# Analysis of Algorithms
# Part I: Analyzing a pseudo-code

**Introduction**

**Pseudo-code representation of an algorithm**

**Analyzing algorithms**

**Measuring the running time and memory size of an algorithm**

**Calculating the running time and memory size of an iterative algorithm**

**Order of growth of a function, asymptotic notations**

**Calculating the algorithmic complexity of an iterative algorithm: practical rules**

**Calculating the algorithmic complexity of a recursive algorithm: recursion tree method**

**References:**

*Introduction to ALGORITHMS*; Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein; MIT Press, 2001.

# Introduction

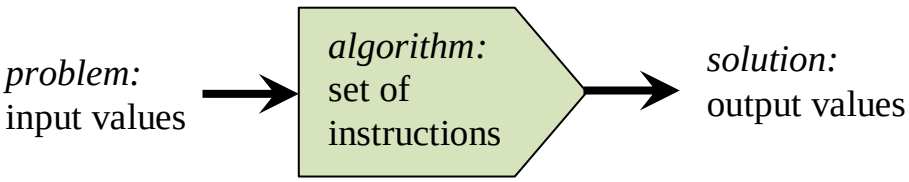
- *what is a computer program?*

a set of instructions written in a specific programming language
that is complete enough for a computer to execute it automatically

- *what is an algorithm?*

fundamental structure of a computer program

set of instructions written in an abstract form, pseudo-code or flowchart,
independently from any specific computer or programming language
→ explains clearly to a human brain the general procedure to solve a problem




- *what are the costs of running a computer program on a computer?*

→ some time and some memory...

→ cost analysis: we must have an idea of the *running time* and *memory size*
required to solve a problem of a certain size

maybe resolution becomes intractable when problem size increases...
(would require a PC to run for even longer than the age of the universe, 13 billion years)

*intractable*: possible in theory but impossible in practice...

no need to write a program if we realize that it will not be usable...

→ we should analyze the algorithm to know in advance the costs of running it

- *two ways of analyzing an algorithm:*

  precise calculation of the running time and memory size
  in function of the problem size

  *or*

  just have an idea of how fast the running time and memory size grow
  when the problem size increases

  → *order of growth* of the running time = *algorithmic complexity*

*example:*

"this algorithm runs in $n^2$ time"

→ its running time grows:
- with $n^2$
- with the square of n
- quadratically with n

→ its running time has the same order of growth as $n^2$

# Pseudo-code representation of an algorithm

- *to analyze easily an algorithm,* we need to represent it in a way that is:
  - clear and straightforward (the brain must easily comprehend its structure)
  - without any detail of implementation (unnecessary for analysis)
  - independent from any programming language (meant for the brain, not for a computer)

flowchart representation of an algorithm is mostly outdated and not suitable for analysis

- *pseudo-code conventions:*

→ based on the principles of *structured programming*
(sequential structure, conditional structure, iterative structure)

→ based on data abstraction:
- no variable declarations, no types
- we do not specify any memory allocation
- no pointers, no pointer arithmetic

no *input - output* operations

*assignment:* variable ← value

*if-then-else* construct

*while* construct

*block of code* represented with indentation and vertically stretched square bracket [

*1D array:* A[1..n] with 1..n range of the index → element: A[i]

*2D array:* A[1..n , 1..m] with 1..n and 1..m ranges of indices → element: A[i , j]

*function call:* similar to C → arguments are passed by value
except for arrays that are passed by reference

*example:*

*C code:*

```
#include <stdio.h>
#define N 100

 void main()
{
   int i , n = 0;
   double v , x[N] , max;

   while (scanf("%lf" , &v) != EOF)
   {  n++;    if (n > N)  exit(0);
      x[n - 1] = v;  }

   max = -1e20;
   for (i = 0 ; i < n ; i++)
      if (max < x[i])
         max = x[i];

   printf("\n max = %f" , max);
}
```

*pseudo-code:*

```
max ← - ∞
for i ← 1 to n
    if max < x[i]
        max ← x[i]
```

# Analyzing algorithms

- we must analyze an algorithm in order to predict the costs of executing it on a computer:
  - *running time* necessary to execute the code
  - *size of memory* necessary for the data structures of the code

why we should know the costs of an algorithm before coding it:

- better to know in advance if the algo can be executed with affordable costs:
  - running time of at most a few months...
  - memory size inferior to the RAM (or virtual memory) of our computer.
- find the *bottlenecks*, i.e. the parts of the program that will take the most time
- compare the efficiencies of two (or more) algorithms

we assume a generic model of computer:

only one CPU

instructions are executed one after the other

→ no parallelism, no concurrency

## quantifying problem size and costs:

- *quantifying the problem size:*
  - in general, **n** number of values that are input to the algorithm
  - maybe, **n** size of a square matrix → **n** = √(number of values in the matrix)
  - usually, a single integer **n**
  - sometimes 2 integers necessary to describe the problem size
    → size of a rectangular matrix = (number of rows , number of columns)
    → size of a graph = (number of vertices , number of edges)

- *quantifying the running time:*
  - real time in (nano)seconds
  - or total number of instructions of a given type (floating point multiplication, test, ...) carried out during the execution of the program
    = can be quasi proportional to physical running time

- *quantifying the memory size:*
  - number of bytes occupied by the data structure(s) of the program
  - number of values that must be stored

## running time as a function:

- for a given algorithm, the running time generally depends on the size of the problem:

  *"the bigger the problem, the more time it takes to solve it"*

  → running time **T(n)** = function of **n**

- **T(n)** may also depend on the input values themselves

even when **T** does not depend only on **n**, we will keep writing "**T(n)**"

best case - average case - worst case:

for a given **n**, **T(n)** varies between

its *best case* value (when the code runs in the shortest possible time)

and

its *worst case* value (when the code runs in the longest possible time)

most of the time, **T(n)** is around its *average case* value

- *best case - worst case analysis:*

  find in the code the test(s) over the data

  → choose for these test(s) the values that minimize - maximize **T(n)**

  → lower bound - upper bound for **T(n)**

- *worst case is the most significant case :*

  - we know things cannot be worse than that!
  - very often, the average case happens to be nearly as bad as the worst case

# Measuring the running time and memory size of an algorithm

*problem:* to measure **T(n)** and the memory size, we must first write the code and run it !

measure the running time:

- direct measure of physical time
  → use function **clock()** in C language

*example:*

```
#include <time.h>
. . . . . .
clock_t t0 = clock();
for (i = 1 ; i <= n ; i++)
   for (j = i + 1 ; j <= n ; j++)
      if (a[i][j] < 0)
         a[i][j] = a[i][j] * a[i][j];
printf("\n running time = %f" , (double)(clock() - t0) / CLOCKS_PER_SEC);
. . . . . .
```

- measure the number of times a certain instruction is executed
  → insert a counter variable inside the code

*example:*

```
. . . . . .
int counter = 0;
for (i = 1 ; i <= n ; i++)
   for (j = i + 1 ; j <= n ; j++)
      if (a[i][j] < 0)
      {  a[i][j] = a[i][j] * a[i][j];
         counter++;  }
printf("\n running time = %d" , counter);
. . . . . .
```

measure the memory size:

- read from the OS the memory being used (Task Manager of Windows...)
- increment a counter each time a new block of memory is allocated during execution

# Calculating the running time and memory size of an iterative algorithm

*advantage:* can be done simply by analyzing the algorithm, long before the code has been written and run

## memory size:

analyze the data structures used by the pseudo-code

→ what is their size?

## running time:

an iterative algorithm is made of loops (eventually nested) and of tests over the data

→ we must count the total number of iterations and express it as an algebraic expression

## loops:

- *header line of loop executed one more time than the number of iterations:*

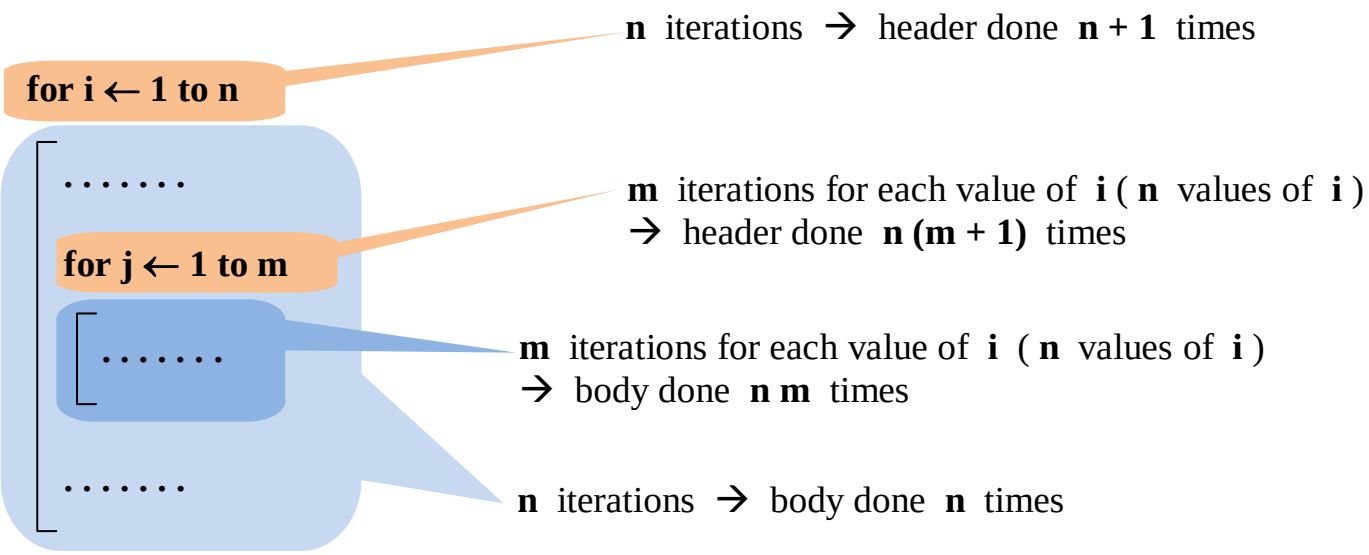
  assignment and test on index executed one more time → exit from the loop

*example:*

```
for (i = 0 ; i < 4 ; i++)
    printf("\n i = %d" , i);
```

| i ← | i < 4 | printing |
|---|---|---|
| 0 | true | i = 0 |
| 1 | true | i = 1 |
| 2 | true | i = 2 |
| 3 | true | i = 3 |
| 4 | false | |

- *nested loops:*




- *example: matrix addition ( A (n × m), B (n × m)*

```
   add_mat(A , B , n , m)
1----for i ← 1 to n
2-------for j ← 1 to m
3----------C[i , j] ← A[i , j] + B[i , j]
   return C
```

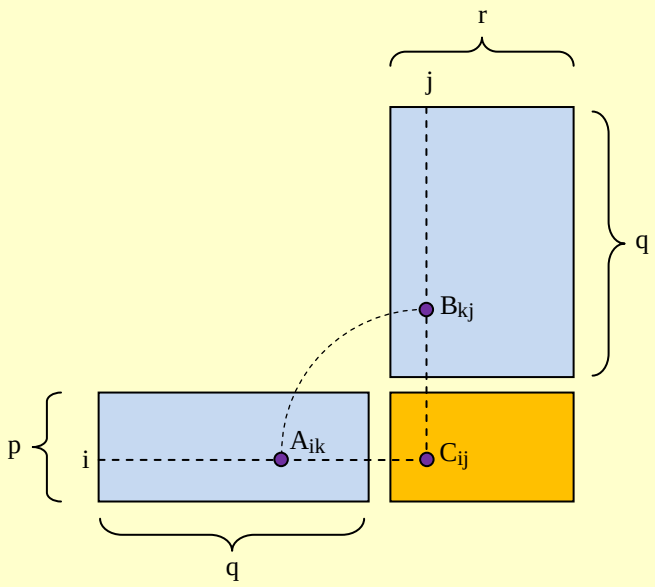
a) calculate **T(n , m)** as real time assuming that line **l** takes $c_l$ nanoseconds to execute once

b) calculate **T(n , m)** as number of floating point additions

. . . . . *resolution on the board*

*example: matrix multiplication ( A (p × q), B (q × r) )*

```
   mul_mat(A , B , p , q , r)
1  for i ← 1 to p
2     for j ← 1 to r
3        C[i , j] ← 0
4        for k ← 1 to q
5           C[i , j] ← C[i , j] + A[i , k] * B[k , j]
   return C
```




a) calculate **T(p,q,r)** as real time assuming that line **l** takes $c_l$ nanoseconds to execute once

b) calculate **T(p,q,r)** as number of floating point multiplications

. . . . . *resolution on the board*

the number of iterations of the inner loop may depend on the index of the outer loop!

→ sum the numbers of iterations of the inner loop for all the values of the outer loop index

→ some formulas to calculate this sum:

$$1 + 2 + 3 + \ldots + n = n (n + 1) / 2$$

→ possible adjustments: $2 + \ldots + n = n (n + 1) / 2 - 1$

$$1 + 2 + \ldots + (n - 1) = n (n + 1) / 2 - n$$

$$1^2 + 2^2 + 3^2 + \ldots + n^2 = n (n+1) (2n+1) / 6$$

$$1^3 + 2^3 + 3^3 + \ldots + n^3 = n^2 (n + 1)^2 / 4$$

*examples:*

```
1-- for i ← 2 to n
2-----for j ← 1 to i
3---------C[i , j] ← i * j
```

```
1-- for i ← 1 to n - 1
2-----for j ← i to n
3---------C[i , j] ← i * j
```

```
1-- for i ← 2 to n + 1
2-----for j ← i to n
3---------for k ← i to n
4-----------C[i , j] ← C[i , j] + i * j * k
```

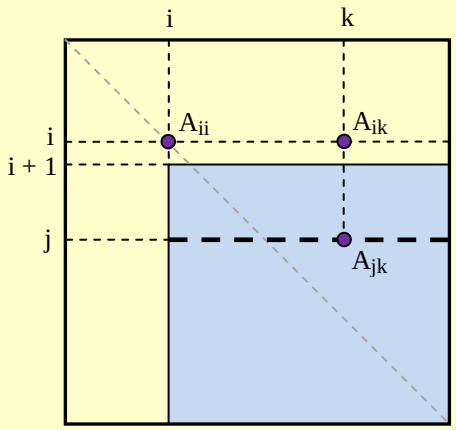
for each of these codes, calculate **T(n)** as number of assignments

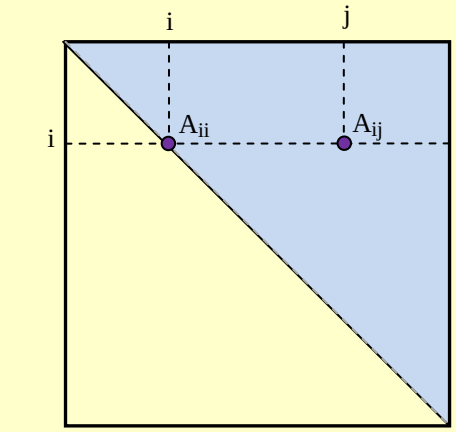
. . . . . *resolution on the board*

*example: Gauss linear system resolution ( A V = B with A (n× n) , V (n) , B (n) )*

```
solve_linear_system(A , B , n)
1   for i ← 1 to n
2       for j ← i + 1 to n
3           r ← A[j , i] / A[i , i]
4           B[j] ← B[j] - r × B[i]
5           for k ← i to n :
6               A[j , k] ← A[j , k] - r × A[i , k]

7   V[n] ← B[n] / A[n , n]
8   for i ← n - 1 down to 1
9       s ← 0
10      for j ← i + 1 to n
11          s ← s + A[i , j] × V[j]
12      V[i] ← (B[i] - s) / A[i , i]

    return V
```

elimination
= triangulation
→ $T_1(n)$




resolution of
triangular system
→ $T_2(n)$

calculate $T_1(n)$ and $T_2(n)$ as numbers of floating point multiplications

. . . . . *resolution on the board*

test(s) on data:

→ *best case - worst case analysis*

*example:* *search problem: find value* **v** *inside* **A[1..n]** → *linear search algorithm*

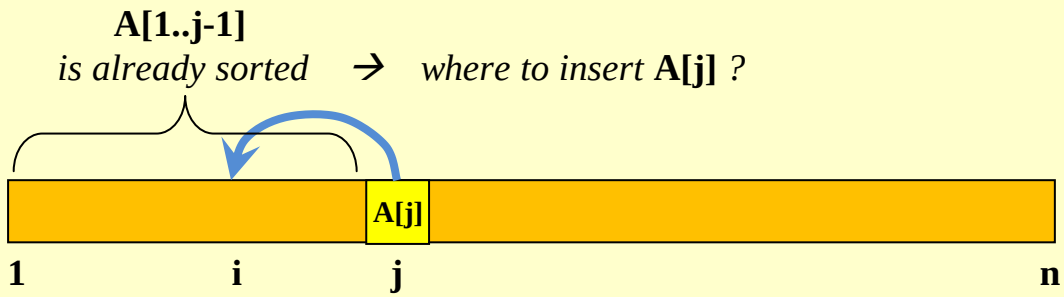
```
   linear_search(A , v , n)
1  i ← 1
2  while i ≤ n and A[i] ≠ v
3      i ← i + 1

4  if i = n + 1
5      return NIL
   else
6      return i
```

a) for best and worst cases, calculate **T(n)** as real time
assuming that line **l** takes $c_l$ nanoseconds to execute once

b) for best and worst cases, calculate **T(n)** as number of tests

. . . . . *resolution on the board*

*example:* *sorting problem: sort in ascending order the elements of A[1..n]*
→ *insertion sort algorithm*




```
   insertion_sort(A , n)
1  for j ← 2 to n
2      v ← A[j]
3      i ← j - 1
4      while i > 0 and A[i] > v
5          A[i + 1] ← A[i]
6          i ← i - 1
7      A[i + 1] ← v
```

a) for best and worst cases, calculate **T(n)** as real time
assuming that line **l** takes $\mathbf{c_l}$ nanoseconds to execute once

b) for best and worst cases, calculate **T(n)** as number of tests

. . . . . *resolution on the board*

# Calculating the running time of a combinatorial algorithm

*combinatorial algorithm* = algorithm that must explore all possible combinations over a given data set in order to find the best combination

the number of such combinations may become huge as the the data set size increases
→ *combinatorial explosion*

**T(n)** = total number of possible combinations
→ simply count this number

*example: traveling salesman problem*

set of **n** fully connected cities (one road between each couple of cities)
→ find the shortest path to visit all the cities (by visiting a given city only once)

```
for each possible city among the n cities
  for each possible next city among the remaining n - 1 cities
    for each possible next city among the remaining n - 2 cities
      for each possible next city among the remaining n - 3 cities
        .......
          .......
            .......
              for each possible next city among the remaining 2 cities
                for each possible next city among the remaining 1 city
                  examine the length of the path
```

**T(n)** = number of possible paths = **n × (n - 1) × (n - 2) × (n - 3) × . . . . . × 2 × 1**

⇒ **T(n) = n!**

| **n** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **T(n)** | 1 | 2 | 6 | 24 | 120 | 720 | 5040 | 40320 | 362880 | 3628800 | 39916800 | 479001600 |

# Order of growth of a function, asymptotic notations

some basic mathematical recalls:

- *logarithms:*

  log base 2: $\lg n$

  $\lg 1 = 0$ $\lg 2 = 1$

  $\lg(a\,b) = \lg a + \lg b$

  relation between logarithms of different bases:

  $\log_b a = \log_c a / \log_c b$

- *exponentials:*

  exponential base 2: $2^n$

  exponential base e: $e^n$

  $2^0 = 1$ $2^1 = 2$

  $2^{a+b} = 2^a \times 2^b$

  $(2^a)^b = (2^b)^a = 2^{a\,b}$

  $a^{\log_b n} = n^{\log_b a}$

  $a = 2^{\lg a}$

a most intuitive introduction to the order of growth:

- *order of growth of a polynomial term:*

**f(n) = a $n^p$** → what happens to **f(n)** when we multiply **n** by 10 ?

**f(10 n) / f(n) = a $(10\ n)^p$ / a $n^p$ = $10^p$**

| **polynomial term** | **when we multiply n by 10, f(n) is multiplied by:** |
|---|---|
| **a n** | 10 |
| **a $n^2$** | $10^2$ = 100 |
| **a $n^3$** | $10^3$ = 1000 |
| **a $n^4$** | $10^4$ = 10000 |

→ **a n** grows in proportion of **n**

the higher the exponent, the faster the growth

the constant multiplying factor **a** of a polynomial term has absolutly no influence on the order of growth

- *order of growth of a logarithmic term:*

**f(n) = lg n** → what happens to **f(n)** when we multiply **n** by 10 ?

**lg(10 n) = lg n + lg 10 = lg n + 3.322** → **f(n)** is just increased by 3.322

= *very slow growth*

- *order of growth of an exponential term:*

**f(n) = $2^n$** → what happens to **f(n)** when we multiply **n** by 10 ?

**f(10 n) = $2^{10\ n}$ = $(2^n)^{10}$ = $(f(n))^{10}$** → **f(n)** is raised to the power 10 = *huge!*

what happens to **f(n)** when we add 10 to **n** ?

**f(n + 10) = $2^{n\ +\ 10}$ = $2^n$ . $2^{10}$ = f(n) . 1024** → **f(n)** is multiplied by 1024

= *extremely fast growth*

asymptotic notations:

there exists no mathematical tool to express directly the order of growth of a function

→ use the *asymptotic notations* to compare the orders of growth of two functions:

| **asymptotic notation** | **intuitive comparison between f(n) and g(n)** | **orders of growth of f(n) and g(n)** |
|---|---|---|
| **f(n) = Θ(g(n))** | **f(n)** grows similarly to **g(n)** | order of growth of **f(n)** = order of growth of **g(n)** |
| **f(n) = O(g(n))** | **f(n)** grows slower or similarly to **g(n)** | order of growth of **f(n)** ≤ order of growth of **g(n)** |
| **f(n) = Ω(g(n))** | **f(n)** grows faster or similarly to **g(n)** | order of growth of **f(n)** ≥ order of growth of **g(n)** |
| **f(n) = o(g(n))** | **f(n)** grows strictly slower than **g(n)** | order of growth of **f(n)** < order of growth of **g(n)** |
| **f(n) = ω(g(n))** | **f(n)** grows strictly faster than **g(n)** | order of growth of **f(n)** > order of growth of **g(n)** |

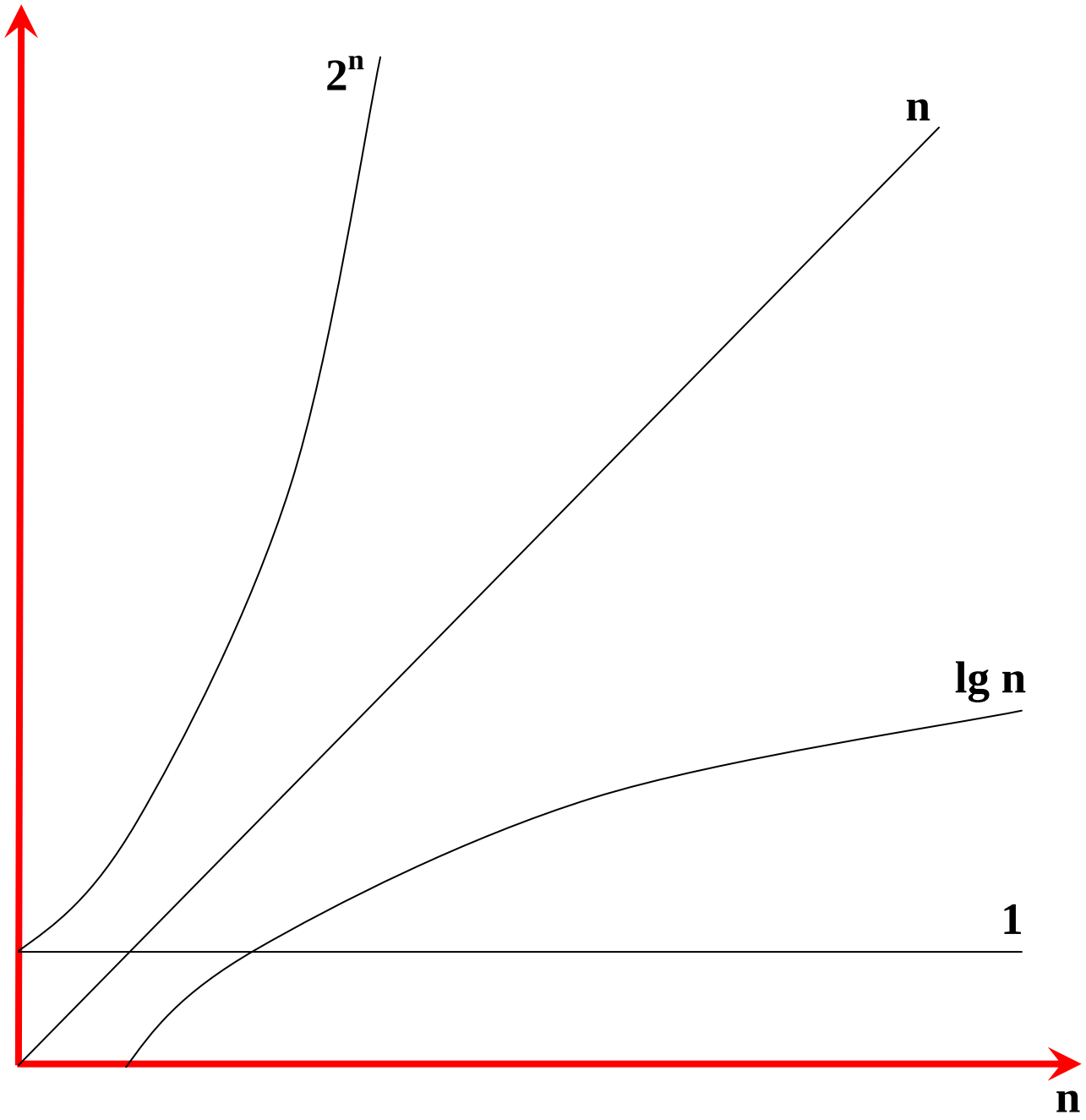
fundamental properties of the asymptotic notations:

| | |
|---|---|
| | $f(n) = \Theta(g(n)) \Rightarrow f(n) = \mathbf{O}(g(n))$ |
| | $f(n) = \Theta(g(n)) \Rightarrow f(n) = \mathbf{\Omega}(g(n))$ |
| | $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \mathbf{O}(g(n))$ *and* $f(n) = \mathbf{\Omega}(g(n))$ |
| *transitivity* for **Θ , O , Ω , o , ω** | $f(n) = \Theta(g(n))$ *and* $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$ |
| *reflexivity* for **Θ , O , Ω** | $f(n) = \Theta(f(n))$ |
| *symmetry* for **Θ** | $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$ |

functions of reference:

Usually, we describe the order of growth of a function by comparing it with the order of growth of a well known function taken as reference

| **function of reference** | **intuitive description** |
|---|---|
| **1** | no growth |
| **lg n** | logarithmic growth, "*very slow*" growth |
| **n** | linear growth, grows linearly with n, "*normal*" growth |
| **n lg n** | "a bit more" than linear growth |
| **$n^2$** | quadratic growth, grows with the square of n, "*quite fast*" growth |
| **$n^3$** | cubic growth, grows with the cube of n, "*fast*"growth |
| **$2^n$** | exponential growth, "*exceedingly fast*" growth |
| **n!** | factorial growth, "*astonishingly fast*" growth |
| others . . . | . . . . . |


2^n
n
lg n
1
n

practical meaning for the running time:

| **algorithmic complexity** | **"quality" of the algorithm** |
|---|---|
| **$T(n) = \Theta(1)$** | same **T(n)** whatever the problem size<br>*too good to be true, "don't even think about it"* |
| **$T(n) = \Theta(\lg n)$** | **T(n)** remains quite small, even for huge problem size<br>*excellent, "dream case"* |
| **$T(n) = \Theta(n)$** | **T(n)** grows in proportion with the problem size<br>*very good*, the minimum we should expect whenever we directly manipulate each of the n input values |
| **$T(n) = \Theta(n \lg n)$** | *still very good*, barely more than **$\Theta(n)$** |
| **$T(n) = \Theta(n^2)$** | *costly* but affordable<br>the case of many algos (matrix addition) |
| **$T(n) = \Theta(n^3)$** | *very costly* but still affordable if **n** is not huge<br>still the case of many algos (matrix multiplication) |
| **$T(n) = \Theta(2^n)$** | *exceedingly costly*<br>affordable only for small problem size<br>intractable as soon as problem size increases<br>*algo of very limited utility, useless most of the time* |
| **$T(n) = \Theta(n!)$** | incredibly costly<br>totally intractable unless problem size is extremely small<br>*useless algo practically always* |

O notation is most often used:

*worst-case* **T(n) = Θ(f(n))** ⇒ *general* **T(n) = O(f(n))**

→ **O** notation provides an upper bound on **T(n)** whatever the case

asymptotic notations **Θ , O , o** in algebraic equations:

assume that $\mathcal{N}$ is either one of the three asymptotic notations **Θ , O , o**

**f(n) = g(n) + $\mathcal{N}$(h(n))** *means that* **f(n) = g(n) + e(n)** *with* **e(n) = $\mathcal{N}$(h(n))**

**e(n)** is often an *error term*

→ we are not interested in knowing **e(n)** exactly,
we just want to know that it will not grow too fast...

*examples:*

**T(n) = 2 $n^2$ + Θ(n)** → error term grows linearly with **n**,

→ error term grows strictly slower than **2 $n^2$**

**T(n) = 2 $n^2$ + O(n)** → error term grows at most linearly with **n**,

→ error term grows strictly slower than **2 $n^2$**

**T(n) = 2 $n^2$ + o($n^2$)** → error term grows strictly less than quadratically with **n**,

→ error term grows strictly slower than **2 $n^2$**

# Calculating the algorithmic complexity of an iterative algorithm: practical rules

calculate the order of growth of a precise expression of T(n):

- *method:*

  we have already calculated: **$T(n) = f(n)$**

  $\Rightarrow$ **$T(n) = \Theta(f(n))$** (or, similarly, **$T(n) = O(f(n))$** )

  → we gradually simplify the expression inside $\Theta$ notation (or inside **O** notation) until a simple *function of reference* is reached

- *practical rules to simplify the expression inside $\Theta$ notation :*

**$T(n) = \Theta(a\, n^p) \Rightarrow T(n) = \Theta(n^p)$** ($a > 0$)

**$T(n) = \Theta(a\, n^p + b\, n^q)$** *with* **$p > q \Rightarrow T(n) = \Theta(n^p)$** ($a > 0$)

*note:* exponents may be fractional or negative:

$$a\,n + b\,\sqrt{n} + c + d/n + e/n^2 = a\,n^1 + b\,n^{0.5} + c\,n^0 + d\,n^{-1} + e\,n^{-2}$$

in general: assume a *polynomial* **P(n)** → *higher-order term* of **P(n)**
= term with biggest exponent for **n**

**$T(n) = \Theta(P(n)) \Rightarrow T(n) = \Theta($** ***higher-order term of*** **$P(n))$**

**$T(n) = \Theta((P(n))^p) \Rightarrow T(n) = \Theta(($** ***higher-order term of*** **$P(n))^p)$**

$T(n) = \Theta( P_1(n) / P_2(n) ) \Rightarrow T(n) = \Theta($ ***higher-order term of*** $P_1(n) /$ ***higher-order term of*** $P_2(n) )$

→ in particular:

$T(n) = \Theta( (a\, n^p + b\, n^q) / (c\, n^r + d\, n^s) )$ *with* $p > q$ , $r > s \Rightarrow T(n) = \Theta(n^p / n^r) = \Theta(n^{p-r})$

an *exponential term* grows much faster than any *polynomial term*:

$T(n) = \Theta(a \times 2^n + b\, n^p)$ *with* $p > 0 \Rightarrow T(n) = \Theta(2^n)$ ($a , b > 0$)

a *polynomial term* grows much faster than any *logarithmic term*:

$T(n) = \Theta(a\, n^p + b\, \lg^q n)$ *with* $p , q > 0 \Rightarrow T(n) = \Theta(n^p)$ ($a , b > 0$)

*composition* of functions:

$T(n) = f( g(n) )$ *with* $g(n) = \Theta( h(n) ) \Rightarrow T(n) = \Theta( f( h(n) ) )$

*examples:*

$T(n) = 2n^2 - 5n \Rightarrow T(n) = \Theta(2n^2 - 5n) \Rightarrow T(n) = \Theta(2n^2) \Rightarrow T(n) = \Theta(n^2)$
$\Rightarrow T(n) = O(n^3)$

$T(n) = 2n^2 + n + 1 \Rightarrow T(n) = \Theta(2n^2 + n + 1) \Rightarrow T(n) = \Theta(2n^2) \Rightarrow T(n) = \Theta(n^2)$

$T(n) = k + 1/n$ *with* $k > 0 \Rightarrow T(n) = \Theta(k\, n^0 + n^{-1}) \Rightarrow T(n) = \Theta(k)$
$\Rightarrow T(n) = \Theta(1)$

$T(n) = (2n^2 - n + 1)^3 \Rightarrow T(n) = \Theta( (2n^2 - n + 1)^3 ) \Rightarrow T(n) = \Theta( (2n^2)^3 )$
$\Rightarrow T(n) = \Theta(8\, n^6) \Rightarrow T(n) = \Theta(n^6)$

$T(n) = n + 1 / (2n^2 - n + 1)^3 \Rightarrow T(n) = \Theta( n + (2n^2 - n + 1)^{-3} )$
$\Rightarrow T(n) = \Theta( n + (2n^2)^{-3} )$
$\Rightarrow T(n) = \Theta( n + (1/8) \times n^{-6} ) \Rightarrow T(n) = \Theta(n)$

$T(n) = (n + 1) / (n + 2) \Rightarrow T(n) = \Theta( (n + 1) / (n + 2) ) \Rightarrow T(n) = \Theta( n / n )$
$\Rightarrow T(n) = \Theta(1)$

$T(n) = (n + 1)^3 / (n + 2) \Rightarrow T(n) = \Theta( (n + 1)^3 / (n + 2) ) \Rightarrow T(n) = \Theta( n^3 / n )$
$\Rightarrow T(n) = \Theta(n^2)$

$T(n) = \lg n \Rightarrow T(n) = O(n)$ *but* $T(n) \neq \Theta(n)$

$T(n) = 0.01 \times \sqrt{n} + \lg^{10} n \Rightarrow T(n) = \Theta(0.01 \times n^{0.5} + \lg^{10} n) \Rightarrow T(n) = \Theta(0.01 \times n^{0.5})$
$\Rightarrow T(n) = \Theta(n^{0.5})$

*examples:*

$T(n) = 2\,n^2 + 1/n$

$T(n) = 2\,n + (0.00001\,n^3 - n)/(1000\,n + 1/n)$

$T(n) = (\lg n + 1) \times (n^2 + 1) / (2 + n)$

$T(n) = 4^{\lg n} + n^3$

$T(n) = \lg(10^{-20} \times 2^n + n^{10} - 10^{20})$

$T(n) = ( (1 + 2n^3) / (n + \lg n) )^3$

. . . . . *resolution on the board*

estimate in a glance the order of growth directly from the pseudo-code:

- *if a loop iterates a variable number of times:*

  → estimate the *average* of its number of iterations

  → number of iterations = **Θ( *average* )**

- *nested loops:*

  assume: { number of iterations of *outer loop* = **Θ( f(n) )**
  number of iterations of *inner loop* = **Θ( g(n) )**

  → total number of times the body of the inner loop is executed = **Θ( f(n) × g(n) )**

- *program with conditional blocks of code:*

  *best case:* *ignore* all conditional blocks of code

  *worst case:* *consider* all conditional blocks of code

*example:*

*general case:*

```
for ... Θ(n)
    if ...
        while ... Θ(n^2)
    while ... Θ(n)
        if ...
            for ... Θ(n)
```

*best case:* → **Θ(n²)**

```
for ... Θ(n)
    while ... Θ(n)
```

*worst case:* → **Θ(n³)**

```
for ... Θ(n)
        while ... Θ(n^2)
    while ... Θ(n)
            for ... Θ(n)
```

*example: insertion sort revisited*

```
insertion_sort(A , n)
    for j ← 2 to n
        v ← A[j]
        i ← j - 1

        while i > 0 and A[i] > v
            A[i + 1] ← A[i]
            i ← i - 1

        A[i + 1] ← v
```

. . . . . *resolution on the board*

# Calculating the algorithmic complexity of a recursive algorithm: recursion tree method

mathematical recalls:

- *floor and ceiling functions:*

  assume **x** real

  *floor:* $\lfloor x \rfloor$ = greatest integer inferior or equal to **x**

  *ceiling:* $\lceil x \rceil$ = smallest integer superior or equal to **x**

*examples:*

| x | $\lfloor x \rfloor$ | $\lceil x \rceil$ |
|---|---|---|
| 1.0 | 1 | 1 |
| 1.1 | 1 | 2 |
| 1.9 | 1 | 2 |
| 2.0 | 2 | 2 |
| 2.1 | 2 | 3 |
| 2.9 | 2 | 3 |
| 3.0 | 3 | 3 |

- *geometric series:*

$$q^0 + q^1 + q^2 + q^3 + \ldots q^n = (1 - q^{n+1}) / (1 - q)$$

*if* $|q| < 1$ : $q^0 + q^1 + q^2 + q^3 + \ldots = 1 / (1 - q)$ (*infinite sum*)

structure of a recursive algorithm:

```
definition of recursive_function
    if test
        terminal case
    else
        . . . . . .
        several recursive calls to recursive_function
        . . . . . .
```

*example: factorial*

```
fact(n)
    if n = 0
        return 1
    else
        return n × fact(n - 1)
```

*example: recursive sum of the values in array* **A[1..n]**

*first call:* **sum(A , 1 , n)**

```
sum(A , p , r)
    if p = r
        return A[p]
    else
        q ← ⌊ (p + r) / 2 ⌋
        s0 ← sum(A , p , q)
        s1 ← sum(A , q + 1 , r)
        return s0 + s1
```

## recurrence formula for T(n):

recursive resolution of a problem of a given size:

- subdivide the problem into subproblems of smaller size
- solve each subproblem recursively
- merge the subproblems' solutions into the problem solution

***solve problem of size*** **n**

- ***if*** **$n = n_0$**
  - ***terminal case***
- ***else***
  - ***subdivide problem into subproblems of sizes*** **$n_1 , n_2 , \ldots , n_q$**
  - ***solve subproblem of size*** **$n_1$**
  - . . . . . .
  - ***solve subproblem of size*** **$n_q$**
  - ***merge solutions of subproblems***

→ total time needed to solve a problem of size **n**:

$$T(n) = \begin{cases} T_{\text{terminal case}} & \text{if } n = n_0 \\ T_{\text{subdivide}} + T(n_1) + T(n_2) + \ldots + T(n_q) + T_{\text{merge}} & \text{if } n > n_0 \end{cases}$$

*example: recursive sum (previous slide)*

```
sum(A , p , r)
  if p = r                                   Θ(1)
    return A[p]
  else                                       Θ(1)
    q ← ⌊ (p + r) / 2 ⌋
    s0 ← sum(A , p , q)                      T(n / 2)
    s1 ← sum(A , q + 1 , r)
    return s0 + s1                           T(n / 2)
                                             Θ(1)
```

$$\Rightarrow T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2\,T(n/2) + \Theta(1) & \text{if } n > 1 \end{cases}$$

recursion tree:

*arborescence* representing all the calls to the recursive function

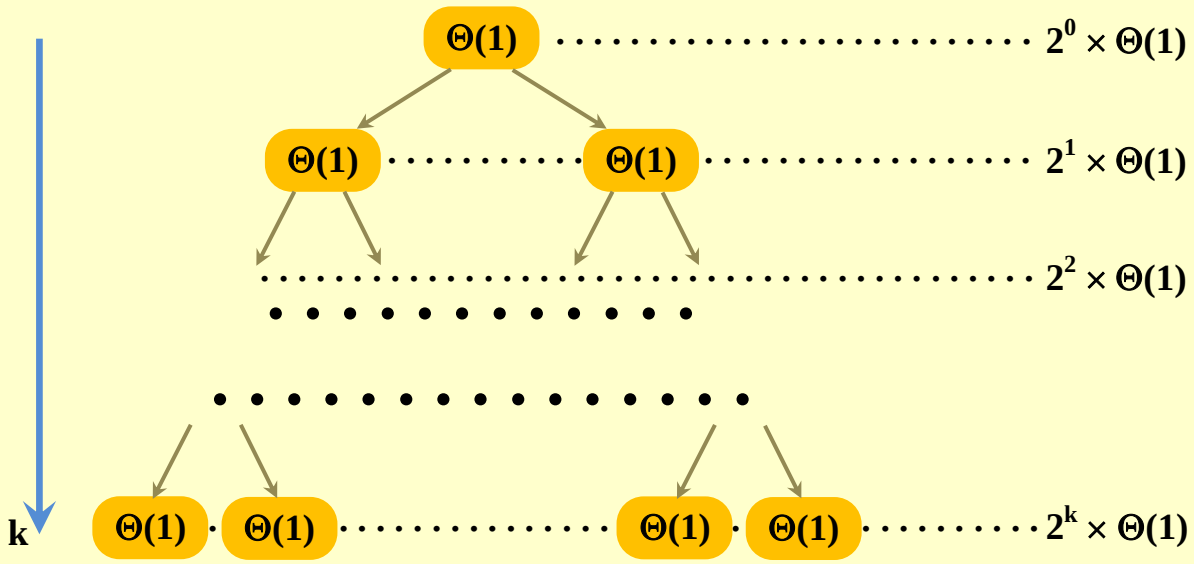
each *call* is represented by a *node* annotated with the time spent locally within that call

recursion tree method:

- *step1:* draw the *recursion tree*, find its *number of levels*
- *step2:* *sum* all the *local times* of the nodes, first over each level, then over all levels

*example:*

$T(n) = 2\,T(n / 2) + \Theta(1)$




$n / 2^k \approx 1$

$\Rightarrow k \approx \lg n$

$\Rightarrow T(n) = (2^0 + 2^1 + 2^2 + \ldots + 2^k) \times \Theta(1)$

$= ((1 - 2^{k+1}) / (1 - 2)) \times \Theta(1) = ((1 - 2 \times 2^{\lg n}) / -1) \times \Theta(1)$

$\Rightarrow T(n) = (2n - 1) \times \Theta(1) = \Theta(n)$

*examples:*

$T(n) = 2\,T(n / 2) + n^2$

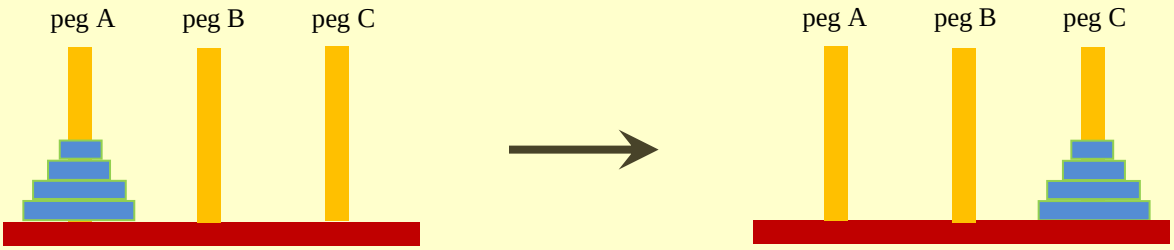
$T(n) = T(n / 3) + T(2\,n / 3) + n$

. . . . . *resolution on the board*

*example: towers of Hanoi puzzle*

transfer all the **n** disks from peg **A** to peg **C** according to the following rules:

- move only one disk at a time, the top disk, from one peg to another peg
- never put a larger disk over a smaller disk




recursive strategy:

*step 1:* transfer the top **n - 1** disks of peg **A** to peg **B**, using peg **C** as a buffer
*step 2:* move the remaining disk from peg **A** to peg **C**
*step 3:* transfer the top **n - 1** disks of peg **B** to peg **C**, using peg **A** as a buffer

n. of disks to transfer | source peg | buffer peg | destination peg

```
transfer_disks(n , A , B , C)
    if n = 1
        move the top disk of peg A to peg C
    else
        transfer_disks(n - 1 , A , C , B)
        move the top disk of peg A to peg C
        transfer_disks(n - 1 , B , A , C)
```

a) assuming **T(n)** is the number of disk moves, establish the recurrence formula for **T(n)**

b) apply the recursion tree method to obtain **T(n)**

. . . . . *resolution on the board*