ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

# Algorithms techniques

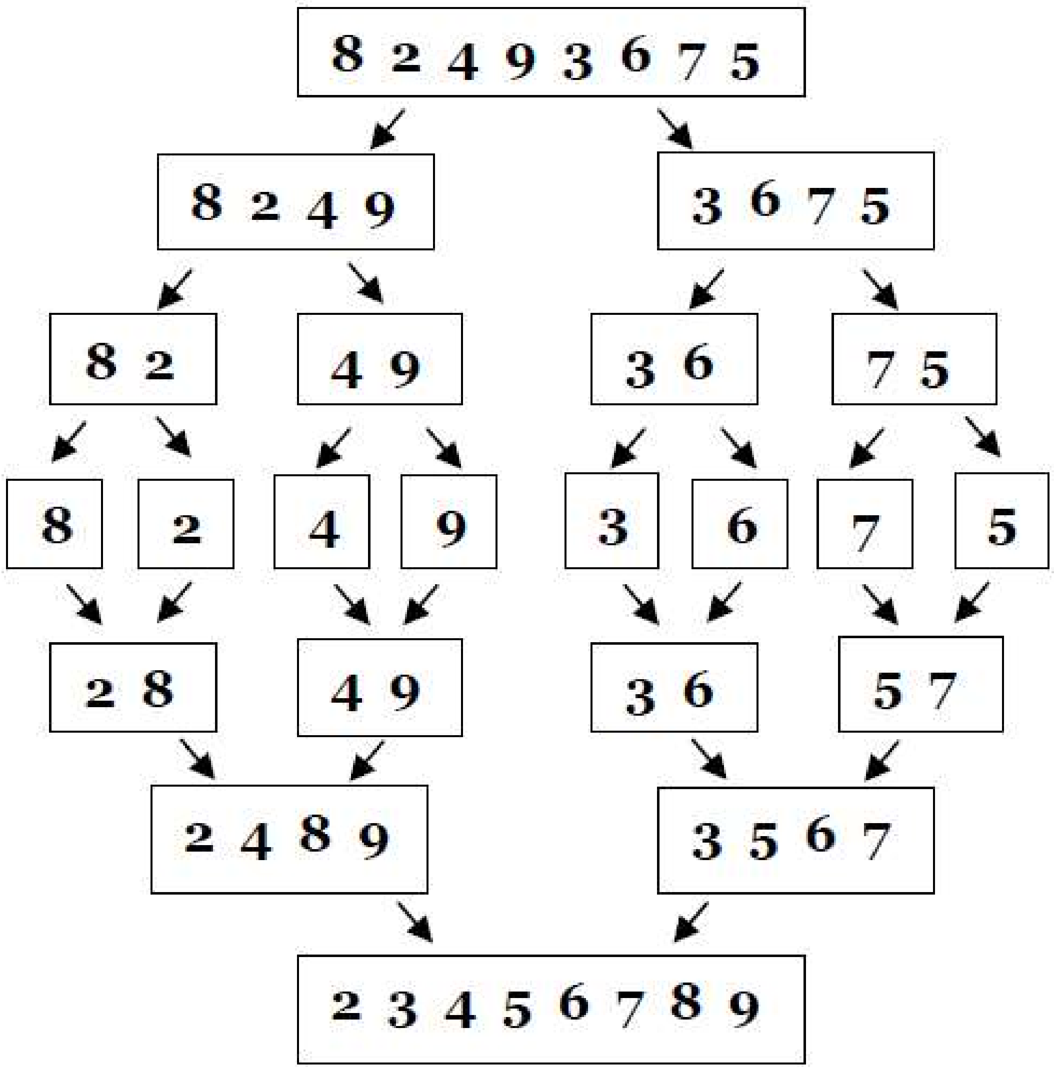
## Algorithms and data structures

## Exercise 4

# Divide and conquer - Merge sort

- “Divide and conquer" algorithm in 3 steps for mergesort of an array [0 … n]:
  - **Divide**: the array a[0..n] is halved into two subarrays
  - **Conquer**: each half a[0 .. (n/2)] and a[(n/2) +1 .. n] is sorted recursively
  - **Merge**: both sorted halves are merged into a sorted array

# Merge sort


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

# Merge sort - Java

```
class DivideAndConquer {
    //spojuvanje na dve sortirani nizi [l, mid], [mid+1, r]
    //rezultatot e nova sortirana niza
    void merge(int a[], int l, int mid, int r) {
        int numel = r - l + 1;
        int temp[] = new int[100]; // nova niza za privremeno cuvanje
                                   // na sortiranite elementi
        int i = l, j = mid+1, k = 0;

        while ((i <= mid) && (j <= r)) {
            if (a[i] < a[j]) {
                temp[k] = a[i];
                i++;
            } else {
                temp[k] = a[j];
                j++;
            }
            k++;
        }
```

# Merge sort - Java

```
    while (i <= mid) {
        temp[k] = a[i];
        i++;
        k++;
    }

    while (j <= r) {
        temp[k] = a[j];
        j++;
        k++;
    }

    for (k = 0; k < numel; k++) {
        a[l + k] = temp[k];
    }
}
```

# Merge sort - Java

```
    void mergesort(int a[], int l, int r) {
        if (l == r) {
            return;
        }

        int mid = (l + r) / 2;
        mergesort(a, l, mid);
        mergesort(a, mid + 1, r);
        merge(a, l, mid, r);
    }
}
```

# Merge sort - Java

```
public static void main(String[] args) {
    int i;

    DivideAndConquer dac = new DivideAndConquer();

    int a[] = new int[]{9, 2, 4, 6, 0, 8, 7, 3, 1, 5};

    dac.mergesort(a, 0, 9);

    for (i = 0; i < 10; i++) {
        System.out.print(a[i] + " ");
    }
    System.out.println();

}
```

# Problem 1

- Write a function that solves the n-th power of a number using “Divide and conquer".

# Problem 1 - Java

```
public static void main(String[] args) {
    int i;

    DivideAndConquer dac = new DivideAndConquer();


    System.out.println("pow: " + dac.pow(2, 10));


}
```

# Problem 1 - Java

```
class DivideAndConquer {

   int pow(int x, int n) {
        int r;

        if(n == 0)
            return(1);
        else if(n % 2 == 0) {
            r = pow(x, (n/2));
            return r*r;
        } else {
            r = pow(x, (n/2));
            return x*r*r;
        }
    }
}
```

# Problem 1 - analysis

- Algorithms complexity analysis:

  $T(n) = T(n/2) + O(1)$

  $T(n/2) = (T(n/4) + 1) \rightarrow$

  $T(n) = T(n/4) + 2$

  $T(n/4) = T(n/8) + 1 \rightarrow$

  $T(n) = T(n/8) + 3$

  …

  $T(n) = T(n/2^k)+k$ and from $k=\log(n)$ it follows: $T(n) = T(1)+\log(n) = \log(n)$

  Hence, the complexity is *O(log n)*

# Problem 1 - analysis

- Solution 2:

```
int pow(int x, int n) {
    int r;

    if(n == 0)
        return(1);
    else if(n % 2 == 0) {
        return pow(x, (n/2))*pow(x,
(n/2));
    } else {
        return x*pow(x, (n/2))*pow(x,
(n/2));
    }
}
```

- What can be concluded?

# Problem 1 (solution 2)- analysis

- Algorithms complexity analysis:

  $T(n) = 2T(n/2)+O(1)$

  $T(n/2) = 2(T(n/4) + 1) \rightarrow$

  $T(n) = 4T(n/4)+2$

  $T(n/4) = 8T(n/8) + 1 \rightarrow$

  $T(n) = 8T(n/8)+3$

  ...

  $T(n) = 2^kT(n/2^k)+k$ and from $k=\log(n)$ it follows: $T(n) = nT(1)+\log(n) = n + \log(n)$

  Hence, the complexity is *O(n)*

# Problem 2.

- Calculate binomial coefficients of a polynomial with power n using the following Pascal's triangle:

$$
\begin{array}{lc}
n = 1 & 1 \\
n = 2 & 1\ 2\ 1 \\
n = 3 & 1\ 3\ 3\ 1 \\
n = 4 & 1\ 4\ 6\ 4\ 1 \\
n = 5 & 1\ 5\ 10\ 10\ 5\ 1
\end{array}
$$

# Problem 2. - analysis

- The binomial coefficient C(n, k) is the number of ways choosing a subset with k elements of a set with n elements
- The mathematical formula is: C(n, k) = n! / (k! (n - k)!)
- The results in between can create an overflow: C(100, 15) = 253338471349988640 can be stored in a 64-bit long, but the binary representation of 100! is 525 bits
- Pascal equation:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

# Problem 2. - analysis

- A naive implementation:

```
// DO NOT RUN THIS CODE FOR LARGE INPUTS
public static long binomial(int n, int k) {
    if (k == 0)
        return 1;
    if (n == 0)
        return 0;
    return binomial(n-1, k) + binomial(n-1, k-1);
}
```

- The same sub-problems are solved repetitively – exponential complexity

# Problem 2. - Java

```
public static void main(String[] args) throws Exception {
        int i, j;
        BufferedReader br = new BufferedReader(new
    InputStreamReader(System.in));

        DP12 dp = new DP12();

        System.out.println(dp.binomial_coefficient(5, 2));


    }
```

# Problem 2. - Java

```
public class DP12 {
int binomial_coefficient(int n, int m) {
    int i, j;
    int bc[][] = new int[n + 1][n + 1];        // tabela so binomni
                                                // koeficienti

    for (i = 0; i <= n; i++)
        bc[i][0] = 1;

    for (j = 1; j <= n; j++)
        bc[j][j] = 1;

    for (i = 1; i <= n; i++)
        for (j = 1; j <= i; j++)
            bc[i][j] = bc[i - 1][j - 1] + bc[i - 1][j];

    return bc[n][m];
}
}
```

# Problem 3.

- *Maximum sum problem*: There is a robot sent to Mars, to collect as much newly discovered stones as possible. The surface of Mars is represented as a table A (m x n), and in each square the number of stones is shown.

| | | |
|---|---|---|
| 1 (старт) | ... | 8 |
| 27 | ... | 1 |
| ... | ... | ... |
| 69 | ... | 10 (крај) |

# Problem 3.

- The robot starts from the upper left corner, and should travel to the lower right corner. The robot can be moved right or down only. Write a program that gives the maximal number of stones the robot can collect.

# Problem 3. - analysis

- As it can be assumed, generating all possible solutions and storing the path with maximal sum is not a good idea, because the number of possible ways increases exponentially according to the table size. (It can be shown that is $\frac{(m+n)!}{m!n!}$ )

# Problem 3. - analysis

- The problem has an optimal substructure:
  - Let the path P be the one with the maximal sum. Then, every part of this path with start at i and end at j contains the maximum number of stones that can be collected between i and j.

# Problem 3. - analysis

- Let B(i, j) be the maximal sum from (1, 1) to (i, j)
- The sum B(m, n) should be calculated
- We check if B(m, n) can be solved recursively
- The recursion is:

  $B(m, n) = \max\{ B(m, n-1), B(m-1, n) \} + A(m, n)$
- The main problem solution is based on solving two sub-problems and their combination

# Problem 3. - analysis

- The trivial problem is finding elements from the first row and the first column of matrix B.
- The sub-problems are not independent. For example, (m-1, n-1) appears at sub-problem (m, n-1) and at (m-1, n). Therefore, for solving this problem dynamic programming can be used.

# Problem 3. - Java

```
public static void main(String[] args) throws Exception {
        int i, j;
        BufferedReader br = new BufferedReader(new
    InputStreamReader(System.in));

        DP12 dp = new DP12();
        System.out.println("Vnesi broj na redici: ");
        int m = Integer.parseInt(br.readLine());
        System.out.println("Vnesi broj na koloni: ");
        int n = Integer.parseInt(br.readLine());

        for (i = 0; i < m; i++) {      // vnesuvanje na broj na kamenja vo
    sekoe pole
            System.out.println("Vnesi ja " +(i+1)+ " redica: ");
            for (j = 0; j < n; j++) {
                dp.a[i][j] = Integer.parseInt(br.readLine());
            }
        }
        dp.maksimalen_zbir(m, n);
        System.out.println("Maksimalniot zbir e " + dp.best[m - 1][n - 1]);
    }
```

# Problem 3. - Java

```
public class DP12 {
int a[][] = new int[100][100];
int best[][] = new int[100][100];

void maksimalen_zbir(int m, int n) {
    int i, j;
    // inicijalizacija na trivijalni reshenija
    best[0][0] = a[0][0];

    for (i = 1; i < m; i++)
        best[i][0] = best[i - 1][0] + a[i][0]; // prva kolona
    for (i = 1; i < n; i++)
        best[0][i] = best[0][i - 1] + a[0][i]; // prva redica

    for (i = 1; i < m; i++)
      for (j = 1; j < n; j++)
        best[i][j] = Math.max(best[i-1][j], best[i][j-1])+a[i][j];
}
}
```

# Problem 3. - variations

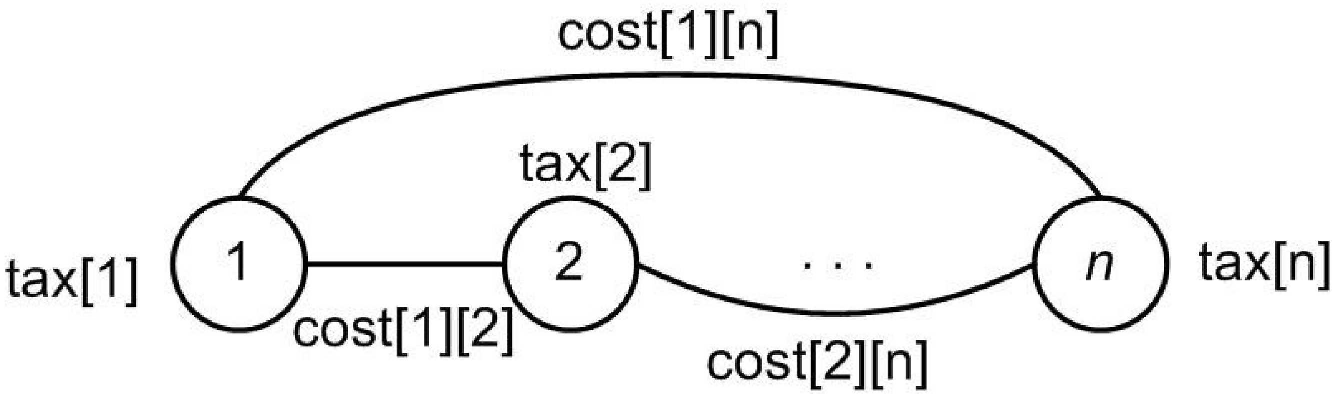
- Variations of the problem for maximal sum:
  - The robot can move diagonally right-down. The recursion is:

    B(m, n) = max{ B(m, n-1), B(m-1, n), B(m-1, n-1) } + A(m, n)
  - If the robot got to a square of the last row, going right-down can teleport to the first row (cylindric map). That means from (m, x) can go to (1, x+1).

# Problem 4.

- *Shortest path problem*: A traveller starts from the first city (at west), and should get to the last city *n* (at east), travelling by a plane. The traveller can not go back, i.e. the only direction allowed is west-east. There is a direct plane flight from each city *i* to every other city *j*. Each city *c* has an airport tax price that should be paid in that city tax[*c*], and every flight from *i* to *j* has a flight cost[*i*][*j*], as shown:

# Problem 4.




- Write a program that calculates the minimal price (sum of the airport taxes and flight costs), for the traveler to get from city 1 to city *n*.

# Problem 4. - analysis

- The number of all possible combinations for the traveler is exponentional (we show that is $2^{n-2}$), and increases with the increase of n.
- If we choose this solution, fast results are only if n<25 (less than a second).

# Problem 4. - analysis

- Let the cheapest total costs from the first city to each of the cities 1, 2, ... , $k$ are known. Using these information can we find the cheapest cost to city $k$+1?
- We should find the city $1 \le c \le k$ from which it is the best to travel directly to city k+1 (the optimal price to $c$ is already calculated)
- The trivial solution to the first city is known, as the starting point, so best[1] = tax[1] (only airport tax should be paid)
- The recursion is:

$$best[k+1] = \min_c\{\ best[c]+cost[c][k+1]+tax[k+1]\ \},\ 1 \le c \le k$$

# Problem 4. - Java

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class DP3 {

    static int cost[][];
    static int tax[], best[];
    static int n;
    static int INFINITY = 1000000;

    static int min(int x, int y) {
        if (x < y)
            return x;
        return y;
    }
```

# Problem 4. - Java

```
public static void main(String[] args) throws Exception {
        int i, j;
        BufferedReader br = new BufferedReader(new
                                InputStreamReader(System.in));

        System.out.println("Vnesi broj na gradovi:");
        n = Integer.parseInt(br.readLine());

        tax = new int[n];
        best = new int[n];
        cost = new int[n][n];

        for(i = 0; i < n; i++) {
            System.out.println("Vnesi aerodromska taksa za gradot "
                                +(i+1)+ " : ");
            tax[i] = Integer.parseInt(br.readLine());
```

# Problem 4. - Java

```
            for(j = i + 1; j < n; j++) {
                System.out.println("Cena na bilet od " +(i+1)+
                                        " do "+(j+1) + " : ");
                cost[i][j] = Integer.parseInt(br.readLine());
            }
        }

        best[0] = tax[0]; // za prviot grad se plakja samo taksa
        for(i = 1; i < n; i++) {
            best[i] = INFINITY; // inicijalizacija

            // go barame opt. grad j, od koj bi patuvale do gradot i
            for(j = 0; j < i; j++)
                best[i] = min(best[i], best[j] + cost[j][i] + tax[i]);
        }

        System.out.println("Najmala cena e "+best[n-1]);
    }
}
```

# Problem 5.

- *0-1 knapsack problem*: Given $n$ objects $O=\{o_1, o_2, o_3, \dots, o_n\}$ and a knapsack. Each object $o_i$ has weight $t_i$, and the knapsack is with capacity $C$. If the object $o_i$ is in the knapsack, then we have profit $p_i$. The aim is to fill the knapsack so that the profit is maximized. The capacity of the knapsack is $C$, and the objects can not be divided.

# Problem 5. - analysis

- The elements indexes form a vector. Let *i* be the index of the last object from the vector of the optimal solution *S* for capacity *C*
- Then $S' = S - \{o_i\}$ is the optimal solution of a sub-problem for a knapsack with capacity $C - t_i$, while the profit of the solution *S* is $p_i$ + the profit of the sub-problem

# Problem 5. - analysis

- Let D[*i*][*j*] be the maximal profit for the objects set 1, 2, 3, ..., *i* with capacity of the knapsack *j*. Then:

```
if (j > t[i])
   D[i][j] = max(p[i] + D[i-1][j-t[i]], D[i-1][j])
else
   D[i][j] = D[i-1][j]
```

- This shows that the solution for *i* objects includes the *i*-th object if higher profit is achieved, or does not include *i*-th object, which is a solution for a sub-problem for *i* - 1 objects and a knapsack with the same capacity *j*.

# Problem 5. - analysis

- Example:
  - $C$=50
  - $(p_1, p_2, p_3)$ = (60, 100, 120)
  - $(t_1, t_2, t_3)$ = (10, 20, 30)
- The dynamic programming matrix is:

| i \ j | 0 | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 60 | 60 | 60 | 60 | 60 |
| 2 | 0 | 60 | 100 | 160 | 160 | 160 |
| 3 | 0 | 60 | 100 | 160 | 180 | 220 |

# Problem 5. - Java

```
public class DP5 {

    int max(int a, int b) {
        if (a > b) {
            return a;
        }
        return b;
    }

    public static void main(String[] args) throws Exception {
        DP5 dp = new DP5();

        int n = 3;
        int C = 50;
        int p[] = new int[]{60, 100, 120};
        int t[] = new int[]{10, 20, 30};

        System.out.println(dp.DPKnapsack(t, p, C));
    }
```

# Problem 5. - Java

```
int DPKnapsack(int t[], int p[], int C) {
    int i, j;
    int n = t.length;
    int D[][] = new int[n + 1][C + 1];

    for (j = 0; j <= C; j++) {
        D[0][j] = 0;
    }

    for (i = 1; i <= n; i++) {
        D[i][0] = 0;
    }

    for (i = 1; i <= n; i++)
        for (j = 1; j <= C; j++)
            if (t[i - 1] <= j)
                D[i][j] = max(p[i-1] + D[i-1][j - t[i-1]], D[i-1][j]);
            else
                D[i][j] = D[i - 1][j];

    return D[n][C];
}
}
```

# Problem 5. - discussion

- How can we get a set of objects in the knapsack with the highest profit?
  - Going back should discover the optimal values
  - If D[$i$][$p$] = D[$i$-1][$j$] the object $i$ does not belong to the solution, and we continue to search with D[$i$-1][$j$]
  - On the other hand, the object $i$ belongs to the solution, so we continue with D[$i$-1][$C$-$j_i$]

# Problem 5. - discussion

- The optimal objects choice for the example is:
  - $S = \{o_2, o_3\}$

| i \ j | 0 | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 60 | 60 | 60 | 60 | 60 |
| 2 | 0 | 60 | 100 | 160 | 160 | 160 |
| 3 | 0 | 60 | 100 | 160 | 180 | 220 |

- 0-1 knapsack algorithm has $\Theta(nC)$ time complexity

# Problem 5. – (homework)

- HOMEWORK. Write code that will return an array with the objects indexes which are in the optimal solution of the 0-1 knapsack problem.

# Problem 6. (homework)

- Write an algorithm that finds the longest “common" sub-sequence of two strings x[] and y[] and will return its lenght.
- Example
  - The longest “common" subsequence of the strings: **ggca**cc**acg** and ac**ggc**gg**a**t**acg** is **ggcaacg**.

# Problem 6. - analysis

- Let NZP[i][j] be the length of the longest “common" subsequence of the sub-strings x[0] … x[i-1] and y[0] … y[j-1]
- The recursion formula for the solution NZP[i][j] is:

$$NZP[i][j] = \begin{cases} 0, \text{ако } i = 0 \text{ или } j = 0 \\ NZP[i-1][j-1] + 1, \text{ако } x[i] = y[j] \\ \max(NZP[i, j-1], NZP[i-1][j]), \text{ако } x[i] \neq y[j] \end{cases}$$

# Problem 6. - Java

```
class NajgolemaPodniza{

int max(int a, int b) {
    if (a > b)
        return a;
    return b;
}
```

# Problem 6. - Java

```
int najdolgaZaednickaPodsekvencaDolzina(String x, String y) {
    int i, j;
    int N = x.length();
    int M = y.length();

    int NZP[][] = new int[N + 1][M + 1];

    for (i = 0; i < N; i++) {
        NZP[i][0] = 0;
    for (j = 0; j < M; j++) {
        NZP[0][j] = 0;

    for (i = 1; i <= N; i++)
        for (j = 1; j <= M; j++)
            if (x.charAt(i - 1) == y.charAt(j - 1))
                NZP[i][j] = NZP[i - 1][j - 1] + 1;
            else
                NZP[i][j] = max(NZP[i - 1][j], NZP[i][j - 1]);

        return NZP[N][M];
    }
```

# Problem 6. - Java

```
String najdolgaZaednickaPodsekvencaString(String x, String y) {
    int i, j;
    int N = x.length();
    int M = y.length();

    int NZP[][] = new int[N + 1][M + 1];

    for (i = 0; i < N; i++)
        NZP[i][0] = 0;

    for (j = 0; j < M; j++)
        NZP[0][j] = 0;

    for (i = 1; i <= N; i++)
        for (j = 1; j <= M; j++)
            if (x.charAt(i - 1) == y.charAt(j - 1))
                NZP[i][j] = NZP[i - 1][j - 1] + 1;
            else
                NZP[i][j] = max(NZP[i - 1][j], NZP[i][j - 1]);

    char rez1[] = new char[max(N, M)];
    int L = 0;
```

# Problem 6. - Java

```
    i = N;
    j = M;

    while ((i != 0) && (j != 0)) {
        if (x.charAt(i - 1) == y.charAt(j - 1)) {
            rez1[L] = x.charAt(i - 1);
            L++;
            i--;
            j--;
        } else {
            if (NZP[i][j] == NZP[i - 1][j])
                i--;
            else
                j--;
        }
    }

    String rez2 = "";
    for (i = 0; i < L; i++)
        rez2 += rez1[L - 1 - i];

    return rez2;
}
```

# Problem 6. - Java

```
public static void main(String[] args) throws Exception {

    DP6 dp = new DP6();

    String x = "ggcaccacg";
    String y = "acggcggatacg";

    System.out.println(dp.najdolgaZaednickaPodsekvencaDolzina(x, y));
    System.out.println(dp.najdolgaZaednickaPodsekvencaString(x, y));

}
```