Algorithms

Agenda

1. What is an algorithm
2. How to design an algorithm
3. Block diagrams
4. Fibonacci numbers
5. Binary search
6. Recursion
7. Computational complexity

What is an algorithm?

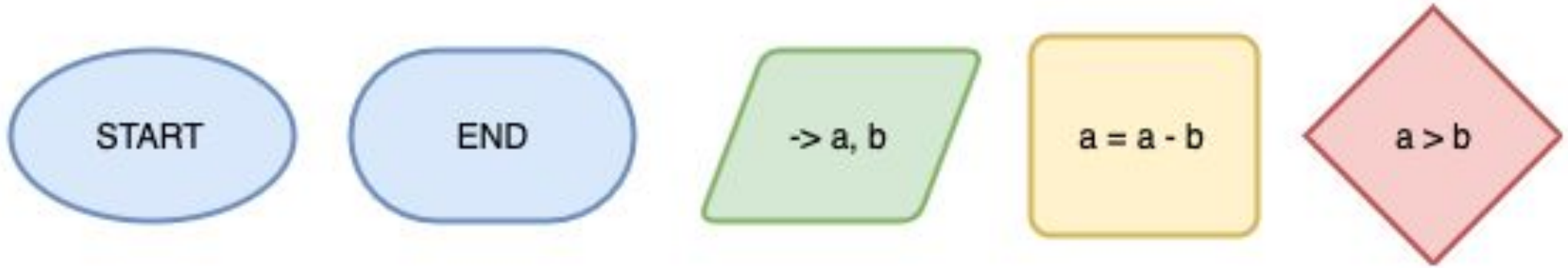
- A process or set of rules to be followed in calculations or other problem solving operations.
- Another way of looking at the above definition is that the algorithm is a set of rules that have to be executed step-by-step in order to obtain the expected result.
- An example is cooking a recipe: follow the instructions one by one in the given order and the result is a dish cooked perfectly.

How to design an algorithm?

1. Identify **the problem** that you want to solve.
2. Find the **constraints** that need to be satisfied by the solution.
3. Identify the **input** for the algorithm.
4. Identify the **output** of the algorithm.
5. Come up with a **solution** to the problem.
6. Consider drawing a **flowchart** of the algorithm.
7. **Implement** the algorithm in your favorite programming language.

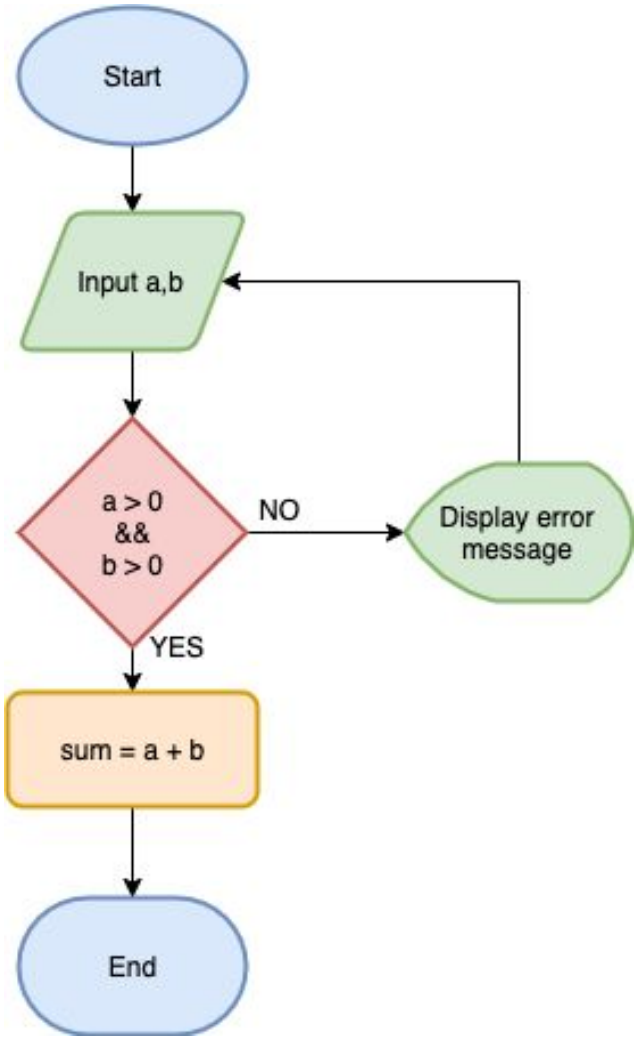
Block diagrams

1. Start/Stop block - indicates start or end of algorithm work.
2. Input/Output block - indicates taking or returning data.
3. Process block - set of instructions - variable assignments, calling functions, arithmetic operations.
4. Conditional block - contains a condition or a set of conditions. In case all conditions are fulfilled it continues through “Yes” branch and with “No” branch otherwise.



Let’s see it in action

1. The problem: add two numbers.
2. Constraints: numbers have to be greater than 0.
3. Input: the two numbers.
4. Output: the sum of the input numbers.
5. Solution: use the ‘+’ operator to calculate the sum.
6. Flowchart.
7. Implement the algorithm.



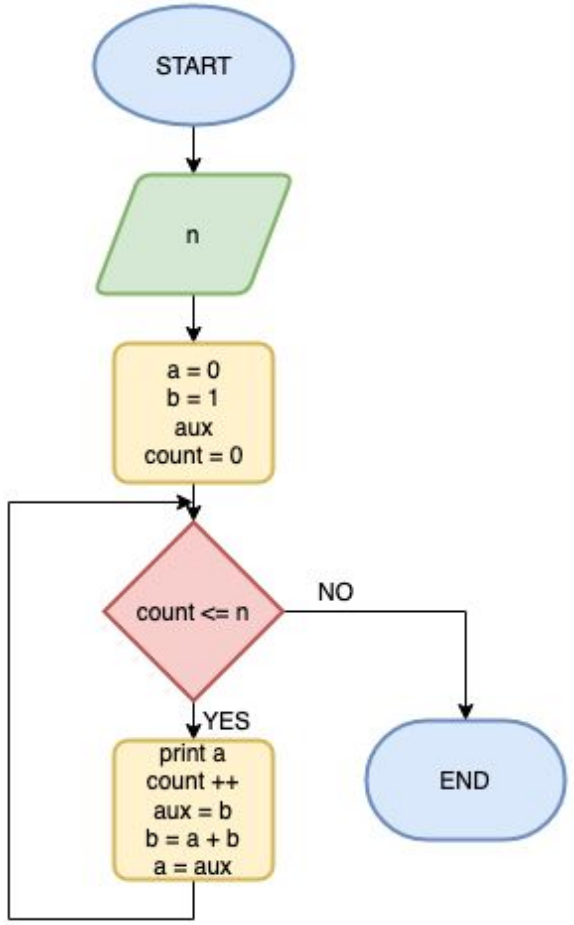
Fibonacci numbers

The Fibonacci Sequence is the series of numbers: 0,1,1,2,3,5,8,13,21,34,…

The next number is found by adding the two numbers before it: 2=1+1; 3=2+1; …

Steps:

1. The problem: display the Fibonacci sequence with n numbers.
2. Constraints: the sequence has n numbers.
3. Input: n – how many numbers to display.
4. Output: the Fibonacci sequence.
5. Solution.
6. Flowchart.
7. Implement the algorithm.



Binary search

Steps:

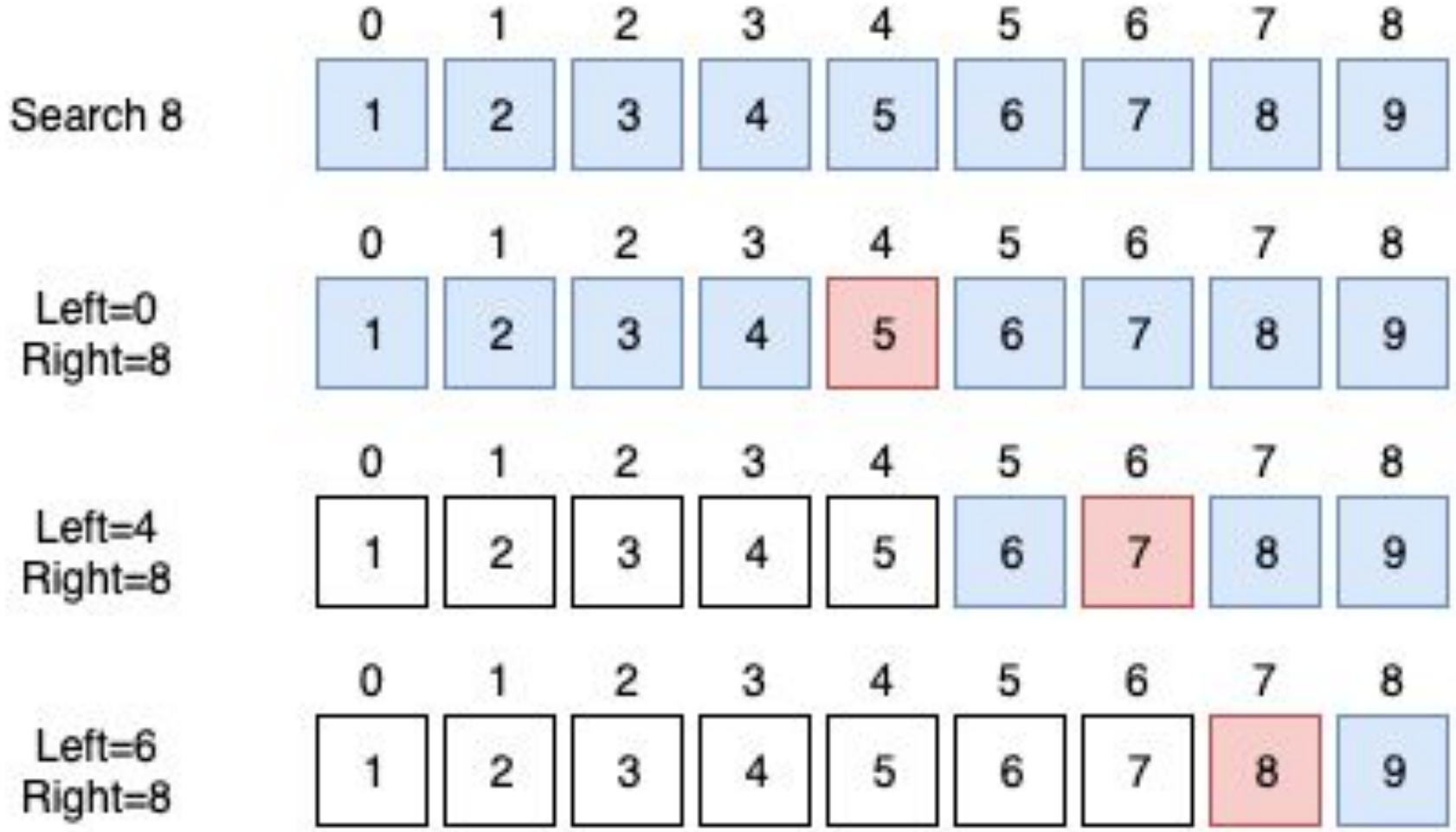
1. The problem: Given a sorted array of elements, write a function to search a given element x.
2. Constraints: the array is sorted.
3. Input: array of elements, value x to be searched.
4. Output: the position where the element was found.
5. Solution: search by repeatedly dividing the search interval in half. Begin with an interval that covers the whole array and compare the value x with the value from the middle of the interval.
   If the x is greater then repeat the process in the right half, else repeat the process in the left half. Stop when the value is found or the interval is empty.
6. Flowchart.
7. Implement the algorithm.

Binary search - example

Input values are ***arr = [1,2,3,4,5,6,7,8,9]*** and value to be searched ***x = 8***.

1. Consider the interval as the whole array with ***left = 0*** and ***right = 8***.
2. The steps below are executed while **left <= right**.
3. Compare the value of x with the value from the middle of interval ***arr[(left+right)/2]***.
4. Since **x > 5** then the interval becomes the right half which means ***left=(left+right)/2 + 1***.
5. Compare the value of x with the value from the middle of interval ***arr[(left+right)/2]***.
6. Since **x > 7** then the interval becomes the right half which means ***left=(left+right)/2 + 1***.
7. Compare the value of x with the value from the middle of interval ***arr[(left+right)/2]***.
8. The value of x is found at position 7.

Binary search - example



Recursion

- Recursion is a phenomenon when a function refers to itself.
- Is the process in which a function calls itself and the corresponding function is called recursive function.
- A recursive function must have a stop condition so it doesn't run infinitely.
- The most common error that appears when using recursion is stack overflow. This is because each function call is added to the call stack which is finite so if the recursion is too deep, the stack will run out of space.

Implementation of factorial

Iterative

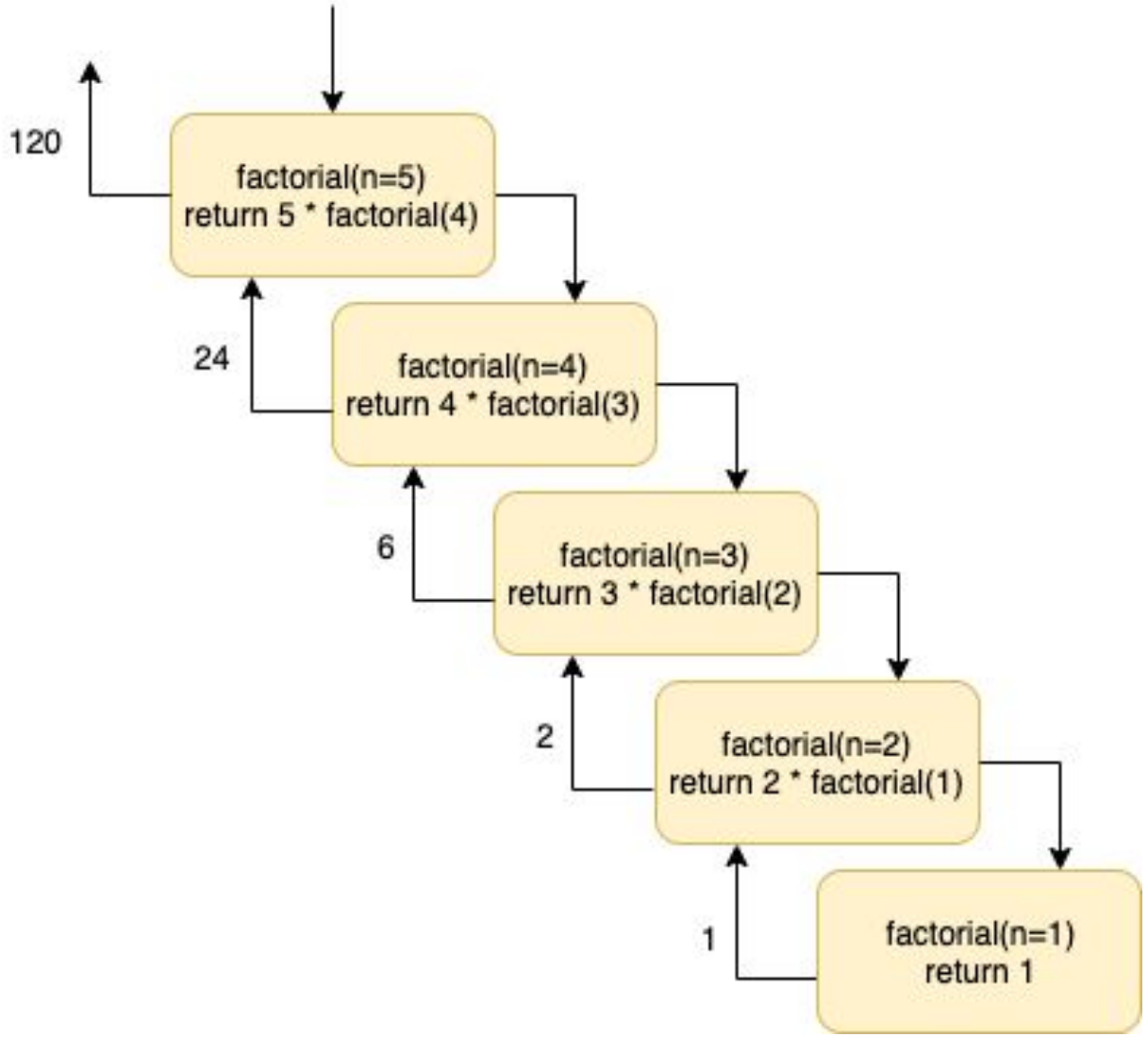
```
int factorial(int n)
   {
      int result = 1;
      int i;
      for (i=2; i<=n; i++)
         result *= i;
      return result;
   }
```

Recursive

```
int factorial(int n)
   {
      if (n == 0 || n == 1)
         return 1;

      return n*factorial(n-1);
   }
```

Representation of recursive factorial



Computational complexity

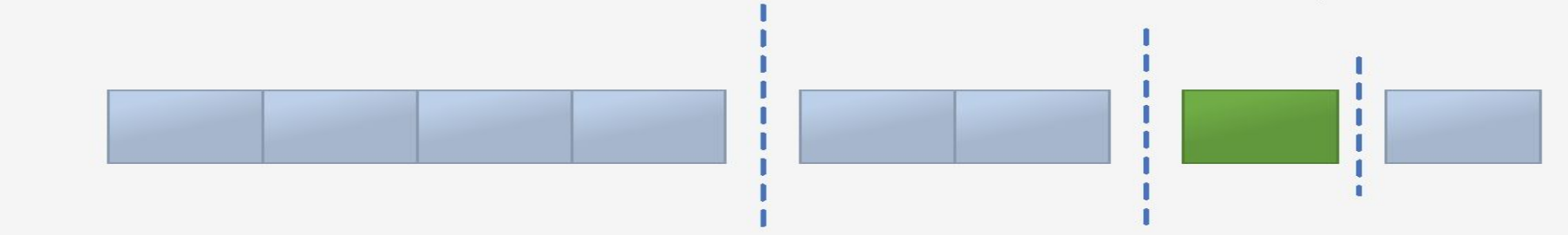
- Computational Complexity - describes the amount of resources (time or memory) required to solve computational problems.
- Is usually described with big-O notation.
- Big-O notation allows us to compare the number of operations to be performed.
- The speed of algorithms execution is not expressed in seconds, but in the rate of increase in the number of operations.

Computational complexity – simple search

1. Simple search (linear, sequential) consists in searching a set of elements element by element.
2. Is the simplest way to search in a set.
3. Number of operations require from 1 up to n operations, where n is the size of the set.
4. Because in the worst-case scenario we would perform n operations, we say that the complexity is O(n).

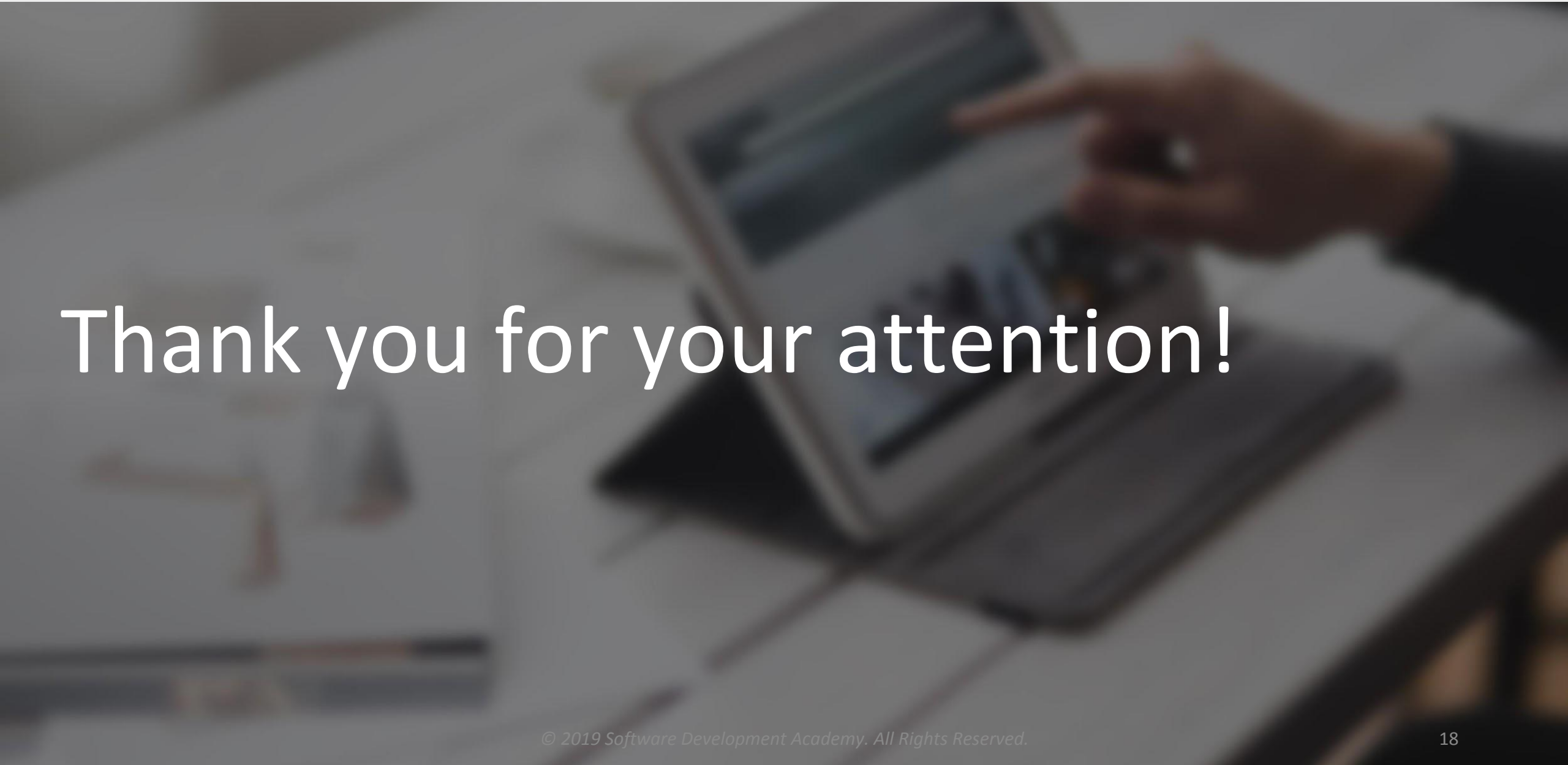
Computational complexity – binary search

1. Given sorted array of elements we divide the array by 2 and search in one of the parts.
2. To find the 74th element in an array of size 100, we only need 7 operations.
3. The complexity of this algorithm is O(log2n).



Common algorithms complexity

- O(1) - single operation or accessing element in an array using index (arr[i]).
- O(logn) - binary search.
- O(n) - reading n numbers.
- O(n*logn) - quicksort.
- O(n2) - bubble sort.
- O(n!) - traveling salesman problem.



Thank you for your attention!