Introduction

In this and next lectures we introduce the concepts of:

- Data structure
- Algorithm
- Complexity of an algorithm

A Fundamental Problem

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

This problem has a large number of applications:

- $S = \{$Names in a phone book$\}$
- $S = \{$Student records$\}$
- $S = \{$Variables in a program$\}$
- $S = \{$Web host names$\}$

The solution of a problem has 2 parts:

1. How to organize data

   Data structure:

   a systematic way of organizing and accessing data

2. 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 integers stored in non-decreasing order in an array L.

L

3	9	11	17	18	26	29	43	48	55
0	1	2	3	4	5	6	7	8	9

Given a value x, the simplest solution is to compare it against each element of L until either:

- an element $L[i]$ of value x is found, or
- x is compared against all elements in L and all of them are different from x.

Linear Search

This way of solving the search problem is called *linear search.*

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 $x \notin L$

```
$i \leftarrow 0$
while $(i < n)$ and $(L[i] \neq x)$ do $i \leftarrow i + 1$

if $i = n$ then return $-1$
else return $i$
```

Another Solution

Linear search is no the only way of solving the search problem, there are many other solutions. Consider, for example, the method that is used to look-up a name $x$in the phone book:

- Compare x with any name y located near the middle of the phone book. If $x = y$ the search ends.
- If x precedes y in lexicographic order, then continue the search for x on the first half of the phone book.
- Otherwise, continue the search for x on the second half of the phone book.

This way of solving the search problem is called *binary search.*

Binary Search

Algorithm BinarySearch($L, x, \text{first}, \text{last}$)
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 $x \notin L$

```
if $\text{first} > \text{last}$ then return $-1$
else {
  $\text{mid} \leftarrow \lfloor(\text{first} + \text{last})/2\rfloor$
  if $x = L[\text{mid}]$ then return mid
  else if $x < L[\text{mid}]$ then
    return BinarySearch($L, x, \text{first}, \text{mid}-1$)
  else
    return BinarySearch($L, x, \text{mid}+1, \text{last}$)
}
```

Comparing Algorithms

We have two 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

  ⋮

In this course we are interested in the last two criteria.

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*. 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.

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

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

- best case: Least amount of time needed by the algorithm to solve an instance of the problem of size n.
- worst case: Largest amount of time needed by the algorithm to solve an instance of the problem of size n.
- average case:

$$\frac{\begin{array}{c}\text{time to solve instance 1 of size } n \,+ \\ \text{time to solve instance 2 of size } n \,+ \\ \vdots \\ \text{time to solve last instance of size } n\end{array}}{\text{number of instances of size } n}$$

In this course we will study worst case complexity functions.

How do we measure the *time* complexity?

We need a clock to measure time.

Experimental way of measuring the time complexity of an algorithm

We need:

- a computer
- a compiler for the programming language in which the algorithm will be implemented
- an operating system

Drawbacks

- Expensive
- Time consuming
- Results depend on the input selected
- Results depend on the particular implementation

Primitive Operations

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

Examples:

$\leftarrow, +, -, \times, /, < ., >, =\leq, \geq, \neq$

Constant, means independent from the size of the input.

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

Time	**Time**			
Complexity	$n = 10$	$n = 20$	$n = 10^3$	$n = 10^6$
$f(n) = n$	10^{-7}s	2×10^{-7}s	10^{-5}s	10^{-2}s
$f(n) = n^2$	10^{-6}s	4×10^{-6}s	10^{-2}s	2.4 hrs
$f(n) = n^3$	10^{-5}s	8×10^{-5}s	10s	360 yrs
$f(n) = 2^n$	10^{-5} s	10^{-1}s	10^{293} yrs	10^{10^5} yrs

Given two algorithms for solving the same problem, if the algorithms are going to be used to solve large size problems, then we should compute their time complexity functions to select the fastest one.

However, if the algorithms are to be used to solve only small problem instances, then it does not make sense to spend time computing their time complexity functions.

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:

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

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

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