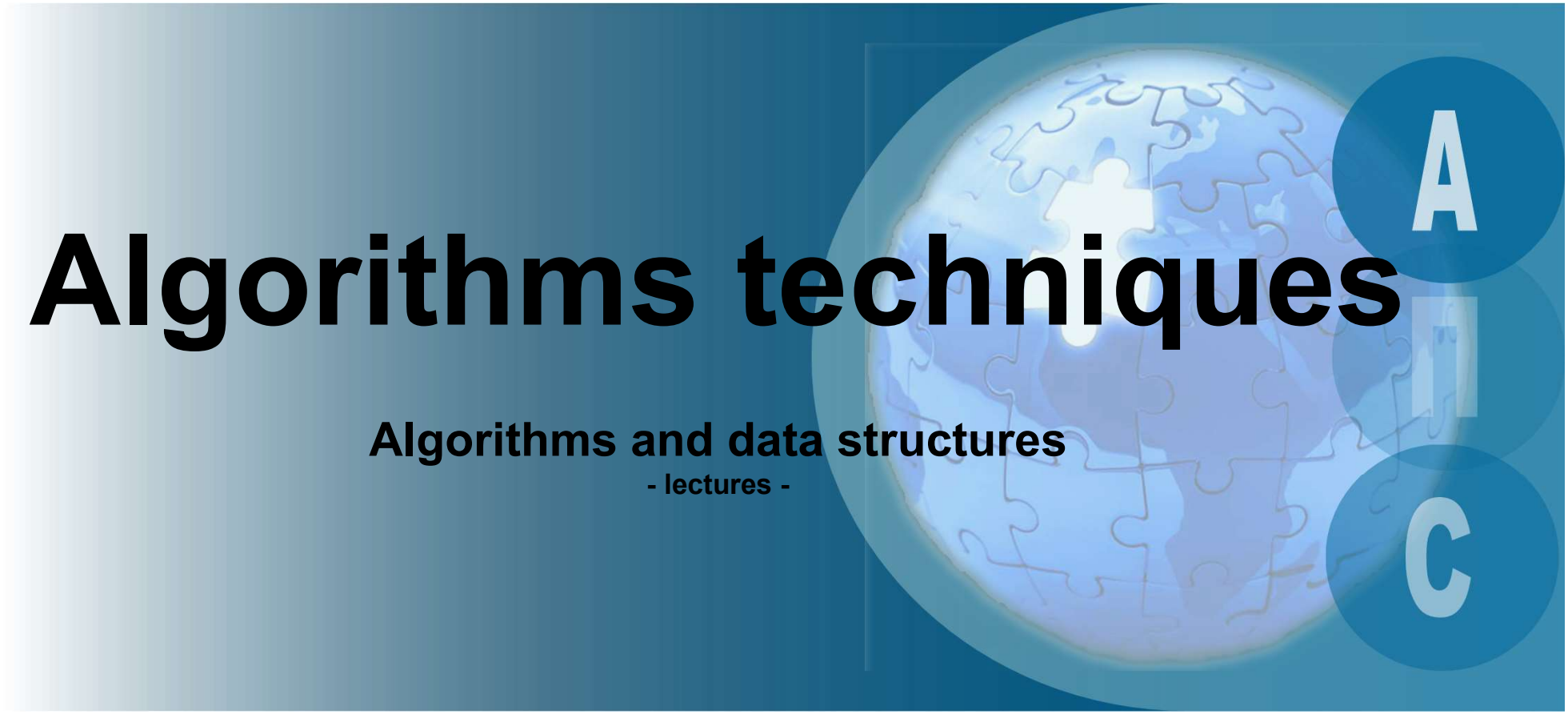
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО




# Algorithms techniques

## Algorithms and data structures

- lectures -

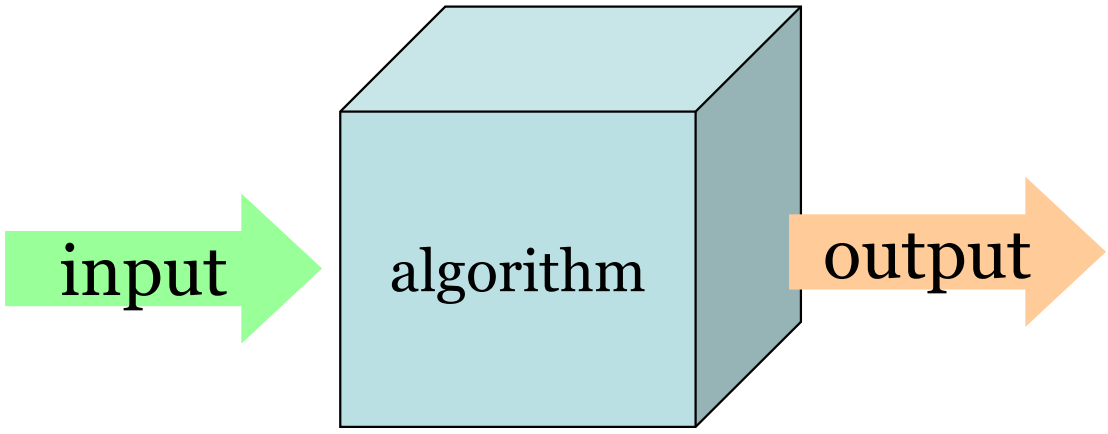
# Outline

- Introduction in computational problems
- Problems description ways
- Pseudocode notation
- Algorithms design techniques
  - **Brute force technique**
  - **Greedy algorithms**
  - Divide and conquer
  - Dynamical programming
  - Random numbers algorithms
  - Rest algorithms

# What is an algorithm?

- **Algorithm** is a sequence of steps/instructions that have to be taken to solve a well defined *problem*

- **Algorithm** is the method for translation of data input in corresponding output



# Computational problems

- The first and the hardest step in solving a computational problem is – defining the problem.

- Two basic questions have to be always in mind:
  - “Does the algorithm work correctly?”
  - “How much time is needed for its execution?”

# Computational problems

- One problem can describe a whole class of computational problems.
- Problem *instance* is one concrete input for the given problem.
- **Example:**
  You buy milk that costs 51 denars with a 100 denars bill and a 49 denars change needs to be given back.
  - Problem: Giving back change.
  - Problem instance: Giving back 49 denars change.

# How one algorithm is constructed?

- Algorithms are consisted of steps that need to be executed
  - Descriptively specifying the steps in a spoken language (easy to understand for a human, but not for a computer)
  - Describing through pseudocode (insufficiently comprehensible to the computer, but abstract enough to be translated into program code)
  - A graphical or visual representation of the steps using a diagram (usually a block diagram)

# Pseudocode description

- ***Pseudocode*** is the most common way for algorithms description
  - Variables, Arrays, and Arguments
  - A special ***return*** operation returns the result that ends the execution of the algorithm
  - Combining basic operations into mini-algorithms called ***sub-algorithms*** (*subroutines*)

# Elementary pseudocode commands

- Value assignment
  - Format: $a \leftarrow b$
  - Effect: The variable $a$ takes the value of $b$
  - Example: $b \leftarrow 2$ $a \leftarrow b$
  - Result: The value of variable $a$ is 2

# Elementary pseudocode commands

- Arithmetic operations

  - Format: $a + b,\ a - b,\ a * b,\ a / b,\ a^b$
  - Effect: Addition, subtraction, multiplication, division and power
  - Example:

```
DIST(x1, y1, x2, y2)
1  dx ← (x2 – x1)²
2  dy ← (y2 – y1)²
3  return sqrt(dx + dy)
```

  - Result: DIST(x1, y1, x2, y2) computes the Euclidean distance between 2 points.
    DIST(0,0,3,4) returns 5

# Elementary pseudocode commands

## Condition check

- Format: **if** *A* is true B **else** C
- Effect: If condition A holds then instructions B are executed, otherwise instructions C are executed
- Example:

```
MAX(a, b)
1   if a<b  return b
2   else  return a
```

- Result: MAX(a, b) returns the bigger number from a and b.
  MAX(1, 99) враќа 99

# Elementary pseudocode commands

- for cycles
  - Format: **for** *i* ← *a* **to** *b* B
  - Effect: The value of *i* is set to *a* and B is executed. Then *I* gets the value *a+1* and again B is executed. This repeats until *i* gets value *b*.
  - Example:

```
SUMINT(n)
1  sum ← 0
2  for i ← 1 to n   sum ← sum + i
3  return sum
```

  - Result: SUMINT(n) computes the sum of the numbers from 1 to n. SUMINT(10) returns 55

# Elementary pseudocode commands

- while cycles
  - Format: **while** *A* is true B
  - Effect: The condition *A* is checked. If it holds *B* is executed. The *A* holding check repeats and if it is true, then *B* is executed again until *A* is false.
  - Example:

```
ADDUNTIL(b)
1   i ← 1,  total ← i
2   while total ≤ b
3       i ← i + 1
4       total ← total + i
5   return i
```

  - Result: ADDUNTIL(b) finds the smallest number i for which it holds that the sum from 1 to that number is greater than b. ADDUNTIL(25) returns 7

# Elementary pseudocode commands

## Array elements access

- Format: $\mathbf{a_i}$
- Effect: Returns *i*-th element of an array defined with $\mathbf{a} = (a_1, \ldots, a_i, \ldots, a_n)$
- Example:

```
FIBONACCI(n)
1   F1 ← 1,   F2 ← 1
2   for i ← 3 to n
3       Fi ← Fi-1 + Fi-2
4   return Fn
```

- Result: FIBONACCI(n) computes the n-the Fibonacci number. FIBONACCI(8) returns 21

# Example - recipe

Analogy between recipe and algorithm

1. input data
2. execution steps
3. output, result

*1 ½ cup boiled pumpkin*
*1 cup brown sugar*
*1 tea spoon salt*
*2 tea spoons cinnamon*
*1 tea spoon ginger*
*1 table spoon molasses*
*3 eggs, slightly mixed*
*1 ½ cup homogenized milk*
*1 unbaked pie dough*

1

3

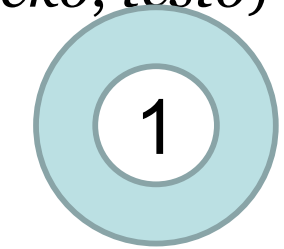
Mix the pumpkin, sugar, salt, ginger, cinnamon and molasses. Add the eggs and milk and mix well. Put the mixture in the unbaked pie dough and bake in a preheated oven (220 °C) for about 40 to 45 minutes or until the knife used to cut the pie comes ou

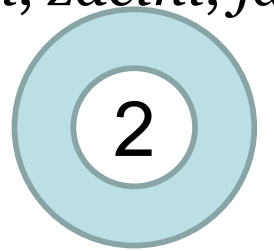
2

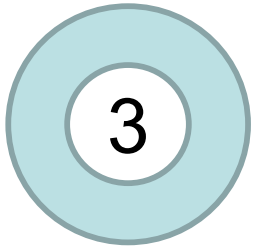
# Example – recipe, pseudocode

NAPRAVI_PITA_OD_TIKVA(*tikva*, *seker*, *sol*, *zacini*, *jajca*, *mleko*, *testo*)
1 ZAGREJ_RERNA(220)
2 *polnenje* ← ZAMESAJ(*tikva*, *seker*, *sol*, *zacini*, *jajca*, *mleko*)
3 *pita* ← SOSTAVI(*testo*, *polnenje*)
4 **while** noz za zasekuvanje ostane cist
5 PECI(*pita*)

6 **output** "pitata e gotova"
7 **return** *pita*




*) NAPRAVI_PITA_OD_TIKVA calls subalgorithm ZAMESAJ

# Example, pseudocode

ZAMESAJ(*tikva*, *seker*, *sol*, *zacini*, *jajca*, *mleko*)
1 *dlabokaCinija* ← zemi dlaboka cinija od plakarot
2 STAVI(*tikva*, *dlabokaCinija*)
3 STAVI(*seker*, *dlabokaCinija*)
4 STAVI(*sol*, *dlabokaCinija*)
5 STAVI(*zacini*, *dlabokaCinija*)
6 MESAJ(*dlabokaCinija*)
7 STAVI(*jajca*, *dlabokaCinija*)
8 STAVI(*mleko*, *dlabokaCinija*)
9 MESAJ(*dlabokaCinija*)
10 *polnenje* ← Sodrzina od *dlabokaCinija*
11 **return** *polnenje*

**Homework:**
Give a graphical representation of this algorithm!

This algorithm does not make a pumpkin pie, but rather **defines the steps** that need to be performed to make the pie.

In order to make a pumpkin pie, a machine must be constructed to follow these steps to produce the final result!

# From pseudocode to implementation

- **Pseudocode**: an abstract sequence of steps that describes a solution to a well-formulated computational problem
- **Computer code**: a set of detailed instructions that can be executed by a computer

**Converting pseudocode to computer code is not a simple process!**

# Change return problem

- Change of 268 denars should be returned if we have at our disposal an unlimited number of banknotes and coins of 1000, 500, 100, 50, 10, 5, 2 and 1 denar. The payment should be with a minimum number of banknotes and coins!

# Change return problem

VRATI_KUSUR(M)
1Return the whole part of the division M/1000
2Let *residual* be the remainder to be returned to the buyer
3Return the whole part of dividing remainder/500
4Let *residual* be the remainder to be returned to the buyer
5Return the whole part of dividing remainder/100
6Let *residual* be the remainder to be returned to the buyer
7Return the whole part of dividing remainder/50
8Let *residual* be the remainder to be returned to the buyer
9Return the whole part of dividing remainder/10
10Let *residual* be the remainder to be returned to the buyer
11Return the whole part of the division remainder/5
12Let *residual* be the remainder to be returned to the buyer
13Return the whole part of division remainder/2
14Let *residual* be the remainder to be returned to the buyer
15Return the whole part of division remainder/1

An intuitive solution
- the most common approach
- what is the technique?

What will this algorithm return if it needs to return 40 denar change with 25, 20, 10, 5 and 1 denar coins?

The same pseudocode, but written in a way closer to the implementation form

VRATI_KUSUR(M)
$1q \leftarrow M$
$2r \leftarrow q/1000$
$3q \leftarrow q - 1000 \cdot r$
$4s \leftarrow q/500$
$5q \leftarrow q - 500 \cdot s$
$6t \leftarrow q/100$
$7q \leftarrow q - 100 \cdot t$
$8u \leftarrow q/50$
$9q \leftarrow q - 50 \cdot u$
$10v \leftarrow q/10$
$11q \leftarrow q - 10 \cdot v$
$12w \leftarrow q/5$
$13q \leftarrow q - 5 \cdot w$
$14x \leftarrow q/2$
$15q \leftarrow q - 2 \cdot x$
$16y \leftarrow q$
**17return** $(r, s, t, u, v, w, x, y)$

# Архетипи на алгорими

Archetypes – techniques of solving algorithms

- brute force technique
- greedy (maximalist)
- divide and conquer
- dynamic programming
- random number algorithms
- backtracking algorithms

- distributed processing algorithms
- algorithms that work with constraints
- other algorithms...

# Brute force technique

- The simplest problems solving technique
- It examines all possible cases through which the result is arrived at
- Guarantees to find a result

**Disadvantage:** Too much time and memory resources to run these algorithms!

# Brute force technique

- Brute force solution to the return change problem:

What will this algorithm return if it needs to return 40 denar change with 25, 20, 10, 5 and 1 denar coins?

BF_CHANGE($M$, **c**, $d$)
1 *smallestNumberOfCoins* $\leftarrow \infty$
2 **for each** $(i_1, \ldots, i_d)$ **from** $(0, \ldots, 0)$ **to** $(M/c_1, \ldots, M/c_d)$
3  *valueOfCoins* $\leftarrow \sum_{k=1} i_k c_k$
4  **if** *valueOfCoins* = $M$
5  *numberOfCoins* $\leftarrow \sum_{k=1} i_k$
6  **if** *numberOfCoins* < *smallestNumberOfCoins*
7  *smallestNumberOfCoins* $\leftarrow$ *numberOfCoins*
8  ***bestChange*** $\leftarrow (i_1, i_2, \ldots, i_d)$
9 **return**(***bestChange***)

# Brute force technique

| | | | | |
|---|---|---|---|---|
| 0 | 0 | … | 0 | 0 |
| 0 | 0 | … | 0 | 1 |
| 0 | 0 | … | 0 | 2 |
| | | … | | |
| 0 | 0 | … | 0 | $M/c_d$ |
| | | … | | |
| 0 | 0 | … | 1 | 0 |
| 0 | 0 | … | 1 | 1 |
| 0 | 0 | … | 1 | 2 |
| | | … | | |
| 0 | 0 | … | 1 | $M/c_d$ |
| | | … | | |
| $M/c_1$ | $M/c_2$ | … | $M/c_{d-1} - 1$ | 0 |
| $M/c_1$ | $M/c_2$ | … | $M/c_{d-1} - 1$ | 1 |
| $M/c_1$ | $M/c_2$ | … | $M/c_{d-1} - 1$ | 2 |
| | | … | | |
| $M/c_1$ | $M/c_2$ | … | $M/c_{d-1} - 1$ | $M/c_d$ |
| $M/c_1$ | $M/c_2$ | … | $M/c_{d-1}$ | 0 |
| $M/c_1$ | $M/c_2$ | … | $M/c_{d-1}$ | 1 |
| $M/c_1$ | $M/c_2$ | … | $M/c_{d-1}$ | 2 |
| | | … | | |
| $M/c_1$ | $M/c_2$ | … | $M/c_{d-1}$ | $M/c_d$ |

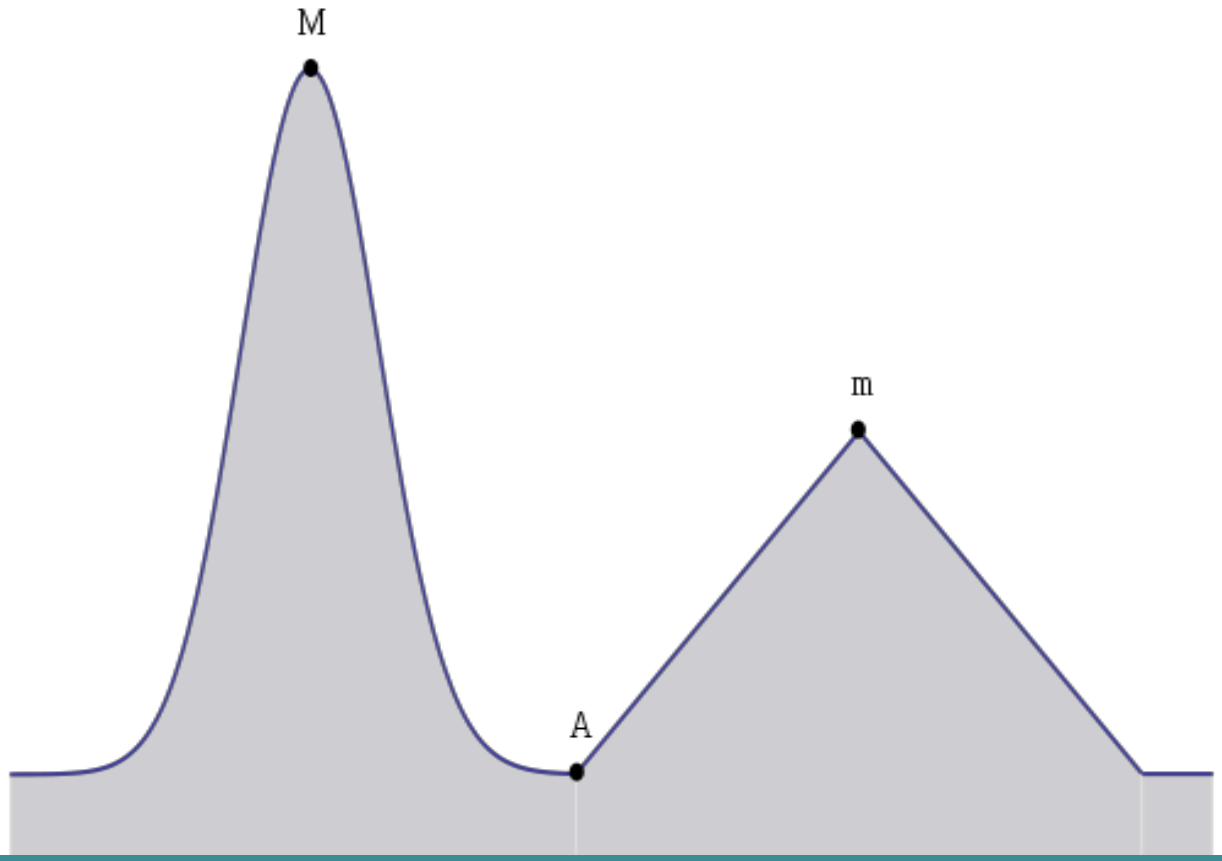
**This technique guarantees that the right correct solution will always be obtained!**

Display of all possible combinations that will be checked during the execution of the algorithm, before reaching the solution

# Greedy algorithms

- they always seek the **local optimal** solution
- it chooses the best that is available at the moment
- they usually create a correct solution, but only for part of the problem domain
- work relatively quickly
- give approximate results
  - obtaining an optimal result is not guaranteed

# Greedy algorithms




Finding two local maxima if starting to the left of point A or to the right of point A

# Greedy algorithms

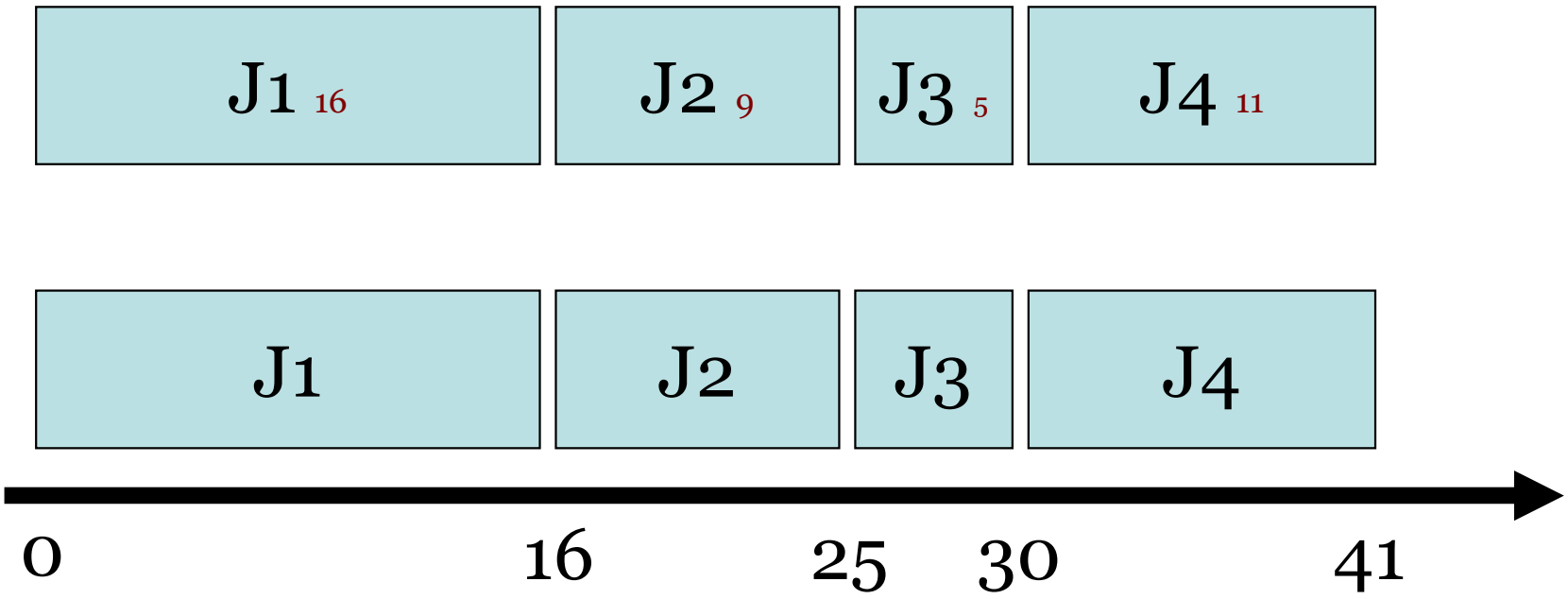
- **Problem**: Algorithm for returning change (calculation of the optimal number of banknotes and coins to be returned)

The first pseudocode we proposed for this problem **belongs to the class of greedy algorithms...**

# Greedy algorithms

- **Problem**: operating system schedule
  - Let the processes (jobs) $j_1, j_2, \ldots, j_n$, be given, with execution times $t_1, t_2, \ldots, t_n$, respectively
  - Processes should be executed on a single processor
  - How can they be arranged to obtain the best average time to complete them if a concurrent request for their execution is received?
  - Assume that once started, the process must not be stopped

# Greedy algorithms

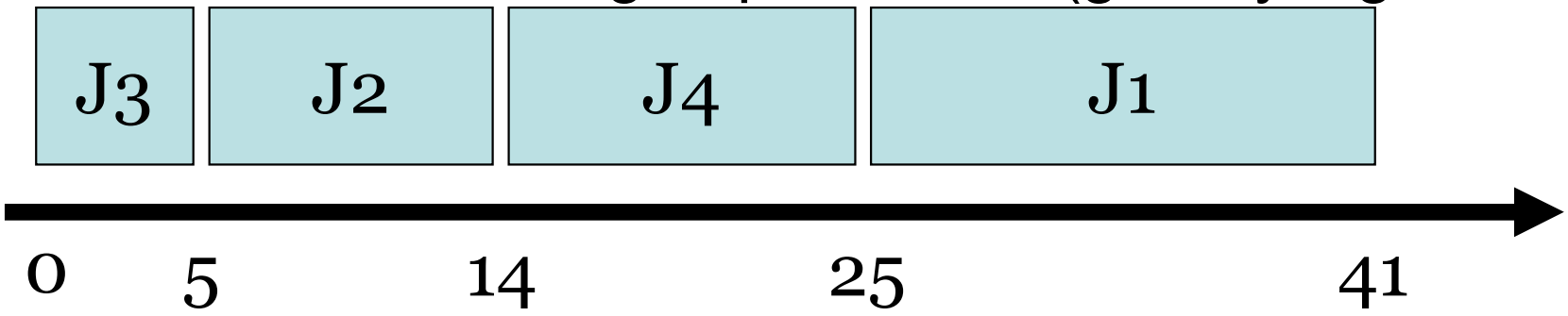



**Average time to complete the processes**

**(16+25+30+41)/4=28**

# Greedy algorithms

Different scheduling of processes (greedy algorithm)




**Average time to complete the processes**

**(5+14+25+41)/4=21,25**

# Explanation

- For a given order of execution of processes

$$j_{i1}, j_{i2}, \ldots, j_{in},$$

it holds that

- The first process $j_{i1}$ ends for $t_{i1}$
- The second process $j_{i2}$ ends for $t_{i1} + t_{i2}$
- The third process $j_{i3}$ ends for $t_{i1} + t_{i2} + t_{i3}$
- ...

# Let's generalize...

$$C = \sum_{k=1}^{n} (n - k + 1) t_{ik}$$

C – cost

- For the previous example, the total time to complete the processes:

$t_1 + t_1 + t_2 + t_1 + t_2 + t_3 + t_1 + t_2 + t_3 + t_4 = 4\,t_1 + 3\,t_2 + 2\,t_3 + t_4$

According to the formula

- n=4

k=1, (4-1+1) $t_1 = 4\,t_1$

k=2, (4-2+1) $t_2 = 3\,t_2$ $\rightarrow$ $C = \sum 4t_1 + 3t_2 + 2t_3 + t_4$

k=3, (4-3+1) $t_3 = 2\,t_3$

k=4, (4-4+1) $t_4 = t_4$

# Greedy algorithms

- The basic idea when building this schedule is to enable the processes to start faster, i.e. their waiting time is shorter

$$C = \sum_{k=1}^{n} (n-k+1)t_{ik} \quad \Rightarrow \quad C = (n+1)\sum_{k=1}^{n} t_{ik} - \sum_{k=1}^{n} k \cdot t_{ik}$$

k – process execution sequence number

**Conclusion:** it is best to multiply the largest *k* by the largest duration of the processes. That's why the "shortest" processes are executed first, and the "longest" processes - last!

# Greedy algorithms

- **Problem**: Huffman codes (1952)
  - We have a text file composed of characters encoded in the standard ASCII set
  - There are 128 symbols in total
  - Each symbol requires 7 bits to represent
  - A parity check bit is added to these seven bits
  - If we have S symbols that need to be encoded in this standard way, $\lceil \log_2 C \rceil$ bits are needed

# Greedy algorithms

| симбол | код | честост | вкупно бити |
|---|---|---|---|
| *а* | 000 | 10 | 30 |
| *е* | 001 | 15 | 45 |
| *и* | 010 | 12 | 36 |
| *с* | 011 | 3 | 9 |
| *т* | 100 | 4 | 12 |
| *празно* | 101 | 13 | 39 |
| *ентер* | 110 | 1 | 3 |

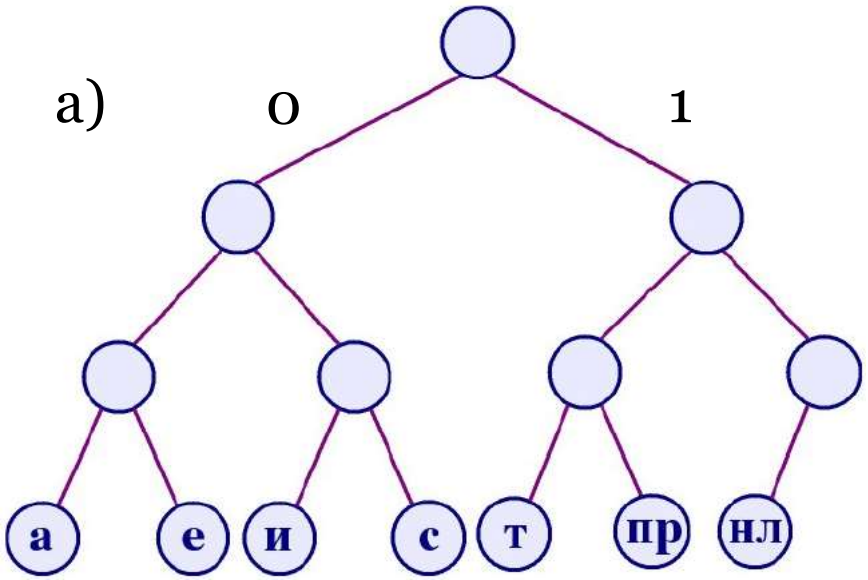
Total: 174 bits

**Note:** in text files, vowels occur more often than consonants!

If the symbol codes are of variable length and the frequent symbol codes are short, 25% to 60% compression can be obtained

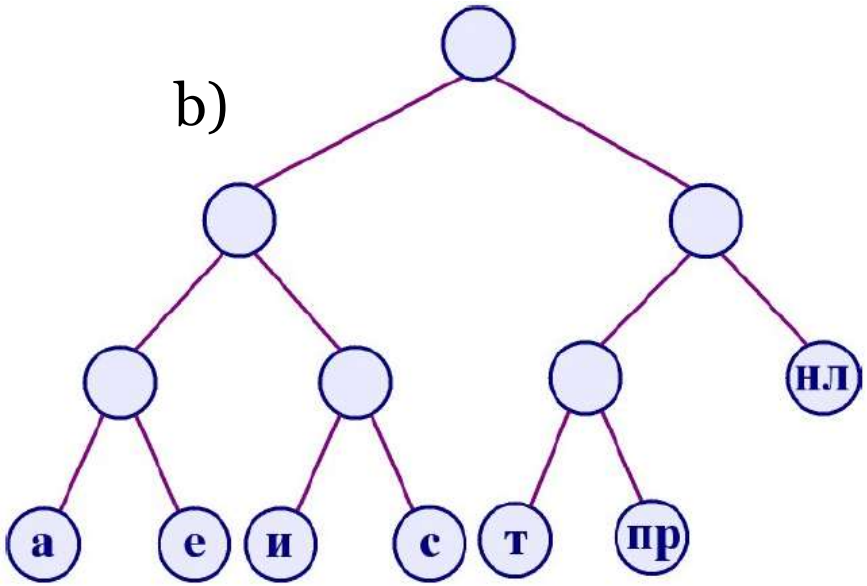
# Greedy algorithms

a)




| | |
|---|---|
| а | 000 |
| е | 001 |
| и | 010 |
| с | 011 |
| т | 100 |
| empty | 101 |
| new line | 110 |

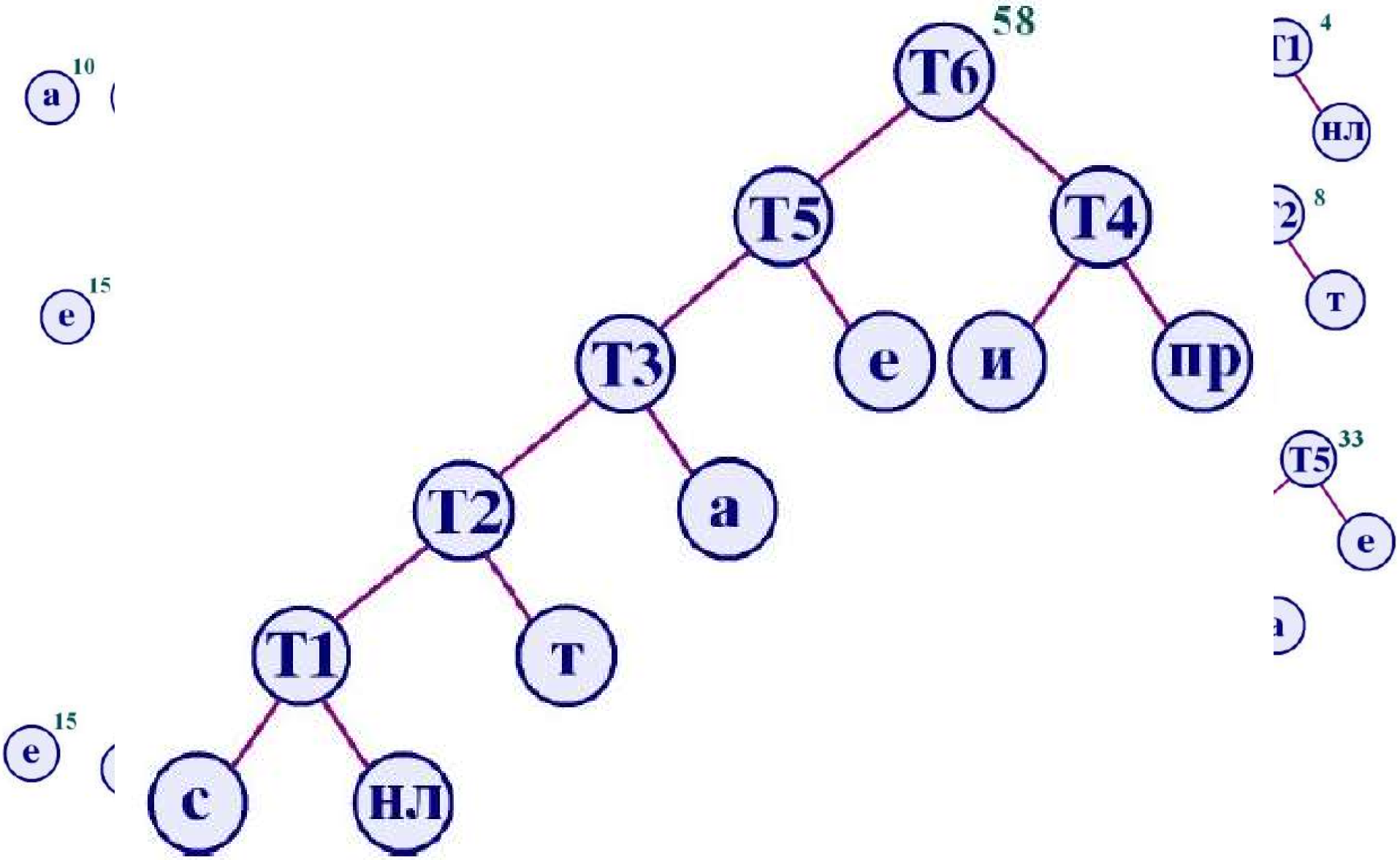
b)




| | |
|---|---|
| а | 000 |
| е | 001 |
| и | 010 |
| с | 011 |
| т | 100 |
| empty | 101 |
| new line | 11 |

What is encoded in this file (b): 0100111100010110001000111?

# Greedy algorithms



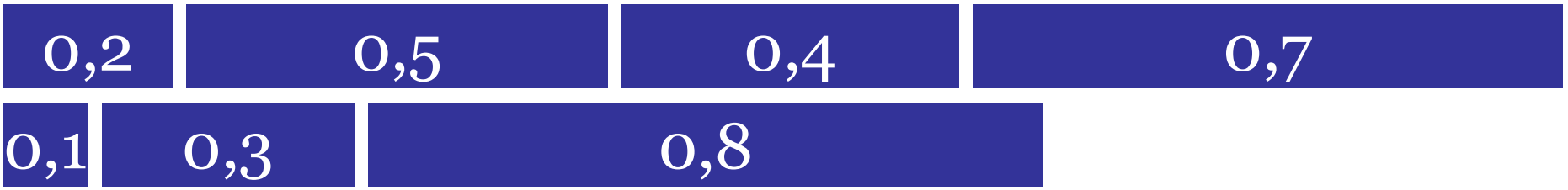
# Greedy algorithms

- In each step of the algorithm, the smallest set of weights is selected
- Coding using this algorithm is realized in two stages:
  - determining the symbols used and their frequency and
  - creating the coding tree
- The codes for all symbols should be given at the beginning of the file
- The encoding principle may not be efficient for large files

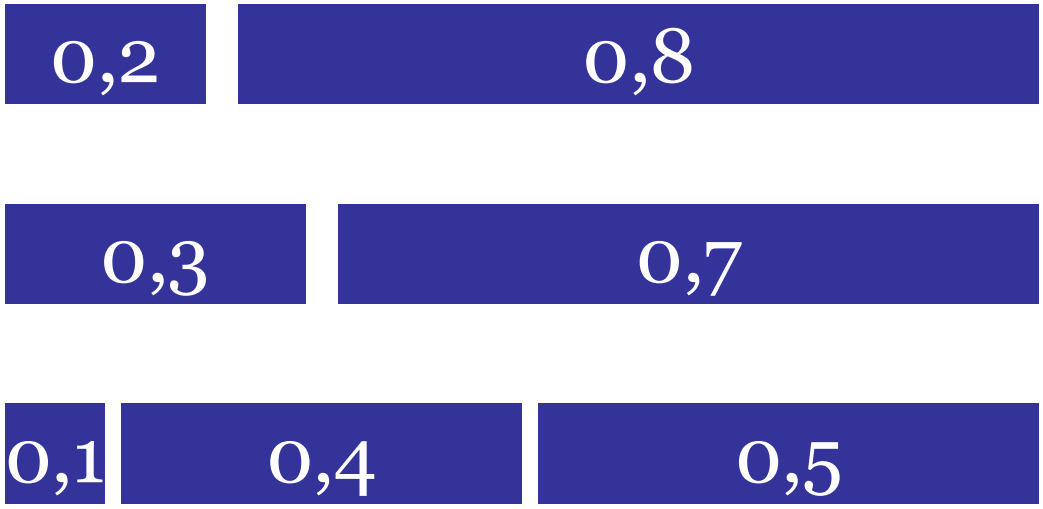
# Greedy algorithms

- **Problem**: knapsack packing (Knapsack algorithm)
  - Let $n$ packages of sizes $s_1$, $s_2$, ..., $s_n$, $0 < s_i \leq 1$ be given
  - The packing problem requires the optimal packing of packages into knapsacks of size 1 (The smallest number of knapsacks should be used)
  - Often this problem is also called the problem of the thief who had bags that could carry X kilograms of gold and gold objects of different weights

# Greedy algorithms




**Problem:** How many size 1 packages are needed to pack these packages?




# Greedy algorithms

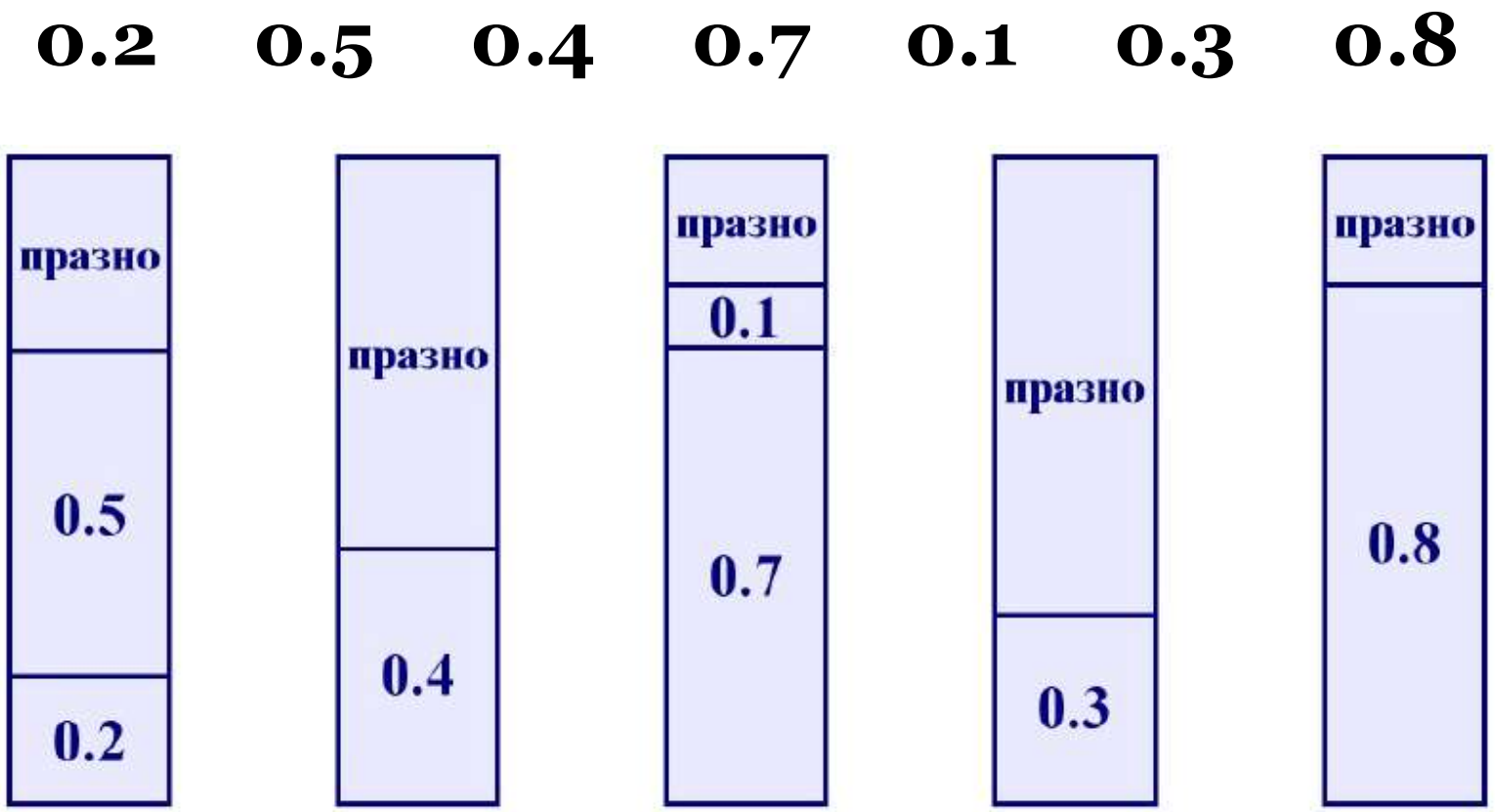
- There are two versions of the algorithm:
  - **First version**: the next package size in the sequence is not known until the current package is placed in the knapsack
  - **Second version**: all packages sizes are known in advance

# Greedy algorithms

- Next fit solution:
  - The first package is placed in the first knapsack
  - When placing each next package, it is checked whether there is a place in the knapsack where the previous package was placed
  - If there is no room, the next knapsack is checked and so on until the first knapsack with enough room to accommodate the package is found.

This algorithm obviously works in linear time, but does not always give optimal results

# Greedy algorithms




In the worst case, this algorithm uses twice as many knapsacks as the optimal solution

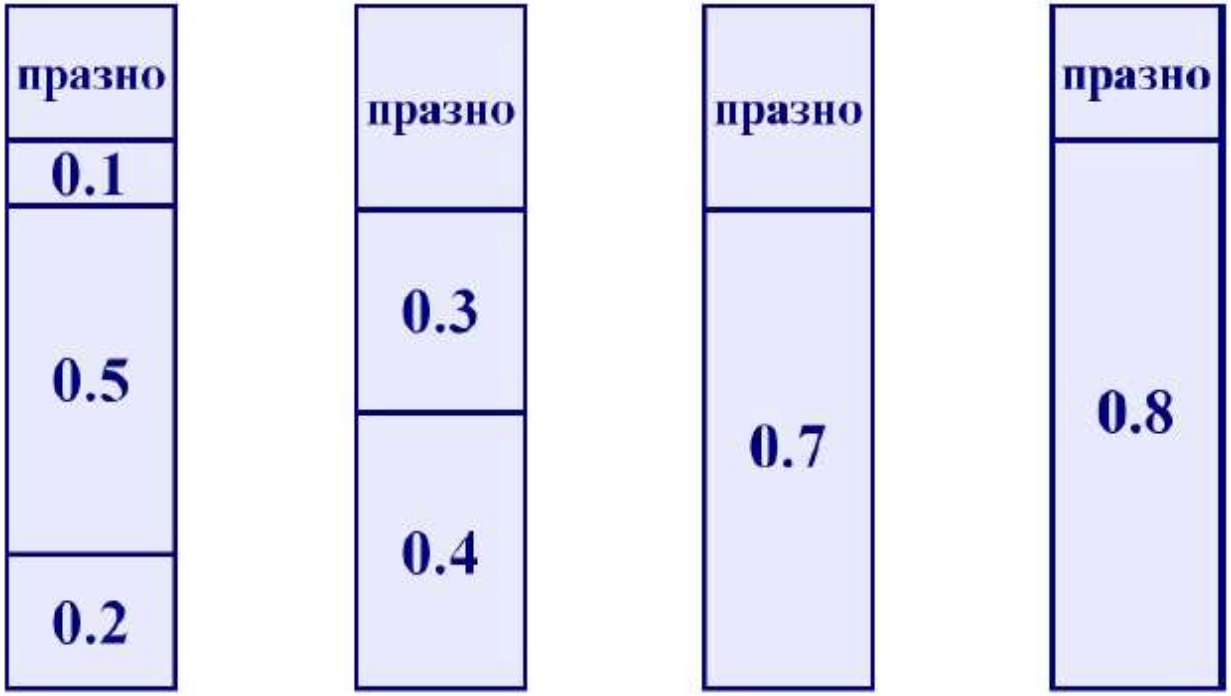
# Greedy algorithms

- First fit solution:
  - The first package is placed in the first knapsack
  - For each subsequent package, all knapsacks are checked from the beginning, and the package is placed in the first knapsack that has room to accommodate it
  - A new knapsack is used only if there is no space in the previously started knapsacks

This algorithm can be implemented in the best case with O(nlogn) complexity, and in the general case $O(n^2)$

# Greedy algorithms

0.2 0.5 0.4 0.7 0.1 0.3 0.8



# Greedy algorithms

- Best fit solution:
  - The first package is placed in the first knapsack
  - For each subsequent package, all knapsacks are checked from the beginning, and the package is placed in the knapsack that has the smallest space sufficient to fit the package of a given size
  - A new knapsack is used only if there is no space in the previously started knapsacks

# Greedy algorithms

0.2 0.5 0.4 0.7 0.1 0.3 0.8