Contents

Introduction
Resources consumptions: spatial and temporal cost
Costs characterization
Cost boundaries: best, worst and average cases
Asymptotic notation: computational costs hierarchy
Additional aspects in computational complexity
Generally, a problem can be solved by using several algorithms or programs. Although, not all the solutions are equally good.

How can we *measure* how good a program (that solves a problem) is?

• How elegant is the idea in which the algorithm is based on?
• How clear is the program code organized?
• How showiness are the results presented?
• How efficient the algorithm is in obtaining the solution?

From a *computational* point of view, the essential factor is *efficiency*. 
**Introduction**

**Algorithm analysis** is a very important activity in the development process, especially in constrained resource environments (in time or memory).

Algorithm analysis is required in order to:

- Compare two or more different algorithms.
- Foresee the behavior of an algorithm in extreme conditions.
- Adjust the algorithm parameters to get the best results.
Algorithm analysis can be carried out in two different ways:

- empirically (experimental, a posteriori), or
- theoretically (a priori)

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>empirically</td>
<td>“Easy”, “real” results</td>
<td>We need same quality in the implementations, real data, machine dependency</td>
</tr>
<tr>
<td>theoretically</td>
<td>Flexible, “cheap”, machine independent</td>
<td>Not “exact” results</td>
</tr>
</tbody>
</table>
**Resources consumption: spatial and temporal costs**

**Efficiency:** capacity of solving the proposed problem using a low consumption of computational resources.

Two fundamental factors of efficiency:

- **Spatial cost** or quantity of memory required
- **Temporal cost** or time required to solve the problem

In order to solve a given problem, an algorithm or program A will be better than another B if A solves the problem in less time and/or uses less quantity of memory than B.

Generally there is a tradeoff between time and space, so a good algorithm is the one that solves the problem with a good *commitment*.

We will focus our attention mainly on *temporal cost*. 
Sometimes, just time or memory are not only suitable in order to appreciate the quality of a program. Let’s consider three simple programs to calculate $10^2$:

```c
int main(){ /*A1*/
    int m;
    m = 10 * 10; /*producto*/
    printf("%d\n", m);
}
```

```c
int main(){ /*A2*/
    int i,m; m=0;
    for (i=1; i<=10; i++)
        m = m + 10; /*suma*/
    printf("%d\n", m);
}
```

```c
int main(){ /*A3*/
    int i,j,m; m=0;
    for (i=1; i<=10; i++)
        for (j=1; j<=10; j++)
            m++; /*sucesor*/
    printf("%d\n", m);
}
```

Be $t_*$, $t_+$, $t_s$ the times required to carried out a \textit{product}, \textit{sum}, and \textit{successor}.

$$T_{A1} = t_* \quad T_{A2} = 10t_+ \quad T_{A3} = 100t_s$$
Let's assume that A1, A2, A3 are executed in four different computers with different characteristics (different times for successor, sum and product execution).

<table>
<thead>
<tr>
<th></th>
<th>t∗</th>
<th>100 µs</th>
<th>50 µs</th>
<th>100 µs</th>
<th>200 µs</th>
</tr>
</thead>
<tbody>
<tr>
<td>t+</td>
<td>t+</td>
<td>10 µs</td>
<td>10 µs</td>
<td>5 µs</td>
<td>10 µs</td>
</tr>
<tr>
<td>ts</td>
<td>ts</td>
<td>1 µs</td>
<td>2 µs</td>
<td>1 µs</td>
<td>0.5 µs</td>
</tr>
<tr>
<td>A1</td>
<td>A1</td>
<td>100 µs</td>
<td>50 µs</td>
<td>100 µs</td>
<td>200 µs</td>
</tr>
<tr>
<td>A2</td>
<td>A2</td>
<td>100 µs</td>
<td>100 µs</td>
<td>50 µs</td>
<td>100 µs</td>
</tr>
<tr>
<td>A3</td>
<td>A3</td>
<td>100 µs</td>
<td>200 µs</td>
<td>100 µs</td>
<td>50 µs</td>
</tr>
</tbody>
</table>

Which program is the best, A1, A2 or A3?

For each computer there’s a best one.
Our “simple problem” was too restrictive. In a normal situation we would like to calculate not only $10^2$ but, in general, $n^2$, where $n$ is a program data.

With this new point of view, the specific problem of calculating $10^2$ is considered as an instance of the general problem of calculating $n^2$.

Given a problem, in general not all of its instances are of the same size. The size of an instance is an integer number that measures how big this instance is.

In our problem of calculating $n^2$, a natural measurement of the size is precisely $n$. So, the size of the instance $10^2$ is 10 and the size of the instance $2345678^2$ is 2345678.
Cost as a function of the size of the problem

We can easily rewrite A1, A2, A3 in order to calculate $n^2$ instead of $10^2$.

```c
int main(){ /*A1*/
    int n, m;
    scanf("%d", &n);
    m = n * n; /*producto*/
    printf("%d\n", m);
}
```

```c
int main(){ /*A2*/
    int i, n, m; m=0;
    scanf("%d", &n);
    for (i=1; i<=n; i++)
        m = m + n; /*suma*/
    printf("%d\n", m);
}
```

```c
int main(){ /*A3*/
    int i, j, n, m; m=0;
    scanf("%d", &n);
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
            m++; /*sucesor*/
    printf("%d\n", m);
}
```

Size=$n$; temporal costs ($\mu$s): $T_{A1} = t_* \quad T_{A2} = t_+ \cdot n \quad T_{A3} = t_* \cdot n^2$

Cost still depends on $t_*, t_+, t_!$
Intuitively, it seems clear that A1 is the best program and A3 the worst. But, even fixing the computer characteristics, we will see that our intuition is not clearly kept.

For example, if we fix $t_\star = 100 \ \mu s$, $t_+ = 5 \ \mu s$, $t_s = 1 \ \mu s$:

<table>
<thead>
<tr>
<th></th>
<th>$T_{A_i}(n)$</th>
<th>$n = 4$</th>
<th>$n = 10$</th>
<th>$n = 1000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>100</td>
<td>100 $\mu s$</td>
<td>100 $\mu s$</td>
<td>100 $\mu s$</td>
</tr>
<tr>
<td>$A_2$</td>
<td>5n</td>
<td>20 $\mu s$</td>
<td>50 $\mu s$</td>
<td>5000 $\mu s$</td>
</tr>
<tr>
<td>$A_3$</td>
<td>$n^2$</td>
<td>16 $\mu s$</td>
<td>100 $\mu s$</td>
<td>1000000 $\mu s$</td>
</tr>
</tbody>
</table>

**Which is the better, A1, A2, A3?**

For each instance there’s a best one.

A good cost characterization should allow to establish the program “quality” independently of the computer and of the particular sizes of the instances to process.
In general, we should appreciate a relative behavior of A1, A2, A3 as follows:

Calculus of $n^2$: relative costs of A1, A2, A3

A good computational characterization of a program:

**Cost functional dependency with the size of input - for large sizes!**
• Generally, programs are useful to solve problems of large sizes of input (if they are small, we could solve them manually)

• Considering large sizes of input, we can carry out simple approximations that simplify considerably cost analysis

• Programs no longer depend on specific execution time values of the different elementary instructions used (if they don’t depend on the size of input), and don’t depend on sizes of input of specific instances of the program to solve
Simplification of cost analysis: the “step” concept

A **STEP** is the execution of a code segment which processing time doesn’t depend on the size of input of the considered program, or it is bounded by a constant.

Computational cost of a program: number of **STEPS** as a function of the size of input of the program

*Elements to which 1 STEP can be assigned:*
- Assignment, arithmetic or logical operations, comparison, access to a vector or matrix element, etc.
- Any finite sequence of the previous instructions which length doesn’t depend on the size of input.

*Elements to which 1 STEP cannot be assigned, but a number of STEPS as a function of the size of input:*
- Assignment of structured variables (ex: arrays) which number of elements depends on the size of input
- Loops (recursion) which number of iteration (calls) depends on the size of input
Cost analysis (STEPS) of $n^2$ programs

```c
int main() { /*A1*/
    int n, m;
    scanf("%d", &n);
    m = n * n; /*producto*/
    printf("%d\n", m);
}

int main() { /*A2*/
    int i, n, m; m=0;
    scanf("%d", &n);
    for (i=1; i<=n; i++)
        m = m + n; /*suma*/
    printf("%d\n", m);
}

int main() { /*A3*/
    int i, j, n, m; m=0;
    scanf("%d", &n);
    for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
            m++; /*sucesor*/
    printf("%d\n", m);
}
```

$$T_{A1}(n) = 1, \quad T_{A2}(n) = n, \quad T_{A3}(n) = n^2$$
Another example on cost analysis with STEPS
(printing the elements of an array in reverse order)

```c
#include <stdio.h>
#define MAXN 1000000
int main()  /* invOrd.c */
{
    int i, n, x, v[MAXN];

    n = 0;    /* Inicializa contador de datos (talla) */
    printf("Teclear datos (fin: `\d\n");
    while ((n < MAXN) && (scanf("%d", &x) == 1)) {  /* Lee datos */
        v[n] = x;
        n++;
        /* Actualiza talla */
    }
    /* Imprime resultados en orden inverso */
    for (i = n-1; i >= 0; i--)
    {
        printf("%d ", v[i]);
        printf("\n");
    }
    return 0;
}
```

**STEP:** Accesses to \( v \);
**Size of input = n;**
**\( T_{invOrd}(n) = n+n=2n \) STEPS**
More examples on cost analysis with STEPS
(searching the closest element to the average)

```c
#include <stdio.h>
#include <math.h>
#define MAX 100000

int main()
{
    int cercano, i, n = 0, aux, A[MAX];
    double media, suma;

    while (n < MAX && scanf("%d", &aux) == 1) {
        A[n] = aux; n++;
    }
    suma = 0;
    for (i = 0; i < n; i++) suma += A[i];
    media = suma/n; cercano = 0; /* Cálculo de la media */
    for (i = 1; i < n; i++) /* Búsqueda del más cercano */
        if (fabs(A[i]-media) < fabs(A[cercano]-media))
            cercano = i;
    printf("El más cercano es A[%d]=%d\n", cercano, A[cercano]);
}
```

STEP: Accesses to A; Size of input = n; \( T_{\text{closest}}(n) = ??? \) STEPS
More examples on cost analysis with STEPS
(statistical mode for a series of ages: data input)

```c
#include <stdio.h>
#define MAXDATOS 100000
#define MAXEDAD 150
int main()
{   /* modax.c: cálculo de la moda */
    int i, j, n, edad, frec, maxFrec, moda, edades[MAXDATOS];
    n = 0;            /* Inicializa contador de datos (talla) */
    printf("Teclear edades, finalizando con \^D\n");
    while ((n < MAXDATOS) && (scanf("%d", &edad) != EOF)) /* Lee edades */
        if ((edad >= 0) && (edad < MAXEDAD)) { /* hasta EOF, */
            edades[n] = edad;                    /* las memoriza */
            n++;                                /* y actualiza n */
        }
}
```
More examples on cost analysis with STEPS (inefficient calculus of statistical mode)

/* moda0.c: cálculo (ineficiente) de la moda */

maxFrec=0;
for (i = 0; i < n; i++) {
    frec = 0;
    for (j = 0; j < n; j++)
        if (edades[i] == edades[j])
            frec++;
    if (frec > maxFrec)
        maxFrec = frec;
    moda = edades[i];
}
printf("Leidos %d datos; Moda=%d (frecuencia=%d)\n", n, moda, maxFrec);
return 0;

STEP: if comparisons; Size of input = n; T_{\text{mod}}(n) = ?? STEPS
More examples on cost analysis with STEPS
(more efficient calculus of statistical mode)

```c
/* moda1.c: cálculo (poco eficiente) de la moda */

maxFrec = 0; /* Explora edades[] maxEdad veces para */
for (i = 0; i < MAXEDAD; i++) { /* determinar cuál es la edad */
  frec = 0;
  for (j = 0; j < n; j++)
    if (edades[j] == i) frec++;
  if (frec > maxFrec) {
    maxFrec = frec;
    moda = i;
  }
}
printf("Leidos %d datos; Moda=%d (frecuencia=%d)\n", n, moda, maxFrec);
return 0;
}
```

STEP: if comparisons; Size of input = n; $T_{mod}(n) = ???$ STEPS
#include <stdio.h>
#define MAXEDAD 150
int main() /* moda.c: cálculo eficiente de la moda */
{
    int n = 0, edad, maxFrec = 0, moda, frecs[MAXEDAD];
    /* Inicializa vector de frecuencias */
    for (edad = 0; edad < MAXEDAD; edad++)
        frecs[edad] = 0;
    printf("Teclear edades, fin con ~D\n"); /* Lee edades hasta EOF */
    while (scanf("%d", &edad) != EOF) /* y actualiza frecuencias */
        if ((edad >= 0) && (edad < MAXEDAD))
            n++; frecs[edad]++;
    for (edad = 0; edad < MAXEDAD; edad++) /* máx. frecuencia (moda) */
        if (frecs[edad] > maxFrec) /* STEP: if comparisons; Size of input = n; T_{mod}(n) = ??? STEPS */
            maxFrec = frecs[edad]; moda = edad;
    printf("Leidos %d datos; Moda=%d (frecuencia=%d)\n", n, moda, maxFrec);
}
Costs for the different mode approaches

![Graph showing costs for different modes](image)

Time (seconds) vs Size of input (number of ages processed)
Costs for the different mode approaches (efficient approaches)
More examples on cost analysis with STEPS (weird functions)

```c
#include <stdio.h>

int f1(int x, int n)
{
    int i;
    for (i = 1; i <= n; i++)
        x += i;
    return x;
}

int f2(int x, int n)
{
    int i;
    for (i = 1; i <= n; i++)
        x += f1(i, n);
    return x;
}

int main()
{
    int a, n;
    scanf("%d", &n);
    a = f2(0, n);
    printf("%d\n", f1(a, n));
    return 0;
}
```

STEP: +=; Size of input = n; $T_{main}(n) = ???$ STEPS
Sometimes, cost is not (only) a function of the size of input

In the examples viewed so far, all the “instances” of a problem had the same computational cost. But this is not always the case. In the following example, which is the cost of the function busca(n) ?

```c
#include <stdio.h>
#define MAXN 1000000

int busca(int *v, int n, int x) /* busca x en v[0...n-1] */
{
    int i;
    for (i=0; i<n; i++)
        if (v[i] == x)
            return i;
    return -1;
}
```
Sometimes, cost is not (only) a function of the size of input

```c
int main() /* busca.c */
{
    int i, x, aux;
    int v[MAXN], n = 0;    /* Vector donde buscar y su talla */
    printf("Teclear datos, fin con ~D)\n");
    while ((n < MAXN) && (scanf("%d", &aux) != EOF)) { /* lee v[] */
        v[n] = aux;
        n++;
    }
    printf("Dato a buscar: ");
    while (scanf("%d", &x) == 1) {
        i = busca(v, n, x);
        printf("Posición de %d: %d\n", x, i);
    }
    printf("Dato a buscar: ");
    return 0;
}
```

**STEP:** if comparison;  size of input=n;  $T_{busca}(n) = ???$

Depends on the array contents and the particular value to look for!
Cost extremes: best, worst and average cases

\[ T^b_{busca}(n) = 1 \quad T^w_{busca}(n) = n \quad T^m_{busca}(n) = n/2 \]
Sometimes it is difficult to calculate the “exact” costs for a size of input. Example:

```c
#include <stdio.h>
define K 64
define maxN 1000000

int main() /* raro.c: Realiza un extraño proceso sobre un vector */
{
    int i, j, k, n, x, X[maxN], pasos;

    n = 0;
pasos = 1;
    printf("Teclear datos\n");
    while ((n < maxN) && (scanf("%d", &x) != EOF)) { /* lee datos */
        X[n] = x;
        n++;
    }
}```
Sometimes it is difficult to calculate the "exact" costs for a size of input. Example:

```c
for (i = 2; i < n; i++) {
    for (k = 1; k <= K; k++)
        X[i] = X[i] + k;
    if (n % 3 == 0) {
        for (k = 1; k <= K; k++)
            X[i] = X[i] - k;
        if ((i % 2 == 0) && (X[i] != X[i-1]))
            for (j = i; j <= n - 2; j++)
                for (k = 1; k <= K; k++)
                    X[i] = k * X[i];
    } else
        for (j = i; j <= n - i; j = j + 2)
            if (X[i] % 2 == 0)
                for (k = 1; k <= K; k++)
                    X[i] = X[i] + 2 * k;
    }
for (i = 0; i < n; i++) printf("%d\n", X[i]);
```

STEP: accesses to X. The total number varies in different ways with the instances of the size of input depending if that is multiple or not of 3.
Generally, it is enough to determine the asymptotic boundaries, i.e. for large size of inputs.
Essential aspects in computational costs determination

• Cost cannot be expressed with just one value. It has to be expressed as a **cost function** the size of input of the problem to solve.

• Comparison of cost functions has to be **insensitive to “implementation constants”**, as particular execution times of individual operations (which costs is independent of the size of input).

• The independency from the constants is achieved by only considering the **asymptotic** behavior of the function cost (i.e., for **large sizes of input**).

• Frequently, for a given size of input, there are several instances with different costs, therefore the **cost cannot be properly expressed as a function of the size of input**.

• In such cases, it is convenient to determine **superior and inferior boundaries** of the cost function, i.e., in the **best and worst cases**.

• There are even situations in which the boundaries for best and worst cases are complex functions. Generally, it will be enough to determine **simple functions** that **bound superiorly and inferiorly the costs of all the instances for large sizes of input**.
Asymptotic notation

Abstraction of constants associated to STEP concept, notion of “large sizes of input” and “asymptotic boundaries”

Be $f : \mathbb{N} \rightarrow \mathbb{R}^+$ a function from natural numbers to the positive real. It is defined:

- $O(f(n)) = \{ t : \mathbb{N} \rightarrow \mathbb{R}^+ | (\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}), \forall n \geq n_0 \ t(n) \leq cf(n) \}$

- $\Omega(f(n)) = \{ t : \mathbb{N} \rightarrow \mathbb{R}^+ | (\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}), \forall n \geq n_0 \ t(n) \geq cf(n) \}$

- $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

From the definition of $\Theta(f(n))$ we have:

\[ \Theta(f(n)) = \{ t : \mathbb{N} \rightarrow \mathbb{R}^+ | (\exists c_1, c_2 \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}), \forall n \geq n_0 \ c_1 f(n) \leq t(n) \leq c_2 f(n) \} \]
Examples

\[ t(n) = 3n + 2 \]

\[ t(n) = 100n + 6 \]

\[ t(n) = 10n^2 + 4n + 2 \]

\[ t(n) = n^2 + 1000n - 100 \]
Examples

\[ t(n) = 3n + 2 \leq 4n \ \forall n \geq 2 \quad \Rightarrow \quad t(n) \in O(n) \]
\[ \geq 3n \ \forall n \geq 1 \quad \Rightarrow \quad t(n) \in \Omega(n) ; \quad t(n) \in \Theta(n) \]

\[ t(n) = 100n + 6 \leq 101n \ \forall n \geq 6 \quad \Rightarrow \quad t(n) \in O(n) \]
\[ \geq 100n \ \forall n \geq 1 \quad \Rightarrow \quad t(n) \in \Omega(n) ; \quad t(n) \in \Theta(n) \]

\[ t(n) = 10n^2 + 4n + 2 \leq 11n^2 \ \forall n \geq 5 \quad \Rightarrow \quad t(n) \in O(n^2) \]
\[ \geq 10n^2 \ \forall n \geq 1 \quad \Rightarrow \quad t(n) \in \Omega(n^2) ; \quad t(n) \in \Theta(n^2) \]

\[ t(n) = n^2 + 1000n - 100 \leq 2n^2 \ \forall n \geq 1000 \quad \Rightarrow \quad t(n) \in O(n^2) \]
\[ \geq n^2 \ \forall n \geq 1 \quad \Rightarrow \quad t(n) \in \Omega(n^2) ; \quad t(n) \in \Theta(n^2) \]
Examples of costs in asymptotic notation

$A1, A2, A3$:  
$T_{A1}(n) \in \Theta(1)$  
$T_{A2}(n) \in \Theta(n)$  
$T_{A3}(n) \in \Theta(n^2)$

Mode: $T_{moda}(n) \in \Theta(n^2)$ (inefficient)  
$T_{moda}(n) \in \Theta(n)$ (efficient)

busca:
Worst case: $O(n)$  
Best case: $\Omega(1)$  
Average case: $O(n)$

raro:
Worst case: $O(n^2)$  
Best case: $\Omega(n)$  
Average case: $O(n^2)$
Asymptotic notation: some useful properties

Order relation among order functions:

- \( f(n) \in \Theta(f(n)) \)
- \( O(f(n)) \subset O(g(n)) \Leftrightarrow f(n) \in O(g(n)) \land g(n) \notin O(f(n)) \)
- \( f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n)) \)

Order of sum of functions: dominant function

- \( (f_i(n) \in \Theta(g_i(n)) \quad 1 \leq i \leq k) \Rightarrow \sum_{i=1}^{k} f_i(n) \in \Theta\left(\max_{1 \leq i \leq k} g_i(n)\right) \)
Asymptotic notation: some useful properties

Limit Rule

For two arbitrary functions $f$ and $g$: $\mathbb{N} \rightarrow \mathbb{R}^+$:

- if $\lim_{n \to \infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+$ then $f(n) \in \Theta(g(n))$

- if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$ then $f(n) \in O(g(n))$ but $f(n) \notin \Theta(g(n))$, and

- if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = +\infty$ then $f(n) \in \Omega(g(n))$ but $f(n) \notin \Theta(g(n))$

For example, $f(n) = \log n$ and $g(n) = \sqrt{n}$. Which is the relative order?

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{\log n}{\sqrt{n}} \xrightarrow{\text{L'Hôpital}} \lim_{n \to \infty} \frac{1/n}{1/(2\sqrt{n})} = \lim_{n \to \infty} \frac{2}{\sqrt{n}} = 0$
Asymptotic notation: some useful properties

Some sums of series

- Arithmetic series \( a_i = a_{i-1} + r \) \( \Rightarrow \)
  \[
  \sum_{i=1}^{n} a_i = n \frac{a_1 + a_n}{2} = a_1 n + \frac{r}{2} n(n - 1)
  \]

- \( \sum_{i=1}^{n} i = n \frac{n + 1}{2} \)

- \( \sum_{i=1}^{n} i^2 = \frac{1}{3} n(n + 1)(n + \frac{1}{2}) \)
Asymptotic notation: some useful properties

Order of polynomial functions

- \( c \in \Theta(1), \quad \forall c \in \mathbb{R}^+ \)

- \( \sum_{i=0}^{k} c_i n^i \in \Theta(n^k), \quad \forall c_k \in \mathbb{R}^+, \ \forall c_i \in \mathbb{R}, \ 1 \leq i < k \)

- \( \sum_{i=1}^{n} i^k \in \Theta(n^{k+1}), \quad \forall k \in \mathbb{N}^+ \)

- \( \sum_{i=1}^{n} (n - i)^k \in \Theta(n^{k+1}), \quad \forall k \in \mathbb{N}^+ \)
Asymptotic notation: some useful properties

Order of exponential, logarithmic etc. functions

- $n! \in \Omega(2^n)$, $n! \in O(n^n)$
- $\log(n!) \in \Theta(n \log n)$
- $\sum_{i=1}^{n} r^i \in \Theta(r^n)$, $\forall r \in R^{>1}$
- $\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n)$
- $\sum_{i=1}^{n} \frac{i}{r^i} \in \Theta(1)$, $\forall r \in R^{>1}$
Asymptotic notation: computational costs hierarchy

Some usual order relations:

\[ O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n^n) \]
Asymptotic notation: computational costs hierarchy

sublinear, linear and superlinear
Asymptotic notation: computational costs hierarchy

superlinear, polynomial and exponential
Computational costs hierarchy: practical consequences

Maximum size of a problem, $n$, that can be solved by several algorithms and computers with different processing performance:

<table>
<thead>
<tr>
<th>Temporal Cost</th>
<th>1 step = 1 ms</th>
<th>1 step = 0.1 ms (10 times faster)</th>
<th>$n' = f(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log_2 n$</td>
<td>$\approx 10^{330}$</td>
<td>$\approx 10^{3.10^3}$</td>
<td>$n^{10}$</td>
</tr>
<tr>
<td>$n$</td>
<td>1000</td>
<td>10003</td>
<td>10$n$</td>
</tr>
<tr>
<td>$n \log_2 n$</td>
<td>141</td>
<td>1003</td>
<td>$\approx 9n$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>32</td>
<td>100</td>
<td>3,16$n$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>10</td>
<td>13</td>
<td>$n + 3$</td>
</tr>
</tbody>
</table>

By increasing the calculus time and/or the computer performance, an algorithm ...

- **logarithmic** one increases **enormously** the size of input of tackled problems
- **linear** ones (or nlogn) achieves **linear** increments (or almost) of the tackled sizes
- **quadratic** (polynomial) ones achieve **proportional moderate improvements**
- **exponential** ones only achieve **negligible additive improvements**
Computational costs hierarchy: practical consequences

Execution times of a machine that executes $10^9$ steps by second (~ 1 GHz), as a function of the algorithm cost and the size of input $n$:

<table>
<thead>
<tr>
<th>Size</th>
<th>$\log_2 n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.322 ns</td>
<td>10 ns</td>
<td>33 ns</td>
<td>100 ns</td>
<td>1 $\mu$s</td>
<td>1 $\mu$s</td>
</tr>
<tr>
<td>20</td>
<td>4.322 ns</td>
<td>20 ns</td>
<td>86 ns</td>
<td>400 ns</td>
<td>8 $\mu$s</td>
<td>1 ms</td>
</tr>
<tr>
<td>30</td>
<td>4.907 ns</td>
<td>30 ns</td>
<td>147 ns</td>
<td>900 ns</td>
<td>27 $\mu$s</td>
<td>1 s</td>
</tr>
<tr>
<td>40</td>
<td>5.322 ns</td>
<td>40 ns</td>
<td>213 ns</td>
<td>2 $\mu$s</td>
<td>64 $\mu$s</td>
<td>18.3 min</td>
</tr>
<tr>
<td>50</td>
<td>5.644 ns</td>
<td>50 ns</td>
<td>282 ns</td>
<td>3 $\mu$s</td>
<td>125 $\mu$s</td>
<td>13 days</td>
</tr>
<tr>
<td>100</td>
<td>6.644 ns</td>
<td>100 ns</td>
<td>664 ns</td>
<td>10 $\mu$s</td>
<td>1 ms</td>
<td>40 $\cdot 10^{12}$ years</td>
</tr>
<tr>
<td>1000</td>
<td>10 ns</td>
<td>1 $\mu$s</td>
<td>10 $\mu$s</td>
<td>1 ms</td>
<td>1 s</td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>13 ns</td>
<td>10 $\mu$s</td>
<td>133 $\mu$s</td>
<td>100 ms</td>
<td>16.7 min</td>
<td></td>
</tr>
<tr>
<td>100000</td>
<td>17 ns</td>
<td>100 $\mu$s</td>
<td>2 ms</td>
<td>10 s</td>
<td>11.6 days</td>
<td></td>
</tr>
<tr>
<td>1000000</td>
<td>20 ns</td>
<td>1 ms</td>
<td>20 ms</td>
<td>16.7 min</td>
<td>31.7 years</td>
<td></td>
</tr>
</tbody>
</table>
Spatial Complexity

- **Memory address** concept: storage space where one or more data is located, which extension is independent of the size of input of the instances of the considered problem.

- **Spatial cost** of a program or algorithm: number of memory addresses required for its execution.

- All the concepts studied in temporal cost analysis are directly applicable to spatial cost.