Graphs and Complexity

2) Complexity and its particularities on graph problems

Irena.Rusu@univ-nantes.fr

LS2N, bât. 34, bureau 303

tél. 02.51.12.58.16

Plan

- Basics (reminders)
- Compute the running time of an algorithm:
  - Easy cases
  - More difficult cases
- Graphs are not easy cases
- P, NP, …

Basics (reminders)

- Qualities of an algorithm/corresponding program
  - Correction : no universal approach to show it
  - Performance : universal unit mesure
    - Memory requirements
    - Time requirements (usually the most acute problem)

- Time requirements not measured in seconds/min etc.

Why?

Why?

```python
def duplicates1(L):
    n = len(L)
    for i in range(n):
        for j in range(n):
            if i != j and L[i] == L[j]:
                return True
    return False
```

```python
import time

for i in range(5):

    n = 1000
    start = time.time()
    duplicates1(list(range(n)))
    timetaken = time.time() - start
    print("Time taken for n = ", n, ": ", timetaken)
```

```
Time taken for n =  1000 :  0.08032798767089844
Time taken for n =  1000 :  0.07732391357421875
Time taken for n =  1000 :  0.07471418380737305
Time taken for n =  1000 :  0.07387709617614746
Time taken for n =  1000 :  0.07915425300598145
```

Different running times for the same instance !

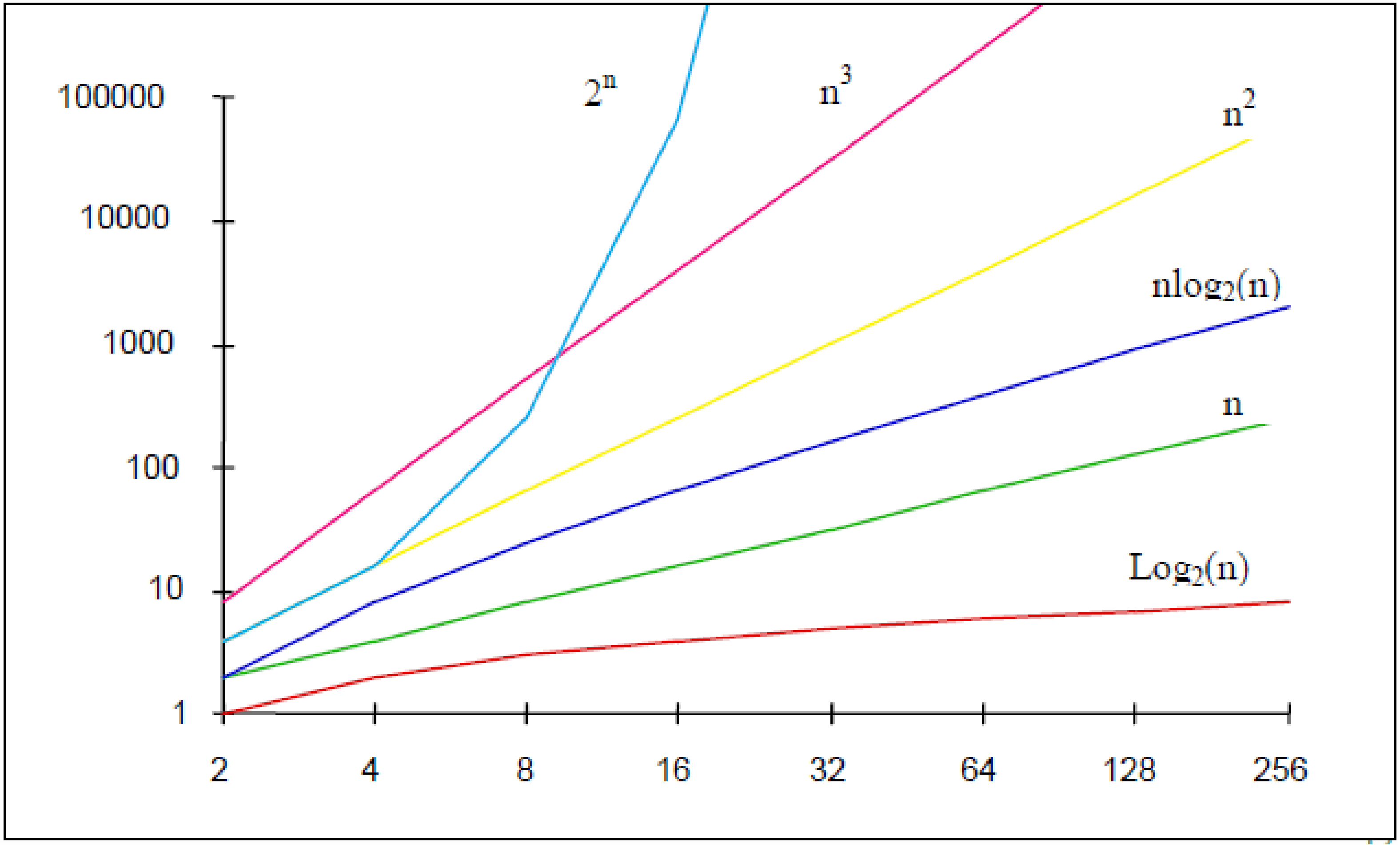
Basics (reminders)

- Complexity of an algorithm, depending on the size n of the input:
  - **Running time:** an estimation based on the number of **unit** operations performed by the algorithm

    Examples: assignment, simple mathematical operations (+, -, /, *), push, pop, comparison of two values, move a pointer to the next element (list)
  - **Memory size:** an estimation of the number of memory cells used by the algorithm

    Example: total size of the data structures used by the algorithm

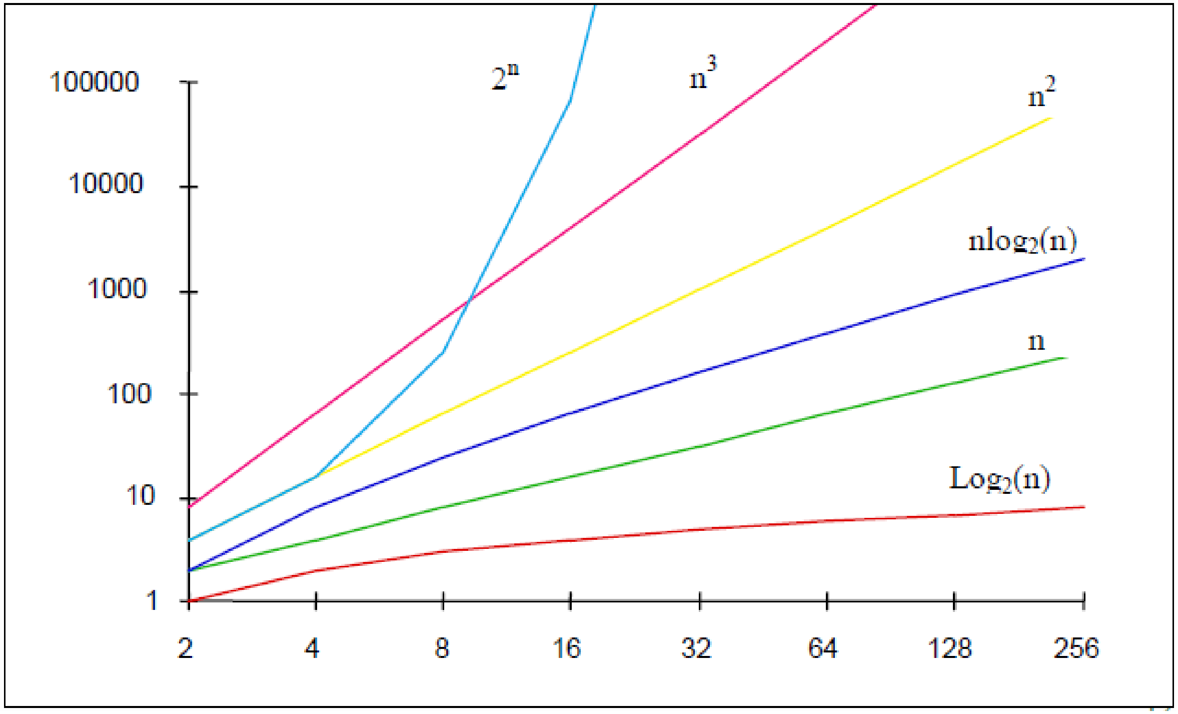
    Note: recursive algorithms use hidden data structures (stacks) whose size must be taken into account (usually the computation of the running time helps)
- We focus on the worst case of the running time, asymptotically (i.e. for large values of n)



Usually we write log n instead of $\log_2 n$

The relative position of the curves representing the functions is not modified for large values of *n*, even when additive or multiplicative constants are used.

Notations O, Θ



- $c(n)=O(g(n))$

means that $c(n)$ is upper bounded by (or below) $g(n)$ for large values of n.

- $c(n)=\Theta(g(n))$

means that $c(n)$ is the same function as $g(n)$, up to the multiplicative or additive constants

Running time of the algorithm : $c(n)$ (initially unknown)

Computing the running time requires to evaluate $c(n)$ as precisely as possible, using Θ if possible, or O with a $g(n)$ as low as possible.

$O(\ )$ and $\Theta(\ )$ simplify the computation

- $O(\ )$ and $\Theta(\ )$ may seem difficult to understand
- But they allow us
  - To forget the multiplicative and additive constants
  - And thus to focus only on the significative unit operations (those that appear in the most « costly » loops, in terms of number of operations).

Example. In a sorting algorithm, we may decide that

$c(n) =$ the number of comparisons performed by the algorithm

(unless the algorithm performs many other useless operations, meaning it is a bad algorithm).

Plan

- Basics
- Compute the running time of an algorithm:
  - Easy cases
  - More difficult cases
- Graphs are not easy cases
- P, NP, …

Compute the running time

- No magic solution, working for all algorithms
- An attempt (if a direct computation fails):
  1. Hope that your algorithm for size *n* builts upon the same algorithm (yours) but for a (unique) smaller size *n-1*, or *n-2*, …, or *n/2, …*
  2. Deduce a relationship between *c(n)* and *c(n-1)* (or *c(n-2)* …). May be *c(n)*=… or *c(n)* ≤…
  3. Solve this recurrence by replacing *c(n-1)* with its own formula and so on, up to *c(0)* or *c(1)* etc., which are known (and not 0).

Example

S=1+2+3+4+5

for i=2 to n do

*S=S*i*

endfor

$c(n)=c(n-1)+3$,

$c(n)=c(n-1)+3$

$= (c(n-2)+3)+3$

$=((c(n-3)+3)+3)+3$

$=\ldots=c(0)+3n$

With $c(0)=5$,

$\boldsymbol{c(n)=3n+5=\Theta(n)}$.

Easy cases

$c(n) = c(n-1)$	$\rightarrow c(n) = \Theta(1)$	constant running time
$c(n) = c(n-1)+1$	$\rightarrow c(n) = \Theta(n)$	linear running time
$c(n) = c(n-1)+n$	$\rightarrow c(n) = \Theta(n^2)$	quadratic running time
(…)	$\rightarrow$	other polynomial running times
$c(n) = 2c(n-1)$	$\rightarrow c(n) = \Theta(2^n)$	exponential running time
$c(n) = 2c(n-1) + n$	$\rightarrow c(n) = \Theta(2^n)$	exponential running time
$c(n) = nc(n-1)$	$\rightarrow c(n) = \Theta(n!)$	factorial running time
(…)	$\rightarrow$	other exponential running times

If $=$ is replaced with $\leq$ (left), then Θ is replaced with O (right)

More difficult cases

- *n/2, 3n/4* etc instead of *n-1*

$$c(n)=c(n/2)+3$$
$$= (c(n/2^2)+3)+3$$
$$= ((c(n/2^3)+3)+3)+3$$
$$\dots$$
$$= c(n/2^k)+3+\dots+3$$
(*k* fois)

where $n/2^k=1$, i.e. $k=\log n$.

With $c(1)=4$ (for instance):

$$c(n)=c(1)+3\log n=3\log n+4$$
$$c(n)=\Theta(\log n).$$

- ... and a multiplicative constant

$$c(n)= 5c(n/2)+7=$$
$$= 5(5c(n/2^2)+7)+7$$
$$= 5(5(5c(n/2^3)+7)+7)+7$$
$$\dots$$
$$= 5^k c(n/2^k)+7(1+5+\dots+5^{k-1})$$
$$= 5^k c(n/2^k)+ 7 * \frac{5^k-1}{5-1}$$

where $n/2^k=1$, i.e. $k=\log n$.

With $c(1)=8$ (for instance):

$$c(n)=5^{\log n} *8+7 * \frac{5^{\log n}-1}{4}$$
$$c(n)=\Theta(5^{\log n})= \Theta(n^{\log 5}).$$

Observations

- The recurrence relationship is not very complex
- The computations may seem complex, but they are routine …

- What can be more difficult than that?
  - $c(n)$ depends on several values among $c(n)$, $c(n\text{-}1)$, …
  - It is impossible to find a recurrence relationship :

    Usually, one can find one with ≤, but we feel it is not precise enough.

Plan

- Basics
- Compute the running time of an algorithm:
  - Easy cases
  - More difficult cases
- Graphs are not easy cases
- P, NP, …

Graphs are not easy cases

```
Algorithm InputGraphList1  (n,m : int) : vector L of pointers
Begin  // we assume that the directed graph is simple
     for i from 1 to n do L[i]← null endfor
     for k from 1 to m do
        Write ('Give the endpoints of the next arc (source, target)')
        Read(i,j)   // The numbers of the two vertices are assigned to i and j respectively
        p ← L[i]        // L[i] points to the first node of the list L[i], if such a node exists;
                        // otherwise p=null
        q ← create_node //only the VERY simple instructions which depend on the used
                        //language  may be assumed already implemented.
        q.val ← j;  q.suiv ← p
        L[i] ← q  // j has been added at the beginning of the list L[i]
     endfor
     return L
End
```

Analysis 1

- Size of the input : n (vertices)+m(edges)

Used in all this course !

- Significative unit operations : assignment

- The running time will depend on n and/or m.

  $\rightarrow$ $c(n,m)$ and not $c(n)$!

- Try a direct computation

$$c(n,m) = n + \sum_{k=1}^{m} c(body\ of\ the\ \mathbf{for}\ loop\ for\ k)$$

$$= n + \sum_{k=1}^{m} \Theta(1) = \Theta(n+m)$$

- Easier than expected !

Algorithm InputGraphList2 (n,m : int) : vector L of pointers

Begin // we assume that the directed graph is simple

for i from 1 to n do $L[i] \leftarrow null$ **endfor**

for k from 1 to m do

Write ('Give the endpoints of the next arc (source, target)')

Read(i,j) *// The numbers of the two vertices are assigned to i and j respectively*

$p \leftarrow L[i]$ *// L[i] points to the first node of the list L[i], if such a node exists;*
// otherwise p=null

$q \leftarrow$ create_node *//only the VERY simple instructions which depend on the used*

//language may be assumed already implemented.

$q.val \leftarrow j;$ $q.suiv \leftarrow null;$

while $(p \neq null)$ and $(p.suiv \neq null)$ do $p \leftarrow p.suiv$ **endwhile**

if $(p=null)$ then $L[i] \leftarrow q$

else $p.suiv \leftarrow q$ *// j has been added at the end of the list L[i]*

endfor

return L

End

Analysis 2

- Try a direct computation

$$c(n,m) = n + \sum_{k=1}^{m} (\Theta(1) + c(\textbf{while}\ loop\ for\ k))$$

$$= n + \Theta(m) + \textstyle\sum_{k=1}^{m} \boxed{c(\textbf{while}\ loop\ for\ k)}$$

= ???

- Solution (we are happy with less)

$c(\textbf{while}\ loop\ for\ k) \leq n\text{-}1$ and $c(while\ loop\ for\ k) \leq m$

thus

$$\boxed{c(\textbf{while}\ loop\ for\ k) \leq \min(m, n\text{-}1)}$$

and $c(n,m) \leq n + \Theta(m) + \sum_{k=1}^{m} \min(m, n-1)$

$$c(n,m) = O(n + m * \min(m,n) + m) = O(n + m * \min(m,n))$$

Analysis 2 (continued)

- Is this a good evaluation of the running time in the worst case ?

  Try an experimental evaluation (with examples).

- Sometimes one do not see a better approach.
- Sometimes one can show that the upper bound we found is also a lower bound.
- $O(\,)$ then becomes $\Theta(\,)$.

  For instance, an upper bound of $O(n)$ for an algorithm that searches a given value in a vector of size n: we know that we must look at each cell of the vector, thus performing **at least** n operations $\rightarrow \Theta(n)$.

What if these methods fail ?

- Change the viewpoint. (Also useful in other situations !)
- Focus on the data (instead of the algorithm)
- Evaluate the number of times each data is used in the unit operations you chose to count, globally (for the entire algorithm)

Example

Assume undirected G is (already) stored using adjacency list L

- Compute the degrees of all the vertices, in a vector d.

Algorithm ComputeDegreesList (*L* : adjacency list) : vector *d* of int

Begin // we assume that the graph is undirected and simple

for *i* from *1* to *n* do *d[i]*← *0* **endfor**

for *i* from *1* to *n* do

p ← *L[i]*

while *(p ≠ null)* do

d[i]← *d[i]+1*

p ← *p.suiv*

endwhile

endfor

return *d*

End

- Unit operation: assignment of a value to a pointer (note: it appears inside the most costly loop)
- Direct computation fails:

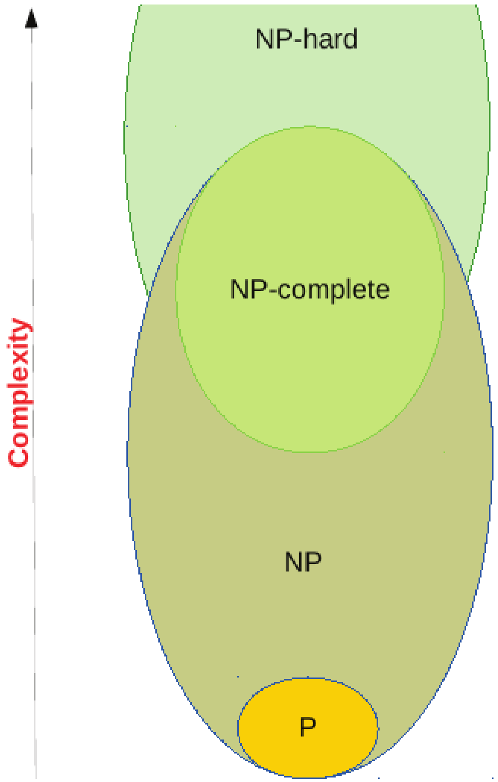
$$c(n,m) = \underbrace{\boxed{\sum_{i=1}^{n}}}_{=n}(1 + \underbrace{\boxed{c(\boldsymbol{while}\ loop\ for\ i)}}_{=???})$$

- Focus on data:
  - Each node of *L* is pointed to exactly once, over all the **while** loops globally
  - #nodes in *L*: *2m* (each edge appears twice)
  - $c(n,m) = \Theta(n+m)$

Plan

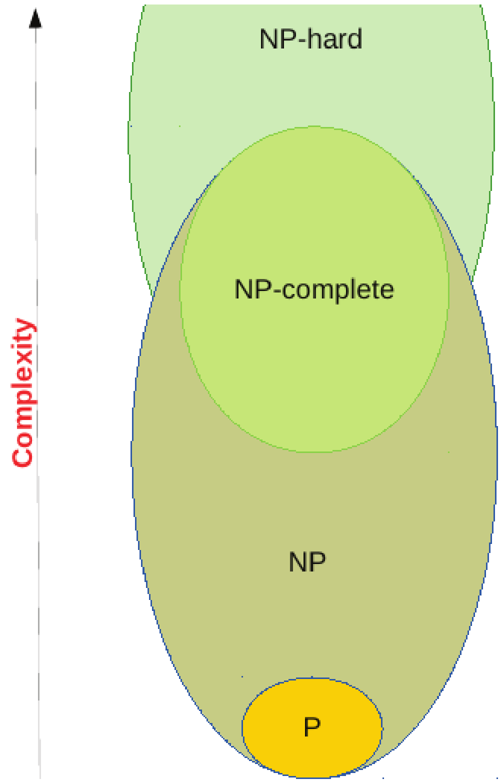
- Basics
- Compute the running time of an algorithm:
  - Easy cases
  - More difficult cases
- Graphs are not easy cases
- P, NP, …

P, NP, ...



Only for decision problems (answer Yes/No)

P, NP, ...



Only for decision problems (answer Yes/No)

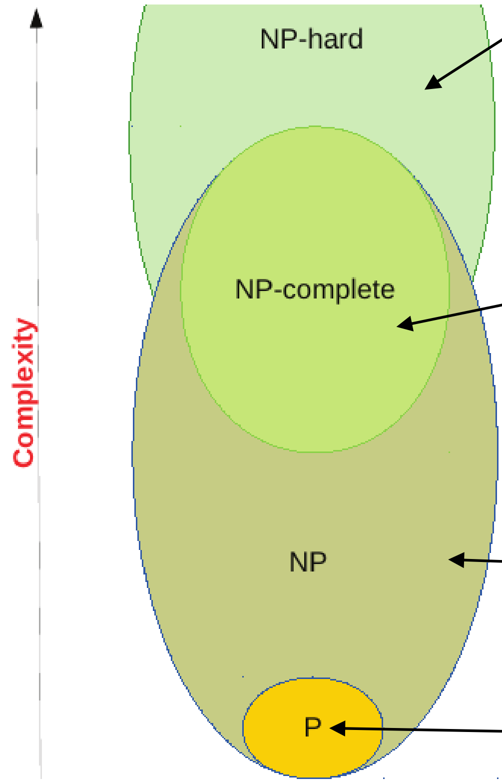
Intuitively (fast=polynomial time algorithm):

- NP (non-deterministic polynomial)
  - A proposed solution can be tested by a fast algorithm.
- P : polynomial problems
  - Can be solved exactly and fast.
- NP-complete problems
  - Cannot be solved exactly and fast

    (under P $\neq$NP hypothesis)
- NP-hard problems
  - At least as difficult as NP-complete

NP-complete = NP ∩ NP-hard

Examples



Halting problem

Satisfiability

Maximum Clique (as a decision pb.)

Minimum Vertex Cover (as a decision pb.)

Hamiltonian Path

Integer programming (as a decision pb.)

Graph isomorphism (?)

(under P≠NP hypothesis)

Graph connectivity

Graph bipartition

Linear programming (as a decision pb.)

Only for decision problems (answer Yes/No)

Conclusions

- Carefully choose the unit operations to count
- Try:
  - Direct computation
  - Find a recurrence relationship and solve it
  - Focus on data and count the number of times each data is used in the unit operations
- Look for identities (”$c(n,m) =$ ”) rather than inequations (”$c(n,m) \leq$ ”) in order to obtain $\Theta(\ )$ rather than $O(\ )$.

A good solution for a problem =

an efficient algorithm + a good evaluation of its running time.

- Efficiency depends on the difficulty of the problem.

  → don’t try to find a polynomial algorithm for an NP-complete problem !