

Състезателно програмиране за 6 клас

Петър Петров, Марин Шаламанов

18 юли 2020 г.

Съдържание

0.1	Въведение	2
1	Стил на писане	3
1.1	Основни правила	3
1.2	Подреждане на кода	3
1.2.1	Инdentация	3
1.2.2	Една команда на ред	3
1.2.3	Къдрави скоби	3
1.2.4	if, if-else, if-else-if-else	4
1.2.5	for	4
1.2.6	while	4
1.2.7	do-while	4
1.3	Променливи	5
1.3.1	Предназначение на променлива	5
1.3.2	Именуване	5
1.3.3	Глобални или локални	5
1.3.4	Създаване	5
1.3.5	Начални стойности	5
1.3.6	Скриване	5
1.4	Функции	6
1.4.1	Къдрави скоби	6
1.4.2	Връщана стойност	6
1.4.3	Частни случай	6
2	Груба сила	7
2.1	Дефиниция	7
2.2	Груба сила с двойки и тройки	7
2.3	Груба сила с комбинации	10
2.4	Груба сила с пермутации	14
2.5	Задачи	16

0.1 Въведение

Неща, които трябва да споменем тук:

- Целта на книгата е представи на състезателите от D група, всички теми и материал, които трябва да знаят за да се представят 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 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.

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 случаи

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 къществената част от кода не се налага да бъде отместена.

Глава 2

Груба сила

2.1 Дефиниция

Дефиниция 2.1 (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.

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.

2.2 Груба сила с двойки и тройки

Задача 2.1. 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 - $(i, 0), (i, 1), \dots, (i, n - 1)$.

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. \square

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



```
    }
}
```

Задача 2.2. 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 за извеждаме повторенията.

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 който 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 - $i + 1, \dots, n - 1$. 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 2.3. 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 2.4. 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 $a[i], \dots, a[j]$.

```
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 2.5. 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 2.6. 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;
}
```

□

2.3 Груба сила с комбинации

Задача 2.7. 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 2.8. Изведете на екрана всички редици с по 4 числа, съставени от числата от 0 до 3, които може да се повтарят.

Задача 2.9. 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 2.10. 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 2.11. Дадени са десет предмета с тегла a_0, \dots, a_9 и три раници с вместимост 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.

```
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 намира 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 2.12. 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 2.13. 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. \square

2.4 Груба сила с пермутации

Дефиниция 2.2 (Пермутация). Пермутация на числа наричаме всяка една тяхна подредба в редица. Всички възможни подредби дават всички пермутации на числата.

Задача 2.14. 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]. \square

Задача 2.15. 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 $n! = 1.2.3...n$. \square

Задача 2.16. 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 2.17. 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 - $\text{int } p[4]$, 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 $\text{int } b[4]$, $b[i]$ 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];
}

```

□

2.5 Задачи

Задача 2.18 (Есенен Турнир 2017, Е група, Карти).

Задача 2.19 (Есенен Турнир 2017, Е група, Израз).

Задача 2.20 (Есенен Турнир 2018, Е група, Дини).

Задача 2.21 (Есенен Турнир 2015, Е група, Асансьор).

Задача 2.22. 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 2.23. Дадени са n предмета, като всеки има тегло w_i и стойност v_i . Дадена е раница с вместимост w . Каква е най-голямата обща стойност на предмети, която 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 2.24 (Зимен Турнир 2017, D група, Кратно на 3).

Задача 2.25 (Есенен Турнир 2018, D група, Промени числото).

Задача 2.26 (НОИ - Национален кръг 2020, D група, Фалшивата монета).