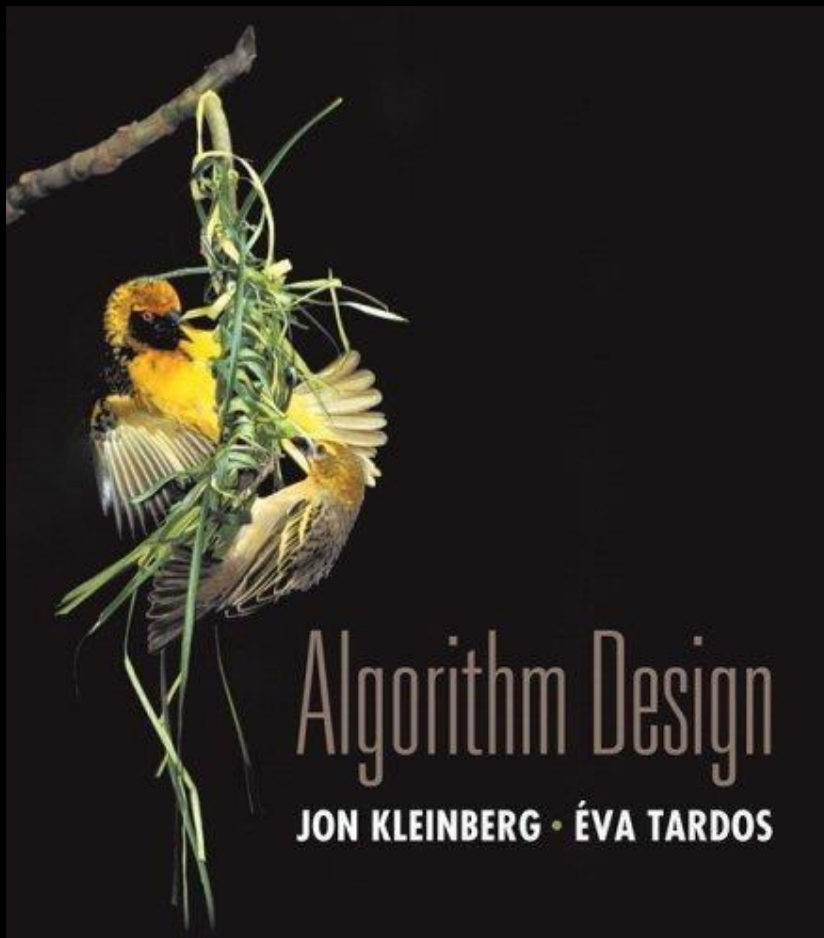


Chapter 1.3

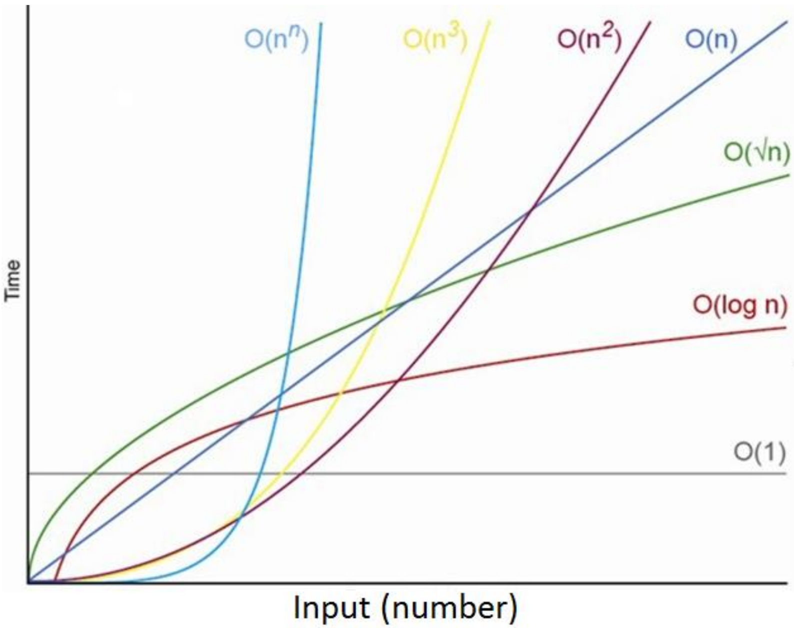
Basics of Algorithm Analysis

Slides by Kevin Wayne.

S1.3: Basics of Algorithm Analysis

Análisis y Diseño de Algoritmos (Algorítmica III)

Prof. Herminio Paucar



O

1.3.1 Time Complexity of an Algorithm

Purpose

- To estimate how long a program will run
- To estimate the largest input that can reasonably be given to the program
- To compare the efficiency of different algorithms
- To choose an algorithm for an application

Time complexity is a function

Time for a sorting algorithm is different for sorting 10 numbers and sorting 1,000 numbers

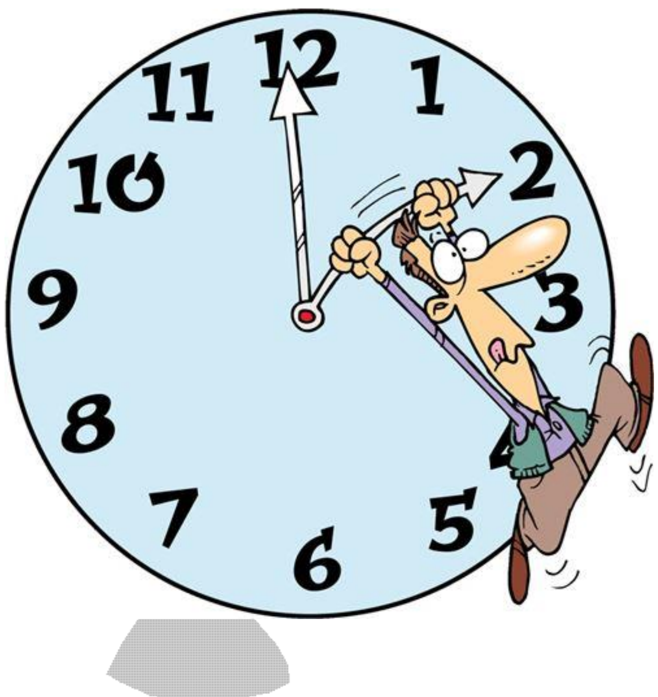
Time complexity is a function: Specifies how the running time depends on the size of the input.

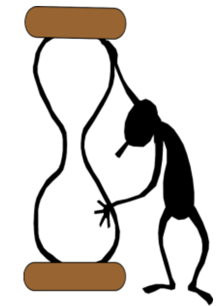
Function mapping

"size" n of input

"time" $T(n)$ executed by algorithm

Definition of time?

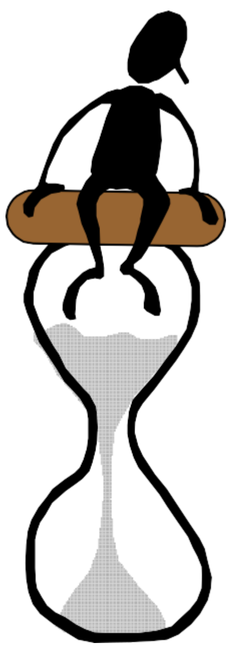




Definition of time?

A. # of seconds
B. # lines of code executed
C. # of simple operations performed

A. Problem: machine dependent
B. Problem: lines of diff. complexity
C. This is what we will use



Size of input instance?

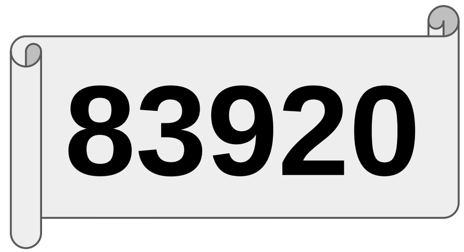
Formally: Size n is number of bits to represent instance

But we can work with anything reasonable

reasonable = within a constant factor of number of bits

Size of input instance

Ex 1:



- **# of bits** → *n = 17 bits*
- **# of digits** → *n = 5 digits*
- **Value: 83920** → *n = 83920*

Which are reasonable?

Size of input instance

Ex 1:

- **# of bits: 17 bits – Formal**
- **# of digits: 5 digits – Reasonable**

#bits and #digits are always within constant factor ($\approx log_2 10 \approx 3.32$),

# of bits = 3.32*# of digits

Size of input instance

Ex 1:

- **# of bits: 17 bits – Formal**
- **# of digits: 5 digits – Reasonable**
- **Value: 83920 – Not reasonable**

  # of bits = $log_2(value)$

  Value= $2^{\#bits}$, much bigger

Size of input instance

Ex 2:

$$\overbrace{\boxed{14,23,25,30,31,52,62,79,88,98}}^{10}$$

- # of elements → $n = 10$

Is this reasonable?

Size of input instance

Ex 2:

$$\overbrace{\boxed{14,23,25,30,31,52,62,79,88,98}}^{10}$$

- **# of elements → n = 10 Reasonable**
  if each element has c bits, total number of bits is:
  #bits = c * #elements

Time complexity is a function

Specifies how the running time depends on the size of the input

Function mapping

# of bits n to represent input

$\Downarrow$

# of basic operations $T(n)$ executed by the algorithm

Which input of size n?

Q: There are 2^n inputs of size n.
Which do we consider for the time complexity T(n)?

Worst instance

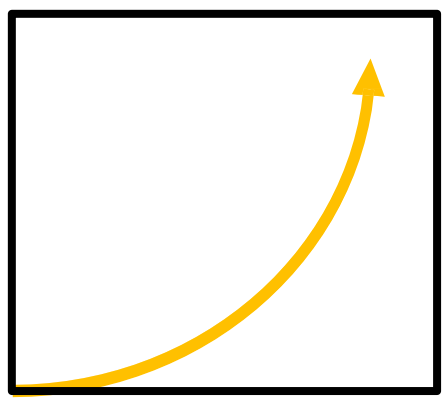
Worst-case running time. Consider the instance where the algorithm uses largest number of basic operations

- Generally captures efficiency in practice
- Pessimistic view, but hard to find better measure

Time complexity

We reach our final definition of time complexity:

> T(n) = number of basic operations the algorithm takes over the worst instance of bit-size n



Example

```
Func Find10(A)                    #A is array of 32-bit numbers
   For i=1 to length(A)
     If A[i]==10
        Return i
```

Q: What is the time complexity $T(n)$ of this algorithm?

A: $T(n) \approx (n/32) + 1$

- Worst instance: the only "10" is in the last position
- A of bit-size n has $n/32$ numbers
- ≈ 1 simple operations per step (if), +1 for Return

1.3.2 Asymptotic Order of Growth

Asymptotic Order of Growth

Motivation: Determining the exact time complexity T(n) of an algorithm is very hard and often does not make much sense.

Algorithm 1:

```
x←100
For i=1...N
    j←j+1
    If x>50 then
            x←x+3
    End If
End For
```

Time complexity:

- 2N +1 assignments
- N comparisons
- 2N additions

Total: 5N+1 basic operations

T(N) = 5N+1

Asymptotic Order of Growth

Algorithm 2:

$x \leftarrow 20$
For $i=1...N$
    $x \leftarrow 3x$
End For

Time complexity

- N + 1 assignments
- N multiplications

Total: 2N+1 basic operations

Can we say Algorithm 2 is going to run faster than Algorithm 1?

Not clear, depends on the time it takes for addition, assignment, multiplication

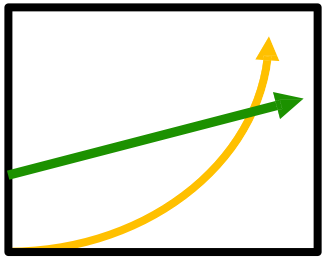
Asymptotic Order of Growth

- No vale mucho la pena complicar la metodología estimando las constantes.
- En lugar de calcular T(n) exactamente, queremos sólo cotas superiores (e inferiores) para T(n) ignorando factores constantes.

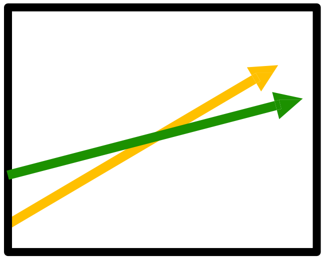
Asymptotic Order of Growth

Upper bounds *O*

Informal: T(n) is O(f(n)) if T(n) grows with at most the same order of magnitude as f(n) grows



T(n) is O(f(n))



T(n) is O(f(n))

both grow at same order of magnitude

Asymptotic Order of Growth

Upper bounds - *Cota Superior* ***O***

Formal: T(n) is O(g(n)) if there exist constants $c \geq 0$ such that for all $n \geq n_0$ we have.

T(n) ≤ c · g(n).

Equivalent: T(n) is O(g(n)) if there exists $\mathrm{k} \geq 0$ such that

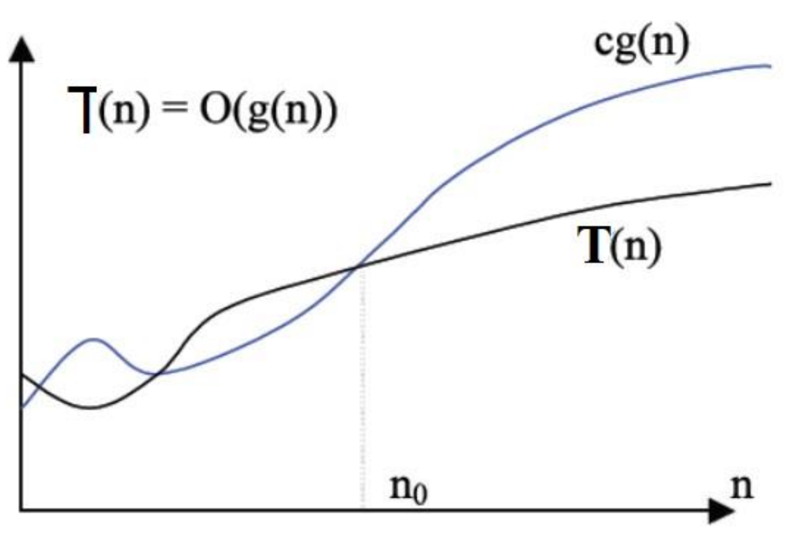
$$\lim_{n \to \infty} \frac{T(n)}{g(n)} \leq \boldsymbol{k}$$

Asymptotic Order of Growth

Upper bounds – *Cota Superior O*

Definición Formal: **Sea *g:* N→[0,∞). Se define el conjunto de funciones de orden O(g(n)) como:**

$$\boxed{T(n) = \{O(g(n)) : \exists\, c, n_0 > 0 \setminus T(n) \leq c * g(n), \forall n \geq n_0\}}$$



Diremos que una función *t:* N →[0,∞) es de orden O de *g* si *t* ∈*O(g)*.
Intuitivamente, *t* ∈*O(g)* indica que *t* está acotada superiormente por algún múltiplo de *g*.
Normalmente estaremos interesados en la menor función *g* tal que *t* pertenezca a *O(g)*

Upper bounds – *Cota Superior O*

Propiedades de O

Veamos las propiedades de la cota superior. La demostración de todas ellas se obtiene aplicando la definición formal.

1. Para cualquier función T se tiene que $T \in O(T)$.
2. $T \in O(g) \Rightarrow O(T) \subset O(g)$.
3. $O(T) = O(g) \Leftrightarrow T \in O(g)$ y $g \in O(T)$.
4. Si $T \in O(g)$ y $g \in O(h) \Rightarrow T \in O(h)$.
5. Si $T \in O(g)$ y $T \in O(h) \Rightarrow T \in O(\min(g,h))$.
6. Regla de la suma: Si $T1 \in O(g)$ y $T2 \in O(h) \Rightarrow T1 + T2 \in O(\max(g,h))$.
7. Regla del producto: Si $T1 \in O(g)$ y $T2 \in O(h) \Rightarrow T1 \cdot T2 \in O(g \cdot h)$.

Upper bounds - *Cota Superior O*

8. Si existe $\lim_{n \to \infty} \frac{T(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:

A. Si $k \neq 0$ y $k < \infty$ entonces $O(T) = O(g)$.
B. Si $k = 0$ entonces $T \in O(g)$, es decir, $O(T) \subset O(g)$, pero sin embargo se verifica que $g \notin O(T)$.

Obsérvese la importancia que tiene el que exista tal límite, pues si no existiese (o fuera infinito) no podría realizarse tal afirmación.

Asymptotic Order of Growth

Exercise 1: $T(n) = 32n^2 + 17n + 32$.

Say if $T(n)$ is:

- $O(n^2)$?
- $O(n^3)$?
- $O(n)$?

Asymptotic Order of Growth

Exercise 1: $T(n) = 32n^2 + 17n + 32$

Say if T(n) is:

- $O(n^2)$?   Yes
- $O(n^3)$?   Yes
- $O(n)$?   No

Solution: To show that T(n) is $O(n^2)$ we can:

- Use the first definition with c = 1000
- Use limits:

$$\lim_{n\to\infty} \frac{T(n)}{n^2} = 32$$, which is a constant

Asymptotic Order of Growth

Exercise 2:

- $T(n) = 2^{n+1}$, is it $O(2^n)$?
- $T(n) = 2^{2n}$, is it $O(2^n)$?

Asymptotic Order of Growth

Exercise 2:

- **$T(n) = 2^{n+1}$, is it $O(2^n)$? Yes**
- **$T(n) = 2^{2n}$, is it $O(2^n)$? No**

Solution (second item): $\lim_{n\to\infty} \frac{T(n)}{2^n} = \lim_{n\to\infty} 2^n = \infty$
is not constant

Solution 2 (second item): **To have $2^{2n} < c.2^n$ we need $c>2^n$. So c is not.**

Asymptotic Bounds for Some Common Functions

Logarithms. $log_a n$ **is** $O(log_b n)$ **for any constants** $a, b>0$

can avoid specifying the base

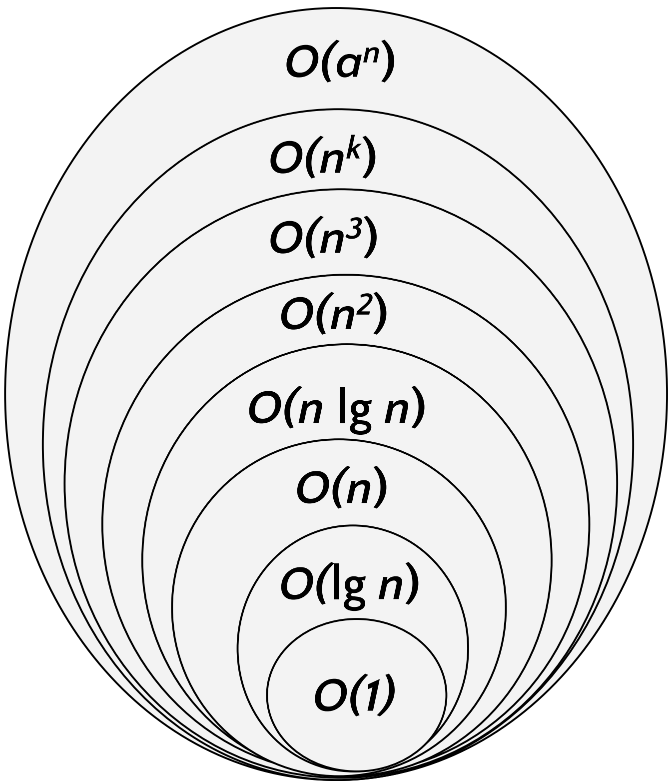
Logarithms. **For every** $x>0$, $log\ n$ **is** $O(n^x)$

log grows slower than every polynomial

Exponentials. **For every** $r>1$ **and every** $d>0$, n^d **is** $O(r^n)$

every exponential grows faster than every polynomial

Asymptotic Bounds for Some Common Functions



Donde: k>=4, a>1

classe	nome
$O(1)$	constante
$O(\lg n)$	logarítmica
$O(n)$	linear
$O(n \lg n)$	$n \log n$
$O(n^2)$	quadrática
$O(n^3)$	cúbica
$O(n^k)$ com $k >= 1$	polinomial
$O(a^n)$ com $a > 1$	exponencial

Asymptotic Bounds for Some Common Functions

Exercise: is T(n) = 21·n·log n

- O(n^2) ?
- O($n^{1.1}$) ?
- O(n) ?

Asymptotic Bounds for Some Common Functions

Exercise: is T(n) = 21·n·log n

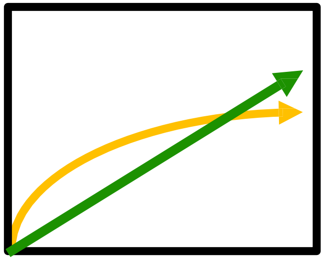
- O(n^2) ? Yes
- O($n^{1.1}$) ? Yes
- O(n) ? No

Solution (first item): Comparing *21·n·log n vs n^2* is the same as comparing *21·log n vs n*, and we know log n grows slower than *n*.

Solution 2 (first item): $\lim_{n\to\infty} \frac{T(n)}{n^2} = \lim_{n\to\infty} \frac{21 \log(n)}{n}$, which is at most a constant since log n grows slower than *n*

Lower Bounds - *Cota Inferior* Ω

Informal: T(n) is Ω(g(n)) if T(n) grows with at least the same order of magnitude as g(n) grows.



Formal: ***T(n)*** is ***Ω(g(n))*** if there exist constants *c>0* such that for all ***n*** we have $T(n) \geq c \cdot g(n)$.

Equivalent: $T(n)$ is $\Omega(g(n))$ if there exist constant $k>0$

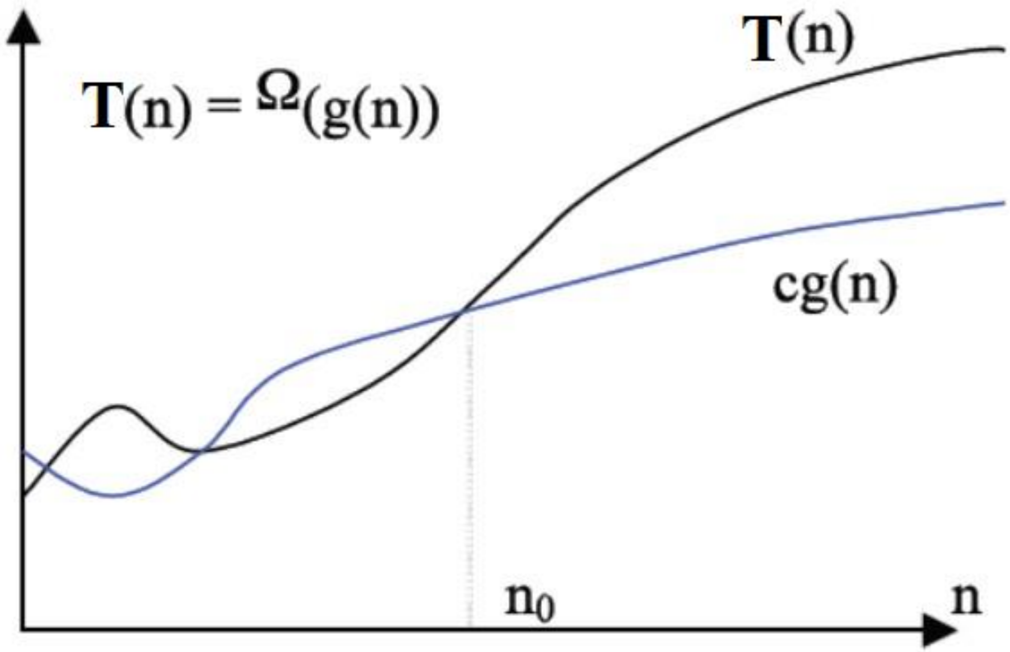
$$\lim_{n \to \infty} \frac{T(n)}{g(n)} \geq \boldsymbol{k}$$

Lower Bounds - *Cota Inferior* Ω

Definición Formal: **Sea** *g: N→[0,∞)*. **Se define el conjunto de funciones de orden Ω(Omega) de** *g* **como:**

$$\boxed{T(n) = \{\Omega(g(n)) : \exists\, c, n_0 > 0 \setminus T(n) \geq c * g(n), \forall n \geq n_0\}}$$

Diremos que una función *t:N →[0,∞)* **es de orden Ω de g si** *t ∈ Ω(g).*



Lower Bounds - *Cota Inferior* Ω

Propiedades de Ω: Veamos las propiedades de la cota inferior Ω. La demostración de todas ellas se obtiene de forma simple aplicando la definición formal.

1. Para cualquier función T se tiene que $T \in \Omega(T)$.
2. $T \in \Omega(g) \Rightarrow \Omega(T) \subset \Omega(g)$.
3. $\Omega(T) = \Omega(g) \Leftrightarrow T \in \Omega(g)$ y $g \in \Omega(T)$.
4. Si $T \in \Omega(g)$ y $g \in \Omega(h) \Rightarrow T \in \Omega(h)$.
5. Si $T \in \Omega(g)$ y $T \in \Omega(h) \Rightarrow T \in \Omega(\max(g,h))$.
6. Regla de la suma: Si $T1 \in \Omega(g)$ y $T2 \in \Omega(h) \Rightarrow T1+T2 \in \Omega(g+h)$.
7. Regla del producto: Si $T1 \in \Omega(g)$ y $T2 \in \Omega(h) \Rightarrow T1 \cdot T2 \in \Omega(g \cdot h)$.

Lower Bounds - *Cota Inferior* **Ω**

8. Si existe $\lim\limits_{n \to \infty} \dfrac{T(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:

a) Si k ≠ 0 y k < ∞ entonces Ω(T) = Ω(g).

b) Si k = 0 se verifica que T ∉ Ω(g), pero sin embargo se verifica que g ∈ Ω(T), es decir, Ω(g) ⊂ Ω(T).

Lower Bounds - *Cota Inferior* Ω

Exercise: $T(n) = 32n^2 + 17n + 32$. Is $T(n)$:

- A) $\Omega(n)$?
- B) $\Omega(n^2)$?
- C) $\Omega(n^3)$?

Lower Bounds - *Cota Inferior* Ω

Exercise: $T(n) = 32n^2 + 17n + 32$ Is $T(n)$:

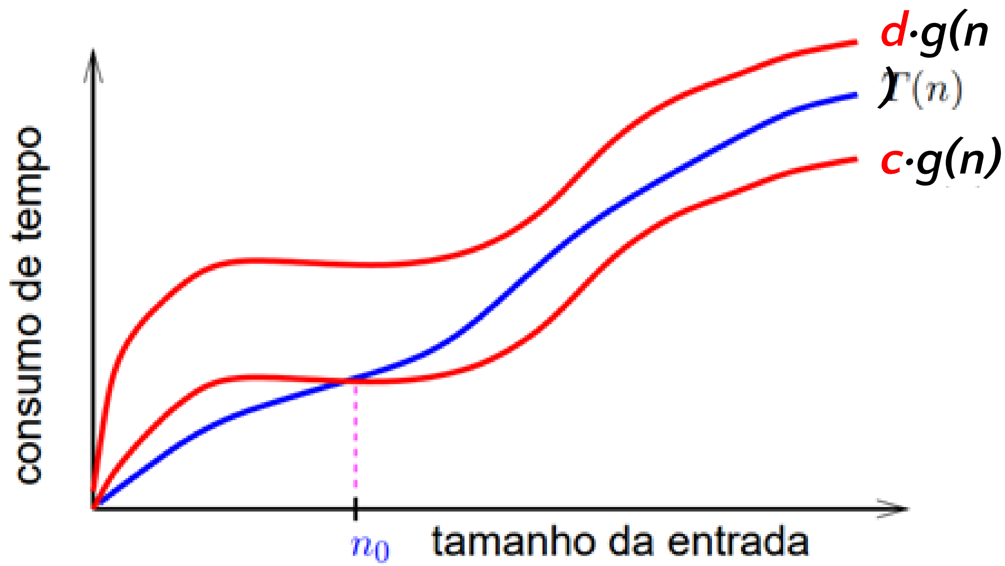
- $\Omega(n)$? Yes
- $\Omega(n^2)$? Yes
- $\Omega(n^3)$? No

Tight Bounds – Cota Promedio Θ

Tight bounds. **T(n) is Θ(g(n)) if T(n) is both O(g(n)) and Ω(g(n)).**

Formal: $T(n)$ **is** $\Theta(g(n))$ **if there exist constants** $c>0$ **and** $d>0$ **such that for all** n **we have**

$$c \cdot g(n) \leq T(n) \leq d \cdot g(n)$$



Tight Bounds – Cota Promedio Θ

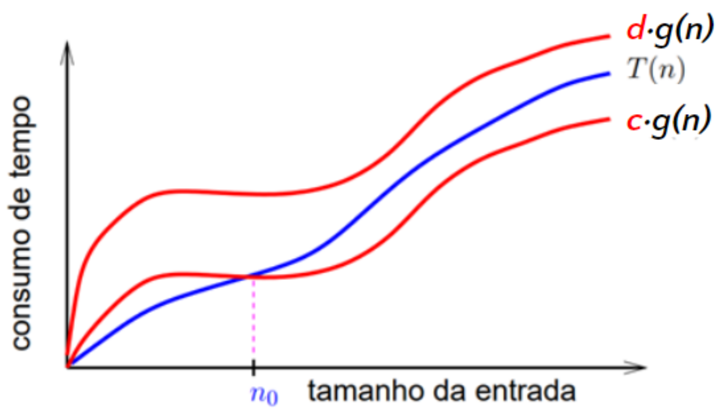
Definición Formal. **Sea T: N→[0,∞). Se define el conjunto de funciones de orden Θ (Theta) de *g* como:**

$$\Theta(T) = O(T) \cap \Omega(T)$$

o, lo que es igual:

$$T(n) = \{\theta(g(n)) : \exists\, c, d, n_0 > 0 \setminus c * g(n) \leq T(n) \leq d * g(n), \forall n \geq n_0\}$$

Diremos que una función t :N→[0,∞) es de orden Θ de *g* si $t \in \Theta(g)$.
Intuitivamente, *t ∈ Θ(g)* indica que *t* está acotada tanto superior como
inferiormente por múltiplos de *g*, es decir, que *t* y g crecen de la misma forma.



Tight Bounds – Cota Promedio Θ

Propiedades Θ: Veamos las propiedades de la cota exacta. La demostración de todas ellas se obtiene también de forma simple aplicando la definición formal y las propiedades de O y Ω.

1. Para cualquier función T se tiene que $T \in \Theta(T)$.
2. $T \in \Theta(g) \Rightarrow \Theta(T) = \Theta(g)$.
3. $\Theta(T) = \Theta(g) \Leftrightarrow T \in \Theta(g)$ y $g \in \Theta(T)$.
4. Si $T \in \Theta(g)$ y $g \in \Theta(h) \Rightarrow T \in \Theta(h)$.
5. Regla de la suma: Si $T1 \in \Theta(g)$ y $T2 \in \Theta(h) \Rightarrow T1+T2 \in \Theta(\max(g,h))$.
6. Regla del producto: Si $T1 \in \Theta(g)$ y $T2 \in \Theta(h) \Rightarrow T1 \cdot T2 \in \Theta(g \cdot h)$.

Tight Bounds - Cota Promedio Θ

7. Si existe $\lim_{n\to\infty} \frac{T(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:

> **a) Si k ≠ 0 y k < ∞ entonces Θ(T) = Θ(g).**
>
> **b) Si k = 0 se verifica que Θ(T) ≠ Θ(g), es decir, los órdenes exactos de T y g son distintos.**

Lower and Tight Bounds

Exercise: $T(n) = 32n^2 + 17n + 32.$

Is T(n):

- A) $\Theta(n^2)$?
- B) $\Theta(n)$?
- C) $\Theta(n^3)$?

Lower and Tight Bounds

Exercise: $T(n) = 32n^2 + 17n + 32$

Is T(n):

- $\Theta(n^2)$? Yes
- $\Theta(n)$? No
- $\Theta(n^3)$? No

Lower and Tight Bounds

Exercise: Show that $\log(n!)$ is $\Theta(n \log n)$

Lower and Tight Bounds

Answer:

- First we show that $\log(n!) = O(n \log n)$

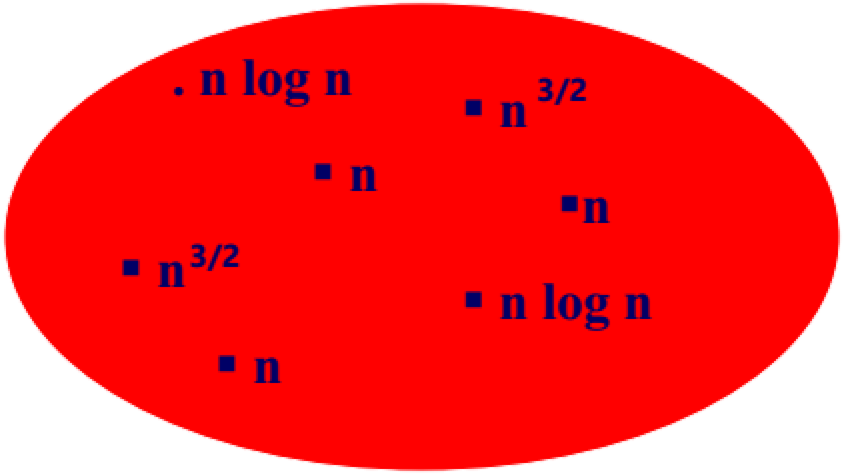
$\log(n!) = \log n + \log(n-1) + \ldots \log 1 < n. \log n$, since the log function is increasing

- Now we show that $\log(n!) = \Omega(n \log n)$

$\log(n!) = \log n + \log(n-1) + \ldots \log 1 > n/2. \log(n/2) = n/2 (\log n - 1)$

Upper and Lower bounds

inputs of size n for algorithm A (cartoon)



Can we say that the time complexity of A is?

- $O(n^2)$? Yes, because largest complexity of algorithm is at most n^2
- $\Omega(n^2)$? No, there are inputs where complexity has larger order
- $\Omega(n)$? Yes
- $O(n)$? No, there is no input where the complexity of the algorithm has order n^2
- $\Omega(n^{3/2})$? No

Implication of Asymptotic Analysis

Hypothesis

- Basic operations (addition, comparison, shifts, etc) takes at least 10ms and at most 50ms seconds (This is time unity = u)

Algorithms

- Algorithm A executes 20n operations for the worst instance ($O(n)$)
- Algorithm B executes n^2 operations for the worst instance ($\Omega(n^2)$)

Conclusion

- For a instance of size n, A spends at most 1000n ms
- For the worst instance of size n, B spends at least 10 n^2 ms
- For n>100, A is faster than B in the worst case, regargless of which operations they execute.

Allows us to tell which algorithm is faster (for large instances)

Notation

Slight abuse of notation. $T(n) = O(f(n))$

- Be careful: Asymmetric:
  - $f(n) = 5n^3$; $g(n) = 3n^2$
  - $f(n) = O(n^3) = g(n)$
  - but $f(n) \neq g(n)$
- Better notation: $T(n) \in O(f(n))$.

Exercícios

Exercícios Kleinberg & Tardos, cap 2 da lista de exercícios

1.3.3 A Survey of Common Running Times

Linear Time: O(n)

Linear time. Running time is at most a constant factor times the size of the input.

Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a1
for i = 2 to n {
   if (ai > max)
      max ← ai
}
```

Remark. For all instances the algorithm executes a linear number of operations

Linear Time: O(n)

Linear time. Running time is at most a constant factor times the size of the input.

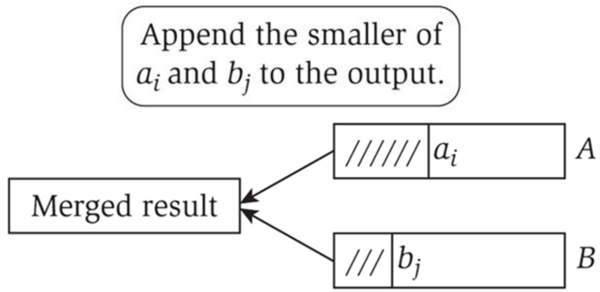
Finding an item x in a list. Test if x is in the list $a_1, \ldots, a_n$

```
Exist ← false
for i = 1 to n {
   if (a_i == x)
      Exist ← true
      break
}
```

Remark. For some instances the algorithm is sublinear (e.g. x in the first position)

Linear Time: O(n)

Merge. Combine two sorted lists $A = a_1,a_2,\ldots,a_n$ with $B = b_1,b_2,\ldots,b_n$ into sorted whole.




```
i = 1, j = 1
while (both lists are nonempty) {
  if (a_i ≤ b_j) append a_i to output list and increment i
  else           append b_j to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size k takes O(n) time (n=total size=2k).
Pf. After each comparison, the length of output list increases by 1.

Claim. Merging two lists of size n takes O(n) time.
Pf. After each comparison, the length of output list increases by 1.

O(n log n) Time

O(n log n) time. Arises in divide-and-conquer algorithms.

Sorting. Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic Time: $O(n^2)$

Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane (x_1, y_1), …, (x_n, y_n), find the distance of the closest pair.

$O(n^2)$ solution. Try all pairs of points.

```
min ← (x1 - x2)² + (y1 - y2)²
for i = 1 to n {
  for j = i+1 to n {
   d ← (xi - xj)² + (yi - yj)²
   if (d < min)
     min ← d
  }
}
```

← don't need to take square roots

← see chapter 5

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion.

Cubic Time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Let $S_1, \ldots, S_n$ be subsets of $\{1, 2, \ldots, n\}$. Is there a disjoint pair of sets?

Set Representation. Assume that each set is represented as an incidence vector.

n=8 and S={2,3,6}, S is represented by

1	2	3	4	5	6	7	8
0	1	1	0	0	1	0	0

n=8 and S={1,4}, S is represented by

1	2	3	4	5	6	7	8
1	0	0	1	0	0	0	0

Algorithm:

```
For i=1…n-1
    For j=i+1…n
        For k=1…n
            If (S_i(k)=S_j(k)=1)
                Return 'There are not disjoint sets'
            End If
        End For
    End For
End For
Return 'There are disjoint sets'
```

Cubic Time: $O(n^3)$

1. A complexidade de tempo do algoritmo é $O(n^3)$?
2. A complexidade de tempo do algoritmo é $\Omega(n^3)$?

Cubic Time: $O(n^3)$

1. A complexidade de tempo do algoritmo é $O(n^3)$? SIM
2. A complexidade de tempo do algoritmo é $\Omega(n^3)$? SIM

"Bad" instance: all sets are equal to $\{n\}$ => algorithm makes $\Omega(n^3)$ basic operations

Exponential Time

1. Independent set. Given a graph, find the largest independent set?
   a. $O(n^2 2^n)$ solution. Enumerate all subsets.

```
S* ← φ
foreach subset S of nodes {
    check whether S in an independent set
    if (S is largest independent set seen so far) update
        S* ← S
    }
}
```

2. Decrypt a numeric (0…9) password of *n* elements

_	_	…	_	_
10	10	…	10	10

n: elements

The complexity of Algorithm is $O(10^n)$

Polynomial Time

Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n.

Ex: $T(n) = 32n^2$ and $T(n) = n \log(n)$ are polynomial time
We consider an algorithm efficient if time complexity is polynomial

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Polynomial Time

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Complexity of Algorithm vs Complexity of Problem

There are many different algorithms for solving the same problem

Showing that an algorithm is $\Omega(n^3)$ does not mean that we cannot find another algorithm that solves this problem faster, say in $O(n^2)$

Exercício

Exercício 1. Considere um algoritmo que recebe um número real x e o vetor $(a_0, a_1, \ldots, a_{n-1})$ como entrada e devolve

$$a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$$

A. Desenvolva um algoritmo para resolver este problema que execute em tempo quadrático. Faça a análise do algoritmo.

A. Desenvolva um algoritmo para resolver este problema que execute em tempo linear. Faça a análise do algoritmo

Exercício – Solução

$a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$

a)

1. sum = 0
2. **Para** i= 0 até n-1 **faça**
3. t ← 1
4. **Para** j:=1 até i
5. t← t . x
6. **Fim Para**
7. sum ← sum + t*a_i
8. **Fim Para**
9. **Devolva** sum

$a_0 + a_1x + \ldots + a_{n-1}x^{n-1}$

Donde: A [], x

Análise: Número de operações elementares é igual a:

$T(n) = 1+2+3+ \ldots + n-1 = n(n-1)/2 = O(n^2)$

Exercício – Solução

b)

```
sum = a0
pot = 1
Para i= 1 até n-1 faça
        pot ← x*pot
        sum ← sum + ai*pot
Fim Para
Devolva sum
```

Análise: A cada loop são realizadas O(1) operações elementares. Logo, o tempo é linear

Recap

T(n) is $O(f(n))$: T(n) grows "slower" than f(n)

T(n) is $\Omega(f(n))$: T(n) grows at least as fast as f(n)

T(n) is $\Theta(f(n))$: T(n) is both $O(f(n))$ and $\Omega(f(n))$

Exercised design and analysis of simple algorithms, giving upper bound O(f(n))

right order of growth

Observations on $\Omega(f(n))$

```
Func Find10(A)
    For i=1 to length(A)        #A is array of 32-bit numbers
      If (A[i]==10)
          Return i
```

Q: What is the time complexity $T(n)$ of this algorithm? Give upper bound O(.) and lower bound Ω(.)

A: $\Theta(n)$: In worst-case, does n iterations of the for => O(n) and Ω(n).

Point: We always consider worst instance, Omega(n) does not mean that all instances take time ≥ **~n**

Observations on $\Omega(f(n))$

Find closest pair of points, given input pairs $(x_1, y_2),\ldots,(x_n, y_n)$

```
min ← (x1 - x2)^2 + (y1 - y2)^2
for i = 1 to n-1 {
    for j = i+1 to n {
        d ← (xi - xj)^2 + (yi - yj)^2
        if (d < min)
            min ← d
    }
}
```

Q: Is this algo $\Omega(n^2)$?

A: Yes. Does exactly "combinação de n itens 2 a 2" iterations of for => n*(n-1)/2 iterations = $n^2/2 - n/2$ => quadratic growth

Observations on $\Omega(f(n))$

Method 2: Write #iterations as big summation, lower bound

n + (n–1) + (n–2) + … + 2 + 1 >= ??

```
min ← (x1 - x2)^2 + (y1 - y2)^2
for i = 1 to n-1 {
    for j = i+1 to n {
        d ← (xi - xj)^2 + (yi - yj)^2
        if (d < min)
            min ← d
    }
}
```

Trick: Just keep the highest n/2 terms

Back to example n + (n–1) + (n–2) + … + 2 + 1

>= n + (n–1) + (n–2) + … + n/2

>= (n/2)*(n/2) = $n^2/4$

Keep largest n/2 terms

Lower bounded each term by minimum

Ex: Show that $1^2+2^2+\ldots+n^2 = \Omega(n^3)$

Complexity in Code Scripts

Exercise 1

Get the complexity “O(n)” of next pseudocode-1

Pseudocodigo-1

$soma \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 1$ to n^2 **do**
        $soma++$
        $aux \leftarrow soma$
        while $aux > 1$ **do**
            $aux \leftarrow aux/2$
        end while
    end for
end for

Exercise 1

Get the complexity "O(n)" of next pseudocode-1

Pseudocodigo-1

$soma \leftarrow 0$ P
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to n^2 **do**
    $soma++$
    $aux \leftarrow soma$ N
    while $aux > 1$ **do** R Q
      $aux \leftarrow aux/2$ M
    end while
  end for
end for

Tenemos:

T(n) = P + Q

Donde:

$$Q = \sum_{i=1}^{n} R$$

$$R = \sum_{j=1}^{n^2} (N + M)$$

Exercise 1

Get the complexity of (P, Q, R, M and N)

Pseudocodigo-1

$soma \leftarrow 0$ P = O(1)

for $i \leftarrow 1$ to n **do**

  for $j \leftarrow 1$ to n^2 **do**

    $soma++$

    $aux \leftarrow soma$ N = O(1)

    while $aux > 1$ **do** R = O(?) Q = O(?)

      $aux \leftarrow aux/2$ M = O(log(aux))

    end while

  end for

end for

Exercise 1

Get the complexity of (P, Q, R, M and N)

Pseudocodigo-1

$soma \leftarrow 0$ P = O(1) => 1

for $i \leftarrow 1$ to n **do**

  for $j \leftarrow 1$ to n^2 **do**

    $soma++$

    $aux \leftarrow soma$ N = O(1) => c

    while $aux > 1$ **do**

      $aux \leftarrow aux/2$ M = O(log(aux)) => log(aux)

    end while

  end for R = O(?)

end for Q = O(?)

Exercise 1

Get the complexity of (P, Q, R, M and N)

Pseudocodigo-1

$soma \leftarrow 0$ P = O(1) => 1
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to n^2 **do**
    $soma++$
    $aux \leftarrow soma$ N = O(1) => c
    while $aux > 1$ **do**
      $aux \leftarrow aux/2$ M = O(log(aux)) => log(aux)
    end while
  end for
end for

Q

Tenemos: T(n) = P + Q

Donde:

$$Q = \sum_{i=1}^{n} R$$

$$R = \sum_{j=1}^{n^2} (N + M)$$

Reemplazando:

$$T(n) = 1 + \sum_{i=1}^{n} \sum_{j=1}^{n^2} (c + \log aux)$$

Además *aux* varia en función de *i* & *j*:

$$aux = soma = f(i, j) = (i-1)n^2 + j$$

Exercise 1

Resolving

$$T(n) = 1 + \sum_{i=1}^{n} \sum_{j=1}^{n^2} (c + \log aux)$$

$$T(n) = 1 + \sum_{i=1}^{n} \sum_{j=1}^{n^2} (c + \log soma) = 1 + \sum_{i=1}^{n} \sum_{j=1}^{n^2} [c + \log((i-1)n^2 + j)];$$

Hacemos $m = (i-1)n^2$; Reemplazando:

$$T(n) = 1 + \sum_{i=1}^{n} \sum_{j=1}^{n^2} [c + \log(m+j)] = 1 + \sum_{i=1}^{n} \sum_{j=1}^{n^2} c + \sum_{i=1}^{n} \sum_{j=1}^{n^2} \log(m+j)$$

$$T(n) = 1 + cn^3 + \sum_{i=1}^{n} [\log(m+1) + \log(m+2) + \log(m+3) + \ldots + \log(m+n^2)] \leq cn^3 + \sum_{i=1}^{n} [\log((m+n^2)!)];$$

De la propiedad: $O(\log(n!)) = O(n \log n)$ Aplicando, tenemos:

$T(n) \leq cn^3 + \sum_{i=1}^{n} c_1 (m+n^2) \log(m+n^2)$, Reemplazando: $m = (i-1)n^2$; Tenemos:

$$T(n) \leq cn^3 + \sum_{i=1}^{n} c_1 (in^2) \log(in^2) = cn^3 + c_1 n^2 \sum_{i=1}^{n} i(\log n^2 + \log i) = cn^3 + c_1 n^2 [\log n^2 \sum_{i=1}^{n} i + \sum_{i=1}^{n} i \log i]$$

$= cn^3 + c_1 n^2 [\cancel{2} \log n (\frac{n(n+1)}{\cancel{2}}) + \sum_{i=1}^{n} i \log i]$, Hacemos: $S = \sum_{i=1}^{n} i \log i$

Exercise 1

Resolving

$$= cn^3 + c_1 n^2[\cancel{2}\log n(\frac{n(n+1)}{\cancel{2}}) + \sum_{i=1}^{n} i\log i], \text{ Hacemos: } S = \sum_{i=1}^{n} i\log i$$

$$S = \sum_{i=1}^{n} i\log i = 1\log 1 + 2\log 2 + 3\log 3 + \ldots n\log n \leq c_2 n^2 \log n \text{ Reemplazando: } S \text{ , tenemos:}$$

$$T(n) \leq cn^3 + c_1 n^2[n^2\log n + n\log n + c_2 n^2 \log n] = cn^3 + c_1 n^4 \log n + c_1 n^3 \log n + c_1 c_2 n^4 \log n$$

$$T(n) \in O(n^4 \log n)$$

Exercise 2

Get the complexity of the pseudocode:

Pseudocódigo

$cont \leftarrow 2^n$
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to n **do**
    $cont \leftarrow cont/2$
    $s \leftarrow cont$
    while $s \geq 1$ **do**
      $i \leftarrow i + 1$
      $s \leftarrow s/2$
    end while
  end for
end for

Exercise 2

Get the complexity "O(n)" of pseudocode

Step 01

Pseudocódigo

$cont \leftarrow 2^n$ **P**

for $i \leftarrow 1$ to n **do**

  for $j \leftarrow 1$ to n **do**

    $cont \leftarrow cont/2$

    $s \leftarrow cont$ **N**

    while $s \geq 1$ **do** **R** **Q**

      $i \leftarrow i + 1$ **M**

      $s \leftarrow s/2$

    end while

  end for

end for

Tenemos:

T(n) = P + Q

Donde:

$$Q = \sum_{i=1}^{n} R$$

$$R = \sum_{j=1}^{n} (N + M)$$

Exercise 2

Get the complexity "O(n)" of pseudocode

Step 01

Pseudocódigo

$cont \leftarrow 2^n$ **P** = O(1)
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** n **do**
    $cont \leftarrow cont/2$
    $s \leftarrow cont$
    N = O(1)
    while $s \geq 1$ **do**
      $i \leftarrow i + 1$
      $s \leftarrow s/2$
    end while
    M = O(log *s*)
  end for
  R
end for
Q

Reemplazando:

T(n) = P + Q

Donde:

$$Q = \sum_{i=1}^{n} R$$

$$R = \sum_{j=1}^{n} (N + M)$$

Tenemos:

$$T(n) \leq 1 + \sum_{i=1}^{n} \sum_{j=1}^{n} (1 + log(s))$$

Ahora debemos buscar ***s*** en función de ***i*** y ***j***

Exercise 2

Calculamos el valor de *s = cont* en función de *i* y *j*

$C_\emptyset = 2^n$

$i=1,$

$j=1$ $\quad cont_{11} \leftarrow C_\emptyset/2 = \frac{2^n}{2}$

$j=2$ $\quad cont_{12} \leftarrow C_\emptyset/2^2 = 2^n/2^2$

$j=3$ $\quad cont_{13} \leftarrow C_\emptyset/2^3 = 2^n/2^3 = 2^n/2^j = C_\emptyset/2^j$

$\vdots$

$i=2,$

$j=1$ $\quad cont_{21} \leftarrow C_{1n}/2 = \frac{C_\emptyset/2^n}{2} \Big\} \Rightarrow cont_{2n} \leftarrow C_{1n}/2^n = \frac{C_\emptyset/2^n}{2^n}$

$i=3,$

$j=1$ $\quad cont_{31} \leftarrow C_{2n}/2 = \frac{C_\emptyset}{2^{2n}} \Big\} \; cont_{3n} \leftarrow C_{2n}/2^n = \frac{C_\emptyset/2^{2n}}{2^n}$

$\forall ij \quad cont_{ij} \leftarrow \left(\frac{C_\emptyset}{2^{(i-1)n}}\right)/2^j, \quad C_\emptyset = 2^n$

$= \frac{2^{(2-i)n}}{2^j} = 2^{(2-i)n-j}$

Reemplazando s = $cont_{i,j}$

$$T(n) \leq 1 + \sum_{i=1}^{n}\sum_{j=1}^{n}(1 + log(s))$$

Exercise 2

Reemplazando $s = cont_{i,j} = 2^{(2-i)n-j}$

Tenemos: $T(n) \leq 1 + \sum_{i=1}^{n}\sum_{j=1}^{n}(1 + log(s))$

$$T(n) \leq 1 + \sum_{i=1}^{n}\sum_{j=1}^{n}\left(1 + \log\left[2^{(2-i)n-j}\right]\right)$$

$$T(n) \leq 1 + \sum_{i=1}^{n}\left(n + \sum_{j=1}^{n}[(2-i)n-j]\log 2\right)$$

$$T(n) \leq 1 + n^2 + \sum_{i=1}^{n}\underbrace{\sum_{j=1}^{n}((2-i)n-j)}_{B} \quad \ldots (I)$$

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

$$B = \frac{n(3n-1)}{2} - n^2 i \quad \ldots (II)$$

(II) en (I)

$$T(n) \leq 1 + n^2 + \sum_{i=1}^{n}\left(\frac{n(3n-1)}{2} - n^2 i\right) \leq 1 + \frac{n^2(3n-1)}{2} - n^2\sum_{i=1}^{n} i$$

$$T(n) \leq 1 + \frac{n^2(3n-1)}{2} - \frac{n^2(n)(n+1)}{2} \leq \frac{-n^4+2n^3-n^2+2}{2}$$

$T(n) \leq (-n^4+2n^3-n^2+2)/2$

para un $n \to \infty$

$T(n) < 0$

Volvamos a revisar el pseudocódigo, pero ahora ......!

Exercise-2

Get the complexity “O(n)” of pseudocode

Step 01

Pseudocódigo

$cont \leftarrow 2^n$ = O(1)

for $i \leftarrow 1$ to n **do**

  for $j \leftarrow 1$ to n **do**

    $cont \leftarrow cont/2$

    $s \leftarrow cont$ = O(1)

    while $s \geq 1$ **do**

      $i \leftarrow i + 1$

      $s \leftarrow s/2$

    end while M

  end for N

end for R

Antes de realizar cualquier operación matemática, debemos primero hacer el ***análisis visual***, i.e. “*Traza del Algoritmo*”. Se observa que el algoritmo sólo funciona para un ***i=1***, puesto que el segundo valor de **i** será un valor muy grande igual a ***n***, debido a que se incrementa dentro del while **M**. Se observa que **R** se realiza sólo una vez para un ***i=1***, entonces sólo queda evaluar el número de operaciones elementales dentro de **N** y **M**.

T(n) = 1 + R = 1 + N

$$N = \sum_{j=1}^{n} (1 + M)$$

Exercise 02

Get the complexity "O(n)" of pseudocode

$$T(n) = 1 + N; \quad N = \sum_{j=1}^{n} (1 + M) = \sum_{j=1}^{n} (1 + \log(s))$$

$N = \sum_{j=1}^{n} (1 + M)$, $M = c \cdot \log(s)$, $c = 1$

$s \leftarrow cont$, $cont \leftarrow C_0 / 2^j = 2^n / 2^j = 2^{n-j}$

en "N"

$N = \sum_{j=1}^{n} [1 + \log(2^{n-j})] = n + \sum_{j=1}^{n} ((n-j) \cancel{\log} 2)$

$N = 3n + n^2 - \sum_{j=1}^{n} j = 3n + n^2 - \frac{(n)(n+1)}{2}$

$N \approx c n^2$, $\therefore T(n) \leq 1 + n^2$

$\therefore$ tenemos que el $T(n) \in O(n^2)$

1.3.4 A First Analysis of a Recursive Algorithm

Binary Search

Problem: Given a sorted list of numbers (increasing order) a_1, a_2, a_3,…, a_n, decide if number x is in the list

```
Function bin_search(i,j, x)
    if i = j
        if a_i = x return TRUE
        else return FALSE
    end if
    mid = floor((i+j)/2)
    if x = a_mid
        return x
    else if x < a_mid
        return bin_search(i, mid-1, x)
    else if x > a_mid
        return bin_search(mid+1, j, x)
    end if
```

```
Function bin_search_main(x)
    bin_search(1,n, x)
```

Ex: x=14

1	2	3	5	7	10	14	17

7	10	14	17

14	17

Binary Search Analysis

Binary search recurrence:

$$T(n) \leq c + T\left(\left[\frac{n}{2}\right]\right)$$

we will always ignore floor/ceiling

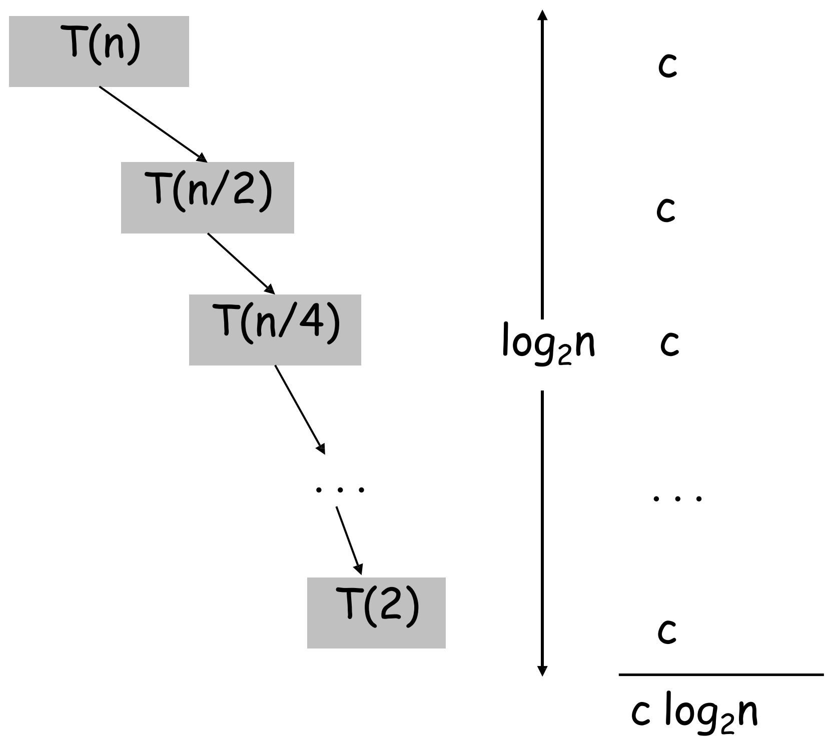
(the “sorting” slides has one slide that keeps the ceiling, so you can see that it works)

Binary Search Analysis

Binary search recurrence: $T(n) \leq c + T\left(\left[\frac{n}{2}\right]\right)$

Claim: The time complexity T(n) of binary search is at most c*log n

Proof 1: $T(n) \quad \leq c + T(n/2) \quad \leq c + c + T(n/4) \leq \underbrace{\quad \ldots\ldots \quad}_{\log n \text{ terms}} \quad \leq c + c + \ldots + c$



Binary Search Analysis

Binary search recurrence:

Claim: The time complexity T(n) of binary search is at most c*log n

$$T(n) \leq c + T\left(\left[\frac{n}{2}\right]\right)$$

Proof 2: (induction) Base case: n=1

Now suppose that for n' ≤ n – 1, $T(n') \leq c * \log(n')$

Then T(n) ≤ c + T(n/2) ≤ c + c*log(n/2) = c + c*(log n – 1) = c*log n

Recursive Algorithms

Exercício 2. Projete um algoritmo (recursivo) que recebe como entrada um número real x e um inteiro positivo n e devolva x^n. O algoritmo deve executar $O(\log n)$ somas e multiplicações

Recursive Algorithms

```
Proc Pot(x,n)
    if n=0 return 1
    if n=1 return x
    if n é par
        tmp←Pot(x,n/2)
        return tmp*tmp
    else n é ímpar
        tmp←Pot(x,(n-1)/2)
        return x*tmp*tmp
    Fim_if
Fim_proc
```

Análise:

$T(n) = c + T(n/2) \Rightarrow T(n)$ é $O(\log n)$