Състезателно програмиране за 6 клас

Петър Петров, Марин Шаламанов

2 август 2020 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 последвано 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 къдрави скоби за начални стойности по подразбиране - `int a[5] = {}`.

1.3.6 Скриване

В обхвата на дадена променлива никога не декларираме 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 Къдрави 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

Вектор

2.1 Дефиниция

Дефиниция 2.1 (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.

2.2 Създаване

2.2.1 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 2.1. Как ще декларираме вектор, които е подобен на следния масив: `int a[1000]`.

Решение.

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

2.2.2 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 2.2. Как ще декларираме вектор с 1000 символа, които в началото са буквата а.

Решение.

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

2.2.3 Празен вектор

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

```
vector<тип> име;
```

2.3 Достъп до елементите

2.3.1 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
```

2.3.2 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 2.3. 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];
}
```

2.3.3 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]`.

2.4 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.

2.4.1 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 2.4. Създали сме празен 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);
}
```

2.4.2 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 2.5. 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();
}
```

2.5 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 3

Стринг

3.1 Дефиниция

Дефиниция 3.1 (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.

3.1.1 Създаване

Имаме няколко възможности за създаване:

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

3.1.2 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`.

3.1.3 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;
```

3.1.4 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;
```

3.2 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.

3.2.1 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
```

3.2.2 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
```

3.3 Стринг във функция

При използването на стринг във функция имам няколко варианта.

3.3.1 Глобални променливи

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

3.3.2 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.

3.3.3 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.

3.3.4 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.

3.4 Число към низ и низ към число

Задача 3.1. 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 3.2. 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 3.3. 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;
}
```

3.5 Поднизове

Задача 3.4. 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 3.5. 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 3.6. 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 3.7. 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 3.8. 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 3.9. 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;
}
```

3.6 Палиндроми

Дефиниция 3.2 (Палиндром)**.** Палиндром е низ, които е огледален - чете се еднакво от ляво надясно и отдясно наляво. Например думите "потоп"и "капак"са палиндроми.

Задача 3.10. 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 3.11. 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 3.12. 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;
}
```

3.7 Задачи

Задача 3.13 (Есенен Турнир 2014, D група, Подниз).

Задача 3.14 (Есенен Турнир 2011, D група, Симетрична редица).

Задача 3.15 (НОИ, Общински кръг 2016, D група, Уравнение).

Задача 3.16 (Есенен Турнир 2019, D група, Отгатни цифрата).

Задача 3.17 (Зимен Турнир 2017, D група, Подобни думи).

Задача 3.18 (НОИ, Областен кръг 2015, C група, Пакети от думи).

Глава 4

Разделяне на последователности

4.1 Дефиниция

Дефиниция 4.1 (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 4.1. 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 4.2. 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 едни, 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 4.2 (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 4.3. 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 4.4. 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 - $3, 5|4|1, 2, 3$.

4.2 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 4.5. 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 4.6. 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 4.7. 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 4.8. 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 4.9. 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;
}
```

4.3 Определяне на начало или край

Задача 4.10. 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 4.11. 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 4.12. 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.

4.4 Определяне на начало и край

Задача 4.13. 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)) {
            ednings.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)) {
            ednings.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 4.14. 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 4.15. 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 4.16. 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 4.17. 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 4.18. 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;
}
```

4.5 Задачи

Задача 4.19 (Есенен Турнир 2008, Е група, Шоколад).

Задача 4.20 (Софийски пролетен турнир 2019, Е група, Съгласни).

Задача 4.21 (Зимен Турнир 2012, Е група, GPS).

Задача 4.22 (НОИ, Областен кръг 2017, Е група, Картончета).

Задача 4.23 (НОИ, Национален кръг 2014, Е група, Символ).

Задача 4.24 (Зимен Турнир 2014, Е група, Спирка).

Задача 4.25 (Зимен Турнир 2014, Е група, Камера).

Задача 4.26 (НОИ, Областен кръг 2013, Е група, Свиване).

Задача 4.27 (НОИ, Областен кръг 2016, Е група, Кодиране).

Задача 4.28 (НОИ, Общински кръг 2016, C група, Тетрис).

Задача 4.29 (Пролетен Турнир 2008, D група, Синоптици).

Задача 4.30 (НОИ, Областен кръг 2010, D група, Тетрис).

Задача 4.31 (НОИ, Областен кръг 2011, D група, Ненамаляваща редица).

Задача 4.32 (НОИ, Областен кръг 2012, D група, Цветни топчета).

Глава 5

Префиксни суми

5.1 Загрявка

Задача 5.1. 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 5.2. 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;
}
```

5.2 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 5.1 (Префиксна сума)**.** Да разгледаме редицата $a_1, a_2, ..., a_n$. Всяка сума от вида $s_i = a_1 + ... + a_i$, която включва първите няколко последователни числа се нарича префиксна сума.

Задача 5.3. 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.

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

const int MAXN = 1000000;
int a[MAXN+1], p[MAXN+1];

int main() {
    int n, q;
    cin >> n >> q;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }
    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;
    }

    return 0;
}
```

Задача 5.4. 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 $p[i] = a[1] + ... + a[i]$.

	0	1	2	3	4	5	6	7
a $\longrightarrow$	0	8	6	3	5	7	3	9

	0	1	2	3	4	5	6	7
p ⟶	0	8	14	17	22	29	32	41

Лесно може да се усетим, че $a[3]+a[4]+a[5]+a[6] = (a[1]+a[2]+a[3]+a[4]+a[5]+a[6])-(a[1]+a[2]+a[3]) = p[6]-p[3]$. 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 число от началото до r минус сумата на всички числа преди l.

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

const int MAXN = 1000000;
int a[MAXN+1], p[MAXN+1];

int main() {
    int n, q;
    cin >> n >> q;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }
    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;
    }

    return 0;
}
```

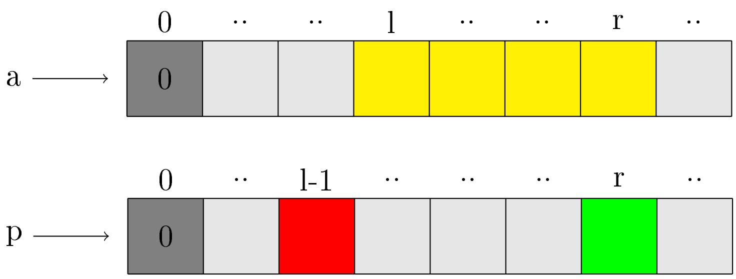
Задача 5.5. Дадена е последователност от n букви - $c_1, c_2, ..., c_n$ и q заявки. Всяка буква е a, b или c. За всяка заявка са дадени две числа l и r, и трябва да намерите най-често срещаната буква в интервала $[l, r]$. Ако има няколко такива букви изведете най-предната в азбуката.
Ограничения: $1 \leq n \leq 10^6, 1 \leq q \leq 10^6, 1 \leq l \leq r \leq 10^6$.

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.

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

const int MAXN = 1000000;
char c[MAXN+1];
int pa[MAXN+1], pb[MAXN+1], pc[MAXN+1];

int main() {
    int n, q;
    cin >> n >> q;
    for (int i = 1; i <= n; i++) {
        cin >> c[i];
    }
    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;
        int answerLetter = 'a';
        int asnwerCount = pa[r]-pa[l-1];

        if (pb[r]-pb[l-1] > answerCount) {
            answerLetter = 'b';
            asnwerCount = pb[r]-pb[l-1];
        }

        if (pc[r]-pc[l-1] > answerCount) {
            answerLetter = 'c';
            asnwerCount = pc[r]-pc[l-1];
        }

        cout << answerLetter << endl;
    }

    return 0;
}
```

Задача 5.6. 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.

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

const int MAXN = 1000000;
int a[MAXN+1], p[MAXN+1];

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
    }
    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;
        }
    }

    if (hasZeroSum) {
        cout << "yes" << endl;
    } else {
        cout << "no" << endl;
    }

    return 0;
}
```

5.3 Суфиксни суми

Дефиниция 5.2 (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 5.7. 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.

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

const int MAXN = 1000000;
int a[MAXN+2], s[MAXN+2];

int main() {
    int n, q;
    cin >> n >> q;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
```

```
    }
    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;
    }

    return 0;
}
```

Задача 5.8. Дадена е редица с n числа - $a_1, a_2, ..., a_n$. 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: $1 \leq n \leq 10^6, 0 \leq a[i] \leq 10^6$.

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.

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

const int MAXN = 1000000;
int a[MAXN+2], s[MAXN+2];

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
```

```
        cin >> a[i];
        a[i] = a[i]%10;
    }
    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;
    }

    return 0;
}
```

5.4 Задачи

Задача 5.9. 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 5.10. Летен Турнир 2018, E група, E3.Редица

Задача 5.11. НОИ 3 2017, E група, E3. Редица от правоъгълници

Задача 5.12. НОИ 3 2015, D група, D4. Пропуснат множител

Задача 5.13. НОИ 2 2013, C група, C3. Думи

Задача 5.14. НОИ 3 2019, D група, D3. Поздрави

Задача 5.15. НОИ 2 2020, C група, C2. Диапазон

Задача 5.16. Втора контрола за младежкия национален 2018, C група, CK4. Баланс

Глава 6

Груба сила

6.1 Дефиниция

Дефиниция 6.1 (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.

6.2 Груба сила с двойки и тройки

Задача 6.1. 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 6.2. 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 6.3. 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 6.4. 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 6.5. 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 6.6. 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;
}
```

6.3 Груба сила с комбинации

Задача 6.7. 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 6.8. Изведете на екрана всички редици с по 4 числа, съставени от числата от 0 до 3, които може да се повтарят.

Задача 6.9. 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 6.10. 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 6.11. Дадени са десет предмета с тегла $a_0, \ldots, a_9$ 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 $i0, i1, \ldots$ ще използваме $p[0], p[1], \ldots$. 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 6.12. 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 6.13. 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 - десет 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.

6.4 Груба сила с пермутации

Дефиниция 6.2 (Пермутация)**.** Пермутация на числа наричаме всяка една тяхна подредба в редица. Всички възможни подредби дават всички пермутации на числата.

Задача 6.14. 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 6.15. 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 6.16. 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 6.17. 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 число a[i]
        b[p[i]] = a[i];
```

```
    }

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

6.5 Задачи

Задача 6.18 (Есенен Турнир 2017, E група, Карти).

Задача 6.19 (Есенен Турнир 2017, E група, Израз).

Задача 6.20 (Есенен Турнир 2018, E група, Дини).

Задача 6.21 (Есенен Турнир 2015, E група, Асансьор).

Задача 6.22. 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 6.23. 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 6.24 (Зимен Турнир 2017, D група, Кратно на 3).

Задача 6.25 (Есенен Турнир 2018, D група, Промени числото).

Задача 6.26 (НОИ - Национален кръг 2020, D група, Фалшивата монета).