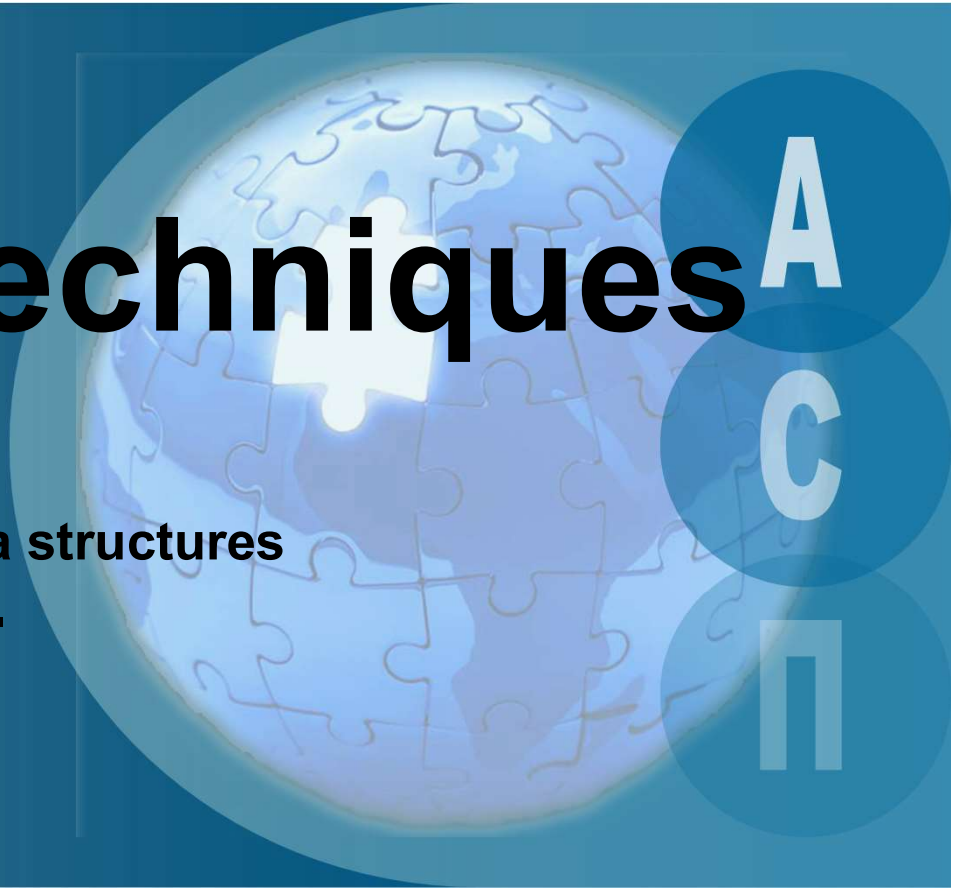
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Algorithms techniques



Algorithms and data structures

- lectures -

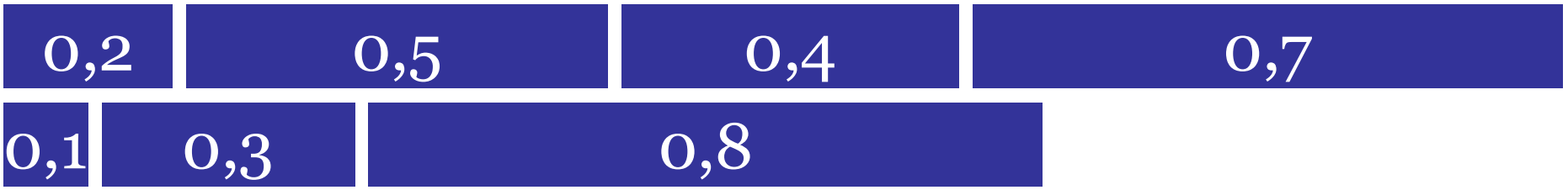
Outline

- Introduction to computational problems
- Ways of describing problems
- Pseudocode notation
- Algorithm Design Techniques
  - Algorithms based on brute force
  - Greedy algorithms
  - **Divide-and-conquer**
  - **Dynamic programming**
  - **Algorithms with random numbers**
  - **Other algorithms**

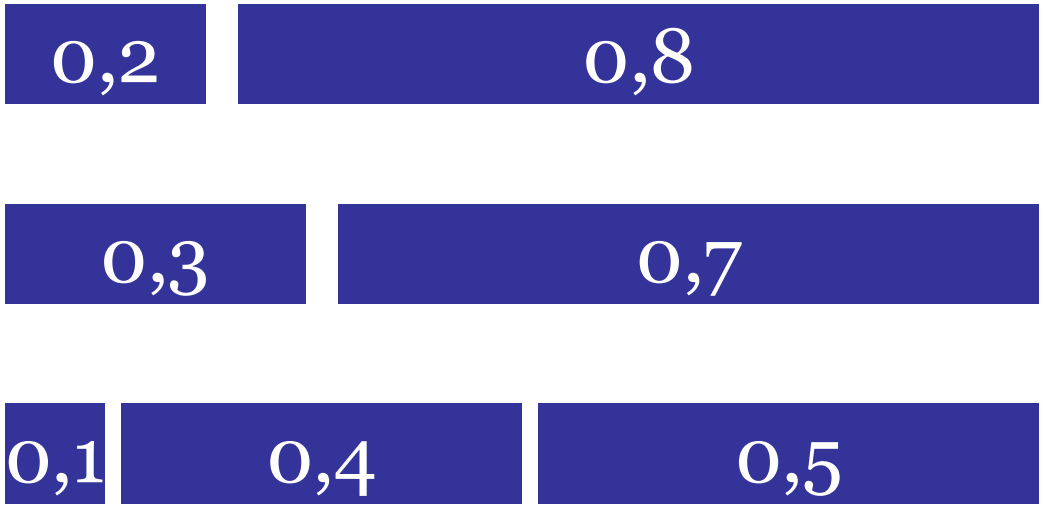
Greedy algorithms

- **Problem**: Knapsack packing algorithm
  - Let there be given *n* packages with sizes
    $s_1, s_2, ..., s_n, \quad 0 < s_i \leq 1, \quad i = 1..n$
  - 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 knapsacks 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 a knapsack
  - **Second** version: all package sizes are known in advance

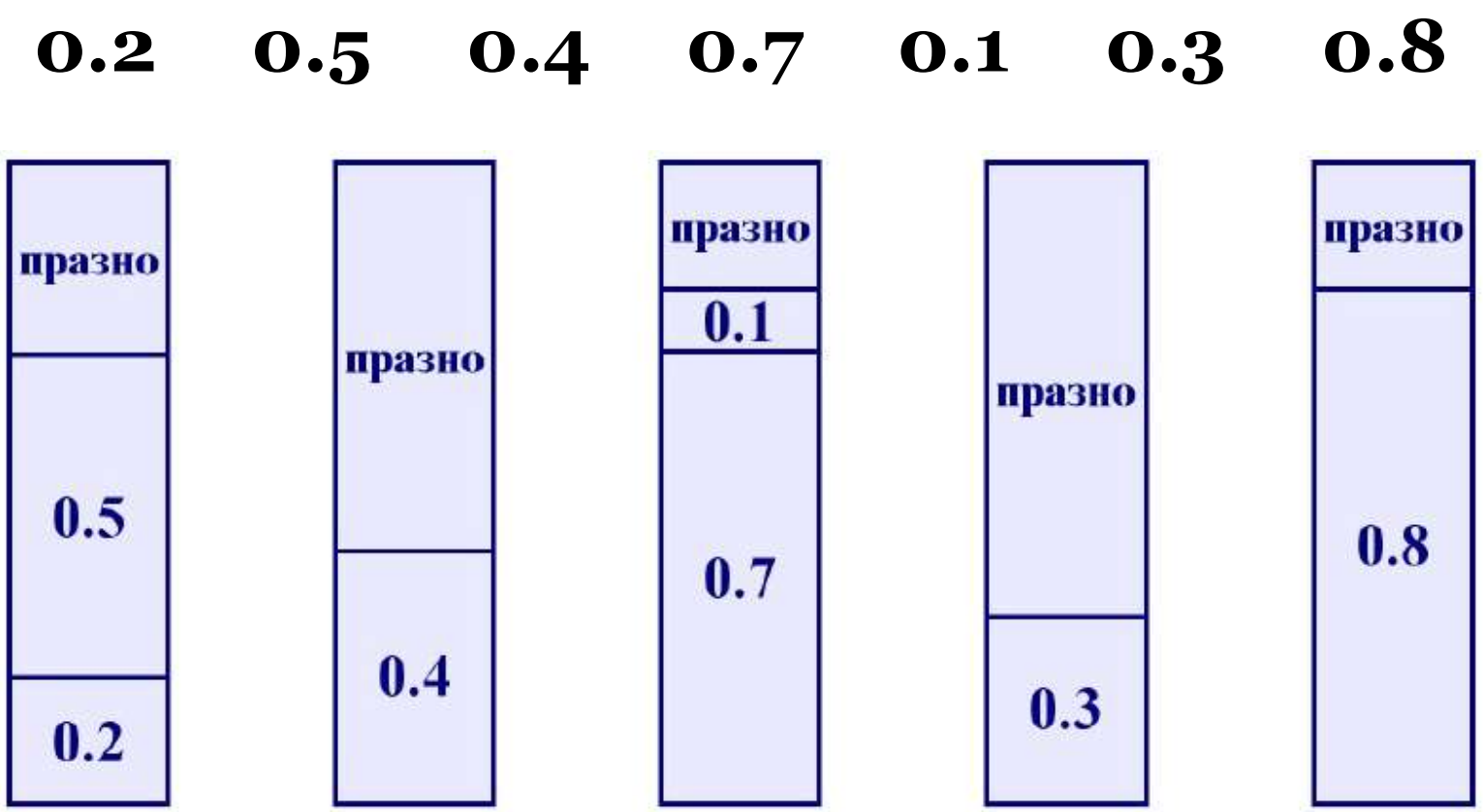
Greedy algorithms

Next fit solution:

- The first package is placed in the first knapsack
- When placing each subsequent package, it is checked whether there is room in the **currently open knapsack**
- If there is no room, a new knapsack is taken in which the package is placed

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

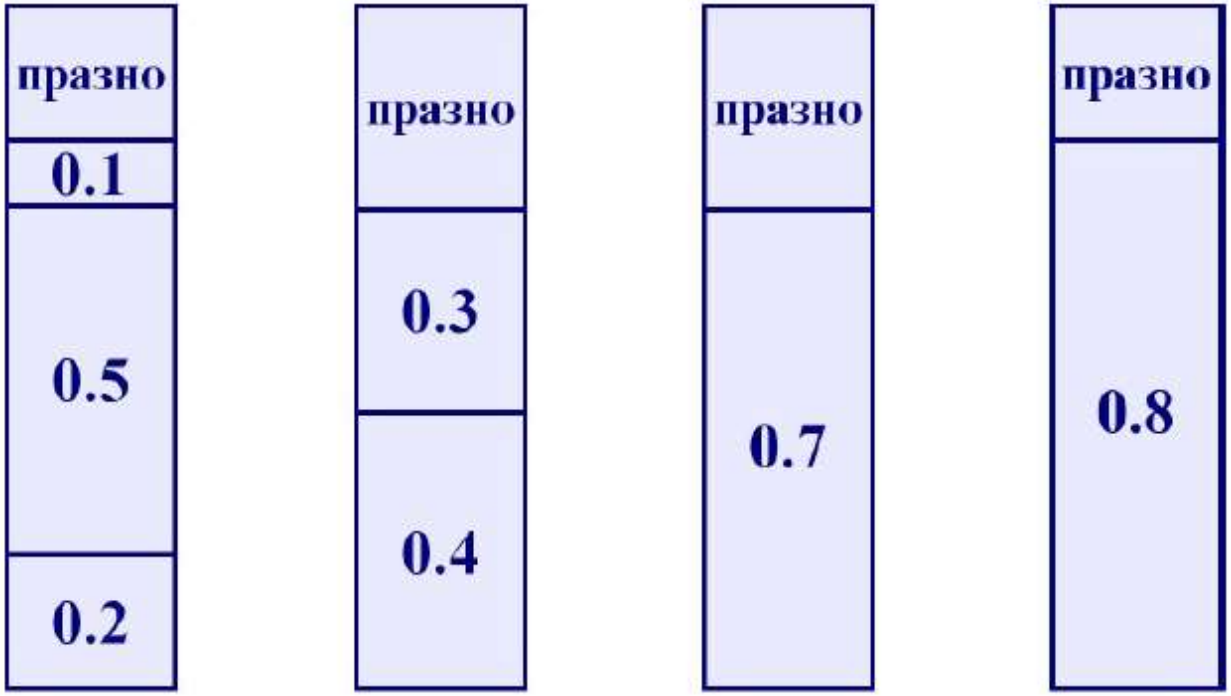
Greedy algorithms

- **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 place 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



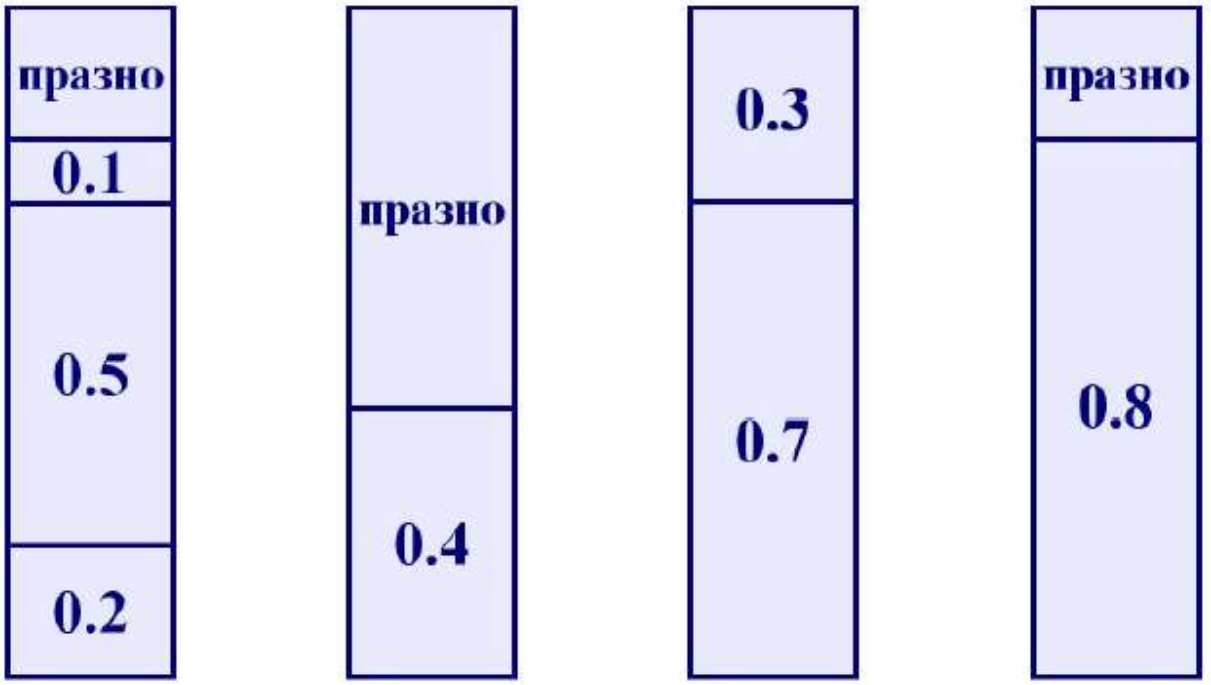
Greedy algorithms

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 backpacks

Greedy algorithms

0.2 0.5 0.4 0.7 0.1 0.3 0.8



Divide and conquer algorithms

- In principle, it always results in a correct algorithm
- It consists of two steps:
  - **Divide**: The problem is divided into smaller subproblems until elementary (base) cases are reached
  - **Conquer**: The solution to the problem is obtained from the solutions of the simpler subproblems using the so-called “merging" of the solution
- Solutions are usually obtained using **recursion**

Divide and conquer algorithms

- **Problem**: Finding the height of a binary tree

The height of the tree is one greater than the height of the left and right sub-trees of the root of the tree

Divide

Subproblems:
Find the heights of the left and right subtrees of a given node

Elementary case:
The height of a leaf is one. The height of a null link is zero

Conquer

Merging the results consists of finding the larger of the two resulting heights and increasing it by one.

Divide and conquer algorithms

- **Problem**: Find the maximal element in an array

Divide

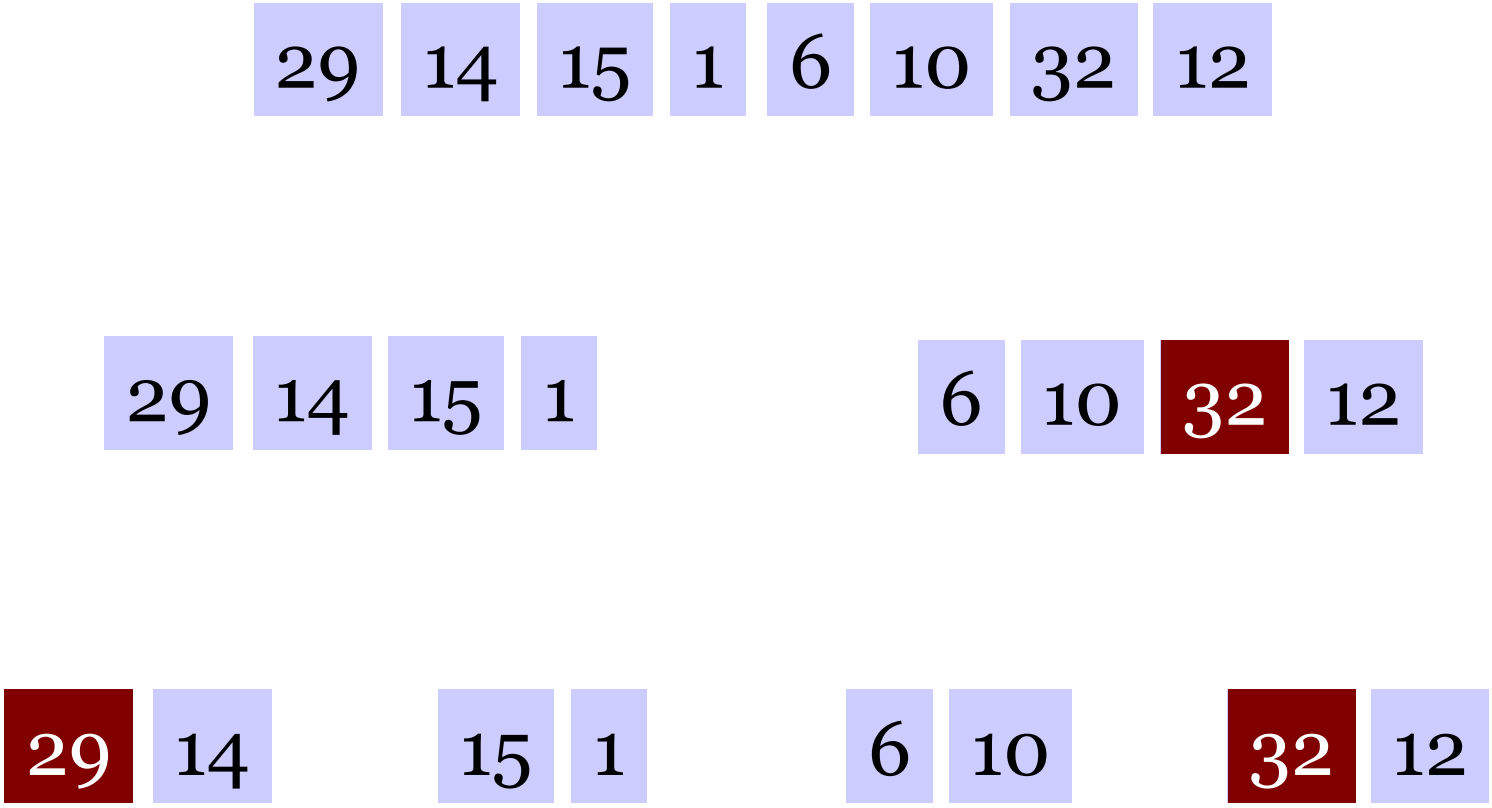
Subproblems:
Find the largest element in the left and right halves of the array

Elementary case:
An array element of length 1 is maximal

Conquer

Merging the results consists of finding the larger of the two results obtained for the left and right subarrays

Divide and conquer algorithms

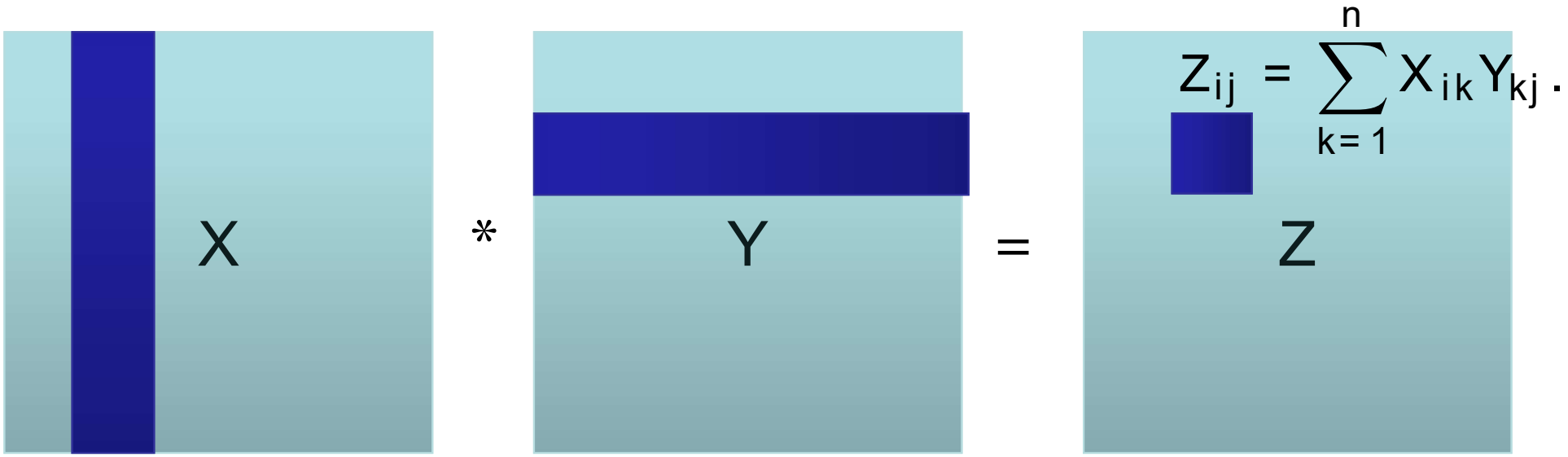


Divide and conquer algorithms

- Classes of problems in which the divide-and-conquer paradigm is applicable:
  - **geometric problems** (closest points in a plane, smallest convex polygon covering a set of points)
  - **computational problems** (fast fourier transform, matrix multiplication)
  - problems working with **graphs**

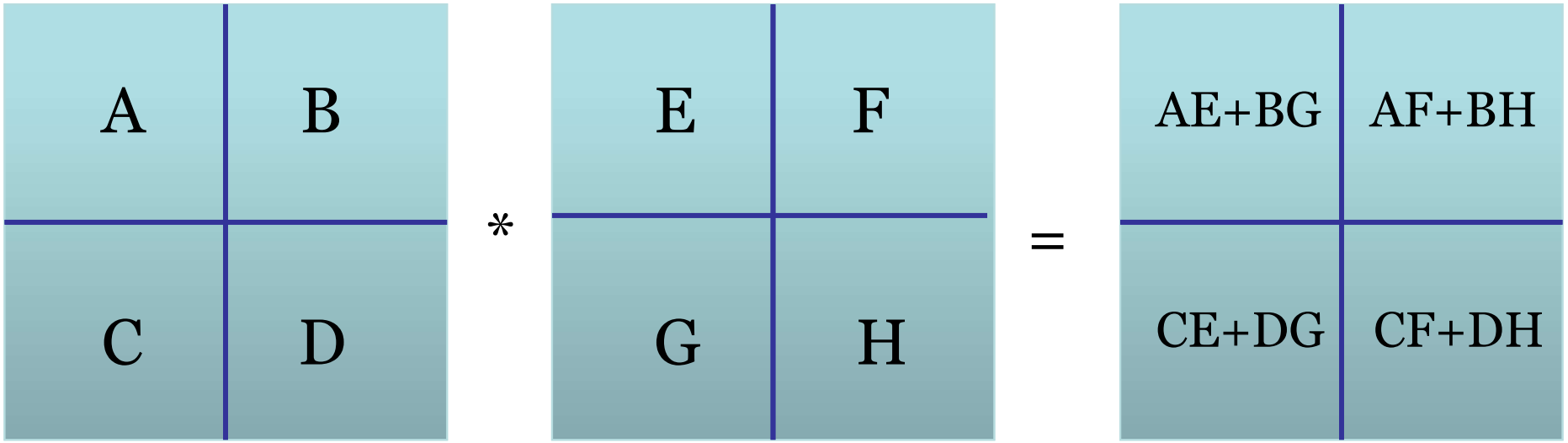
Divide and conquer algorithms

- Example: matrices multiplication



The complexity of this approach is $O(n^3)$

Divide and conquer algorithms



The total execution time can be calculated through the formula

$$T(n) = 8T(n/2) + O(n^2)$$

What is the complexity of this approach?

Divide and conquer algorithms

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

$$\begin{aligned} P_1 &= A(F - H) \\ P_2 &= (A + B)H \\ P_3 &= (C + D)E \\ P_4 &= D(G - E) \end{aligned}$$

$$\begin{aligned} P_5 &= (A + D)(E + H) \\ P_6 &= (B - D)(G + H) \\ P_7 &= (A - C)(E + F) \end{aligned}$$

The execution time is: $T(n) = 7T(n/2) + O(n^2)$

The resulting complexity is: $O(n^{\log 7}) \approx O(n^{2.81})$

Divide and conquer algorithms

- Matrices multiplication pseudocode

MMult(A, B, n)

1. If $n = 1$ Output $A \times B$
2. Else
3. Compute $A^{11}, B^{11}, \ldots, A^{22}, B^{22}$ % by computing $m = n/2$
4. $X_1 \leftarrow MMult(A^{11}, B^{11}, n/2)$
5. $X_2 \leftarrow MMult(A^{12}, B^{21}, n/2)$
6. $X_3 \leftarrow MMult(A^{11}, B^{12}, n/2)$
7. $X_4 \leftarrow MMult(A^{12}, B^{22}, n/2)$
8. $X_5 \leftarrow MMult(A^{21}, B^{11}, n/2)$
9. $X_6 \leftarrow MMult(A^{22}, B^{21}, n/2)$
10. $X_7 \leftarrow MMult(A^{21}, B^{12}, n/2)$
11. $X_8 \leftarrow MMult(A^{22}, B^{22}, n/2)$
12. $C^{11} \leftarrow X_1 + X_2$
13. $C^{12} \leftarrow X_3 + X_4$
14. $C^{21} \leftarrow X_5 + X_6$
15. $C^{22} \leftarrow X_7 + X_8$
16. Output C
17. End If

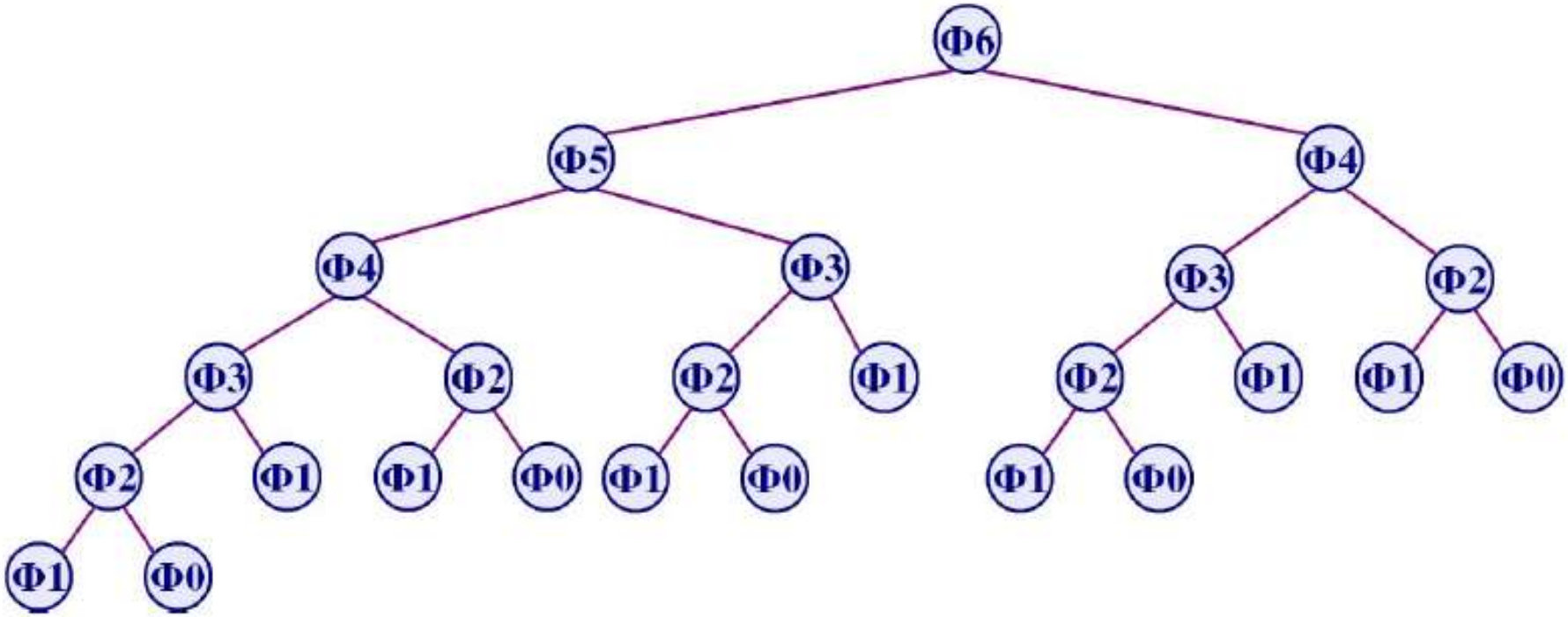
Dynamical programming

- Recursive solutions to certain problems can often result in inefficient code due to repeating recursive calls.

```
FIBONACII(n)
1  if(n ≤ 1)
2     return(1);
3  else
4     return(FIBONACCI(n–1) +FIBONACCI(n–2));
```

The complexity of the recursive algorithm for calculating Fibonacci numbers is $O(2^n)$

Dynamical programming



It is obvious that to calculate F6, the function that calculates F2 is called as many as five times

So, for n=50, $O(n)$ = 1125899906842624

Dynamical programming

- Dynamic programming is applied to **overlapping** problems (functions).
- Uses additional memory to store the results of subproblems

Dynamical programming

- Recursive approach (memoization)

$F = (1, 1, \text{undefined}, \text{undefined}, \ldots, \text{undefined})$

...

MEMFIBO(n)

1 **if** (n < 2)

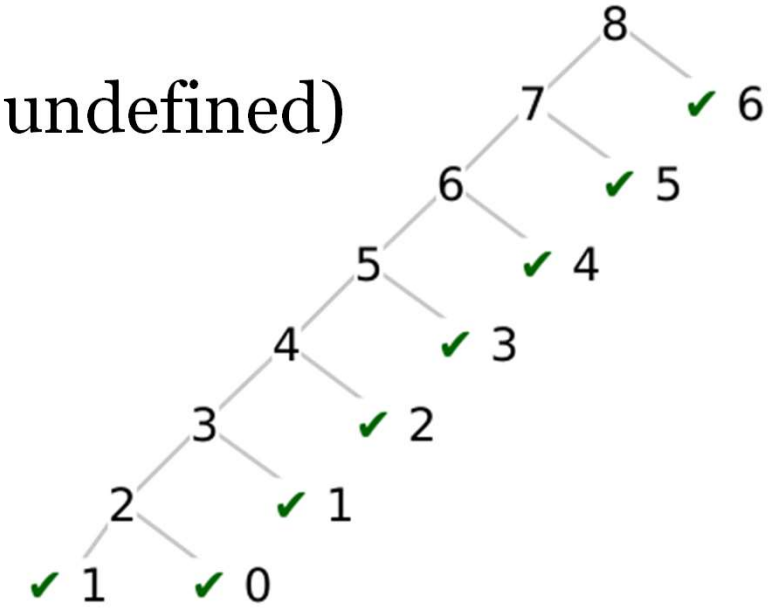
2 **return** (n)

3 **else**

4 **if** F_n is undefined

$F_n \leftarrow \text{MEMFIBO}(n-1) + \text{MEMFIBO}(n-2)$

5 **return** (F_n)



The complexity of such realization is O(n)

Dynamical programming

- Iterative approach (memoization)

```
INTERFIBO(n)
1  F_0 ← 1
2  F_1 ← 1
3  for i ← 2 to n
4      F_i ← F_{i-1} + F_{i-2}
5  return (F_n)
```

1	1	2	3	5	8	13	21

The complexity of this approach is O(n)

Dynamical programming

- The key to solving dynamic programming problems is finding a good **state** to remember
- The state should not take up much memory
- There should not be too many states, which would significantly save memory requirements of the algorithm

Dynamical programming

- Modification of the solution: instead of an array of n elements only two memory locations (two variables) are used:

```
INTERFIBO2(n)
1 prev ← 1
2 curr ← 1
3 for i ← 2 to n
4     next ← curr + prev
5     prev ← curr
6     curr ← next
7 return (curr)
```

1	1	2
1	2	3
2	3	5
3	5	8
5	8	13
8	13	21

Is it possible to further reduce the complexity of calculating the nth Fibonacci number?

Can constant complexity be reached?

Dynamical programming

- Dynamic programming is usually used in problems where some kind of **optimization** is required
- In such problems, several solutions are possible
- Each solution has its own **cost**, and we strive to find the solution with the most desirable cost.

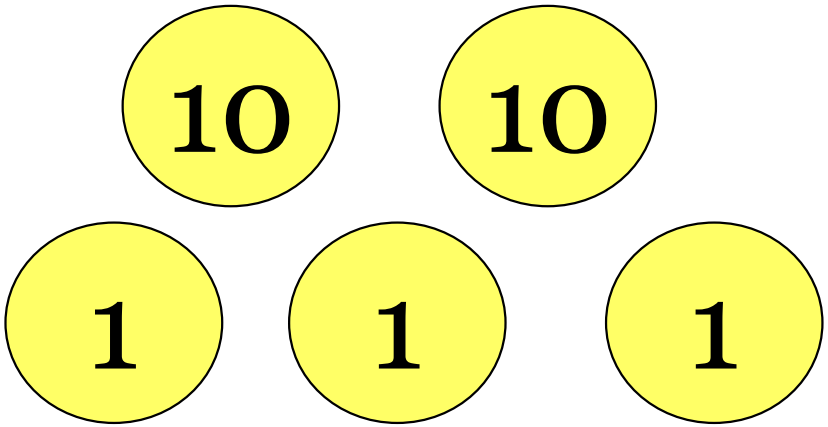
Dynamical programming

- **Problem**: A modified version of the change return problem
- Let us be given the following coins: 1, 5, 8 and 10 denars in unlimited quantity
- Find the optimal number of coins that should be returned for the change of 23 denars!

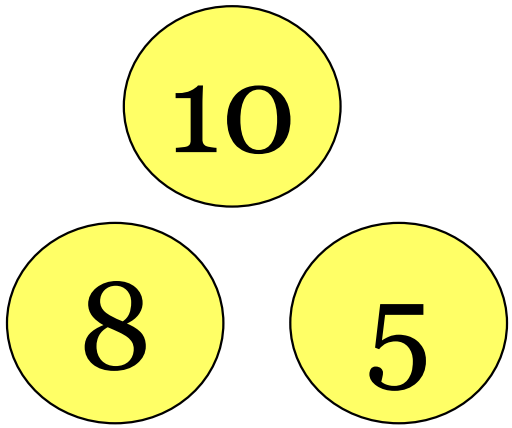
What is the solution if a greedy approach is used?

Dynamical programming

Greedy algorithm



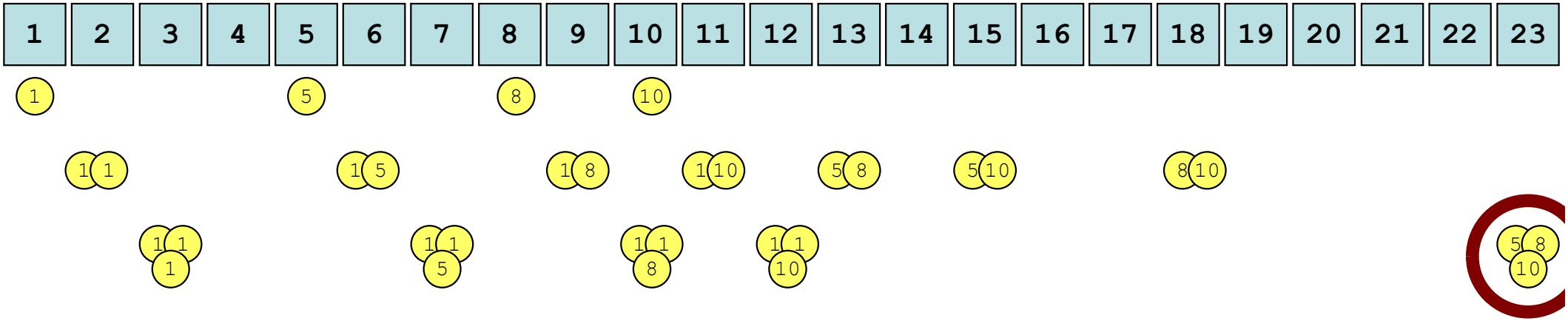
Optimal solution



Dynamical programming

- What is the state that can be used to solve the problem?
- A state is a way of describing a particular situation, which is a sub-solution to a problem
- If we take the solution of the problem as the final state, it is necessary to be able to find (or generate) all the other states that would bring us to the final state

Dynamical programming




```
// Initialization
for (i = 0; i < SUMS; ++i)
{   sum[i] = 0;  }

// Starting conditions
for (i = 0; i < COINS; ++i)
{   sum[coins[i]] = 1;  }
```

```
for (i = 0; i < SUMS; ++i)
{
  if (sum[i] == 0)
  {  continue;  }
  for (j = 0; j < COINS; ++j)
  {
    if((sum[i+coins[j]]==0)||(sum[i+coins[j]]>sum[i]+1))
    {
        sum[i+coins[j]] = sum[i] + 1;
    }
  }
}
```

Random number algorithms

- **Problem**: Student evaluation through tests
  - The professor can afford to check 50% of the material through tests, without affecting the time needed to carry out the teaching
  - Disadvantages:
    - the tests only cover the important topics
    - login to the tests
    - a set "pattern" of testing

Random number algorithms

- Solution to the problem:
  - The professor prepares tests for each topic
  - After each class, he throws a coin in front of the students.
  - If a "head" falls, testing will take place

- Whether a "head" will fall or not, is a **random process** that is realized by a random number generator

- Every programming language has a function to generate a random number (pseudo-random number)

Other algorithms

Algorithms for backtracking from the result

- They belong to the "brute force" type of algorithms
- They recognize the paths that do not lead to the result
- In this way, they reject unnecessary executions and checks
- Increasing the efficiency of the algorithm