Analysis of Algorithms

Comparing Algorithms

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

Criteria that we can use to compare algorithms:

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

These are the criteria we are going to use in this course

Complexity

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

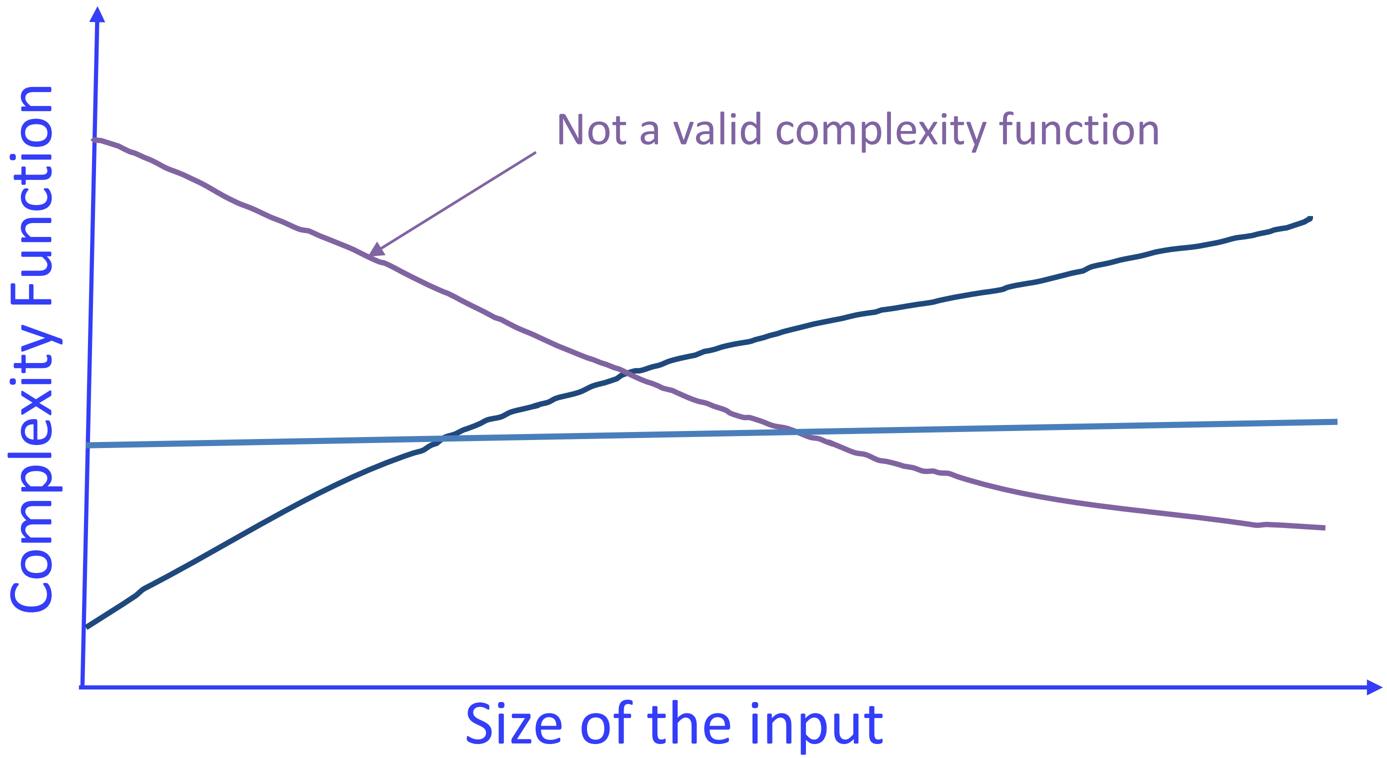
We are particularly interested in two computer resources: memory and time.

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.

Complexity Function

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



Linear Search

The complexity depends on the input itself, not only on its size.

1) L

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

$x = 3$

2) L

5	7	11	21	24	26	33	41	67	89
0	1	2	3	4	5	6	7	8	9

$x = 67$

Linear search takes less time on the first instance

Types of Complexity Functions

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

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

For linear search the best case is when *x* is in the first entry of L

L	*x*	9	11	17	18	26	29	43	48	55

Types of Complexity Functions

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

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

The best case for an algorithm is NOT when n = 0 or when n is very small!

Types of Complexity Functions

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

- **Worst case complexity**: Largest amount of resources needed by the algorithm to solve an instance of the problem of size n.

The worst case for an algorithm is NOT when n is very large!

Types of Complexity Functions

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

- **Worst case complexity**: Largest amount of resources needed by the algorithm to solve an instance of the problem of size n.

For linear search the worst case is when *x* is not in L.

x not in *L*	3	9	11	17	18	26	29	43	48	55

Types of Complexity Functions

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

- **Average case complexity**:

$$\frac{\begin{array}{c}\text{amount of resources to solve instance 1 of size n} \\ + \\ \text{amount of resources to solve instance 2 of size n} \\ + \\ \cdots \\ + \\ \text{amount of resources to solve last instance of size n}\end{array}}{\text{number of instances of size n}}$$

Types of Complexity Functions

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

- **Average case complexity**:

L	x	9	11	17	18	26	29	43	48	55
L	3	x	11	17	18	26	29	43	48	55

…

L	3	9	11	17	18	26	29	43	48	x
x not in L	3	9	11	17	18	26	29	43	48	55

Types of Complexity Functions

In this course we will study worst case complexity.

How do we compute the time complexity of an algorithm?

We need a clock to measure time.

Experimental way of measuring the time complexity

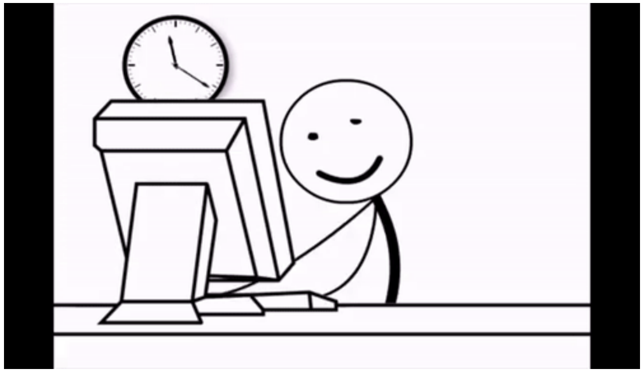
We need:

- 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



Software

Operating System

Compiler

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

1									
33									
5	66								
8	17								
2	47	76							
1	9	11	17						
15	14	32	35	55					
6	14	32	33	65	88				

11	16	32	33	57	88					…
7	12	32	55	57	60	70				…
0	10	20	55	57	60	61				…
3	12	31	44	57	60	88				…
2	4	6	8	10	12	14	16			…
3	6	9	12	15	18	21	24			…
5	12	20	25	30	35	40	45	50		…
3	12	11	17	18	26	29	43	48	55	…

Experimental way of measuring the time complexity

Drawbacks

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

Java program running on computer 1: 12 ms
C program running on computer 2: 8 ms
Python program running on computer 3: 15 ms

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, $0 \le i < n$, 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
```

We wish to compute the time complexity of an algorithm by analyzing only its pseudocode

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.

Primitive Operations

The number of primitive operations performed by an algorithm is proportional to the running time of the algorithm regardless of which programming language and computer is used to execute the algorithm.

Hence, we can determine the running time of an algorithm by counting the number of primitive operations that it performs.

When do we need to compute the time complexity function?

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

Time Complexity	Time			
	n = 10	n = 20	n = 1000	$n = 10^6$
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	10 s	360 yrs
$f(n) = 2^n$	10^{-5} s	10^{-1} s	10^{293} yrs	10^{10^5} yrs

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

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

Constant = independent from n

Note. Some books sometimes write f(n) = O(g(n)) or f(n) ϵ O(g(n)) instead of f(n) is O(g(n)).

"Is an element of"

Computing the Order of a Function

Prove that $4n^2+3n$ is $O(n^2)$

We must find constants $c > 0$ and $n_0 \geq 1$ such that

$$4n^2 + 3n \leq c\, n^2 \text{ for all } n \geq n_0. \qquad (1)$$

Simplify the inequality: Move $4n^2$ to the right side of the inequality:

$$3n \leq (c - 4)n^2 \text{ for all } n \geq n_0$$

Divide both sides by n (allowed since $n > 0$):

$$3 \leq (c - 4)n \text{ for all } n \geq n_0$$

Now we can choose, for example, $c = 5$ to get

$$3 \leq (5 - 4)n = n \text{ for all } n \geq n_0$$

The inequality $3 \leq n$ is valid for all values of n which are at least 3, so we can choose, for example $n_0 = 3$.

Since we have found constant values $c = 5$, $n_0 = 3$ that make inequality (1) true, then we have proven that $4n^2+3n$ is $O(n^2)$.

Computing the Order of a Function

Prove that $T_1 n + T_2$ is $O(n)$ where $T_1 > 0$ and $T_2 > 0$ are constants

We must find constants $c > 0$ and $n_0 \geq 1$ such that

$$T_1 n + T_2 \leq c\, n \text{ for all } n \geq n_0. \qquad (2)$$

Simplify the inequality: Move $T_1 n$ to the right side of the inequality:

$$T_2 \leq (c - T_1) n \text{ for all } n \geq n_0$$

Now we can choose, for example, $c = T_1 + 1$ (note that this value is constant) to get

$$T_2 \leq (T_1 + 1 - T_1) n = n \text{ for all } n \geq n_0$$

The inequality $T_2 \leq n$ is valid for all values of n that are at least equal to T_2, so we can choose, for example $n_0 = T_2$.

Since we have found constant values $c = T_1 + 1$, $n_0 = T_2$ that make inequality (2) true, then we have proven that $T_1 n + T_2$ is $O(n)$

Computing the Order of a Function

Prove that $4n$ is $O(n^2)$

We must find constants $c > 0$ and $n_0 \geq 1$ such that

$$4n \leq c\, n^2 \text{ for all } n \geq n_0. \qquad (3)$$

Simplify the inequality: Divide both sides by n (allowed since $n > 0$):

$$4 \leq cn \quad \text{for all } n \geq n_0$$

Now we can choose, for example, $c = 1$ to get

$$4 \leq n = n \text{ for all } n \geq n_0$$

The inequality $4 \leq n$ is valid for all values of n larger than or equal to 4, so we can choose, for example $n_0 = 4$.

Since we have found constant values $c = 1$, $n_0 = 4$ that make inequality (3) true, then we have proven that $4n$ is $O(n^2)$.

Computing the Order of a Function

Prove that n^2 is not $O(n)$

We will use a proof by contradiction: Assume that the claim is false, i.e., n^2 is $O(n)$ and derive a contradiction from this assumption.

If n^2 is $O(n)$ then by definition of big Oh, there are constants $c > 0$ and $n_0 \geq 1$ such that

$$n^2 \leq c\,n \text{ for all } n \geq n_0. \qquad (4)$$

Simplify the inequality: Divide both sides by n (allowed since $n > 0$):

$$n \leq c \text{ for all } n \geq n_0$$

The inequality $n \leq c$ is valid only for values of n that are at most c, so this inequality cannot be true for all values n larger than some constant n_0. Specifically, if we choose $n \geq c + n_0$, then note that these values of n are larger than or equal than n_0 but they are not at most c.

Therefore, we have reached a contradiction as there are no constant values $c > 0$ and $n_0 \geq 1$ such that $n^2 \leq c\,n$ for all $n \geq n_0$. Consequently, n^2 is not $O(n)$.

Computing the Time Complexity of Linear Search

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

Primitive operations: ←, <, ≠, and, +, =, return

Let $t_{\leftarrow}$ be the amount of time needed to perform an assignment operation on a particular programming language and computer

Let $t_{<}$, $t_{\neq}$, t_{+}, $t_{=}$, t_{and}, and t_{ret} be the amount of time needed to perform one of the respective primitive operations on a particular implementation

Computing the Time Complexity of Linear Search

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

	←	<	≠	and	+	=	return
	1						
i = 0	1	1	1	1	1		
i = 1	1	1	1	1	1		
i = 2	1	1	1	1	1		
…							
i = n-1	1	1	1	1	1		
i = n		1		1		1	1
total:	n+1	n+1	n	n+1	n	1	n

$$f(n) = (n+1)\, t_{\leftarrow} + (n+1)\, t_{<} + n\, t_{\neq} + (n+1)\, t_{and} + n\, t_{+} + t_{=} + t_{ret} =$$

$$\underbrace{(t_{\leftarrow} + t_{<} + t_{\neq} + t_{and} + t_{+})}_{T_1} n + \underbrace{(t_{\leftarrow} + t_{<} + t_{and} + t_{=} + t_{ret})}_{T_2} = T_1 n + T_2 \text{ is } O(n)$$

Second Method

```
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
```

To compute the time complexity we can just count the total number of primitive operations without counting the number of each specific operation.

Computing the Time Complexity of Linear Search

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

	number of primitive operations
	1
i = 0	5
i = 1	5
i = 2	5
…	
i = n-1	5
i = n	9
total:	5n + 5

$f(n) = 5n + 5$ is $O(n)$

Third Method

Algorithm LinearSearch (L,n,x)

c_1 i ← 0

while (i < n) **and** (L[i] ≠ x) **do**

    i ← i+1

c_3

The loop is repeated once for i = 0, 1, …, n-1, so the total number of iterations is n

c_2 **if** i= n **then return** -1

else return i

$$f(n) = \underbrace{c_1 + c_2}_{c} + c_3 n \text{ is } O(n)$$

Time Complexity Example

```
Algorithm foo (n)
i ← 1
k ← 1
while  i < n do
    if i = k then {
        A[i] ← k
        k ← k + 1
        i ← 1
    }
    else i ← i + 1
```

Rules for Computing the Order of a Function

Rule 1. $k\, f(n)$ is $O(f(n))$ for any constant $k > 0$.

Proof

We must find constants $c > 0$ and $n_0 \geq 1$ such that

$$k\, f(n) \leq c\, f(n) \text{ for all } n \geq n_0. \qquad (1)$$

Move $k\, f(n)$ to the right side of the inequality:

$$0 \leq c\, f(n) - k\, f(n) = (c - k)\, f(n) \text{ for all } n \geq n_0.$$

Hence, we can choose $c = k$ to get

$$0 \leq (c - c)\, f(n) = 0 \text{ for all } n \geq n_0$$

Since this inequality holds for all integer values $n \geq 1$ we can choose $n_0 = 1$.

As we have found constant values $c = k$ and $n_0 = 1$ for which the inequality (1) holds, then we have proven that $k\, f(n)$ is $O(f(n))$.

Rules for Computing the Order of a Function

Rule 2. f(n) + g(n) is O(maximum{f(n),g(n)}).

Proof

We must find constants $c > 0$ and $n_0 \geq 1$ such that

$$f(n) + g(n) \leq c \text{ maximum}\{f(n),g(n)\} \text{ for all } n \geq n_0. \qquad (2)$$

Note that

$$f(n) \leq \text{maximum } \{f(n), g(n)\} \text{ for all } n \geq 1, \text{ and}$$

$$g(n) \leq \text{maximum } \{f(n), g(n)\} \text{ for all } n \geq 1$$

Adding these two last inequalities we get

$$f(n) + g(n) \leq 2 \text{ maximum}\{f(n),g(n)\} \text{ for all } n \geq 1$$

Hence, choosing $c = 2$ and $n_0 = 1$ prove that inequality (2) is true, so f(n)+g(n) is O(maximum{f(n),g(n)}.

Compute the Order of the Following Functions

- $5n^3+3n^2$-3
- $6\,n\log n + 3/\,n$
- $1000\,n + 0.00001\,n^2$
- $\frac{n(n+1)}{2}$ - $3n$