# Algorithms and Data Structures 2

## 2 - Algorithm analysis

**Dr Michele Sevegnani**

School of Computing Science
University of Glasgow

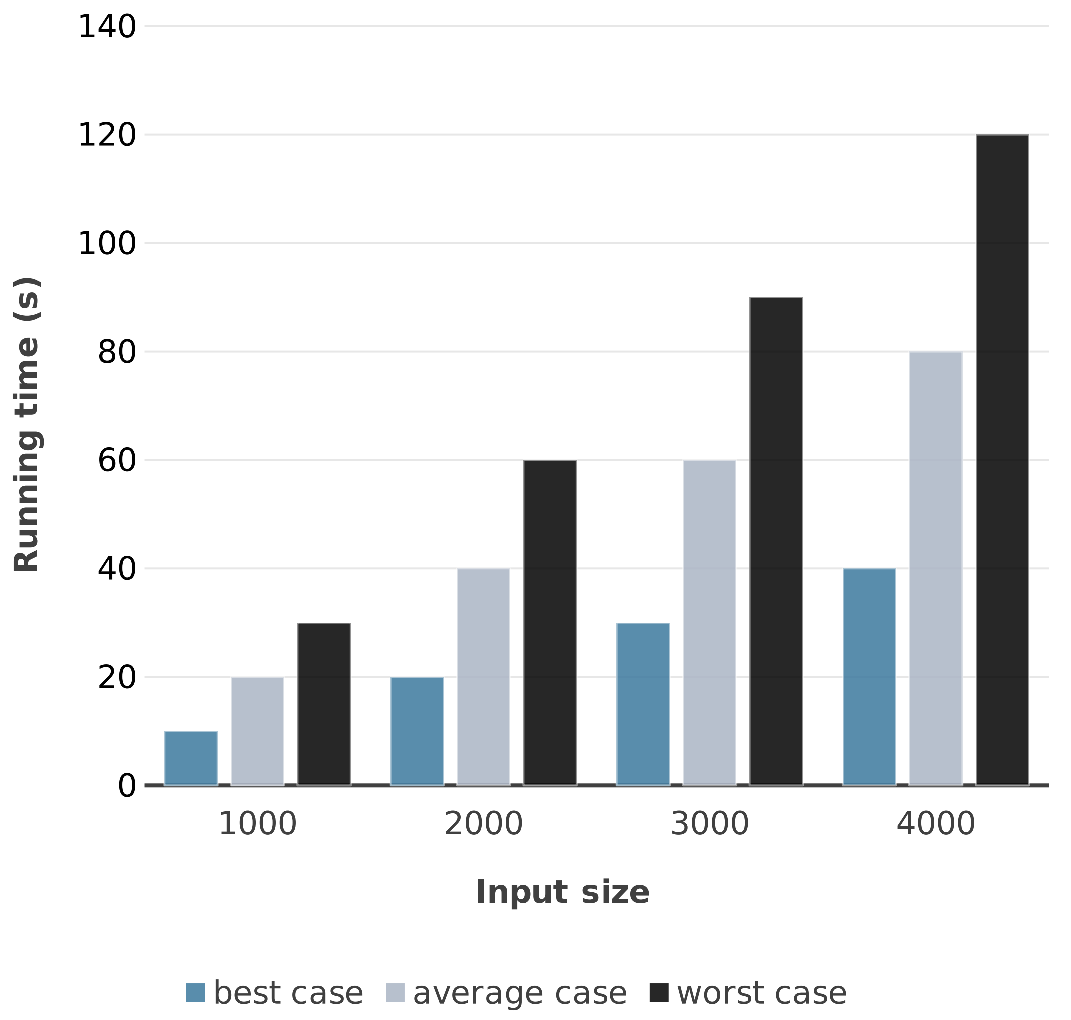
*michele.sevegnani@glasgow.ac.uk*

## Outline

- **Running time**
- **Experimental studies vs theoretical analysis**
- **Counting primitive operations**
- **Growth rate of running time**
- **Big-Oh notation**

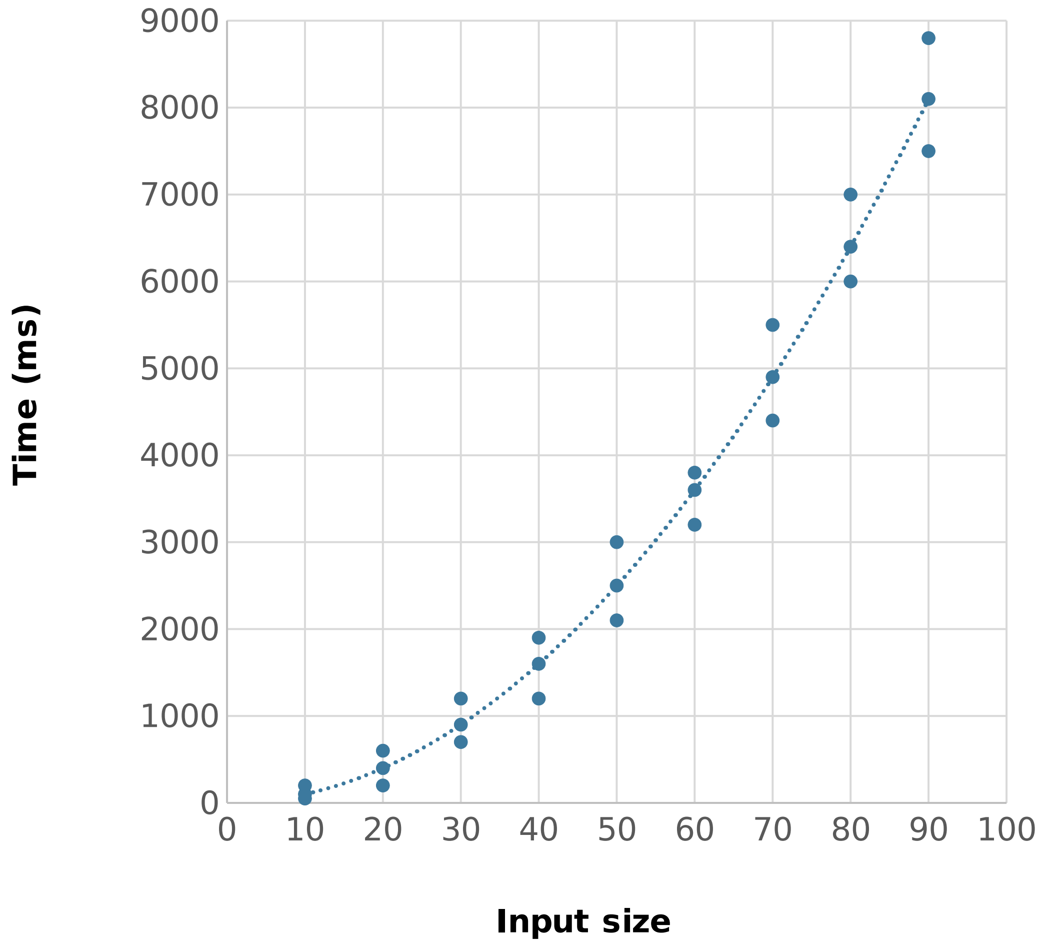
## Running time

- **Most algorithms transform input objects into output objects**
- **The running time of an algorithm typically grows with the input size**
- **Average case time is often difficult to determine**
- **We mainly focus on the worst case running time**
  - Easier to analyse
  - Crucial to applications such as games, finance and robotics



## Experimental studies

- **Write a program implementing the algorithm**
- **Run the program with inputs of varying size and composition**
- **Use a method like** `System.currentTimeMillis()` **to get an accurate measure of the actual running time**
- **Plot the results**
- **This is also called empirical algorithmics or performance profiling**



## Experimental studies vs theoretical analysis

### Limitations of experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment
- In order to compare two algorithms, the same hardware and software environments must be used

### Theory

- Uses a high-level description of the algorithm instead of an implementation
- Characterises running time as a function of the input size (n)
- Considers all possible inputs
- Allows us to evaluate the speed of an algorithm independently of the hardware/software environment

## Primitive operations

- **Basic computations performed by an algorithm**
- **Identifiable in pseudocode**
- **Largely independent from the programming language**
- **Exact definition not important (we will see why later)**
- **Assumed to take a constant amount of time**

- **Examples**
  - Expression evaluation
  - Value assignment to a variable
  - Array indexing
  - Method call
  - Returning from a method

## Counting primitive operations

- **By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size**
- **Example: find the maximum value in an array A of integers (with indices between 0 and n-1)**

```
ARRAY-MAX(A)
  max := A[0]
  for i = 1 to n-1
    if A[i] > max then
      max := A[i]
  {increment counter i}
  return max
```

Operations

## Counting primitive operations

- **By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size**
- **Example: find the maximum value in an array A of integers (with indices between 0 and n-1)**

```
ARRAY-MAX(A)
  max := A[0]
  for i = 1 to n-1
    if A[i] > max then
      max := A[i]
  {increment counter i}
  return max
```

Operations

2 assignment and array indexing

## Counting primitive operations

- **By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size**
- **Example: find the maximum value in an array A of integers (with indices between 0 and n-1)**

```
ARRAY-MAX(A)
  max := A[0]
  for i = 1 to n-1
    if A[i] > max then
      max := A[i]
  {increment counter i}
  return max
```

Operations

| | |
|---|---|
| 2 | |
| 2n | assignment and test |

In general, the loop header is executed one time more than the loop body

## Counting primitive operations

- **By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size**
- **Example: find the maximum value in an array A of integers (with indices between 0 and n-1)**

```
ARRAY-MAX(A)                      Operations
  max := A[0]                         2
  for i = 1 to n-1                    2n
    if A[i] > max then                2(n-1) array indexing and test
      max := A[i]
  {increment counter i}
  return max
```

## Counting primitive operations

- **By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size**
- **Example: find the maximum value in an array A of integers (with indices between 0 and n-1)**

```
ARRAY-MAX(A)
  max := A[0]
  for i = 1 to n-1
    if A[i] > max then
      max := A[i]
  {increment counter i}
  return max
```

| Operations | |
|---|---|
| 2 | |
| 2n | |
| 2(n-1) | |
| 2(n-1) | array indexing and assignment |
| 2(n-1) | |

Worst case analysis

## Counting primitive operations

- **By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size**
- **Example: find the maximum value in an array A of integers (with indices between 0 and n-1)**

| ARRAY-MAX(A) | Operations |
|---|---|
| max := A[0] | 2 |
| **for** i = 1 **to** n-1 | 2n |
| **if** A[i] > max **then** | 2(n-1) |
| max := A[i] | 2(n-1) |
| {increment counter i} | 2(n-1) assignment and addition |
| **return** max | |

```
ARRAY-MAX(A)
  max := A[0]
  for i = 1 to n-1
    if A[i] > max then
      max := A[i]
  {increment counter i}
  return max
```

## Counting primitive operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size
- Example: find the maximum value in an array **A** of integers (with indices between **0** and **n-1**)

```
ARRAY-MAX(A)                      Operations
  max := A[0]                       2
  for i = 1 to n-1                  2n
    if A[i] > max then              2(n-1)
      max := A[i]                   2(n-1)
  {increment counter i}             2(n-1)
  return max                        1        return
```

## Counting primitive operations

- **By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size**
- **Example: find the maximum value in an array A of integers (with indices between 0 and n-1)**

```
ARRAY-MAX(A)
  max := A[0]
  for i = 1 to n-1
    if A[i] > max then
      max := A[i]
  {increment counter i}
  return max
```

| | Operations |
|---|---|
| max := A[0] | 2 |
| for i = 1 to n-1 | 2n |
| if A[i] > max then | 2(n-1) |
| max := A[i] | 2(n-1) |
| {increment counter i} | 2(n-1) |
| return max | 1 |
| Total | 8n - 3 |

## Estimating running time

- **Algorithm ARRAY-MAX executes 8n - 3 primitive operations in the worst case**
- **Define**
  - a = time taken by the fastest primitive operation
  - b = time taken by the slowest primitive operation
- **Let T(n) be worst-case time of ARRAY-MAX**
- **The following holds: $a\,(8n - 3) \le T(n) \le b\,(8n - 3)$**
- **We say the running time T(n) is bounded by two linear functions (i.e. polynomials of degree 1)**

- **What is the number of primitive operations in the best case?**
  - How does the input look like?

## Growth rate of running time

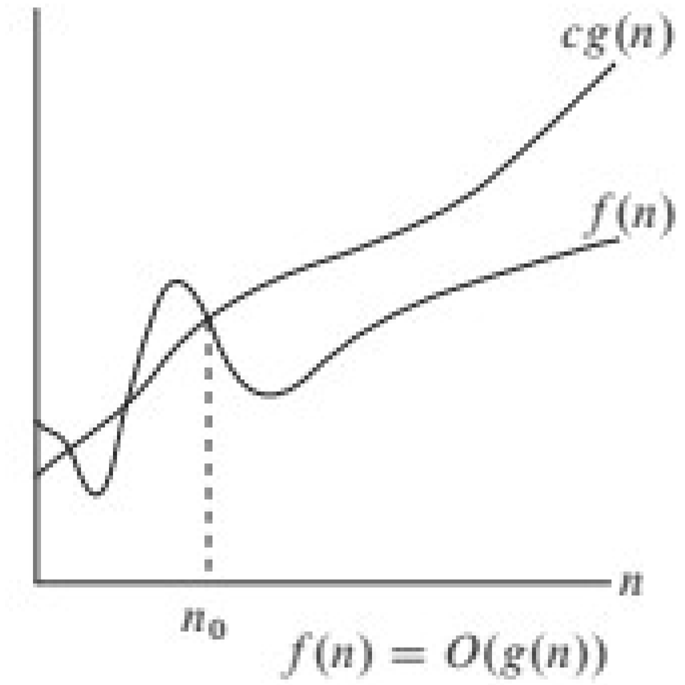
- **Changing the hardware/software environment**
  - Affects T(n) by a constant factor
  - Does not alter the growth rate of T(n)
- **The linear growth rate of the running time T(n) is an intrinsic property of algorithm ARRAY-MAX**
  - We need to scan the entire array to find the maximum
  - No assumptions on the input
- **Assume we know the input is always sorted in a descending order**
  - We could get the maximum by simply accessing the first element A[0], i.e. constant growth rate
  - We will exploit this fact to define data structures that support finding the maximum more efficiently (e.g. priority queues and heaps)

## Constant factors

- **The growth rate is not affected by**
  - constant factors
  - lower-degree terms
- **Examples**
  - $10^2 n + 10^5$ is a linear function
  - $10^2 n$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function
  - …

## Big-Oh notation

- Given functions **f(n)** and **g(n)**, we say that **f(n)** is **O(g(n))** if there are positive constants **c** and $n_0$ such that $f(n) \le cg(n)$ for $n \ge n_0$
- O-notation gives an **upper bound** for a function to within a constant factor



## Big-Oh and growth rate

- **The big-Oh notation gives an upper bound on the growth rate of a function**
- **The statement "f(n) is O(g(n))" means that the growth rate of f(n) is no more than the growth rate of g(n)**
- **We can use the big-Oh notation to rank functions according to their growth rat**

| | f(n) is O(g(n)) | g(n) is O(f(n)) |
|---|---|---|
| g(n) grows more | Yes | No |
| f(n) grows more | No | Yes |
| Same growth | Yes | Yes |

## Asymptotic algorithm analysis

- **When we look at input sizes large enough to make only the growth rate of the running time relevant, we are studying the asymptotic efficiency of algorithms**
- **We analyse how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound**
- **Big-Oh notation is used to express asymptotic upper bounds**

## Big-Oh rules

- **If $f(n)$ is a polynomial of degree $d$, then $f(n)$ is $O(n^d)$**
  - Drop lower-order terms
  - Drop constant factors
- **Use the smallest possible class of functions**
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- **Use the simplest expression of the class**
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

## Some examples

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$

- **$2n + 10$ is $O(n)$**

  $2n + 10 \leq cn$

  $n(c - 2) \geq 10$

  $n \geq 10/(c - 2)$

  Pick $c = 3$ and $n_0 = 10$

- **$n^2$ is not $O(n)$**

  $n^2 \leq cn$

  $n \leq c$

  Since $c$ must be a constant, inequality cannot hold for all $n$

## Some examples

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$

- **7n-2 is O(n)**

  $7n-2 \leq cn$

  Pick $c = 8$ and $n_0 = 2$

- **$3n^3 + 20n^2 + 5$ is $O(n^3)$**

  $3n^3 + 20n^2 + 5 \leq cn^3$

  Pick $c = 4$ and $n_0 = 21$

- **3 log n + 5 is O(log n)**

  $3 \log n + 5 \leq c \log n$

  Pick $c = 4$ and $n_0 = 32$

log stands for $\log_2$
$\log_2 2 = 1$

## Big-Oh rules (cont.)

- **If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$ then $T_1(n) + T_2(n) = O(\max(f(n),g(n)))$**
- **Proof**
  - Recall

    Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$

  - There are constants $c_1$ and $c_2$ such that $T_1(n) \leq c_1 f(n)$ and $T_2(n) \leq c_2\, g(n)$ for some $n$ sufficiently large
  - Then $T_1(n) + T_2(n) \leq c_1 f(n) + c_2\, g(n)$
  - $T_1(n) + T_2(n) \leq c\,(f(n) + g(n))$, with $c = \max(c_1, c_2)$
  - Using $a + b \leq 2 \max(a,b)$ we have $T_1(n) + T_2(n) \leq 2c \max(f(n),g(n))$
  - Hence, $T_1(n) + T_2(n)$ is $O(\max(f(n),g(n)))$

## Big-Oh rules (cont.)

- **If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$ then $T_1(n)\ T_2(n) = O(f(n)\ g(n))$**
- **Proof**
  - Recall

    Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for $n \geq n_0$

  - $T_1(n) \leq c_1 f(n)$ and $T_2(n) \leq c_2\ g(n)$ so $T_1(n)\ T_2(n) \leq c_1 c_2\ f(n)\ g(n)$
  - Hence, $T_1(n)\ T_2(n)$ is $O(f(n)\ g(n))$

- **If $T(n) = (\log n)^k$ then $T(n) = O(n)$**
  - True for any value of $k$
  - Proof at the end

## Rules to compute running times

- **Rule 1 – Loops**
  - The running time of a loop is at most the running time of the statements inside the loop (including tests) multiplied by the number of iterations
- **Rule 2 – Nested loops**
  - Should be analysed inside out. Total running time of a statement inside a group of nested loops is running time of statement multiplied by the product of the sizes of all the loops

```
ALG1(n)
  for i = 0 to n-1
   for j = 0 to n-1
     for k = 0 to n-1
        increment x
```

$O(n^3)$

## Rules to compute running times

- **Rule 3 – Consecutive statements**
  - These just add (so take the maximum - see slide 24)
- **Rule 4 – If-then-else**
  - Running time is never more than the time of the test (condition) plus the maximum of the running times of the two branches
  - Similarly for multiple/nested else statements

## Analysis of INSERTION-SORT

- **Input: an array A of integers (with indices between 0 and n-1)**
- **Output: a permutation of the input such that A[0] ≤ A[1] ≤ … ≤ A[n-1]**
- **We ignore constants**

```
INSERTION-SORT(A)                              Operations
  for j = 1 to n-1                                 n
      key := A[j]                                  n-1
      i := j-1                                     n-1
      while i ≥ 0 and A[i] > key
        A[i+1] := A[i]                             -1)
        i := i-1                                   -1)
      A[i+1] := key                                n-1
```

# Analysis of INSERTION-SORT (cont.)

- **The running time is computed by summing the number of operations**
  - T(n)= n + (n-1) + (n-1) +  + -1) + (n-1)
- **Best case: array already sorted**
  - A[i] $\leq$ key then while loop is never executed: $t_j = 1$
  - T(n) = n + (n-1) + (n-1) + (n-1)+ 0 + 0 + (n-1) = O(n)
- **Worst case**
  - At every iteration of the while loop we need to shift j elements: $t_j = j$
  - =  = n(n-1)/2 = O($n^2$)
  - = n(n-1)/2 – (n – 1) = O($n^2$)
  - T(n) = n + (n-1) + (n-1) + O($n^2$) + O($n^2$) + O($n^2$) + (n-1) = O($n^2$)

Summation rules