Introduction

In this and next lectures we introduce the two core components of every computer program:

• Data structures
• Algorithms
A Fundamental Problem

Given a set $S$ of elements and a particular element $x$ the **search problem** is to decide whether $x$ is in $S$. 
The Search Problem

This problem has a large number of applications:

- $S =$ Names in a phone book
  - $x =$ name of a person
The Search Problem

This problem has a large number of applications:

- $S =$ Student records
  - $x =$ student ID
The Search Problem

This problem has a large number of applications:

• $S =$ Variables in a program

  $x =$ name of a variable

    /* When the user has selected a play, this method is invoked to
    process the selected play */
    public void actionPerformed(ActionEvent event) {
      if (event.getSource() instanceof JButton) { /* Some position of the
        board was selected */

      int row = -1, col = -1;
      PosPlay pos;

      if (game_ended) System.exit(0);
      /* Find out which position was selected by the player */
      for (int i = 0; i < board_size; i++) {
        for (int j = 0; j < board_size; j++)
          if (event.getSource() == board[i][j]) {
            row = i;
            col = j;
            break;
          }
      }
      if (row != -1) break;
    }
The Search Problem

This problem has a large number of applications:

- $S =$ Web host names
- $x =$ URL
Solving a Problem

The solution of a problem has 2 parts:

• How to organize data

• How to solve the problem
Solving a Problem

The solution of a problem has 2 parts:

• How to organize data
  
  **Data structure:**
  
  ➢ a systematic way of organizing and accessing data

• How to solve the problem
Solving a Problem

The solution of a problem has 2 parts:

• How to organize data
  **Data structure:**
  ➢ a systematic way of organizing and accessing data

• How to solve the problem
  **Algorithm:**
  ➢ a step-by-step procedure for performing some task in finite time
A First Solution

For simplicity, let us assume that S is a set of \( n \) different integers stored in non-decreasing order in an array \( L \).

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|}
L & 3 & 9 & 11 & 17 & 18 & 26 & 29 & 43 & 48 & 55 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]
Algorithm LinearSearch (L,n,x)

Input: Array L of size n and value x

Output: Position i, 0 ≤ i < n, such that L[i] = x, if x in L, or -1, if x not in L
To prove that an algorithm is correct we need to show 2 things:
To prove that an algorithm is correct we need to show 2 things:
• The algorithm terminates
Proving the Correctness of an Algorithm

To prove that an algorithm is correct we need to show 2 things:

• The algorithm terminates
• The algorithm produces the correct output
**Algorithm LinearSearch (L, n, x)**

**Input:** Array L of size n and value x

**Output:** Position i, $0 \leq i < n$, such that $L[i] = x$, if x in L, or -1, if x not in L

\[
i \leftarrow 0
\]

while $(i < n)$ and $(L[i] \neq x)$ do
  \[
i \leftarrow i + 1
\]

if $i = n$ then return -1
else return i
Correctness of Linear Search

Termination

• $i$ takes initial value 0 and in each iteration of the while loop it increases by 1, so $i$ takes values 0, 1, 2, 3, ...

• The while loop cannot perform more than $n$ iterations because of the condition $(i < n)$
Correctness of Linear Search

Correct Output

• The algorithm compares $x$ with $L[0]$, $L[1]$, $L[2]$, ...
Correctness of Linear Search

Correct Output

• The algorithm compares $x$ with $L[0]$, $L[1]$, $L[2]$, ...

• Hence, if $x$ is in $L$ then $x = L[i]$ in some iteration of the while loop; this ends the loop and then the algorithm correctly returns the value $i$
Algorithm LinearSearch (L, n, x)

Input: Array L of size n and value x

Output: Position i, 0 ≤ i < n, such that L[i] = x, if x in L, or -1, if x not in L

i ← 0
while (i < n) and (L[i] ≠ x) do
    i ← i + 1
if i = n then return -1
else return i
Correctness of Linear Search

Correct Output
• The algorithm compares $x$ with $L[0]$, $L[1]$, $L[2]$, ...
• Hence, if $x$ is in $L$ then $x = L[i]$ in some iteration of the **while** loop; this ends the loop and then the algorithm correctly returns the value $i$
• If $x$ is not in $L$ then in some iteration $i = n$; this ends the loop and the algorithm returns -1.
Binary Search

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>9</th>
<th>11</th>
<th>17</th>
<th>18</th>
<th>26</th>
<th>29</th>
<th>43</th>
<th>48</th>
<th>55</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Algorithm BinarySearch (L, x, first, last)

Input: Array L of size n and value x

Output: Position i, 0 ≤ i < n, such that L[i] = x, if x in L, or -1, if x not in L
Correctness of Binary Search

Termination

• If \( x = L[mid] \) the algorithm terminates
Correctness of Binary Search

Termination

• If \( x = L[mid] \) the algorithm terminates
• If \( x < L[mid] \) or \( x > L[mid] \), the value \( L[mid] \) is discarded from the next recursive call. Hence, in each recursive call the size of \( L \) decreases by at least 1.
Correctness of Binary Search

Termination

• If $x = L[mid]$ the algorithm terminates

• If $x < L[mid]$ or $x > L[mid]$, the value $L[mid]$ is discarded from the next recursive call. Hence, in each recursive call the size of $L$ decreases by at least 1.

• After a finite number of recursive calls either $x = L[mid]$ or the size of $L$ is zero and thus in both cases the algorithm ends.
Algorithm  BinarySearch (L,x, first, last )
Input:  Array L of size n and value x
Output: Position i, 0 ≤ i < n, such that L[i] = x, if x in L, or -1, if x not in L

if first > last  then return  -1
else  {
    mid ← [(first +last )/2]
    if  x = L[mid ] then return  mid
    else if  x < L[mid ] then
        return  BinarySearch (L,x,first,mid -1)
    else return  BinarySearch (L,x,mid +1,last )
}
Correctness of Binary Search

Correct Output

• If $x = L[mid]$ the algorithm correctly returns $mid$
Correctness of Binary Search

Correct Output

• The algorithm only discards values different from $x$ so
  – if all values of $L$ are discarded (so $L$ is empty) it is because $x$ is not in $L$ and the algorithm correctly returns $-1$.
  – if $x$ is in $L$ then in some recursive call it must be that $x = L[mid]$ so the algorithm correctly returns $mid$. 
Comparing Algorithms

We have several algorithms for solving the same problem. Which one is better?
Comparing Algorithms

We have several algorithms for solving the same problem. Which one is better?

Criteria that we can use to compare algorithms:

• Conceptual simplicity
Comparing Algorithms

We have several algorithms for solving the same problem. Which one is better?

Criteria that we can use to compare algorithms:

• Conceptual simplicity
• Difficulty to implement
Comparing Algorithms

We have several algorithms for solving the same problem. Which one is better?

Criteria that we can use to compare algorithms:

• Conceptual simplicity
• Difficulty to implement
• Difficulty to modify
Comparing Algorithms

We have several algorithms for solving the same problem. Which one is better?

Criteria that we can use to compare algorithms:

- Conceptual simplicity
- Difficulty to implement
- Difficulty to modify
- Running time
Comparing Algorithms

We have several algorithms for solving the same problem. Which one is better?

Criteria that we can use to compare algorithms:

- Conceptual simplicity
- Difficulty to implement
- Difficulty to modify
- Running time
- Space (memory) usage
Complexity

We define the complexity of an algorithm as the amount of computer resources that it uses.
Complexity

We define the complexity of an algorithm as the amount of computer resources that it uses. We are particularly interested in two computer resources: memory and time.
We define the complexity of an algorithm as the amount of computer resources that it uses. We are particularly interested in two computer resources: memory and time. Consequently, we define two types of complexity functions:

- Space complexity: amount of memory that the algorithm needs.
We define the complexity of an algorithm as the amount of computer resources that it uses. We are particularly interested in two computer resources: memory and time. Consequently, we define two types of complexity functions:

- **Space complexity**: amount of memory that the algorithm needs.
- **Time complexity**: amount of time needed by the algorithm to complete.
Complexity Function

The complexity of an algorithm is a non-decreasing function on the size of the input.
Types of Complexity Functions

For both kinds of complexity functions we can define 3 cases:

- **Best case**: Least amount of resources needed by the algorithm to solve an instance of the problem of size $n$. 
Types of Complexity Functions

For both kinds of complexity functions we can define 3 cases:

• **Worst case**: Largest amount of resources needed by the algorithm to solve an instance of the problem of size n.
Types of Complexity Functions

For both kinds of complexity functions we can define 3 cases:

• **Average case:**
  amount of resources to solve instance 1 of size \( n \) +
  amount of resources to solve instance 2 of size \( n \) + 
  \[\ldots\]
  amount of resources to solve last instance of size \( n \)
  number of instances of size \( n \)
Types of Complexity Functions

In this course we will study worst case complexity.
How do we compute the time complexity of an algorithm?
How do we compute the time complexity of an algorithm?

We need a clock to measure time.
Experimental way of measuring the time complexity
Experimental way of measuring the time complexity

We need:

• a computer
Experimental way of measuring the time complexity

We need:

• a computer

• a compiler for the programming language in which the algorithm will be implemented
Experimental way of measuring the time complexity

We need:

• a computer
• a compiler for the programming language in which the algorithm will be implemented
• an operating system
Experimental way of measuring the time complexity

Drawbacks

- Expensive
- Time consuming
- Results depend on the input selected
- Results depend on the particular implementation
Experimental way of measuring the time complexity

Drawbacks

• Expensive
• Time consuming
Experimental way of measuring the time complexity

Drawbacks

• Expensive
• Time consuming
• Results depend on the input selected
Experimental way of measuring the time complexity

Drawbacks
• Expensive
• Time consuming
• Results depend on the input selected
• Results depend on the particular implementation
Computing the time complexity

• We wish to compute the time complexity of an algorithm without having to implement it.
Computing the time complexity

- We wish to compute the time complexity of an algorithm without having to implement it.
- We want the time complexity to characterize the performance of an algorithm on ALL inputs and all implementations (i.e. all computers and all programming languages).
Algorithm LinearSearch (L,n,x)

Input: Array L of size n and value x

Output: Position i, $0 \leq i < n$, such that $L[i] = x$, if x in L, or -1, if x not in L

i ← 0
while (i < n) and (L[i] ≠ x) do
  i ← i+1
if i=n then return -1 else return i
Primitive Operations

A basic or primitive operation is an operation that requires a constant amount of time in any implementation.

Examples:

←, +, -, x, /, <, >, =, ≤, ≥, ≠
Primitive Operations

A basic or primitive operation is an operation that requires a constant amount of time in any implementation.

Examples:

\[←, +, -, x, /, <, >, =, ≤, ≥, ≠\]

Constant, means independent from the size of the input.
Algorithm LinearSearch (L,n,x)
    i ← 0
    while (i < n) and (L[i] ≠ x) do
        i ← i+1
    if i=n then return -1
    else return i
When do we need to compute the time complexity function?

Assume a computer with speed $10^8$ operations per second.

<table>
<thead>
<tr>
<th>Time Complexity</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(n) = n$</td>
<td>10^{-7}s</td>
</tr>
<tr>
<td>$f(n) = n^2$</td>
<td>10^{-5}s</td>
</tr>
<tr>
<td>$f(n) = n^3$</td>
<td>10^{-2}s</td>
</tr>
<tr>
<td>$f(n) = 2^n$</td>
<td>2.4 hrs</td>
</tr>
<tr>
<td></td>
<td>10 yrs</td>
</tr>
<tr>
<td></td>
<td>360 yrs</td>
</tr>
</tbody>
</table>
When do we need to compute the time complexity function?

Assume a computer with speed $10^8$ operations per second.

<table>
<thead>
<tr>
<th>Time Complexity</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 10</td>
<td></td>
</tr>
<tr>
<td>$f(n) = n$</td>
<td>$10^{-7}$ s</td>
</tr>
<tr>
<td>$f(n) = n^2$</td>
<td>$10^{-6}$ s</td>
</tr>
<tr>
<td>$f(n) = n^3$</td>
<td>$10^{-5}$ s</td>
</tr>
<tr>
<td>$f(n) = 2^n$</td>
<td>$10^{-5}$ s</td>
</tr>
</tbody>
</table>
When do we need to compute the time complexity function?

Assume a computer with speed $10^8$ operations per second.

<table>
<thead>
<tr>
<th>Time Complexity</th>
<th>Time</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n = 10</td>
<td>n = 20</td>
</tr>
<tr>
<td>$f(n) = n$</td>
<td>$10^{-7}$ s</td>
<td>$2 \times 10^{-7}$ s</td>
</tr>
<tr>
<td>$f(n) = n^2$</td>
<td>$10^{-6}$ s</td>
<td>$4 \times 10^{-6}$ s</td>
</tr>
<tr>
<td>$f(n) = n^3$</td>
<td>$10^{-5}$ s</td>
<td>$8 \times 10^{-5}$ s</td>
</tr>
<tr>
<td>$f(n) = 2^n$</td>
<td>$10^{-5}$ s</td>
<td>$10^{-1}$ s</td>
</tr>
</tbody>
</table>
When do we need to compute the time complexity function?

Assume a computer with speed $10^8$ operations per second.

<table>
<thead>
<tr>
<th>Time Complexity</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n = 10</td>
</tr>
<tr>
<td>$f(n) = n$</td>
<td>$10^{-7}$ s</td>
</tr>
<tr>
<td>$f(n) = n^2$</td>
<td>$10^{-6}$ s</td>
</tr>
<tr>
<td>$f(n) = n^3$</td>
<td>$10^{-5}$ s</td>
</tr>
<tr>
<td>$f(n) = 2^n$</td>
<td>$10^{-5}$ s</td>
</tr>
</tbody>
</table>
When do we need to compute the time complexity function?

Assume a computer with speed $10^8$ operations per second.

<table>
<thead>
<tr>
<th>Time Complexity</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n = 10</td>
</tr>
<tr>
<td>f(n) = n</td>
<td>$10^{-7}$ s</td>
</tr>
<tr>
<td>f(n) = $n^2$</td>
<td>$10^{-6}$ s</td>
</tr>
<tr>
<td>f(n) = $n^3$</td>
<td>$10^{-5}$ s</td>
</tr>
<tr>
<td>f(n) = $2^n$</td>
<td>$10^{-5}$ s</td>
</tr>
</tbody>
</table>
When do we need to compute the time complexity function?

Assume a computer with speed $10^8$ operations per second.

<table>
<thead>
<tr>
<th>Time Complexity</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n = 10$</td>
</tr>
<tr>
<td>$f(n) = n$</td>
<td>$10^{-7}$ s</td>
</tr>
<tr>
<td>$f(n) = n^2$</td>
<td>$10^{-6}$ s</td>
</tr>
<tr>
<td>$f(n) = n^3$</td>
<td>$10^{-5}$ s</td>
</tr>
<tr>
<td>$f(n) = 2^n$</td>
<td>$10^{-5}$ s</td>
</tr>
</tbody>
</table>
Asymptotic Notation

We want to characterize the time complexity of an algorithm for large inputs irrespective of the value of implementation dependent constants.
Asymptotic Notation

We want to characterize the time complexity of an algorithm for large inputs irrespective of the value of implementation dependent constants.

The mathematical notation used to express time complexities is the asymptotic notation:
Asymptotic or Order Notation

Let f(n) and g(n) be functions from \( \mathbb{I} \) to \( \mathbb{R} \). We say that f(n) is O(g(n)) (read "f(n) is big-Oh of g(n)" or "f(n) is of order g(n)") if there is a real constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that

\[
f(n) \leq c \times g(n) \quad \text{for all} \quad n \geq n_0
\]
Asymptotic or Order Notation

Let \( f(n) \) and \( g(n) \) be functions from \( \mathbb{N} \) to \( \mathbb{R} \). We say that \( f(n) \) is \( O(g(n)) \) (read ``\( f(n) \) is big-Oh of \( g(n) \)'' or ''\( f(n) \) is of order \( g(n) \)') if there is a real constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that

\[
f(n) \leq c \times g(n) \text{ for all } n \geq n_0
\]

Constant = independent from \( n \)
Asymptotic or Order Notation

Let $f(n)$ and $g(n)$ be functions from $\mathbb{I}$ to $\mathbb{R}$. We say that $f(n)$ is $O(g(n))$ (read "$f(n)$ is big-Oh of $g(n)$" or "$f(n)$ is of order $g(n)$") if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \times g(n) \text{ for all } n \geq n_0$$

Constant = independent from $n$

We sometimes write $f(n) = O(g(n))$ or $f(n) \in O(g(n))$. 