

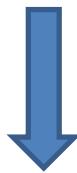


Алгоритми и програмиране

Задачи

Sample Input 0

```
5  
5 2 3 4 1
```



Sample Output 0

```
5 4
```

- **Най-голяма стойност.** Дадена е редица a_1, a_2, \dots, a_n от цели положителни числа. Да се намери двойка цели числа (a_i, a_j) , за които $a_i > a_j$ и $P_{a_i}^{a_i}$ е максимално:

$$P_{a_j}^{a_i} = \frac{a_i!}{(a_i - a_j)!}$$

за $i = 1, 2, \dots, n, j = 1, 2, \dots, n$

- **Input Format**

- Стандартният вход за всеки пример съдържа два реда. На първия ред се въвежда стойност за n , а на втория - редица от n числа, разделени с интервал.

- **Constraints**

$$2 \leq n \leq 10^5;$$

$$1 \leq a_i \leq 30, i = 1, 2, \dots, n.$$

- **Output Format**

- На един ред на стандартния изход за всеки пример се извеждат две положителни цели числа $a_i, a_j (a_i > a_j)$, които принадлежат на входната редица и за които $P_{a_j}^{a_i}$ е максимално.

Задачи

- Най-голяма стойност - Решение

- **Наивен подход:** Разглежда се всяка двойка от цели числа a_i, a_j , изчислява се $P_{a_j}^{a_i}$, като се търси максималната стойност за $P_{a_j}^{a_i}$.

```
int main()
{
    int n;
    while(cin>>n)
    {
        int arr[n];
        for(int i = 0; i < n; i++)
            cin>>arr[i];

        // Brute force
        int tmpRes = 0, first, second, maxV = 0;
        for(int i = 0; i<n-1; i++)
        {
            for(int j = i+1; j<n;j++)
            {
                if (arr[i]>=arr[j])
                    tmpRes = fact(arr[i])/(fact(arr[i] - arr[j]));
                else
                    tmpRes = fact(arr[j])/(fact(arr[j] - arr[i]));
                if (tmpRes > maxV)
                {
                    maxV = tmpRes;
                    first = arr[i];
                    second = arr[j];
                }
            }
            cout<<"\n"<<first<<" "<<second<<endl;
        }
        return 0;
    }
}

int fact(int x)
{
    int res = 1;
    for(int i = 1; i <= x; i++)
        res = res * i;
    return res;
}
```

Задачи

- **Най-голяма стойност - Решение**

- **Ефективен подход:** Нека $A = \{a_1, a_2, a_3, \dots, a_n\}$. Тъй като

$$P_{a_j}^{a_i} = \frac{a_i!}{(a_i - a_j)!} = a_i * (a_i - 1) * (a_i - 2) * \dots * (a_i - a_j + 1),$$
$$a_i, a_j \in A, a_i \leq a_j .$$

то може да се докаже, че $P_{a_j}^{a_i}$ е максимално, когато a_i е най-голямото число в A , а $a_i - a_j$ е минимално, където $a_i, a_j \in A$. По този начин задачата се свежда до намиране на двета най-големи елемента във входната редица.

Задачи

- **Тройки.** Петър много обичал математиката и по-точно обичал да групира по различен начин различни числа. Веднъж Петър си намислил три цели числа X , Y и Z . След това създал групи от по три числа (x, y, z) , където $1 < x \leq X$, $1 < y \leq Y$ и $1 < z \leq Z$ и се опитал да намери броя на тройките (x, y, z) , които задоволяват условието $x * z > y^2$, но не успял. Напишете програма, която да помогне на Петър да намери броя на тройките (x, y, z) , за които $x * z > y^2$, където $1 < x \leq X$, $1 < y \leq Y$ и $1 < z \leq Z$.

- **Constraints**

- $1 \leq X \leq 10^5$
- $1 \leq Y \leq 10^5$
- $1 \leq Z \leq 10^5$

Примерен Вход	Примерен Изход
3 2 2	6
3 3 3	11

- **Input Format**

- За всеки тестов пример на единственият ред от стандартният вход се задават стойности за X , Y и Z , разделени с интервал.

- **Output Format**

- За всеки тестов пример на единственият ред на стандартният изход се извежда броя на тройките (x, y, z) , за които $x * z > y^2$.

Задачи

- Тройки – Решение

```
int main()
{
    int X, Y, Z;

    while(cin>>X>>Y>>Z)
    {
        cout<<threeLoops(X, Y, Z);
    }
    return 0;
}
```

```
// Brute force
long long threeLoops(int X, int Y, int Z)
{
    long long ans = 0;
    for (int x = 1; x <= X; x++) {
        for (int y = 1; y <= Y; y++) {
            for (int z = 1; z <= Z; z++) {
                if (x * z > y * y)
                    ans++;
            }
        }
    }
    return ans;
}
```

Задачи

- Тройки - Решение
 - **Ефективен подход:** Нека да преброим всички тройки за $y = k$, $k = 1 \dots Y$.
 1. За $y = k$ трябва да намерим всички $x = i$ и $z = j$, които удовлетворяват $i * j > k^2$;
 2. За $x = i$, намираме най-малкото $z = j$, което отговаря на условието. Тъй като $z = j$ удовлетворява зададеното условие, следователно $z = j + 1, z = j + 2, \dots$, също ще удовлетворяват условието. Лесно може да преброим всички тройки, за които $x = i$ и $y = k$.
 3. Също така, ако за някои $x = i, z = j$ е най-малката стойност, за която даденото условие е изпълнено, то може да кажем, че за $x = j$ и всички $z \geq i$ също удовлетворяват условието. Условието ще бъде изпълнено за $x = j + 1$ и всички $z \geq i$, което означа, че за всички стойности $x \geq j$ и $z \geq i$ условието ще бъде удовлетворено.
 - Горното наблюдение ни помага да преброим всички тройки, за които $y = k$ и $x \geq j$ лесно. Сега трябва да преброим всички тройки, за които $y = k$ и $i < x < j$. Следователно за дадена стойност $y = k$, трябва да отидем нагоре до $x = \sqrt{k}$.

Задачи

- **Сума на подредица.** Дадена е редица от n различни цели числа, a_1, a_2, \dots, a_n . Да се намери подредица от m елементи,

$$a_{i_1}, a_{i_2}, \dots, a_{i_m}, 1 \leq i_k < i_{k+1} \leq n, k = 1, 2, \dots, m$$

с най-голяма възможна сума $S = \sum_{k=1}^m a_{i_k}$

- **Input Format**

- За всеки пример на стандартния вход са зададени числата n и m , след които още n числа – членовете на редицата.

- **Constraints**

$$\begin{aligned} 1 \leq m \leq n \leq 200 \\ -1000 \leq a_i \leq 1000 \end{aligned}$$

- **Output Format**

- За всеки пример на отделен ред да се отпечата намерената подредица.

Sample Input 0

```
5 3
1 8 2 9 3
```



Sample Output 0

```
8 9 3
```

Задачи

- **Маска.** Мaska наричаме низ състоящ се от малки латински букви и една звезда (*). По зададена маска и редица от низове, да се определят низовете, които отговарят на маската. Даден низ отговаря на маската, ако след заместването на звездата с подходящо избрани (произволен брой) малки латински букви, маската и низа съвпадат. Например *he^llo*, *hom^o* и *hohohoho* отговарят на маската *h^{*}o*, а *hoh* - не.
 - **Input Format**
 - На първия ред на стандартния вход е зададен броят на тестовите примери. Всеки от тях започва с числото N – броят на низовете, които ще се тестват. Вторият ред съдържа маската – низ, съставен от малки латински букви и точно една звезда (ASCII код 42). Следват N реда, като на всеки от тях е зададен по един тестов низ, съставен от не повече от 100 малки латински букви.
 - **Constraints**
 - $1 \leq N \leq 100$
 - Дължината на маската е не повече от 100.
 - **Output Format**
 - На стандартния изход на отделен ред да се извежда YES или NO за поредния низ от текущия пример в зависимост от това дали той отговаря на маската или не.

Задачи

- Маска

Sample Input 0

```
2
4
a*a
alabala
ananas
abracadabra
aaa
6
h*n
hkjdfjfdshodfhscbajkfnxyemfvsn
honijezakon
atila
je
bio
hun
```



Sample Output 0

```
YES
NO
YES
YES
YES
YES
NO
NO
NO
YES
```

Състезателно програмиране

Задачи

Sample Input 0

```
4
0 0
1 1
2 0
2 2
```



Sample Output 0

```
3
```

- **Брой триъгълници.** Дадени са (x, y) координатите на n точки в двумерна декартова координатна система. Напишете програма, която намира броя на възможните триъгълници (с положителни лица), които могат да се образуват с дадените точки.
- **Input Format**
 - За всеки тестов пример на първия ред от стандартния вход се въвежда стойност за n . След това n реда съдържащи координати на точки в двумерна декартова координатна система. Всяка точка е на отделен ред и координатите са разделени с интервал.
- **Constraints**
 - $3 \leq n \leq 1000$
 - Координатите на всички точки са цели числа и са по-малки или равни на 100.
- **Output Format**
 - За всеки тестов пример на отделен ред на стандартния изход се извежда броя на възможните триъгълници с дадените точки.

Задачи

- **Брой триъгълници – Решение**

- Едно просто решение е да се провери дали детерминантата на трите избрани точки е ненулева или не. Детерминантата дава площта на триъгълника (правило на Крамер). Например, площта на триъгълника с ъгли при (x_1, y_1) , (x_2, y_2) и (x_3, y_3) се дава от:

$$Area = \pm \frac{1}{2} \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix}$$

- Можем да решим задачата, като вземем всички възможни комбинации от 3 точки и да намерим детерминантата.

Задачи

- Брой триъгълници – Решение

```
1 #include <iostream>
2
3 using namespace std;
4
5 // A point in 2D
6 struct Point
7 {
8     int x, y;
9 };
10
11 // Returns determinant value of three points in 2D
12 bool determinantValue(int x1, int y1, int x2, int y2, int x3, int y3)
13 {
14     if((x1*(y2 - y3) - y1*(x2 - x3) + 1*(x2*y3 - y2*x3)) != 0)
15         return true;
16     else
17         return false;
18 }
19
20 // Returns the number of possible triangles
21 // from given array of points in 2D.
22 int countPoints(Point arr[], int n)
23 {
24     int result = 0; // Initialize result
25
26     // Consider all triplets of points given in inputs
27     for (int i=0; i<n; i++)
28     {
29         for (int j=i+1; j<n; j++)
30         {
31             for (int k=j+1; k<n; k++)
32             {
33                 if (determinantValue(arr[i].x, arr[i].y, arr[j].x, arr[j].y, arr[k].x, arr[k].y))
34                     result++; // Increment the result when determinant of a triplet is not 0.
35             }
36         }
37     }
38     return result;
39 }
40 }
```

Задачи

- Брой триъгълници – Решение

```
41 int main()
42 {
43     //Point arr[] = {{0, 0}, {1, 1}, {2, 0}, {2, 2}};
44     //Point arr[4];
45     int n;
46     while(cin>>n) {
47         //cin>>n;
48         Point arr[n];
49         for(int i = 0; i < n; i++)
50         {
51             cin>>arr[i].x>>arr[i].y;
52         }
53         //cin>>arr[0].x>>arr[0].y;
54         //int n = sizeof(arr)/sizeof(arr[0]);
55         cout << countPoints(arr, n)<<endl;
56     }
57     return 0;
58 }
```

Задачи

- **Брой триъгълници – Решение**
 - Времева сложност: $O(n^3)$.
 - Оптимизация:
 - Можем да оптимизираме това решение, така че то да работи в $O(n^2)$, като използваме факта, че трите точки не могат да образуват триъгълник, ако са колинеарни.
 - Можем да използваме хеширане, за да съхраняваме наклони на всички двойки и да намираме всички триъгълници за $O(n^2)$ време.



Алгоритми и програмиране

Задача 1

- Професионален крадец планира да ограби няколко къщи на улица „Иван Вазов“ в София. Във всяка от къщите на тази улица има скрити определена сума пари. **Съседните къщи на улицата имат свързана охранителна система, която автоматично се свързва с полицията, ако две съседни къщи бъдат ограбени в една и съща нощ.** Това е единственото нещо, което спира крадеца да ограби всяка от къщите на улицата.
 - Даден е списък с неотрицателни цели числа, представляващи парите, които са скрити във всяка къща. Намерете максималната сума, която може да бъде открадната за една вечер, без да се сигнализира на полицията.

Задача 1

Input: [1, 2, 3, 1]

Output: 4

Explanation:

Option 1:

Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total amount he can rob = $1 + 3 = 4$.

Option 2:

Rob house 2 (money = 2) and then rob house 4 (money = 1).
Total amount he can rob = $2 + 1 = 3$

So, option 1 maximizes the money stolen.

Задача 1 - Алгоритъм

- Задачата се свежда до отговор на следния въпрос: Да се обере ли $i^{\text{та}}$ къща или не?
 - Вземането на решението дали да се обере $i^{\text{та}}$ къщата или не, зависи изцяло от това къде парите са най-много.
- Съседните къщи са свързани, следователно обирджията не може да ограби две съседни къщи.
 - Може да ограби $(i-1)^{\text{та}}$ къща или $i^{\text{та}}$ къща, но след ограбване на $(i-2)^{\text{та}}$ къщата.
- Целта е да се намери стратегията, която ще доведе до максимална сума.
 - Рекурсивна връзка:

$$f(i) = \max(f(i-2) + \text{пари от } i^{\text{та}} \text{ къща}, f(i-1))$$

Задача 1 - Стъпки

1. Bases case:

```
IF (n==0)
    Return 0;
IF (n==1)
    Return money[0];
```
2. Initialize DP matrix

```
int house[n]; // DP matrix
//initialize all elements with their respective money
for(int i=0;i<n;i++)
    house[i]=money[i];
```
3. IF there is only one house, then simply rob the money.

```
house[0]=money[0];
```
4. IF there is two house rob from the house having maximum amount

```
house[1]=(money[0]>money[1])?money[0]:money[1];
```
5. Formulate the recursive function:

```
for(int i=2;i<n;i++){
    house[i]=max(house[i]+house[i-2],house[i-1]);
}
//house[i-2]= f(i-2) // as updated by DP matrix
// house[i-2]= f(i-2) // as updated by DP matrix
// house[i]( on R.H.S) = money of ith house // as not still updated by DP matrix
// house[i] (on L.H.S) = f(i) // as it's going to be updated by DP matrix
```
6. Return house[n-1]

money - 1D
масив,
където се
съхраняват
 pari, скрити
в къщите.

Задача 1 - Стъпки

Iteration 0:

```
house[2]=max(house[2]+house[0],house[1]);
house[2]=6 // house[2]+house[0]=6 &house[1]=4
that means rob at house 0 & then at house 2
4 (0th) | 4 (1st) | 6(2nd) | 10 (3rd) | 4 (4th) | 2 (5th)
```

Iteration 1:

```
house[3]=max(house[3]+house[1],house[2]);
house[3]=14 // house[3]+house[1]=14&house[2]=6
that means rob at house 1 & then at house 3 (decision changed)
4 (0th) | 4 (1st) |       6(2nd) | 14 (3rd) |       4 (4th) | 2 (5th)
```

Iteration 2:

```
house[4]=max(house[4]+house[2],house[3]);
house[4]=14 // house[4]+house[2]=10&house[3]=14
that means rob at house 1 & then at house 3 (no change in last decision)
4 (0th) | 4 (1st) |       6(2nd) | 14 (3rd) |       10 (4th) | 2 (5th)
```

Iteration 3:

```
house[5]=max(house[5]+house[3],house[4]);
house[5]=16 // house[5]+house[3]=16&house[4]=10
that means rob at house 1 & then at house 3 & then at house 5(final)
4 (0th) | 4 (1st) |       6(2nd) | 14 (3rd) |       10 (4th) | 16 (5th)
```

Let's solve an example using the algo

No of houses: 6

Money stashed at houses:

4, 2, 2, 10, 4, 2

Initially DP matrix

4 (0th) | 2 (1st) | 2(2nd) | 10 (3rd) | 4 (4th) | 2 (5th)

After step -3 & step - 4:

4 (0th) | 4 (1st) | 2(2nd) | 10 (3rd) | 4 (4th) | 2(5th)

At step-5:

House index starts from 0

Задача 2

- Дадена е стойност N и безкрайно количество от монети със стойности $S = \{S_1, S_2, \dots, S_m\}$. Намерете броя на начините, по които може да получите N цента. Редът на монетите няма значение.

Input:

```
N = 4  
S = {1, 2, 3} //infinite number of 1 cent, 2 cent, 3 cent coins
```

Output:

```
4  
{1,1,1,1} //four 1 cent coins  
{1,1,2} //two 1 cent coins, one 2 cent coins  
{2,2} //two 2 cent coins  
{1,3} //one 1 cent and one 3 cent coins  
Thus total four ways (Order doesn't matter)
```

Задача 2 - Решение

1. Направете проверка дали сумата N може да бъде получени от монети с един и същи номинал.

 - 4 цента - четири монети от 1 цент.
 - 4 цента - две монети от 2 цента.
2. Вземете монети с различна стойност и съхранете подсумите, които могат да бъдат получени.

Две монети от 1 цент дават 2 цента. Така 2 цента е подсума.
3. Изберете още две монети от 1 цент, което ще даде 4.
4. В противен случай, изберете само една монета от 2 цента, което също ще дъде 4.

Задача 2 - Алгоритм

1. Create a DP table of size amount

```
Table[amount+1]={0};
```

2. Base case table[0]=1

3. For each coins[i] from the coins array

```
    For j= 1: amount //j be the sub-amount
```

```
        IF j>=coins[i]
```

```
            Table[j]=table[j]+table[j-coins[i]]
```

```
        END IF
```

```
    END For
```

```
END For
```

4. Return table[amount]

Table[amount] refers to all possible way to make the amount

```
We have pre added coin 0-cent as coin[0]=0  
For Coin index: 1 //coins[1]=1
```

```
Sub amount:1
```

```
Table status:
```

```
1 | 1 | 0 | 0 | 0
```

```
Sub amount:2
```

```
Table status:
```

```
1 | 1 | 1 | 0 | 0
```

```
Sub amount:3
```

```
1 | 1 | 1 | 1 | 0
```

```
Sub amount:4
```

```
1 | 1 | 1 | 1 | 1
```

Задача 3

- Дадено е множество от думи и низ, който не съдържа интервали и специални думи. Разделете дадения низ сmisлени думи, базирайки се на даденото множество от думи .

Case-I:

If the dictionary contain the words:

{"like", "i" , "ice" , "cream", "is"};

Input : "ilikeicecream"

Output: "i like ice cream"

Case-II:

If the dictionary contain the words:

{"like", "i" , "ice" , "cream", "is"}

Input: "ilikeeicecream"

Output: False

(There is no combination possibleout from the dictionary)

Задача 3 - Алгоритъм

- Решението на задачата се състои от две части
 1. Откриване на думите.
 2. Извличане на думите.
- Случай-I
 1. Създава се булев масив с дължина = дължината на входния низ и се инициализира с false.
 2. Създава се вектор и се инициализира с -1.
 3. Започва (от позиция 0) сравняване на входния низ, като всеки път дължината на сравняваната част се увеличава с 1.
 4. При намиране на смислена дума (базирайки се на речника) се извиква метода `add()` на вектора с определен индекс и на същата тази позиция се съхранява стойност `true` в булевия масив.

Задача 3 - Алгоритъм

- Решението на задачата се състои от две части
 1. Откриване на думите.
 2. Извличане на думите.

Случай-I

5. След обработката на текущата дума, започва проверката на следващата дума от достигнатата позиция.
6. Стъпки 2-5 се повтарят докато не се обходи целия низ.
7. Ако на позиция ($length-1$) има стойност *true* в булевия масив, то тогава низа се разделя.

Задача 3 - Алгоритъм

- Решението на задачата се състои от две части
 1. Откриване на думите.
 2. Извличане на думите.
- Случай-II
 1. Взема се подниз, започвайки от последния елемент на вектора до последния елемент от низа и се проверява в даденото множество от думи.
 2. Ако текущия подниз се намери в даденото множество то думи, то при следващо обхождане последният подниз ще бъде последният елемент във вектора.
 3. Ако текущия подниз не бъде намерен, то се преминава само към предпоследния елемент на вектора.
 4. Процеса се повтаря, докато се стигне до първия елемент на вектора.

Задача 4

- Дадено е неотрицателно цяло число n , пребройте всички числа с уникални цифри x , $0 \leq x < 10^n$.

Input & output:

$n=2$

Total numbers that can be formed is 91

For $n = 2$.

Total numbers = 0 to 10^2 (excluding)

except 11, 22, 33, 44, 55, 66, 77, 88, 99 which has repeated digits.

Задача 4 - Решение

- Ако броят на цифрите е 0, тогава уникалните числа, които могат да се образуват, са 0
- Ако броят на цифрите е 1, тогава уникалните числа, които могат да се образуват, са 10: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- Ако броят на цифрите е 2:
 - За първата цифра (отляво) има 9 възможности (с изключение на 0) [1 до 9]
 - За втората цифра има 9 възможности (от 0 до 10 с изключение на предходната цифра). По този начин числото е:
$$ij, \text{ където } i = [1, 9] \text{ и } j = [0, 9], i \neq j$$
 - Следователно, общият възможен брой е $9 * 9 = 81$

Задача 4 - Решение

- Ако броят на цифрите е 3:
 - За първата цифра (отляво) има 9 възможности (с изключение на 0) [1 до 9].
 - За втората цифра има 9 възможности (от 0 до 10 с изключение на предходната цифра)
 - За трета цифра има 8 възможности (от 0 до 10 с изключение на предходните две цифри). По този начин числото е:
$$ijk, \text{ където } i = [1, 9] \text{ и } j, k = [0, 9], i \neq j \& j \neq k, i$$
 - Следователно, общият възможен брой е $9 * 9 * 8 = 648$
-
- Ако броят на цифрите е i , тогава за $i^{\text{тата}}$ цифра ще има $(10 - i + 1)$ възможности.

Задача 4 - Решение

$f(n)$ = total number of unique number with exactly n digits

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 10 & \text{if } n = 1 \\ 9 * \prod_{i=2}^n (10 - i + 1) & \text{if } n \geq 2 \end{cases}$$

DP[i] = f(i)=total number of unique number with exactly i digits

- 1) Initialize DP[n+1] with 0
- 2) DP[0]=0, DP[1]=10, DP[2]=81
- 3) Result = count of total numbers(x) that can be formed with unique digits, where $0 \leq x < 10^n$
- 4) Base value of result=91(for n=2)
- 5) for i=3 to n
 DP[i]=DP[i-1]*(10-i+1);
 result=result+DP[i];
end for
- 6) Return result

Задачи 5

Даден е масив A с размер n , който съдържа цели числа и е с четна дължина. Елементите на масива представляват n монета със стойности x_1, x_2, \dots, x_n . Правилото на играта е:

1. Всеки от играчите получава алтернативни ходове.
2. На всеки ход играчът избира първата или последната монета от реда, премахва я и получава нейната стойност.
3. И двамата играчи играят по оптимален начин, т.е. и двамата играчи искат да максимизират спечелената сума.

Определите максималната възможна сума, която може да спечелите, ако започнете играта.

Задачи 6

- Дадена е златна мина с размери $p \times m$. Всяка клетка в тази мина съдържа цяло положително число, което е количеството злато в тонове. Първоначално миньорът е в първата колона, но може да бъде на всеки ред. Той може да се движи само надясно, надясно и нагоре или надясно и надолу от текущата клетка, т.е. миньорът може да стигне до клетката, която е диагонално нагоре и надясно, надясно или диагонално надолу и надясно.
- Намерете максималното количество злато, което миньорът може да събере.

CSCB034 Упражнения по алгоритми и програмиране

АЛГОРИТМИ ЗА СОРТИРАНЕ ОТ КЛАС $O(n^2)$

гл. ас. д-р Слав Ангелов, НБУ

Загрявка

Напишете функция `print_array(int arr[], int n)`, която с вход масив и съответната му дължина принтира масива в конзолата.

Решение

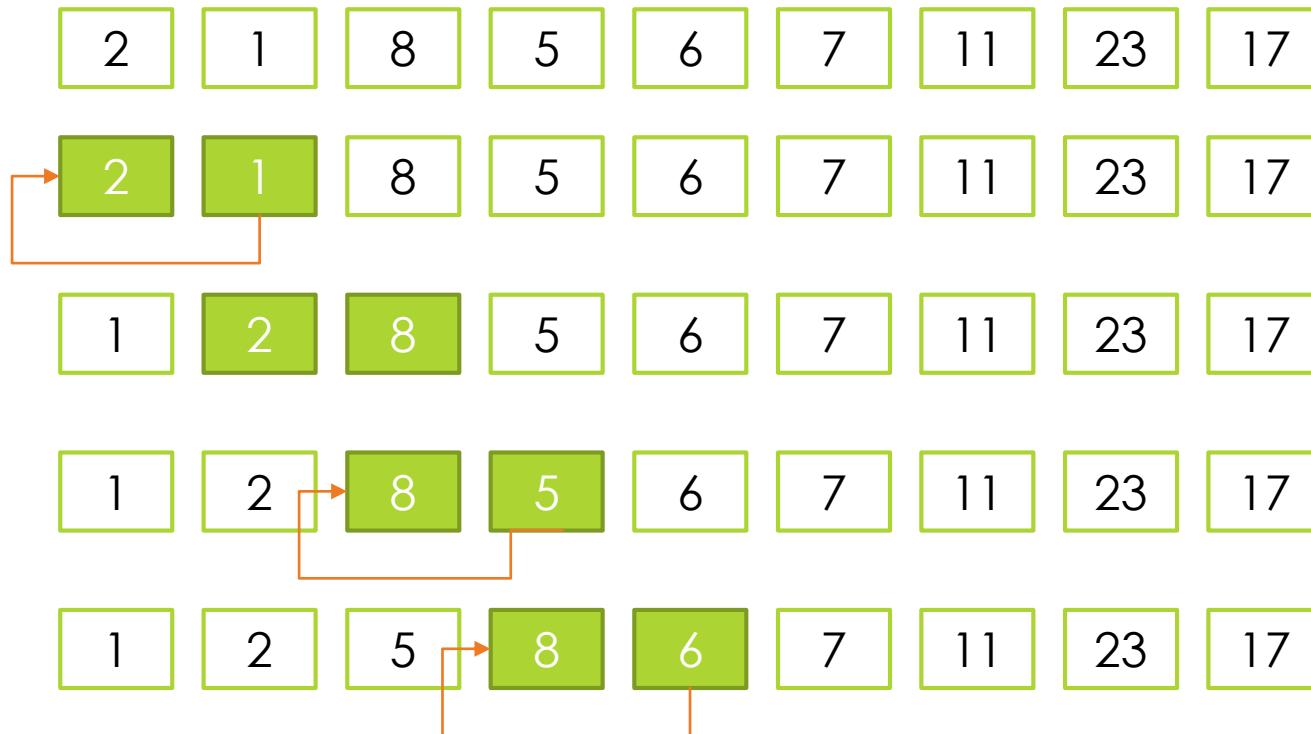
```
void print_array (int arg[], int length) {  
    for (int n=0; n < length; ++n)  
        cout << arg[n] << ' ';  
    cout << '\n';  
}
```

Сортиране чрез метода на „мехурчето“ (Bubble sort)

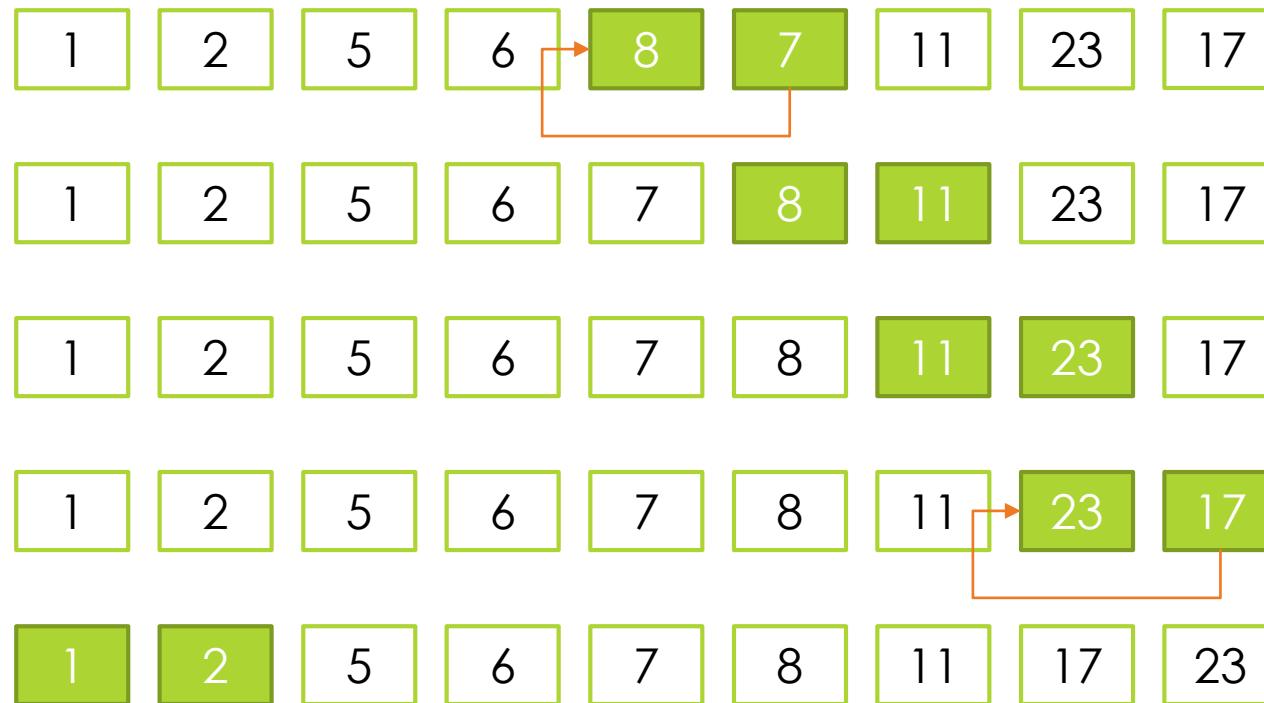
Вход:



Обхождане 1:



Продължение на примера (1/2)



Продължение на примера (2/2)

Повторно обхождаме целия масив без последния елемент (той вече е нареден).
Понеже не е реализирано нито едно разместване, то алгоритъмът спира.

Ако имаме поне едно реализирано разместване, то отново обхождаме целия масив без последните два елемента, те вече са наредени.

Ако отново имаме разместване, обхождаме четвърти път без последните три елемента и т.н..

Максималната изчислителна сложност на метода на мехурчето е $O(n^2)$.

Време за размисли

- ▶ Ако имаме масив с да кажем 9 елемента, кое ще е най-лошо примерно подреждане в смисъл, че такова подреждане ще забави най-много алгоритъмът на мехурчето.

Подсказка: При този алгоритъм за сортиране големите стойности се наричат „зайци“, понеже се подреждат много бързо, а малките стойности „костенурки“, защото те се подреждат най-бавно.

Време за размисли

- ▶ Ако имаме масив с да кажем 9 елемента, кое ще е най-лошо примерно подреждане в смисъл, че такова подреждане ще забави най-много алгоритъмът на мехурчето.

Подсказка: При този алгоритъм за сортиране големите стойности се наричат „зайци“, понеже се подреждат много бързо, а малките стойности „костенурки“, защото те се подреждат най-бавно.

Към подсказката: Запишете пример с девет числа като по-малките са в края на масива и мислено разиграйте метода на мехурчето, за да видите защо имаме зайци и костенурки.

Задача

Реализирайте алгоритъма на мехурчето в най-прост вид, т.е. без „стоп“ (с минаване през всички възможни обхождания). Нека функцията се казва `void bubble_sort()`. Входни данни да са масив от цели числа и дължината на съответния масив.

Можем да разбием задачата на две стъпки:

- 1) Да конструираме цикъл за обхождане на масива веднъж с метода на мехурчето;
- 2) Да допълним всички възможни обхождания с още един цикъл.

Решение

```
void bubble_sort(int a[], int n)
{ int i,j,t;
  for(i=n-2; i>=0; i--)
    for(j=0; j<=i; j++)
      if(a[j]>a[j+1])
      { t=a[j];
        a[j]=a[j+1];
        a[j+1]=t;
      }
}
```

**Можем ли да запишем това решение с
лека промяна във for циклите?**

Решение 2

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

Задача

Реализирайте алгоритъма на мехурчето със „стоп“. Функцията отново се казва `void bubble_sort(int arr[], int n)`.

Разяснение: Поставяме „стоп“, за да предотвратим многократно повтаряне на обхождания без това да е необходимо, т.е. ние вече сме сортирали масива.

Решение

```
void bubble_sort(int a[], int n)
{ int i,j,t; bool swapped;
  for(i=n-2; i>=0; i--){
    swapped=false;
    for(j=0; j<=i; j++)
      if(a[j]>a[j+1])
        { t=a[j]; a[j]=a[j+1]; a[j+1]=t;
          swapped=true;
        }
    if (swapped==false) break; }
```

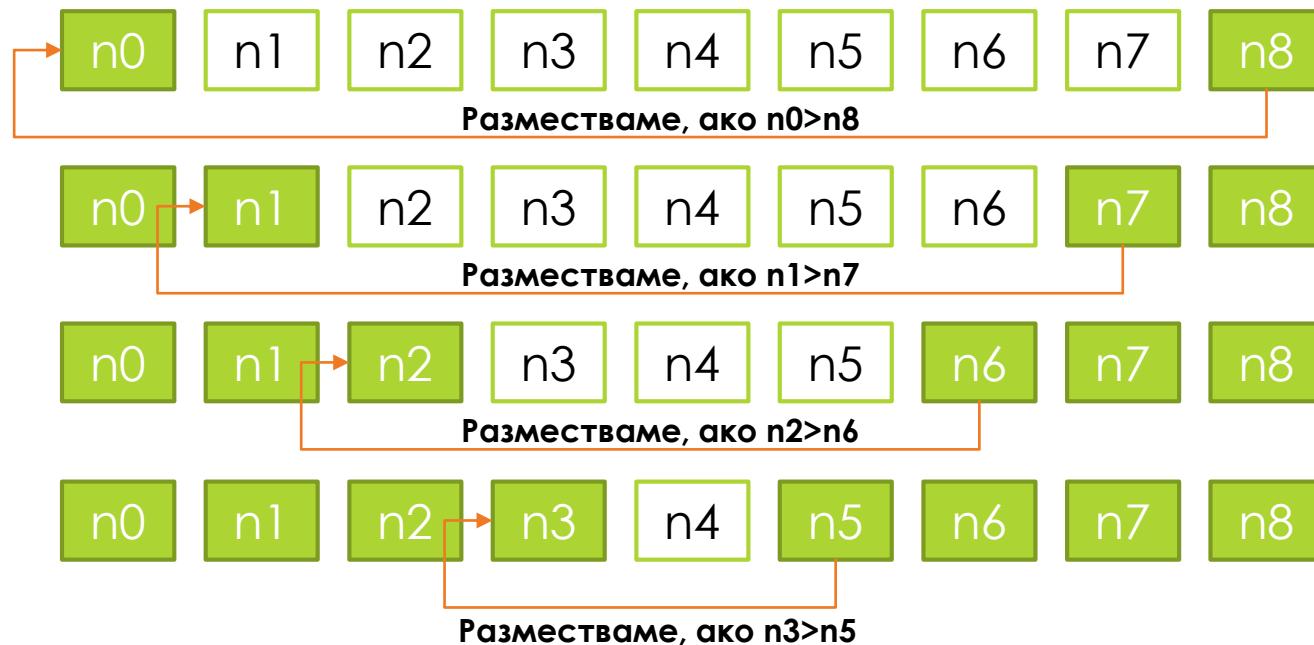
А какво щеше да се случи, ако по погрешка бяхте сложили проверката за swapped вътре във втория цикъл?

Задача

Напишете метода на „мехурчето“ така, че да сортира в низходящ ред.

Задача

Ще се подобри ли производителността на алгоритъма на мухурчето, ако преди неговото прилагане приложим следната процедура:



Алгоритъм „С вмъкване“ (1/2)

Вход:



Стъпка 1:



Стъпка 2:



Стъпка 3:



Стъпка 4:



Стъпка 5:



Алгоритъм „с вмъкване“ (2/2)

Стъпка 6:



Стъпка 7:



Стъпка 8:



Изход:



За разлика от метода на „мехурчето“ този алгоритъм сортира всяка стойност на ляво максимално, а не само по едно отместване.

Задача

Конструирайте примери с по 9 стойности, където метода на мехурчето е по-ефикасен и където метода „с вмъкване“ е по-ефикасен.

Задача

Реализирайте сортиране “с вмъкване” на C++. Нека функцията е:

void insertion_sort(int arr[], int n).

Решение

```
void insertion_sort(int arr[], int n){  
    int i, key, j;  
    for (i=1, i<n, i++){  
        key=arr[i];  
        j=i-1;  
        while ( j>=0 && arr[j]>key){  
            arr[j+1]=arr[j];    j --; }  
        arr[j+1]=key;  
    }  
}
```

**Има ли някъде в реализациата „стоп“,
който да предпазва от излишни итерации?**

Решение

```
void insertion_sort(int arr[], int n){  
    int key, j;  
    for (int i=1; i<n; i++){  
        key=arr[i];  
        j=i-1;  
        while ( j>=0 && arr[j]>key){  
            arr[j+1]=arr[j];    j--; }  
        arr[j+1]=key;  
    }  
}
```

Има ли някъде в реализациата „стоп“, който да предпазва от излишни итерации?

В `while` цикъла има условие, което предотвратява излишни „разходки“.

Сортиране чрез метода на „коктейла“

Той представлява подобрение на метода на мехурчето. Идеята е, че вместо при първото обхождане да се врне в началалото на масива той започва от края, като прескача последния елемент, защото той вече е сортиран. Щом отново стигне до началото, тръгва от началото към края, но вече от втория елемент, защото първият вече е сортиран. Щом стигнем до края тръгваме към началото, но стартираме преди последните два елемента, защото те вече са сортирани. Щом стигнем началото тръгваме след първите два елемента, защото те вече са сортирани и т.н.. Ако при някое обхождане не е било налично разместване, то масивът от числа вече е сортиран и прекратяваме процедурата.

Реализация (1/2)

```
void CockteilSort(int arr[], int n){  
    bool swapped = true;    int start=0;    int end= n-1;  
    while (swapped){  
        swapped=false;  
        for (int i=start; i<end; ++i){  
            if( arr[i]>arr[i+1]){  
                t=arr[i]; arr[i]=arr[i+1]; arr[i+1]=t;  
                swapped=true;}  
        }  
        if (!swapped) break;  
        --end;  
    }  
}
```

Реализация (2/2)

```
for (int i=end-1; i>=start; --i){  
    if(arr[i]>arr[i+1]){  
        t=arr[i]; arr[i]=arr[i+1]; arr[i+1]=t;  
        swapped=true;  
    }  
}  
++start;  
}  
}
```

Можем ли да подобрим стопа?

Коктейл сорт - модификация

- ▶ Удачно ли е да реализираме метода на коктейла чрез метода на вмъкването ?
- ▶ Ще има ли по-висока ефикасност, ако преди оригиналния метод на коктейла приложим симетричното сравняване и размяна при нужда на стойности (техниката, която коментирахме като възможна добавка с метода на мехурчето)?

CSCB034 Упражнения по алгоритми и програмиране

АЛГОРИТМИ ЗА СОРТИРАНЕ С КВАДРАТИЧНА СЛОЖНОСТ ЧАСТ 2

гл. ас. д-р Слав Ангелов, НБУ

„Драсканици“

Загрявка – selection sort

Напишете функция `void selection_sort(int a[], int n)`, която с вход масив и съответната му дължина сортира масива по метода на „пряката селекция“.

Описание:

- 1) Обхождаме масива и записваме най-малката стойност като първи елемент;
- 2) Обхождаме масива от втори елемент и записваме най-малкия елемент на втора позиция;
- 3) Обхождаме от третия елемент и записваме най-малкия елемент на трета позиция;
- 4) Продължаваме докато не останат елементи за обхождане.

Решение

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

Сравнения

Твърди се, че методът на „пряката селекция“ е по-ефикасен от метода на „мехурчето“. Допълнете кодът на тези два метода така, че да броим броя на размените (int as) и броя на сравняванията (int comp) по време на сортиране на масив. Тествайте с масива [9, 8, 7, 6, 5, 4, 3, 2, 1], който е един от най-лошите възможни случай за метода на мехурчето.

Отговор:

Сравнения

Твърди се, че методът на „пряката селекция“ е по-ефикасен от метода на „мехурчето“. Допълнете кодът на тези два метода така, че да броим броя на размените (int as) и броя на сравняванията (int comp) по време на сортиране на масив. Тествайте с масива [9, 8, 7, 6, 5, 4, 3, 2, 1], който е един от най-лошите възможни случай за метода на мехурчето.

Отговор:

Алгоритъм	Размени	Сравнения
Bubble sort	36	36
Selection sort	4	36

Реализация (1/3)

```
void selection_sort(int a[], int n){  
    int i, j;  
    for (j = 0; j < n-1; j++){  
        int iMin = j;  
        for (i = j+1; i < n; i++){      comp++;  
            if (a[i] < a[iMin]){  
                iMin = i;  
            }  
        }  
        if (iMin != j) {                  as++;  
            swap(a[j], a[iMin]);  
        }  
    } }
```

Реализация (2/3)

```
void bubble_sort(int a[], int n)
{ int i,j,t; bool swapped;
  for(i=n-2; i>=0; i--){
    swapped=false;
    for(j=0; j<=i; j++){ comp++;
      if(a[j]>a[j+1])
        { t=a[j]; a[j]=a[j+1]; a[j+1]=t; as++;
          swapped=true;
        }
    }
    if (swapped==false) break;
  }
}
```

Реализация (3/3)

```
#include <iostream>
using namespace std;
int as=0; int comp=0;
int main () {
    int arr[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};
    //bubble_sort(arr,9);
    selection_sort(arr,9);

    cout << "Comparisons = " << comp << ", Assignments = " << as << endl;
    return 0;
}
```

Забележка

Следва да се отбележи, че, Да, размените при метода на „пряката селекция“ са по-оптимизирани, но не и толкова, колкото показва брояча. Причината е, че запазвахме индексите на минималните стойности преди реално да направим разместване. Това може и да е добра оптимизация при сортиране на по-сложни данни, но при числа няма голяма разлика, т.е. трябва да се брои за разместване.

За да оценим по-прецизно ефикасността, да отидем на следващия слайд.

Комплексни сравнения - задача

Нека за **as** брои всички оператори =, ++, --, а за **comp** броим всички логически оператори >, <, && и т.н.. Нека да отбележим, че в един:

- ▶ `for(i=n-2; i>=0; i--)` на всяка итерация имаме **as** и **comp** в условието на цикъла. Да отбележим и че for ще тръгне с първоначална проверка на `i>=0` преди първа итерация.
- ▶ `while (j>=0 && arr[j]>key)` тук ще броим по три **comp** на всяка обиколка, като първият от тях се брои преди влизане в while-а (Зашо?).
- ▶ При тези условия сравнете по **as** и **comp** bubble_sort(), selection_sort() и insertion_sort() за [9, 8, 7, 6, 5, 4, 3, 2, 1].
- ▶ Функцията `swap()` ще я броим за три **as**, макар от C++11 нататък да е с една идея по-бърза.

Реализация (1/3)

```
void selection_sort2(int a[], int n){  
    int i, j;                                as+=2;  
    for (j = 0; j < n-1; j++){                as+=2; comp++;  
        int iMin = j;  
        for (i = j+1; i < n; i++){          comp+=2; as++;  
            if (a[i] < a[iMin]){           as++;  
                iMin = i;  
            }  
        }  
        if (iMin != j) {                    comp++;  
            swap(a[j], a[iMin]);          as+=3;  
        }  
    } }
```

Реализация (2/3)

```
void bubble_sort2(int a[], int n)
{ int j,t; bool swapped;
  for(int i=n-2; i>=0; i--){
    swapped=false;
    for(j=0; j<=i; j++){
      if(a[j]>a[j+1]){
        swap(a[j], a[j+1]);
        swapped=true;
      }
    }
    if (swapped==false) break;
  }
}
```

Реализация (3/3)

```
void insertion_sort2(int a[], int n){  
    int key, j;                                as=2;  
    for (int i=1; i<n; i++){                  as++; comp++;  
        key=a[i];  
        j=i-1;                                as+=2; comp++;  
        while ( j>=0 && a[j]>key){          comp+=2;  
            a[j+1]=a[j];   j --;              as+=2;  
        }  
        a[j+1]=key;                            as++;  
    }  
}
```

Резултат

Алгоритъм	Присвоявания	Сравнения
Bubble sort	199	88
Selection sort	86	88
Insertion sort	106	88

За конкретния пример явен „победител“ е сортирането чрез „пряка селекция“.

Следва да отбележим, че масивът [9, 8, 7, 6, 5, 4, 3, 2, 1] е най-лош случай за Bubble и Insertion sort, както и за много други сортиращи алгоритми, но не е най-лош случай за Selection sort(), защото подрежда масива още на 6-цата, а след това само потвърждава, че го е подредил.

Можете ли да конструирате пример, където selection sort() да се затрудни повече?

Най-тежък вариант за selection sort()

Нека разгледаме следния масив: **[9, 1, 8, 2, 7, 3, 6, 4, 5]**, методът на пряката селекция ще реализира максимален брой присвоявания, за да нареди 9-ката на съответната и позиция, т.е. $n-1$ размествания, $n=9$. Допълнително се извършват и максимален брой презаписвания на индекси. Обхожданията по дефиницията на метода са максимален брой.

Допълнително – Метод на коктейла

Вариант 2

Вече говорихме за „метода на коктейла“, който се явява модификация на „метода на мехурчето“. Съществува втори вариант на „метода на коктейла“, който се реализира чрез „метода на прямата селекция“, но на всяка стъпка освен най-малкия се търси и най-големия елемент (той се записва най-отдясно).

Тази реализация не намалява сравняванията и размените, но спестява обхождания.

Можете ли да реализирате този вариант на C++?

Подготовка

Напишете два вложени `for` цикъла, които да обхождат масив от край до край, после от втори елемент до предпоследния елемент, после от трети до предпредпоследния, после от четвърти до $n-4$ -ти и т.н..

Реализация 1

```
int j,  
int n=7; // дължина на масива  
for (int i = 0; i < n-1; i++){  
    for (j = i+1; j < n; j++){  
        cout << "i = "<< i << ", j = " << j << endl;  
    }  
    n--;  
}
```

**Можем ли да оптимизираме
още малко кода, всичко ли е
наред ?**

Реализация 2

```
int j,  
int n=7; // дължина на масива  
int q=n/2;  
for (int i = 0; i < q; i++){  
    for (j = i+1; j < n; j++){  
        cout << "i = "<< i << ", j = " << j << endl;  
    }  
    n--;  
}
```

Обърнете внимание, че n
Намалява (по-ефикасно ?)

**Можем ли да оптимизираме
още малко кода ?**

В условието на първия цикъл
имаме аритметични
операции, които не е
необходимо да се
изпълняват на всяка
итерация. Допълнително
фиксираме обхожданията
на $n/2$.

Реализация на Коктейл сорт Версия 2

```
void selection_sort_kockteil (int a[], int n){  
    int j, jMin, jMax, q=n/2;  
    for (int i = 0; i < q; i++){  
        jMin = i; jMax=i;  
        for (j = i+1; j < n; j++){  
            if (a[j] < a[jMin]) jMin = j;  
            if (a[j] > a[jMax]) jMax = j; }  
        if (jMin != i)  
            swap(a[i], a[jMin]);  
        if (jMax != n-1)  
            swap(a[n-1], a[jMax]);  
        n--;    } }
```

За сравнение с другите три алгоритъма за масива **[9, 8, 7, 6, 5, 4, 3, 2, 1]**, тази имплементация прави **84** присвоявания и **72** сравнения, което е рекорд до тук.

Допълнително - Трета версия

Може да дефинираме разновидност на алгоритъма на „коктейла“, където при първото обхождане, което е аналогично на метода на „мехурчето“ се намира и най-малкия елемент, който след завършване на това обхождане се записва най-отпред на масива (първоначалният първи елемент отива на мястото на най-малкия). На второто обхождане тръгваме от втория елемент достигаме до предпоследния отново по метода на мехурчето, но помним индекса на втория най-малък елемент и го записваме на втора позия в масива ни и продължаваме с трето обхождане от третия елемент и т.н.

Процедурата се повтаря докато няма повече размествания или не се изчерпат обхожданията.

Можете ли да реализирате подобен алгоритъм?

По-ефикасен ли е от другите варианти?

Има ли и други възможни разновидности на метода на коктейла?

Допълнителни задачи

От алгоритмите със сложност $O(n^2)$ не сме говорили за „метода на гнома“, наподобява как хипотетичен гном реди саксийте си с цветя. Повече информация в: https://en.wikipedia.org/wiki/Gnome_sort

- 1) Реализирайте този алгоритъм на C++.**
- 2) Сравнете идеята зад метода с трите алгоритъма от преди малко.
Прилики? Разлики? Евентуални предимства в конкретни случаи?**
- 3) Сравнете този метод емпирично с останалите за [9, 8, 7, 6, 5, 4, 3, 2, 1].**
- 4) Добавете модификацията на алгоритъма описана в линка и повторете горните стъпки.**



Тестване на алгоритми с квадратична сложност

Метод на межурчето

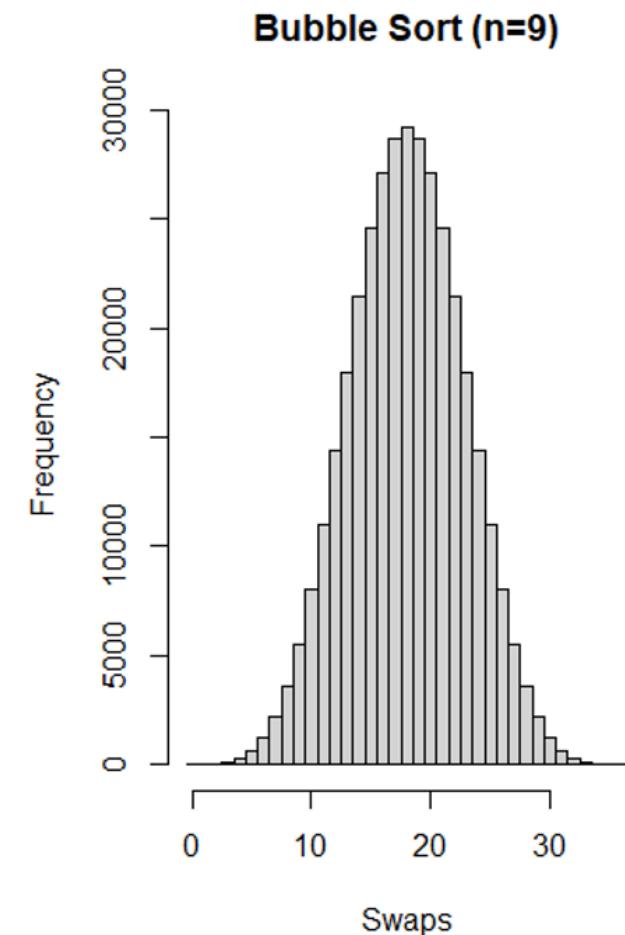
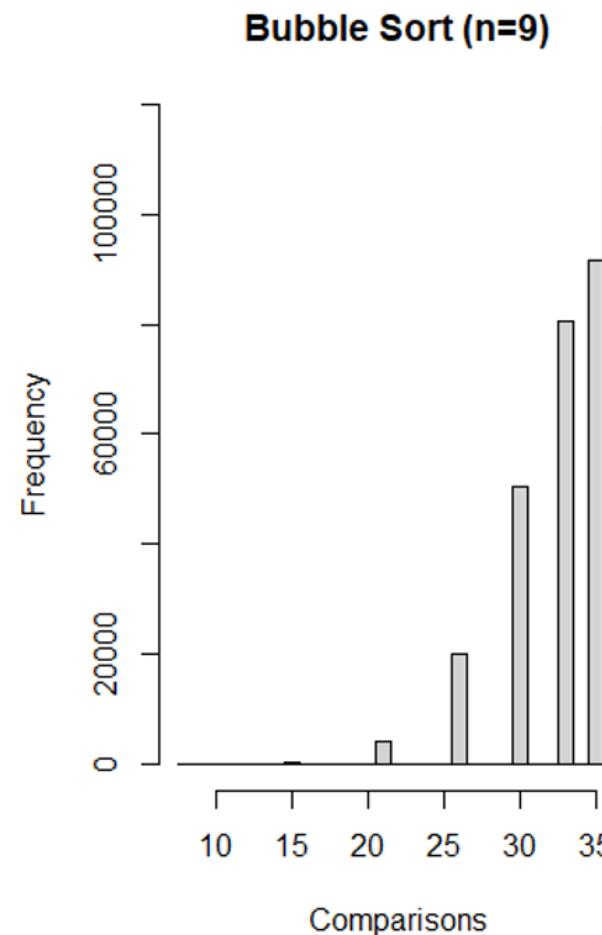
Алгоритъмът е тестван върху всички възможни случаи на разместявания на до 9 елемента.

Брой n=	Случаи	Сравнения				
		<i>Min</i>	<i>Q1</i>	<i>Median</i>	<i>Q3</i>	<i>Max</i>
5	120	4	9	10	10	10
6	720	5	14	14	15	15
7	5040	6	18	20	21	21
8	40320	7	25	27	28	28
9	362880	8	33	35	36	36

Метод на межурчето

Алгоритъмът е тестван върху всички възможни случаи на разместявания на 9 елемента.

Колко такива случаи са разгледани?



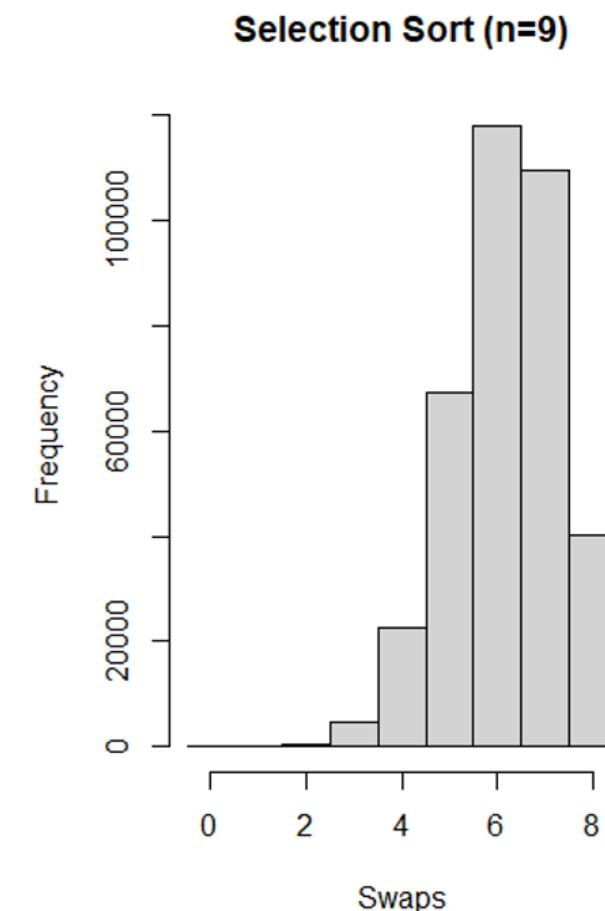
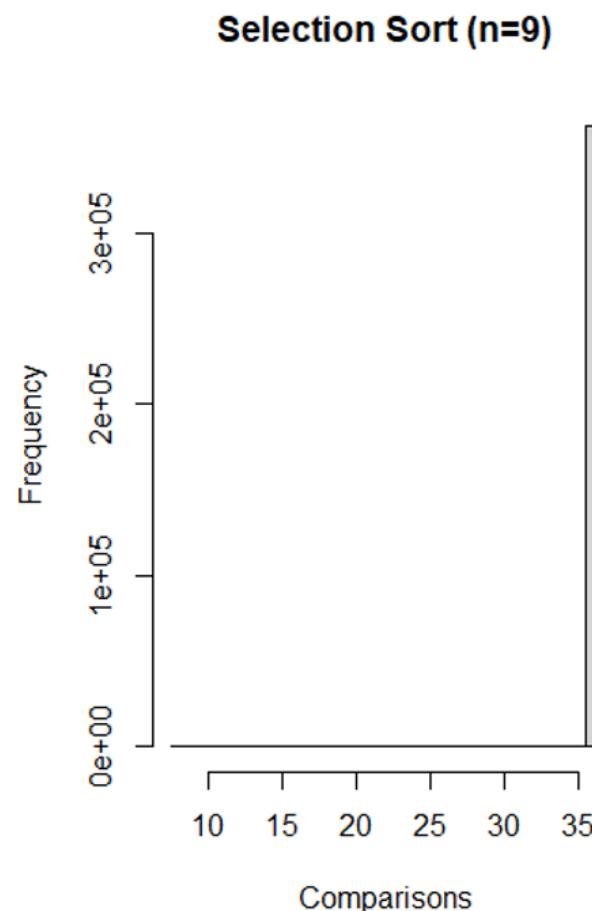
Метод на пряката селекция

Алгоритъмът е тестван върху всички възможни случаи на разместявания на до 9 елемента.

Брой n=	Случаи	Сравнения				
		<i>Min</i>	<i>Q1</i>	<i>Median</i>	<i>Q3</i>	<i>Max</i>
5	120	10	10	10	10	10
6	720	15	15	15	15	15
7	5040	21	21	21	21	21
8	40320	28	28	28	28	28
9	362880	36	36	36	36	36

Метод на пряката селекция

Алгоритъмът е тестван върху
всички възможни случаи на
размествания на 9 елемента.



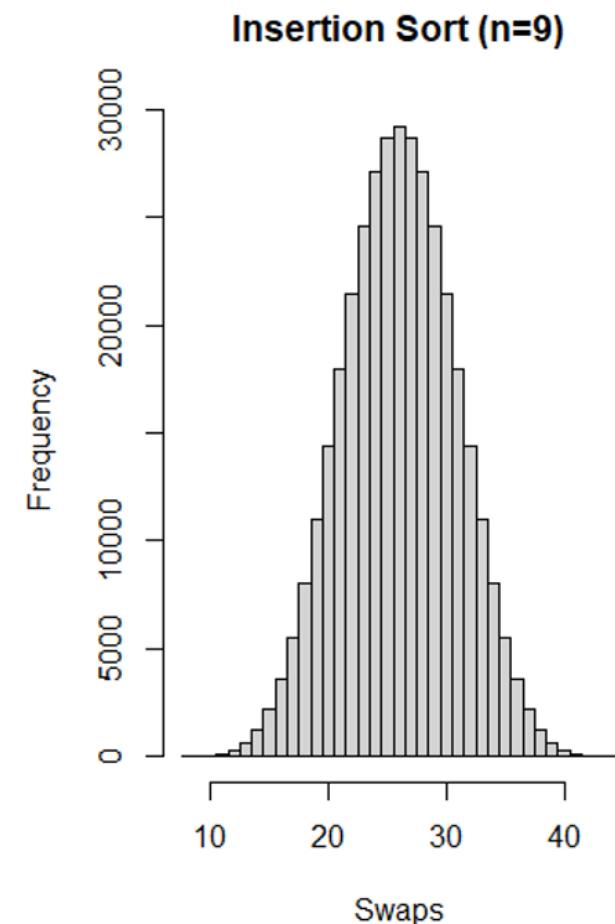
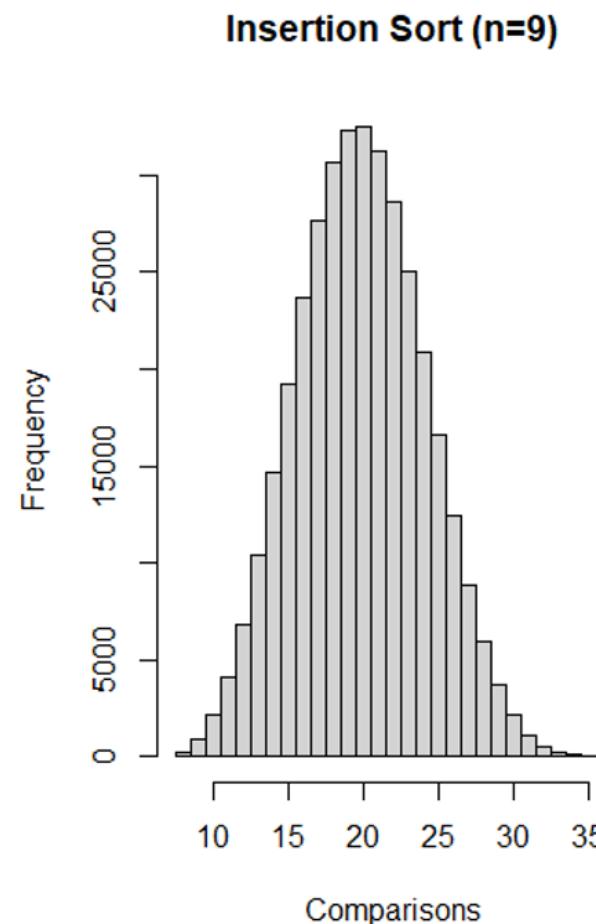
Метод на вмъкването

Алгоритъмът е тестван върху всички възможни случаи на разместявания на до 9 елемента.

Брой n=	Случаи	Сравнения				
		<i>Min</i>	<i>Q1</i>	<i>Median</i>	<i>Q3</i>	<i>Max</i>
5	120	4	5	6	7	10
6	720	5	7	9	10	15
7	5040	6	10	12	14	21
8	40320	7	13	16	18	28
9	362880	8	17	20	23	36

Метод на вмъкването

Алгоритъмът е тестван върху
всички възможни случаи на
размествания на 9 елемента.



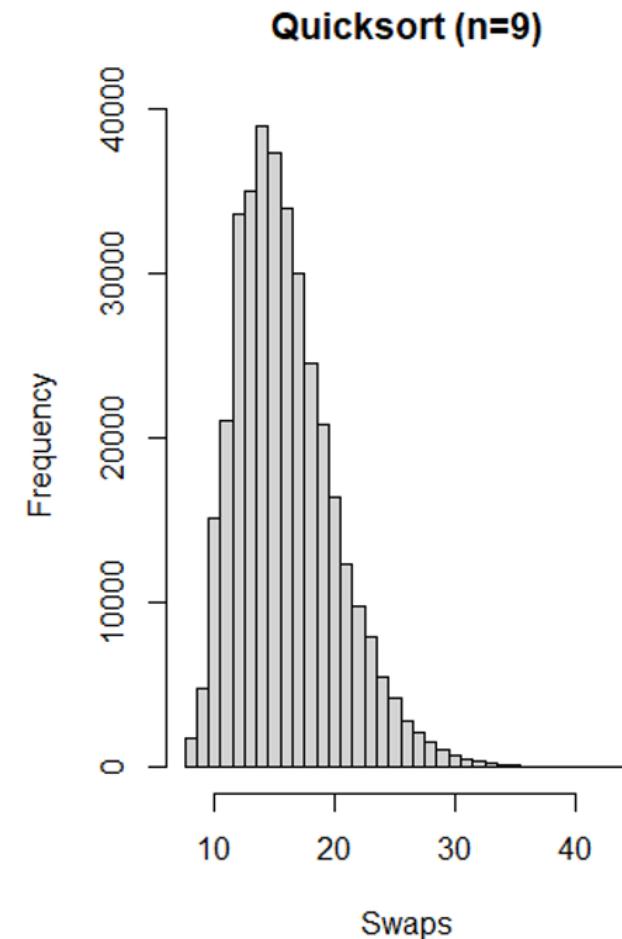
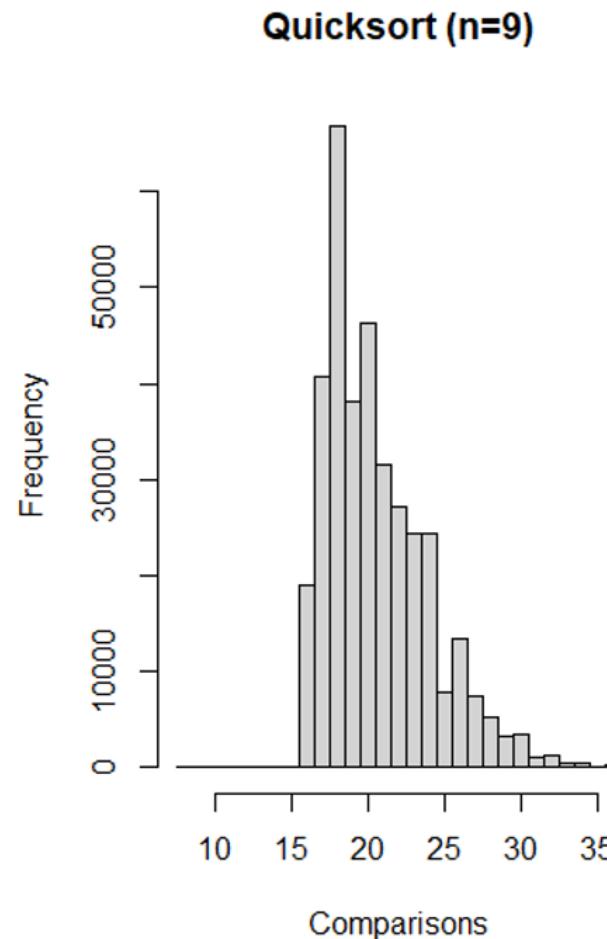
Класически Quicksort

Алгоритъмът е тестван върху всички възможни случаи на разместявания на до 9 елемента.

Брой n=	Случаи	Сравнения				
		<i>Min</i>	<i>Q1</i>	<i>Median</i>	<i>Q3</i>	<i>Max</i>
5	120	6	6	7	8	10
6	720	8	9	9.5	11	15
7	5040	10	12	13	15	21
8	40320	13	15	16	18.25	28
9	362880	16	18	20	23	36

Класически Quicksort

Алгоритъмът е тестван върху
всички възможни случаи на
размествания на 9 елемента.



Време за изпълнение

Елементи n=	Общо случаи	Сортиращ алгоритъм				
		Bubble sort	Selection sort	Insertion sort	Quicksort	Мой алгоритъм
5	120	0	0	0	0	0
6	720	0	0	0	0	0
7	5040	0	0	0	0	0
8	40320	15600	15600	15600	15600	15600
9	362880	125000	117000	82000	140600	97000

CSCB034 Упражнения по алгоритми и програмиране

АЛГОРИТМИ ЗА СОРТИРАНЕ С $O(N \cdot \log(N))$ СЛОЖНОСТ

гл. ас. д-р Слав Ангелов, НБУ

Любопитно: branchless programming

Целта е да може да реализираме нашият код без if, else, switch... По този начин избягваме създаването на разклонения по кода, което може да ускори изпълнението.

```
int Smaller(int a, int b) {  
    if(a < b)  
        return a;  
    else  
        return b;  
}
```

branchless


```
int Smaller_Branchless(int a, int b) {  
    return a * (a < b) + b * (b <= a);  
}
```

Компиляторите имат вградени функции за премахване на разклонения. Това важи с пълна сила за конкретния пример.

Пример 2

```
int Smaller3(int a, int b, int c){  
    if(a<b)  
        return a;  
    else if (b<c)  
        return b;  
    else  
        return c;  
}
```

branchless


```
int Smaller_Branchless3(int a, int b, int c){  
    return a*(a<b)*(a<c)+b*(b<a)*(b<c)+c*(c<=a)*(c<=b);  
}
```

Branchless programming - заключителни думи

Представените примери бяха тествани при 10, 50, 100, 200, 400 итерации многократно. Не беше забелязана статистически значима промяна в скоростта на изпълнение, което води до извода, че компилаторите сравнително добре успяват да преобразуват кода в branchless версия за if оператора или поне това е случая за прости сравнения и аритметични операции.

Загрявка

Когато говорим за ефикасност на алгоритми, то ние трябва да сме наясно колко най-малко операции са необходими, за да наредим един масив от n стойности.

Загрявка

Когато говорим за ефикасност на алгоритми, то ние трябва да сме наясно колко най-малко операции са необходими, за да наредим един масив от n стойности.

Можем да забележим, че масивът [3,2,1] може да се нареди само с едно размествания, но [2,3,1] се нарежда с две размествания независимо, че знаем къде да поставим цифрите.

Загрявка

Когато говорим за ефикасност на алгоритми, то ние трябва да сме наясно колко най-малко операции са необходими, за да наредим един масив от n стойности.

Можем да забележим, че масивът [3,2,1] може да се нареди само с едно размествания, но [2,3,1] се нарежда с две размествания независимо, че знаем къде да поставим цифрите.

Можете ли да конструирате такова разместване на 6 цифри, че дори да знаете къде трябва да ги поставите, за да сортирате наредената шесторка, да са ви необходими максимален брой размествания?

Забелязваме ли какви конфигурации затрудняват нареждането?

Решение (1/2)

Нека започнем от следното:

- ▶ при две числа, то няма как да реализираме повече от едно разместване.
- ▶ Ако числата са три, нека да допуснем, че масивът е подреден. Нека например имаме [1,2,3], ако разместим само 1 и 3, имаме [3,2,1]. Тогава с лекота може да върнем разместването и да подредим масива. За да усложним подреждането, ще разместим и 1 с 2, имаме [3,1,2]. Всяко едно друго допълнително разместване всъщност „неутрализира“ усложняването на наредбата и играе ролята на стъпка към сортиране на масива.
- ▶ За масив с 6 цифри [1,2,3,4,5,6] може да започнем като го представим като два слепени масива от по 3 цифри. И нека ги разместим по най-сложния за тях възможен начин, налични са по две такива най-сложни конфигурации от по три елемента. Частта с по-малките цифри да поставим отзад.

Решение (2/2)

Имаме:

[6,4,5,3,1,2] (забележете, че по-маките елементи са в края, това ще създаде допълнителни усложнения при сортиране)

Когато го редим със знанието къде да слагаме, подхождаме подобно на метода на „пряката селекция“.

1) [1,4,5,3,*6*,2]; 2) [1,*2*,5,3,6,*4*]; 3) [1,2,*3,5*,6,4]; 4) [1,2,3,*4,6,5*]; 5) [1,2,3,4,*5,6*]

От примера се забелязва, че по-сложна конфигурация не е възможна за 6 елемента.

Извод за n елемента: Забелязва се и че в най-лошия случай ще са ни необходими $n-1$ стъпки, за да наредим масив с дължина n при условие, че знаем къде да поставяме елементите така, че той да бъде сортиран.

Усложнения при сортирането настъпват, когато докато редим елементите в началото разбъркваме реда им в края на масива.

Стабилност на алгоритъм

Ще казваме, че даден сортиращ алгоритъм е стабилен, ако той запазва реда на еднаквите елементи.

Стабилност на алгоритъм

Ще казваме, че даден сортиращ алгоритъм е стабилен, ако той запазва реда на еднаквите елементи.

Пример:

Нека имаме следните стойности $[4, 2, 2, 1, 0, 1]$, ако след прилагане на избран сортиращ алгоритъм получим сортиране от рода: $[0, 1, 1, 2, 2, 4]$, т.е. има размяна на позициите на еднакви елементи, то алгоритъма е нестабилен.

Задача

Да разгледаме Вариант 3 на cocktail sort, т.е. вървим напред с метода на мехурчето, но едновременно с това пазим индекса на най-малкия елемент, който поставяме в началото.

Дайте контрапример, с който да докажете, че този алгоритъм не е стабилен.

Решение

Да разгледаме например [3,8,2,3,1,5].

На първо обхождане най-големият елемент вече ще е в края, а 1-цата ще отиде в началото:

[1,2,3,3,5,8], местата на двете тройки са разменени поради разместването на 3 и 1.

Някои размисли относно стабилността

- 1) Стабилността няма значение, ако ще сортираме числа;
- 2) Няма значение и ако ще сортираме абсолютно различни обекти;
- 3) Ако използваме индекси да маркираме реда на еднаквите елементи, то няма никакъв проблем да използваме бърз алгоритъм , който не е стабилен, а след това за линейно време да пренаредим в желания стабилен ред повтарящите се елементи (дайте прост пример и проверете твърдението).



Изчислителна сложност
 $O(n \log(n))$

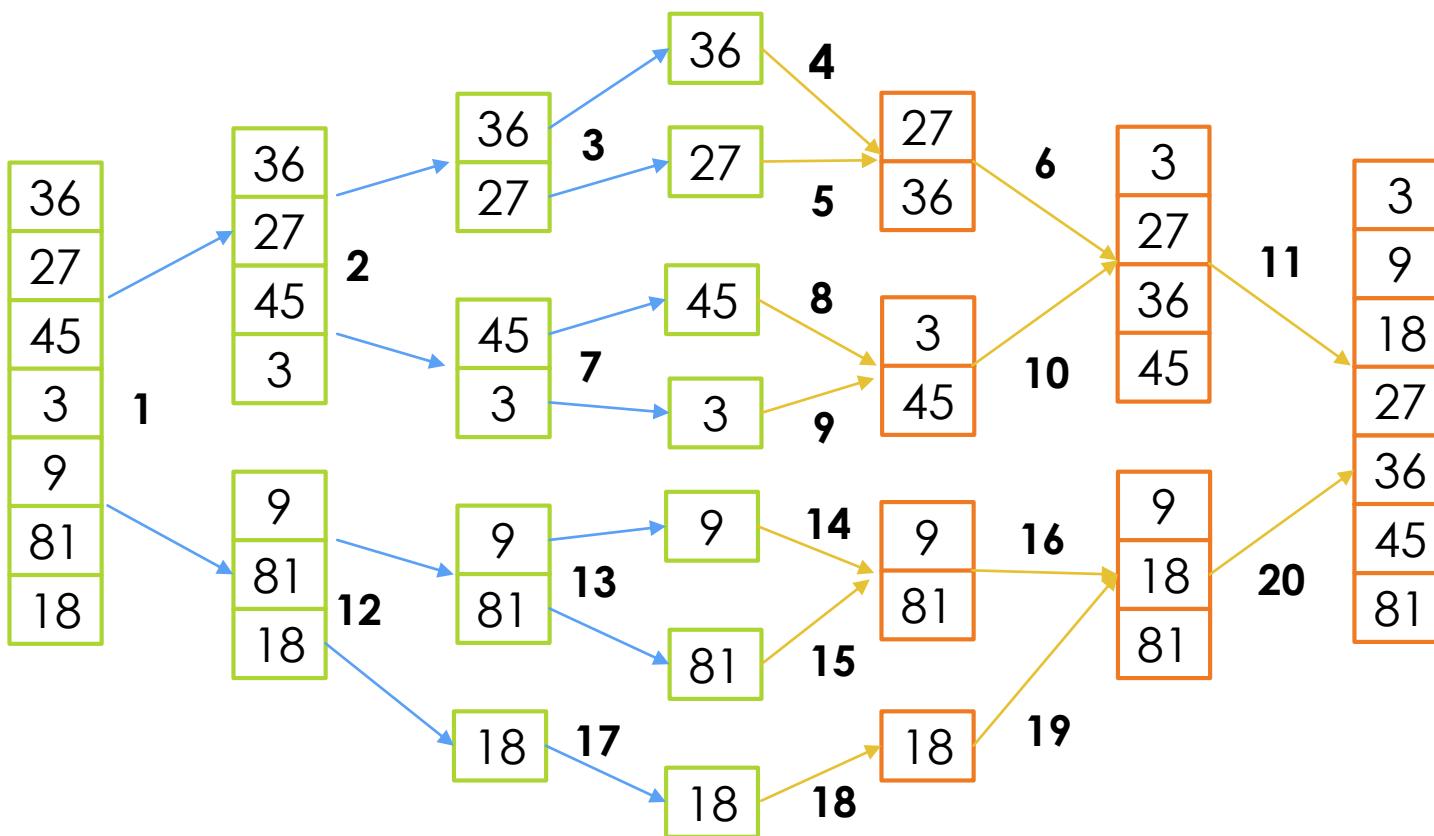
Merge sort

Това е сравняващ алгоритъм от типа „разделяй и владей“. Един от най-използваните алгоритми за сортиране. Налице са няколко версии и модификации. Редица модификации използват помощта на простички алгоритми като „метода на вмъкването“ в началните си стъпки, например Тимсорт (Tim sort).

Идеята е, че масивът започва да се разполовява рекурсивно, докато не се стигне до единични елементи. В последствие тези единични елементи се сглобяват два по два, докато не се сглоби един пълен сортиран масив. По време на сглабянето всяка част се сортира чрез смесване (merging).

Merge sort е стабилен алгоритъм за сортиране с изчислителна сложност $O(n \log(n))$ и изисква памет $O(n)$.

Пример



Последователността на операциите е номерирана.

21

Смесване на два сортирани масива

Вече споменахме, че в основата на Merge sort стои обединяването на два вече сортирани масива в един сортиран такъв. Съществуват няколко варианта, но основният „смесва“ по следната процедура:

- 1) Имаме два сравнително еднакви по големина сортирани масива А и Б, Създаваме празен масив С, който да е голям колкото А и Б сумарно;
- 2) Ако първият елемент от А е по-голям от първия елемент от Б, то го записваме на първа позиция в С (повече първия елемент на А не ни интересува) в противен случай записваме първия елемент от Б в С и той вече не ни интересува.
- 3) Продължаваме процедурата докато не останат елементи в един от двата масива А или Б и след това допълваме С с оставащите елементи, които по дефиниция са си подредени.

Демонстрация на „смесването“ (1/2)

$$A = \boxed{1} \boxed{3} \boxed{9} \boxed{10}$$

$$B = \boxed{5} \boxed{6} \boxed{6}$$

$$C = \boxed{} \boxed{} \boxed{} \boxed{} \boxed{} \boxed{}$$

Фаза 1:

$$A = \boxed{1} \boxed{3} \boxed{9} \boxed{10}$$

$$B = \boxed{5} \boxed{6} \boxed{6}$$

$$C = \boxed{1} \boxed{} \boxed{} \boxed{} \boxed{} \boxed{}$$

Фаза 2:

$$A = \boxed{1} \boxed{3} \boxed{9} \boxed{10}$$

$$B = \boxed{5} \boxed{6} \boxed{6}$$

$$C = \boxed{1} \boxed{3} \boxed{} \boxed{} \boxed{} \boxed{}$$

Фаза 3:

$$A = \boxed{1} \boxed{3} \boxed{9} \boxed{10}$$

$$B = \boxed{5} \boxed{6} \boxed{6}$$

$$C = \boxed{1} \boxed{3} \boxed{5} \boxed{} \boxed{} \boxed{}$$

Фаза 4:

$$A = \boxed{1} \boxed{3} \boxed{9} \boxed{10}$$

$$B = \boxed{5} \boxed{6} \boxed{6}$$

$$C = \boxed{1} \boxed{3} \boxed{5} \boxed{6} \boxed{} \boxed{}$$

Демонстрация на „смесването“ (2/2)

Фаза 4:	A =		B =		C =	
Фаза 5:	A =		B =		C =	
Фаза 6:	A =		B =		C =	

Задача – сложност на смесването

Можем ли да преbroим колко сравнения ще са нужни, за да извършим смесване между два подредени масива от по съответно n и k елемента ?

Решение

Обхождаме едновременно два масива и сравняваме елементите им. В най-лошият случай ще имаме, че елементите от масива с n елемента ще се изчерпят напълно, когато се изчерпят елементите от масива с k елемента, т.е. $n + k - 1$ сравнения.

Обобщение: $O(n+k)$, т.е. всичко се извършва за линейно време.

Задача – реализация на „смесване“

Нека имаме наредени масиви `int` A с дължина n, масив `int` B с дължина m и един деклариран на глобално ниво масив `int` C[n+m].

Напишете функция `void Merge(int A[], int n, int B[], int m)`, която да смеси тези масиви и да ги запази в C.

Реализация

```
void Merge (int a[], int n, int b[], int m){  
    int i=0; int j=0; int k=0;  
    while (i!=n && j!=m){  
        if(a[i]<b[j]) {  
            C[k]=a[i]; i++; k++; }  
        else {  
            C[k]=b[j]; j++; k++; }  
    }  
    while (i<n){ C[k]=a[i]; k++; i++; }  
    while (j<n){ C[k]=b[j]; k++; j++; }  
}
```

**Има ли нещо
нередно в тази
реализация?**

Реализация

```
void Merge (int a[], int n, int b[], int m){  
    int i=0; int j=0; int k=0;  
    while (i!=n && j!=m){  
        if(a[i]<b[j]) {  
            C[k]=a[i]; i++; k++; }  
        else {  
            C[k]=b[j]; j++; k++; }  
    }  
    while (i<n){ C[k]=a[i]; k++; i++; }  
    while (j<m){ C[k]=b[j]; k++; j++; }  
}
```

**Има ли нещо
нередно в тази
реализация?**

**Разбира се, че
има.**

Реализация 2

```
void Merge2 (int a[], int n, int b[], int m){  
  
    int i=0; int j=0; int limit=n+m;  
  
    for (int k=0; k < limit; k++){  
        if (i==n){ C[k]=b[j]; j++; continue;}  
        if(j==m){ C[k]=a[i]; i++; continue;}  
        if (a[i]>b[j]){ C[k]=b[j]; j++;}  
        else { C[k]=a[i]; i++;}  
    }  
}
```

**Има ли разлика в
ефикасността на
реализациите?**

Размисли – реализация на „смесване“

Колко ще е най-бързото сортиране на масив, който вече има две подредени почти равни части, ако не искаме да използваме допълнителна памет? Идеи за реализация?

Реализация на Merge sort (1/3)

```
void MERGE(int arr[], int l, int m, int r)
{
    int i, j, k; int n1 = m - l + 1; int n2 = r - m;
    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];
```

Реализация на Merge sort (2/3)

```
i = 0; j = 0; k = 1;  
while (i < n1 && j < n2) {  
    if (L[i] <= R[j]) {  
        arr[k] = L[i];  
        i++; }  
    else {  
        arr[k] = R[j];  
        j++; }  
    k++;}  
}
```

Реализация на Merge sort (3/3)

```
while (i < n1) {  
    arr[k] = L[i]; i++; k++;  
}  
  
while (j < n2) {  
    arr[k] = R[j];  
    j++;  
    k++;  
}  
}
```

Реализация на Merge sort – рекурсивното извикване

```
void mergeSort(int arr[], int l, int r)
{
    if (l < r) {

        int m = l+(r-1)/2; //същото като (l+r)/2, но избягва проблемно големи l и h.

        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        MERGE(arr, l, m, r);
    }
}
```

Източник: <https://www.geeksforgeeks.org/merge-sort/>

Natural Merge Sort

Това е bottom-up версия на Merge sort, която отчита естествените подреждания в даден ред от числа. Поради това, че той проверява за естествени подреждания между елементите в най-добрия случай той може да приключи със сложност $O(n)$ (дайте пример). Natural Merge Sort е ключов елемент в един от най-добрите алгоритми за сортиране – Timsort.

Natural Merge Sort - пример

Вход:



Начало:



Отчитаме
естествените
наредби

Фаза 1:



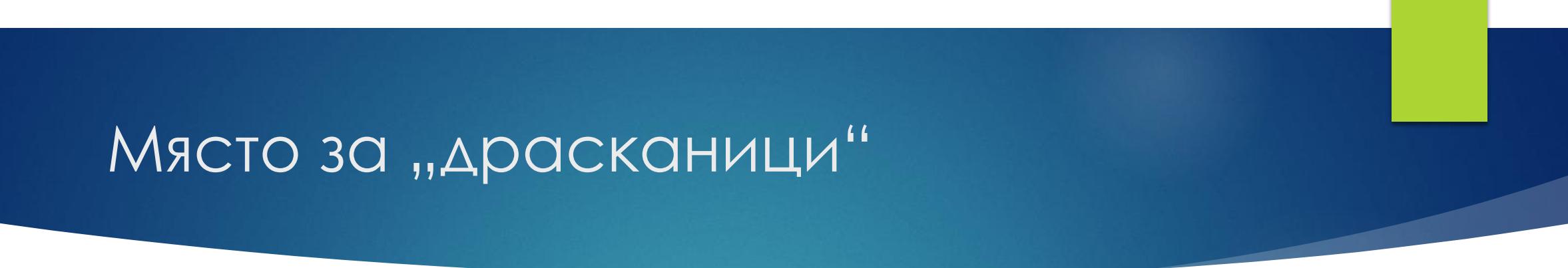
Фаза 2:



CSCB034 Упражнения по алгоритми и програмиране

АЛГОРИТМИ ЗА СОРТИРАНЕ С ПО-МАЛКА ОТ КВАДРАТИЧНА СЛОЖНОСТ

гл. ас. д-р Слав Ангелов, НБУ



Място за „драсканици“

Любопитно – историята на Bitonic sort

Кенет Батчър открива алгоритъма през 60-те, тогава работи за Goodyear Aerospace. Той забелязва забавления на разпарелените версии на Merge sort от един етап нататък и измисля Bitonic sort, за да се справи с проблема. Полученият алгоритъм е със сложност $O(n \log^2 n)$ при едноядрен процесор и $O(\log^2 n)$ при наличие на n ядра.

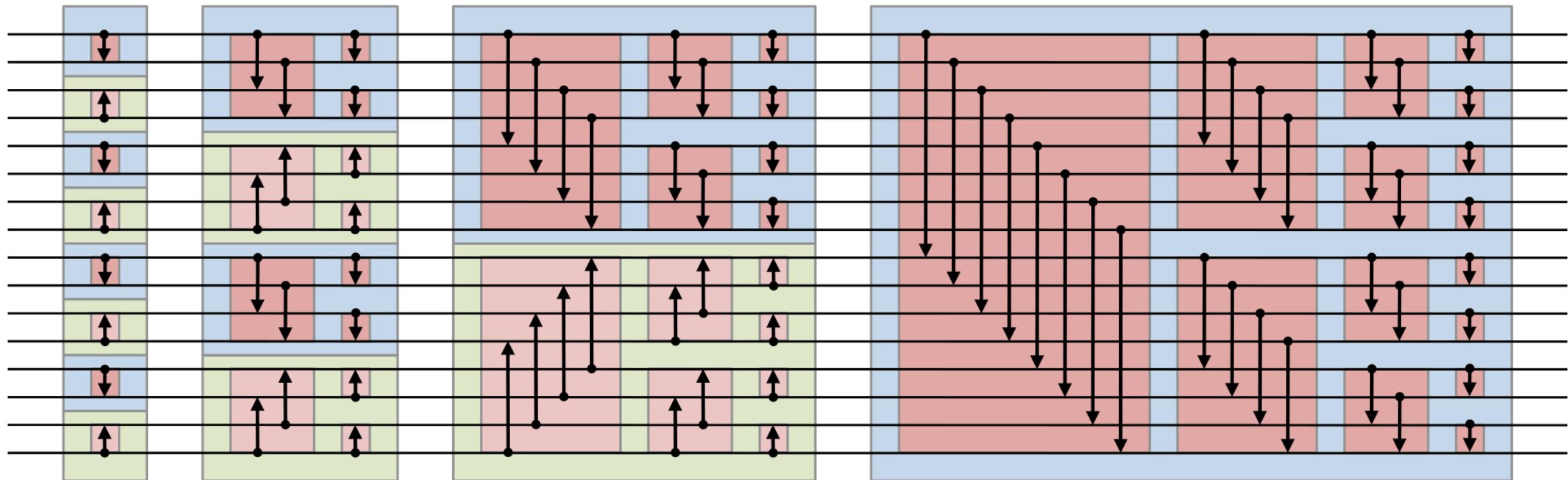
Bitonic sort се базира на така наречените битоник редици от числа (bitonic sequence), които се дефинират: $a_1 \leq a_2 \leq \dots \leq a_k > a_{k+1} \geq \dots \geq a_n$, $k \in [1, n]$. На всяка итерация редим битоник подредици, докато не наредим масива. Сортирането на подобни редици е добре дефинирано само при редици с 2^n елемента.

Bitonic sort НЕ влиза в изпитното съдържание на курса.

Повече информация може да намерите в Уикипедия или на:

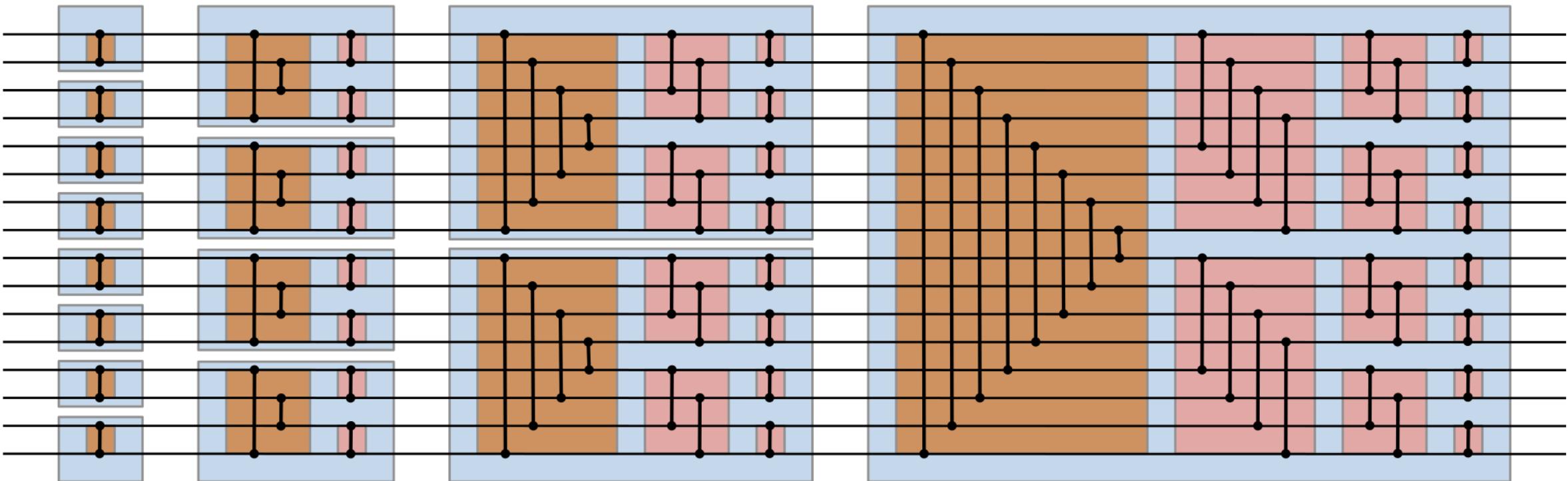
<https://www.geeksforgeeks.org/bitonic-sort/>

Bitonic sort – пример с 16 элемента



Източник: <https://commons.wikimedia.org/w/index.php?curid=21961929>

Bitonic sort – алтернативна схема



Източник: By Bitonic at English Wikipedia - Own work, CC0,
<https://commons.wikimedia.org/w/index.php?curid=21961917>



Изчислителна сложност $O(n)$

Загрявка – подреждане на индекси

Дайте идея за алгоритъм, който да сортира разбърканите числа от 1 до n за линейно време и $O(n)$ заделяне на памет.

Идея за решение

- 1) Създаваме допълнителен масив int temp[] с големината на a[];
- 2) Ако числата за подредба са съхранени в масив int a[], то докато обхождаме a[] с индекс int i попълваме temp[]:

$\text{temp}[a[i]-1]=a[i]$.

- 3) Презаписваме temp[] в a[].

Задача 2 – подреждане на индекси

Дайте идея за алгоритъм, който да сортира разбърканите числа от 1 до n за линейно време и **без заделяне на допълнителна памет**, т.е. **искаме алгоритъма да е in-place**.

Решение на Задача 2

Ако нашият масив е `int a[]` и го обхождаме с `int i`, то имаме:

$$a[i-1]=i;$$

Ще наредим ли числата от 1 до n по този начин? - Да. Бързо ли? Много бързо, само с n присвоявания и 0 проверки.

Задача – редене на елементи през еднакъв интервал

Дайте идея за алгоритъм, който да сортира произволни различни числа за линейно време и $O(n)$ заделяне на памет като единственото, което знаем е, че те са през еднакъв интервал.

Примери за числа през еднакъв интервал:

- A. 2, 4, 6, 8, 10, 12, 14, 16, ...
- B. 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, ...
- C. 0, -100, -200, -300, -400, -500, -600, ...

Идея

- 1) Създаваме допълнителен масив `int temp[]` с големина $n-1$ (това ли е дължината?);
- 2) Ако числата за подредба са съхранени в масив `int a[]`, то с едно обхождане намираме `int min` и `int max`, които съответно са минималния и максималния елемент в `a[]`.

- 3) По тях намираме интервала `int d`, през който са елементите:

$$d = (\max - \min) / (n-1)$$

- 4) Докато обхождаме `a[]` с индекс `int i` попълваме `temp[]`:

$$\text{temp}[(a[i] - \min) / d] = a[i]$$

- 5) Презаписваме `temp[]` в `a[]`.

Проверете за необходимост от корекции, ако боравим с отрицателни числа.

Задача

Реализирайте гореупоменатата идея на C++.

Задача 2 – редене на елементи през еднакъв интервал

Дайте идея за *in-place* алгоритъм, който да сортира произволни различни числа за линейно време като единственото, което знаем е, че те са през еднакъв интервал.

Идея

1) Ако числата за подредба са съхранени в масив `int a[]`, то с едно обхождане намираме `int min` и `int max`, които съответно са минималния и максималния елемент в `a[]`.

2) По тях намираме интервала `int d`, през който са елементите:

$$d = (\max - \min) / (n - 1)$$

3) Докато обхождаме `a[]` с индекс `int i` го подреждаме:

$$a[i] = \min + d * i.$$

Задача 3

А, може ли да наредим произволни цели числа, с възможни повторения на стойности, без да използваме сравнения между тях (non-comparison sorting algorithm)?

Отговор

Една от възможностите е **counting sort**. Този алгоритъм изисква $O(n+k)$ памет и има $O(n+k)$ сложност, където k е размаха на данните (разликата между максималната и минималната стойност), а n е броят на елементите. При по-горните характеристики counting sort е стабилен алгоритъм.

Забележете, че ако $k < n$ (може и по-голямо да е, но да не е в много пъти по-голямо), то този алгоритъм се доближава много до $O(n)$ сложност, т.е. реди масиви от цели числа за линейно време.

Counting sort – съкратена схема

- ▶ Създаваме допълнителен масив, в който ще броим всеки елемент колко пъти се повтаря. За да избегнем обхождания на допълнителния масив го правим толкова голям, колкото е размаха в данните;
- ▶ Всяка една стойност на първоначалния масив се отчита в допълнителния масив като тя директно отговаря на негов индекс. Обхождаме веднъж първоначалния масив и така в допълнителния вече сме отчели всички уникални стойности и по колко пъти се повтарят;
- ▶ Обхождаме допълнителния масив като стойността на всяка следваща клетка е сума на стойността на предходната и самата тя.
- ▶ Използваме новополучения масив и първоначалния, за да сортираме стойностите в трети масив.

Анимация за counting sort

Вижте следния линк:

<https://www.cs.usfca.edu/~galles/visualization/CountingSort.html>

Реализация

Удобни реализации вижте в:

<https://www.geeksforgeeks.org/counting-sort/>

(използвани са някои допълнителни библиотеки)

Реализация без допълнителни библиотеки:

<https://www.includehelp.com/cpp-tutorial/counting-sort-with-cpp-example.aspx>

Тя обхваща ли случаите с отрицателни стойности?

Задача

Може ли да реализираме counting sort без третия масив, където редим сортираниите стойности?

Остава ли алгоритъмът стабилен при подобна реализация?

Задача

Можете ли да конструирате пример с до 9 елемента, такъв че counting sort да стане с изчислителна сложност $O(n^2)$?

Решение

Ако разгледаме масива $[0,1,3,12]$. Тогава $n=4$, $k=12-0=12$.

За counting sort знаем, че е $O(n+k)$ в нашия случай е $O(4+12)=O(16)=O(4^2)$.

Забележете, че така може да конструираме и примери със сложност $O(n^3)$ и по-висока.

Допълнително – Radix sort

Counting sort използва техника различна от стандартното сравняване на елементи. Много ефикасен друг подобен алгоритъм е **Radix sort**.

Специфики:

- ▶ Изчислителна сложност – $O(w * n)$, w – броят битове необходими да се запази всяка уникална стойност;
- ▶ Допълнителна памет – $O(w + n)$;
- ▶ Стабилен алгоритъм е (има и нестабилни разновидности, например in-place реализация).

Допълнително – LSD Radix sort

LSD (least significant digit) Radix sort е стабилен сортиращ алгоритъм. Ще го онагледим с пример:

Вход:

170	45	75	90	2	802	2	66
-----	----	----	----	---	-----	---	----

Сортираме по последна цифра:

170	90	2	802	2	45	75	66
-----	----	---	-----	---	----	----	----

Сортираме по предпоследна:

02	802	02	45	66	170	75	90
----	-----	----	----	----	-----	----	----

Сортираме по първа цифра:

002	002	045	066	075	090	170	802
-----	-----	-----	-----	-----	-----	-----	-----

Допълнително

Знаете ли всички свойства на един сортиращ алгоритъм? Вече знаем за стабилност и **in-place**, има поне още две важни свойства:

- ▶ **Приспособяващ се** (adaptive) – приключва по-бързо при частично наредени масиви;
- ▶ **Online** – може да сортира коректно докато в реално време му се добавят нови елементи за сортиране.

Задача:

Определете алгоритмите, които разглеждахме по **in-place, **adaptive**, **online**, **stable**.**

Допълнително

Може да видите и тествате някои алгоритми за сортиране чрез следния линк:

<https://web.archive.org/web/20150308232109/http://www.sorting-algorithms.com/insertion-sort>

CSCB034 Упражнения по алгоритми и програмиране

ПИРАМИДАЛНО СОРТИРАНЕ

гл. ас. д-р Слав Ангелов, НБУ

Загрявка

Кой е най-бързият начин да намерите сумата на елементите в един масив?

Размисли – Radix sort

Защо Radix sort не е основния сортиращ алгоритъм ?

Отговор

Понеже не е базиран на сравнявания, той среща трудности при:

- ▶ Отрицателни числа – сравнително прост проблем за третиране, но все пак отнема изчислително време;
- ▶ Числа с плаваща запетая – налична е такава модификация, но отново е свързана с усложняване на алгоритъма;
- ▶ Стойности, които не са с еднакъв брой цифри – досещате се как radix добавя по една допълнителна 0 пред по-малките числа, това отново струва ресурси.

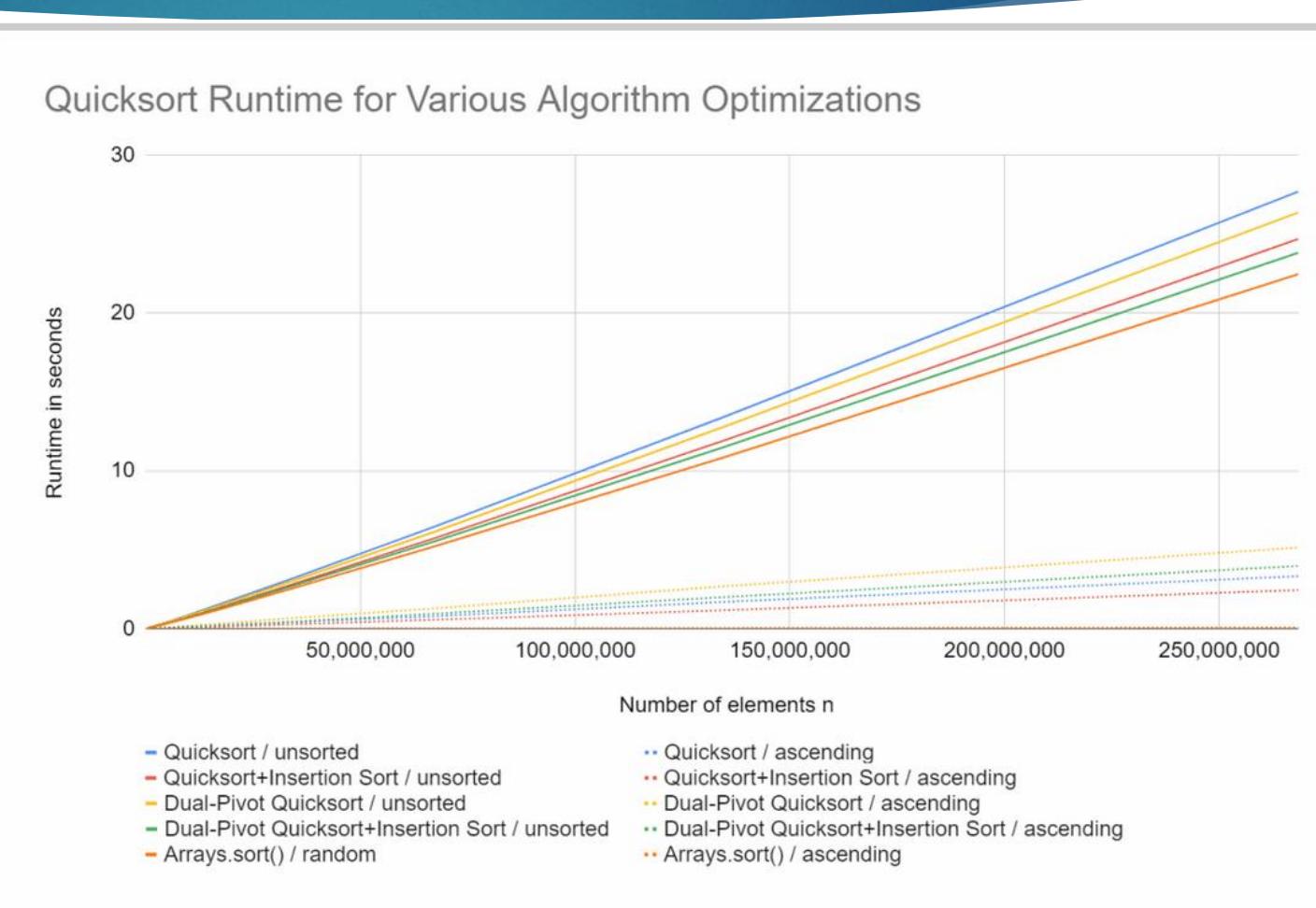
Също така, допълнителната памет, която изисква е с една идея повече от на другите бързи алгоритми за сортиране.

Полезни факти - Quicksort

Сравнения на разновидности на набързият алгоритъм за сортиране. Тестван е върху случаен масиви, но и върху сортирани във възходящ (ascending) и низходящ (descending) ред.

Източник:

<https://www.happycoders.eu/algorithms/quicksort/#Comparing AI Quicksort Optimizations>



Упражнение

Погледнете реализацията на counting sort с масиви:

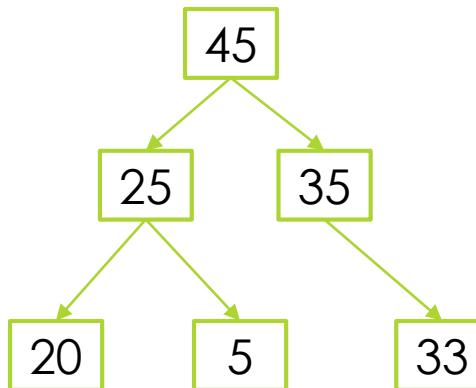
<https://www.includehelp.com/cpp-tutorial/counting-sort-with-cpp-example.aspx>

Работи ли изобщо? Ако има проблеми, коригирайте ги.

Min и Max Heap

Max heap е двоично дърво, в което елементите на всяко следващо ниво са по-малки от „родителите си“ на предходното. За Min heap е обратното.

Пример за Max heap:



Размисли

Можем ли да реализираме двоично дърво чрез масив?

Отговор

Началото на масива ще е корена, в последствие всеки два наследника ще са по формулата **$2*i+1, 2*i+2$** , родителят е под индекс **i**.

Heap sort

Този алгоритъм за сортиране е базиран на двоично дърво, изчислителната му сложност е $O(n * \log(n))$, не изисква допълнителна памет, но е нестабилен.

Алгоритъм:

1. Строим max heap от стойностите за сортиране;
2. На този етап най-големият елемент е в корена на дървото. Заменяме го с последния елемент на дървото. Намаляваме размера на дървото с 1. Отново превръщаме дървото в max heap (процедурата се казва `heapify`);
3. Повтаряме горните стъпки докато дървото не стане с един елемент.

Процедура heapify

Ръководим се от основен принцип. Един възел от дървото е heapified само, ако наследниците му също са heapified.

От тук следва, че дървото е heapified, ако корена му е heapified.

Реализация на heapify (1/2)

```
void heapify(int arr[], int n, int i) {
    int largest = i; // най-големият елемент трява да е в корена
    int l = 2*i + 1; // left
    int r = 2*i + 2; // right

    // Ако left е по-голям от корена:
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // Ако right е по-голям от корена:
    if (r < n && arr[r] > arr[largest])
        largest = r;
```

Реализация на heapify (2/2)

```
void heapify(int arr[], int n, int i) {  
    // .....  
  
    // Ако largest не е корена:  
    if (largest != i) {  
        swap(arr[i], arr[largest]);  
  
        // Рекурсивно heapify по засегнатото под-дърво:  
        heapify(arr, n, largest);  
    }  
}
```

Реализация на основната част на Heap sort()

```
void heapSort(int arr[], int n) {  
    // Строим Max heap в масива:  
    for (int i = n/2-1; i >= 0; i--)  
        heapify(arr, n, i);
```

Обяснете си това
индексиране

```
// Намаляваме heap-а с по един елемент стъпка по стъпка:  
for (int i=n-1; i>=0; i--) {
```

// Разменяме корена с най-малкия елемент:

```
swap(arr[0], arr[i]);
```

Това прилагане на heapify
същото ли е като горните?

// Отново превръщаме heap-а в max heap:

```
heapify(arr, i, 0);
```

```
} }
```

Размисли

Можем ли да реализираме „троично дърво“ в масив? Да ли троичното дърво ще доведе до някакви ползи в Heap sort реализациата?

Отговор

Лесно може да реализираме троично дърво в масив. Ако на индекс i отговаря родител, то децата са на индекси $3*i+1$, $3*i+2$, $3*i+3$.

Забележете, че процедурата може да се продължи за дърва с по четири, пет или повече разклонения от родител.

За дискусия по въпроса с ползите, вижте: <https://en.wikipedia.org/wiki/Heapsort>

Раздел „Variations“, “other variations”, **внимавайте за допуснати грешки.**

Задача

Реализирайте `void heapify(int arr[], int n, int i)` за троично дърво. Трябва ли нещо да пипнем във `void heapSort(int arr[], int n)`, че да стартираме алгоритъма в троично дърво?

Допълнително – двоично търсене

Двоичното търсене (binary search) е метод за по-бързо намиране на позицията на елемент или стойност от масив. Единственото изискване е масивът да е подреден. Предимството му е, че при него размерът на претърсваното пространство намалява наполовина с всяка стъпка;

Основната идея е следната:

- 1) Разделяме масива на две части;
- 2) Сравняваме последния елемент от първата част и първия от втората с търсеното;
- 3) Така разбираме, в коя част е търсеното;
- 4) Разположаваме тази част и повтаряме стъпките.

Binary search

```
int binarySearch(int a[], int item, int low, int high) {  
    if (high <= low)  
        return (item > a[low])? (low + 1): low;  
  
    int mid = (low + high)/2;  
  
    if(item == a[mid])  
        return mid+1;  
  
    if(item > a[mid])  
        return binarySearch(a, item, mid+1, high);  
    return binarySearch(a, item, low, mid-1);  
}
```

Binary insertion sort

```
void insertionSort(int a[], int n) {  
    int i, loc, j, k, selected;  
    for (i = 1; i < n; ++i) {  
        j = i - 1;  
        selected = a[i];  
        // find location where selected should be inserted  
        loc = binarySearch(a, selected, 0, j);  
        // Move all elements after location to create space  
        while (j >= loc) {  
            a[j+1] = a[j]; j--;  
        }  
        a[j+1] = selected;  
    } }
```

Допълнително – големи масиви от данни

Знаете ли колко голям `int array[]` може да създадете в C++?

Разсъждения

При създаването на големи масиви трябва да следим за:

- **От какъв тип са данните в масива** – типове данни с по-малък обем значи повече възможни елементи в масива;
- **Колко бита е операционната система** - 32 битовите поддържат до 4 GB RAM, а ако искаме изчисленията да са по-бързи, то масива трябва да се пази в RAM паметта в стека (stack), т.е. си го дефинираме, както до момента;
- **Ако масивът е в Стека**, то трябва да сме наясно за лимита за Стека от компилатора;
- **Ако масивът е дефиниран в heap-а**, то единственото, което го ограничава е виртуалната памет, тоест зависим от твърдия диск. **Той автоматично ще се дефинира в heap-а**, ако го дефинираме като глобална променлива (**всяка глобална променлива е в heap-а**).
- **Масивът НЕ може да е по-дълъг от максималното число**, което поддържа операционната система.

Реализация на големи масиви

Ако използваме стандартното дефиниране в дадена функция, например `int a[n]`, то масивът се съхранява в stack-а и не може да надхвърля свободната RAM.

За да го запишем в heap-а трябва да използваме команда `new` или `malloc`. Пример:

```
int* a1= new int[SIZE];
```

Допълнително Част 2

Интересували ли сте се какъв сортиращ алгоритъм използва C++, Python или Java:

<https://www.geeksforgeeks.org/know-sorting-algorithm-set-1-sorting-weapons-used-programming-languages/>

- ▶ **C++ sort()** – Introsort (хибрид между quick sort, heap sort и insertion sort);
- ▶ **C++ stable.sort()** – използва Mergesort.

Допълнително Част 3

Множество сортиращи алгоритми и имплементациите им на няколко дузина езици за програмиране:

https://rosettacode.org/wiki/Category:Sorting_Algorithms

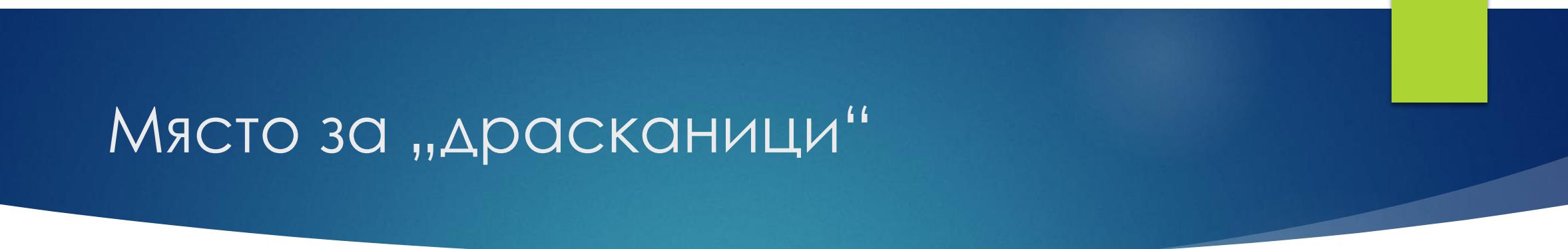
Много добро итеративно графично онагледяване на основни сортиращи алгоритми:

<https://visualgo.net/en/sorting>

CSCB039 Алгоритми и програмиране

УПРАЖНЕНИЯ ЗА КОНТРОЛНО 1

гл. ас. д-р Слав Емилов Ангелов, НБУ



Място за „драсканици“

Задача 1

Имаме n елемента. Оценете сложността на метода на вмъкването (insertion sort) по броя сравнения, които прави. Нека означим тази оценка с $f(n)$. Умножете $f(n)$ по последната цифра на вашият факултетен номер (ако е 0, продължете без умножаване). Означаваме получената функция с $F(n)$. Проверете чрез изчисления следните твърдения:

- 1.* $F(n) \in O(n^2)$;
- 2.* $F(n) \in o(n^3)$;
- 3.* $F(n) \in \Theta(n)$.

Бонус задача

Същото условие, но гледаме Merge sort. Проверете:

1. $F(n) \in O(n)$;
2. $F(n) \in O(\sqrt{n})$;
3. $F(n) \in o(n \log n)$;
4. $F(n) \in \Theta(n^2)$.

Задача 2

Имате следния вход за МПД:

Индекс:	0	1	2	3	4	5	6	7	8
Стойност:	1	9	9	0	2	5	0	3	4

Какво извежда следния код, променливата **a** е последната цифра от факултетния ви номер:

- 1) LOAD# 0
- 2) STORE 0
- 3) INPUT
- 4) OUTPUT
- 5) LOAD 0
- 6) ADD# 1
- 7) STORE 0
- 8) SUB# **a**
- 9) JMPN 3
- 10) END

Задача 3

Напишете програма за МПД, която при входа от Задача 2 да обходи всички стойности и да изпринтира само четните стойности.

Задача 4

Напишете програма за МПД, която при входа от Задача 2 да обходи всички стойности и да изпринтира сумата им, ако тя е по-голяма от 20.

Задача 5

Напишете програма за МПД, която при входа от Задача 2 да обходи всички стойности и да изпринтира всички стойности, които отговарят на четни позиции(индекси), нулата се брои за четно число.

Задача 6

Използвайки МПД, оценете сложността на метода на вмъкването:

```
void insertion_sort(int arr[], int n){  
    int key, j;  
    for (int i=1; i<n; i++){  
        key=arr[i];  
        j=i-1;  
        while ( j>=0 && arr[j]>key){  
            arr[j+1]=arr[j]; j --; }  
        arr[j+1]=key;  
    }  
}
```



CSCB034 Упражнения по алгоритми и програмиране

„РАЗДЕЛЯЙ И ВЛАДЕЙ“ ПОДХОД

гл. ас. д-р Слав Ангелов, НБУ

Размисли

Можем ли да кажем, че строенето на минимално или максимално дърво е на принципа разделяй и владей?

Отговор

При дърветата се наблюдава подразделяне на всяка стъпка, което е сходно на принципа „разделяй и владей“. Например при двоични дървета на всяка стъпка дървото се дели на две разклонения, които могат да се третират като поддървета. Ако искаме да оформим Max или Min heap, трябва първо да наредим тези поддървета, т.е. решаваме две подзадачи на аналогичен принцип и ги обединяваме в една. **А какво назова принципа разделяй и владей? Имаме задача, разделяме я на подзадачи, решаваме ги и сглобяваме общото решение.**

Извод: Може да разглеждаме строенето на max или min heap като задача от типа „разделяй и владей“.

Въпрос

А можем ли да третираме всяка рекурсия като „разделяй и владей“?

Отговор

Всяко едно рекурсивно извикване вика в себе си друго рекурсивно извикване. Този процес се повтаря докато не се стигне дъното на рекурсията. В последствие тя спира или се връща до първоначалното извикване.

Това подразделяне на подзадачи и евентуалното им сглабяне отново е сходно на „разделяй и владей“, но дали винаги е базирано на този принцип.

Множествената рекурсия често е обвързана с принципа „разделяй и владей“, докато единичната рекурсия е по-близо до динамично програмиране. Но дори и множествената рекурсия може да не представлява този принцип, например при числата на Фиbonачи.

Бонус: пародия на „разделяй и владей“

Съществува и пародия на принципа „разделяй и владей“, наречена „**размножавай и предай се**“(multiply and surrender). Пример за такава неефикасна техника е **Slowsort**. Дори и най-добрият му случай е по-бавен от метода на мехурчето.

Псевдокод:

```
procedure slowsort(A, i, j)
    if i ≥ j then return m := floor((i+j) / 2)
    slowsort(A, i, m)
    slowsort(A, m+1, j)
    if A[j] < A[m] then swap A[j] and A[m]
    slowsort(A, i, j-1)
```

Идея:

1. Сортираме рекурсивно първата половина;
2. Сортираме рекурсивно втората половина;
3. Намираме по-големият от най-големите елементи от стъпки 1. и 2. и го записваме в края;
4. Повтаряме цялата процедура без последния елемент.

Загрявка – троични дървета

Реализирайте троично дърво.

Heapify за троично дърво (1/2)

```
void heapify(int arr[], int n, int i) {
    int largest = i; // най-големият елемент трябва да е в корена
    int l = 3*i + 1; // left
    int c = 3*i + 2; // center
    int r = 3*i + 3; // right

    // Ако left е по-голям от корена:
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // Ако right е по-голям от корена:
    if (r < n && arr[r] > arr[largest])
        largest = r;
```

Heapify за троично дърво (2/2)

```
void heapify3(int arr[], int n, int i) {  
    // .....  
    // Ако center е по-голям от корена:  
    if (c < n && arr[c] > arr[largest])  
        largest = c;  
  
    // Ако largest не е корена:  
    if (largest != i) {  
        swap(arr[i], arr[largest]);  
  
        // Рекурсивно heapify по засегнатото под-дърво:  
        heapify3(arr, n, largest);  
    } }
```

Има ли проблем, че НЕ
сложихме този код между
проверките за left и right ?

Главна функция за троично дърво

```
void heapSort3(int arr[], int n) {  
    // Строим Max heap в масива:  
    for (int i = n/3; i >= 0; i--)  
        heapify3(arr, n, i);  
  
    // Намаляваме heap-а с по един елемент стъпка по стъпка:  
    for (int i=n-1; i>=0; i--) {  
        // Разменяме корена с най-малкия елемент:  
        swap(arr[0], arr[i]);  
  
        // Отново превръщаме heap-а в max heap:  
        heapify3(arr, i, 0);  
    } }
```

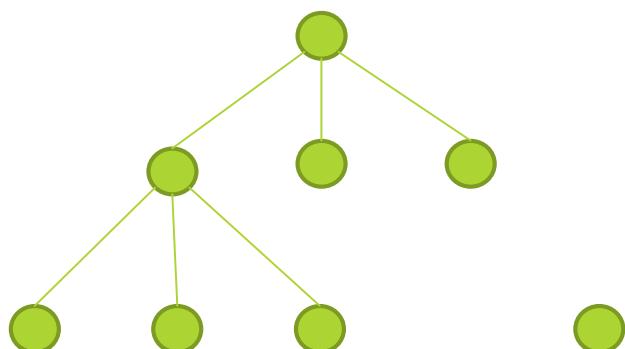
На следващия слайд ще коментираме този индекс

Защо точно този индекс? (1/2)

При първоначалното строене на max heap с по три разклонения използваме стартова точка в масива $n/3$ вместо $n/3-1$, което е случая на двоичното дърво. Защо избрахме този индекс?

Антитипимер (масив с 8 елемента) при старт $n/3-1$:

$n = 8$, старт = $8/3 - 1 = 1$. Тогава, ако следваме алгоритъма дървото ще изглежда по следния начин:

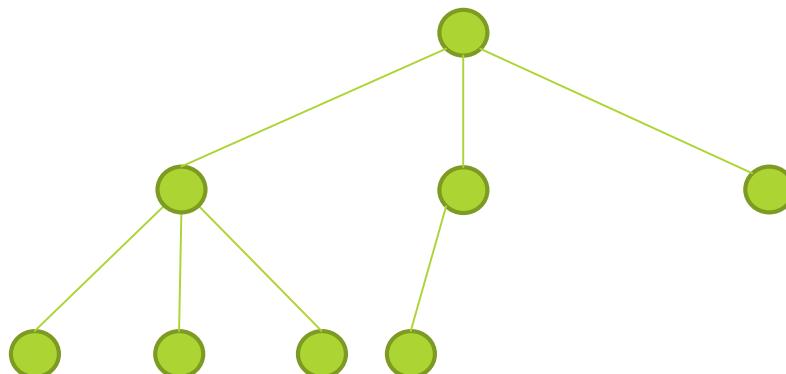


Едно от децата Не е свързано с дървото.

Защо точно този индекс? (2/2)

Разсъждения за старт $n/3$:

$n = 8$, старт = $8/3 = 2$. Тогава, ако следваме алгоритъма дървото ще изглежда по следния начин:



Въобще всеки старт по-голям от $n/3$ ни върши работа, но той е оптимален от гледна точка на операции.

Практика

Разпишете няколко двоични и третични дървета за масиви с различна дължина (4-9). Следете по колко сравнения и размествания ще направи всяко дърво преди да нареди числата по метода на heap sort (допускайте винаги най-лошия случай). Какво установявате?

„Разделяй и владей“ при алгоритмите за сортиране

- ▶ **Merge sort** – в детайли разгледахме принципа му на действие.
- ▶ **Binary insertion sort** – метода на вмъкването, но с двоично търсене къде да се постави следващия елемент сред сортираниите. Забележете, че „разделяй и владей“ се проявява именно в двоичното търсене.
- ▶ **Quicksort** – на всяка стъпка избираме стойност (*pivot*) и подразделяме множеството от стойности на такива на ляво от *pivot*-а и на дясно от него.
- ▶ **Average sort** – разделяме масивът на две части благодарение на средната стойност. За подмасивите повтаряме същата операция.
- ▶ **Bucket sort** – стойностите се групират на подмножества. В последствие тези подмножества се групират на по-малки подмножества или директно се редят с някой друг сортиращ алгоритъм.

Bucket sort

Нуждата от Bucket sort възниква, защото counting sort не може да се справя с числа с плаваща запетая.

Изчислителна сложност: $O(n^2)$ (най-лош случай); $O(n)$, ако $n \approx k$.

Памет: $O(n * k)$ (най-лош случай), k – брой кофи (контейнери).

Стабилност: Зависи от имплементацията (помислете по въпроса).

Съществуват множество реализации и версии на алгоритъма.

Визуализация на реализация с linked lists:

<https://www.cs.usfca.edu/~galles/visualization/BucketSort.html>

Реализация с вектори

```
#include <algorithm>
#include <vector>
void bucketSort(float arr[], int n){
    // Създаваме n кофи:
    vector<float> b[n];
    // Разпределяме стойностите по кофите:
    for (int i=0; i<n; i++) {
        int bi = n*arr[i];
        b[bi].push_back(arr[i]);
    }
    // ....
    // Сортираме всяка кофа:
    for (int i=0; i<n; i++)
        sort(b[i].begin(), b[i].end());
    // Обединяваме кофите в arr[]:
    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
            arr[index++] = b[i][j];
}
```

Упражнение

Можете ли да конструирате „най-тежък случай“ за реализацията на bucket sort с вектори (източник: <https://www.geeksforgeeks.org/bucket-sort-2/>), ако всички стойности са < 1 ?

Примерно решение

```
float arr[10] = {0.009 , 0.008, 0.007, 0.006, 0.005, 0.004, 0.003, 0.002, 0.001, 0};
```

Така всички стойности отиват в кофа с индекс 0 и се налага допълнителния сортиращ алгоритъм да сортира цялото множество наведнъж. В настоящата реализация този допълнителен сортиращ алгоритъм се явява вградения в C++, т.е. Интросорт, а Интросорт ще изпозва метода на „вмъкването“ за толкова малко елементи в Кофа 0, поради подредбата вмъкването ще реализира пълната си сложност $O(n^2)$.

Реализацията на Bucket sort в този вид позволява бързо да се сортират равномерно разпределни стойности с плаваща запетая, но ще среща множество забавяния в много други случаи, когато стойностите са концентрирани около един или няколко индекса на кофи.

Размисли

Ще работи ли реализацията с вектори на Bucket sort за стойности ≥ 1 ?

Отговор

Не, понеже ще излезем извън идексациите на $b[n]$.

Размисли

Ще работи ли реализацията с вектори на Bucket sort за стойности <0?

Отговор

Не, понеже ще излезем извън идексациите на $b[n]$. Но може лесно да оправим проблема като намерим най-малкия елемент и прибавим неговата абсолютна стойност към всички елементи в масива, така той ще започва от 0 (да не забравите след това отново да върнете трансформацията при наредения масив).

Задача

Направете реализацията с вектори на bucket sort, така че да работи с цели числа и с числа с плаваща запетая (всичките са положителни).

Упътване

Всичко остава непроменено. Проблемът беше във формулата за индексация на кофите. Сменете я с тази от реализацията на bucketsort със свързани списъци, която я имаше на визуализацията, т.е. $bi=(n*arr[i])/(Max(arr[]))+1$.

А какво ще стане, ако използваме индексацията на кофите (k кофи) от линка в GeekforGeeks, т.е. $bi=k*(arr[i]-Min(arr[]))/(Max(arr[])-Min(arr[]))$???

Упражнение със средна трудност

Може ли да се реализира bucketSort() само с масиви ?

Допълнително – универсална функция print_array()

Как да реализираме функция за принтиране на стойностите на масив така, че да няма значение от кой тип са данните в масива?

Решение

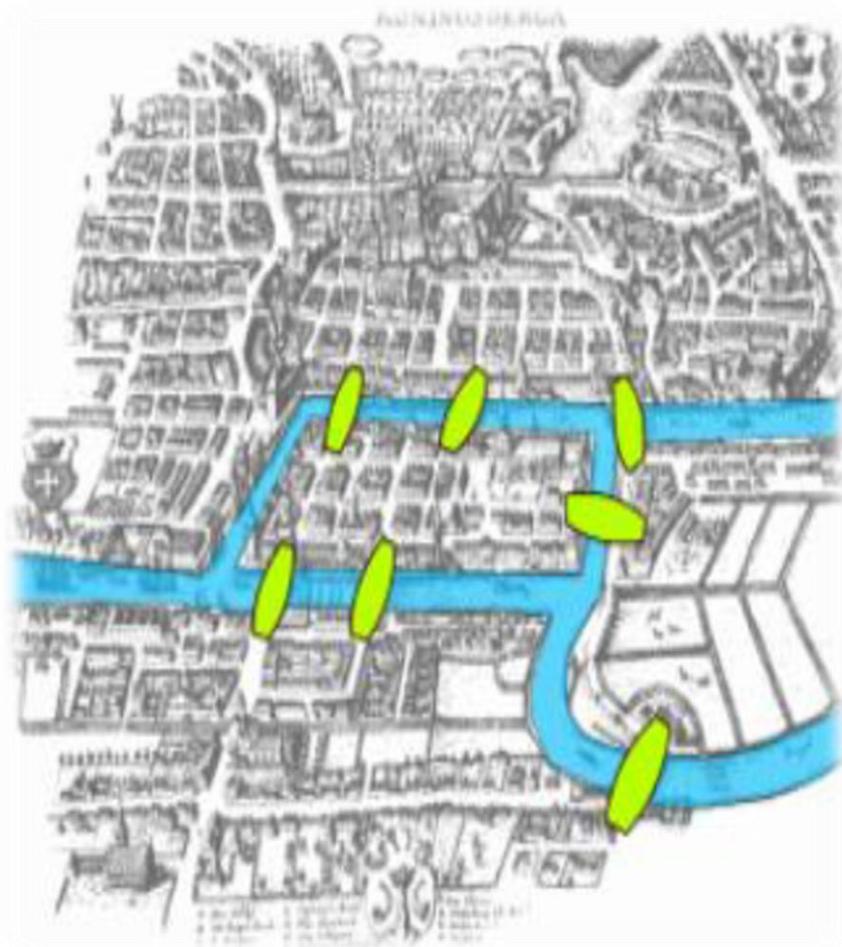
```
void print_array (auto arg[], int length) {  
    for (int n=0; n < length; ++n)  
        cout << arg[n] << ' ';  
    cout << '\n';  
}
```

CSCB039 Упражнения по алгоритми и програмиране

ГРАФИ

гл. ас. д-р Слав Ангелов, НБУ

Ойлеров цикъл

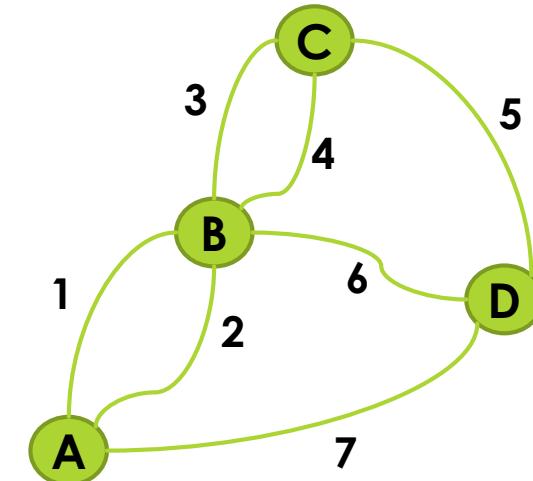
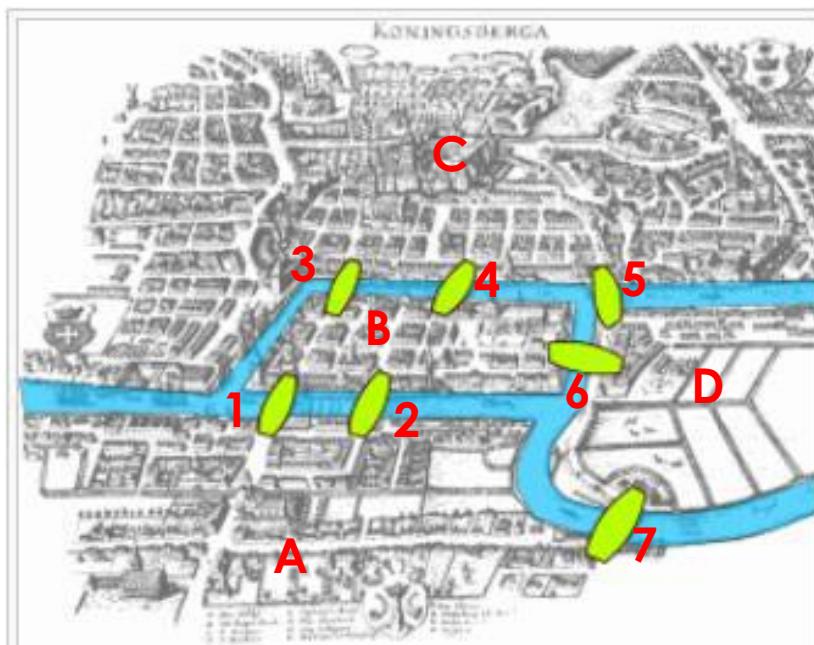


Наблюдаваме мостовете в град Кьонингсберг, Прусия. Два бряга, 2 острова и 7 моста, които ги свързват. Можем ли да започнем от даден мост, да обиколим всичко мостове и да се върнем в стартова точка без да повторим нито един мост?

Леонард Ойлер (1707 – 1783) през 1736 дава решение на този проблем. Това се счита за първата задача от теория на графите.

Онагледяване

Нека всяко парче суша представим като точка, а мостовете да са връзките между тези точки, тях ще онагледим с криви:

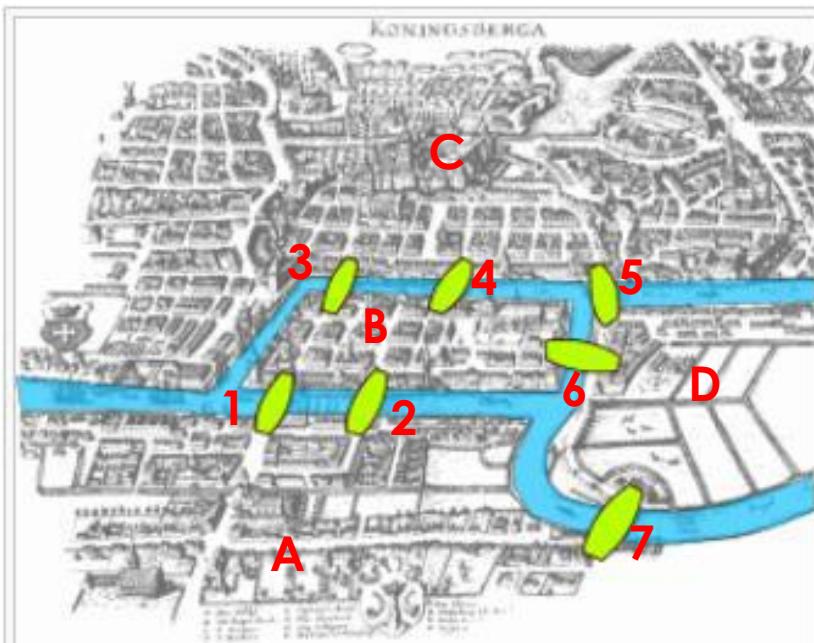


Казус

Можем ли да минем по всеки мост веднъж и да се върнем в началото?
Обосновете отговора си.

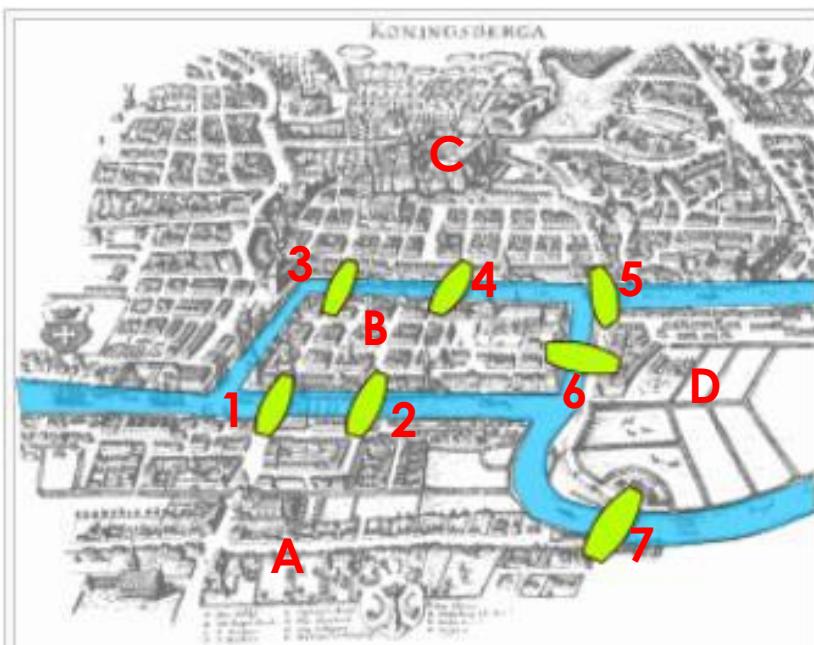
Казус 2

**Кои мостове да премахнем, че задачата да има решение? Придържайте се към
минимален брой премахнати мостове.** (възможни са няколко отговора)

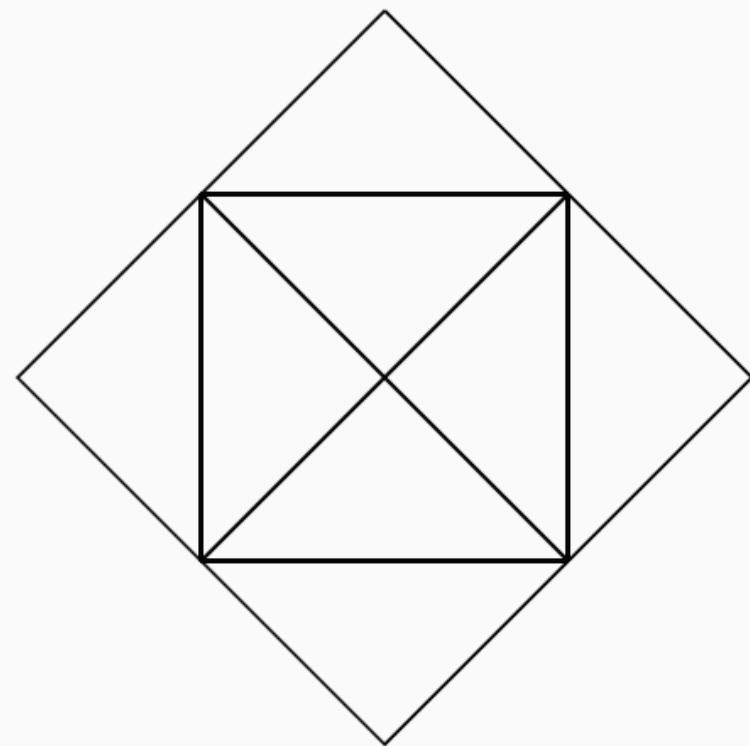


Казус 3

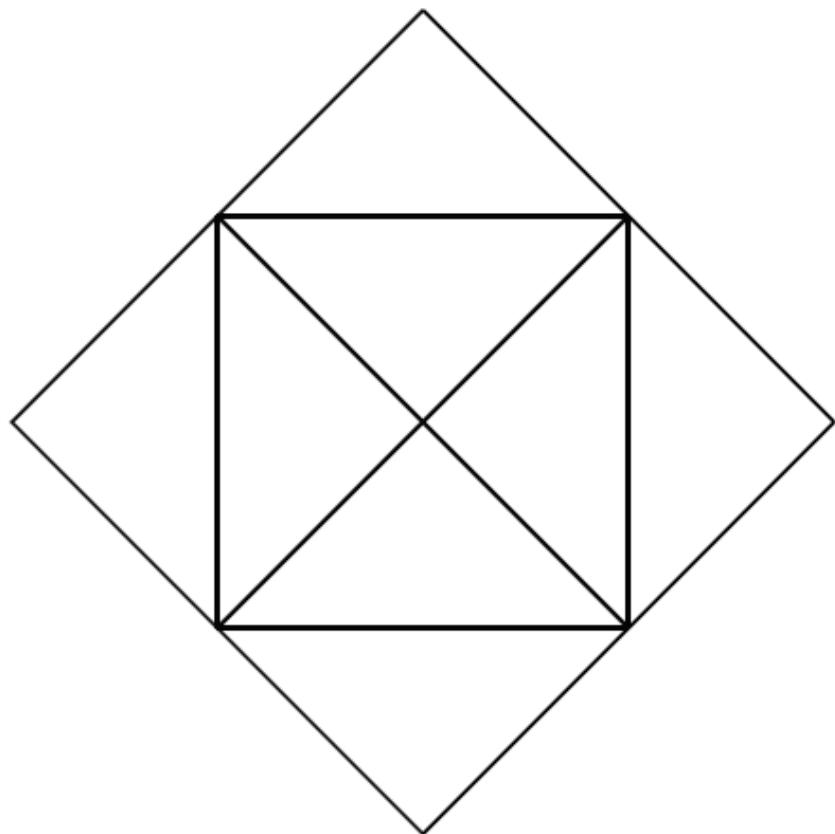
**Колко моста да добавим, че задачата да има решение? Придържайте се към
минимален брой добавени мостове.** (възможни са няколко отговора)



Можете ли да нарисувате фигурата
без да си вдигате ръката ?

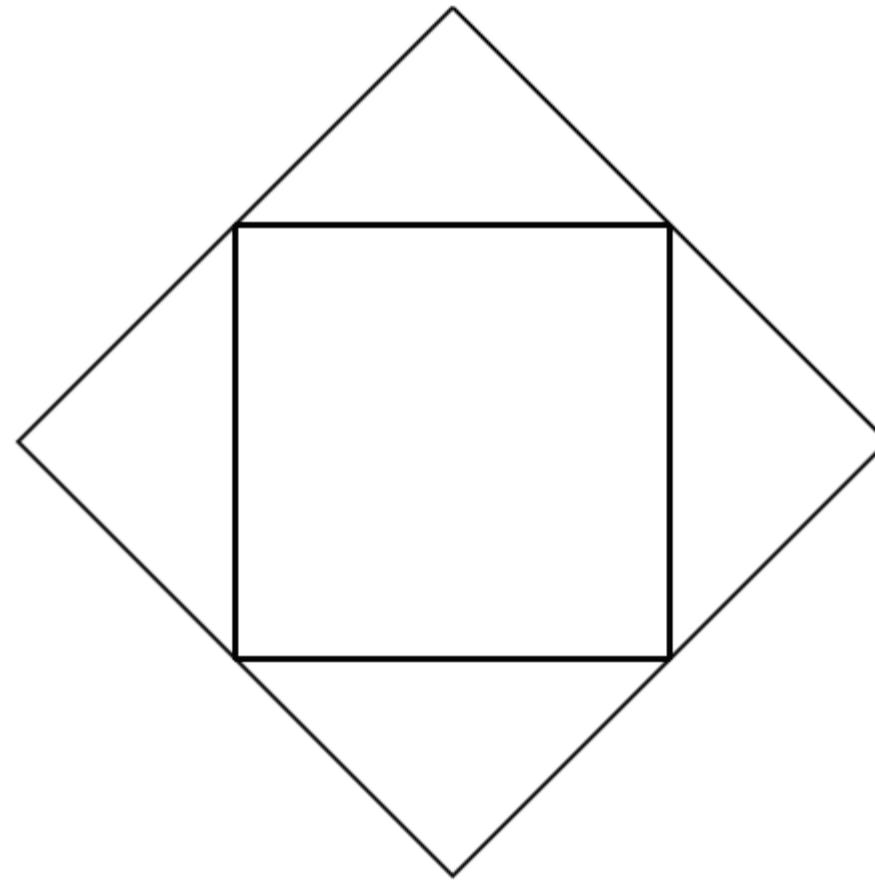
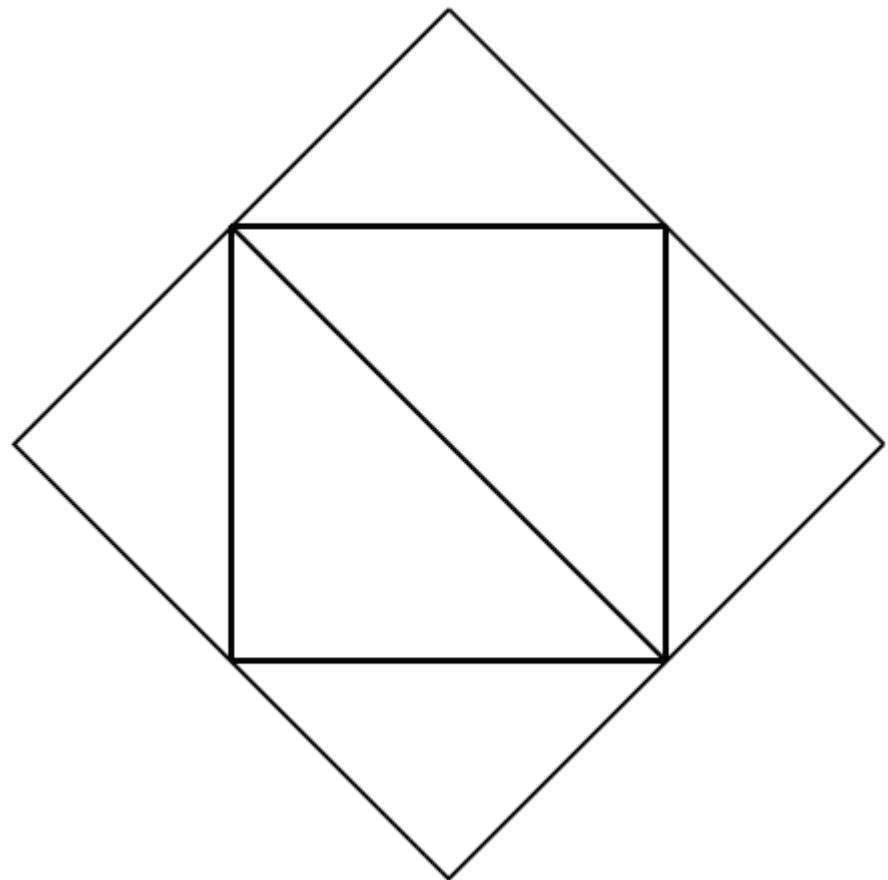


Можете ли да нарисувате фигурата
без да си вдигате ръката ?



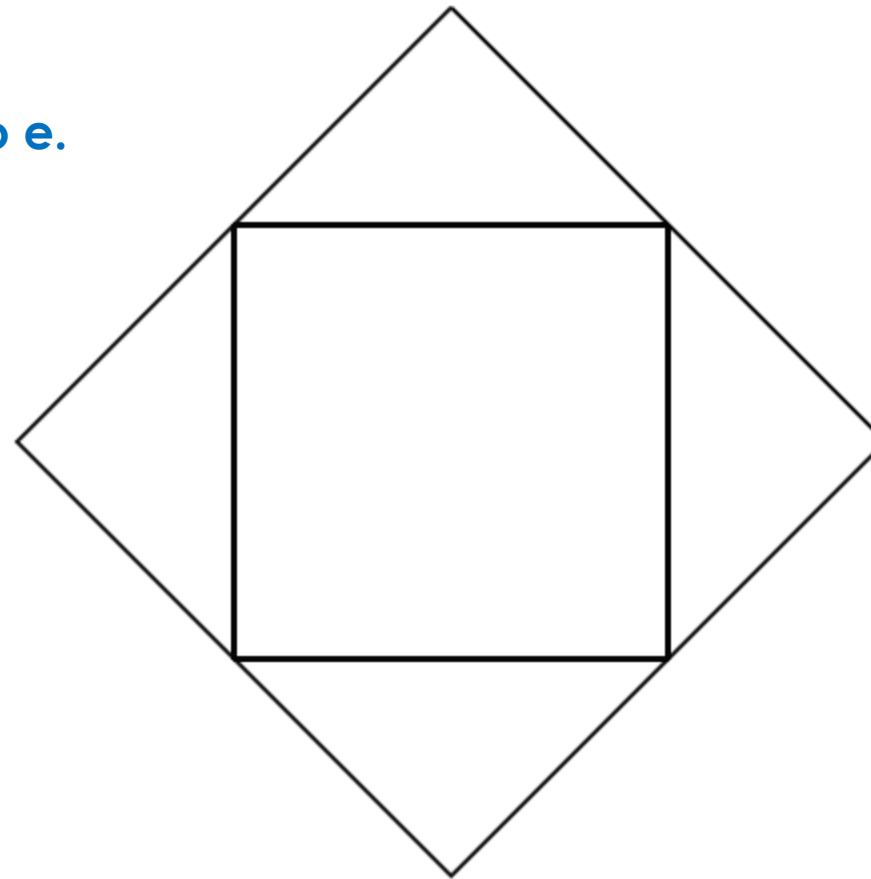
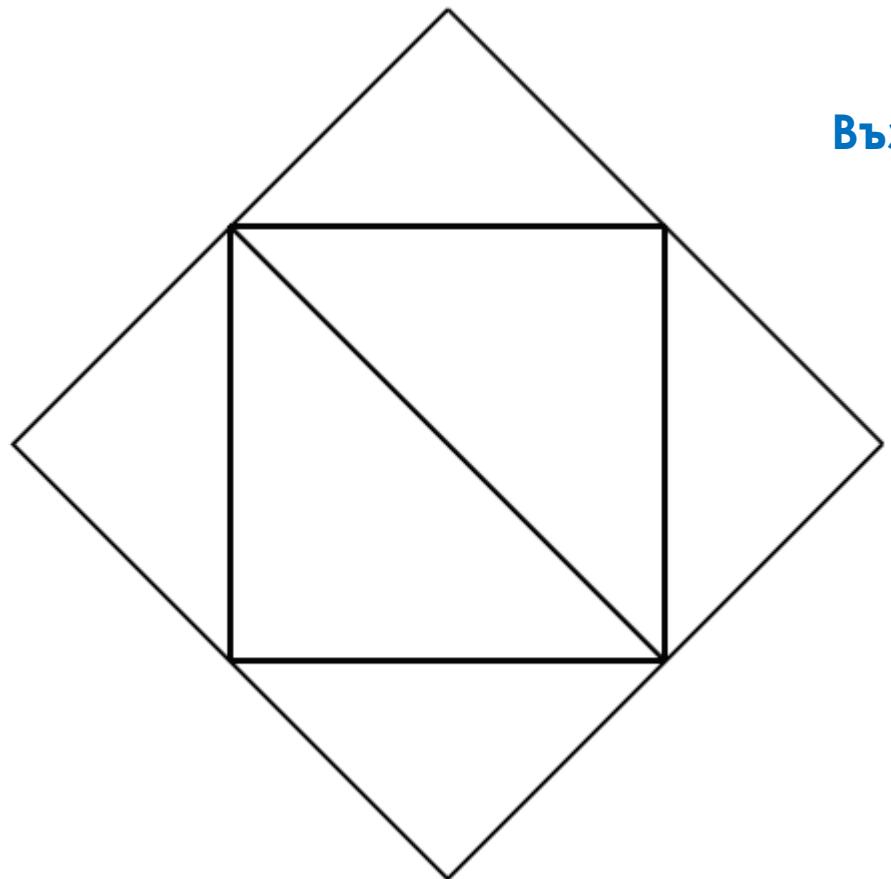
Уви, Не е възможно.

Можете ли да нарисувате фигурите
без да си вдигате ръката ?

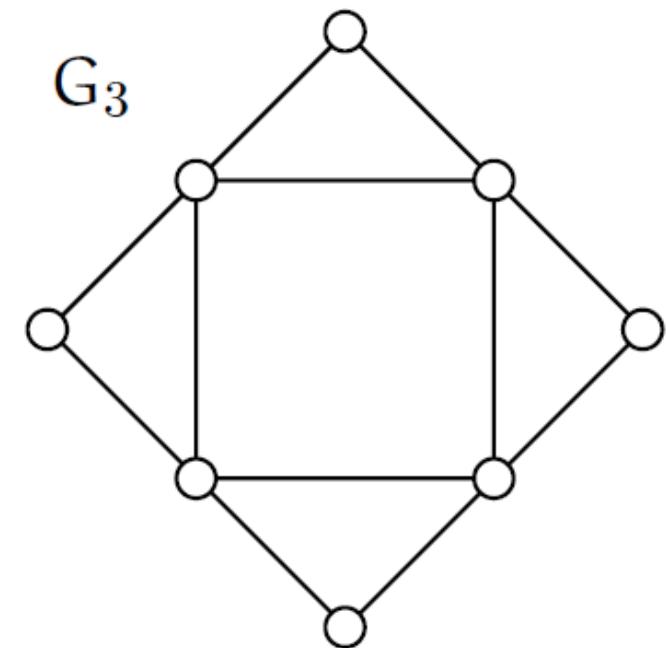
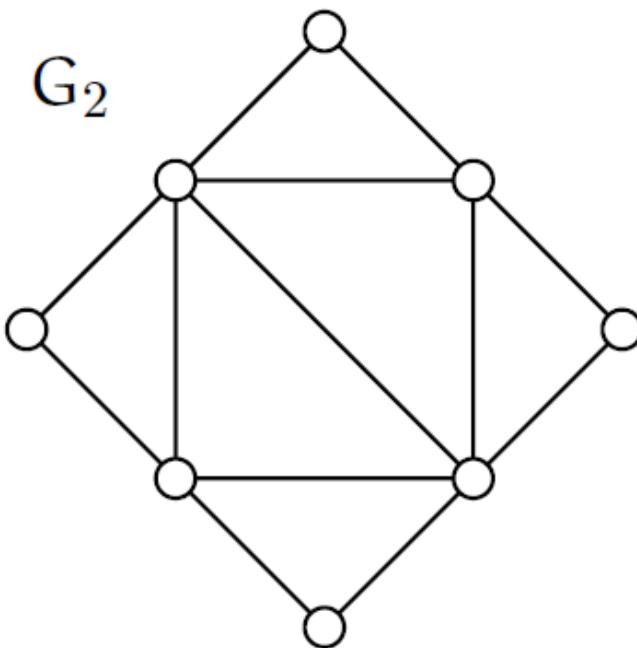
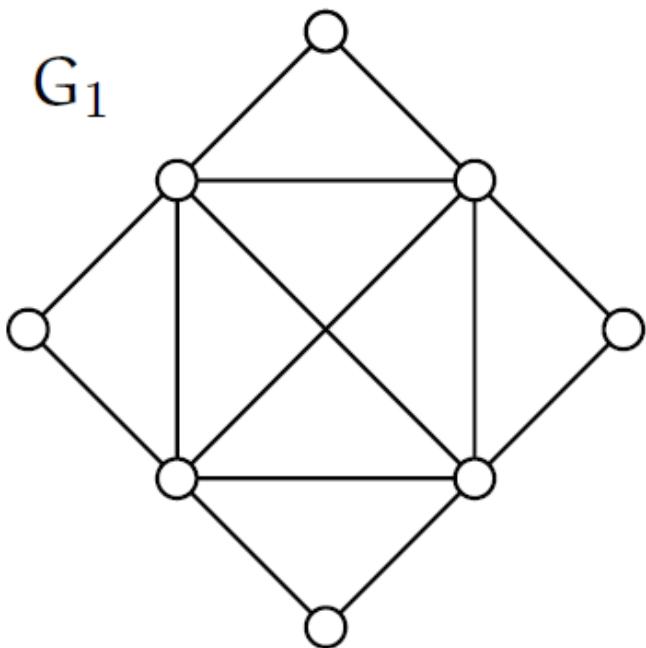


Можете ли да нарисувате фигурите
без да си вдигате ръката ?

Възможно е.



Онагледяване на фигури чрез графи



Теорема: Ойлеров цикъл в свързан мултиграф

G има Ойлеров цикъл тогава и само тогава, когато всеки връх има четна степен.

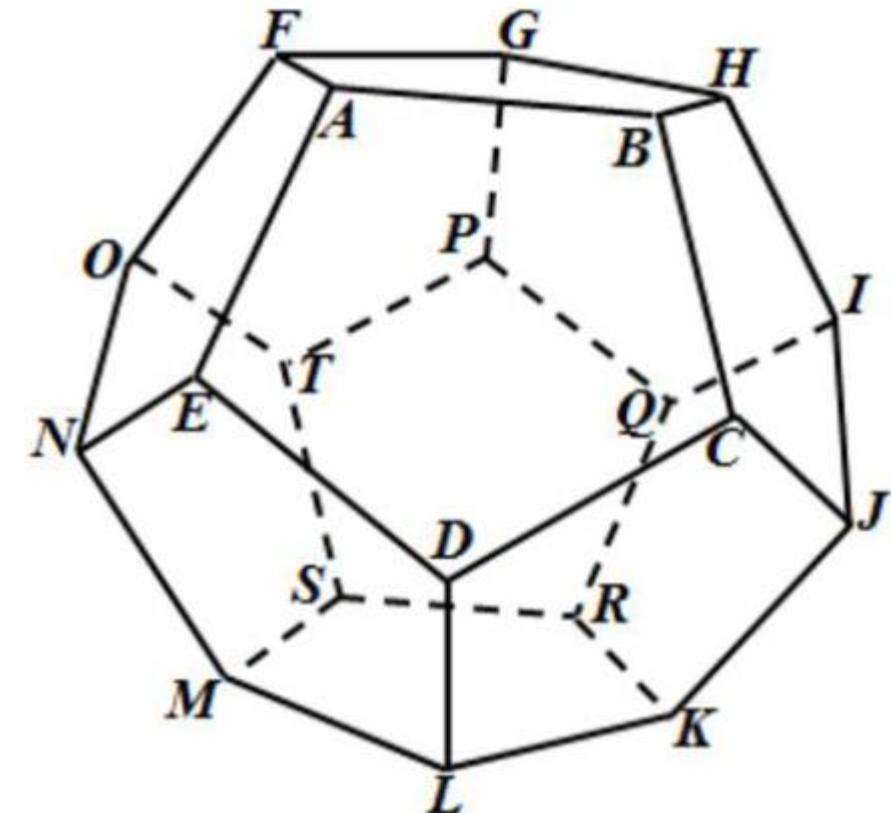
Степен на връх: Броят на ребрата, които го свързват с други върхове.

Хамилтонов цикъл

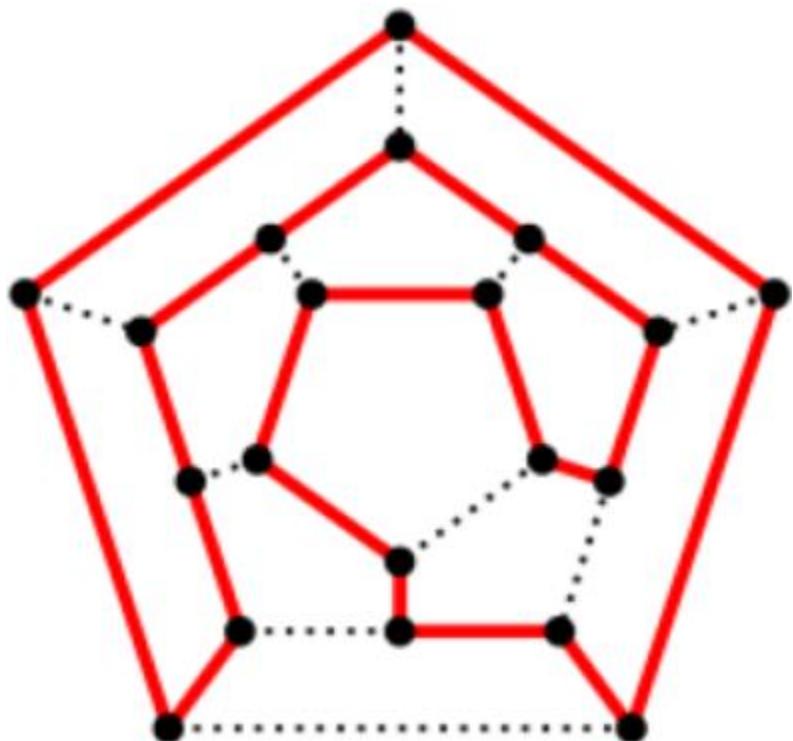
Уилям Хамилтон (1805-1865) формулира втората задача от теория на графиките.

Задачата: Да се тръгне от произволен връх на додакаедрон, да се върви само по ръбовете му и като обходим всичките му върхове без повторения да се върнем в начална точка.

Додакаедрон: Правилен изпъкнал многостен. Състои се от 12 стени, правилни петоъгълници.



Примерно решение



За упражнение може да именувате
върховете, като гледате додакаедрона
от предходния слайд.

Допълнително

Знаете ли, че додакаедрона (или додакаедър) е едно от платоновите тела?

Те са общо 5 на брой:

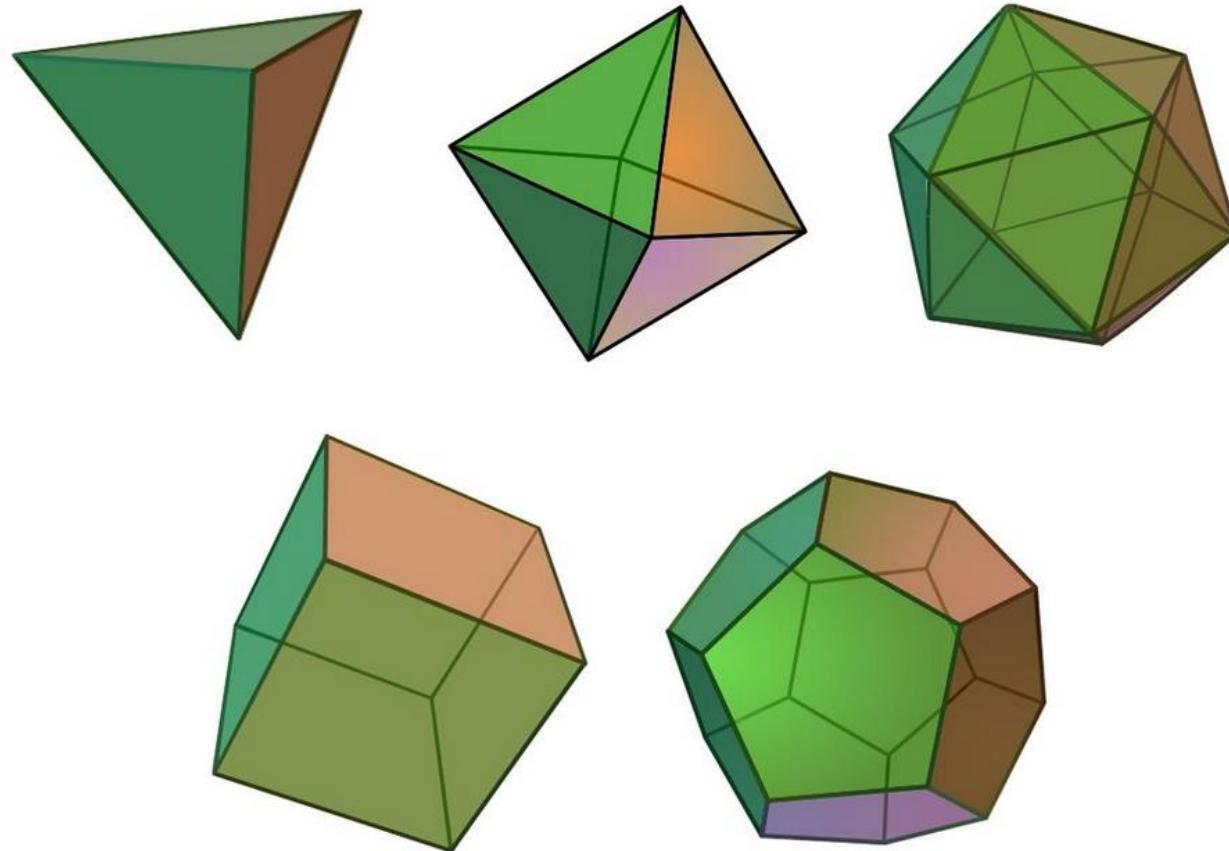
https://bg.wikipedia.org/wiki/%D0%9F%D0%BB%D0%B0%D1%82%D0%BE%D0%BD%D0%BE%D0%B2%D0%BE_%D1%82%D1%8F%D0%BB%D0%BE.

Всяко едно от тях символизира един от четирите елемента (огън, земя, вода и въздух). Интересно е, че додакаедъра не представлява един от стандартните „стихии“, а е отъждествен като Цялото, „космоса“, „етера“, поради това е бил най-ревниво пазената фигура от Платон и неговите последователи.

В тримерно пространство няма как да създадете други правилни многостенни освен платоновите такива, интересно е какво се случва в по-високи размерности:
<https://www.youtube.com/watch?v=2s4TqVAbfz4>

Размисли

Дали е възможно да решим задачата на Хамилтон за всяко от Платоновите тела?



Елементи от теория на графите

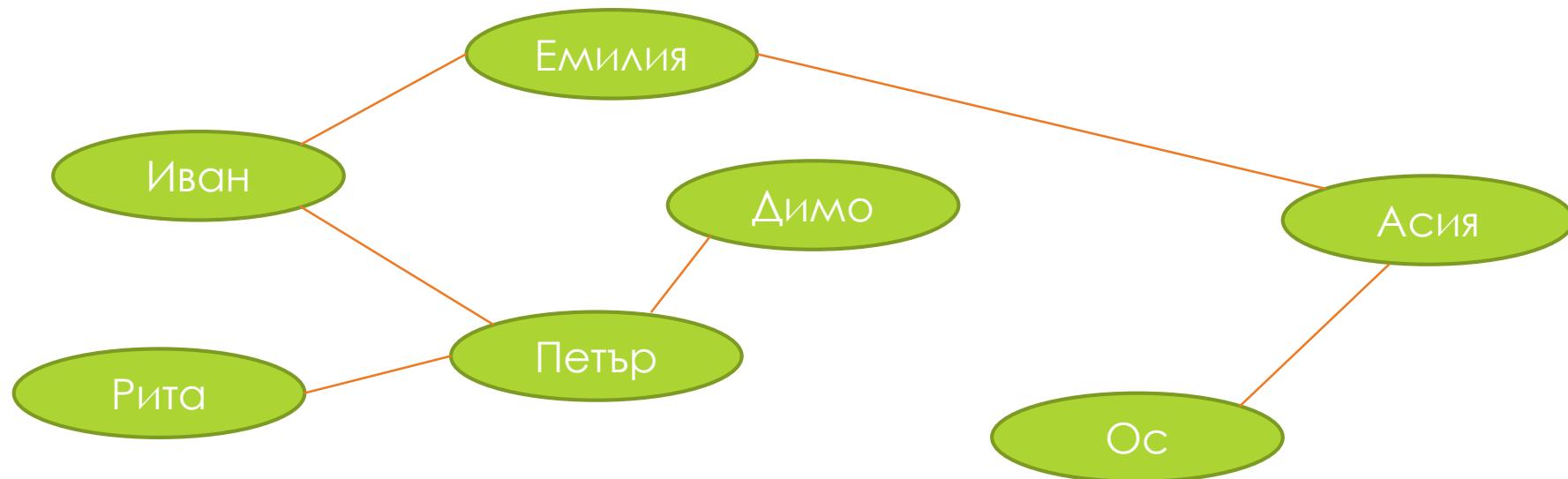
Следва да се знаят следните термини:

- ▶ Граф – връх, ребро;
- ▶ Мултиграф;
- ▶ Ориентиран граф;
- ▶ Свързан граф;
- ▶ Път;
- ▶ Цикъл;
- ▶ Матрица на инцидентност, матрица на съседство.

За повече информация вижте слайдовете на проф. Манев.

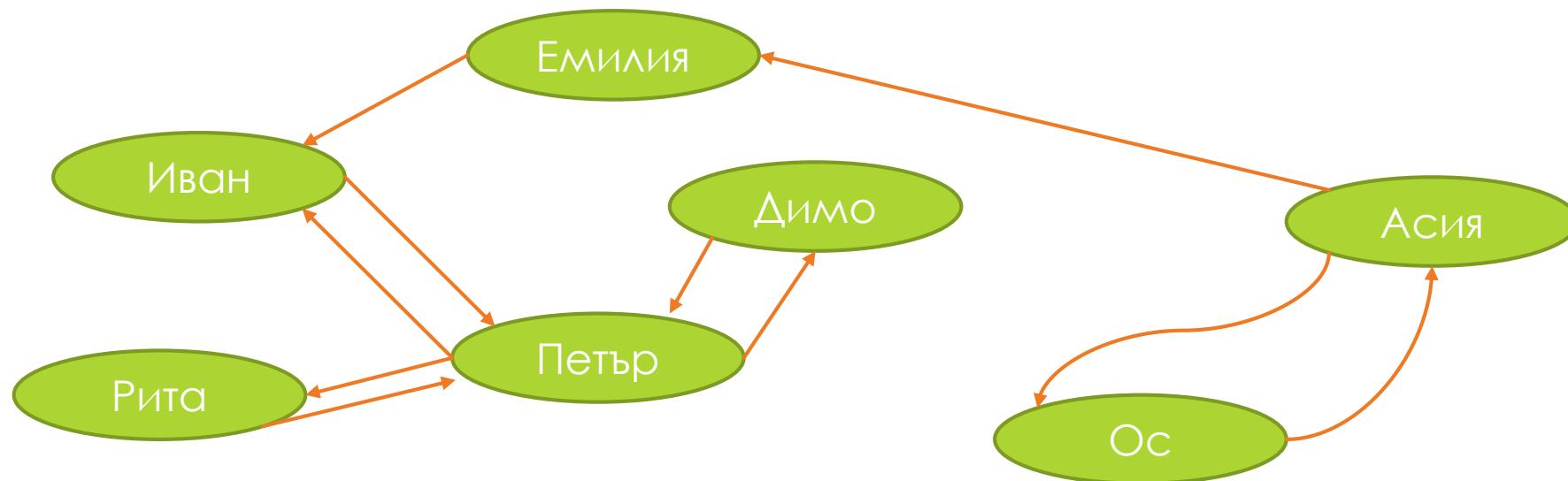
Приложение в социалните мрежи

Контактите между отделни индивиди може лесно да се моделират чрез графи.
Фейсбук:



Приложение в социалните мрежи

Контактите между отделни индивиди може да не са двустранни. **Twitter:**



Проблемът с оцветяването

Нека например имаме картата на България по региони:

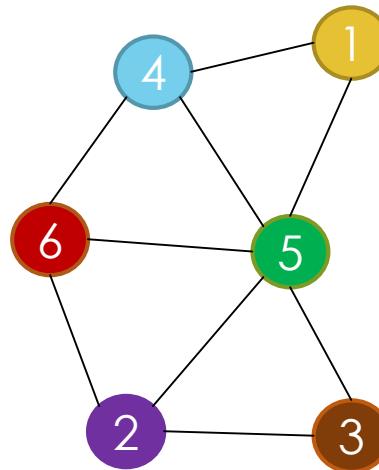
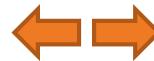
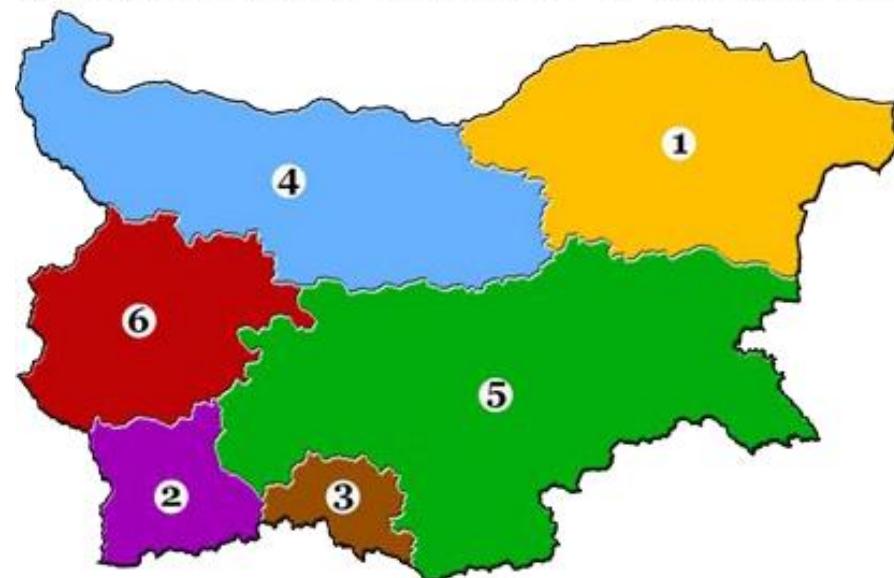


С колко най-малко цвята можем да я оцветим така, че да няма съседни региони с един и същи цвят?

(Може да експериемнтирате като копирате картата в Paint и оцветявате регионите един по един)

Защо проблемът с оцветяването е проблем от теория на графиките ?

ЕТНОГРАФСКИ ОБЛАСТИ В БЪЛГАРИЯ



Теорема за четирите цвята

Ако разделим една равнина на региони (например да направим карта), то ще са ни необходими четири цвята така, че всеки два съседни региона да са с различни цветове.

Забележка:

Ако два региона се допират само в една точка, то те не се считат за съседни.

Факт: Проблемът е формулиран през 1913 година от H.Dudeney, а е решен едва след век от K.Appel и W.Haken.

Упражнение

Можем ли етнографската карта на България от предходните слайдове да я сведем до четири цвята, така че да няма две съседни области с еднакви цветове?

Размисли

Можете ли да се сетите за условия, при които теоремата за четирите цвята няма да сработи при оцветяване на страните по света?

Отговор ще намерите в следния видеоклип:

<https://www.youtube.com/watch?v=ANY7X-wpNs>

Проблеми от теория на графите и алгоритмите

- ▶ Най-къс път в мрежа;
- ▶ Планарност на граф – **какво е планарен граф ?;**
- ▶ Свързаност на граф;
- ▶ Намиране на минимално покриваща дърво;
- ▶ Алгоритми за намиране на матрицата на съседство;
- ▶ Алгоритми за намиране на циклите в граф;
- ▶ Алгоритми за търсене на елемент в структура от данни;
- ▶ Други.

Задача

Имате граф, зададен по списък на ребрата в двумерен масив `int G[][]`, където $(G[i][0], G[i][1])$ е i -тото ребро. Напишете програма, която да извежда броя на върховете в този граф.

Хипер тежка задача за Гикове

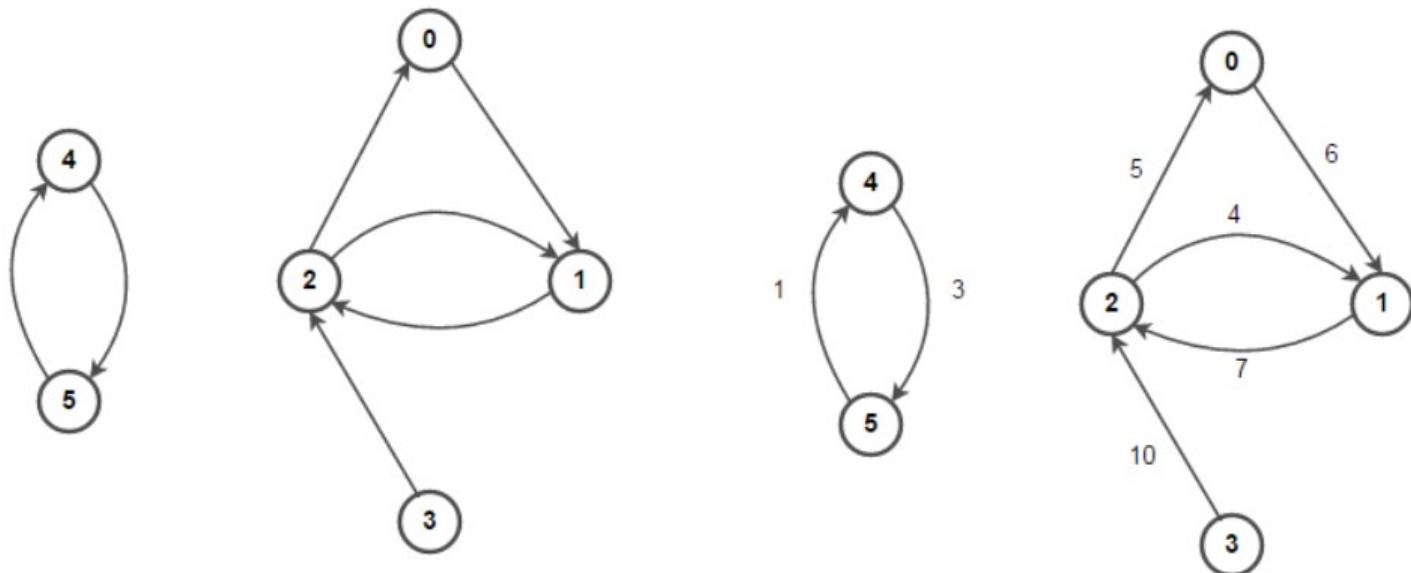
Допускаме, че имаме въведен чрез списък на ребрата граф, който представлява кубът на Метатрон (Metatron's Cube). Напишете програма, която да проверява дали дадено платоново тяло се съдържа в този куб и къде точно се намира в него, т.е. изходът трябва да е точните върхове и ребра, които участват в изграждането на конкретния платоноид, например списък на ребрата.



Алгоритми и програмиране Графи

Задача 1

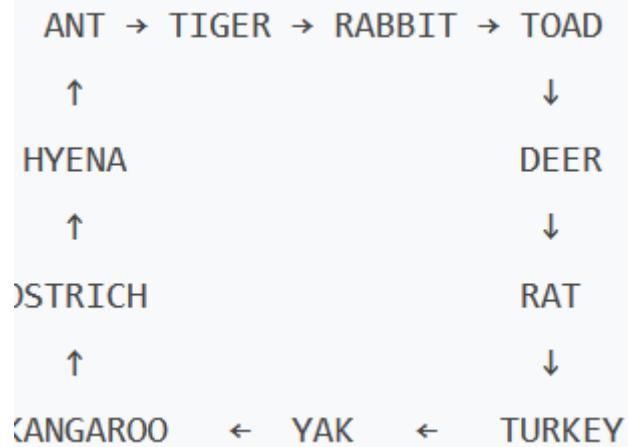
- Даден е ненасочен или насочен граф. Имплементирайте структура от данни за граф, използвайки STL.
 - Решете задачата както за претеглени, така и за непретеглени графи, използвайки матрица на съседство за представяне на графа.
 - Решете задачата без да използвате STL.



Задача 2

- Дадено е множество от думи. Напишете програма, която проверява дали отделните думи могат да бъдат пренаредени, така че да образуват кръг.
 - Две думи, x и y , могат да бъдат част от кръг, ако последният символ на x е същият като първия символ на y или обратно.

[ANT, OSTRICH, DEER, TURKEY, KANGAROO, TIGER, RABBIT, RAT, TOAD, YAK, HYENA]

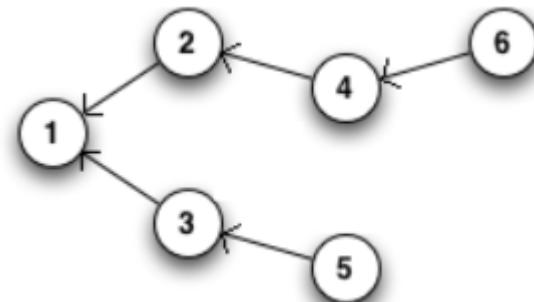
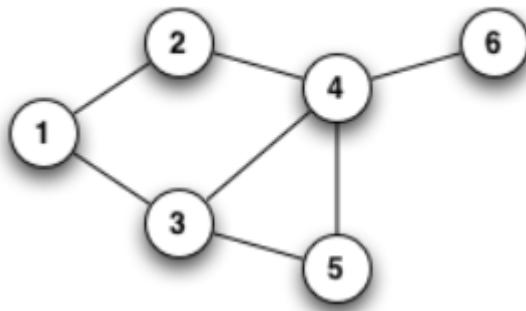


Задача 2 - Решение

- Изградете насочен граф, като за всяка дума от даденото множество добавяте ребро в граф за първия и последния символ.
 - Проблемът се свежда до намиране на Ойлеров цикъл в граф.
- Ако полученият граф има Ойлеров цикъл, то тогава може да се образува кръг, в противен случай не.
- Един насочен граф има Ойлеров цикъл, ако и само ако:
 1. Всеки връх има $in-degree == out-degree$ и
 2. Всички негови неизолирани върхове принадлежат към един-единствен силно свързан компонент.

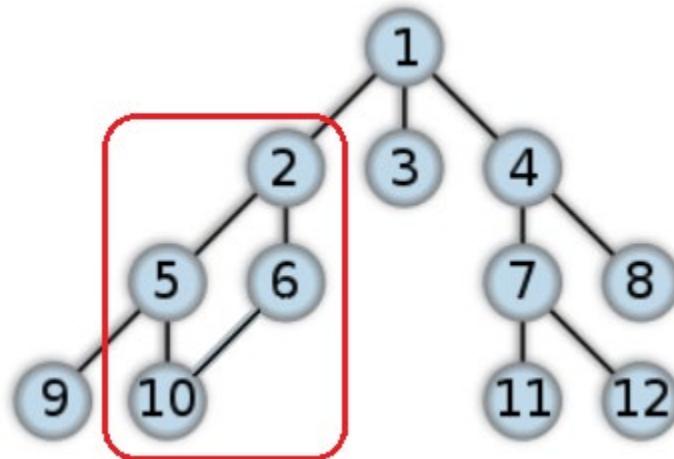
Задача 3

- Даден е свързан ненасочен граф и връх в него. Създайте насочен граф, така че всеки път да води до посочения връх.



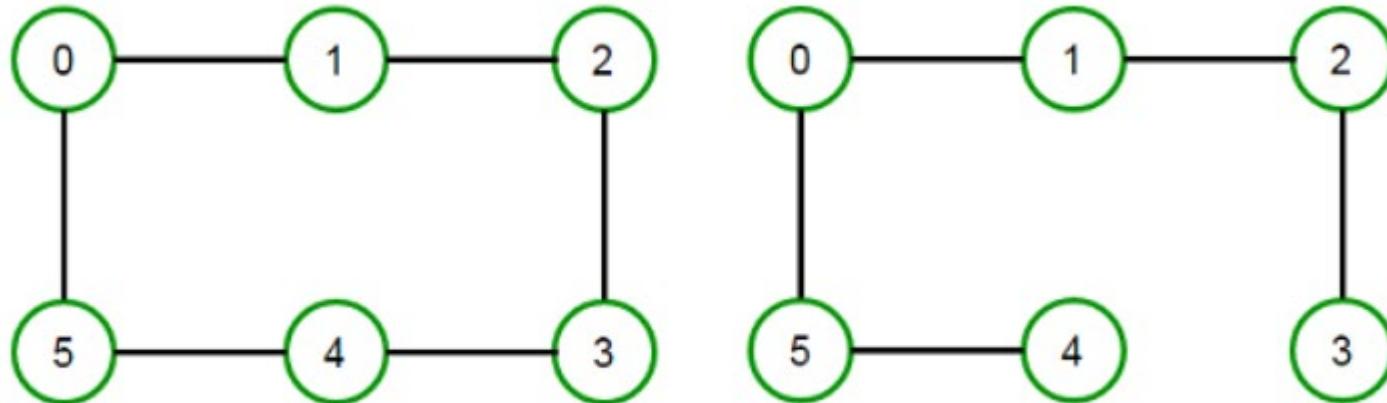
Задача 4

- Даден е свързан неориентиран график. Напишете програма, която проверява дали даденият граф съдържа цикъл.



Задачи 5

- Даден е неориентиран граф. Напишете програма, която проверява дали даденият граф е дърво или не. С други думи, проверете дали даден неориентиран граф е ацикличен свързан график или не.



CSCB034 Упражнения по алгоритми и програмиране

ГРАФИЧАСТ 2

гл. ас. д-р Слав Ангелов, НБУ

Загрявка

Каква е разликата между Хамилтонов и Ойлеров цикъл?

Отговор

Ойлеров цикъл – искахме да минем по всички мостове по веднъж, т.е. обхождаме ребрата.

Хамилтонов цикъл – искахме да минем през всеки връх по веднъж, т.е. обхождаме върховете.

Загрявка

Напишете функция `void Mmult(int A[][], int B[][], int C[][])`, която да умножава с матрично умножение две матрици A и B, като използва правилото “ред по стълб”.

Забележете, че ако матрицата A има n реда и k стълба, т.е. $n \times k$, а матрицата B е $k \times m$, то $C = A * B$ е $n \times m$.

Реализация

```
//...  
int A[n][k]; int B[k][m]; int C[n][m];  
for (int i = 0; i < n; i++){  
    for (int j = 0; j < m; j++){  
        C[i][j] = 0;  
        for (int q = 0; q < k; q++){  
            C[i][j] += A[i][q] * B[q][j];  
        }  
    }  
}  
//...
```

Някои факти за матричното умножение

Матричното умножение $A * B$ е изпълнима операция, когато броят на стълбовете на A съвпада с броя на редовете на B .

Забележка: В общия случай $A * B \neq B * A$. Това важи и за квадратни матрици (проверете с контрапример).

Размисли

Ако A и B са две квадратни матрици $n \times n$, колко е изчислителната сложност на правилото за умножение „ред по стълб“? (за простота нека преброим умноженията на числа)

Допълнително

Обикновено умножението на матрици не се имплементира директно чрез правилото „ред по стълб“. Една от причините е, че, ако числата са с плаваща запетая, то на всяко умножение ще имаме грешка, а ние имаме много умножения , за да получим всеки един елемент. Това води до голяма грешка от закръгляне.

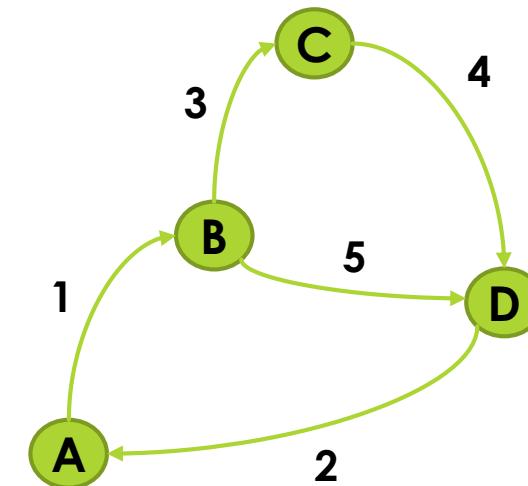
Основен метод за справяне с този проблем е singular value decomposition. При него всяка от матриците се разпада на няколко матрици с добри качества, което спестява умножения.

Удобен линк: https://en.wikipedia.org/wiki/Singular_value_decomposition

Матрица на инцидентност за ориентиран граф

Отразява наличните връзки между върхове и ребра:

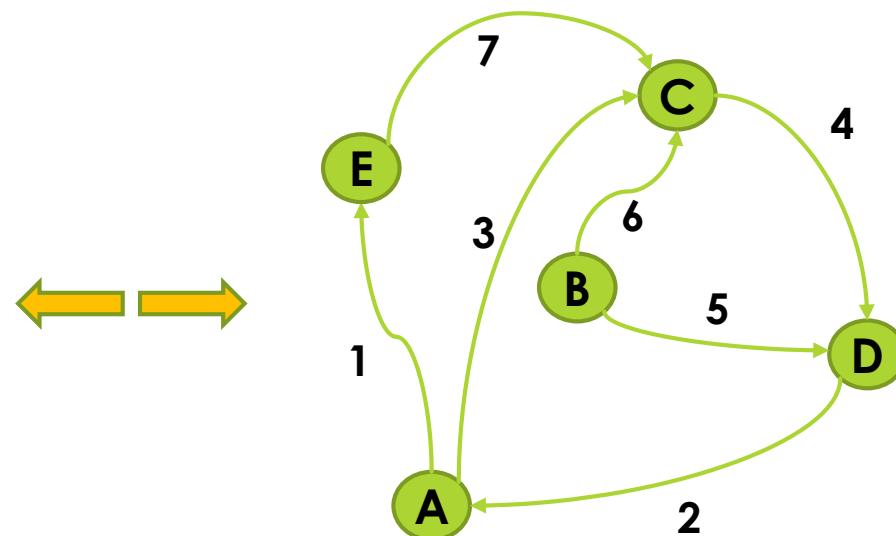
	1	2	3	4	5
A	-1	1	0	0	0
B	1	0	-1	0	-1
C	0	0	1	-1	0
D	0	-1	0	1	1



Матрица на инцидентност за ориентиран граф - упражнение

Попълнете матрицата на инцидентност:

	1	2	3	4	5	6	7
A							
B							
C							
D							
E							



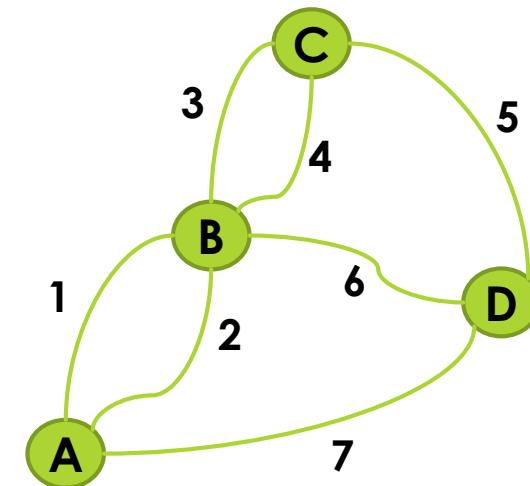
Размисли

- ▶ **Може ли да дефинираме матрица на инцидентност за претеглен ориентиран граф?**
- ▶ **Какво става с матрицата на инцидентност при неориентиран граф?**
- ▶ **Може ли да създадем матрица на инцидентност за неориентиран мултиграф?**
- ▶ **А за ориентиран мултиграф?**
- ▶ **А за неориентиран претеглен мултиграф?**

Матрица на съседства за неориентиран мутиграф

Отразява наличните връзки между върховете:

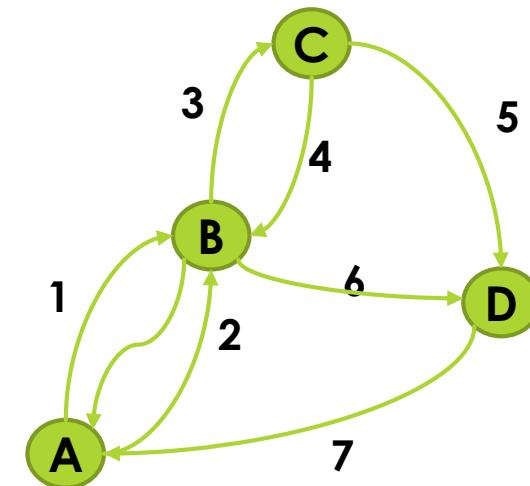
	A	B	C	D
A	0	2	0	1
B	2	0	2	1
C	0	2	0	1
D	1	1	1	0



Матрица на съседства за ориентиран мутиграф

Отразява наличните връзки между върховете:

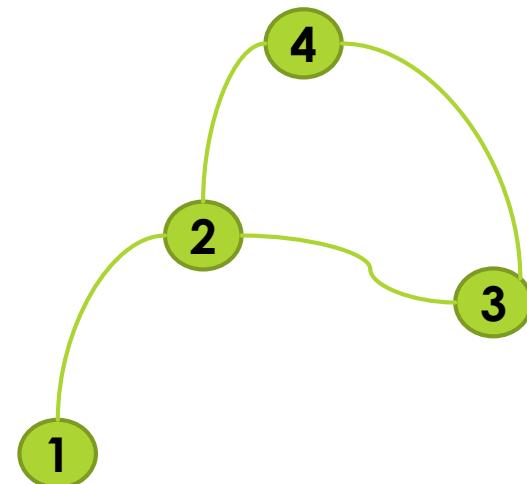
	A	B	C	D
A	0	2	0	0
B	1	0	1	1
C	0	1	0	1
D	1	0	0	0



Списъци на съседите за неориентиран граф

Отразява съседните върхове на всеки връх (спестява памет):

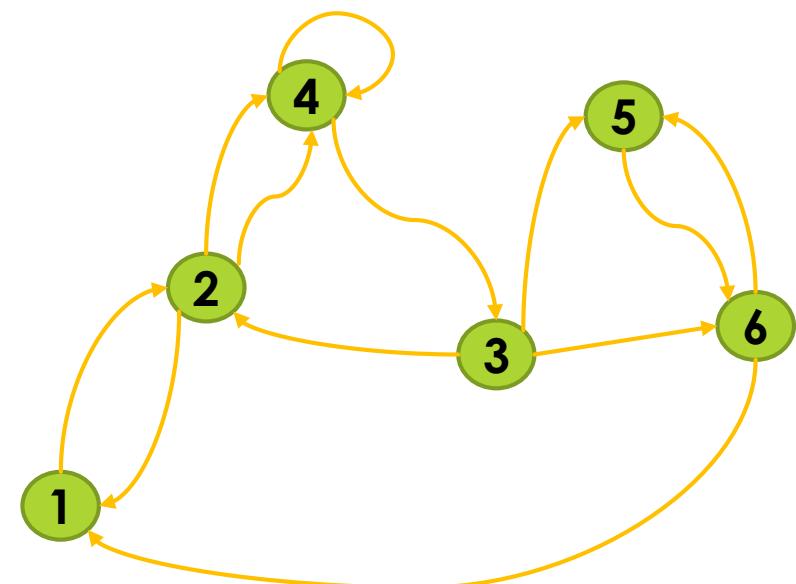
	0	1	2	3	4
0					
1	1	2	0	0	0
2	2	3	4	0	0
3	2	2	4	0	0
4	2	2	3	0	0



Задача

Намерете матрицата на съседство за:

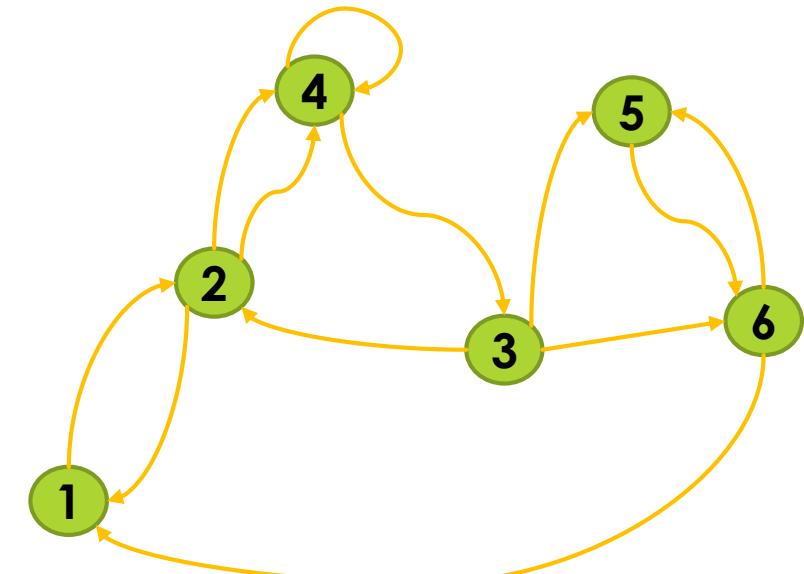
	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						



Решение

Намерете матрицата на съседство за:

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



Свързан граф - размисли

Свързан граф е граф, където между всеки два върха съществува път.

Ако разполагаме с матрицата на съседство, можем ли да определим дали един ориентиран мултиграф е свързан без да го обхождаме?

Подсказка

Теорема:

Нека A е матрицата на съседство на ориентиран мултиграф. Нека я повдигнем на степен k , където k е по-малко от броя на върховете, т.е. $A^k = \{a_{i,j}^{(k)}\}$. Тогава $a_{i,j}^{(k)}$ е броят на пътищата между i -ти и j -ти връх с дължина k .

Лема:

Ако имаме свързан граф с n върха, то винаги може да намерим път с дължина не повече от $n-1$, който е без цикъл.

Възможно решение

Разполагаме с матрицата на съседства A , която е $n \times n$.

- 1) Ако стойностите на A са различни от 0 (с изключение на главния диагонал, защо?), то край.
- 2) Вдигаме A на квадрат;
- 3) Събираме A с A^2 , ако стойностите на получената матрица са различни от нула, това са всички пътища от връх до връх с дължина 1 и 2, то край (отново не гледаме главния диагонал);
- 4) Продължаваме процедурите докато не получил само ненулеви елементи или не получим и матрицата A^{n-1} .
- 5) Сумираме $A + A^2 + A^3 + \dots + A^{n-1}$, ако и тогава има нулеви елементи извън главния диагонал означава, че има върхове, които не могат да бъдат достигнати. Край.

Планарни графи - размисли

Един граф е планарен, ако може да се разположи в равнината без да има пресичания на негови ребра.

На база на досегашните ни знания по графи можем ли да измислим начин за установяване на планарност на граф?

Решение

От теоремата за четирите цвята знаем, че можем да оцветим всяка карта в четири цвята, така, че еднакви цветове да не са един до друг. Всяка карта е планарен граф. Тогава един граф е планарен, ако може да се оцвети в четири цвята без два съседни върха да са в един и същи цвят.

Графите лесно може да ги представим в двумерен масив чрез списък на ребрата, а ако ще ги оцветяваме каква структура да използваме?

Обхождане на граф в ширина

Върховете ги разпределяме в нива по следната схема:

- 1) Тръгваме от избран връх, това ни е нулевото ниво;
- 2) Съседите на този връх са Ниво 1;
- 3) Непосетените съседи на върховете от Ниво 1 са Ниво 2;
- 4) Продължаваме схемата докато не изчерпим върховете.

Повече информация може да видите от слайдовете в Мудъл или на:

<http://vista-2008.com/wp-content/uploads/custom/Grafi/obhojdane%20v%206irina.html>

Възможна реализация (1/2)

```
#include <iostream>
using namespace std;
#include<conio.h>
#include<stdlib.h>

int cost[10][10],i,j,k,n,qu[10];
int Front,rare,v,visit[10],visited[10];
int main () {
    int m;
    cout << "Number of vertices: ";
    cin >> n;
    int main () { \\
        cout <<"Number of edges: ";
        cin >> m;
        cout <<"\nEDGES \n";
        for(k=1;k<=m;k++){
            cin >>i>>j;
            cost[i][j]=1;
        }
    \\
}
```

Възможна реализация (2/2)

```
int main () { \\ ...
    cout <<"Enter initial vertex: ";
    cin >> v;
    cout <<"Visited vertices \n";
    cout << v;
    visited[v]=1;
    k=1;
    // ...
    while(k<n){
        for(j=1;j<=n;j++)
            if(cost[v][j]!=0 && visited[j]!=1 &&
               visit[j]!=1){
                visit[j]=1;
                qu[rare++]=j;
            }
        v=qu[Front++];
        cout<<v << " ";
        k++;
        visit[v]=0; visited[v]=1;
    }
    return 0;
}
```

Упражнения

1. Тествайте кода с различни свързани и несвързани графи.
2. Преработете реализациите така, че да използва структурата „опашка“ чрез библиотека `<queue>`, ако сметнете за удачно, може да използвате и библиотека `<vector>`.
3. Каква е сложността на BFS.

CSCB034 Упражнения по алгоритми и програмиране

ГРАФИЧАСТ 3

гл. ас. д-р Слав Ангелов, НБУ

Покриващо дърво (Spanning tree)

Дърво: свързан граф без цикли.

Кореново дърво (дърво с корен): Всеки връх е кореново дърво. Ако към дадено кореново дърво добавим връх с ребро, които не са били в него, то отново ще получим кореново дърво.

Покриващо дърво е дърво с корен избран връх в даден свързан граф G , което съдържа всички върхове на G .

Максимално покриващо дърво/гора на граф G ще наричаме разбиване на върховете на G в непресичащи се множества V_i , сформиращи подграфи $G_i(V_i, A_i)$, такива, че съществува покриващо дърво G^t_i на графа G_i , в което участват всичките върхове от V_i .

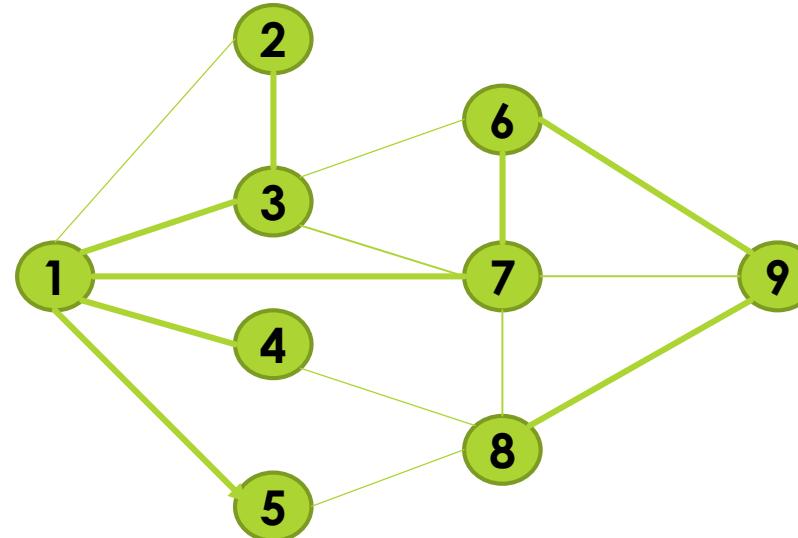
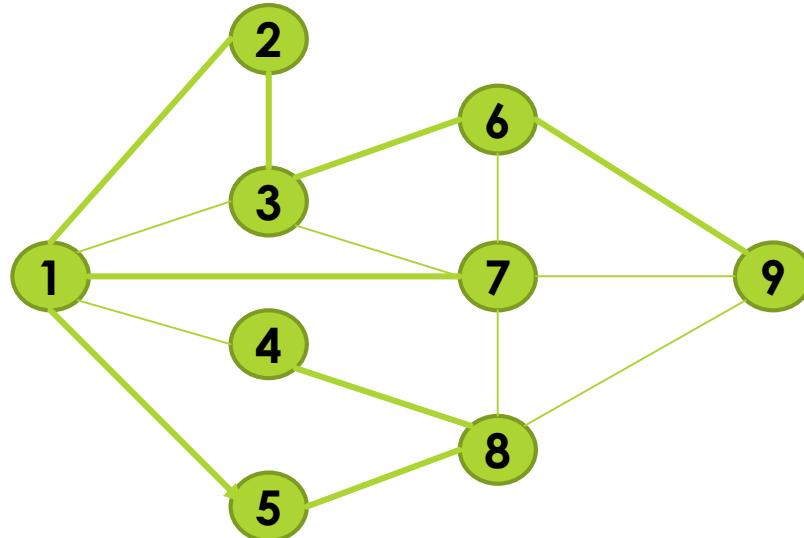
Твърдение: Максимална гора на произволен граф G с r свързани компоненти има $n - r$ ребра, където n е броят на върховете в G .

Размисли

Уникално ли е покриващото дърво в даден граф?

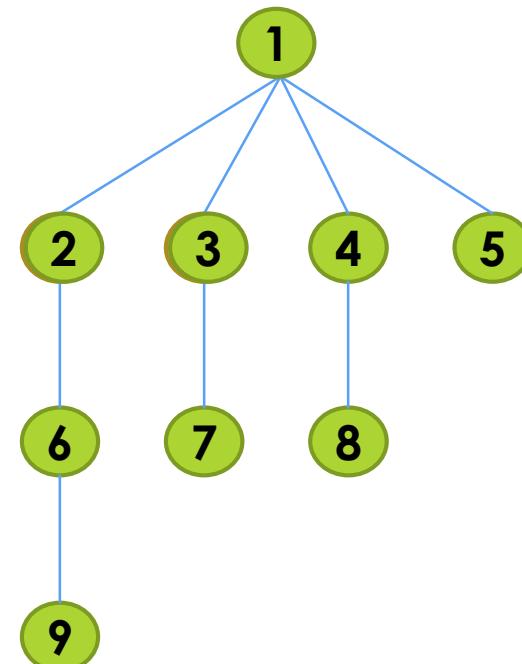
Размисли

Уникално ли е покриващото дърво в даден граф?



Представяне на дърво чрез списък на бащите

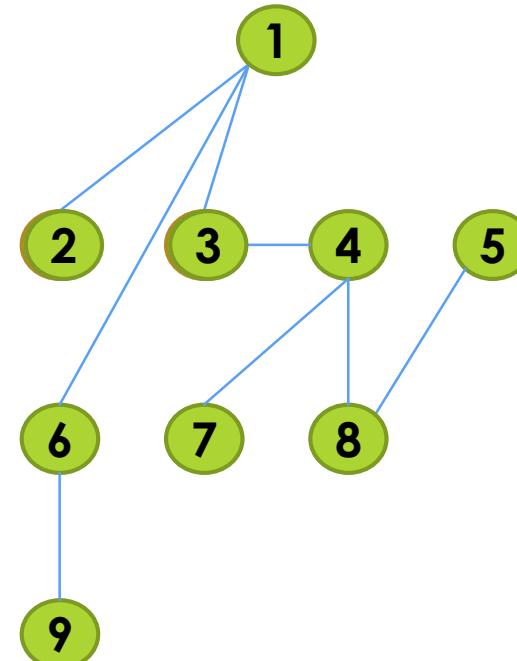
i	P[i]
1	0
2	1
3	1
4	1
5	1
6	2
7	3
8	4
9	6



Представяне на дърво чрез списък на бащите - упражнение

Popълнете списъка на бащите:

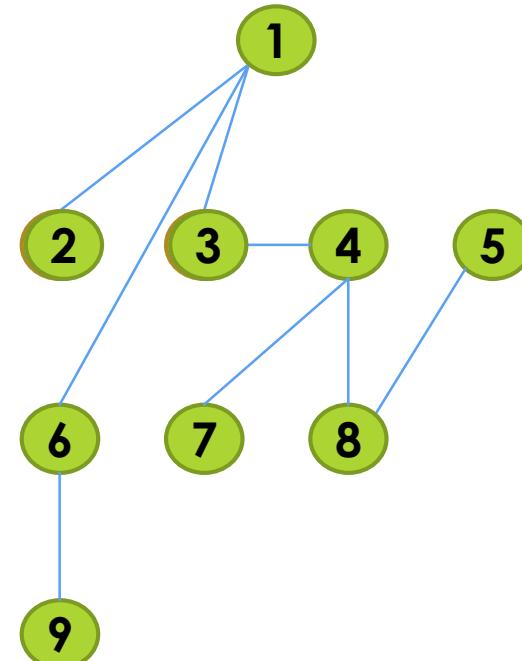
i	P[i]
1	
2	
3	
4	
5	
6	
7	
8	
9	



Представяне на дърво чрез списък на бащите - упражнение

Popълнете списъка на бащите:

i	P[i]
1	0
2	1
3	1
4	3
5	8
6	1
7	4
8	4
9	6



Размисли

Ако имаме граф с n върха и m ребра, то колко трябва да е дълъг списъка на бащите?

Размисли

Ако имаме граф с n върха и m ребра, то колко трябва да е дълъг списъка на бащите?

Колкото върхове имаме.

Упражнение

Ако разполагаме със списъците на съседите на граф и резултата по нива след BFS, можем ли да построим покриващо дърво ? (за простота графът е свъзан)

Входът е представен с:

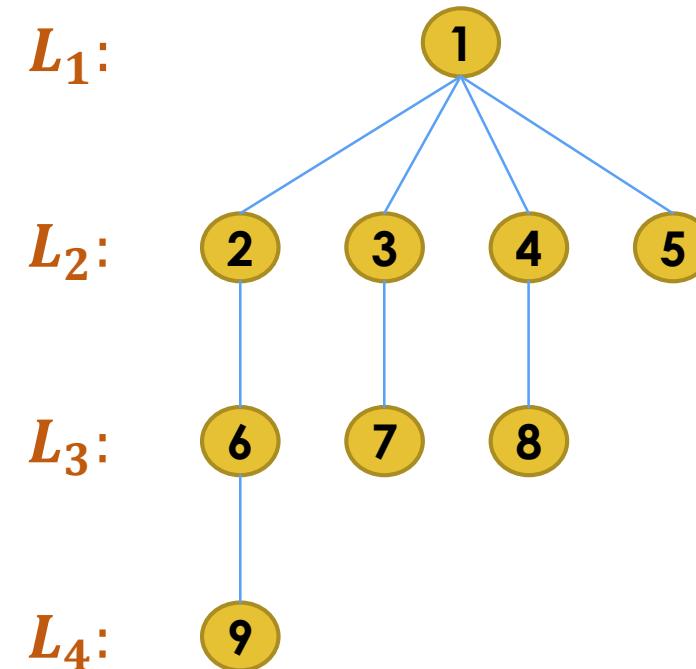
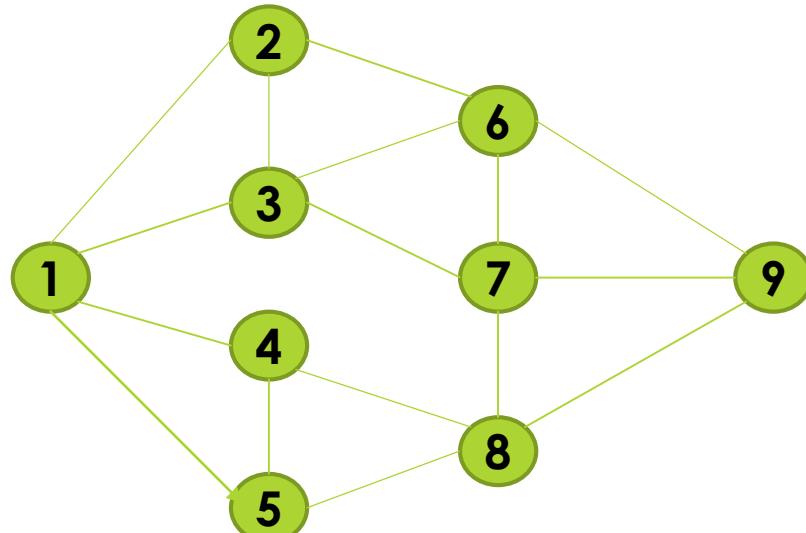
Списъци на съседите – масив от вектори;

Резултат от BFS - масив от вектори, дължината на масива е броят на нивата.

Изход:

Покриващо дърво, представено със масив(списък) на бащите.

Алгоритъм за покриващо дърво чрез „обхождане в ширина“ - онагледяване



Упражнение - решение

Ако разполагаме със списъците на съседите на граф и резултата по нива след BFS, можем ли да построим покриващо дърво ?

1. Започваме от върха в Ниво 1;
2. Съседите му са децата, т.е. върховете от Ниво 2;
3. Търсим децата на върховете от Ниво 2 в Ниво 3 чрез списъците на съседите като внимаваме всеки връх от Ниво 3 да е дете само на един връх от Ниво 2;
4. Продължаваме процедурата докато не изчерпим нивата.

Къде съхраняваме новополученото покриващо дърво?

Дърветата и някои специални графи

1. **Максимално ацикличен граф** (каквото и ребро да добавим към него ще получим цикъл);
2. **Минимално свързан граф** (което и ребро да отстраним, графът вече няма да е свързан);
3. Граф, в който между всеки 2 върха съществува единствен път.

Каква е връзката между изброените специални графи и дърветата в граф ?

Дървета и цикли

Твърдение: Нека имаме свързан граф G . Нека G^t е негово покриващо дърво. Тогава, ако добавим точно едно ребро от G в G^t , различно от тези в G^t , ще получим граф с точно един цикъл.

А какво ще стане, ако добавим две ребра към G^t ?

Обхождане на граф в дълбочина

Върховете ги разпределяме в нива по следната схема:

- 1) Тръгваме от избран връх, който поставяме в стека и маркираме за обходен;
- 2) Поставяме съсед на първия връх в стека и той вече е обходен;
- 3) Поставяме съсед на върха от Стъпка 2 в стека, обходен е;
- 4) Продължаваме схемата докато стигнем до съсед, който няма необходими съседи;
- 5) Премахваме от стека върховете, които нямат необходими съседи и се връщаме на Стъпка 2.

Повече информация може да видите от слайдовете в Мудъл, а за визуализация на:

<https://www.cs.usfca.edu/~galles/visualization/DFS.html>

Възможна реализация (1/2)

```
#include <iostream>
using namespace std;
#include <cstdio>
#include <vector>
#include <algorithm>
int n,v,used[101];
vector<int>a[101];
void dfs(int k)
{
    printf("%d ", k);
    for(int i=0; i<a[k].size(); i++){
        if(used[a[k][i]]) continue;
        used[a[k][i]]=1;
        dfs(a[k][i]);
    }
}
```

Възможна реализация (2/2)

```
int main () { \\ ...
    scanf("%d %d",&n,&v); /* Брой
върхове на графа и връх, от който да
обхождаме*/
    int b,c;
    while(b!=0 || c!=0){ /* Спираме
въвеждането на ребра при две нули*/
        scanf("%d %d",&b,&c);
        a[b].push_back(c);
        a[c].push_back(b);
    }
    for (int i=0; i<n; i++)
        sort(a[i].begin(),a[i].end());
    used[v]=1;
    dfs(v);
    return 0;
}
```

Упражнения по имплементацията

- ▶ Тествайте кода с различни свързани графи.
- ▶ Тази имплементация използва списъците на съседите, списъка на ребрата или матрицата на съседство ?
- ▶ За какво е използвана библиотеката `<cstdio>` ?
- ▶ А библиотеката `<algorithm>` ?
- ▶ Какво всъщност прави сегментът от кода, където се използва функцията `sort()` ? Можем ли да премахнем този сегмент ?
- ▶ Можем ли така да реализираме имплементацията, че да пропуснем декларирането на глобалните променливи, или поне да дефинираме точен размер на графа (а не фиксиран 101) ?
- ▶ Каква е сложността на DFS ?

Покриващо дърво при DFS

Можем ли да построим покриващо дърво с търсене в дълбочина? Дайте идеи за алгоритъм.

Рекурсивен алгоритъм за ПД с „обхождане в дълбочина“

```
int U[MAXN+1], P[MAXN], G[MAXN][MAXN], N;  
  
void DFS_R(int r){  
    int y;  
    while(G[r][0]>0){  
        y=G[r][G[r][0]--];  
        if(!U[y]) {U[y]=1; P[y]=r; DFS_R(y);} }  
    }  
  
void first_call(){    int i;  
    for(i=1;i<=N;i++) U[i]=0;  
    for(i=1;i<=N;i++) if(U[i]==0){ U[i]=1; P[i]=0; DFS_R(i);} }  
}
```

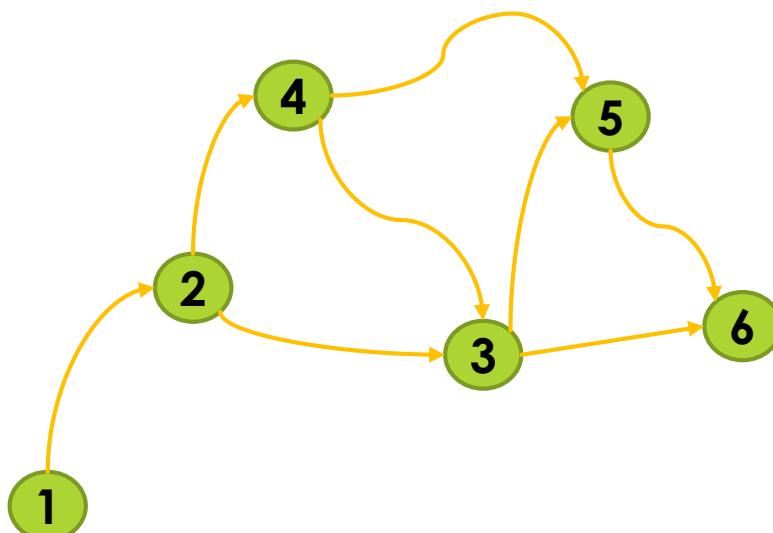
Рекурсивен алгоритъм за ПД с ОД със запазване на броячите

```
int U[MAXN+1], P[MAXN], G[MAXN][MAXN], N;  
  
void DFS_R(int r){ int y;  
    while(G[0][r]>0){  
        y=G[r][G[0][r]--];  
        if(!U[y]) {U[y]=1; P[y]=r; DFS_R(y);} }  
}  
  
void first_call() {int I;  
    for(i=1;i<=N;i++) {U[i]=0; G[0][i]=G[i][0];}  
    for(i=1;i<=N;i++) if(U[i]==0) {U[i]=1; P[i]=0; DFS_R(i); } }
```

Ориентиран ацикличен граф

Определение: Директно от името може да изведем, че това е ориентиран граф без цикли.

Топологическо сортиране: Върховете на един ориентиран ацикличен граф могат да се подредят в редица така, че върх v да е преди върх d , ако съществува път от v до d и върх v ни е по-близо до отправния ни върх.



Топологично сортиране - пример

Теорема: При обхождане в дълбочина върховете напускат стека в ред обратен на едно топологическо сортиране.

Пример: Разглеждаме графа от предния слайд. Възможно обхождане в дълбочина е:

1, 2, 3, 5, 6, после изпразваме стека до 2 и имаме 1, 2, 4. После премахваме всички елементи от стека, защото нямаме необходими върхове.

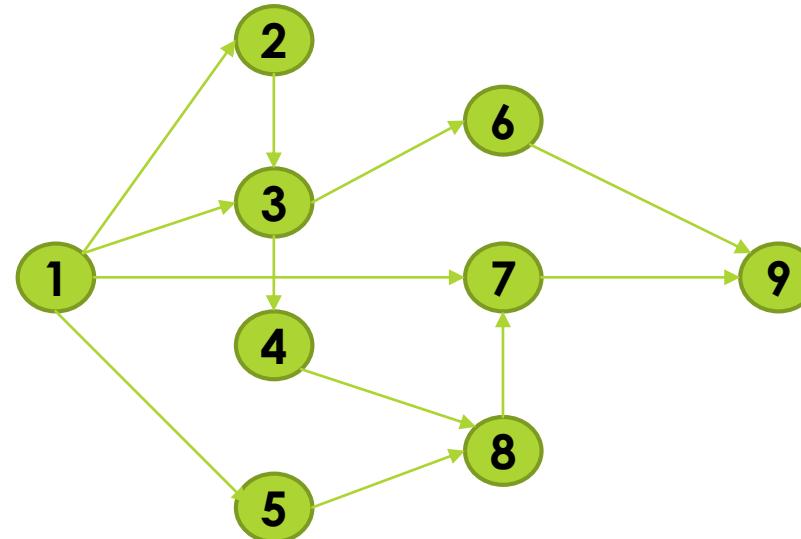
Последователността на премахване на върховете от стека е 6, 5, 3, 4, 2, 1. Това означава, че обратната на тази подредба е топологически сортирана, т.е. 1, 2, 4, 3, 5, 6.

Топологично сортиране - размисли

Единствено ли е топологичното сортиране в общия случай на ориентиран ацикличен граф ? Дайте контрапример.

Задача

Реализирайте на C++ топологическо сортиране с DFS. Използвайте за тестов пример следния граф:



Топологическо сортиране чрез „обхождане в дълбочина“

```
int U[MAXN], S[MAXN], G[MAXN][MAXN], t;  
int N, M;  
void DFS_RT(int r){ int y;  
    while(G[r][0]>0)  
    { y=G[r][G[r][0]--];  
        if(!U[y]) {U[y]=1; DFS_RT(y); }  
    } T[t--]=r;  
}  
void first_call_RT()  
{ int i; for(i=1;i<=N;i++) U[i]=0; t=N-1;  
    for(i=1;i<=N;i++) if(U[i]==0) { U[i]=1; DFS_RT(i); } }
```

Топологическо сортиране чрез „обхождане в дълбочина“

```
main(){ int x,y,i;
    scanf("%d %d",&N,&M);
    for(i=1;i<=N;i++) G[i][0]=0;
    for(i=1;i<=M;i++)
    {   scanf("%d %d",&x,&y);
        G[x][++G[x][0]]=y;
        // в ориентирания няма ребро (y,x)
    }
    first_call();
    for(i=0;i<N;i++) printf("%d ",T[i]);
    printf("\n"); }
```

Задачи

1. Променете рекурсивната имплементация на DFS така, че да отчита момента на напускане на стека.
2. Напишете главна програма, която да въвежда в представянето със списъци на децата, след което сортира топологически дага.
3. Тествайте програмата с примера от лекцията.

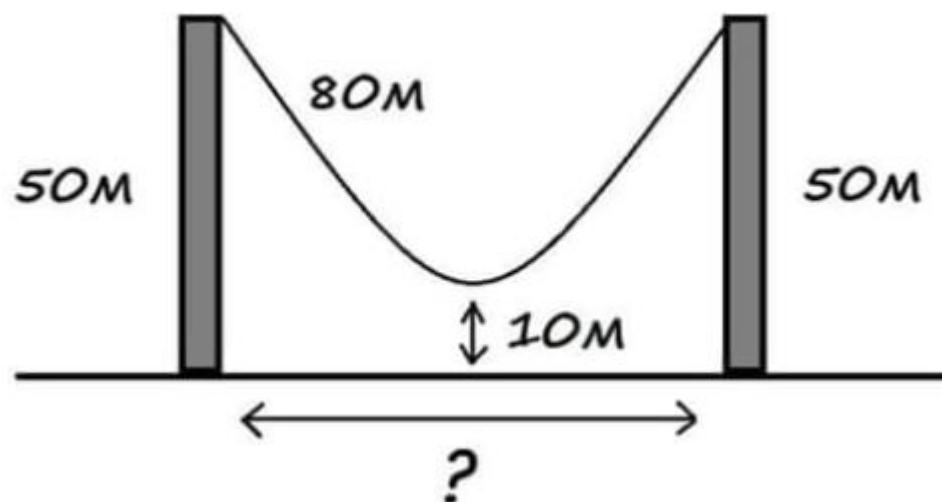
CSCB034 Упражнения по алгоритми и програмиране

ГРАФИ С ЦЕНИ НА РЕБРАТА

гл. ас. д-р Слав Ангелов, НБУ

Загръвка

Можете ли да решите загадката, която Амазон дава на интервю за работа?



Как се задава стандартно вход за граф – важно за Контролно 2

Вход:

- Брой върхове** (допускаме, че върховете са номериране от 1 до техният брой);
- Брой ребра** (не е задължително, може например да прекратяваме добавянето на ребра след въвеждане на 0);
- Изброяваме ребрата**, като всяко задаваме с два върха.

Допълнително във входа може да се изиска (зависи от задачата) да се въведе начален върх или друг параметър.

Интерпретиране на запис

Можете ли да разчетете следния запис:

$$G(V, E), \quad c: E \rightarrow R^-.$$

Отговор

Така означаваме граф с отрицателни цени на ребрата.

Въпроси

Като използвате презентацията от лекцията отговорете на следните въпроси:

1. Има ли смислова разлика в следните термини – цени на ребрата, тегла на ребрата, дължини на ребрата, капацитети на ребрата, ширини на ребрата, стойности на ребрата?
2. Какво е “дърво на най-късите пътища” (ДНКП) от връх v до всички останали върхове?
3. Можем ли директно да използваме ДНКП с корен връх v , за да намерим най-късите пътища от друг връх в конкретния граф?

Въпроси 2

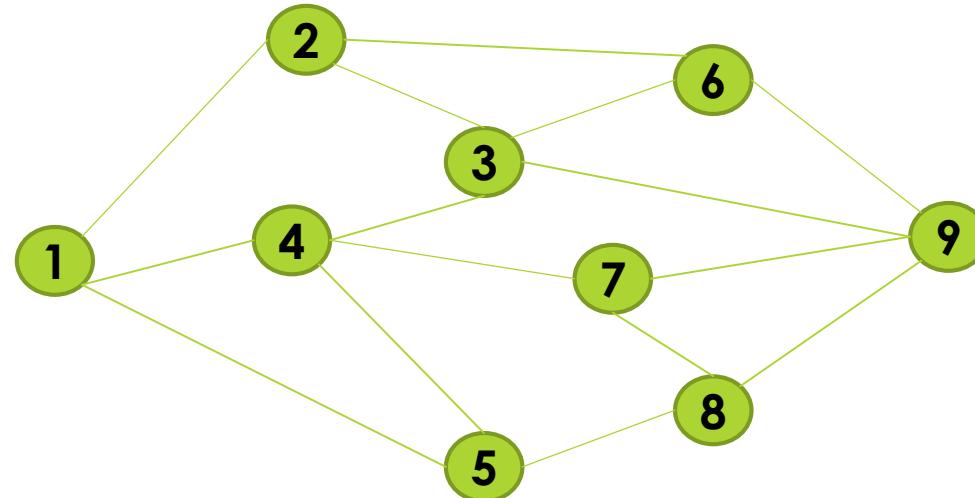
Като използвате презентацията от лекцията, отговорете на следните въпроси:

1. Ако всички ребра са с фиксирана еднаква дължина, обяснете за себе си защо покриващо дърво с корен връх v , образувано с търсене в ширина, е ДНКП;
2. Можем ли да решим със същата сложност задача за „най-дълъг път“ ?

Задача

Процесите в дадена компания са свързани помежду си, като образуват графовидна структура. Напишете алгоритъм, който по графа на връзките между процесите да намира колко най-малко процеси трябва да се преминат, за да се стигне от процес x в процес y.

Примерен граф на връзките между процесите:

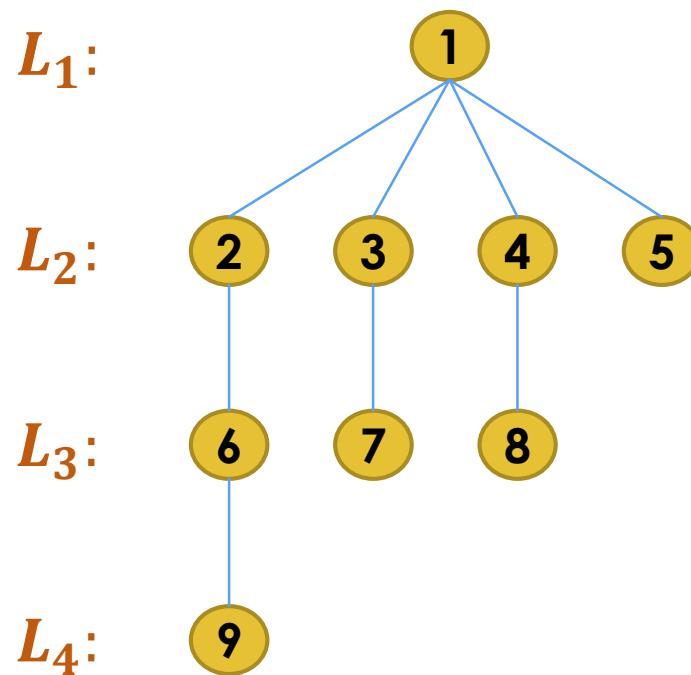
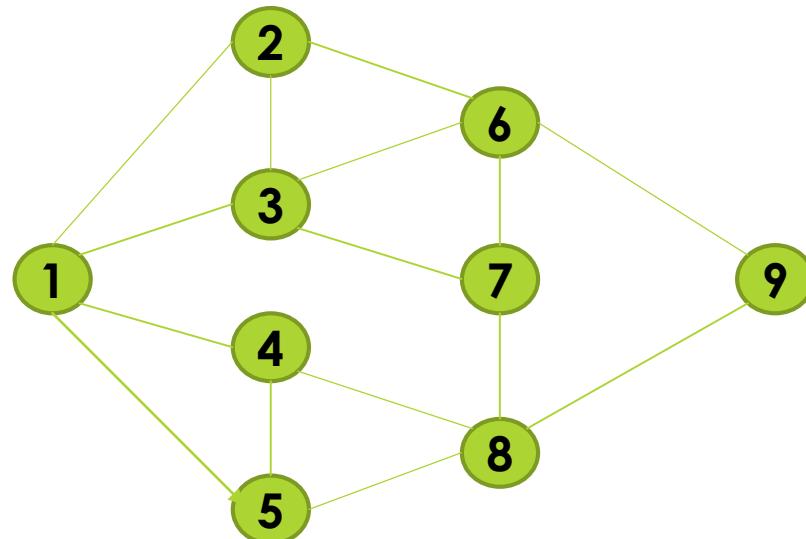


Задача – идея за решение

- ▶ Реализираме търсене в ширина, като започнем от процесът, който ни интересува. Това ще генерира покриващо дърво с този връх;
- ▶ Разглеждаме полученото покриващо дърво. Понеже връзките нямат тегла, то ние буквально им присвояваме тегло 1, т.е. през колко процеса преминаваме еквивалентно на през колко ребра преминаваме минус 1 от корена до конкретно избран връх.

Размисли

Имаме граф с равни тегла. Ако сме построили едно покриващо дърво чрез търсене в ширина, то може ли само чрез това ПД да намерим най-късите пътища от произволен връх до произволен връх?



Алгоритъм на Дейкстра

Предназначение: Строи ДНКП от даден връх на граф с тегла.

Алгоритъм:

1. Всички върхове са именувани от **1** до **n**; ребрата са представени със списъци на съседите и са **m** на брой; теглата са записани в отделен двумерен масив **c[i][j]**, там където няма ребро теглото е много голямо число **INF**; подготвен е и списък на бащите **p[i]**; **d[i]** в този масив пазим най-късите разстояния от връх до корена; масив за отбелязване на посетени върхове **U[i]**;
2. Започваме с едно „грубо дърво“, където всички върхове са свързани с стартовия такъв;
3. Постепенно „релаксираме върховете“, там където е възможен по-къс път.

Визуализация: <https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html>

Имплементация на Дейкстра

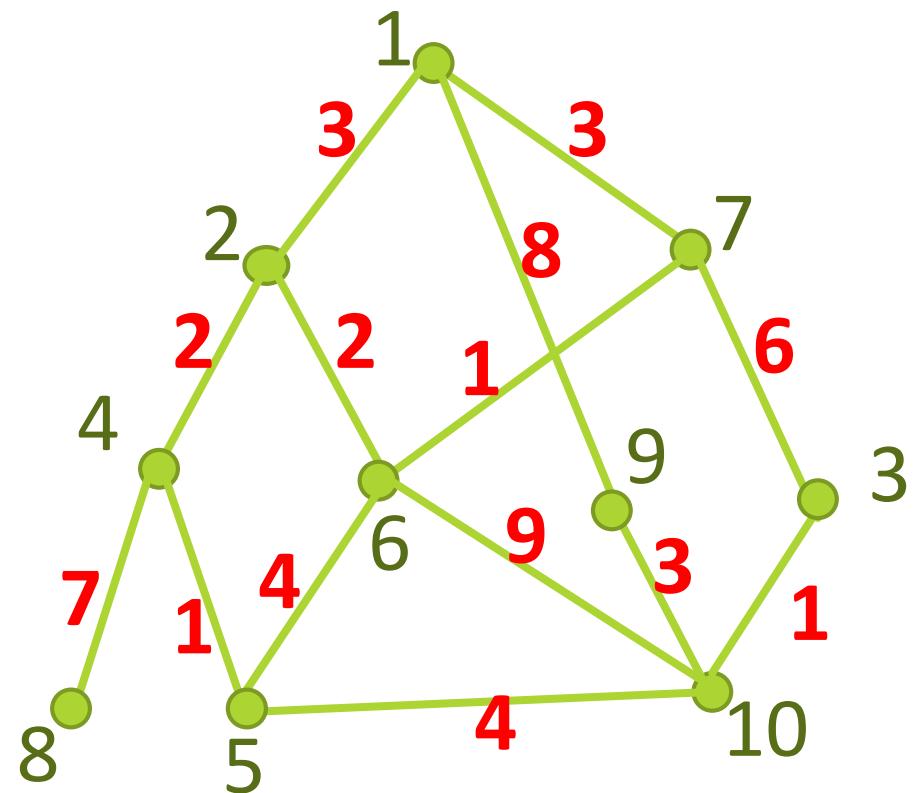
```
void dijkstra(int r) {  
    int v, w, min, i, j;  
    for(i=1; i<=n; i++) {U[i]=0; p[i]=r; d[i]=INF;}  
    d[r]=p[r]=0;  
    for(i=1; i<=n; i++) {  
        v=0; min=INF;  
        for(j=1; j<=n; j++) if(U[j]==0 && d[j]<min) { v=j; min=d[j]; }  
        U[v]=1;  
        for(j=1; j<=G[v][0]; j++) { w=G[v][j];  
            if(U[w]== 0 && d[v]+c[v][w]<d[w]) { d[w]=d[v]+c[v][w]; p[w]=v; }  
        } } }
```

Декларираме на глобално ниво:

```
#define INF 1000000000  
int G[MAX][MAX], c[MAX][MAX],  
U[MAX], p[MAX], d[MAX], n, m, r;
```

За какво ни е?

Пример (проф. Манев)



10	14	14
1	2	3
1	7	3
1	9	8
2	4	2
2	6	2
7	6	1
7	3	6
4	8	7
4	5	1
6	5	4
6	10	9
9	10	3
3	10	1
5	10	4

Какво е написано?

Пример: масивът d[] на всяка фаза

Пример: масивът d[] на всяка фаза

Всеки ред е състоянието на d[] в конкретната фаза. Имаме 10 реда, толкова фази, колкото и върхове

U	V:	2	3	4	5	6	7	8	9	10
1	3	∞	∞	∞	∞	∞	3	∞	8	∞
2		∞	5	∞	5	3	∞	8	∞	
7		9	5	∞	4		∞	8	∞	
6		9	5	8			∞	8	13	
4		9		6			12	8	13	
5		9					12	8	10	
9		9					12		10	
3							12		10	
10							12			
8										

В зелено са окончательните най-къси пътища от корена до конкретните върхове

Пример: масивът р[] на всяка фаза

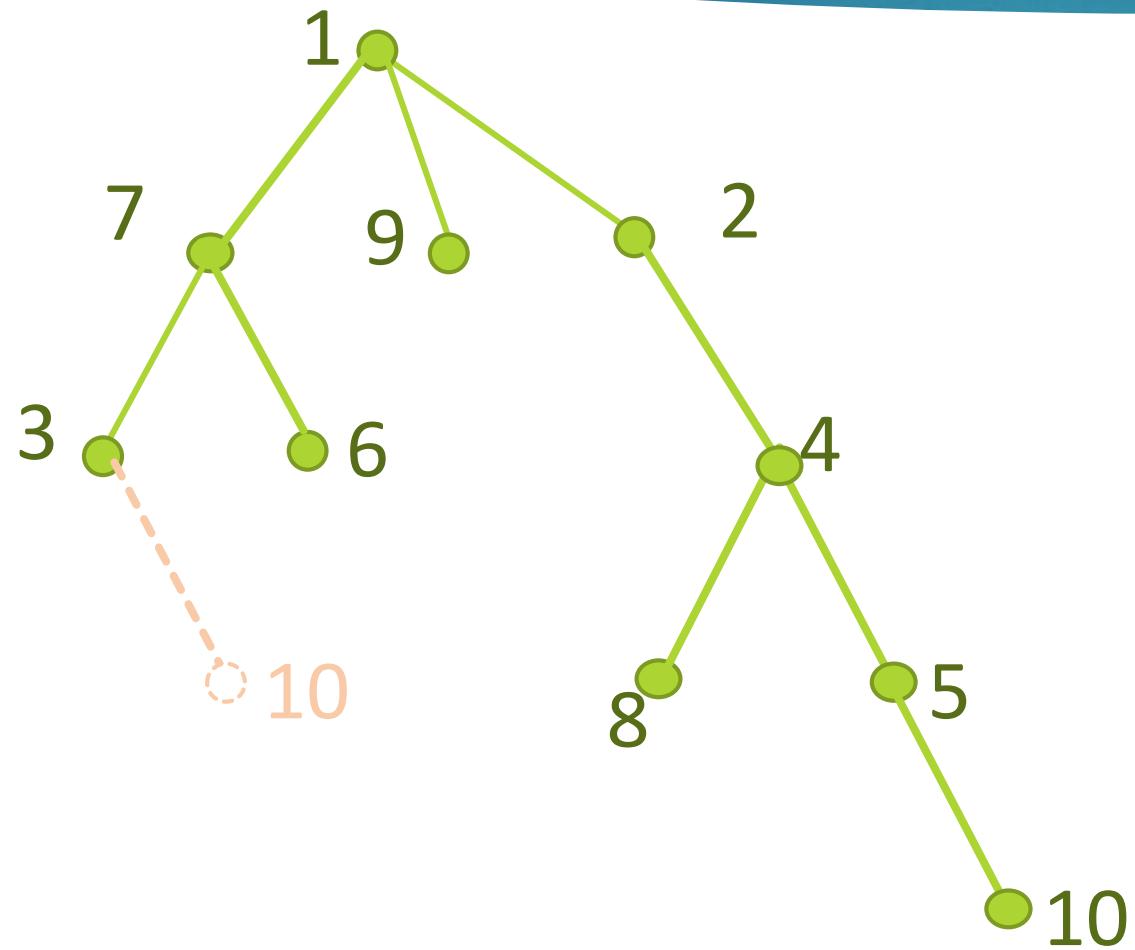
Пример: масивът p[] на всяка фаза

Всеки ред е състоянието на p[] през конкретната фаза

u	v:	2	3	4	5	6	7	8	9	10
1		1	1	1	1	1	1	1	1	1
2			1	2	1	2	1	1	1	1
7		7	2	1	7		1	1	1	
6		7	2	6			1	1	6	
4		7		4			4	1	6	
5		7					4	1	5	
9		7					4		5	
3							4		5,3	
10							4			
8										

В зелено са окончательните башти

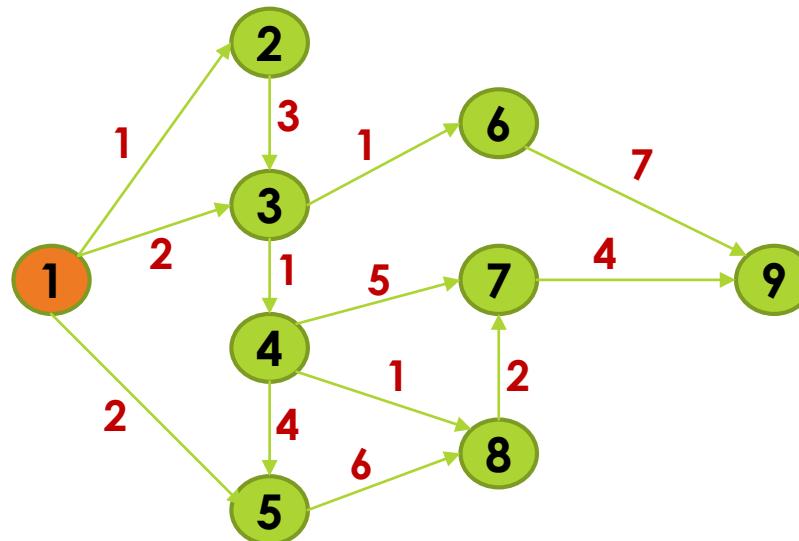
Пример: ДНКП



Възможни са
две решения

Задача

Експериментирайте с кода на Дейкстра. Код за `main()` функцията може да вземете от презентацията „Най-къс път в граф“ на проф. Манев (използвайте библиотеката `<cstdio>`, така че да може да го стартирате на C++). Тествайте върху следния граф (нека началото е Връх 1):

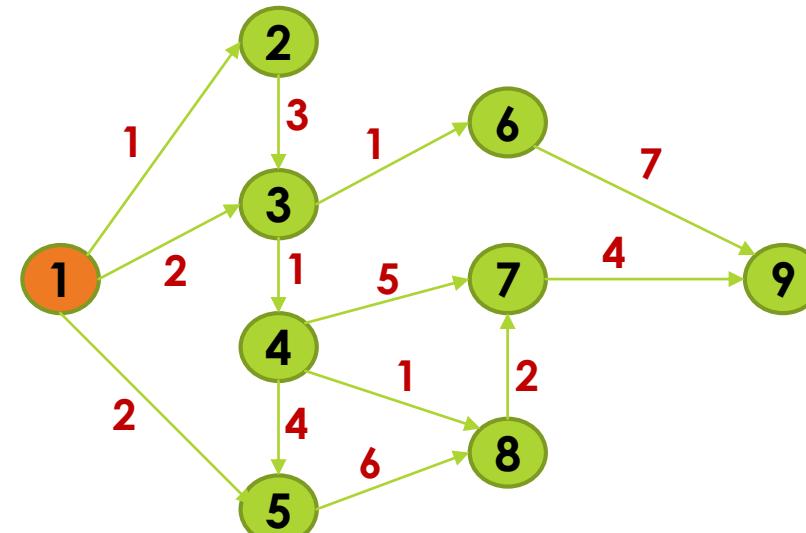


Задача – разпишете фазите за d[]

d[]

1	2	3	4	5	6	7	8	9	10

p[]



Размисли

Ще работи ли представената имплементация на Дейкстра за мултиграфи?

Размисли

- ▶ **Може ли да запишем теглата на ребрата в граф във матрицата на съседите, списъците на съседите или матрицата на инцидентност?**
- ▶ **Как да се съхраняват стойностите на ребрата в мултиграф?**

Задача

Използвате реализациите на Дейкстра от слайдовете в презентацията „Най-къс път в граф“. Пренапишете кода така, че да използва библиотеката `<vector>` и да е на C++.

Съвет: За векторното представяне на списъците на съседите и начина на въвеждане на графа използвайте векторната реализация от предходната лекция засегната при обхождането в дълбочина.

CSCB034 Упражнения по алгоритми и програмиране

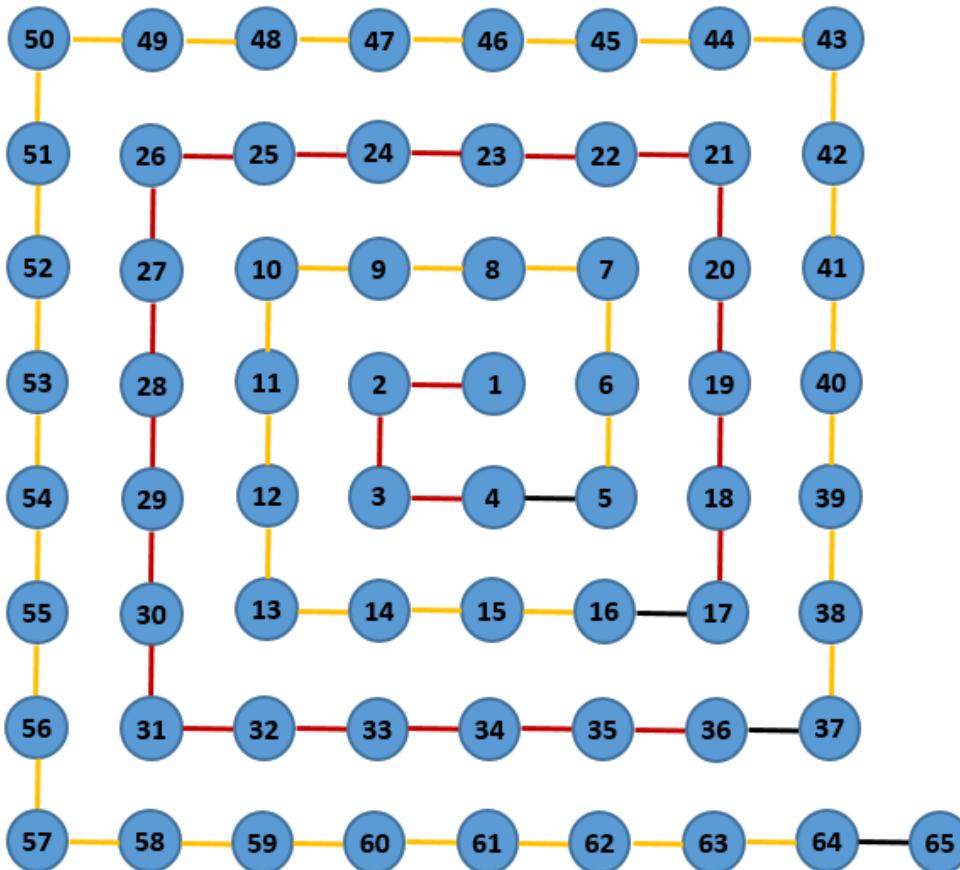
ДИНАМИЧНО ПРОГРАМИРАНЕ И ГРАФИ

гл. ас. д-р Слав Ангелов, НБУ

Загръвка (странична задачка)

Имаме спирала на числата от 1 до n.
Дефинираме Ниво 1 като множеството на стойностите $\{1, 2, 3, 4\}$. Ниво 2 съдържа ниво едно плюс $\{5, 6, \dots, 16\}$.
Ниво 3 съдържа Ниво 2 плюс $\{17, \dots, 36\}$.
Идеята е, че нивата образуват точно квадрати. Напишете максимално ефикасен алгоритъм, който да казва, от кое ниво е произволно зададена стойност.

Пример: Числото 3 е от Ниво 1, 10 е от Ниво 2, 24 е от Ниво 3, 54 е от Ниво 4, 64 е последната стойност от Ниво 4 и т.н..



Принцип на Белман

Ако можем да разбием една задача на n части, то оптиманото решение на задачата се достига, когато решим оптимално всяка от тези подзадачи.

Принцип на Белман

Ако можем да разбием една задача на n части, то оптималното решение на задачата се достига, когато решим оптимално всяка от стъпките.

Забележка: При използване на техниката „разделяй и владей“ може да се случи многократно изпълняване на една и съща задача, което увеличава сложността.
Динамичното програмиране се опитва да разреши този проблем.

Динамично програмиране

Динамично програмиране: Разбиваме задачата на подзадачи. Задачите са «една в друга», така, че решавайки тези в дъното на рекурсията поетапно да може да решим и останалите без да се налага да повтаряме вече решени такива.

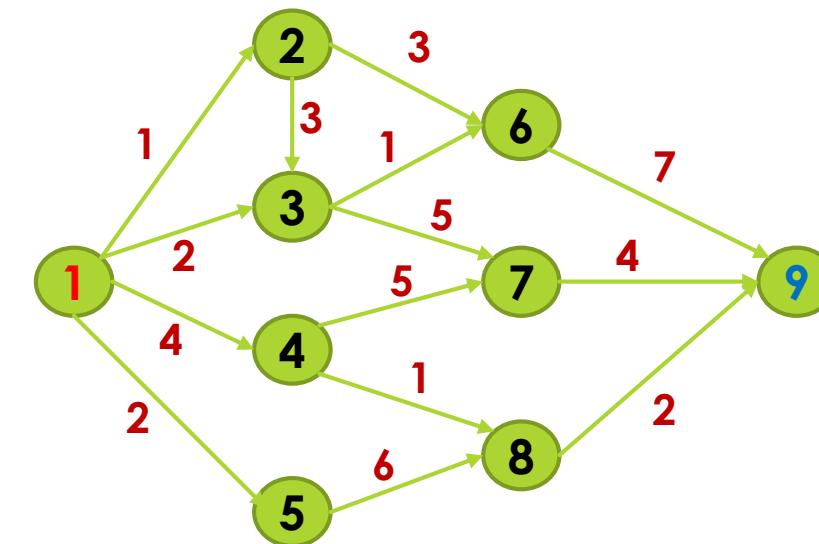
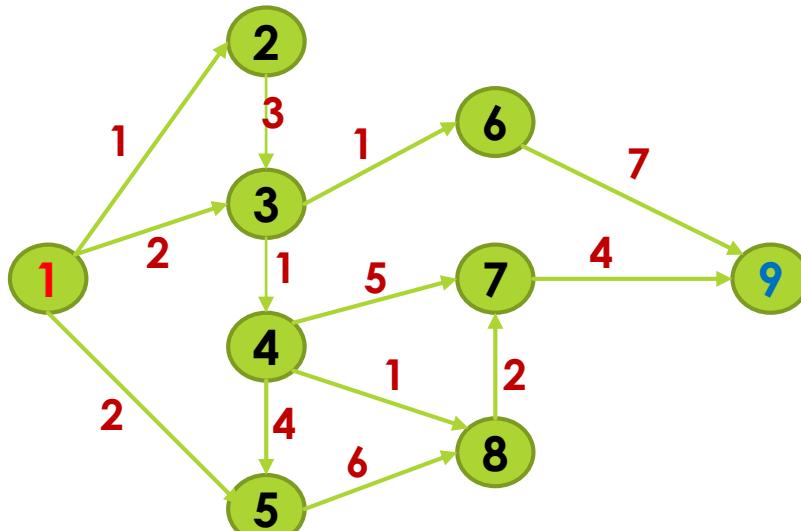
Практически всяка задача, която се решава с ДП може да се постави в две форми:

- ▶ **слаба**, при която се търси някаква „оптимална“ стойност;
- ▶ **силна**, при която се търси някаква структура в данните, при която се получава „оптимална“ стойност, т.е. силната форма на разглежданата задача е да се поиска да се посочи коопериранието, при което се получава оптимумът.

Мрежи

Мрежата е ориентиран ацикличен граф с уточнен начален и краен връх (по подразбиране съществува път между тях). Често ребрата са му с цени.

Примери:

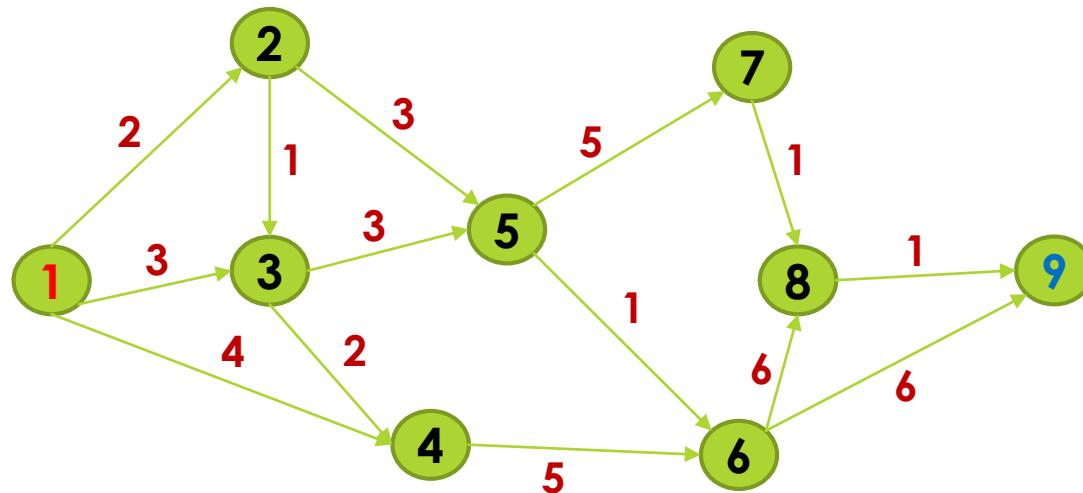


Намиране на най-къс път в мрежа

Под намирането на най-къс (дълъг) път в мрежа ще разбираме намирането на такъв път между началния и крайния връх.

Най-къс път в мрежа с динамично програмиране (частен случай)

Нека имаме следната мрежа:



Ще означаваме с $F(i)$ разстоянието от i -ти връх до крайния връх, а с $f(i,j)$ стойността на дъгата от връх i към j .

Алгоритъмът за примера (1/2)

Тръгва се от последния връх, знаем, че $F(9)=0$:

- 1) Търсим $F(8)$, имаме само един път от 8 до 9, така $F(8)=f(8,9)+F(9)$;
- 2) Търсим $F(7)$, Връх 7 имам само един съсед и това е Връх 8. Най-късият път до края е $F(7)=f(7,8)+F(8)$;
- 3) Търсим $F(6)$, 6 има достигими съседи 8 и 9, тоест разглеждаме $f(6,8)+F(8)$ и $f(6,9)+F(9)$. Така установяваме, че $F(6)=f(6,9)+F(9)$;
- 4) Търсим $F(5)$, 5 има съседи 6 и 7, избираме между $f(5,6)+F(6)$ и $f(5,7)+F(7)$, така $F(5)=f(5,6)+F(6)$;
- 5) Търсим $F(4)$, 4 има съседи само 6, $F(4)=f(4,6)+F(6)$.

Алгоритъмът за примера (2/2)

- 6) Търсим $F(3)$, 3 има съседи 4 и 5, търсим измежду $f(3,4)+F(4)$ и $f(3,5)+F(5)$, $F(3)=f(3,5)+F(5)$.
 - 7) Търсим $F(2)$, 2 има съседи 3 и 5, търсим измежду $f(2,3)+F(3)$ и $f(2,5)+F(5)$, $F(2)=f(2,3)+F(3)$.
 - 8) Търсим $F(1)$, 1 има съседи 2, 3 и 4; търсим измежду $f(1,2)+F(2)$, $f(1,3)+F(3)$, $f(1,4)+F(4)$, $F(1) = f(1,2)+F(2)$.
- Сега, ако погледнем от връх едно имаме $F(1) = f(1,2)+F(2) = f(1,2)+ f(2,3)+F(3) = f(1,2)+ f(2,3)+f(3,5)+F(5) = f(1,2)+ f(2,3)+f(3,5)+ f(5,6)+F(6) = f(1,2)+ f(2,3)+f(3,5)+ f(5,6)+ f(6,9)+F(9)$, т.е. пътя е 1, 2, 3, 5, 6, 9.

Последното, което направихме е реално да изведем силната форма на ДП.

Задача

Напишете код, който да реши задачата при такъв частен случай.

Размисли

Забележете, че докато търсехме най-късия път от зад напред намерихме най-късите пътища от всеки връх до края. Можем ли да кажем, че имаме дърво на най-късите пътища с корен крайния връх?

Размисли

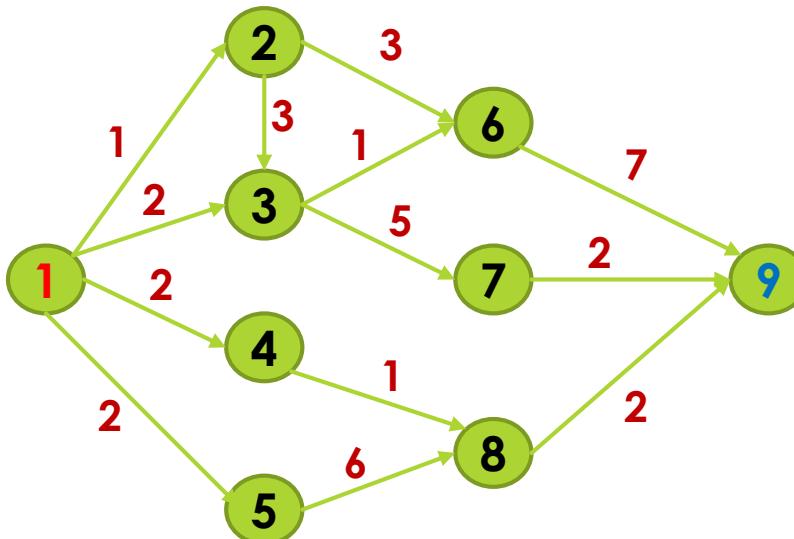
Забележете, че докато търсехме най-късия път от зад напред намерихме най-късите пътища от всеки връх до края. Можем ли да кажем, че имаме нещо като дърво на най-късите пътища с корен крайния връх?

Отговор:

Имаме дърво на най-късите пътища от началото, но не и от края. Причината е, че това е насочен граф и ние реално не може да се връщаме назад по него.

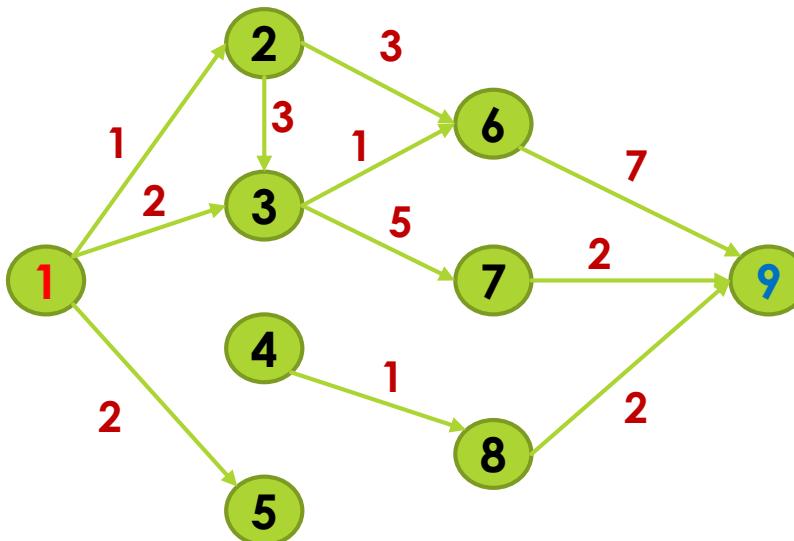
Задача 2

Ще възникне ли проблем, ако някъде възникнат връзки до връх, които са с еднакви тегла ?



Задача 26

Ще възникне ли проблем, ако някой връх не е свързан с изходния? А с началния?



Задача 3

Има ли прилика между представения алгоритъм за динамично програмиране и алгоритъма на Дейкстра?

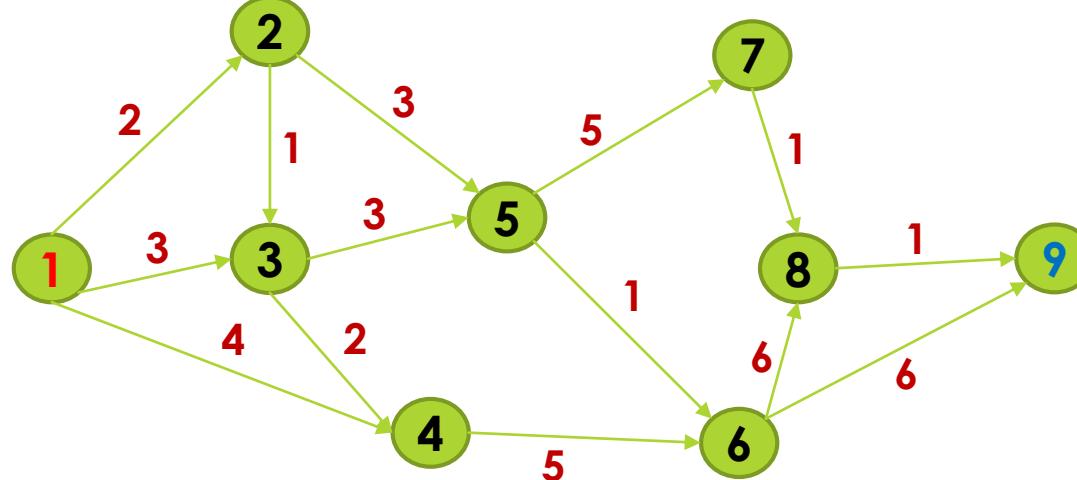
Задача 4

Защо това е само частен случай? Дайте пример където този алгоритъм ще има проблеми.

Задача 4 - подсказка

Защо това е само частен случай? Дайте пример където този алгоритъм ще има проблеми.

Разменете стойностите
на върхове 3 и 5. Какво става с
алгоритъма?



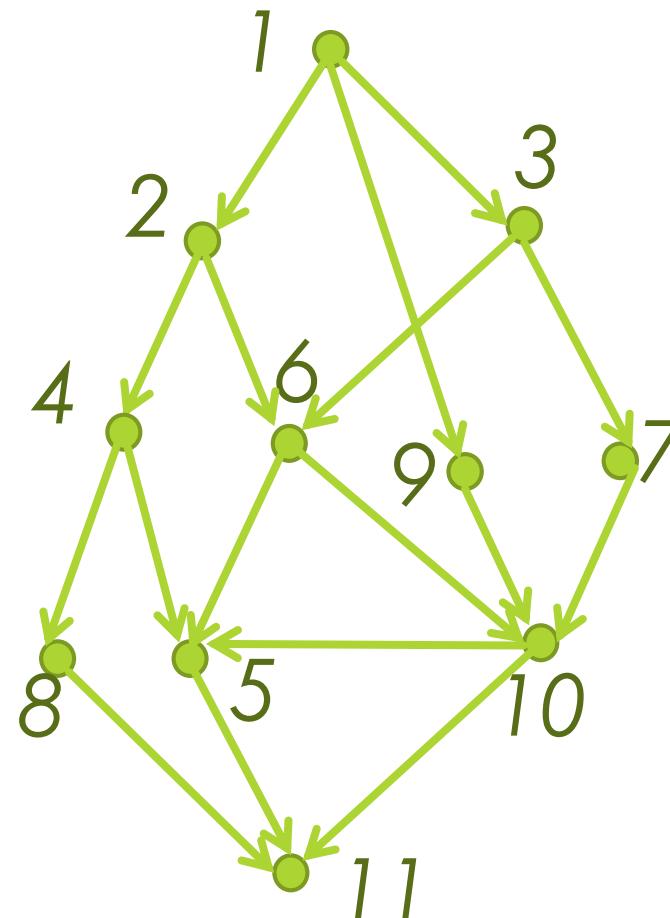
Задача 5

Разширете представената техника за динамично програмиране така, че да работи за произволна мрежа.

На следващите слайдове ще видите потенциално решение, но може ли на база на рекурсия с мемоизация ?

ДП с топологическо сортиране

Задача π. Даден е ориентиран ацикличен граф (даг) $G(V, E)$. Да се намери дължината на най-дълъг маршрут (ориентиран път) в G .



ДП с топологическо сортиране

Нека $|V| = N$, $V=\{1, 2, \dots, N\}$.

Разбиваме задачата π на подзадачи $\pi(i)$, $i=1, 2, \dots, N$, където $\pi(i)$ е задачата да се намери дълчината на най-дългата верига, завършваща в i .

Тъй като решението на всяка от подзадачите е число, решенията им запомняме в „линейна“ таблица $L[1:N]$, като в $L[i]$ записваме решението на $\pi(i)$.

Решение на задачата π е решението на някоя от подзадачите $\pi(i)$, където i е някой от върховете без наследници.

ДП с топологическо сортиране

Теорема.

Ако върхът i няма предшественици

$$L[i] = 0,$$

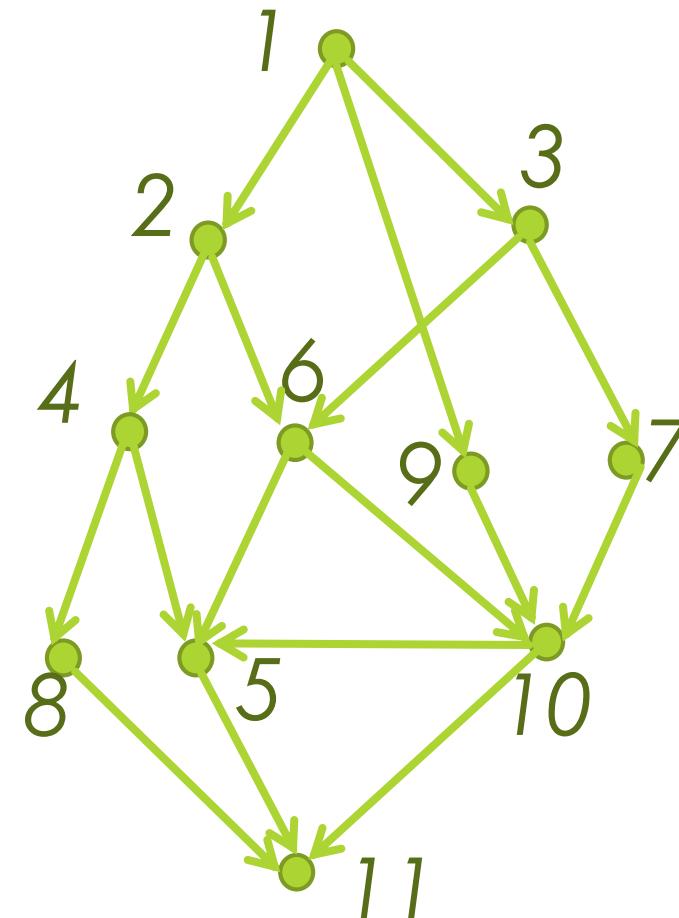
Иначе,

$$L[i] = \max\{L[j_1], L[j_2], \dots, L[j_k]\} + 1$$

където j_1, j_2, \dots, j_k са преките предшественици на i .

Представяне на граф със списъци на предшествениците

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



ДП с топологическо сортиране

```
int G[MAXN][MAXN]; //сп. на съседите int H[MAXN][MAXN]; //сп. на предшеств.
int L[MAXN], N, S[MAXN]; //топ. сортирани върхове
void opt(){    top_sort(G,S);
    for(int i=1;i<=N;i++)
    { int k=S[i]; L[k]=0;
        if(H[k][0]!=0)
        { for(int j=1;j<=H[k][0];j++)
            if(L[H[k][j]]>L[k]) L[k]=L[H[k][j]];
        L[k]++;
    }
}
```

Бонус задача (допълнителен материал)

Имаме масив от числа. Можете ли да изчислите извадково очакване и популационната дисперсия само с един for цикъл.

Ако имаме случаина извадка от стойности x_1, \dots, x_n , то дефинираме **извадковото очакване**:

$$E(X) = \mu = \frac{x_1 + \dots + x_n}{n}.$$

Дефинираме **популационната дисперсия**:

$$Var(X) = E((x - \mu)^2) = \frac{(\mu - x_1)^2 + \dots + (\mu - x_n)^2}{n}.$$

Бонус задача - подсказка

Възможно е следното представяне на популационната дисперсия:

$$\text{Var}(X) = E((X - \mu)^2) = E(X^2 - 2X\mu + \mu^2) = E(X^2) - 2\mu E(X) + \mu^2 = E(X^2) - E^2(X).$$

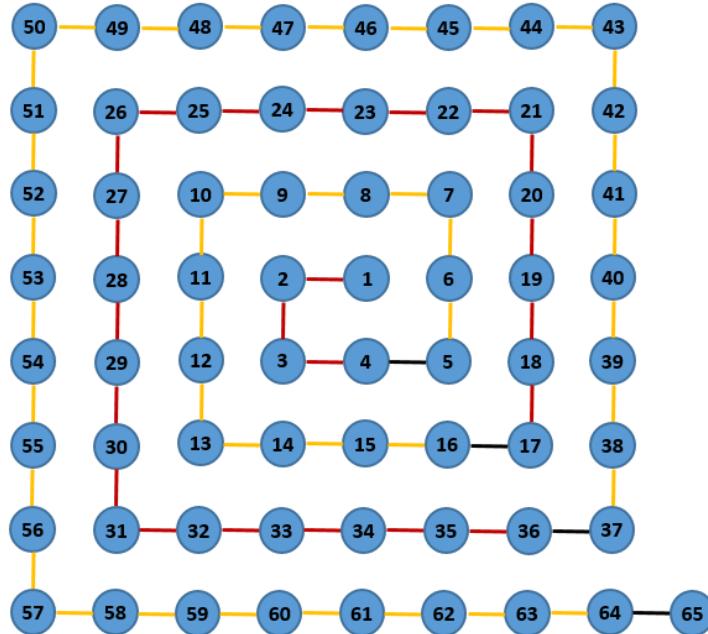
Така стигнахме до следните два израза за пресмятане:

$$E(X^2) = \frac{x_1^2 + \dots + x_n^2}{n}, \quad E^2(X) = \mu^2.$$

Решение на бонус задачата от Седмица 5

ас. д-р Слав Ангелов

Имахме следната картишка:



Нека разгледаме броят елементи на всяко ниво:

- Ниво 1 -> 4
- Ниво 2 -> $12 = 2 \cdot 4 + 4$
- Ниво 3 -> $20 = 4 \cdot 4 + 4$
- Ниво 4 -> $28 = 6 \cdot 4 + 4$
- Ниво 5 -> $36 = 8 \cdot 4 + 4$
-
- Ниво N -> $2 \cdot (N-1) \cdot 4 + 4$

Колко общо елементи $S(N)$ имаме до конкретно ниво N:

$$S(N) = \sum_{i=1}^N (8(i-1) + 4) = \sum_{i=1}^N (8i - 4) = 8 \sum_{i=1}^N i - 4N = 8 \frac{N(N+1)}{2} - 4N = 4N^2.$$

Така намерихме, че броят на елементите на пълно ниво N е $4N^2$.

За произволен индекс n, на когото търси нивото знаем, че се колебае между две пълни нива N-1 и N, т.е. е равен на:

$$n = 4(N-1)^2 + e,$$

където e е това, което е в повече на n над ниво N-1. Сега:

$$\frac{\sqrt[2]{n}}{2} = \frac{\sqrt{4(N-1)^2 + e}}{2},$$

Ако e е:

- $= 0$, то този израз дава резултат $N-1$ (п съвпада с пълно ниво $N-1$);
- $< 8N-4$, то изразът е между $N-1$ и N ;
- $= 8N-4$, то п съвпада с пълно ниво N и формулата ще върне N .

Тоест $\frac{\sqrt[2]{n}}{2}$ няма да ни даде нивото само случая $e < 8N - 4$ (стойността ще е между $N-1$ и N), но ако закръглим нагоре $\frac{\sqrt[2]{n}}{2}$ този проблем се решава.

Финална формула за получаване на N от n :

$$N = roundup\left(\frac{\sqrt[2]{n}}{2}\right).$$

CSCB034 Упражнения по алгоритми и програмиране

ОЩЕ ЗА ГРАФИ

гл. ас. д-р Слав Ангелов, НБУ

Размисли

Можем ли да използваме представената техника за динамично програмиране за намиране на най-къс път в мрежа (вижте презентацията от миналия път) за произволен граф ?

Допълнително

При претеглените пълни графи, които са с неотрицателни тегла, намирането на най-дълъг път е еквивалентно на задачата за търговския пътник.

Отговор

Уви, без мащабна модификация, не можем. Това лесно се вижда като съставите контрапример с цикъл (в мрежи няма цикли, но в графи не е изключено).

Алгоритъм на Флойд-Уоршал

Имаме ориентиран граф с цени. За разлика от Дейкстра алгоритъмът на Флойд-Уоршал търси най-къси пътища от всеки връх до всеки връх.

Основна идея: Имаме ориентиран $G(V, E, c)$, $V = \{1, 2, 3, \dots, n\}$.

Търсим най-къс път между върхове i и j , междинните върхове в този път търсим измежду върхове $\{1, 2, \dots, k\}$. Имаме два случая:

- 1) k участва в този най-къс път като междинен връх. Тогава търсим най-къс път между i и j с междинни върхове измежду $\{1, 2, \dots, k-1\}$;
- 2) k не участва в този най-къс път. Тогава вече търсим два най-къси пътища, такъв между i и k , и такъв между k и j , и двата пътища с междинни върхове измежду $\{1, \dots, k-1\}$.

Реализация

```
int G[MAXN][MAXN], N, P[MAXN][MAXN];  
  
void floyd(){  
    int i, j, k;  
    for(k=1; k<=N; k++)  
        for(i=1; i<=N; i++)  
            for(j=1; j<=N; j++)  
                if(G[i][k]+G[k][j]<G[i][j]) {  
                    G[i][j]=G[i][k]+G[k][j];  
                    P[i][j]=k;  
                }  
}
```

```
void path(int v,int w){  
    printf("%d", v);  
    if(P[v][w]!=0){  
        path(v, P[v][w]);  
        path(P[v][w], w);  
    }  
}  
#define INF 2000000  
int main() { int i, j, k;  
    for(i=1; i<=N; i++){  
        for(j=1; j<=N; j++){  
            G[i][j]=INF;  
            P[i][j]=0;}  
        G[i][i]=0;  
    } ...
```

Реализация - осмисляне

Обяснете си кода като го разиграете върху малки графи на хартия.

Върши ли работа кодът за неориентирани графи?

Работили реализацията при наличие на отрицателни тегла?

Контролно 2

Контролното отново ще се проведе в Хакерранк. Не ви е необходимо нищо друго освен регистрация.

Задачите ще са три и ще са върху графи (включително динамично програмиране върху мрежа). Първата задача ще е по-лесна от останалите и дава 100 точки, което е достатъчно за 3.00. Всяка от останалите дава по още 150 точки (100 точки = 1 единица).

Разрешава се копиране на предварително приготвен код, така че си съставете стратегия за бързодействие.

Основни акценти

- ▶ Търсен в дълбочина и ширина;
- ▶ Покриващи дървета;
- ▶ Дейкстра;
- ▶ Флойд-Уоршал;
- ▶ Алгоритъмът за динамично намиране на оптимален път в мрежа.

Универсален вход за всички задачи по графи

На контролното ще се изисква следния вход:

Последователно въвеждаме следните стойности:

n – брой върхове, r - брой ребра,

ребрата: i – начален връх, j – краен връх , w – цена на съответното ребро (ако ни трябват цени),

други стойности (например връх за корен на покриващо дърво).

Записвайте графа чрез списъци на съседите и сортирайте всеки ред (без стойност с индекс 0)!

Внимание, НЕ пишете нищо друго освен необходимите стойности! Изрази от вида „Въведете графа:“ и други подобни няма да се прочетат от проверяващата програма.

Възможна задача за контролно (1/6)

Разширете представената техника за динамично програмиране в мрежа така, че да работи за произволна мрежа и да намира най-дълъг път.

Изход:

- 1) **Едно число** – дължината на най-дългия път.
- 2) **Последователност от числа** – върховете от началото до края на мрежата, които задават най-дългия път.

Упътване за Задача 1

Достатъчно е да обхождате от последния до първия връх докато не намерите $F(1)$. Например повтаряйте алгоритъма с един while цикъл и той да спира, когато $F(1)$ е намерен.

Възможна задача за контролно (2/6)

Създайте програма, която да преброи всички различни пътища от началото до края в мрежа.

Изход:

Едно число – брой пътища.

Идеи

Начин 1: Използвайте лемата, която гласи, че докато вдигате на степен к матрицата на съседство получавате всички пътища от връх до връх с дължина k . Тук сложност е да прецените кога да спрете с повдигането на степен, т.е. колко да е максималното k , но понеже е мрежа сравнително лесно може да се измисли груба граница.

Начин 2 (за предпочтение): Започнете от крайният връх и се движете по мрежата. Алгоритъмът прилича на този за намирането на най-къс път, но този път броите колко пътища имате от съответния връх до края.

Забележка: Възможни са и други подходи.

Възможна задача за контролно (3/6)

Създайте програма, която по зададен връх да извърши търсене в ширина и да изчисли всички възможни покриващи дървета с корен този връх.

Изход:

Едно число – брой възможни различни дървета.

Упътване

Получете нивата чрез обхождане в ширина. Търсим всички дървета на база на тези нива, т.е. няма да следим за ребра между върхове на едно и също ниво.

- 1) Ниво едно не е интересно , там е корена на дървото. Ниво две също, защото всичките му върхове са свързани с корена.
- 2) Гледаме върховете от Ниво 3. Всеки връх от Ниво 3 трябва да има точно една връзка с връх от Ниво 2. Умножаваме броят на бащите на върховете от Ниво три с Ниво 2 и получаваме всички възможни дървета на тази етап.
- 3) Повтаряме Стъпка 2 между Нива 3 и 4. Получаваме всички възможности от за Ниво 4 от Ниво 3. Сега, ако искаме да знаем всички покриващи дървета от корена до Ниво 4, то трябва да умножим възможностите от Ниво 3 с възможностите от Ниво 4.
- 4) Продължаваме процедурата, докато не изчерпим нивата.

Възможна задача за контролно (4/6)

Бизнес дама притежава магазини за дрехи в N сгради и знае разстоянията между тези сгради. Тя иска да се настани в една от тези сградите, така че сумарно да е най-близо до всички останали нейни магазини.

Изход:

Едно число – номер на върха, който отговаря на исканата сграда.

Упътване (4/6)

Подход 1: Алгоритъмът на Флойд-Уоршал. Така от всяка точка до всяка точка ще знаем най-късите пътища. След това с прости сумирания ще преценим сумарно от коя сграда сме най-близо до всички.

Подход 2: Дейкстра за всеки връх. След това продължаваме аналогично на Подход 1.

Възможна задача за контролно (5/6)

Пренапишете алгоритъма на Дейкстра така, графът да може да има отрицателни цени на ребрата.

Изход:

Едно число – стойността на най-късия път.

Имплементация на Дейкстра

```
void dijkstra(int r) {  
    int v, w, min, i, j;  
    for(i=1; i<=n; i++) {U[i]=0; p[i]=r; d[i]=INF;}  
    d[r]=p[r]=0;  
    for(i=1; i<=n; i++) {  
        v=0; min=INF;  
        for(j=1; j<=n; j++) if(U[j]==0 && d[j]<min) { v=j; min=d[j]; }  
        U[v]=1;  
        for(j=1; j<=G[v][0]; j++) { w=G[v][j];  
            if(U[w]== 0 && d[v]+c[v][w]<d[w]) { d[w]=d[v]+c[v][w]; p[w]=v; }  
        } } }
```

Декларираме на глобално ниво:

```
#define INF 1000000000  
int G[MAX][MAX], c[MAX][MAX],  
U[MAX], p[MAX], d[MAX], n, m, r;
```

За какво ни е?

Упътване (5/6)

- 1) Проверяваме дали има отрицателни тегла;
- 2) Ако имаме, намираме най-малкото от тях;
- 3) Умножаваме неговата стойност по -1 и я събираме със всички останали;
- 4) Сега Дейкстра работи;
- 5) Сега сами преценете как да намерите стойността на най-късия път преди модификацията на теглата.

Възможна задача за контролно (6/7)

Старшина разполага с N задачи, които може да дава на подчинените си, всяка задача отнема време на отряда, което той добре знае. Възможно е времето от различни задачи към една и съща задача да е различно, например времето от А към В да е различно от С към В, поради сложности в подготовката от един вид задачи към друг. Възможно е и времето от А до Б, да е различно от времето от Б до А. Да допуснем, че старшината на случаен принцип избира първата задачка. Войниците изпълняват задачите последователно. Помогнете на старшината да подбере последователност от K различни задачи, $K < N$, така че войниците да ги свършат последователно максимално бързо.

Вход: N, K, I – стартови връх.

Изход:

t – общото време ангажираност,

Последователност от числа – те са номерата на задачките в последователността на изпълняване.

Упътване

Независимо, че се опитваме да минимизираме цени, тази задача не може да бъде решена с Дийкстра (и да даде най-къс път от даден връх до всички останали няма гаранция, че ще имате път през точно K точки, който да ви върши работа).

Задачата е NP-пълна и изисква изчерпване на всички възможно случаи....

Възможна задача за контролно (7/7)

Старшина разполага с N задачи, които може да дава на подчинените си, всяка задача отнема време на отряда, което той добре знае. Възможно е времето от различни задачи към една и съща задача да е различно, например времето от А към В да е различно от С към В, поради сложности в подготовката от един вид задачи към друг. Възможно е и времето от А до Б, да е различно от времето от Б до А. Да допуснем, че старшината на случаен принцип избира първата задачка. Войниците изпълняват задачите последователно. Помогнете на старшината да подбере последователност от K различни задачи, $K < N$, така че войниците да са ангажирани **максимално много време при последователното им изпълнение.**

Вход: N, K, I – стартови връх.

Изход:

t – общото време ангажираност,

Последователност от числа – те са номерата на задачките в последователността на изпълняване.

Упътване

**Подобно на предходната задача, независимо , че максимизираме, имаме NP-
пълна задача и изиска изчерпване на всички възможно случаи....**

Послеслов

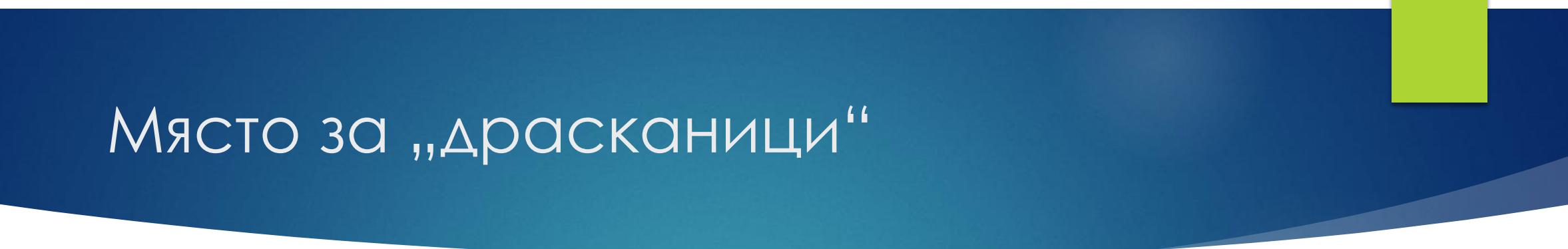
Възможно е задачи от контролното да не влизат точно в рамките на представените седем задачи (всъщност тези на контролното ще са по-лесни), но те ще са ви от полза при подготовката.

Прегледайте всички поставени задачи от другите ми презентации по графи.

CSCB039 Алгоритми и програмиране

ПОДГОТОВКА ЗА КОНТРОЛНО 2

гл. ас. д-р Слав Емилов Ангелов, НБУ



Място за „драсканици“

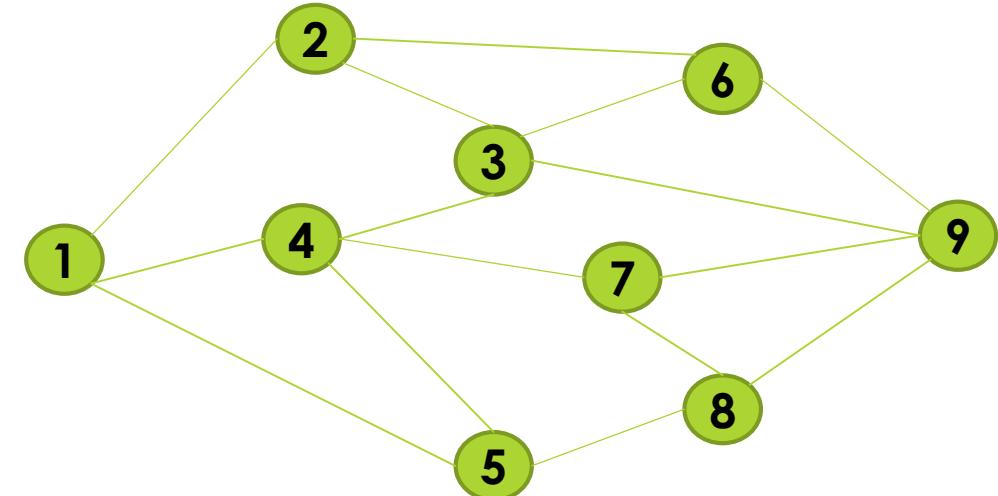
Структура на Контролно 2

1. На лист хартия разписвате задачи свързани със сложност на рекурсия – носят **1.5** точки;
2. Отделно имате и **кратък тест** в Мудъл, който ще е сравнително индивидуален – носят **3** точки.

Формула: 1.5 + брой точки.

Задача 0

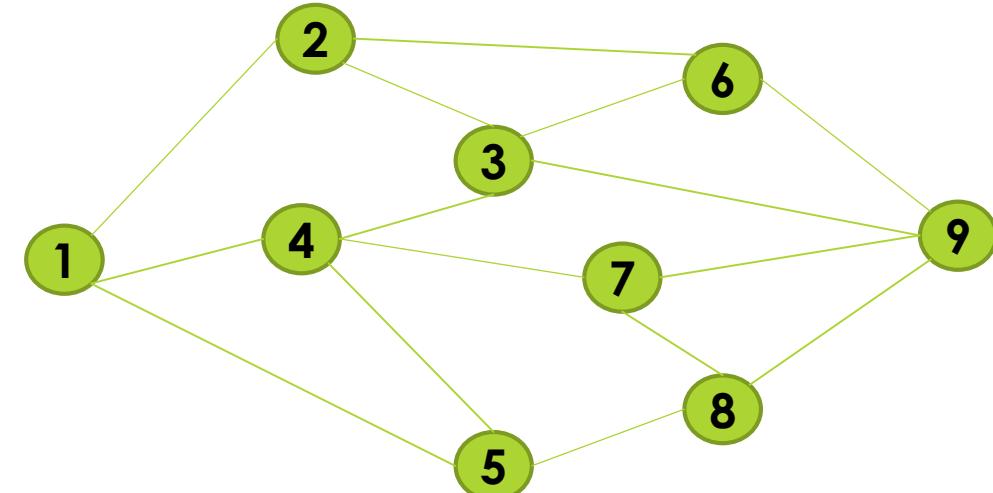
Попълнете списъците на съседите за следния граф:



Задача 0 - отговор

Попълнете списъците на съседите за следния граф:

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



Задача 1

Какво е предимството на динамичното програмиране пред „разделяй и владей“?

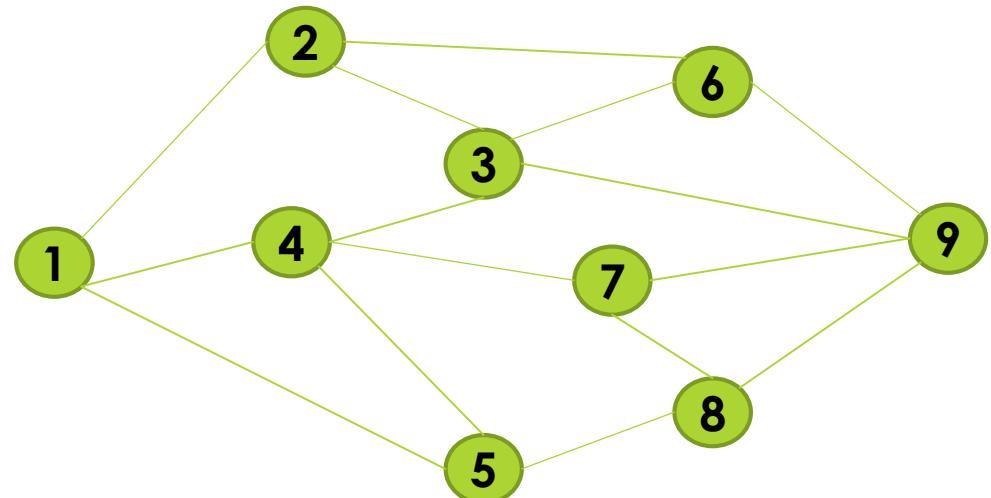
Отговор

Динамичното програмиране строи решението на база на предходните, така че да избягва повторение на вече решени подзадачи.

Задача 2

Имаме следната последователност от върхове: 1, 2, 4, 8, 5, 6, 9.

Имаме ли топологическо сортиране. Защо?



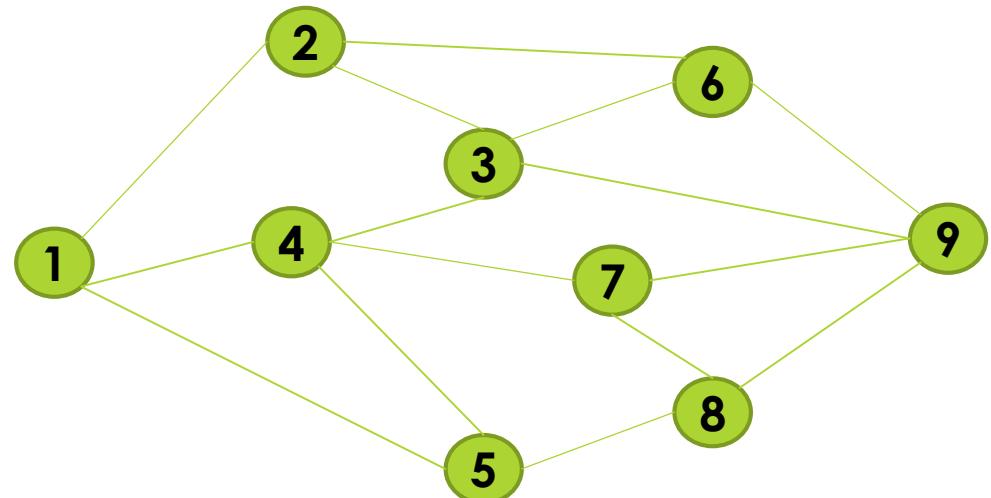
Отговор

НЕ, нямаме, защото при топологическото сортиране върховете са подредени така, че връх между два върха трябва да е на разстояние от корена по-малко или равно на разстоянието на върха му отдясно, но това не е изпълнено. Например ..., 4, 8, 5, ... , тук 8 е по-далеч от 5, а е преди него.

Задача 3

Какво е вярно за графа:

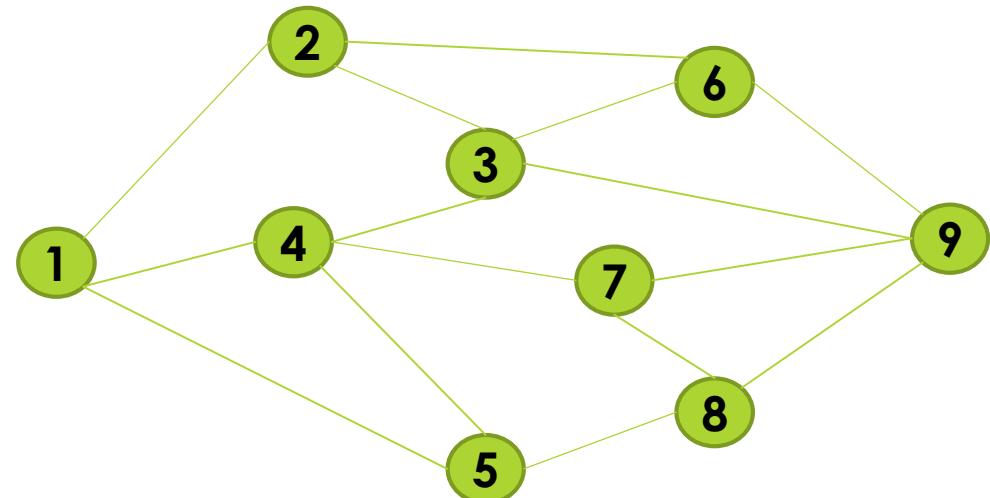
- a) Ацикличен;
- b) С тегла;
- c) Мрежа;
- d) Има 10 върха;
- e) Неориентиран.



Задача 3 - отговор

Какво е вярно за графа:

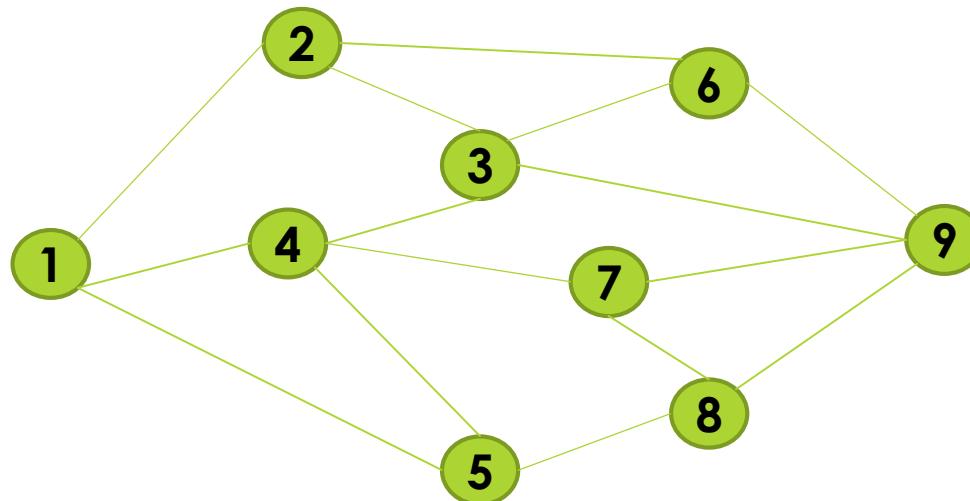
- a) Ацикличен;
- b) С тегла;
- c) Мрежа;
- d) Има 10 върха;
- e) Неориентиран.



Задача 4

Колко нива има при обхождане в дълбочина (броим нулевото):

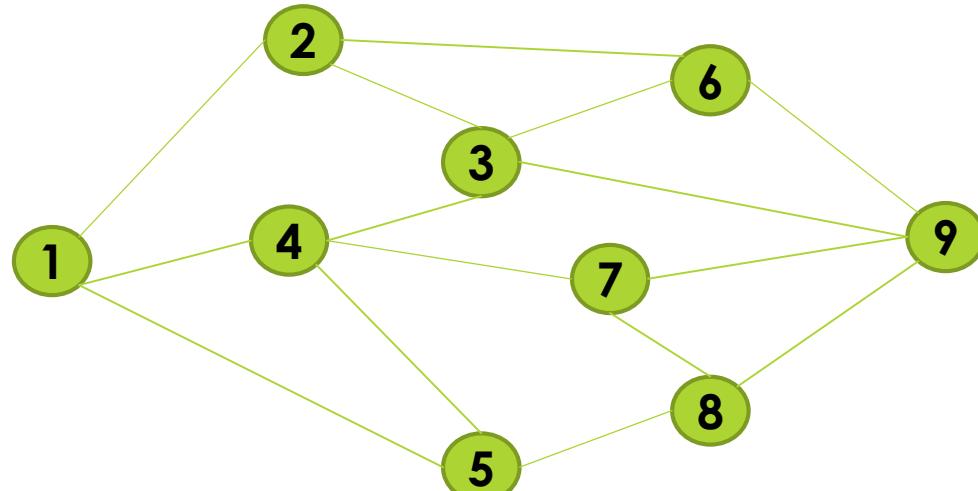
- a) 2
- b) 3
- c) 4
- d) 5
- e) 6



Задача 4 - отговор

Колко нива има при обхождане в дълбочина (броим нулевото):

- a) 2
- b) 3
- c) 4
- d) 5
- e) 6



Задача 5

**Можем ли да кажем, че при обхождане в ширина имаме толкова нива,
колкото е дължината на пътя от корена до най-отдалечения връх,
измерен в ребра, +1?**

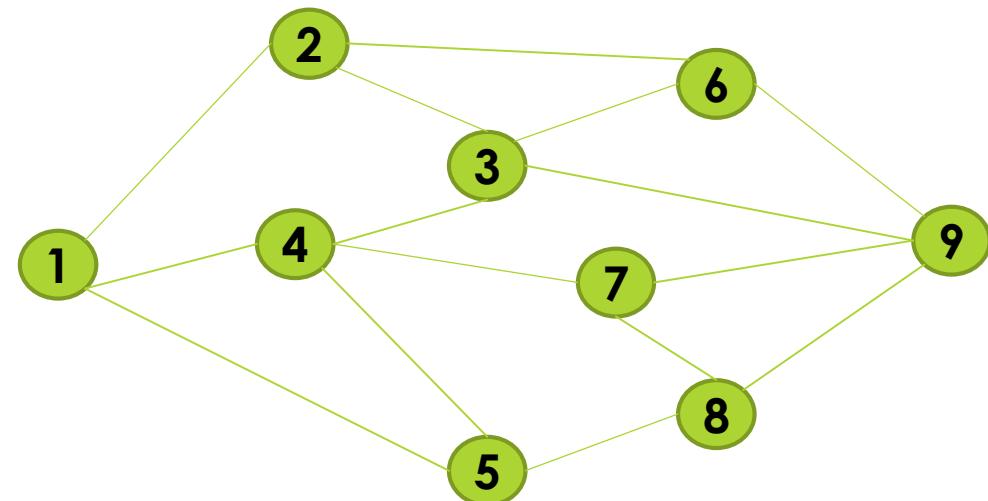
Задача 5 - решение

Можем ли да кажем, че при обхождане в ширина имаме толкова нива, колкото е дължината на пътя до най-отдалечения връх, измерен в ребра, +1?

Да, можем, защото при прехода от ниво към ниво минаваме през точно едно ребро. Тогава при n нива, минаваме през $n-1$ ребра.

Задача 6

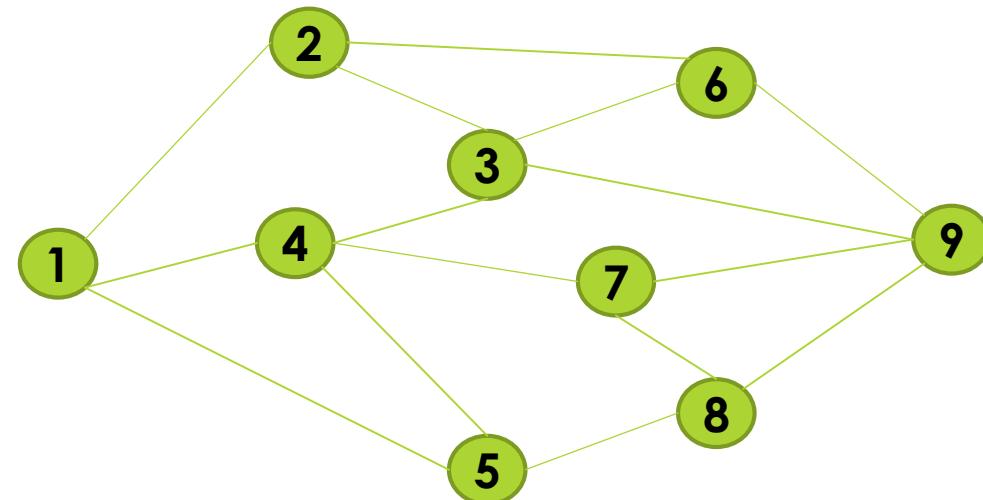
Кои са върховете от Ниво 2 при обхождане в ширина от 1?



Задача 6 - отговор

Кои са върховете от Ниво 2 при обхождане в ширина от 1?

Върховете са 3, 6, 7, 8.

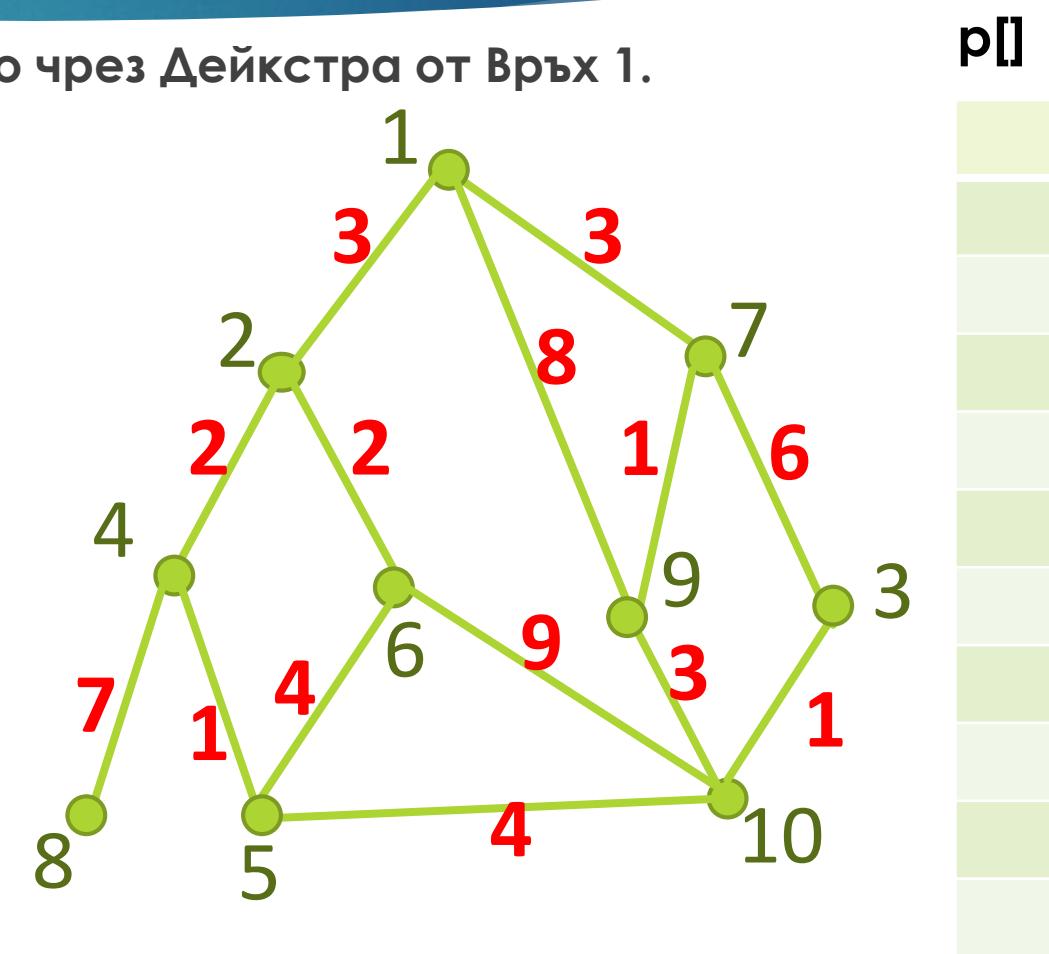


Задача 7

Попълнете $P[]$ с минималното покриващо дърво чрез Дейкстра от Връх 1.

1	2	3	4	5	6	7	8	9	10

$d[]$



Имплементация на Дейкстра

```
void dijkstra(int r) {
    int v, w, min, i, j;
    for(i=1; i<=n; i++) {U[i]=0; p[i]=r; d[i]=INF;}
    d[r]=p[r]=0;
    for(i=1; i<=n; i++) {
        v=0; min=INF;
        for(j=1; j<=n; j++) if(U[j]==0 && d[j]<min) { v=j; min=d[j]; }
        U[v]=1;
        for(j=1; j<=G[v][0]; j++) { w=G[v][j];
            if(U[w]==0 && d[v]+c[v][w]<d[w]) { d[w]=d[v]+c[v][w]; p[w]=v; }
        }
    }
}
```

Декларираме на глобално ниво:

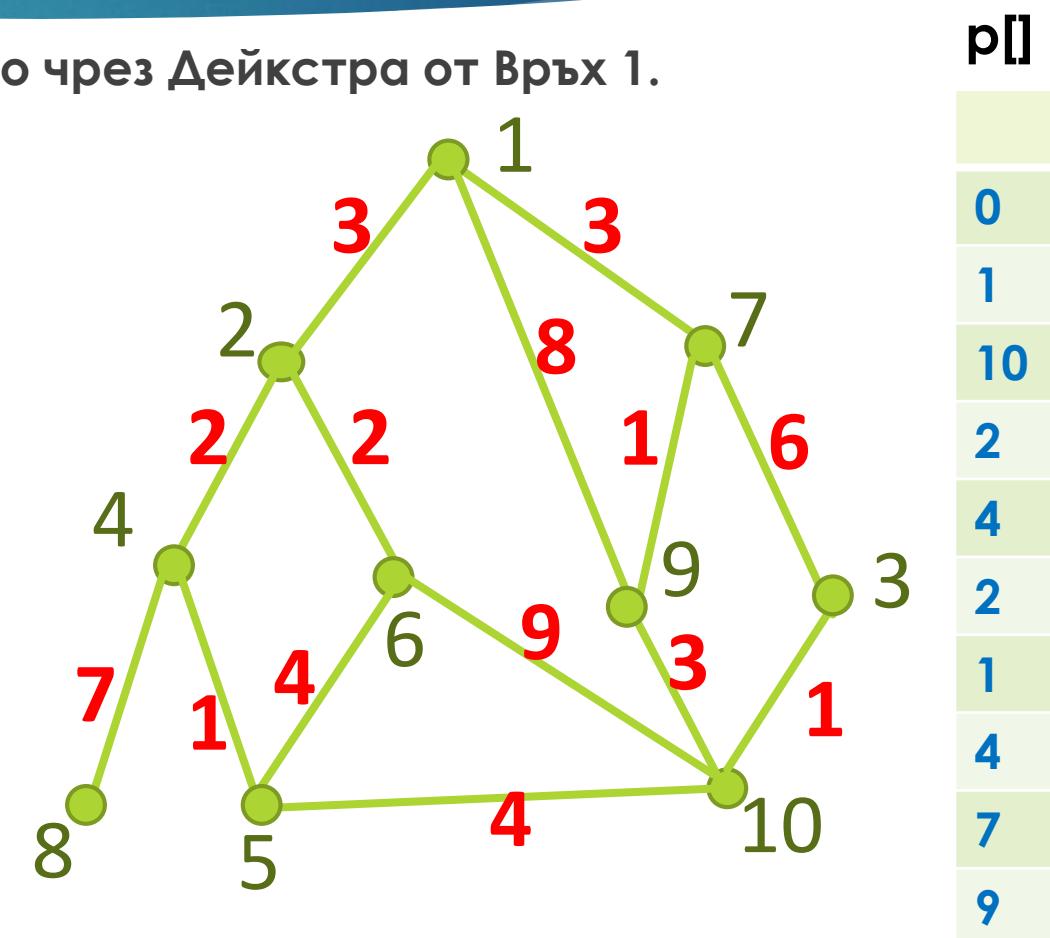
```
#define INF 1000000000
int G[MAX][MAX], c[MAX][MAX],
U[MAX], p[MAX], d[MAX], n, m, r;
```

Тръгваме по върха,
генерирал най-къс
път до сега !

Задача 7 - отговор

Попълнете $P[]$ с минималното покриващо дърво чрез Дейкстра от Връх 1.

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

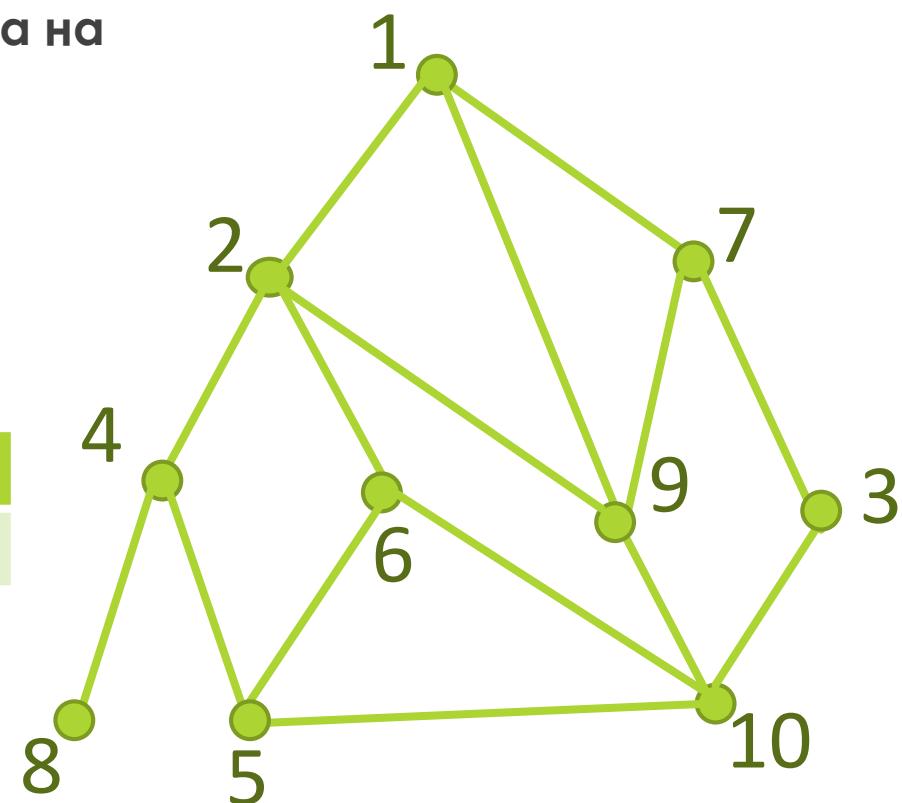


Задача 8

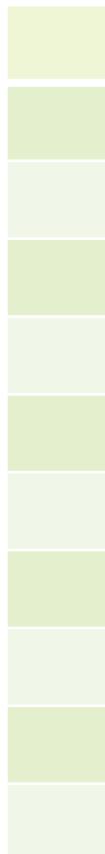
Попълнете $P[]$ с покриващото дърво чрез DFS. Разиграйте и самият метод. За DFS гледате имплементацията на следващия слайд.

Списъците на съседите $G[]][]$ са сортирани!

Stack											
Out:											



$p[]$



Алгоритъм за построяване ПД с „обхождане в дълбочина“

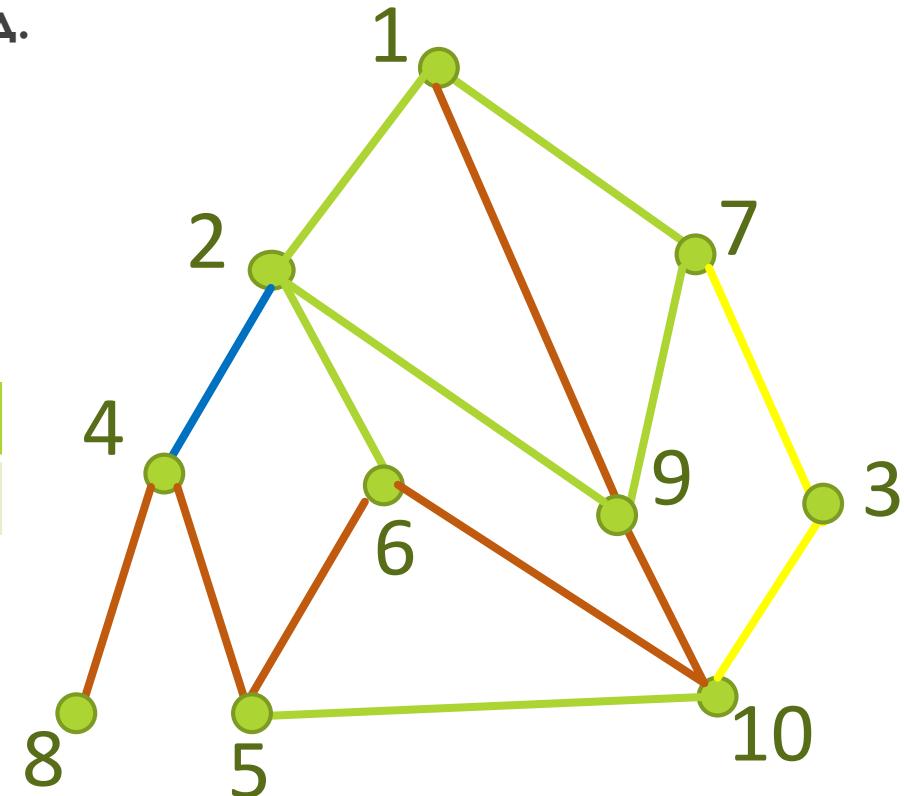
```
int U[MAXN], S[MAXN], P[MAXN], G[MAXN][MAXN], N;  
  
void DFS (int r){  
    for(int i=1; i<=N; i++) U[i]=0;  
    empty_stack();  
    push(r); U[r]=1; P[r]=0;  
    while(not_empty()){ int t=look();  
        if(G[t][0]>0){  
            int v=G[t][G[t][0]--];  
            if(!U[v]) {U[v]=1; P[v]=t; push(v);} } else pop();  
    } } }
```

```
int S[MAXN], top;  
  
void empty_stack(){top=-1;}  
  
void push(int x){S[++top]=x;}  
  
void pop(){top--;}  
  
int look(){return S[top];}  
  
bool not_empty(){return top>-1;}
```

Задача 8 - решение

Попълнете $P[]$ с покриващото дърво чрез DFS. За DFS гледате имплементацията на следващия слайд.

Stack	1	9	10	6	5	4	8	2	3	7
Out:	8	2	4	5	6	7	3	10	9	1



Задача 9

Проверете дали сме получили топологическо сортиране в Задача 8. Обосновете отговора си.

Задача 9 - отговор

Проверете дали сме получили топологическо сортиране в Задача 8. Обосновете отговора си.

Да, имаме топологически сортиране, защото стойността между всеки две стойности не е по-отдалечена от ограждащите я стойности от корена.

Задача 10

Чрез итеративния подход изчислете сложността на следната РО:

$$T(n) = 3T(n-1)+3, \quad T(1) = c = \text{const.}$$

Задача 10 – решение

$$T(n) = 3 \cdot T(n-1) + 3 \quad \times 3^0 = 1$$

$$T(n-1) = 3 \cdot T(n-2) + 3 \quad \times 3^1$$

...

$$T(2) = 3 \cdot T(1) + 3 \quad \times 3^{n-2}$$

$$T(1) = c; \quad \times 3^{n-1}$$

$$T(n) = 3 \cdot T(n-1) + 3$$

$$3T(n-1) = 3 \cdot T(n-2) + 3$$

...

$$3^{n-2} T(2) = 3^{n-1} \cdot T(1) + 3^{n-1}$$

$$3^{n-1} T(1) = 3^{n-1} c.$$

И след като съберем и направим приведението получаваме:

$$T(n) = 3^1 + 3^2 + \dots + 3^{n-2} + 3^{n-1} + c3^{n-1} = 0.5 * 3^{n-1} - 3.5 + c3^{n-1} = O(3^{n-1}).$$

За формулата вижте следващия слайд
и я донагласете

НЯКОИ ВАЖНИ СУМИ

Както се вижда, при итериране трябва да може да се оцени някаква сума. Ето някои такива суми:

$$1 + 2 + \dots + n = n(n+1)/2 = O(n^2)$$

$$1^2 + 2^2 + \dots + n^2 = n(n+1)(2n+1)/6 = O(n^3)$$

$$2^0 + 2^1 + \dots + 2^n = 2(2^n - 1) = O(2^n)$$

$$3^0 + 3^1 + \dots + 3^n = (3(3^n - 1))/2 = O(3^n)$$

$$\log 1 + \log 2 + \dots + \log n = \log n! = O(n \cdot \log n)$$

Задача 11

Според опростената Мастър теорема (вижте следващия слайд) колко е сложността на:

$$T(1)=\text{const}; \quad T(n)=T(n/2)+n, \quad n>1.$$

Мастър теорема - опростен вариант

Тази теорема решава всички рекурсии от следния вид:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^k), \quad \text{където } a \geq 1, b > 1.$$

Имаме задача с размер n , която се дели на a подзадачи. Всяка от подзадачите в с размер n/b , където $a \geq 1, b > 1$. Всяка от a -те подзадачи се решава рекурсивно за време $T\left(\frac{n}{b}\right)$. Използва се $O(n^k)$, за да се покаже сложността на разделянето на подзадачи и тяхното сглобяване. Разграничават се три случая:

1. $k < \log_b a \rightarrow T(n) = O(n^{\log_b a})$;
2. $k = \log_b a \rightarrow T(n) = O(n^k \log n)$;
3. $k > \log_b a \rightarrow T(n) = O(n^k)$.

Задача 11 - решение

Според опростената Мастър теорема (вижте следващия слайд) колко е сложността на:

$T(1)=\text{const}; \quad T(n)=T(n/2)+n, \quad n>1.$ Следователно $a=1, b=2, k=1.$

$$1 = k > \log_b a = 0 \rightarrow T(n) = O(n^k) = O(n).$$

Задача 12

Имаме числова редица, такава че всеки следващ член се получава чрез събиране на предходните три члена като се започва с членовете 1, 2 и 1. Какво РО поражда тази задача? Колко е 6-тият член?

Задача 12 - решение

Имаме числова редица, такава че всеки следващ член се получава чрез събиране на предходните три члена като се започва с членовете 1, 2 и 1. Какво РО поражда тази задача? Колко е 6-тият член?

$$T(1)=1, T(2)=2, T(3)=3; T(n)=T(n-1)+T(n-2)+T(n-3)+\text{const.}$$

Първите членове са: 1, 2, 1, 4, 7, 12, 23, 42, 77, 142, 261...

Задача 13

Можем ли да решим с Мастър теоремата следната РО:

$$T(1)=\text{const}; \quad T(n)=2 \cdot T(n-1)-T(n-2), \quad n>1.$$

Задача 13 - решение

Можем ли да решим с Мастър теоремата следната РО:

$$T(1)=\text{const}; \quad T(n)=2 \cdot T(n-1)-T(n-2), \quad n>1.$$

Уви, формата на това РО не позволява решаване с Мастър теоремата.

Задача 14

Можем ли да решим с Мастър теоремата рекурсивното обръщане на масив, РО:

$$T(1)=\text{const}; \quad T(n)=T(n-2)+\text{const}, \quad n>1.$$

Задача 14 – решение

Можем ли да решим с Мастър теоремата рекурсивното обръщане на масив, РО:

$$T(1)=\text{const}; \quad T(n)=T(n-2)+\text{const}, \quad n>1.$$

Отново входните данни не са за Мастър теоремата и е удачно да използваме итеративния подход.

Задача 15

Колко е сложността на Mergesort според Матър теоремата? РО:

$$T(n)=2T(n/2)+cn.$$

Задача 15 - решение

Колко е сложността на Mergesort според Матър теоремата? РО:

$$T(n) = 2T(n/2) + cn.$$

Имаме $a=2$, $b=2$, $k=1$, следователно $k = \log_b a \rightarrow T(n) = O(n^k \log n) = O(n \log n)$.