 2, 1]$ означава че първият и третият предмет отиват в раница 2, докато вторият и четвъртият в раница 1. Всички възможни поставяния се определят от всички четирицифрени редици с числата 1 и 2. Третата стъпка е да премахнем подредби, които не отговарят на условието, но тя не е приложима в тази задача - всяко разполагане на предметите в раниците е валидно. Последната стъпка е да определим, коя от тези подредби е най-добре балансирана.

```
#include<bits/stdc++.h>
using namespace std;

int a, b, c ,d; // теглата на четирите предмета

// тази функция приема конкретно разпределение на предметите по раниците
// i0 - в коя раница е първия предмет
// i1 - в коя раница е втория предмет
// i2 - в коя раница е третия предмет
// i3 - в коя раница е четвъртия предмет
// и връща като резултат разликата в теглата на двете раници
int calcDiff(int i0, int i1, int i2, int i3) {
    int t1 = 0; // теглото на първата раница
    int t2 = 0; // теглото на втората раница
    if (i0 == 1) t1 += a; // ако първият предмет е в първата раница
    else t2 += b; // ако първият предмет е във втората раница
    if (i1 == 1) t1 += b;
    else t2 += b;
    if (i2 == 1) t1 += c;
    else t2 += c;
    if (i3 == 1) t1 += d;
    else t2 += d;

    return abs(t1-t2); // връщаме разликата в теглата на двете раници
}

int main() {
    cin >> a >> b >> c >> d;

    int ans = a+b+c+d;

    // генерираме всички възможни подредби на предметите в раниците
    for (int i0 = 1; i0 <= 2; i0++) {
        for (int i1 = 1; i1 <= 2; i1++) {
            for (int i2 = 1; i2 <= 2; i2++) {
                for (int i3 = 1; i3 <= 2; i3++) {
                    int diff = calcDiff(i0, i1, i2, i3);
                    ans = min(ans, diff);
                }
```

```
            }
        }
    }
    cout << ans << endl;
    return 0;
}
```

Задача 10.10. Дадени са три предмета с тегла a, b и c и три раници, които побират максимално тегло m, n и p. Колко най-много предмета може да поберем в раниците, ако в една раница може да се сложи повече от един предмет.

Решение. Първата стъпка е да открием всички възможности. Много е важно да забележим, че не е задължително всички предмети да влязат в раница. Така за всеки предмет имаме четири варианта - да отиде в една от трите раници или да остане извън раница.
Втората стъпка е да генерираме тези възможности. Ще номерираме трите раници с 1, 2 и 3, и ще използваме числото 0, ако предмета е останал извън раниците. Сега искаме на всеки предмет да съпоставим число 0,1,2 или 3. Едно поставяне на предметите в раниците се определя от три числа, които са 0,1,2 или 3. Например комбинацията $[3, 0, 1]$ означава че първият предмет отиват в раница 3, третият в раница 1 и вторият остава извън раниците. Всички възможни поставяния се определят от всички трицифрени редици с числата 0,1,2 и 3. Третата стъпка е да премахнем подредби, които не отговарят на условието. Това са случаите, в които сме надвишили вместимостта на някоя от раниците. Последната стъпка е да определим, в коя от валидните подредби сме сложили най-много предмети по раниците. Ще напишем една функция, която проверява дали една подредба $(i0, i1, i2)$, където това са раниците на трите предмета е валидна и една функция, която проверява броя предмети, които са вътре в раница.

```
bool isValid(int i0, int i1, int i2) {
    int t1 = 0; // теглото на първата раница
    int t2 = 0; // теглото на втората раница
    int t3 = 0; // теглото на третата раница
    if (i0 == 1) t1 += a; // ако първият предмет е в първата раница
    if (i0 == 2) t2 += a; // ако първият предмет е във втората раница
    if (i0 == 3) t3 += a; // ако първият предмет е в третата раница
    if (i1 == 1) t1 += b;
    if (i1 == 2) t2 += b;
    if (i1 == 3) t3 += b;
    if (i2 == 1) t1 += c;
    if (i2 == 2) t2 += c;
    if (i2 == 3) t3 += c;

    return t1 <= m && t2 <= n && t3 <= p;
}

int itemsInside(int i0, int i1, int i2) {
    int ans = 0; // брой предмети вътре в раница
```

```
    if (i0 != 0) ans++; // ако първият предмет е в раница
    if (i1 != 0) ans++;
    if (i2 != 0) ans++;

    return ans;
}
```

Задача 10.11. Дадени са десет предмета с тегла $a_0, ..., a_9$ и три раници, които побират максимално тегло m, n и p. Колко най-много предмета може да поберем в раниците, ако в една раница може да се сложи повече от един предмет.

Решение. Задачата като идея и решение е същата като предишната. Трябва да направим 10 цикъла, с които да определим мястото на всеки от десетте предмета, да проверим дали конфигурацията е валидна и да преброим предметите вътре в раниците. Това което ще отбележим е, че може да използваме масиви. Ще сложим теглата на предметите в един масив `int a[10]` и техните позиции в друг масив - `int p[10]`, като `p[i]` ще показва в коя раница се намира i-ия предмет. Така няма да спестим многото вложени цикли, но функциите `isValid` и `itemsInside` ще станат доста по-приятни. Все пак трябва да внимаваме, че и в циклите има промяна. Вместо променливите да са $i0, i1, ...$ ще използваме $p[0], p[1], ...$ Така циклите ще имат подобен вид - `for(p[0] = 0; p[0] <= 3; p[0]++)`. Ето останалите подобрения:

```
bool isValid() {
    int t[4] = {}; // t[1], t[2], t[3] - теглата на раниците
    for (int i = 0; i < 10; i++) { // обхождаме всички предмети
        // i-ия предмет се намера в раница p[i]
        // към теглото на раница p[i], добавяме теглото на предмета
        t[p[i]] += a[i];
    }
    return t[1] <= m && t[2] <= n && t[3] <= p;
}

int itemsInside() {
    int ans = 0; // брой предмети вътре в раница
    for (int i = 0; i < 10; i++) {
        if (p[i] != 0) ans++; // ако i-ия предмет е в раница
    }

    return ans;
}
```

Задача 10.12. Шест човека трябва да прекосят реката с лодка. Колко най-малко килограма трябва да побира лодката, за да могат да преминат с най-много три возения.

Решение. Първата стъпка е да открием всички възможности. За всеки човек трябва да определим на кое возене ще премине - първо, втори или трето.

Втората стъпка е да генерираме тези възможности. Ясно е, че ще стане със шест вложени цикъла.
Третата стъпка е да премахнем подредби, които не отговарят на условието - тук такива няма.
Накрая трябва да намерим най-оптималното решение. За всяка подредба трябва да намерим най-тежкото от трите возения. От всички подредби търсим минималното.

Задача 10.13. n човека, между един и десет, трябва да прекосят реката с лодка. Колко най-малко килограма трябва да побира лодката, за да могат да преминат с най-много три возения.

Решение. Тук проблема идва от факта, че не знаем точния брой хора. Един вариант е да напишем отделни решения за всеки възможен брой, но това няма да е много приятно. Основният трик е, че може да генерираме всички възможности за най-лошия случай - десен човека. При самите проверки ще използваме първите n от генерираните числа. Понеже сме генерирали за десет, ще имаме повторения на някои от вариантите, но в случая това не е проблем.

10.4 Груба сила с пермутации

Дефиниция 10.2 (Пермутация)**.** Пермутация на числа наричаме всяка една тяхна подредба в редица. Всички възможни подредби дават всички пермутации на числата.

Задача 10.14. Кои са всички пермутации на числата 0, 1 и 2.

Решение. Пермутациите са $[0,1,2], [0,2,1], [1,0,2], [1,2,0], [2,0,1], [2,1,0]$.

Задача 10.15. Колко са различните пермутации с n числа.

Решение. Фиксираме числата в редицата едно по едно. За да изберем първото число имаме на разположение всичките n. След това за второто вече не може да използваме едно от тях - имаме $n-1$ варианта. И т.н. за всяко следващо имаме по 1 възможност по-малко. Общия брой е $n! = 1.2.3...n$.

Задача 10.16. Напишете програма, която извежда на екрана всички пермутации на числата от 0 до 3.

Решение. Имаме $4! = 24$ варианта. Все пак ще видим как може да ги генерираме с код. Логично е да фиксираме елементите един по един. Първо ще сложим този на първо място, после този на второ и т.н. Основния проблем е как да следим за повторенията.
Един вариант е когато слагаме числата да не проверяваме нищо и накрая като получим редица от 4 числа да проверим дали отговаря на условието. За да направим проверката трябва да сравним всяко с всяко и да види дали няма повтарящи се.

```
for (int i1 = 0; i1 < 4; i1++) {
    for (int i2 = 0; i2 < 4; i2++) {
        for (int i3 = 0; i3 < 4; i3++) {
            for (int i4 = 0; i4 < 4; i4++) {
                if (i1 == i2 || i1 == i3 || i1 == i4
                    || i2 == i3 || i2 == i4 || i3 == i4) continue;
```

```
                cout << i1 << " " << i2 << " " << i3 << " " << i4 << endl;
            }
        }
    }
}
```

Много добра оптимизация на този вариант е в момента, в който сложим дадено число да проверим дали вече не е използвано:

```
for (int i1 = 0; i1 < 4; i1++) {
    for (int i2 = 0; i2 < 4; i2++) {
        if (i2 == i1) continue;
        for (int i3 = 0; i3 < 4; i3++) {
            if (i3 == i1 || i3 == i2) continue;
            for (int i4 = 0; i4 < 4; i4++) {
                if (i4 == i1 || i4 == i2 || i4 == i3) continue;
                cout << i1 << " " << i2 << " " << i3 << " " << i4 << endl;
            }
        }
    }
}
```

Задача 10.17. Дадени са четири двуцифрени числа. По колко начина може да ги наредим в редица, така че числото което се образува като ги долепим да е кратно на 7.

Решение. Стъпките са ясни - гледаме всички пермутации на числата и за всяка проверяваме дали след долепването числото е кратно на 7.
Как да генерираме всички пермутации на произволни числа? Тук има добре познат трик, които ще припомним. Най-лесно е числата да са номерирани, за което помага ползването на масив - $a[0], a[1], a[2], a[3]$. По този начин е достатъчно да направим пермутациите на техните индекси. Така пермутацията $[3, 1, 0, 2]$ ще съответства на числата $a[3], a[1], a[0], a[2]$. Ако пермутациите ги пазим в друг масив - `int p[4]`, то $p[i]$ ще е позицията на числото $a[i]$. Сега остава като имаме числата и позициите им да ги долепим. Всъщност има още нещо - ние знаем позицията всяко число $a[i]$, но трябва да разберем кое число е на първа, втора, трета и четвърта позиция. Най-лесно е да използваме трети масив където да запазим числата в реда на пермутацията определена от масива p. Нека този масив е `int b[4]`, $b[i]$ ще бъде числото, което стои на i-та позиция. Да видим в масива b е мястото на числото $a[i]$. Позицията на $a[i]$ е $p[i]$. Значи $b[p[i]] = a[i]$. Ето как изглежда сливането ако имаме масивите a и конкретна пермутация p.

```
int calcMergedNumber() {
    int b[4] = {};  // Тук ще пазим числата в реда на пермутацията p
    for (int i = 0; i < 4; i++) {
        // Позицията на числото a[i] е p[i]
        // Запазваме, че на позиция p[i] се намира числото a[i]
        b[p[i]] = a[i];
```

```
    }

    // Долепваме числата в правилния ред
    return b[0]*1000000+b[1]*10000+b[2]*100+b[3];
}
```

10.5 Задачи

Задача 10.18 (Есенен Турнир 2017, E група, Карти).

Задача 10.19 (Есенен Турнир 2017, E група, Израз).

Задача 10.20 (Есенен Турнир 2018, E група, Дини).

Задача 10.21 (Есенен Турнир 2015, E група, Асансьор).

Задача 10.22. Намерете колко най-малко цифри трябва да изтрием от числото n, така че числото образувано от останалите цифри да е просто. Изведете -1 ако задачата няма решение.
Ограничения: $1 \leq n < 10^{10}$.

Задача 10.23. Дадени са n предмета, като всеки има тегло w_i и стойност v_i. Дадена е раница, която побират максимално тегло w. Каква е най-голямата обща стойност на предмети, които може да съберем в раницата.
Ограничения: $1 \leq n \leq 10, 1 \leq w \leq 1000, 1 \leq w_i, v_i \leq 100$.

Задача 10.24 (Зимен Турнир 2017, D група, Кратно на 3).

Задача 10.25 (Есенен Турнир 2018, D група, Промени числото).

Задача 10.26 (НОИ - Национален кръг 2020, D група, Фалшивата монета).

Глава 11

Теория на числата

11.1 Деление с частно и остатък

Теорема 11.1 (за деление с частно и остатък)**.** За всеки две цели числа a и b, $b \neq 0$, съществуват единствени цели числа q, наречено частно, и r, наречено остатък, такива че $a = q.b + r$ и $0 \leq r < |b|$.

Задача 11.1. Намерете r и q, за двойките (a, b) - $(3, 10)$, $(11, 3)$, $(5, -2)$, $(-7, 2)$, $(-11, -4)$.

Задача 11.2. Дадени са две цели положителни числа a и b. Нека $a = q.b + r$, където $0 \leq r < b$. Намерете r и q.

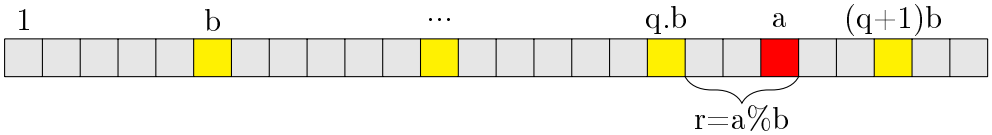
Решение. Това което търсим са частното и остатъка като делим a на b. В c++ при деление на положителни цели числа операциите / и % връщат точно това, като $r = a/b$ и $q = a\%b$. Така $a = (a/b).b + a\%b$.

Задача 11.3. Дадени са две цели числа a и b, $b \neq 0$. Нека $a = q.b + r$, където $0 \leq r < |b|$. Намерете q и r.

Решение. Ще отбележим, че целочисленото деление в c++ не винаги дава q и r. Така например при $a = -9, b = -2$, $-9 = 5(-2) + 1$ и $q = 5$, докато деленето в c++ ще вземе резултата $-9/-2 = 4.5$ и ще премахне дробната част, при което ще получим $-9/-2 = 4$. Делението с остатък в c++ работи така, че да е изпълнено $a = (a/b)b + r$, т.е. $r = a - (a/b)b$. Пробвайте да намерите формули за q и r чрез операциите в c++.

Задача 11.4. Дадени са две цели положителни числа a и b. Намерете най-голямото число, което е по-малко или равно на a и се дели на b.

Решение. Едно решение е да проверяваме последователно числата $a, a-1, ...,$ докато достигнем такова, което се дели на b.
Ако погледнем отново формулата $a = q.b + r$, то q представлява колко пъти b се съдържа в a. Така най-голямото число, което е кратно на b и по-малко или равно на a ще бъде $q.b$. Съответно може да получим резултата поне по два начина: $q.b = a - r = a - a\%b$ или $q.b = (a/b).b$.



Задача 11.5. Дадени са две цели положителни числа a и b. Намерете най-малкото число, което е по-голямо от a и се дели на b.

Решение. Търсим число, което е строго по-голямо от a. Ако $a = q.b+r$, понеже $q.b$ е най-голямото число кратно на b по-малко или равно на a, то ние търсим следващото кратно на b - $q.b+b$. Замествайки $q.b$ от предното решение имаме поне два варианта: $q.b + b = a - a\%b + b = a + b - a\%b$ или $q.b + b = (a/b).b + b = (a/b + 1).b$.

Задача 11.6. Дадени са две цели положителни числа a и b. Намерете най-малкото число, което е по-голямо или равно на a и се дели на b.

Решение. Разликата с предната задача е, че ако $b|a$ преди връщахме $a + b$ сега трябва да върнем a. Един вариант е да добавим един `if`. Все пак е интересно да решим задачата без проверка. За целта може да я сведем до предишната. Знаем как се решава задача за число $> a$ и кратно на b. Трябва да решим за $\geq a$ и кратно на b. Всъщност да търсим число $\geq a$, означава да търсим измежду числата $a, a + 1, ...$, което е еквивалентно на това да търсим число $> a - 1$. Така сегашната задача е еквивалентна на това да намерим най-малкото число по-голямо от $a - 1$ и кратно на b. За целта може да използваме решенията от предишната задача. Едно такова е $a - 1 + b - (a - 1)\%b$, другото е $((a - 1)/b + 1).b$.

Задача 11.7. Дадени са две цели положителни числа a и b. Намерете най-голямото число, което е по-малко от a и се дели на b. Решете задачата с директна формула без if.

11.2 Делители

Задача 11.8. Дадени са две цели положителни числа a и b. Проверете дали b дели a.

Решение. Ако разгледаме представянето $a = qb + r$, то за да е b делител на a, значи $a = qb$, т.е. $r = 0$. Така b е делител на a тогава и само тогава, когато $a\%b$ е 0.

Задача 11.9. Намерете броя на делителите на n.

Решение. Знаем, че делители на n могат да са само числа между 1 и n. Един вариант е да проверим за всяко от тях дали е делител или не. Ще имаме един брояч който ще го увеличаваме всеки път когато срещнем делител. Тази идея решава задачата, но ще работи доста бавно.
За да направим подобрение обаче ни трябва нещо повече, например да потърсим връзка между самите делители. Основното наблюдение е нещо сравнително очевидно. Ако намерим, че i е делител на n, то от $n = q.i$ следва, че и q е делител на n. Така с всеки един делител i ние също така намираме и друг $q = n/i$.
Така свеждаме задачата до това да намерим половината делители. Сега може пак да използваме идеята от бавното решение и да проверяваме за $i = 1, 2, 3, ...$ Само че вместо да проверим всичко до n, да се опитаме да спрем когато сме намерили половината делители. За целта ако $n = ab$ трябва да намерим тези два делителя, когато $i = a$ или $i = b$. Понеже i нараства последователно, то първо ще стигне до по-малкото от двете числа. Сега вече знаем кои делители търсим - за всяка двойка

делители $a, b (a \leq b)$ на n с произведение $a.b = n$, търсим по-малкият от двата - a. Остана да разберем кога сме намерили търсените делители. Трябва да обходим всички възможности за по-малкият делител - a. За големият делител имаме $b = n/a$. Това, че $a \leq b$, ни дава, че $a \leq n/a$, откъдето следва че $a.a \leq n$. Всъщност това ни дава кога да спрем проверките.
Сега остава да напишем програмата, като за всеки намерен малък делител, броим и по един голям, освен ако не са равни. Трябва да внимаваме за специалния случай, при който $a = b$ и тогава да броим само един делител. Това се случва само тогава когато $a.a = n$.

```
int countDivisors(long long n) {
    int ans = 0;
    for (long long i = 1; i*i <= n; i++) {
        if (n%i != 0) continue;

        if (i*i == n) {
            ans++;
        } else {
            ans += 2;
        }
    }
    return ans;
}
```

Задача 11.10. Ако n=*33*, колко възможности за по-малкият делител a, трябва да проверим?

Проверяваме докато $a.a \leq 33$. Знаем, че $5.5 = 25 \leq 33$, но $6.6 = 36 > 33$. Така достатъчно е да търсим по-малък делител измежду числата 1, 2, 3, 4 и 5.

Задача 11.11. Намерете всички делители на n.

11.3 Прости числа

Дефиниция 11.1. Просто число се нарича всяко положително число, по-голямо от 1, което има точно два положителни делителя – 1 и самото себе си.

Задача 11.12. Дадено е цяло положително число n. Проверете дали n е просто число.

Решение. Можем да използваме факта, че едно число е просто тогава и само тогава, когато има точно два делителя. Имайки вече функция, която намира броя делители проверката дали едно число е просто е лесна.

```
bool isPrime(long long n) {
    return divisors(n) == 2;
}
```

Разбира се това ще работи, но няма нужда да търсим всички делители за да проверим дали едно число е просто. Достатъчно е да намерим дали има поне един делител, различен от 1 и n. Ако n е съставно, искаме да намерим най-малкия делител различен от 1. Нека това е d. Тогава $n = d.(n/d)$. От $d \leq n/d$, следва че $d.d \leq n$. Значи е достатъчно да търсим делител от 2 докато $d.d \leq n$. Ако не намерим такъв делител, то n не е съставно. Все пак за да е просто, трябва да проверим, че не е по-малко от 2.

```
bool isPrime(long long n) {
    if (n < 2) return false;
    for (long long i = 2; i*i <= n; i++) {
        if (n%i == 0) return false;
    }
    return true;
}
```

Най-голямото разстояние между две съседни прости числа по-малки от един милион е 114. До август 2018 най-голямото намерено разстояние е 1550 и е след простото числото 18361375334787046697. Ето още няколко интересни за нас:

прости числа до	най-голямо разстояние
10^3	20
10^6	114
10^9	282
10^{12}	540
10^{15}	906

Задача 11.13. Дадено е цяло положително число n. Намерете най-малкото просто число, което е по-голямо от n.

Решение. Понеже простите числа са сравнително близо едно до друго е достатъчно да проверяваме последователно $n+1, n+2, ...$, докато намерим първото просто число.

Хипотеза 11.1 (на Голдбах)**.** Всяко цяло четно число по-голямо от две може да се представи като сума на две прости числа. Въпреки, че не е доказано за всички числа със сигурност е вярно за числата до 4.10^{18}.

Задача 11.14. Дадено е цяло положително число n по-голямо от 1. Представете n като сума на възможно най-малък брой прости числа, може и едно. При наличие на повече от едно решение намерете кое да е от тях.

11.4 Канонично разлагане на прости множители

Теорема 11.2. Всяко цяло положително число $n > 1$ може да се представи като произведение на прости множители, което е единствено с точност до реда на множителите. Това представяне се нарича канонично разлагане.

Обикновено за да бъде записването уникално подреждаме простите множители в нарастващ ред. Също така използваме степени, за да не повтаряме един и същ прост множител.

Задача 11.15. Докажете, че всяко съставно число $n > 1$ има прост делител p, такъв че $p.p \leq n$.

Ако има прост делител, който да изпълнява условието, това със сигурност ще е най-малкият прост делител на n. Нека p е най-малкият прост делител на n. Ще докажем, че $p.p \leq n$.
Тъй като n е съставно, то $1 < p < n$. Имаме, че $n = p.(n/p)$. Понеже p е най-малкият прост делител, то n/p ще има прост делител, който е по-голям или равен на p и значи $p \leq n/p$. Следователно $p.p \leq n$.

Задача 11.16. Дадени са число n и просто число p. Намерете степента на p в каноничното разлагане на n.

Трябва да проверим дали n дели $p, p^2, p^3, ...$ Проверяваме последователно докато стигнем до първата степен, която не е делител на n.

```
int maxPow(long long n, long long p) {
    int ans = 0;
    for (long long i = p; n%i == 0; i *= p) {
        ans++;
    }
    return ans;
}
```

Друг вариант да делим n на p, докато стигнем момента, в който не може да делим повече.

```
int maxPow(long long n, long long p) {
    int ans = 0;
    while (n%p == 0) {
        ans++;
        n /= p;
    }
    return ans;
}
```

Задача 11.17. Проверете дали n е p на някоя степен.

Докато n се дели на p, делим - $n = n/p$. Ако накрая n е 1, то значи е степен на p.

Задача 11.18. Намерете каноничното разлагане на дадено число n, като списък от прости множители.

Решение. Най-лесният вариант е да обикаляме по всички числа от 1 до n, и за всяко просто, което дели n да проверяваме колко пъти го има в каноничното разлагане.

```
vector<int> primeDecomposition(int n) {
    vector<int> ans;
    for (int i = 1; i <= n; i++) {
        if (n%i != 0) continue;
        if (!isPrime(i)) continue;
```

```
        for (int j = n; j%i == 0; j /= i) {
            ans.push_back(i);
        }
    }
    return ans;
}
```

Реда `if (n%i != 0) continue;` не е нужен, но е много полезен понеже ще спести проверката за всяко число дали е просто.
За да броим срещанията използваме копие на n, за да не променяме оригиналната стойност. Всъщност да видим какво ще стане ако за всеки намерен прост делител i, делим n на i, докато спре да се дели. Това означава, че когато стигнем до дадено i, то n няма да има прости делители по-малки от i, защото вече ще сме го разделили на тях. Всъщност сега проверката дали i е просто число става излишна. Ако стигнем до съставно число i, то със сигурност ще има просто число p, което дели i и $p < i$. Понеже $p < i$ следва, че p не дели n. И така няма как i да дели n.
Последно ще отбележим, че в този вариант ще трябва да започваме цикъла с възможни делители от 2. Програмата добива следния вид:

```
vector<int> primeDecomposition(int n) {
    vector<int> ans;
    for (int i = 2; i <= n; i++) {
        while (n%i == 0) {
            ans.push_back(i);
            n /= i;
        }
    }
    return ans;
}
```

Сега програмата изглежда много по-добре. Относно бързината може да кажем, че най-лошият случай е n да е просто число и тогава ще обходим всички числа от 2 до n. Да видим какво става в противен случай - ако n не е просто. Със сигурност n ще има прост делител p, такъв че $p * p <= n$, т.е. ако n е съставно, ние можем да намерим поне един прост делител обхождайки докато $p * p \leq n$. Тогава ще разделим n на p и ако n е съставно пак ще намерим прост делител по този начин. Така докато n е съставно ще намираме прост делител. Когато излезем от цикъла n ще бъде или просто число, или 1. Ако не е 1, трябва да добавим още един прост делител.

```
vector<long long> primeDecomposition(long long n) {
    vector<long long> ans;
    for (long long i = 2; i*i <= n; i++) {
        while (n%i == 0) {
            ans.push_back(i);
            n /= i;
        }
    }
    if (n != 1) {
        ans.push_back(n);
```

```
    }
    return ans;
}
```

Задача 11.19. В кода по-горе, за какви първоначални стойности на n условието след цикъла ще бъде изпълнено - `n != 1`?

Задача 11.20. Дадено е цяло число $n > 1$. Намерете най-малкият прост делител p на n?

Решение. Функцията за канонично разлагане намира всички прости делители в нарастващ ред. Достатъчно е да използваме същата логика, но в първия момент, в който намерим делител може да спрем и да го върнем като резултат.

```
long long minPrimeDivisor(long long n) {
    for (long long i = 2; i*i <= n; i++) {
        if (n%i == 0) return i;
    }
    return n;
}
```

Задача 11.21. Дадено е цяло число $n > 1$. Намерете най-малкият прост делител p на n?

Решение. Отново може да тръгнем от функцията за канонично разлагане с тази разлика, че сега трябва да минем през всички прости делители и да запазим най-големия. Друг вариант е да върнем този прост делител, при който n става равно на 1.

```
long long maxPrimeDivisor(long long n) {
    for (long long i = 2; i*i <= n; i++) {
        while (n%i == 0) {
            n /= i;
        }
        if (n == 1) return i;
    }
    return n;
}
```

11.5 Числа с даден брой делители

Задача 11.22. Кои числа имат нечетен брой делители?

Тази задача лесно може да решим ако намерим броя делители на първите няколко числа. Така ще забележим, че числата с нечетен брой делители са 1, 4, 9, 16,... Сега вече може да направим предположение, че това са всички числа, който са произведение на едно число със себе си - 1.1, 2.2, 3.3, 4.4,...
Друг начин да забележим това е да си припомним, че едно число се разбива на двойки делители - $a.b = n$, по подобие на бързия алгоритъм за намиране на брой делители. Така всяка двойка дава два различни делителя, с едно единствено изключение - ако a е равно на b. Така четната бройка може да бъде нарушена единствено ако има число, което умножено по себе си дава n.

Задача 11.23. Ако каноничното разгалане на $n = p_1^{x_1}.p_2^{x_2}...p_k^{x_k}$, то броят на делителите на n е равен на $(x_1 + 1).(x_2 + 1)...(x_k + 1)$.

Всеки делител на n има вида $p_1^{y_1}.p_2^{y_2}...p_k^{y_k}$, като $0 \leq y_1 \leq x_1$, $0 \leq y_2 \leq x_2$, ..., $0 \leq y_k \leq x_k$. Така всеки делител е произведение на k числа, като за всеки от множителите имаме съответно x_1+1, x_2+1, ..., x_k+1 опции. Това прави точно $(x_1+1).(x_2+1)...(x_k+1)$ различни варианта.

Задача 11.24. Колко делителя има 12.

Каноничното разлагане на 12 е $2^2.3^1$. Броят делители според формулата е $(2 + 1)(1 + 1) = 3.2 = 6$.

Задача 11.25. Колко делителя има 90.

Каноничното разлагане на 90 е $2^1.3^2.5^1$. Броят делители според формулата е $(1+1)(2 + 1)(1 + 1) = 2.3.2 = 12$.

11.6 Реше на Ератостен

Задача 11.26. Намерете броят на делителите на всички числа от едно до n, за $n \leq 10^6$?

Решение. Стандартния подход към задачата е за всяко число поотделно да намерим колко делители има. Когато търсим делителите на едно число въртим цикъл по възможните делители и на всяка стъпка проверяваме дали текущото число е делител. Това води като резултат до много излишни проверки на числа, които всъщност не са делители. Ако гледаме числата независимо няма как да спестим тези излишни проверки.
Може да използваме факта, че имаме много последователни числа. Всъщност е нужно да сменим гледната точка - трябва да обърнем стандартния подход. Вместо за всяко число да търсим кои са му делители, ще видим за всеки делител, кои числа дели. Така за всеки възможен делител i от 1 до n, ще гледаме кои числа се делят на i. Това са - $i, 2i, 3i, ...$, докато не надминем n. За всяко от тези числа ще добавим по един делител - i. Това ще ни спести излишните проверки при стандартния подход.
Да разгледаме по-подробно как ще работи алгоритъма за $n = 25$. В началото за всички числа от 1 до 25, имаме по 0 намерени делителя.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

На първата стъпка ще видим кои числа имат за делител числото 1 - това са всички числа. Към броя на делителите на всяко число ще прибавим единица - намереният делител 1.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

На втората стъпка ще видим кои числа имат за делител числото 2 - това са всички четни числа. Към броя на делителите на всяко четно число ще прибавим единица - намереният делител 2.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1

На третата стъпка ще видим кои числа имат за делител числото 3 и ще увеличим броя им делители с единица.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	1	2	2	2	1	3	1	2	2	2	1	3	1	2	2	2	1	3	1	2	2	2	1	3	1

Продължавайки така на последната 25та стъпка ще добавим единица към единственото число кратно на 25, за да получим крайния резултат.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	1	2	2	3	2	4	2	4	3	4	2	6	2	4	4	5	2	6	2	6	4	4	2	8	3

Да видим как ще изглежда самата имплементация:

```
vector<int> sieve(int n) {
    vector<int> br(n+1);
    for (int i = 1; i <= n; i++) {
        for (int j = i; j <= n; j+=i) {
            br[j]++;
        }
    }
    return a;
}
```

Използваме масив или вектор, като стойността на елемента $br[i]$ ще бъде броя делители на i.
За всеки възможен делител i, j обхожда всички числа кратни на i - $i, 2i, 3i, ...$, без да надвишаваме n.

Задача 11.27. Ако извикаме $sieve(100)$, то колко пъти ще увеличим $br[100]$? За кои стойности на i ще стане това.

Решение. Ще увеличим $a[100]$ 9 пъти, по веднъж за всеки делител на 100 - 1, 2, 4, 5, 10, 20, 25, 50, 100.

Задача 11.28. Намерете броят на простите делители на всички числа от едно до n, за $n \leq 10^6$?

Решение. Може да използваме решето, като в масива `br[i]` ще прибавяме едно за всеки прост делител на i. Пак ще обхождаме по възможните делители, но сега ще отбелязваме, че числото i е делител на $i, 2i, 3i, ...$ само когато i е просто число. Идеята е добра, но доста важно се оказва как всъщност проверяваме дали i е просто.

Един вариант е до викаме функция, която проверява дали i е просто.
Да вникнем малко в това решение. За $i = 1$, няма да отбелязваме нищо понеже 1 не е просто. За $i = 2$ ще отбележим, по 1 делител за $2, 4, 6, ...$ За $i = 3$ ще отбележим, по 1 делител за $3, 6, 9, ...$ За $i = 4$, няма да отбелязваме нищо понеже 4 не е просто. Нека продължаваме така и дойде ред да отбелязваме с някое число i. Да видим каква е стойността на `br[i]` преди да почнем да отбелязваме с i. `br[i]` сме го увеличили с едно за всеки прост делител на i, по-малък от i. Всъщност `br[i]` ще е 0, тогава и само тогава, когато i е просто. Това наблюдение ни дава възможност вместо да правим допълнителна проверка дали i е просто, да проверим дали `br[i]==0`. Разбира се трябва да започнем външния цикъл от $i = 2$.

```
vector<int> sieve(int n) {
    vector<int> br(n+1);
    for (int i = 2; i <= n; i++) {
        if (br[i] != 0) continue;

        for (int j = i; j <= n; j+=i) {
            br[j]++;
        }
    }
    return a;
}
```

Задача 11.29. Намерете най-малкия прост делител на всички числа от 2 до n, за $n \leq 10^6$?

Решение. В предишната за всяко число минаваме толкова пъти, колкото прости делители има. Достатъчно е при първото минаване да отбележим, че намерения делител е търсения.

Задача 11.30. Дадено е цяло число $n \leq 10^6$. За всяко число между 2 и n намерете броя на множителите в каноничното му разлагане?

Решение. Един вариант е да търсим за всяко число поотделно, но това ще е бавно.
Нека `br[i]` е броят множители в каноничното разлагане на i. Основната идея за оптимизация идва от това, че ако имаме един прост делител на i, например p, то $i = p.(i/p)$. Каноничното разлагане на i го разглеждаме като p умножено с каноничното разлагане на i/p. Така броя делители в каноничното разлагане на i е равен на 1 плюс броя делители в каноничното разлагане на i/p, т.е. `br[i]=1+br[i/p]`. Сега ако сме сметнали отговора за числата преди i, то ще имаме `br[i/p]` и ще може да приложим формулата. За да сме сигурни, че сме сметнали отговора за всички числа до i, трябва да смятаме последователно за $i = 1, 2, ...$ и да попълваме `br[i]` по формулата. Остава за всяко i да намерим прост делител p, но за това може да използваме реше с най-големия или най-малкия прост делител.

Задача 11.31. За всяко число от 1 до n, намерете дали е просто или не, при $n \leq 10^6$?

Сега имаме поне два варианта наум. Първият е да обходим всички числа едно по едно и да извикаме функцията *isPrime(i)*. Вторият е да използваме функцията *sieve*(n), след което, знаем че простите числа ще имат точно по два делителя, т.е. i ще е просто, ако $a[i] == 2$.
Вторият вариант е доста добър. За това няма да правим нещо коренно различно, а само ще видим как може да подобрим идеята с мисълта, че точния брой делители не ни интересува. В действителност за всяко число трябва да знаем дали има делител различен от 1 и самото число. От тук идва и първата идея за подобрение. В началото приемаме, че всички числа от 2 до n са прости. Сега за всеки възможен делител $i > 1$, разглеждаме всички числа кратни на i, освен самото число i - $2i, 3i, ...$ Всички тези числа ги маркираме като съставни. Това няма да даде кой знае какво подобрение, но да видим как ще изглежда като код:

```
vector<bool> sieve(int n) {
    vector<bool> a(n+1, true);
    a[0]=false;
    a[1]=false;
    for (int i = 2; i <= n; i++) {
        for (int j = i+i; j <= n; j+=i) {
            a[j]=false;
        }
    }
    return a;
}
```

Всяко съставно число ще го маркираме с всички делители различни от 1 и самото число. Ако едно число е съставно, то със сигурност има прост делител. Вместо да го маркираме с всички делители, може да го правим само с прости такива. Когато започнем да маркираме всички числа, кратни на i, ние вече знаем дали i е просто или не. Това е така понеже i, няма делители по-големи от i, а с по-малките маркирането вече е направено. Така ако i е съставно, т.е. $a[i]$ е $false$, няма нужда да маркираме нищо. С тази малка промяна спестяваме доста ненужни операции. Ето новият код:

```
vector<bool> sieve(int n) {
    vector<bool> a(n+1, true);
    a[0]=false;
    a[1]=false;
    for (int i = 2; i <= n; i++) {
        if (!a[i]) continue;

        for (int j = i+i; j <= n; j+=i) {
            a[j]=false;
        }
    }
    return a;
}
```

Сега всяко съставно число, ще го маркираме с всички прости делители. Достатъчно е да осигурим маркиране с поне едно просто число, например с най-малкият прост делител. Нека съставното число n го маркираме с неговия най-малък прост делител

i. Тогава $n = i.(n/i)$. Ако $n/i < i$, то n/i ще има прост делител, които е по-малък от i. Следователно $n/i \geq i$, т.е. $i.i \leq n$. Маркирането с най-малък прост делител води до значително олекотяване на външния цикъл. Да видим пак кода:

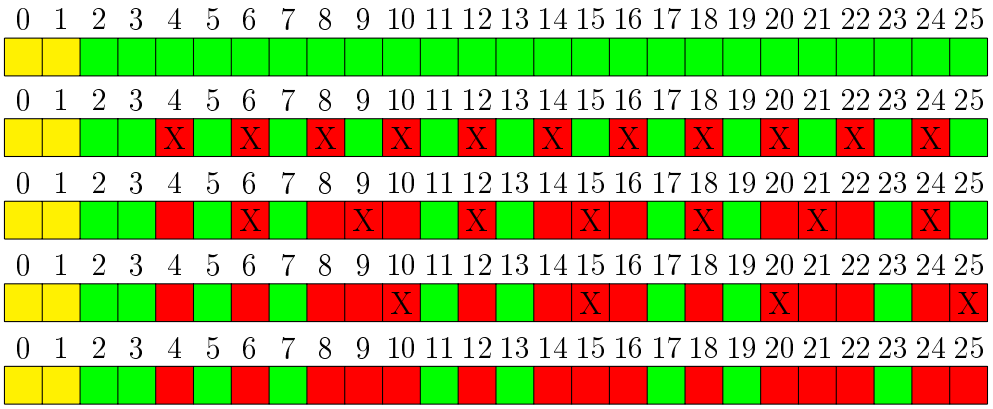
```
vector<bool> sieve(int n) {
    vector<bool> a(n+1, true);
    a[0]=false;
    a[1]=false;
    for (int i = 2; i*i <= n; i++) {
        if (!a[i]) continue;

        for (int j = i+i; j <= n; j+=i) {
            a[j]=false;
        }
    }
    return a;
}
```

Последното нещо, което ще оптимизираме е началото на вътрешния цикъл. С i, маркираме числата $2i, 3i, ...$ Обаче $2i$ има прост делител 2 и значи ще го макираме с него. $3i$ ще го маркираме с 3. Всички числа $2i, 3i, ..., (i-1).i$, ще имат прост делител по-малък от i, т.е. няма нужда да бъдат маркирани с i, тъй като вече ще са маркирани. Така първото число, което има смисъл да маркираме с i ще бъде $i.i$. С него ще започнем вътрешния цикъл, за да стигнем до последната подобрена версия:

```
vector<bool> sieve(int n) {
    vector<bool> a(n+1, true);
    a[0]=false;
    a[1]=false;
    for (int i = 2; i*i <= n; i++) {
        if (!a[i]) continue;

        for (int j = i*i; j <= n; j+=i) {
            a[j]=false;
        }
    }
    return a;
}
```



```
vector<int> sieve(int n) {
    vector<bool> a(n+1, true);
    vector<int> p;
    for (int i = 2; i <= n; i++) {
        if (!a[i]) continue;

        p.push_back(i);
        for (int j = i+i; j <= n; j+=i) {
            a[j]=false;
        }
    }

    return p;
}
```

11.7 НОД

Задача 11.32. Намерете НОД на две числа A и B?

Трябва да намерим най-голямото число което дели и двете дадени числа. Може с едно обхождане по всички възможни делители да намерим най-големия. За да е делител едно число едновременно на A и B, то трябва да е по-малко или равно на по-малкото. Ако има няколко възможни делители ни интересува най-големия затова можем да обхождаме отзад напред и в първия момент в който намерим делител спираме.

```
int gcd(int a, int b) {
    for (int i = min(a, b); i >= 1; i--) {
        if (a%i == 0 && b%i == 0) return i;
    }
}
```

Въпреки малките подобрения тази програма може да обходи всички числа от 1 до по-малкото, което ще има линейна сложност.

Задача 11.33. Нужно ли е да проверяваме всички числа от едно до по-малкото за възможни делители?

Вече знаем, че да намерим делителите на едно число имам много по-бърз алгоритъм. И тук може да приложим подобна оптимизация като обходим всички възможни делители на едното число и проверим кои от тях са делители на второто.

```
int gcd(int a, int b) {
    if (a > b) swap(a, b);

    int nod = 1;
    for (int i = 1; i*i <= a; i++) {
        if (a%i != 0) continue;
```

```
        if (b%i == 0) nod = max(nod, i);
        if (b%(a/i) == 0) nod = max(nod, a/i);
    }

    return nod;
}
```

Задача 11.34. Вярно ли е, че НОД(a, b) = НОД(a, a-b)?

11.8 НОК

11.9 Дроби

Задача 11.35. Как можем да пазим дроби в компютъра?

Една дроб се състои от числител и знаменател. Следователно за да пазим дроби в компютъра трябва да пазим две числа. Един вариант е да пазим две променливи, но може би още по-добър вариант е да използваме *pair*. Така пак ще пазим две числа, но ще имаме само една променлива и по-лесно ще знаем, че двете числа имат някаква връзка.
Първата версия на метода за създаване на дроб ще изглежда така:

```
pair<int, int> fraction(int a, int b) {
    return {a, b};
}
```

Задача 11.36. Как можем да определим знака на една дроб?

За да определим знака на дроб трябва да проверим знаците на числителя и знаменателя. Ако те са равни, то знака на дробта е плюс, ако са различни - минус. За улеснение обаче ще направим нещо малко по-различно. Ще пазим знака само в числителя. Ако в даден момент искаме да създадем нова дроб, в която знаменателя е отрицателен, ще умножаваме и числителя и знаменателя по -1.

Новата втора версия на метода за създаване на дроб ще изглежда така:

```
pair<int, int> fraction(int a, int b) {
    if (b < 0) {
        a = -a;
        b = -b;
    }
    return {a, b};
}
```

Задача 11.37. Как можем да проверим дали две дроби да равни?

За целта искаме просто да можем да сравним двата *pair*-а. Проблема обаче е че така 1/2 ще е различно от 2/4. Един вариант е всеки път преди да сравняваме да съкращаваме дробите. Вместо това обаче всеки път когато създаваме дроб ние ще я съкращаваме още в началото. Така ще пазим само съкратени дроби и ще може да сравняваме без проблеми.

Финалната трета версия на метода за създаване на дроб ще изглежда така, като сме си написали *gcd(a, b)* да намира най-големия общ делител на a и b:

```
pair<int, int> fraction(int a, int b) {
    int d = gcd(abs(a), abs(b));
    a /= d;
    b /= d;
    if (b < 0) {
        a = -a;
        b = -b;
    }
    return {a, b};
}
```

Задача 11.38. Напишете функция, която събира две дроби?

За да съберем две дроби ще използваме познатия метод с привеждане към общ знаменател. Един вариант е да намерим НОК на двата знаменателя и да пресметнем с колко трябва да разширим всяка от двете дроби. Ние обаче ще си спестим смятането на НОК и ще използваме за общ знаменател произведението на двата знаменателя.
Да разгледаме пример как събираме две дроби с намиране на НОК. Да съберем *1/6+1/9*. Първо за да намерим общ знаменател намираме *НОК(6, 9) = 18*. Така първата дроб 1/6-та ще я разширим с 18/6=3, за да получим 3/18. Аналогично втората я разширяваме с 18/9=2 и получаваме 2/18. Накрая събираме 3/18+2/18=5/18. Резултатната дроб е несъкратима и следователно това е краяйният резултат.
Алтернативния вариант вместо да търсим НОД(6, 9) е да вземем произведението 6.9=54. Така първата дроб 1/6-та е ясно че трябва да я разширим със знаменателя на втората - 9, за да получим 9/54. Аналогично втората я разширяваме със знаменателя на първата - 6 и получаваме 6/54. Резултата определяме от сбора 9/54+6/54=15/54. Този резултат е различен от предходния защото тази дроб можем да я съкратим. В случая НОД(15, 54) = 3, следователно разделяме числителя и знаменателя на 3 за да получим 5/18.
На пръв поглед съкращаване се налага само във втория случай, но ако разгледате например дробите a/6+c/9=(3a+2c)/18

```
pair<int, int> sum(pair<int, int> a, pair<int, int> b) {
    return {a.first*b.second+a.second*b.first, b.first*b.second};
}
```

11.10 Задачи

Задача 11.39 (НОИ - Национален кръг 2017, Е5, Степени на двойката).

Задача 11.40 (НОИ - Областен кръг 2019, Е2, Нечетен).

Задача 11.41 (НОИ - Областен кръг 2016, Е2, Правоъгълници).

Задача 11.42 (НОИ - Национален кръг 2012, D2, Сума).

Задача 11.43 (Пролетен турнир 2008, D1, Лампи).

Задача 11.44 (Есенен турнир 2009, D1, Правоъгълници).

Задача 11.45 (Есенен турнир 2016, D2, Брой делители).

Задача 11.46 (Пролетен турнир 2016, D3, Пет делителя).

Задача 11.47 (НОИ - Национален кръг 2010, D5, Последна цифра).

Задача 11.48 (НОИ - Национален кръг 2011, D3, Лотария).

Задача 11.49 (НОИ - Национален кръг 2011, D6, Супер прости).

Задача 11.50 (НОИ - Национален кръг 2019, D1, Решето на Ератостен).

Задача 11.51 (НОИ - Общински кръг 2013, D2, Портокали).

Задача 11.52 (НОИ - Областен кръг 2019, D3, Квадрат).

Задача 11.53 (НОИ - Областен кръг 2014, D3, Египетски дроби).

Задача 11.54 (НОИ - Национален кръг 2014, D2, Агитки).

Задача 11.55. Дадено е цяло положително число n, $2 \leq n \leq 10^9$. Нека p е най-голямото просто число, което е делител на n. Намерете най-малкото число в редицата $n, n+p, n+2p, ...$, което има прост делител по-голям от p.

Глава 12

Бройни системи

12.1 Бройни системи

Бройните системи са метод за представяне на числата чрез символи, наречени цифри. Бройните системи се разделят на два вида - непозиционни и позиционни.
Непозиционните бройни системи са тези, при които стойността на цифрата най-общо не зависи от нейното място(позиция) в записа на числото. Известен пример за такава бройна система е римската. В римската бройна система някои от използваните цифри са (1000), D(500), C(100), L(50), X(10), V(5), I(1) и действа правилото: когато тези цифри са написани в намаляващ ред на стойностите им, те се събират, а когато по-малък числов знак стои пред по-голям, те се изваждат – например: $VI = 5+1 = 6$, $IV = 5 - 1 = 4$.
Позиционните бройни системи са тези, при които стойността на цифрата зависи от нейното място(позиция) в записа на числото, като тя се умножава с т.нар. тегловен коефициент. Той представлява основата на бройната система (например 2, 10 или 16), повдигната на различна степен: нула – за най-младшия разряд, единица за следващия и т.н. – степента нараства с единица за всеки следващ по-старши разряд („наляво“).
В десетичната бройна система за основа се използва числото 10. Затова цифрите на числото се умножават по степените на 10, като последната цифра се умножава по 10^0, предпоследната по 10^1 и т.н. Например числото 1234 се записва като $1234 = 1.10^3 + 2.10^2 + 3.10^1 + 4.10^0$. Това се нарича разширен запис на числото 1234.

12.2 Преобразуване

Задача 12.1. Колко цифри се използват за изписване на числата в десетична бройна система?

Разбира се използваме 10 цифри това са 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Задача 12.2. Колко цифри има в N-ична бройна система? Какви се техните стойности?

Аналогично на десетичната, в N-ична бройна система има N цифри. Техните стойности са числата от 0 до N-1.

Задача 12.3. Как може да има повече от десет цифри в някоя бройна система?

Цифрите в N-ична бройна система всъщност не са нищо повече от символи, на които даваме стойности числата от 0 до N-1. Понеже използваме десетична бройна система, има десет стандартни символа за стойностите от 0 до 9. Ако ни трябват повече просто ще си харесаме някакви символи и ще им дадем стойности от 10 до N-1. В практиката след цифрите, започваме да използваме латински букви.

Задача 12.4. Кои са цифрите в шестнадесетична бройна система?

Това са познатите 10 и първите 6 букви от латинската азбука - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Задача 12.5. Каква е десетичната стойност на цифрата E в шестнадесетична?

Десетичните стойности на буквите започват от 10, което съответства на A. На E съответства числото 14.

Задача 12.6. Да разгледаме числото 79763 в десетична бройна система. Какъв е разширеният му запис?

$79763 = 7.10^4 + 9.10^3 + 7.10^2 + 6.10^1 + 3.10^0$.

Задача 12.7. Да разгледаме числото 11101 в двоична бройна система. Какъв е разширеният му запис?

Понеже числото е в двоична бройна система неговите цифри ще умножаваме по степените на двойката - 1, 2, 4, 8 като започнем с поседната. Разширеният запис ще е $11101 = 1.2^4 + 1.2^3 + 1.2^2 + 0.2^1 + 1.2^0$.

Задача 12.8. В каква бройна система е разширеният запис, който намерихме $11101 = 1.2^4 + 1.2^3 + 1.2^2 + 0.2^1 + 1.2^0$? Двоична ли е тя?

Отговорът е отрицателен. В двоична бройна система има само цифрите 0 и 1, но в разширения запис не използваме само тях. Цифрите от числото са изписани както са си в бройната система. Обаче за двойката и нейните степени по навик използваме добре познатите числа от десетична бройна система.

Задача 12.9. Как е две в двоична бройна система?

Отговор - $2_{(10)} = 10_{(2)}$.

Задача 12.10. Кои са първите 8 числа в двоична бройна система?

Това са всички числа с по 1, 2 и 3 цифри - 0, 1, 10, 11, 100, 101, 110, 111.

Задача 12.11. Кои са първите степени на двойката записани в двоична бройна система - 2^0, 2^1, 2^2, 2^3, 2^4?

Това са числата, в които цифрата едно се среща веднъж в началото - $2^0_{(10)} = 1(2)$, $2^1_{(10)} = 10(2)$, $2^2_{(10)} = 100(2)$, $2^3_{(10)} = 1000(2)$, $2^4_{(10)} = 10000(2)$.

Задача 12.12. Да разгледаме отново числото 1101 в двоична бройна система. Какъв е разширеният му запис изцяло в двоична бройна система?

В намерения вече запис $1101 = 1.2^4 + 1.2^3 + 1.2^2 + 0.2^1 + 1.2^0$, трябва да заместим всички числа, които не са в двоична бройна система с техните еквиваленти. $1101 = 1.10^{10000} + 1.10^{1000} + 1.10^{100} + 0.10^{10} + 1.10^1$.

Задача 12.13. Как може да превърнем едно число от N-ична бройна система в 10-ична?

Може би сте го забелязали при задачите за разширения запис. Като изпишем разширения запис с числа в десетична $\overline{A_0A_1..A_{m(K)}} = A_0.K^m + A_1.K^{m-1} + ... + A_m.K^0$ остава само да извършим аритметичните операции също десетична бройна система и ще получим крайния резултат.

Задача 12.14. Напишете програма която да преобразува число N от дадена бройна система в десетична.

Трябва да пресметнем $\overline{A_0A_1..A_{m(K)}} = A_0.K^m + A_1.K^{m-1} + ... + A_m.K^0$:

```
long long toDecimal(string s, int K) {
  long long N = 0;
  for (int i = 0; i < s.size(); i++) {
    long long weight = pow(K, s.size()-i-1) + 0.5;
      if (isdigit(s[i])) N += (s[i]-'0') * weight;
      else N += (s[i]-'A'+10) * weight;
  }
  return ans;
}
```

Всъщност може да обхождаме отзад напред и всеки път да си променяме теглото вместо са използваме *pow*. Така първия път имаме $weight = K^0$, после $weight = K^1$ и т.н. Получаваме доста по-добър вариант:

```
long long toDecimal(string s, int K) {
    long long N = 0, weight = 1;
    for (int i = s.size()-1; i >= 0; i++) {
        if (isdigit(s[i])) N += (s[i]-'0') * weight;
        else N += (s[i]-'A'+10) * weight;
        weight *= K;
    }
    return ans;
}
```

Има още един начин, който може да ви се стори по-сложен в началото. Идеята е да смятаме числото постепенно. Първо ще пресметнем числото $\overline{A_{0(K)}}$, след това $\overline{A_0A_{1(K)}}$, после $\overline{A_0A_1A_{2(K)}}$, ... и накрая $\overline{A_0A_1..A_{m(K)}}$. Сега остава да видим как от едното получаваме следващото. $\overline{A_0A_{1(K)}} = k.\overline{A_{0(K)}} + A_1$, $\overline{A_0A_1A_{2(K)}} = k.\overline{A_0A_{1(K)}} + A_1$. Всеки път резултата до момента го умножаваме по K и прибавяме следващия коефициент. Следва имплементация на този алгоритъм:

```
long long toDecimal(string s, int K) {
  long long N = 0;
  for (int i = 0; i < s.size(); i++) {
    N = N*K+s[i]-'0';
  }
  return N;
}
```

И накрая ако използваме съкратения *for* функцията добива най-кратък вид:

```
long long toDecimal(string s, int K) {
  long long N = 0;
  for (char c: s) {
    N = N*K+c-'0';
  }
  return N;
}
```

Задача 12.15. Как може да превърнем числото 21 от десетична бройна система в двоична?

Нека запишем представянето на 21 в двоична по следния начин $21_{(10)} = \overline{A_0 A_1 .. A_m}_{(2)}$. За да намерим отговора трябва да решим уравнението $21 = A_0.2^m + A_1.2^{m-1} + ... + A_{m-1}.2^1 + A_m.2^0$, където $0 \leq A_i \leq 1$.
Числото, в лявата част на уравнението, 21 е нечетно. Всички събираеми, в дясната част, без последното са четни, понеже имат множител 2. Така дали дясната страна на равенството е четна или нечетна зависи от последното събираемо. В случая искаме $A_m.2^0 = A_m$ да е нечетно. От вариантите, които имаме - 0 и 1, само 1 е нечетно, следователно $A_m = 1$. Така получаваме $21 = A_0.2^m + A_1.2^{m-1} + ... + A_{m-1}.2^1 + 1$, като извадим едно $20 = A_0.2^m + A_1.2^{m-1} + ... + A_{m-1}.2^1$ и разделим на две получаваме $10 = A_0.2^{m-1} + A_1.2^{m-2} + ... + A_{m-1}.2^0$.
Продължаваме по същия начин, като в случая 10 е четно, следователно дясната страна също трябва да е четна, т.е. $A_{m-1}.2^0 = A_{m-1}$ и значи $A_{m-1} = 0$. Заместваме в уравнението $10 = A_0.2^{m-1} + A_1.2^{m-2} + ... + 0$, разделяме на две $5 = A_0.2^{m-2} + A_1.2^{m-3} + ... + A_{m-2}.2^0$.
Продължаваме така докато в лявата страна на уравнението не получим нула. Това което всъщност правим е първо да определим последния неизвестно, като гледаме остатъка на лявата част при деление на две. След това го елиминираме и делим на две уравнението. Вляво всъщност получаваме старата стойност разделена на две. Довършваме задачата като в момента вляво имаме 5 и търсим A_{m-2}. Значи $A_{m-2} = 5\%2 = 1$ и след преобразуванията вляво ще имаме $\lfloor 5/2 \rfloor = 2$ и ще търсим A_{m-3}. Така $A_{m-3} = 2\%2 = 0$ и вляво остава $\lfloor 2/2 \rfloor = 1$. $A_{m-4} = 1\%2 = 1$, вляво остава $\lfloor 1/2 \rfloor = 0$ и така търсенето завършва. A_{m-4} е първата цифра и всъщност това е A_0, т.е $m - 4 = 0$ и $m = 4$. Коефициентите, които намерихме са $A_0 = A_{m-4} = 1$, $A_1 = A_{m-3} = 0$, $A_2 = A_{m-2} = 1$, $A_3 = A_{m-1} = 0$, $A_4 = A_m = 1$. Замествайки в първоначалното уравнение $21 = 1.2^4 + 0.2^3 + 1.2^2 + 0.2^1 + 1.2^0$. Така 21 в двоична бройна система е равно на 10101.

Задача 12.16. Как може да превърнем едно число N от 10-ична бройна система в K-ична?

Нека запишем представянето на N в K-ична по следния начин $N_{(10)} = \overline{A_0 A_1 .. A_m}_{(K)}$. Трябва да решим уравнението $N = A_0.K^m + A_1.K^{m-1} + ... + A_{m-1}.K^1 + A_m.K^0$, където $0 \leq A_i < K$.
Идеята е същата като в предната задача. И двете страни трябва да дават един и същ остатък при деление на K. Т.е. $N\%K = A_m\%K$, но понеже $0 \leq A_m < K$, то $A_m\%K = A_m$ и така $A_m = N\%K$. След това изваждаме получената стойност за A_m от двете страни и делим на K. Вляво ще остане $(N - A_m)/K = (N - N\%K)/K = \lfloor N/K \rfloor$.

След това намираме A_{m-1}. И продължаваме по същия начин, докато вляво не остане нула. Резултата, който получаваме е отзад напред, затова трябва да го обърнем накрая. Също трябва да внимаваме ако имаме бройна система по-голяма от десетична, може да се наложи да използваме букви.

```
string fromDecimal(long long n, int base) {
    string ans;
    while (n > 0) {
        int digit = n%base;
        if (digit < 10) ans += char (digit+'0');
        else ans += char (digit-10+'A');
        n /= base;
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

12.3 Задачи

Задача 12.17 (Зимен Турнир 2015, D група, Двоични числа).

Задача 12.18 (Есенен Турнир 2014, D група, Равенство).

Глава 13

Дълги числа

13.1 Представяне

Задача 13.1. Кое е най-голямото число което може да се запише в променлива от тип *long long*?

Решение. За съвременни компютри отговорът е $2^{63}-1$. Типът *long long* е 64 битов, което означава че всяко число се представя със 64 знака - нули и единици. Първият бит показва дали числото е положително или отрицателно. Остават 63 бита, което значи 2^{63} положителни и още толкова отрицателни числа. Понеже нулата заема от местата на положителните числа, то най-голямото е $2^{63}-1$.

Задача 13.2. Кое е най-голямото число което може да се запише в променлива от тип *unsigned long long*?

Решение. При unsigned типове няма отрицателни числа. За разлика от *long long* тук не се губи място, което да пази дали числото е положително или не. Затова всичките 64 позиции дават неотрицателни числа и така възможните числа са 2^{64}. Разбира се като махнем един вариант за числото нула, най-голямото число ще е $2^{64}-1$.

Задача 13.3. Как може да пазим число което има много цифри, например 100-цифрено.

Решение. Определено няма как да го пазим в стандартен целочислен тип. Има няколко варианта, но ние ще споменем два. Първият е масив с променлива дължина, където всяка цифра от числото е отделен елемент от масива - *vector*. Втори вариант е да го пазим като текст - *string*.
Има две съществени разлики между двата варианта. Предимството на *string* е, че можем директно да го използваме със *cin* и *cout*. Например ако трябва да прочетем дълго число във *vector*, трябва първо да го прочетем като низ и след това да го обърнем във *vector*. От своя страна *vector* има предимство при работа с големи бройни системи. На пръв поглед това няма голям смисъл, но всъщност ни позволява да програмите да работят по-бързо. В тази глава ще пазим дългите числа в *string*.

Задача 13.4. При кои аритметични операции смятането се извършва отляво надясно по цифрите на числата.

Решение. При делението е по този начин, за разлика от събиране, изваждане и умножение.
По тази причина за удобство ще пазим цифрите в обратен ред. Първият елемент ще е цифрата на единиците, вторият на десетиците и т.н.
Всъщност този начин на пазене на числата ни дава още едно важно предимство. Когато в резултат на някоя от операциите останат водещи нули, те ще са в края на низа. Операциите за добавяне и махане на елементи в края на низ са много по-бързи от тези в началото.

Задача 13.5. Как може да махнем водещите нули на едно число, записано в обратен ред.

Решение. Водещите нули ще се намират в края на низа. Докато имаме водеща нула(последната цифра на низа е нула) ще я махаме. Все пак трябва да внимаваме и да оставим поне една цифра в числото.

```
while (c.size() > 1 && c.back() == '0') c.pop_back();
```

13.2 Сравняване

Задача 13.6. Как се сравняват две неотрицателни числа.

Решение. Първо сравняваме броя на цифрите им. Ако той е различен, то числото с повече цифри е по-голямо. Ако имат равен брой цифри започваме да сравняваме цифрите отляво надясно, докато намерим първата разлика, ако има такава.

Задача 13.7. Ако имаме масив от дълги числа *string a[n]* какъв ще е резултата след извикването на *sort(a, a+n)*?

Решение. Нека си припомним как по подразбиране се сравняват два низа - това става на база на лексикографската наредба. Първо се сравняват първите елементи на двата низа. Ако те са различни връщаме резултат. Ако са равни се сравняват вторите елементи и т.н. Първите два елемента които са различни определят кой е по-малкият низ. Ако няма разлика и на някой от низовете свършат елементите, то той е по-малък. Ако няма разлика и низовете имат равни дължини, то двата низа са равни.

Задача 13.8. Каква е лексикографската наредба на следните низове 100, 11, 123, 2, 1, 22, 3?

Решение. Подредбата е 1, 100, 11, 123, 2, 22, 3.

Задача 13.9. Кога стандартното сортиране на масив от числа ги подрежда в нарастващ ред?

Решение. Когато числата са записани отляво надясно и имат равен брой цифри.

Задача 13.10. Напишете функция, която приема две числа от тип *string*, записани отляво надясно и връща дали първото число е по-малко от второто.

Решение. Прилагаме стандартния алгоритъм. Ако числата имат различен брой цифри, то първото е по-малко когато има по-малко цифри. Ако двете числа имат равен брой цифри трябва да сравняваме една по една, докато намерим първата разлика. В този случай, знаем че лексикографското сравнение на низовете прави същото нещо и ще използваме него.

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

Задача 13.11. Напишете функция, която приема две числа от тип *string*, записани в обратен ред и връща дали първото число е по-малко от второто.

Решение. Разбира се може да обърнем числата и да извикаме нормалното сравнение, но това ще доведе до доста излишни операции, както и промяна на низовете. За разлика от преди малко, ако двете числа имат равен брой цифри, не може да използваме лексикографското сравнение и трябва да обхождаме отзад напред, докато намерим първата разлика.

```
bool isLess(const string& a, const string& b) {
    if (a.size() != b.size()) return a.size() < b.size();

    for (int i = a.size()-1; i >= 0; i--) {
        if (a[i] != b[i]) return a[i] < b[i];
    }

    return false;
}
```

Задача 13.12. Как да сортираме масив с дълги числа?

Решение. Нека имаме масив *string a[n]*, с n дълги числа. Подредбата на числата по подразбиране очевидно няма да ни свърши работа. Все пак за да подредим числата по големина можем да използваме вградената функция *sort*, но специфичното в случая е, че трябва да подадем наша функция която да сравнява числата. Нашата функция трябва по зададени две числа да връща резултат от тип *bool*. Резултатът трябва да е *true*, тогава и само тогава, когато първото число е преди второто в желаната подредба. В случая, в който подреждаме числата в нарастващ ред трябва да кажем дали първото число е строго по-малко от второто.
Всъщност функцията за сравнение *isLess*, която вече написахме прави точно това, така че може да използваме нея. Така за да сортираме използваме:

```
sort(a, a+n, isLess);
```

13.3 Събиране

Задача 13.13. Колко цифри има сбора на две числа с по n и m цифри съответно?

Решение. Сборът ще има поне толкова цифри колкото има по-голямото от двете числа - $max(n, m)$. И също така е ясно, че може да има една допълнителна цифра, ако е останало нещо за пренасяне след последното събиране. Т.е. сбора ще има или $max(n, m)$, или $max(n, m) + 1$ цифри.

Задача 13.14. Напишете функция, която събира две числа *string a* и *string b*, които са записани в обратен ред.

Решение. Функцията ще събира числата както са ни учили по математика. Първо събираме цифрите на единиците, после на десетиците и т.н., като ще имаме една променлива в която ще пазим колко е преноса. На всяка стъпка ще събираме три неща $a_i + b_i + prenos$ - поредната цифра от първото число, поредната цифра от второто число и преноса, като в началото *prenos* ще е 0. От полученият резултат последната цифра добавяме към крайния резултат, а от останалите цифри определяме преноса. Ако $current = a_i + b_i + prenos$, то към резултата прибавяме $current\%10$, а новият пренос ще е $current/10$. Нека да видим колко пъти трябва да извършим тези стъпки. Вече знаем, че крайния резултат ще има $max(a.size(), b.size())$ или $max(a.size(), b.size())+1$ цифри. Ако в условието на цикъла използваме $i < max(a.size(), b.size())$, може да пропуснем една цифра. За да имаме $max(a.size(), b.size()) + 1$ цифри в крайния резултат означава, че сме имали пренос преди последната стъпка. Така може да използваме един трик и ако имаме пренос също да влизаме с цикъла. Така добавяме това към проверката $i < max(a.size(), b.size())$ || $prenos$. В случая това ни спестява допълнителна проверка след цикъла дали има останал пренос.
Понеже е възможно в някоя стъпка да са свършили цифрите и на двете числа трябва да добавим проверки за това. Също така понеже числата са низове, трябва да внимаваме кога превключваме между символен и целочислен тип.

```
string sum(const string& a, const string& b) {
    string c;
    int prenos = 0;
    for (int i = 0; i < max(a.size(), b.size()) || prenos != 0; i++) {
        int current = prenos;
        if (i < a.size()) current += a[i]-'0';
        if (i < b.size()) current += b[i]-'0';
        c.push_back(current%10+'0');
        prenos = current/10;
    }
    return c;
}
```

Задача 13.15. Каква е най-голямата стойност, която променливата *current* може да приема.

Решение. Преноса в началото е сбор на две цифри, а след това сбор на две цифри добавени към останалия пренос от предния път. В началото може да съберем най-много две цифри, които са равни на 9, за да получим 18. При това записваме 8 към сбора и оставяме пренос 1, който ще добавим към сбора на следващите цифри. Към него пак може да добавим две 9ки, за да получим 19. Така преноса пак ще остане 1 и лесно се вижда, че няма как да получим число по-голямо от 19.

Задача 13.16. Във функцията за събиране, можем ли да заменим операциите за взимане на частно и остатък($current/10$, $current\%10$) с други, които да са по-бързи?

Решение. По-горе видяхме, че най-голямата стойност на $current$ е 19. Ако $current < 10$, то $current\%10$ ще е равно на $current$ и $current/10$ ще е 0. Ако $current \geq 10$, то $current\%10$ ще е равно на $current - 10$ и $current/10$ ще е 1.

Задача 13.17. Променете функцията за събиране, така че да няма операции за деление и остатък.

Решение.

```
string sum(const string& a, const string& b) {
    string c;
    int prenos = 0;
    for (int i = 0; i < max(a.size(), b.size()) || prenos != 0; i++) {
        int current = prenos;
        if (i < a.size()) current += a[i]-'0';
        if (i < b.size()) current += b[i]-'0';
        if (current < 10) {
            c.push_back(current+'0');
            prenos = 0;
        } else {
            c.push_back(current-10+'0');
            prenos = 1;
        }
    }
    return c;
}
```

13.4 Изваждане

Задача 13.18. Колко цифри има разликата на две неотрицателни числа с по n и m цифри съответно?

Решение. Разликата може да има между 1 и $max(n, m)$ цифри.

Задача 13.19. Напишете функция, която намира разликата на две числа *string a* и *string b*, които са записани в обратен ред, и първото е по-голямо или равно на второто.

Решение. Функцията за изваждане е много подобна на тази за събиране. Преноса в случая е нещото което ще вземем от следващата цифра на умаляемото. Разликата ще има дължината на умаляемото, което може да доведе до излишни нули в началото. Затова преди да върнем резултата добавяме един цикъл, който да премахне водещите нули.

```
string subtract(const string& a, const string& b) {
    string c;
    int prenos = 0;
    for (int i = 0; i < a.size(); i++) {
        int current = (a[i]-'0')-prenos;
        if (i < b.size()) current -= b[i]-'0';
        if (current >= 0) {
            c.push_back(current+'0');
            prenos=0;
        } else {
            c.push_back(current+10+'0');
            prenos=1;
        }
    }
    while (c.size() > 1 && c.back() == '0') c.pop_back();
    return c;
}
```

13.5 Умножение с късо число

Задача 13.20. Какъв е математическия алгоритъм за умножаване на едно дълго число от тип *string* с цифра?

Задача 13.21. Умножете числото 9427 с цифрата 8?

Решение. Започваме със $7.8 = 56$, записваме 6 към резултата и добавяме пренос 5. След това умножаваме 2 по 8 и добавяме преноса - $2.8 + 5 = 21$. Записваме 1 към резултата и добавяме пренос 2. Продължаваме така и след като умножим 9 по 8 и запишем резултата остава пренос 7. Този последен пренос записваме в резултата, с което приключваме.

$$\begin{array}{cccccc} & {\scriptstyle 7} & {\scriptstyle 3} & {\scriptstyle 2} & {\scriptstyle 5} & {\scriptstyle 0} \\ & & 9 & 4 & 2 & 7 \\ + & & & & & 8 \\ \hline & 7 & 2 & 4 & 1 & 6 \end{array}$$

Задача 13.22. Напишете функция за умножение на едно дълго число *string a* записано отзад напред с цифра *int b*?

Решение. На всяка стъпка умножаваме поредната цифра от дългото число с b и добавяме преноса. Нека $current = a_i * b + prenos$. Тогава към резултата добавяме $current\%10$ и новият пренос става $current/10$. В първата стъпка нямаме пренос, за

това началната му стойност ще е 0. Възможно е на последната стъпка да нямаме цифра от дългото число. По тази причина трябва да проверяваме за това. Остана да уточним колко пъти ще изпълняваме стъпката. По подобие на събирането ще имаме поне $a.size()$ стъпки, но след това може да има останал пренос. За това в условието на цикъла проверяваме и за двете неща.
Трябва да внимаваме за един специален случай и той е ако умножаваме по нула. Ако приложим описания алгоритъм ще получим резултат с много нули. За да решим проблема ще направим отделна проверка в началото на функцията.

```
string multiply(const string& a, int b) {
    if (b == 0) return "0";

    string c;
    int prenos = 0;
    for (int i = 0; i < a.size() || prenos != 0; i++) {
        int current = prenos;
        if (i < a.size()) current += (a[i]-'0')*b;
        c.push_back(current%10+'0');
        prenos = current/10;
    }
    return c;
}
```

Задача 13.23. Умножете числото 9427 с 11?

Решение. Последователно умножаваме всяка от цифрите на 9427 с 11, като започваме със 7.11. След всяко умножение добавяме отместване отдясно. Накрая събираме получените резултати.
За умножението на цифра с 11 използваме стандартния алгоритъм за умножение на число с цифра.

$$\begin{array}{cccccc} & & 9 & 4 & 2 & 7 \\ + & & & & 1 & 1 \\ \hline & & & & 7 & 7 \\ & & & 2 & 2 & . \\ & & 4 & 4 & . & . \\ & 9 & 9 & . & . & . \\ \hline 1 & 0 & 3 & 6 & 9 & 7 \end{array}$$

Задача 13.24. Умножете числото 9427 с двуцифреното число 11, без да пазите междинни резултати?

Решение. Като за начало ще отбележим, че междинните резултати са лесни за смятане понеже са произведения на цифри с числото 11.
Първото нещо което можем да сметнем от крайния резултат е цифрата на единиците. Тя зависи само от произведението 7.11. От резултата 77 цифрата на единиците отива в крайния резултат. Цифрата на десетиците трябва да я запазим и да я добавим към

произведението 2.11. От тази сума $7 + 2.11 = 29$ цифрата на единиците отива към крайния резултат, а сега тази на десетиците трябва да я запазим. Продължаваме по същия начин, като на всяка стъпка умножаваме поредната цифра по 11 и я събираме с числото останало от предната стъпка, което всъщност нормално наричаме пренос.

$$\begin{array}{cccccccc} & & {\scriptstyle 10} & {\scriptstyle 4} & {\scriptstyle 2} & {\scriptstyle 7} & {\scriptstyle 0} \\ & & & 9 & 4 & 2 & 7 \\ + & & & & & 1 & 1 \\ \hline & 1 & 0 & 3 & 6 & 9 & 7 \end{array}$$

Ще отбележим, че този алгоритъм е същия като този за умножение на дълго число с цифра.

Задача 13.25. Напишете функция за умножение на едно дълго число *string a* записано отзад напред с число *int b*

Решение. Основната цел тук е да използваме, че малкото число е от целочислен тип. Всъщност това ни позволява бързо да пресмятаме $a_i.b$. Това ни подсказва да използваме алгоритъма от предишната задача и да смятаме без да пазим междинни резултати. Така на всяка стъпка ще имаме $current = a_i.b + prenos$, към резултата добавяме $current\%10$ и новия пренос ще е $current/10$. До тук всичко е като алгоритъма за умножение на дълго число с цифра.
Една разлика е, че след последното умножение може да остане голям пренос. Ще отбележим, че това не е проблем понеже ние влизаме в цикъла докато има пренос. Така на една такава стъпка към резултата ще добавяме последната цифра от преноса и новия пренос ще е без нея.
Друго нещо важно е, че $a_i * b + prenos$ може да е голямо число и да не се побира в *int*. Най-лесно се справяме с това като използваме *long long b*.

```
string multiply(const string& a, long long b) {
    if (b == 0) return "0";

    string c;
    long long prenos = 0;
    for (int i = 0; i < a.size() || prenos != 0; i++) {
        long long current = prenos;
        if (i < a.size()) current += (a[i]-'0')*b;
        c.push_back(current%10+'0');
        prenos = current/10;
    }
    return c;
}
```

Задача 13.26. Ще работи ли тази програма за числа които не се побират в *int*?

Решение. Важното е $a_i * b + prenos$ да се събира в *long long*. Понеже a_i е цифра, то b може да е доста голямо число - около 17 цифри.

13.6 Деление на дълго с късо число

Задача 13.27. Напишете функция за деление на едно дълго число от тип *string* с число от тип *int*?

Решение. По подобие на останалите операции, ще пазим числото в обратен ред. Понеже делението се извършва отпред назад, то ние ще обхождаме низа в обратен ред. Във всеки момент пазим какво е остатъка от предишното деление.

```
pair<string, long long> divide(const string& a, long long b) {
  string c(a.size(), '0');
  long long prenos = 0;
  for (int i = a.size()-1; i >= 0; i--) {
    prenos = 10*prenos+a[i]-'0';
    c[i] += prenos/b;
    prenos %= b;
  }
  while (c.size() > 1 && c.back() == '0') c.pop_back();
  return {c, prenos};
}
```

13.7 Умножение на две дълги числа

Задача 13.28. Колко цифри има произведението на две положителни числа с по n и m цифри съответно?

Решение. Най-лесно може да видим отговора, ако умножим най-малките и най-големите числа с дадения брой цифри. Най-малките числа с n и m цифри са 10^{n-1} и 10^{m-1}. Тяхното произведение е $10^{n-1} * 10^{m-1} = 10^{n+m-2}$, което има $n+m-1$ цифри. Най-големите числа с n и m цифри са $10^n - 1$ и $10^m - 1$. Тяхното произведение е $(10^n - 1) * (10^m - 1) = 10^{n+m} - 10^n - 10^m + 1$. Оставяме на вас да докажете, че това число има $n + m$ цифри.

Задача 13.29. Напишете функция за умножение на две дълги числа?

Решение. Да си припомним как се умножават две числа.

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

Ако трябва да го напишем по този начин трябва да използваме умножение на дълго число с цифра, като добавим отместване и събиране на дълги числа. Всъщност може да го направим доста по кратко и лесно. За целта ще пазим едно число за

резултат и всеки път като смятаме ще добавяме към този резултат.
Умножението ще изглежда така:

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

В началото създаваме низ с дължина максималната възможна дължина на произведението, който е запълнен с нули. Текущият резултат е равен на нула.
Първата стъпка е да умножим $123 * 6$ и да го добавим към текущия резултат. Така получаваме временен резултат равен на $123 * 6$
Втората стъпка е да умножим $123 * 5$ и да го добавим към текущия резултат, но започнем добавянето с отместване на една цифра.
Новият временен резултат е равен на $123*6+123*50 = 123*56$. Накрая умножаваме $123 * 6$ и този път добавяме към текущия резултат с отместване на две цифри, за да получим крайния резултат.

Остана да се справим с отместването. Може да забележим, че когато умножаваме $a[0]$ с $b[j]$, то резултата ще отиде в $c[0 + j]$. Всъщност когато умножаваме $a[i] * b[j]$, ще го добавяме към $c[i + j]$.

```
string multiply(const string& a, const string& b) {
  string c(a.size()+b.size(), 0);

  for (int i = 0; i < a.size(); i++) {
    int prenos = 0;
    for (int j = 0; j < b.size() || prenos; j++) {
      int curr = prenos+c[i+j]-'0';
      if (j < b.size()) curr += (a[i]-'0')*(b[j]-'0');
      c[i+j] = curr%10+'0';
      prenos = curr/10;
    }
  }

  while (c.size() > 1 && c.back() == '0') c.pop_back();
  return c;
}
```

Задача 13.30. Защо добавяме нули в началото на вектора, а не ползваме $push_back$ за добавяне на елементите?

13.8 Оптимизации

Ще разгледаме една задача, която е показателна за доста от оптимизациите, които може да използваме.

Задача 13.31. Пресметнете $n!$ възможно най-бързо?

Решение. За да пресметнем $n!$ може да използваме умножение на дълго число с късо n пъти. Да видим какво може да подобрим.
Оптимизация 1: Като първа оптимизация ще споменем една често срещана грешка. Тя се дължи на навика винаги да пазим числата отляво надясно. Така при всяка операция ги обръщаме наобратно, извършваме операцията и накрая го обръщаме. Това бави излишно програмата. Много по-добре е да обърнем числата в началото, да направим всички операции с дълги числа и чак накрая ако се налага да обърнем резултата.
Оптимизация 2: Пресмятането на частното $current/10$ и остатъка $current\%10$ са едни от бавните операции при работа с дълги числа. Като знаем, че $a\%b = a - \lfloor a/b \rfloor * b$, то може да използваме частното, за да сметнем остатъка. Обикновено запазваме $prenos = current/10$ и така $current\%10 = current - 10 * prenos$.
Оптимизация 3а: При всяка една операция за резултата създаваме нов низ. В началото той е празен и на всяка стъпка добавяме по една цифра към резултата. Добавянето на символ по символ към низ е по-бавно от създаването на дълъг низ още в началото. Имаме ориентир за броя цифри в резултата и има вариант да го използваме - *string c(a.size(), '0')*. След това по време на стъпките от операцията да използваме *c[i]* и чак когато прехвърлим елементите му да използваме *c.push_ back()*.
Оптимизация 3б: Може да подходим още по-крайно и да не създаваме нов низ, а да запазваме резултата на мястото на дългия множител. Когато на i-тата стъпка сме сметнали *current*, то i-тата цифра на дългия множител не ни трябва и може там да запазим i-тата цифра от резултата. Отново трябва да внимаваме когато надхвърлим дължината на първоначалното число.
Оптимизация 3в: Като продължение на 3б може още в началото да създадем голям низ и да пазим в променлива дължината на текущото число в него.
В случай че не гоним оптимизации по памет от варианти 3а, 3б и 3в първият може да е достатъчно добър и лесен за писане вариант.
Оптимизация 4: Сложността на една операция се определя главно от броя цифри на дългото число. Така ако трябва да умножим число с n цифри по 2, после по 3 ще направим $n+n$ операции. Ако обаче го умножим направо по 6(2.3) ще имаме само n операции. Така може да комбинираме няколко малки числа в едно по-голямо малко число, като внимаваме да не превишаваме 17 цифри и да не излизаме от *long long* при проверките.
Тези оптимизации са сравнително дребни и е добре да се прилагат след като сме направили и качили работещо решение, но имаме нужда да го забързаме.
Следва имплементация, която демонстрира тези оптимизации:

```
string multiply(const string& a, long long b) {
    string c(a.size(), '0');
    long long prenos = 0;
    for (int i = 0; i < a.size() || prenos != 0; i++) {
        long long current = prenos;
        if (i < a.size()) current += (a[i]-'0')*b;
```

```
        else c.push_back('0');
        prenos = current/10;
        c[i] = current-prenos*10+'0';
    }
    return c;
}

string nfact(int n) {
    string ans = "1";
    long long combined = 1;
    for (int i = 1; i <= n; i++) {
        combined *= i;
        if (i == n || 1e17/combined < i+1) {
            ans = multiply(ans, combined);
            combined = 1;
        }
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

13.9 Задачи

Задача 13.32 (Есенен Турнир, 2012, D група, Номер на страница).

Задача 13.33 (Есенен Турнир, 2014, D група, Редица).

Задача 13.34 (Зимен Турнир, 2017, D група, Кратно на 3).

Задача 13.35 (Есенен Турнир, 2018, D група, Промени числото).

Задача 13.36 (НОИ, Областен кръг, 2018, D група, Дълго число).

Задача 13.37 (НОИ, Областен кръг, 2017, D група, Степен на двойката).

Задача 13.38 (НОИ, Областен кръг, 2018, D група, Сума от цифри на N!).

Задача 13.39 (НОИ, Областен кръг, 2015, D група, Максимално произведение).

Задача 13.40 (Есенен Турнир, 2018, D група, Равенство).

Задача 13.41 (Младежка Балканиада, Румъния, 2018, Ден 1, Хармонично число).

Глава 14

Двумерни масиви

14.1 Въведение

Дефиниция 14.1. Масив от масиви (масив, чиито елементи са масиви) се нарича двумерен масив. Двумерните масиви са известни като матрица или таблица с редове и колони.

Най-лесно когато мислим за двумерен масив, е да си представяме таблица с редове и колони. За да създадем таблица с N реда и M колони използваме *type name*$[N][M]$, където *type* е типът на данните, които ще съхраняваме в таблицата - *bool*, *int*, *long long*, *char*, *string*, ..., а *name* - името на таблицата.
Например може да създадем таблица, в която ще пазим цели числа по следния начин - *int table*$[3][4]$. Тя ще има 3 реда и 4 колони, и ще изглежда ето така:

table[0][0]	table[0][1]	table[0][2]	table[0][3]
table[1][0]	table[1][1]	table[1][2]	table[1][3]
table[2][0]	table[2][1]	table[2][2]	table[2][3]

Номерацията на редовете и колоните по подобие на едномерните масиви започва от нула. Първият елемент на първия ред е *table*$[0][0]$. Последният елемент на последния ред е *table*$[N-1][M-1]$, в примера това е *table*$[0][3]$.

Важно е да отбележим, че по подобие на едномерните масиви, създаването на двумерни масиви с голям брой елементи не трябва да е вътре във функция! Също така ако масивът е създаден извън функция ще има нулеви начални стойности, а вътре във функция стойностите ще са случайни.

Може да зададем стойности на масива при самото създаване.

```
int table[2][4] = {
  {1, 2, 3, 4},
  {5, 6, 7, 8}
};
```

По този начин създаваме таблица с два реда и четири колони.

Задача 14.1. Дадени са две числа N и M. Създайте двумерен масив с N реда и M колони.

За да създадем таблица с точно N реда и M колони първо трябва да прочетем N и M и след това да създадем двумерния масив. Това може да се случи във функция и би изглеждало така:

```
int main() {
    int N, M;
    cin >> N >> M;
    int table[N][M];
}
```

Това обаче ще работи твърде бавно за големи таблици и за това почти никога не бихме го направили така. Вместо това предварително ще създадем достатъчно голяма таблица според ограниченията за N и M, и след това ще използваме само първите N реда и M стълба от нея.

```
int table[1000][1000];
int main() {
    int N, M;
    cin >> N >> M;
    // използваме елементите от table[0][0] до table[N-1][M-1]
}
```

Задача 14.2. Дадени са две числа N и M - броя редове и стълбове на една таблица. Следват N реда с по M числа - самата таблица. Прочетете и запазете първия ред от таблицата.

Числата на първия ред са $table[0][0]$, $table[0][1]$, ..., $table[0][M-1]$. Единствено трябва да завъртим един цикъл по колоните от 0, до $M-1$.

```
int table[1000][1000];
int main() {
    int N, M;
    cin >> N >> M;
    for (int j = 0; j < M; j++) {
        cin >> table[0][j];
    }
}
```

В цикли стандартно ще използваме i за обхождане по редовете и j - за колоните.

Задача 14.3. Дадени са две числа N и M - броя редове и стълбове на една таблица. Следват N реда с по M числа - самата таблица. Прочетете и запазете цялата таблица в двумерен масив.

Знаем как се чете един ред. Остава това да го оградим в един цикъл, който се мени по броя на редовете.

```
int table[1000][1000];
int main() {
  int N, M;
  cin >> N >> M;
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
      cin >> table[i][j];
    }
  }
}
```

Задача 14.4. Как ще изкараме на екрана стойността, която се намира в шестата колона на трети ред.

Трети ред ще отговаря на индекс 2, а шеста колона ще има индекс 5 - *cout* << *table*[2][5] << *endl*;.

14.2 Обхождане

Ще разгледаме серия от задачи при които ще обхождаме масива по различен начин.

Задача 14.5. Дадена е таблица с N реда и M колони. Изпишете таблицата по стълбове. На първия ред да са числата от първата колона, на втория ред от втората колона и т.н.

Числата в първия стълб са $table[0][0]$, $table[1][0]$, ..., $table[N-1][0]$, във втория - $table[0][1]$, $table[1][1]$, ..., $table[N-1][1]$ и т.н. Всъщност лесно се вижда, че просто трябва да обърнем двата цикъла.

```
for (int j = 0; j < M; j++) {
  for (int i = 0; i < N; i++) {
    cout << table[i][j] << " ";
  }
  cout << endl;
}
```

Задача 14.6. Дадена е таблица с N реда и M колони. Изкарайте таблицата на екрана по редове, като във всеки втори ред числата трябва да са в обратен ред.

За всеки ред ще проверяваме дали е четен или нечетен. Четните редове - 0, 2, ... ще изписваме по нормален начин, а нечетните наобратно.

```
for (int i = 0; i < N; i++) {
  if (i%2 == 0) {
      for (int j = 0; j < M; j++) {
        cout << table[i][j] << " ";
      }
      cout << endl;
    } else {
      for (int j = M-1; j >= 0; j--) {
```

```
        cout << table[i][j] << " ";
      }
      cout << endl;
    }
}
```

Дефиниция 14.2. Главен диагонал на двумерен масив $int\ table[N][M]$ ще наричаме множеството от всички стойности $int\ table[i][j]$, за които $i = j$.

Дефиниция 14.3. Вторичен диагонал на двумерен масив $int\ table[N][M]$ ще наричаме множеството от всички стойности $int\ table[i][j]$, за които $i + j = M - 1$.

Задача 14.7. Дадена е числова таблица с N реда и N колони. Изпишете на екрана елементите от главния диагонал.

Понеже таблицата е квадратна елементите по главния диагонал ще са $table[0][0]$, $table[1][1]$, ..., $table[N-1][N-1]$. Изписването трябва да е очевидно.

```
for (int i = 0; i < N; i++) {
    cout << table[i][i] << " ";
}
cout << endl;
```

Задача 14.8. Дадена е числова таблица с N реда и M колони. Изпишете на екрана елементите от главния диагонал.

Сега вече е малко по-трудно. Не е ясно колко елемента има по-главния диагонал. Имаме различни случай в зависимост дали редовете или колоните са повече. Реално за нас това не трябва да има значение. Трябва да обхождаме докато сме в границите на масива. Момента, в който един от индексите или и двата излязат от границите на масива, ще означава край.

```
for (int i = 0; i < N && i < M; i++) {
  cout << table[i][i] << " ";
}
cout << endl;
```

Задача 14.9. Дадена е числова таблица с N реда и N колони. Изпишете на екрана елементите от вторичния диагонал.

Понеже таблицата е квадратна елементите по вторичния диагонал ще са $table[0][N-1]$, $table[1][N-2]$, ..., $table[N-1][0]$. Може да използваме една променлива за реда и една за стълба и всеки път да увеличаваме реда и да намаляваме стълба.

```
for (int i = 0, j = N-1; i < N; i++, j--) {
    cout << table[i][j] << " ";
}
cout << endl;
```

Всъщност може да използваме факта, че $i + j = N - 1$. Така може да не ползваме втората променлива.

```
for (int i = 0; i < N; i++) {
    cout << table[i][N-1-i] << " ";
}
cout << endl;
```

Задача 14.10. Дадена е числова таблица с N реда и M колони. Изпишете на екрана елементите от главния диагонал.

Отново трябва да проверяваме дали не сме излезли от границите на масива, както по редове така и по колони. Понеже тук колоните намаляват проверката ще е $i < N \&\& j >= 0$, като използваме i за реда и j за колоната.

```
for (int i = 0, j = M-1; i < N && j >= 0; i++, j--) {
    cout << table[i][j] << " ";
}
cout << endl;
```

Ако ползваме зависимостта $i + j = M - 1$:

```
for (int i = 0; i < N && M-1-i >= 0; i++) {
    cout << table[i][M-1-i] << " ";
}
cout << endl;
```

Задача 14.11. Дадена е числова таблица с N реда и M колони. Изпишете на екрана елементите от главния диагонал в обратен ред.

Разбира се един лесен вариант е да ги запишем някъде отпред назад и след това да ги обходим наобратно.
Как обаче може направо да ги обходим отзад напред. С кой елемент трябва да започнем. Чисто теоретично елементите трябва да са $table[0][M-1]$, $table[1][M-2]$, ..., $table[M-1][0]$. Обаче таблицата може да няма M-1-ви ред. Може да обхождаме отзад напред и всеки път да проверяваме дали сме в границите на таблицата.

```
for (int i = M-1, j = 0; i >= 0; i--, j++) {
    if (i < N) {
        cout << table[i][j] << " ";
    }
}
cout << endl;
```

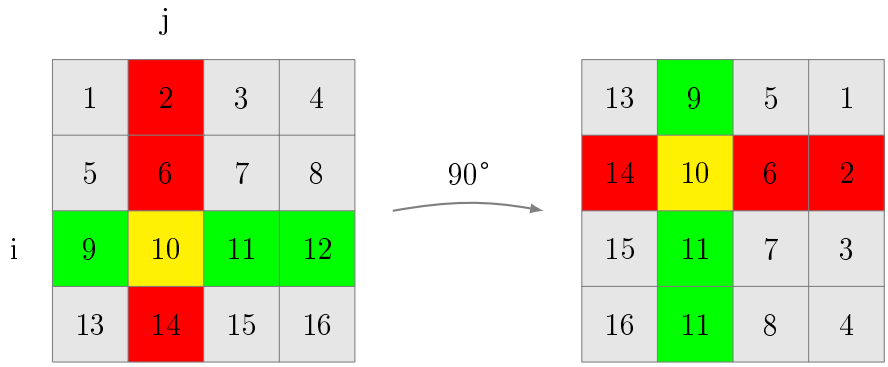
Може да помислите как направо да намерите кой е последния елемент по вторичния диагонал.

Задача 14.12. Дадена е числова таблица $int\ table[N][M]$. За дадени числа x и y, проверете елемента $table[x][y]$ е част от таблицата или излиза от нея.

За да е елемента в таблицата трябва реда да е в интервала $[0; N-1]$ и колоната в интервала $[0; M-1]$. Един вариант е да проверим по следния начин.

```
if (0 <= x && x < N && 0 <= y && y < N)
```

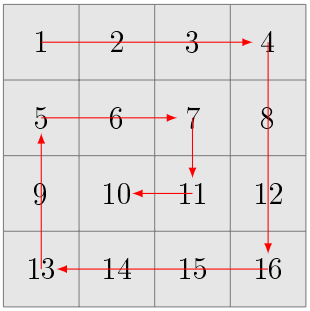
Задача 14.13. Дадена е квадратна таблица $int\ table[N][N]$. Завъртете я 90° надясно.



Да видим къде се премества клетката с координати (i, j), оцветена в жълто.
При завъртането цялата колона j става ред, като при това всички колони които са били преди j, и като редове ще са по напред. Така (i, j) ще отиде на ред j след завъртането.
При завъртането реда i става колона. Обаче колоните които са преди новата всъщност са редовете които са били след i преди завъртането. За това този път се интересуваме от разстоянието до последния ред. Новата колона на клетката (i, j) ще бъде $N - 1 - i$.
Така стигаме до извода, че клетката (i, j) ще отиде в $(j, N - 1 - i)$.
Ако създадем нова таблица $rotated[N][N]$ можем да обходим всички елементи от първата и да ги сложим на правилните места в новата таблица.

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        rotated[j][N-1-i] = table[i][j];
    }
}
```

Задача 14.14. Дадена е правоъгълна таблица $int\ table[N][M]$. Изпишете числата като обходите таблицата във формата на спирала като започнете обхождането надясно.

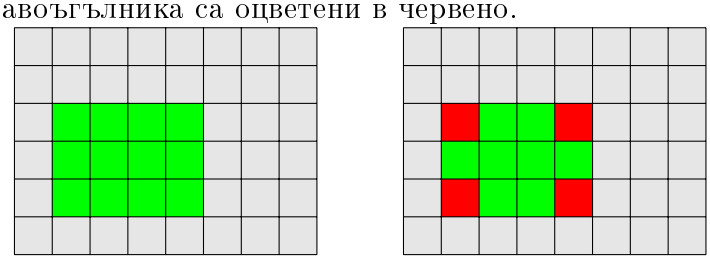


14.3 Рамка

Задача 14.15. Даден е двумерен масив $table[N][M]$ запълнен с цели числа. За всяка клетка (i, j) пресметнете сумата на числата от клетките, които имат обща страна с (i, j).

14.4 Правоъгълници

В двумерните масиви доста често ще разглеждаме правоъгълници от клетки. За сега ще се интересуваме единствено от правоъгълници със страни успоредни на страните на таблицата. Т.е. страните на правоъгълниците ще са само хоризонтални и вертикални. Клетките в четирите края ще наричаме върхове на правоъгълника. На чертежа по-долу е даден примерен правоъгълник, оцветен в зелено. Върховете на правоъгълника са оцветени в червено.

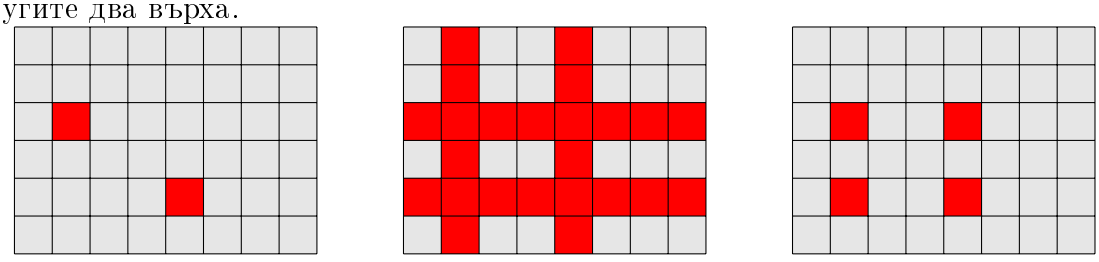


Задача 14.16. Колко върха има един правоъгълник.

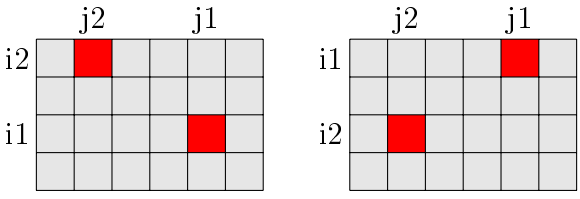
На теория е ясно, че един правоъгълник има четири върха. Гледайки таблицата може да ги назовем горен-ляв, горен-десен, долен-ляв и долен-десен. Въпреки че така изглеждат четири е възможно клетките на някой от тях да съвпадат. Например ако имаме правоъгълник само с един ред, то горен-ляв и долен-ляв връх ще бъде една и съща клетка. Аналогично ще съвпадат и двата десни върха. Важно е да се отбележи, че е възможно правоъгълника де се състои само от една клетка и тогава всички върхове ще са в тази клетка.

Задача 14.17. С колко най-малко клетки от таблицата може да дефинираме целия правоъгълник.

Ясно е че четирите върха определят еднозначно един правоъгълник. Също ако махнем един връх пак има един вариант.
Интересно е какво се случва ако имаме два върха. Ако двата върха са на един ред, не може да определим на кой ред са другите два върха. Аналогично ако върховете са в една колона не може да определим другата колона. Обаче ако имаме два противоположни върха обаче няма проблем да определим правоъгълника. Разглеждаме хоризонталните и вертикалните прави през тях и където се пресичат получаваме другите два върха.

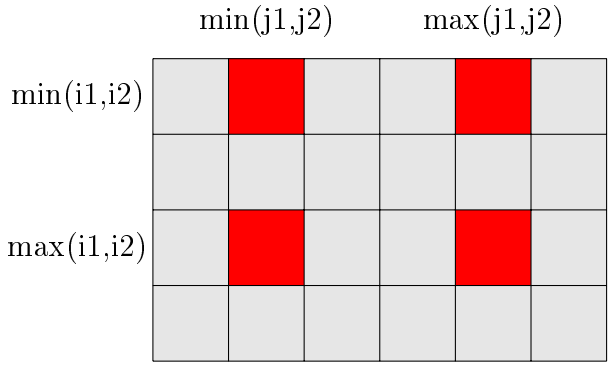


Задача 14.18. Дадени са два срещуположни върха на правоъгълник $(i1, j1)$, $(i2, j2)$. Намерете координатите горния ляв връх.



Като начертаем няколко случая е ясно, че като вземем редовете на двете точки

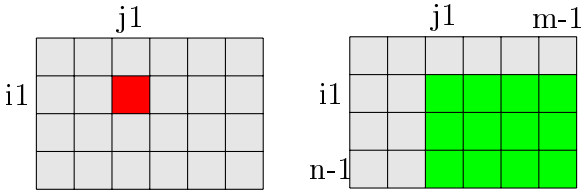
получаваме горния и долния ред на върховете на правоъгълника. Съответно горния ще е $min(i1, i2)$ и долния ще е $max(i1, i2)$. Аналогично и за колоните стигаме до следния извод, на картинката.



Сега вече са ясни координатите на всичките четири върха. Например горния ляв връх има координати $(min(i1, i2), min(j1, j2))$.

Задача 14.19. Колко правоъгълника съществуват с горен ляв връх $(i1, j1)$.

Вече знаем, че един правоъгълник е определен ако имаме два противоположни върха. Противоположен на горния ляв връх е долния десен. Трябва да намерим кои клетки са възможните варианти за долен десен. Знаем, че за всеки долен десен връх (i, j) трябва $i1 \leq i$ и $j1 \leq j$. Това са всички клетки надолу и надясно от дадената.



Сега остава да намерив бройката. Всички редове от $i1$ до $n-1$ са $n-1-i1$. Всички колони от $j1$ до $m-1$ са $m-1-j1$. Така броя на възможните правоъгълници е $(n-i1)*(m-j1)$.

Задача 14.20. Колко правоъгълника с горен ляв връх на i-ия ред съществуват вътре в даден правоъгълник с n реда и m колони.

Може да преброим правоъгълниците с горен ляв ъгъл всяка от клетките $(i, 0)$, $(i, 1)$, ..., $(i, m-1)$. Ще използваме формулата от предната задача, че броят е $(n-i)*(m-j)$.

```
int countRectangles(int n, int m, int i) {
    int rectangles = 0;
    for (int j = 0; j < m; j++) {
        rectangles += (n-i)*(m-j);
    }
    return rectangles;
}
```

Това решение изисква да обходим всички клетки от i-ия ред.
Може да подобрим това решение. Да разгледаме формулата за всяка клетка от реда.

За $(i, 0)$ имаме $(n - i) * (m - j) = (n - i) * (m - 0) = (n - i) * m$. За $(i, 1)$ имаме $(n - i) * (m - j) = (n - i) * (m - 1) = (n - i) * (m - 1)$. И т.н. за $(i, m - 1)$ имаме $(n-i)*(m-(m-1)) = (n-i)*(1) = (n-i)*1$. Като съберем всички правоъгълници получаваме $(n-i)*m+(n-i)*(m-1)+...+(n-i)*1 = (n-i)*(1+2+...+m) = (n-i)*m*(m+1)/2$.

Задача 14.21. Колко правоъгълника съществуват вътре в даден правоъгълник с n реда и m колони.

По подобие на горната задача може да минем през всяка клетка от таблицата и да приложим формулата за нея.

```
int countRectangles(int n, int m) {
    int rectangles = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            rectangles += (n-i)*(m-j);
        }
    }
    return rectangles;
}
```

Това решение изисква обхождане на цялата таблица.
Едно подобрение е за всеки ред да използваме формулата от предната задача.

```
int countRectangles(int n, int m) {
    int rectangles = 0;
    for (int i = 0; i < n; i++) {
        rectangles += (n-i)*m*(m+1)/2;
    }
    return rectangles;
}
```

С още малко математика може да се справим още по-бързо. Да разгледаме сумата за всеки от редовете прилагайки формулата за i-ия ред $(n - i) * m * (m + 1)/2$. За първия ред имаме $(n-i)*m*(m+1)/2 = (n-0)*m*(m+1)/2 = n*m*(m+1)/2$. За втория ред - $(n - i) * m * (m + 1)/2 = (n - 1) * m * (m + 1)/2$. И т.н. за n-ия ред - $(n - (n - 1)) * m * (m + 1)/2 = 1 * m * (m + 1)/2$. Като съберем всичките получаваме $n * m * (m + 1)/2 + (n - 1) * m * (m + 1)/2 + ... + 1 * m * (m + 1)/2 = (1 + 2 + ... + n) * m * (m + 1)/2 = n * (n + 1) * m * (m + 1)/4$.

```
int countRectangles(int n, int m) {
    return n*(n+1)*m*(m+1)/4;
}
```

Задача 14.22. Даден е двумерен масив $table[n][m]$ запълнен с цели числа. Даден е правоъгълник с горен ляв връх $(i1, j1)$ и долен десен $(i2, j2)$. Намерете сбора на числата в този правоъгълник.

Обхождаме всички клетки в правоъгълника и ги прибавяме към общата сума.

```
int sumInRectangle(int i1, int j1, int i2, int j2) {
    int sum = 0;
    for (int i = i1; i <= i2; i++) {
        for (int j = j1; j <= j2; j++) {
            sum += table[i][j];
        }
    }
    return sum;
}
```

Задача 14.23. Даден е двумерен масив $table[n][m]$ запълнен с цели числа. Даден е правоъгълник с два противоположни върха $(i1, j1)$ и $(i2, j2)$. Намерете сбора на числата в този правоъгълник.

От дадените два върха намираме горен ляв и долен десен и пак добавяме всички клетки в правоъгълника към общата сума.

```
int sumInRectangle(int i1, int j1, int i2, int j2) {
    int sum = 0;
    for (int i = min(i1, i2); i <= max(i1, i2); i++) {
        for (int j = min(j1, j2); j <= max(j1, j2); j++) {
            sum += table[i][j];
        }
    }
    return sum;
}
```

Понеже ни интересува кои са по-малките и по-големите редове и колони вместо всеки път да ги смятаме може просто да подсигурим, че $i1 \leq i2$ и $j1 \leq j2$. В случай, че това не е така ги разменяме. Така ще знаем, че $(i1, j1)$ е горен ляв връх, $(i2, j2)$ - долен десен.

```
int sumInRectangle(int i1, int j1, int i2, int j2) {
    if (i1 > i2) swap(i1, i2);
    if (j1 > j2) swap(j1, j2);
    int sum = 0;
    for (int i = i1; i <= i2; i++) {
        for (int j = j1; j <= j2; j++) {
            sum += table[i][j];
        }
    }
    return sum;
}
```

Задача 14.24. Даден е правоъгълник с горен ляв връх $(i1, j1)$ и долен десен $(i2, j2)$. Проверете дали клетката (i, j) е част от него.

За да е вътре трябва едновременно реда i да е между двата крайни реда и същото за колоната - $i1 \leq i \leq i2$ и $j1 \leq j \leq j2$.

Задача 14.25. Даден е двумерен масив $table[n][m]$ запълнен с цели числа. Дадени са два правоъгълника с горен ляв и долен десен връх. Първият с върхове $(i1, j1)$ и $(i2, j2)$, вторият с върхове $(i3, j3)$ и $(i4, j4)$. Намерете дали имат общи точки.

Един вариант е да ползваме двумерен масив. Да обходим клетките от първия правоъгълник и да отбележим, че сме го обходили. След това да обходим тези от втория и срещнем такава която е вече отбелязана значи тя ще е обща. За целта ще използваме двумерен масив *used*, който в началото е запълнен с $false$.

```
bool hasCommonCell(
        int i1, int j1, int i2, int j2, int i3, int j3, int i4, int j4) {

    for (int i = i1; i <= i2; i++) {
        for (int j = j1; j <= j2; j++) {
            used[i][j] = true;
       }
    }
    for (int i = i3; i <= i4; i++) {
        for (int j = j3; j <= j4; j++) {
            if (used[i][j]) return true;
        }
    }
    return false;
}
```

Сега искаме да подобрим това решение и да не обикаляме по всички клетки на двата правоъгълника. Ако двата правоъгълника имат обща точка, то тя трябва да е вътре в единия и вътре в другия. Т.е. $i1 \leq i \leq i2$, $j1 \leq j \leq j2$, $i3 \leq i \leq i4$ и $j3 \leq i \leq j4$. Сега остава да проверим дали има точки, които отговарят на това условие. $i1 \leq i$ и $i3 \leq i$ може да го запишем като $max(i1, i3) \leq i$. Аналогично за другите стигаме до следните изисквания - $max(i1, i3) \leq i \leq min(i2, i4)$ и $max(j1, j3) \leq j \leq min(j2, j4)$. Тези равенства ще имат решение когато $max(i1, i3) \leq min(i2, i4)$ и $max(j1, j3) \leq min(j2, j4)$. Всъщност това е необходимо и достатъчно условие да правоъгълниците да имат общи клетки. Така програмата добива нов вид

```
bool hasCommonCell(
int i1, int j1, int i2, int j2, int i3, int j3, int i4, int j4) {
    return max(i1,i3) <= min(i2,i4) && max(j1,j3) <= min(j2,j4);
}
```

Задача 14.26. Даден е двумерен масив $table[n][m]$ запълнен с цели числа. Дадени са два правоъгълника с горен ляв и долен десен връх. Първият с върхове $(i1, j1)$ и $(i2, j2)$, вторият с върхове $(i3, j3)$ и $(i4, j4)$. Колко са всички общи клетки и каква фигура образуват.

Всички общи точки (i, j) са решения на неравенствата - $max(i1, i3) \leq i \leq min(i2, i4)$ и $max(j1, j3) \leq j \leq min(j2, j4)$. Реда и колоната са ограничени от двете страни. Фигурата която ще образуват клетките ще бъде правоъ гълник. Общия брой клетки ще е равен на лицето на правоъгълника. Този правоъгълник ще има горен ляв връх $(max(i1, i3), max(j1, j3))$ и долен десен - $(min(i2, i4), min(j2, j4))$. Съответно броят редове ще е $min(i2, i4) - max(i1, i3) + 1$ и колони - $min(j2, j4) - max(j1, j3) + 1$. Така лицето ще е равно на $(min(i2, i4) - max(i1, i3) + 1) * min(j2, j4) - max(j1, j3) + 1$.

14.5 Префиксни суми

В тази секция приемаме, че имаме дадена целочислена таблица *table* с N реда и M колони.
За да не стане объркване още от сега първия ред и стълб няма да ги използваме, но накрая ще разберем причината. Така номерацията ще започва от едно и горния ляв елемент ще бъде $table[1][1]$. Разбира се като създаваме таблицата трябва да сложим един ред и една колона в повече.
Когато говорим за правоъгълник вътре в таблицата ще считаме, че той е със страни успоредни на страните на таблицата. Така един правоъгълник се определя еднозначно от два противоположни върха.

Дефиниция 14.4. Сумата на елементите в правоъгълник имащ връх $table[1][1]$ ще наричаме префиксна сума. Префиксна сума за елемента (x, y) е сумата на числата в правоъгълника с върхове $table[1][1]$ и $table[x][y]$.

Задача 14.27. За дадени числа x и y. Намерете префиксната сума за елемента (x, y).

Разбира се ще обходим всички елементи в правоъгълника и ще ги съберем.

```
int sumInRange(int x, int y) {
    int sum = 0;
    for (int i = 1; i <= x; i++) {
        for (int j = 1; j <= y; j++) {
            sum += table[i][j];
        }
    }
    return sum;
}
```

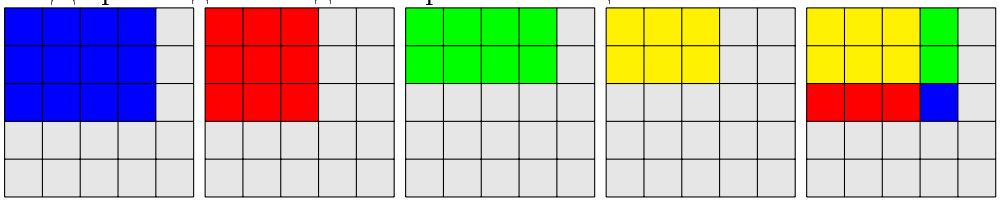
Задача 14.28. Сега искаме да решим предната задача за всеки две числа x и y, и да запишем резултатите в друга таблица с име $prefix$.

Първо трябва да намерим всички възможни двойки (x, y). След това за всяка двойка прилагаме горното решение.

```
void fillSums() {
    for (int x = 1; x <= N; x++) {
        for (int y = 1; y <= M; y++) {
            prefix[x][y] = sumInRange(x, y);
        }
    }
}
```

Задача 14.29. Може ли да намерим по-бързо решение на предходната задача, като използваме пресметнатите до момента суми?

Да разгледаме следните правоъгълници



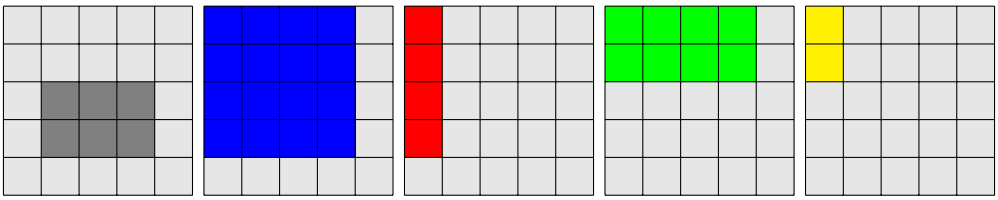
За да намерим сумата на числата в синия правоъгълник може да използваме сумите на числата в червения, зеления и жълтия. Като съберем числата от червения и зеления тези от жълтия ще се повтарят по два пъти, затова ще извадим жълтия. И накрая остава да добавим стойността на последния елемент. Така достигаме до следната зависимост:
$prefix[x][y] = prefix[x][y-1] + prefix[x-1][y] - prefix[x-1][y-1] + table[x][y]$
С което функцията ще изглежда така:

```
void fillSums() {
    for (int x = 1; x <= N; x++) {
        for (int y = 1; y <= M; y++) {
            prefix[x][y] =
            prefix[x][y-1]+prefix[x-1][y]-prefix[x-1][y-1]+table[x][y];
        }
    }
}
```

Задача 14.30. Може ли за дадено x и y да намерим стойността $table[x][y]$ само чрез таблицата $prefix$?

В горната формула прехвърляме всичко с $prefix$ от едната страна и получаваме $table[x][y] = prefix[x][y] - prefix[x][y-1] - prefix[x-1][y] + prefix[x-1][y-1]$.

Задача 14.31. За даден правоъгълник с два противоположни върха, горен ляв $(x1, y1)$ и долен десен $(x2, y2)$ как може да намерим сумата на числата в правоъгълника чрез таблицата $prefix$? Намерете връзка подобна на предишната.



За да получим сумата на числата в сивия правоъгълник трябва от синия да извадим червения и зеления и да добавим жълтия.

```
int rectangleSum(int x1, int y1, int x2, int y2) {
    return prefix[x2][y2]-prefix[x2][y1-1]-prefix[x1-1][y2]+prefix[x1-1][y1-1];
}
```

Задача 14.32. Защо не използваме първия ред и стълб в таблицата?

Всъщност ги използваме, като разчитаме стойностите там да са нули. Във формулата за сума от числата в правоъгълник използваме индексите $prefix[x1-1][y1-1]$, които могат да излизат от границите на таблицата. С допълнителни нулев ред и стълб си спестяваме този проблем.

Задача 14.33. Дадена е таблица с N реда и M колони, съставена от нули и единици. Да се намери броят на всички правоъгълници с a реда и b колони, които са съставени само от нули.

Един правоъгълник е съставен само от нули тогава и само тогава, когато сумата от числата му е нула.
Правоъгълниците които ни интересуват имат предварително зададена ширина и дължина. Т.е. всички такива правоъгълници можем да ги определим по един връх, например горен ляв или долен десен.
Ние ще използваме долен десен ъгъл да определим един правоъгълник. Така всички правоъгълници, които ни интересуват ще имат за долен десен връх ред между a и N, и колона между b и M. Всички долни десни ъгли на правоъгълниците които ни интересуват ще обходим лесно:

```
for (int x2 = a; x2 <= N; x2++) {
    for (int y2 = b; y2 <= M; y2++) {
    }
}
```

След като имаме долния десен ъгъл $(x2, y2)$ и знаем броя редове и колони, лесно може да определим горния ляв ъгъл $(x2 - a + 1, y2 - b + 1)$. Сега остава само да проверим дали сумата на числата в него е нула.

```
int ans = 0;
for (int x2 = a; x2 <= N; x2++) {
    for (int y2 = b; y2 <= M; y2++) {
        int x1 = x2-a+1;
        int y1 = y2-b+1;
        if (rectangleSum(x1, y1, x2, y2) == 0) {
            ans++;
        }
    }
}
```

Задача 14.34. Дадена е таблица с N реда и M колони, съставена от нули и единици. Да се намери най-голямото лице на правоъгълник съставен само от нули.

Едно очевидно решение е да разгледаме всички правоъгълници и от тези съставено само от нули да намерим този с най-голямо лице.
Първо трябва да обходим всички правоъгълници. Един вариант е за всяка точка да намерим всички правоъгълници с горен ляв край тази точка. Ако имаме горния ляв връх всички точки надолу и надясно са възможности за долен десен връх.

```
for (int x1=1;x1<=N;x1++) {
    for (int y1=1;y1<=M;y1++) {
        for (int x2=x1;x2<=N;x2++) {
            for (int y2=y1;y2<=M;y2++) {
                // (x1,y1) - горен ляв връх
                // (x2, y2) - долен десен връх
            }
        }
    }
}
```

Като използваме таблицата *prefix* ще може бързо да проверяваме дали даден правоъгълник е съставен само от нули.

```
int area = 0;
for (int x1=1;x1<=N;x1++) {
    for (int y1=1;y1<=M;y1++) {
        for (int x2=x1;x2<=N;x2++) {
            for (int y2=y1;y2<=M;y2++) {
                if (rectangleSum(x1, y1, x2, y2) == 0) {
                    area = max(area, (x2-x1+1)*(y2-y1+1));
                }
            }
        }
    }
}
```

Това решение е лесно за писане, но сложността е $O(N^4)$ и не е най-доброто което можем да постигнем за тази задача.

14.6 Динамично програмиране

По принцип динамичното програмиране е една доста сериозна тема за която още ни е рано. Сега обаче това което ще имаме на идея е че, за да сметнем отговора за (i, j) може да използваме вече пресметнатите отговори за предходните елементи. Всъщност бързото смятане на префиксните суми се основаваше точно на тази идея. Там за да сметнем $prefix[i][j]$ използвахме вече пресметнатите $prefix[i-1][j], prefix[i][j-1], prefix[i-1][j-1]$.
Резултатите които пресмятаме за полето (i, j) най често представляват отговора на цялата задачата за правоъгълника с върхове $(1, 1)$ и (i, j). Ако самото поле (i, j) обаче не включва този отговор почти сигурно трябва да променим нещо, така че самото поле (i, j) да е част от отговора който намерим за полето. Винаги е важно да обмисляме добре какво може да пазим като отговор за дадено поле (i, j) и как ако имаме тази информация за предходните полета може да я намерим за даденото поле (i, j).

Задача 14.35. Дадена е таблица с N реда и M колони. По колко начина можем да се придвижим от клетката $(1, 1)$ до клетката (N, M), като се движим само надолу и надясно?

Нека във втора таблица, наречена dp, да пазим резултата, като в $dp[i][j]$ ще пазим броя на всички пътища от $(1, 1)$ до (i, j). Това което ще пазим е точно каквото се иска от задачата и полето (i, j) със сигурност ще участва в отговора.
Сега въпросът е как да смятаме $dp[i][j]$. За да използваме вече получените резултати, трябва да мислим една или повече стъпки назад. В случая основния въпрос е как може да достигнем до полето (i, j). Понеже се движим надолу и надясно може да сме дошли или отгоре или отляво. Броя начини да достигнем до (i, j) е броя начини да достигнем горното поле плюс броя начини да достигнем лявото поле - $dp[i][j] = dp[i-1][j] + dp[i][j-1]$. И за да започнем от някъде трябва да кажем, че $dp[1][1] = 1$;

```
dp[1][1]=1;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= M; j++) {
```

```
        dp[i][j] = dp[i-1][j]+dp[i][j-1];
    }
}
```

Задача 14.36. Дадена е таблица с N реда и M колони, като във всяка клетка на таблицата има ябълки, записани в масива *table*. Колко най-много ябълки може да съберем като се придвижим от клетката $(1, 1)$ до клетката (N, M), като се движим само надолу и надясно и вземем всичко по пътя?

Нека във втора таблица, наречена *dp*, да пазим колко най-много ябълки може да съберем до момента. Да видим как ще намираме $dp[i][j]$. Ако дойдем отгоре може да имаме най-много $dp[i-1][j] + table[i][j]$ ябълки. Ако идваме отляво максимума е $dp[i][j-1] + table[i][j]$ ябълки. Отговорът за текущото поле е по-голямото от двете възможни числа. Така стигаме до формулата, която ни интересува $dp[i][j] = max(dp[i-1][j] + table[i][j], dp[i][j-1] + table[i][j])$ или по друг начин записано $dp[i][j] = max(dp[i-1][j], dp[i][j-1]) + table[i][j]$. Сега програмата е лесна:

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= M; j++) {
        dp[i][j] = max(dp[i-1][j],dp[i][j-1])+table[i][j];
    }
}
```

Задача 14.37. Дадена е таблица с N реда и M колони, която е разделена на правоъгълни области. Във всяка клетка на таблицата(*table*) има цяло положително число. Съседните клетки с равни числа образуват правоъгълна област. Може да има различни области съставени от клетки с едни и същи числа. На колко правоъгълни области е разделена таблицата?

Да разгледаме примерна таблица

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

Кога ще броим една област, за вече срещната? Това което правоъгълника със сигурност ще има са върховете(може и само една клетка). За това ще се концентрираме върху тях и по-конкретно върху горния ляв връх на всеки правоъгълник. Кога едно поле (i, j) е горен-ляв край на нова област? Като се загледаме в чертежа лесно се забелязва, че това е така когато полетата отгоре и отляво имат различни числа от полето (i, j).

```
int regions = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= M; j++) {
```

```
        if (table[i][j] != table[i-1][j] && table[i][j] != table[i][j-1]) {
            regions++;
        }
    }
}
```

Задача 14.38. Дадена е таблица(*table*) с N реда и M колони, която е съставена от нули и единици. Намерете страната на най-големия квадрат които може да се разположи изцяло върху полета с нули?

Първото нещо което ни идва на ум е за полето (i, j) да пазим страната на най-големия квадрат от нули, разположен в правоъгълника с върхове $(1, 1)$ и (i, j). Основния проблем с това е, че така самото поле (i, j) може да не участва в отговора (полето съвсем не е сигурно, че ще е част от най-големия квадрат). Доста трудно бихме намерили връзката за изчисляването на отговора за (i, j) с вече получените резултати. Сега да видим как може да включим полето в отговора, който ще търсим?
Най-лесно е да търсим квадрат, който съдържа това поле. Ако търсим най-големия квадрат, който съдържа полето пак ще имаме доста неясноти и варианти къде може да е разположен квадрата спрямо полето.
За да имаме по-добре дефинирано разположение на квадрата може да фиксираме, че за полето (i, j) търсим най-големия квадрат, който има за връх полето (i, j). И понеже стандартно обхождаме таблицата надясно и надолу, нека полето (i, j) е долен-десен ъгъл на квадрата който търсим.
Така за всяко поле (i, j) ще пазим страната на най-големия квадрат съставен изцяло от нули който има за долен десен връх полето (i, j). Отговорите ще записваме в допълнителната таблица *dp*.

0	1	1	1	1	1
0	0	0	0	0	0
1	0	0	0	0	0
0	1	0	0	0	1
1	0	1	1	0	0

table

1	0	0	0	0	0
1	1	1	1	1	1
0	1	2	2	2	2
1	0	1	2	?	

dp

Очевидно е, че когато $table[i][j] = 1$, то (i, j) няма как да участва в квадрат съставен изцяло от нули и $dp[i][j] = 0$.
За да сметнем $dp[i][j]$, когато $table[i][j] = 0$ нямаме много варианти. Трябва да се насочим към съседните полета и да потърсим връзка между приетите за намерени $dp[i-1][j]$ и $dp[i][j-1]$, и търсеното $dp[i][j]$. Ако разгледаме възможността за най-дълга долна страна на квадрата, може да вземем полето (i, j) и най-дългата страна на квадрат получена за полето $(i, j-1)$. Няма как да имаме по-дълга страна понеже бихме подобрили отговора за полето $(i, j-1)$. Така за дължината на долната страна на търсения квадрат имаме горна граница от $dp[i][j-1]+1$. Аналогично дясната страна не може да е по-голяма от $dp[i-1][j]+1$. По-малкото от двете числа ще ни даде

нова най-голяма потенциална страна на квадрата - $min(dp[i][j-1], dp[i-1][j]) + 1$. Сега трябва да проверим дали квадрата с такава страна и долен десен ъгъл в (i, j) е съставен само от нули. Нека $x = min(dp[i][j-1], dp[i-1][j])$. Разглеждаме квадрата със старата $x+1$ с долен десен ъгъл в (i, j). Знаем, че $table[i][j] = 0$ и полетата $(i, j-1)$ и $(i-1, j)$ са долни десни върхове на квадрати със страни x. Кои полета в квадрата с долен десен ъгъл (i, j) и страна $x+1$, не влизат в гореизброените квадрати? Ако си го представим ще видим, че това е единствено горния ляв ъгъл на квадрата, който е полето $(i-x, j-x)$. Така ако това поле е нула($table[i-x][j-x] = 0$), то $dp[i][j] = x+1$. Ако полето обаче не е нула, то $dp[i][j] = x$.

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= M; j++) {
        if (table[i][j] == 0) {
            int x = min(dp[i][j-1], dp[i-1][j]);
            if (table[i-x][j-x] == 0) {
                dp[i][j] = x+1;
            } else {
                dp[i][j] = x;
            }
        }
    }
}
```

14.7 Задачи

Задача 14.39. Дадена е числова таблица с N реда и M колони. За всеки елемент пресметнете сумата на съседните му. Съседите ако ги има са отгоре, отдолу, отляво и отдясно.

Задача 14.40. Дадена е числова таблица с n реда и m колони. Намерете най-голямото число по главния диагонал.

Задача 14.41. Дадена е числова таблица с n реда и m колони. Намерете най-малкото число по обратния диагонал.

Задача 14.42. Дадена е числова таблица с n реда и m колони. Намерете на кой ред сумата от числата е най-голяма. При повече от един ред с най-голяма сума, намерете броя на тези редове.

Задача 14.43. Дадена е числова таблица с n реда и m колони. Намерете в коя колона сумата от числата е най-голяма. При повече от една колона с най-голяма сума, намерете броя на тези колони.

Задача 14.44. Даден е квадрат, проверете дали е магически. Един квадрат е магически ако сумата от числата във всички редове, стълбове и двата диагонала са равни.

Задача 14.45. Дадена е числова таблица с n реда и m колони. Отпечатайте нейните елементи на един ред, като ги обхождате по следния начин: първия ред отляво надясно, втория отдясно наляво, третия отляво надясно и т.н.

Задача 14.46. Дадена е числова таблица с n реда и m колони. Отпечатайте нейните елементи на един ред, като ги обходите в спираловиден вид, започвайки от първия ред.

Задача 14.47. Дадена е таблица с n реда и m колони. Запълваме клетките на таблицата с числата от 1 до n^2, ред по ред, като започваме от първия ред. По дадени ред и колона на клетка намерете числото, с което сме я запълнили. И обратното, по дадено число, намерете реда и колоната.

Задача 14.48. Дадена е числова таблица с n реда и m колони. Отпечатайте таблицата като сложите всички стойности в рамки, както на примера.

Задача 14.49. Дадена е таблица с n реда и m колони. Завъртете я на 90 градуса по посока на часовниковата стрелка.

Задача 14.50. Дадена е квадратна таблица с n реда и n колони. Завъртете я на 90 градуса по посока на часовниковата стрелка без да използвате допълнително памет.

Задача 14.51. Дадена е числова таблица с n реда и m колони. Трябва да отговаряте на заявки, като по даден ред и колона трябва да пресметнете сумата на всички елементи в таблицата без тези в дадените ред и колона.

Задача 14.52. Дадена е таблица с нули и единици. Да се изведе най-голямото лице на квадрат, който е запълнен само с единици.

Задача 14.53. Дадена е таблица с нули и единици. Да се изведе най-голямото лице на квадрат, който е запълнен само с единици.

14.8 Задачи

Задача 14.54 (Есенен Турнир, 2012, D група, Номер на страница).

Задача 14.55 (Есенен Турнир, 2014, D група, Редица).

Задача 14.56 (Зимен Турнир, 2017, D група, Кратно на 3).

Задача 14.57 (Есенен Турнир, 2018, D група, Промени числото).

Задача 14.58 (НОИ, Областен кръг, 2018, D група, Дълго число).

Задача 14.59 (НОИ, Областен кръг, 2017, D група, Степен на двойката).

Задача 14.60 (НОИ, Областен кръг, 2018, D група, Сума от цифри на N!).

Задача 14.61 (НОИ, Областен кръг, 2015, D група, Максимално произведение).

Задача 14.62 (Есенен Турнир, 2018, D група, Равенство).

Задача 14.63 (Младежка Балканиада, Румъния, 2018, Ден 1, Хармонично число).

Глава 15

Дати

15.1 Основни моменти

Задача 15.1. Напишете функция, която проверява дали една година е високосна.

Трябва да отбележим, че не всички години, които са кратни на четири са високосни. Една година е високосна, ако е изпълнено едно от условията:

- годината се дели на 4 и не се дели на 100
- годината се дели на 400

Следват няколко варианта за функцията.

```
bool isLeap(int year) {
  if (year%4 == 0 && year%100 != 0) return true;
  if (year%400 == 0) return true;
  return false;
}
```

```
bool isLeap(int year) {
    if (year%4 == 0 && year%100 != 0) return true;
    return year%400 == 0;
}
```

```
bool isLeap(int year) {
    return (year%4 == 0 && year%100 != 0) || year%400 == 0;
}
```

Задача 15.2. Напишете функция, която по даден номер на месец във високосна година връща колко дена има.

Един вариант е да напишем функция с 12 условия. Ако разделим месеците на по 29, 30 и 31 дни, може да стигнем до малко по-кратка версия.

```
int monthdays(int month) {
    if (month == 2) return 29;
    if (month == 4 || month == 6 || month == 9 || month == 11) return 30;
    return 31;
}
```

Друг вариант да постигнем същото е предварително да създадем масив, където на i-ия елемент ще сложим стойност равна на броя на дните в i-ия месец.

```
int days[13] = {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int monthdays(int month) {
    return days[month];
}
```

Задача 15.3. Намерете през колко години имаме повтаряемост на високосните години. Иначе казано намерете най-малкото число *period*, така че за всяко число *year*, годините *year* и *year*+*period* да имат по равен брой дни(или и двете да са високосни, или и двете да не са).

Задачата щеше да е много лесна ако всяка четвърта година беше високосна. Тогава щяхме да имаме повтаряемост през четири години и годините *year* и $year + 4$ винаги щяха да са с равен брой дни.
В нашия случай $period = 4$ не е отговор. Ако вземем $year = 100$ и $year+period = 104$, то двете години ще имат различен брой дни.
Все пак лесно може да забележим, че *period* трябва да е кратно на четири. В противен случай *period* няма да е кратно на 4 и ако *year* е високосна година, то *year*+*period* няма да е. Значи *period* е кратно на 4.
Сега ако вземем например $year = 100$ трябва да получим две не високосни години. Понеже *period* е кратно на четири, за да бъде $year + period$ не високосна година, то *period* трябва да е кратно на 100.
И последния пример, който ще разгледаме е ако $year = 400$. Сега трябва двете години *year* и $year + period$ да са високосни. Понеже *period* е кратно на 100, то това ще е изпълнено само ако *period* е кратно на 400.
Всъщност това е и отговора $period = 400$. От функцията за високосна година по-горе е видно, че резултатът за *year* и $year + 400$ ще е един и същ.

Задача 15.4. Колко високосни години има на 400 години?

Ако гледаме точно дефиницията трябва да видим колко са кратни на 4 и не са кратни на 100. Кратните на 4 са 100, като от тях 4 са кратни на 100. Значи $100 - 4 = 96$ са кратни на 4 и не са кратни на 100. Остава да добавим тези които са кратни на 400. Всъщност има само една такава. Общо получаваме $96 + 1 = 97$ високосни години.

Задача 15.5. Ако днес е понеделник, то какъв ден ще е след точно 400 години.

За 400 години има 97 високосни и останалите 303 не са високосни. Всяка не високосна година има 365 дни. $365\%7 = 1$, което значи, че следващата година ще започне с един ден от седмицата по-напред от сегашната година. Аналогично след висикосна година имаме промяна с $366\%7 = 2$ дена. Така за всичките 400 години промяната на дните ще е $97 * 2 + 303 * 1 = 194 + 303 = 497$ дни. Но $497\%7 = 0$ което означава, че всъщност разлика като ден от седмицата няма да има, т.е. денят ще е същият като днес.

Задача 15.6. Ако днес е трети март, то какъв ден ще е след точно 400 години.

Тук идеята е вече да се усетим, че през 400 години имаме пълна повтаряемост както на дати, така и на дни от седмицата. Съответно след 400 години деня отново ще е трети март. Но важното в случая е, че ако знаем какво се случва с датите в рамките на 400 години ще може лесно да получим информация за всяка дата.

Задача 15.7. Според Грегорианския календар(този който в момента използваме) първи януари първа година е понеделник. По дадена дата $d.m.y$ намерете кой ден от седмицата е.

Ще използваме факта, че през 400 години всичко се повторя. Така вместо година y ще гледаме $y\$400$. С лека уговорка, че ако $y\$400$ е нула, ще вземем $y = 400$. Така задачата се свежда да намерим кой ден от седмицата е $d.m.y$, където $1 \leq y \leq 400$. Сега просто ще минем през всички дни от първия, докато стигнем до търсения и ще броим колко дни сме минали. Трябва да внимаваме дали да броим 29ти февруари само ако годината е високосна. Накрая трябва да вземем остатъка от номера на деня разделен на 7. Ден едно е понеделник, две ще е вторник и т.н. Само неделя няма да е 7 понеже остатъка ще бъде нула.

```
int days[13] = {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
string weekDays[7] = {"Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat"};
bool isLeap(int year) {
    return (year%4 == 0 && year%100 != 0) || year%400 == 0;
}
string dayInWeek(int d, int m, int y) {
    y = y%400;
    if (y == 0) y = 400;
    int br = 0;
    for (int year = 1; year <= 400; year++) {
        for (int month = 1; month <= 12; month++) {
            for (int day = 1; day <= days[month]; day++) {
                if (month == 2 && day == 29 && !isLeap(year)) continue;
                br++;
                if (year == y && month == m && day == d) return weekDays[br%7];
            }
        }
    }
}
```

Броя операции които ще извършим за да решим задачата в най-лошия случай ще е приблизително колкото е броя на дните през първите 400 години. Това е $97 * 366 + 303 * 365 = 400 * 365 + 97 = 146097$. Броя операции не е голям за да решим задачата за един ден, но ако имаме много дни няма да е толкова оптимално. Всъщност няма нужда всеки път да правим едни и същи сметки. Може да ги направим веднъж, да ги запишем някъде и след това само да проверяваме какво сме записали. Така ще създадем един тримерен масив, като в $daysNumber[day][month][year]$ ще запишем поредния ден на който отговаря съответната дата. Така в началото ще викаме еднократно функцията *precalculate* и след това само *dayInWeek*.

```
int daysNumber[32][13][401];
int days[13] = {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

```
string weekDays[7] = {"Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat"};
bool isLeap(int year) {
    return (year%4 == 0 && year%100 != 0) || year%400 == 0;
}
void precalculate() {
    int br = 0;
    for (int year = 1; year <= 400; year++) {
        for (int month = 1; month <= 12; month++) {
            for (int day = 1; day <= days[month]; day++) {
                if (month == 2 && day == 29 && !isLeap(year)) continue;
                br++;
                daysNumber[day][month][year] = br;
            }
        }
    }
}
string dayInWeek(int d, int m, int y) {
    y = y%400;
    if (y == 0) y = 400;
    int number = daysNumber[d][m][y]%7;
    return weekDays[number];
}
```

Задача 15.8. По дадена дата $d.m.y$ намерете кой ден поред е, ако първи януари първа година е ден едно.

Ако извикаме функцията *precalculate* ще решим задачата за първите 400 години. Ако имаме по-голяма година може да видим колко пъти 400 години се съдържат в нашата и като знаем колко дена има в 400 години ще сведем годината до по-малки или равна на 400. Текущата година не трябва да я гледаме понеже може да не е пълна. Например ако имаме дата през 2021 година. 400 се съдържа $2020/400{=}5$ пъти и така в първите 2000 години имаме 5 пъти броя на дните в първите 400 години. Остава да решим задачата да останалите години от 2001-ва до 2021-ва. Което знаем, че е същото като от 1-ва до 21-ва. А това вече сме го пресметнали. За броя на дните в първите 400 години може да използваме вече пресметнатото число 146097 или да използваме поредния номер на 31.12.400година, който е $daysNumber[31][12][400]$.

```
int dayNumber(int d, int m, int y) {
    int ans = (y-1)/400 * daysNumber[31][12][400];
  y = y%400;
  if (y == 0) y = 400;
  return ans + daysNumber[d][m][y];
}
```

Задача 15.9. Намерете на коя дата се пада n-ия пореден ден, ако първи януари първа година е ден едно.

До сега отговаряхме на обратния въпрос, като в един масив по ден, месец и година си пазехме поредния ден. Сега трябва да пазим нещата по друг начин така, че да имаме достъп до датата по номер на ден. В момента в който записваме информацията

трябва да я записваме още някъде. Един вариант е да имаме три масива, където по номер на деня да пазим ден, месец и година. В случая ще използваме масив от $pair < pair < int, int >, int >$. Така лесно ще решим задачата за номер на ден, които е в първите 400 години. В противен случай ще трябва да видим колко пъти имаме дните за 400 години и да сметнем правилно годината.

Така функцията *precalculate* ще изглежда ето така

```
pair<pair<int, int>, int> daysReverse[146098];
int days[13] = {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
bool isLeap(int year) {
    return (year%4 == 0 && year%100 != 0) || year%400 == 0;
}
void precalculate() {
  int br = 0;
  for (int year = 1; year <= 400; year++) {
    for (int month = 1; month <= 12; month++) {
      for (int day = 1; day <= days[month]; day++) {
        if (month == 2 && day == 29 && !isLeap(year)) continue;
        br++;
        daysReverse[br] = {{day, month}, year};
      }
    }
  }
}
pair<pair<int, int>, int> day(int n) {
    pair<pair<int, int>, int> ans = daysReverse[(n-1)%146097];
    ans.second += (n-1)/146097*400;
    return ans;
}
```

15.2 Задачи

Задача 15.10 (НОИ, Общински кръг, 2015, D група, Мобилно приложение).

Задача 15.11 (НОИ, Областен кръг, 2017, D група, Биологични ритми).

Глава 16

Метод на показалките

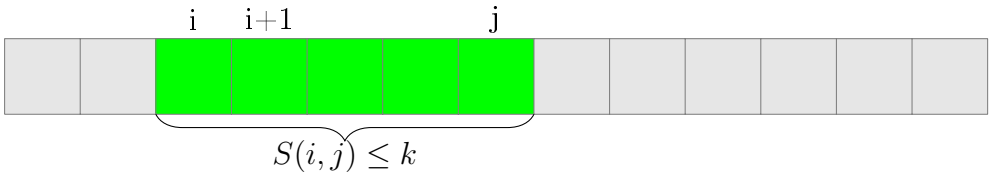
16.1 Уводни задачи за най-дълга последователност

Задача 16.1. Дадена е редицата $a_0, a_1, ..., a_{n-1}$ с положителни числа и число k. Намерете дължината на най-дългата последователност от числа със сума по-малка или равна на k.

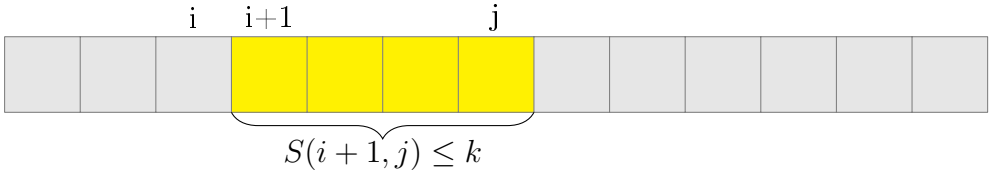
Решение. Тази задача има различни варианти за бавни решения, които включват намирането на сумата във всички възможни последователности. Например можем за всяко възможно начало и край на последователност да съберем числата в нея, което ще има кубична сложност. Ако използваме префиксни суми ще забързаме събирането и сложността ще стане квадратна.
Ще разгледаме още едно квадратно решение, с което ще подложим основите на метода на показалките. Целта отново е да намерим сумата на всички последователности. Нека $S(i,j) = a_i + a_{i+1} + ... + a_j$ На i-тата стъпка ще търсим сумата на всички последователности с начало числото a_i. Това са сумите $S(i,i)$, $S(i,i+1)$, ..., $S(i,n-1)$. Основното наблюдение тук е, че за да намерим сумата $S(i,j)$ може да използваме току що пресметнатата сума $S(i,j-1)$, понеже $S(i,j) = S(i,j-1) + a_j$. Ето как би изглеждало намирането на всички суми с начало i-ия елемент.

```
int sum = 0;
for (int j = i; j < n; j++) {
    sum += a[j];
    // тук sum е равно на S(i,j) = a_i+...+a_j
}
```

Като извикаме този код за всички възможни начални елементи ще получим сумите на всички последователности. Може да добавим една оптимизация и когато sum стане по-голямо от k да спрем цикъла, но така или иначе решението ще е квадратно. От тук нататък става интересно - да решим задачата линейно. Броя различни последователности е $n(n+1)/2$ и ако търсим сумата на всички няма как да стане по-бързо от квадратно. За това трябва да променим подхода. Преди малко за всяко възможно начало a_i търсихме всички суми започващи от a_i. В условието е казано, че се търси най-дългата последователност със сума по-малка или равна на k. Така за всяко възможно начало a_i трябва да се насочим към търсеното в условието, а не към всички възможни суми. Следващото важно нещо е, да видим може ли да използваме намереният отговор за едно начало, при търсенето на отговор за следващото начало.



Нека (i, j) е най-дългата последователност с начало i-ия елемент и сума по-малка или равна на k. Да разгледаме сумите с начало $i+1$-ия елемент.



Това което лесно се забелязва е, че всички суми от $i+1$-ия до j-ия елемент са по-малки или равни на k, понеже $S(i+1, i+1) \leq ... \leq S(i+1, j) \leq S(i, j) \leq k$. Без да смятаме нищо знаем, че последователността $a_{i+1}, ..., a_j$ има сума по-малка или равна на k и тази сума е равна на $S(i+1, j) = S(i, j) - a_i$. От тук нататък има смисъл само да добавяме елементи след j-ия с цел да получим по-дълга последователност с начало $i+1$-ия елемент. Така на всяка стъпка за дадено начало в i-ия елемент, ще намираме най-голямото j, такова че, $S(i, j) \leq k$ - т.е. j е края на най-дългата последователност с начало i-ия елемент и сума по-малка или равна на k. Ако такава последователност няма ($a_i > k$), то ще намираме $j = i - 1$.
В началото на всяка стъпка i ще сме си гарантирали, че j не носи полезна информация от предната стъпка ($j = i - 1$), или $i \leq j$ и $S(i, j) \leq k$, както и че имаме сумата на числата в интервала $[i, j]$. И в двата случая започваме да разширяваме интервала с $j+1$-ия, $j+2$-ия, и т.н, докато можем. Можем да добавим $j+1$-ия елемент ако сумата до момента плюс a_{j+1} не надвишава k. Ако няма $j+1$-ви елемент или не може да го добавим, то значи вече сме намерили най-дългата търсена последователност с начало i-ия елемент. Остава да приготвим нещата за следващата стъпка. Понеже следващото начало е $i+1$-ия елемент, то ако имаме някаква последователност трябва да махнем числото a_i от текущата сума. Ако нямаме последователност означава, че $j = i - 1$ и интервала е $[i, i-1]$. В този случай искаме на следващата стъпка да започнем с интервал $[i+1, i]$, а не $[i+1, i-1]$, за това слагаме $j = i$.

```
int solve() {
    int sum = 0;
    int j = -1;
    int ans = 0;
    for (int i = 0; i < n; i++) {
        // докато има следващ елемент и сумата не надвишава k
        // добавяме следващия елемент към текущата последователност
        while (j+1 < n && s+a[j+1] <= k) {
            s += a[j];
            j++;
        }
        ans = max(ans, j-i+1);

        // ако i-ия елемент участва в текущата последователност го изваждаме
        // иначе слагаме първоначално края на последоватерността от $i+1$ да е i
```

```
        if (j >= i) s -= a[i];
        else j = i;
    }
    return ans;
}
```

Задача 16.2. Дадена е редицата $a_0, a_1, ..., a_{n-1}$ с положителни числа и число k. Намерете броя на последователностите от числа със сума по-малка или равна на k.

Решение. Ако за дадено число i най-дългата последователност с начало този елемент е (i, j), то всички последователности $(i, i), ..., (i, j)$ ще имат сума по-малка или равна на k. Така броя на търсените последователности с начало i-ия елемент е равен на $j - i + 1$. Остава да направим същото за всяко възможно начало да и да съберем получените резултати.

16.2 Обща рамка на метода на показалките за намиране на последователности

Метода на показалките за последователност най-често се използва в следните два случая:

- намиране на най-дългата последователност, която отговаря на дадено условие
- намиране на броя последователности отговарящи на дадено условие

Важно е да правим разлика с метода за разделяне на последователности. При разделянето на последователности има прегради между елементите и така един елемент е част от точно една последователност. При показалките всеки елемент може да участва в повече от една последователност, което значително усложнява нещата. Метода на показалките разчита на две основни свойства на търсените последователности:

- ако имаме последователност (i, j) отговаряща на търсенето, бързо да проверяваме дали последователността $(i, j + 1)$ също върши работа и съответно да добавяме $j + 1$-ия към текущата последователност
- ако имаме последователност (i, j) отговаряща на търсенето, то последователността $(i + 1, j)$ също да върши работа и съответно бързо да премахваме информацията за i-ия елемент от текущата последователност.

Има различни варианти за писането на този алгоритъм, но тук ще се опитаме да покажем една обща имплементация, която с леки промени решава голяма част от срещаните задачи.

```
void solve() {
    // маркер за край на текущия интервал
    int j = -1;

    // намиране на най-дългата търсена последователност с начало i-ия елемент
```

```
    for (int i = 0; i < n; i++) {
        // докато има следващ елемент и
        // може да го добавим към текущата последователност
        while (j+1 < n && canAdd(a[j+1])) {
            // добявяме a[j+1] към текущата последователност
            // добавяме информация за новия елемент, ако е необходимо
            j++;
        }

        // най-дългият търсен интервал с начало i-ия елемент е [i,j]
        // тук проверяваме каквото ни трябва за намерения интервал

        // ако i-ия елемент участва в текущата последователност го изваждаме
        // иначе слагаме първоначално края на последоватерността от i+1 да е i
        if (j >= i) {
            // изтриваме a[i] от текущата последователност
            // махаме добавената информация за този елемент, ако има такава
        } else {
            j = i;
        }
    }

}
```

16.3 Още задачи за последователности

Задача 16.3. Дадена е редицата $a_0, a_1, ..., a_{n-1}$ с положителни числа не по-големи от един милион. Намерете дължината на най-дългата последователност без повтарящи се числа.

Решение. За да докараме задачата до решение с метода на показалките трябва да видим две неща. Първо как намираме най-дългата търсена последователност с начало даден елемент. След това ако най-дългата последователност с начало i-ия елемент е (i, j) трябва последователността $(i + 1, j)$ да отговаря на условията и за да намерим най-дългата последователност с начало $i + 1$-ия елемент трябва да продължим от j-ия елемент за десен край.
Да видим как може да намерим най-дългата последователност с начало i-ия елемент. Разглеждаме последователностите $(i, i), (i, i + 1), ...$ Трябва за дадена последователност (i, j) бързо да проверяваме дали има повтарящ се елемент. За целта може да използваме някаква информация за предишната последователност $(i, j - 1)$. Понеже последователността $(i, j - 1)$ трябва да отговаря на условието, то там няма повтарящ се елемент и в последователността (i, j) единственото число, което може да се повтаря е a_j. Така трябва да проверим дали в интервала $(i, j - 1)$ се среща числото a_j. Най-бързият начин за това е ако имаме масив за броене, не случайно числата са до един милион. Така ще поддържаме масив `bool br[]`, където `br[x]` ще показва дали числото x вече сме го срещнали.
Сега ако сме намерили, че най-дългата последователност с начало i-ия елемент е (i, j), да видим какво може да кажем за последователностите с начало $(i + 1, j)$. От

това, че в интервала (i, j) нямаме повтарящ се елемент следва, че и в интервала $(i + 1, j)$ също няма повтарящ се елемент. Продължаваме да увеличаваме j докато намерим най-дългата търсена редица с начало $i + 1$-ия елемент. Махането на i-ия елемент се изразява в това да отбележим премахването от масива за броене.
Последно ще отбележим, че в тази задача всяка последователност от един елемент отговаря на условието. Така винаги трябва да махаме информацията за i-ия елемент.

```
int solve() {
    int br[1000001] = {};
    int j = -1;
    int ans = 0;
    for (int i = 0; i < n; i++) {
        while (j+1 < n && !br[a[j+1]]) {
            br[a[j+1]] = true;
            j++;
        }
        ans = max(ans, j-i+1);

        br[a[i]] = false;
    }
    return ans;
}
```

16.4 Задачи

Задача 16.4 (Зимен Турнир, 2016, D група, Улица с паметници).

Задача 16.5 (НОИ, Национален кръг, 2018, E група, Двуцветна лента).

Задача 16.6 (Летен Турнир, 2019, D група, Подредица).

Задача 16.7 (НОИ, Общински кръг, 2020, B група, Редица).

Задача 16.8 (Пролетен Турнир, 2008, C група, Подредици).

Задача 16.9. Проверете дали дадено число n може да се представи като сума от последователни прости числа.

Задача 16.10. За дадена редица с положителни числа до един милион и дадено число k, намерете броя последователности, където никой елемент не се среща повече от k пъти.

Задача 16.11 (Есенен Турнир, 2012, D група, Пощальон).