Metodologies de la Programació

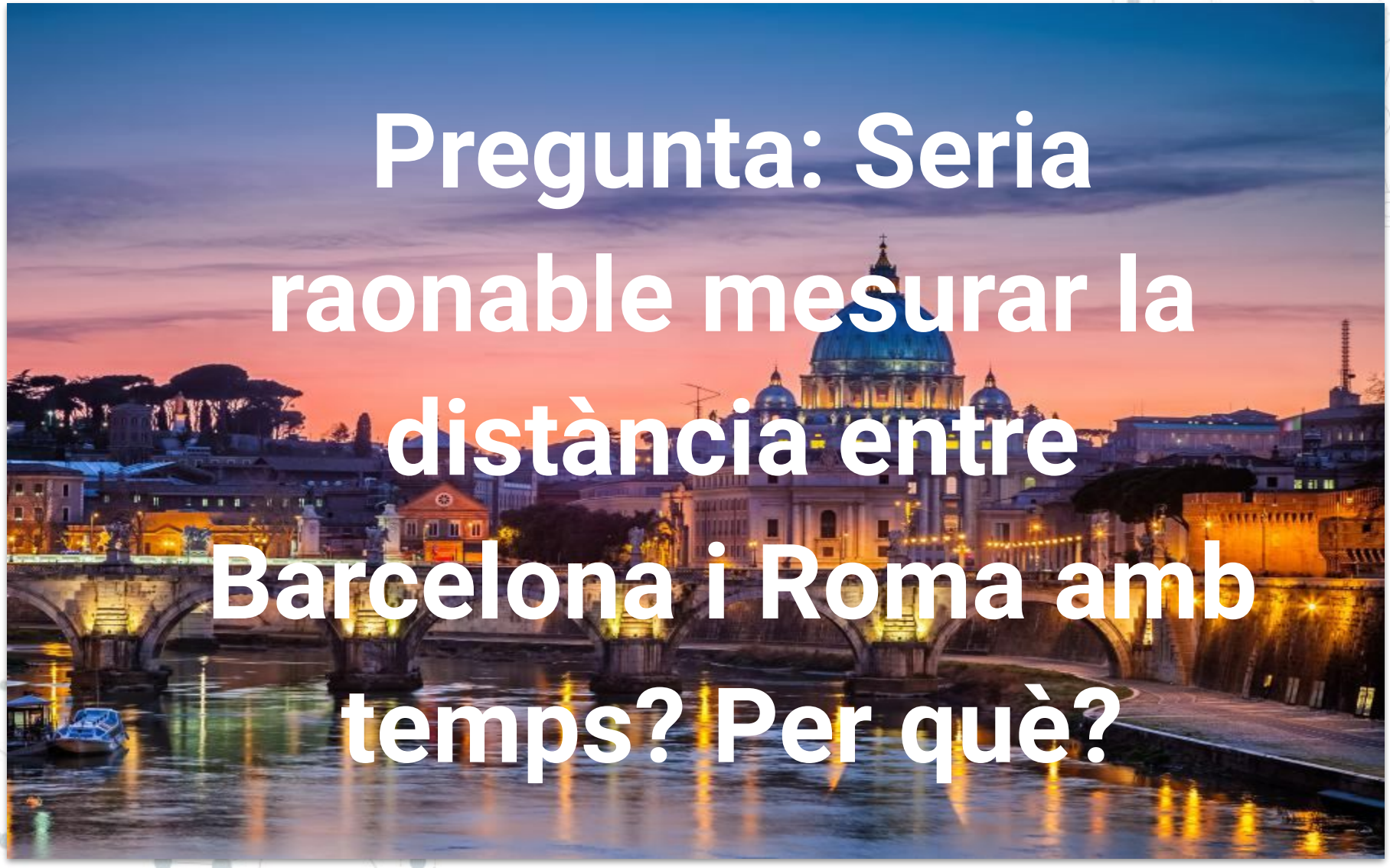
Repàs de Cost algorítmic

Laboratoris Simeó Reig MP 2021-2022

Què és la complexitat algorítimica

Representa la quantitat de recursos (temporals) que necessita un algoritme per resoldre un problema

Què és la complexitat algorítimica

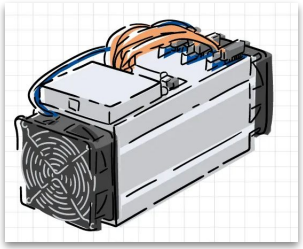


Depén del mitjà de transport

Depèn del conductor

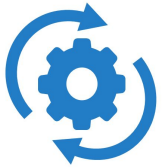
Depèn de la congestió

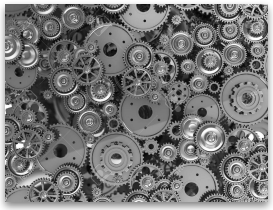
El temps NO és doncs una bona mesura



Transport/hardware

- Int vs integer (2x)
- JIT compiler (java -Xint nomJar)
- Estructura de dades
- Quan seria vàlid emprar una mesura de temps?



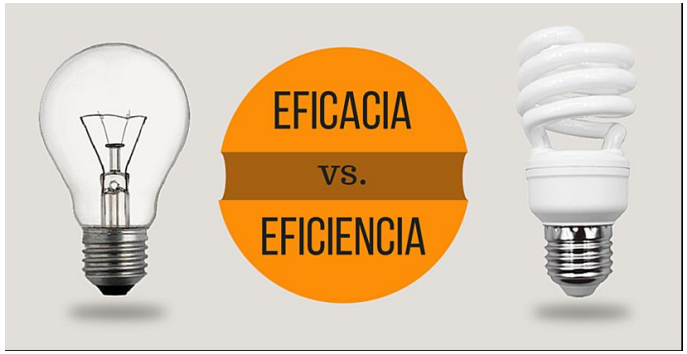


Congestió / Processos

conductor / llenguatge

Com calcular l'eficiència d'un algoritme

- Independentment de la velocitat del processador
- Independentment del llenguatge de programació
- Independentment de com s'accedeixen les dades
- Independentment de les optimitzacions



Big O: Creixement assimptòtic

Ens indica com creix els **passos** d'execució a mesura que creix la mida del conjunt de dades a tractar

Big O: Millor cas, pitjor cas i mitja

- Big (O), pitjor cas
- Omega, millor cas
- Theta, mitja

Exemple: Buscar un element dins un vector NO ordenat de 100 element

Big O: Complexitat

- Temporal
- Espaial
- Energètica

Només treballarem el cost temporal

Big O: Com calculem la complexitat

- Eliminem temps constants
- Valor més rellevant
- Ens interessa classificar

Un cost de $O(n^2 + n + 5) \sim O(n^2)$

Constant, logarítmic, linial, logarítmic, polinòmic, exponencial o factorial

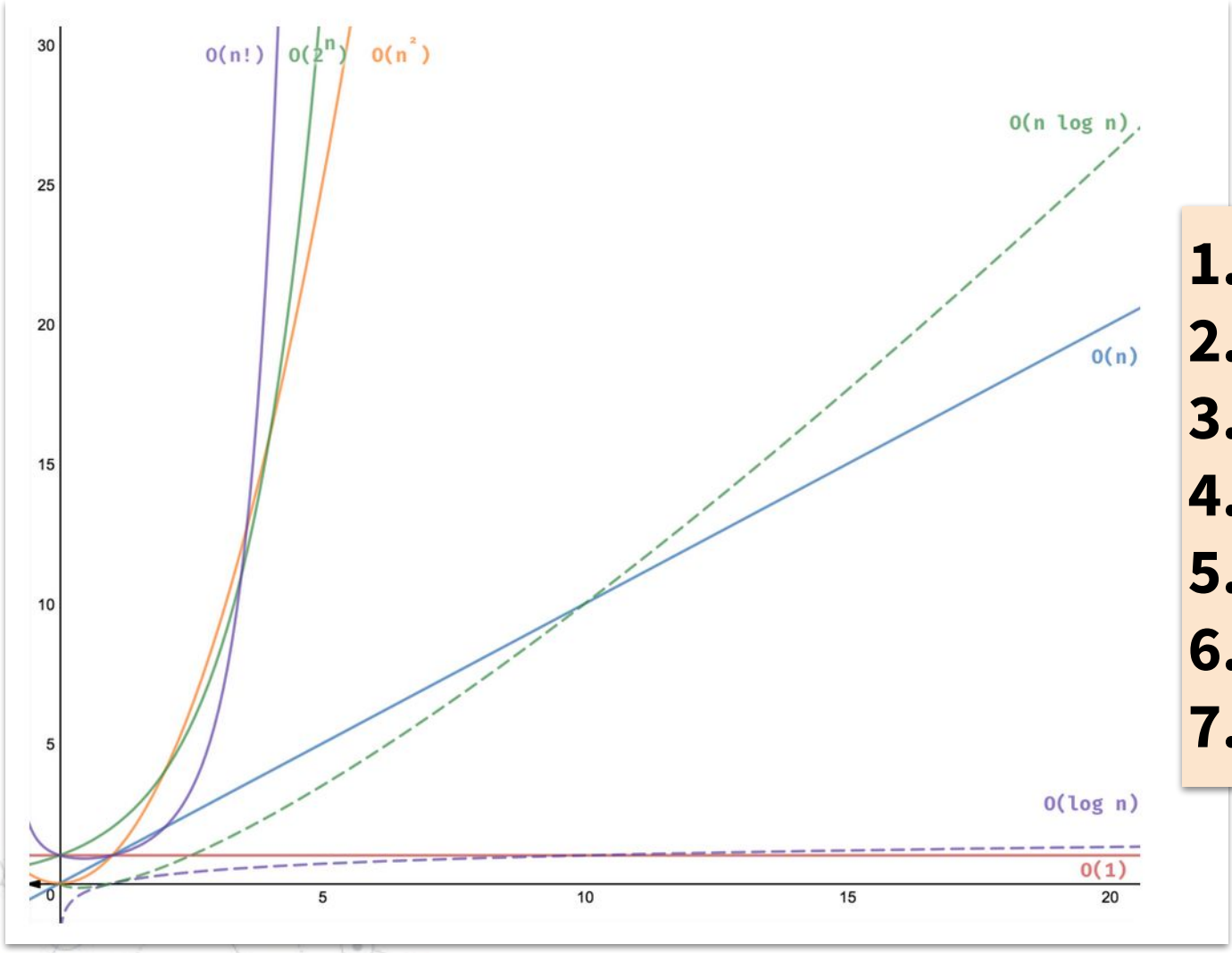
n^2+n+5: Valor més rellevant, K fora

N	N^2	n	5	Total	% variació
1	1	1	5	7	100.0000
2	4	2	5	11	50.0000
3	9	3	5	17	33.3333
4	16	4	5	25	25.0000
5	25	5	5	35	20.0000
6	36	6	5	47	16.6667
7	49	7	5	61	14.2857
8	64	8	5	77	12.5000
9	81	9	5	95	11.1111
10	100	10	5	115	10.0000
1,000	1,000,000	1,000	5	1,001,005	0.1000
10,000	100,000,000	10,000	5	100,010,005	0.0100
100,000	10,000,000,000	100,000	5	10,000,100,005	0.0010

A mida que n es fa gran n i 5 són irrellevants per calcular el cost total

Evidentment a nivell pràctic NO és el mateix un $4*n^2$ que un $n^2/4$

Complexitat: Classificació



1. Constant
2. Logarítmic
3. Linial
4. logLineal
5. Polinòmic
6. Exponencial
7. factorial

O (1)

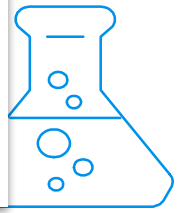
Cost constant

Cost constant: Punts clau

- El cost no varia amb n
- No vol dir que sigui petit! Pot ser enorme!

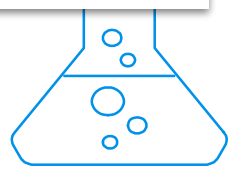
Cost constant: Exemple 1

```
int funcio (int n) {
    int i, result = 0;
    for (i=0; i<10000; i++) {
        result = result + i;
    }
    return result;
}
```



Cost constant: Exemple 2

```
int instalaOrdinadors(int numOrdinadors) {

    //Descarregar 10GB internet, procés lent
    imatgeWin10 = descarregaImatgeInternet(win10);

    //Procés en local, molt més ràpid
    for (int i=1; i<=numOrdinadors; i++){
        carregaImatgeAOrdinador(i,imatgeWin10);
    }

}
```

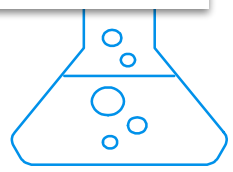


Cost constant: Exemple 2

```
int instalaOrdinadors(int numOrdinadors) {

    //Descarregar 10GB internet, procés lent
    imatgeWin10 = descarregaImatgeInternet(win10);

    //Procés en local, molt més ràpid
    for (int i=1; i<=numOrdinadors; i++){
        carregaImatgeAOrdinador(i,imatgeWin10);
    }

}
```

O(1)

O(n)



Cost constant: Exemple 3

```
int funcio (int vector[n]) {

    int result = 0;
    result = vector[0] * vector[n];
    return result;
}
```

Mutiplicar el primer i l'ultim element d'un vector té un cost constant (igual) independentment de la mida del vector. Perquè per accedir a un vector el compilador multiplica la mida de l'element per n

O (logn)

Cost logarítmic

Cost logarítmic: Punts clau

DIVIDE / CONQUER

- És la inversa del exp
- Un log és una divisió repetida
- L'algoritme va dividint el problema (D&C)

Cost logarítmic: Què és el log?

- Quin és el log2(16)?

1. 16/ 2 = 8
2. 8/2 = 4
3. 4/ 2 = 2
4. 2/ 2 =1

Hem fet 4 divisions, per tant el resultat es 4

A nivell de costos és igual la base del logaritme

Cost logarítmic: Dividint el problema

```
int functionDivide (int n) {

      while (n > 1){
            printf("N:%d \n",n);
            n = n / 2;
      }

      printf("N:%d \n",n);
}
```

Si dividim per dos és el log2, si dividim per 3 seria el log3 etc.

Cost logarítmic: Dividint el problema

```
int recursive (float n) {

      if (n < 1) return;

      printf("N:%.2f \n",n);

      recursive(n/2);
}
```

If (n<1) return és el cas base

```
int main()
{
      recursive (100);
      return 0;
}
```

Cost logarítmic: Execucions algoritme

```
N:100.00
N:50.00
N:25.00
N:12.50
N:6.25
N:3.12
N:1.56
```

```
N:1000.00
N:500.00
N:250.00
N:125.00
N:62.50
N:31.25
N:15.62
N:7.81
N:3.91
N:1.95
```

```
N:100000.00
N:50000.00
N:25000.00
N:12500.00
N:6250.00
N:3125.00
N:1562.50
N:781.25
N:390.62
N:195.31
N:97.66
N:48.83
N:24.41
N:12.21
N:6.10
N:3.05
N:1.53
```

```
N:1000000.00
N:500000.00
N:250000.00
N:125000.00
N:62500.00
N:31250.00
N:15625.00
N:7812.50
N:3906.25
N:1953.12
N:976.56
N:488.28
N:244.14
N:122.07
N:61.04
N:30.52
N:15.26
N:7.63
N:3.81
N:1.91
```

Log2(100) = 6,64
Log2(1K) = 9,96
Log2(100K) = 16,60
Log2(1M) = 19,93

Cost logarítmic: Àtoms de l'univers

Si posem tot els àtoms de l'univers en un vector ordenat, amb 266 iteracions de l'algoritme de la cerca dicotòmica (cost logarítmic) trobaríem l'àtom buscat

Diferència entre lineal i logarítmic

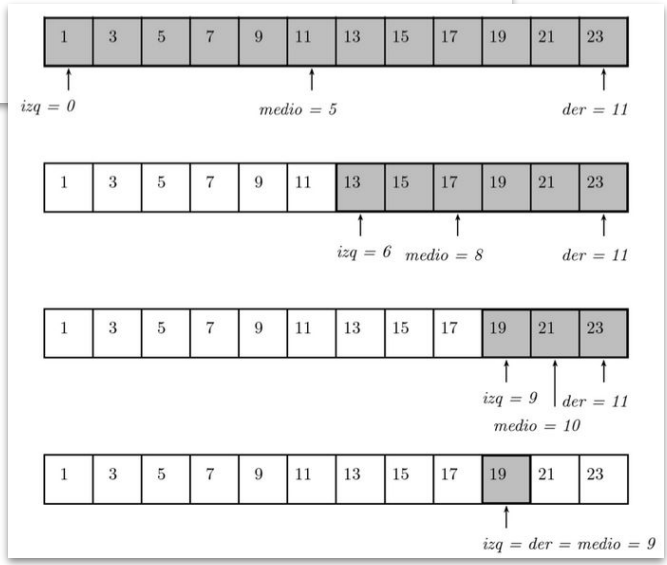
(10^80÷(10^10×86400×365×13.700.000)) = $2{,}32\times10^{55}$

Fent 10mil milions de búsquedes per segon necessitem $2{,}32\times10^{55}$ temps de l'univers per trobar l'àtom desitjat

Cost logarítmic: Cerca dicotòmica

```
int busquedaBinaria(int  vector[], int n, int dada) {

(1)     int centre,inf=0,sup=n-1;

(2)     while(inf<=sup){
(3)        centre=((sup-inf)/2)+inf;
(4)        if(vector[centre]==dada)          return centre;
(5)        else if(dada < vector[centre]) sup=centre-1;
(6)        else                           inf=centre+1;
        }
(7)     return -1;
}
```

Serà complicat, per no dir impossible trobar un algoritme de cerca dicotòmica que faci més de 266 iteracions al bucle



Binary Search: Exemple de cost log(n)

Número de comparacions: $\log^2(n)+1$

Mida V[]	Comparacions
10	3,2
100	6,6
1.000	9,96
10K	13,2
100K	16,6

Mida V[]	Comparacions
1M	19,93
100M	26,57
1.000M	29,89
100B	39,86
10.000B (15)	53,15

Quin seria el cost si féssim una cerca lineal?

Cost pitjor cas : O(n) ← No existeix l'element

Cost millor cas: Ω (n) ← Busco el primer casualment

Cost mig : T (n) ← Existeix n/2 comparacions

$O(n)$

Cost lineal

Cost lineal: Punts clau

- El cost creix linealment
- Exemples n, 2n, 3n, 10n ..
- És el més intuïtiu
  - Consum benzina
  - Quan tardaràs
  - Etcètera

Cost lineal: Suma valors

```
int sumaValors(int n) {

    int i, result = 0;

    for (i=0; i<n; i++) {
        result = result + i;
    }

    return result;
}
```

Cost lineal: CountDown recursiu

```
int countDown (int n) {
        if (n < 1) return;
        printf("N:%d \n",n);
        countDown(n-1);
}
```

Cost lineal: Accés a dos elements de la llista

```
int funcio (int linkedList[n]) {

    int result = 0;
    result = linkedList[0] * linkedList[n];
    return result;
}
```

Error típic, no cal tenir un bucle per a ser n! , aquí no tenim un loop i té un cost O(n)

O(n*log(n))

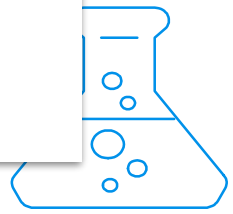
Cost loglineal

Cost loglineal: Punts claus

Algoritme amb un bucle O(logn) dins un O(n) o a l'inrevés

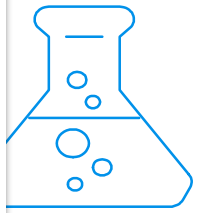
Cost loglineal: Exemple 1

```
int functionNlogn (int n) {

    int y;

    while ( n > 0 ) { // O(n)
        y = n
        while ( y > 0 ) { // O(logn)
            y = y / 2
        }
        n--;
    }
}
```

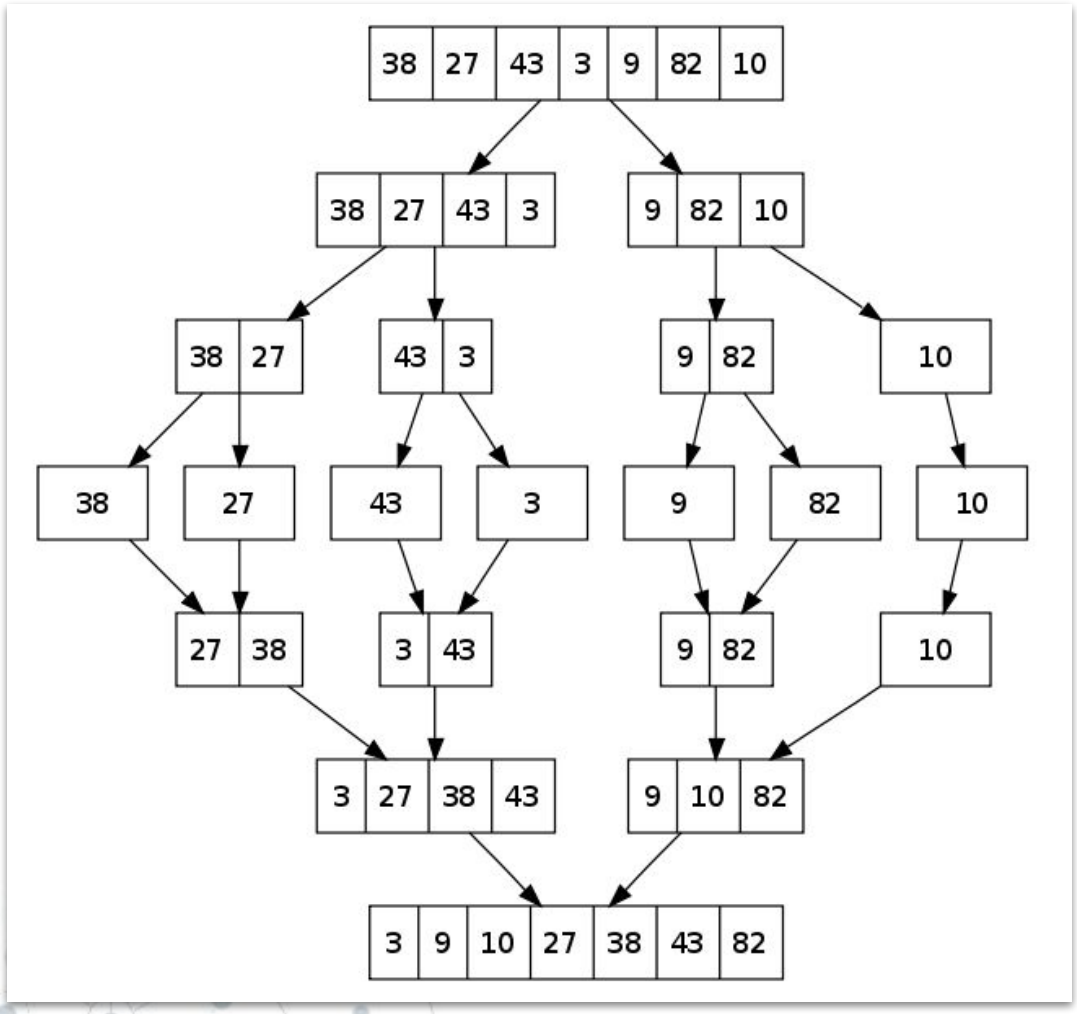


Cost loglineal: Exemple 2

```
int functionNlogn2 (int n) {

    int y;

    while ( n > 0 ) { // O(logn)
        y = n
        while ( y > 0 ) { // O(n)
            y--:
        }
        n = n / 2;
    }
}
```



Cost loglineal: Merge Sort



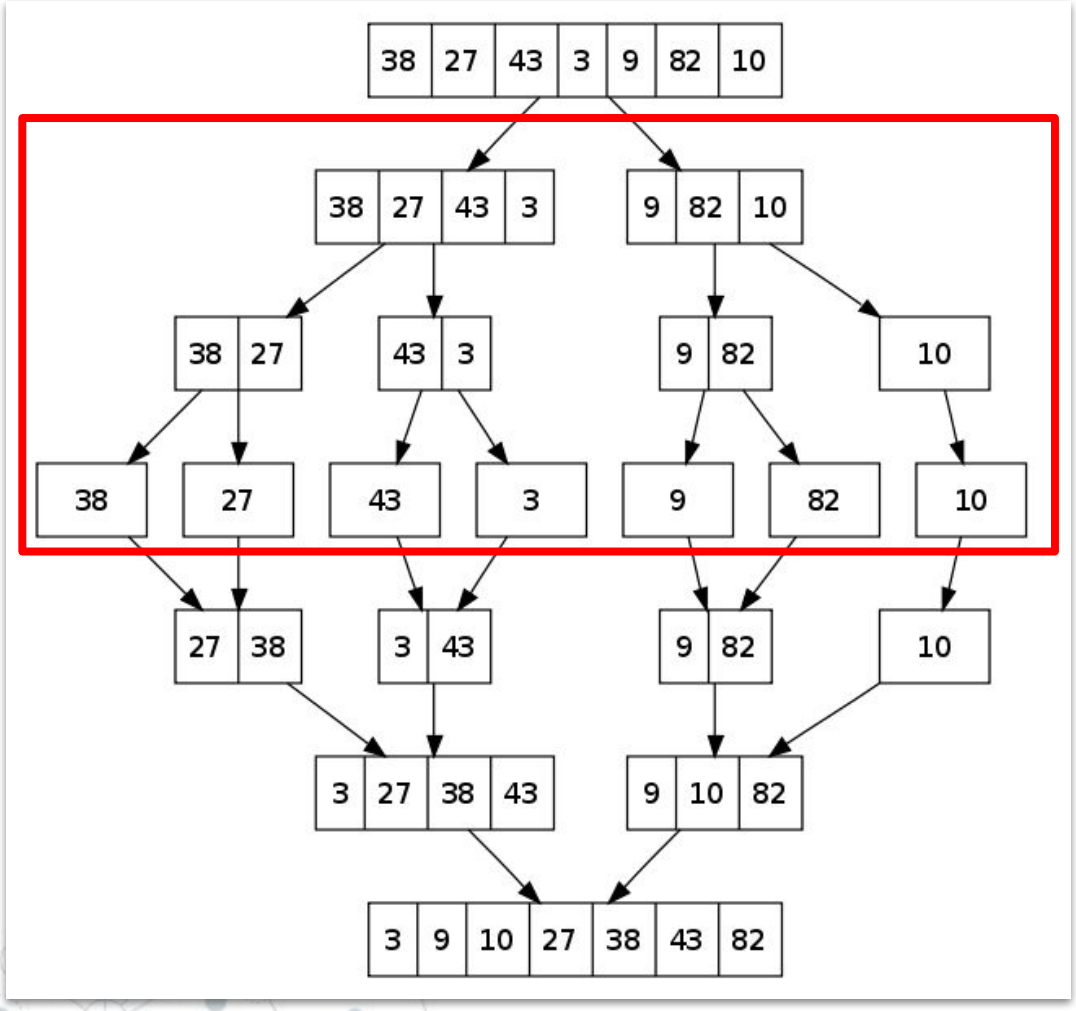
MergeSort

Divide/Split_Step: Partim pel punt mig el vector O(1)

Conquer Step: Fem la crida recursiva al merge Sort

Combine/merge Step: Ordenem els vectors

Cost loglineal: Merge Sort,

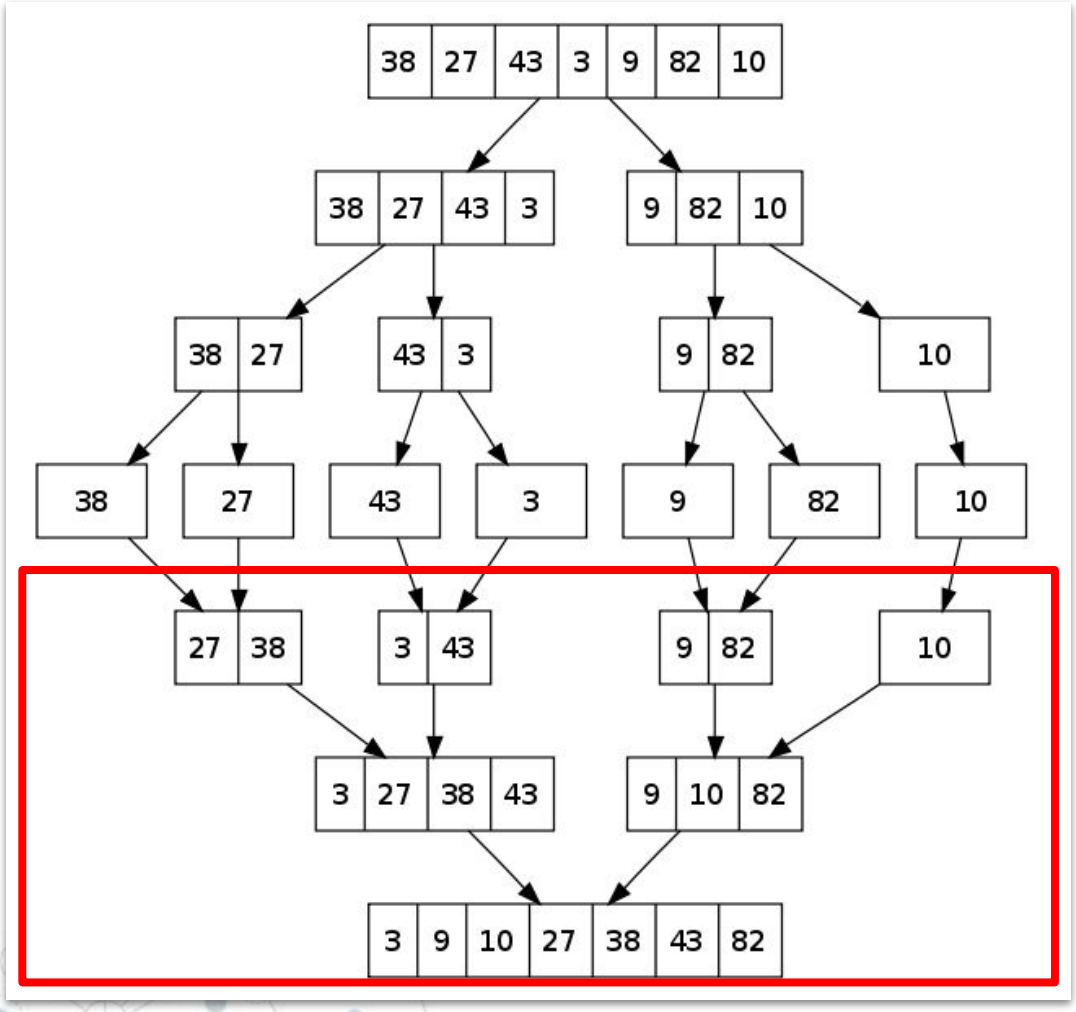


Anem dividint recursivament el vector. **O(logn)**

Floor(log(7)) + 1 = 3 nivells

```
void sort(int arr[], int l, int r)
{
    if (l < r) {
        // Find the middle point
        int m =l+ (r-l)/2;

        // Sort first and second halves
        sort(arr, l, m);
        sort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
```

Cost loglineal: Merge Sort,



Ordenem els vectors de log(n) nivells amb un cost de **O(n)**

Cost = O(log(n)*n)

```
void merge(int arr[], int l, int m, int r)
{
    // Find sizes of two subarrays to be merged
    int n1 = m - l + 1; int n2 = r - m;

    /* Create temp arrays */
    int L[] = new int[n1]; int R[] = new int[n2];

    /*Copy data to temp arrays*/
    for (int i = 0; i < n1; ++i) L[i] = arr[l + i];
    for (int j = 0; j < n2; ++j) R[j] = arr[m + 1 + j];

    /* Merge the temp arrays */
    // Initial indexes of first and second subarrays
    int i = 0, j = 0; int k = l;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];}

    /* Copy remaining elements of L[],R[] if any */
    while (i < n1) { arr[k++] = L[i++]};
    while (j < n2) { arr[k++] = R[j++]};
}
```

$O(n^c)$

Cost polinòmic

Cost polinòmic O(n^c): Punts clau

- N es la mida
- C és una constant
- $O(n^2)$ quadràtic, $O(n^3)$ cúbic etcètera

Cost polinòmic : Exemple n² i n^3

```
for (int i = 0; i < llongitudArray ; i++) {
    for (int j = 0; j < llongitudArray - 1; j++) {
            if (v[j]>v[j+1]) {
                swap (v,j,j+1);
            }
    }
}
```

Fixem-nos que tots els for depenen del valor de l'entrada (n)

```
void funcioCubica (int n) {

    int i,j,k;

    for (i=0; i<n ;i++){
        for (j=0; j<n ;j++){
            for (k=0; k<n ;k++){
                printf("i:%d,j:%d,k:%d",i,j,k)
            }
        }
    }

}
```

Cost polinòmic : Exemple sense loop

```
int funcioMultiplicaV(int vector[n]) {

      int i;

      for (i=0; i<n; i++) {
          vector[i] = vector[i] * 2;
          ordenaBombolla(vector);
      }

 }
```

Cost polinòmic : Exemple sense loop

```
int funcioMultiplicaV(int vector[n]) {

      int i;

      for (i=0; i<n; i++) {
          vector[i] = vector[i] * 2;
          ordenaBombolla(vector);
      }

 }
```

Tenim un sol loop i té cost $O(n^3)$

Cost polinòmic : Exemple sense loop

```
int funcioMultiplicaV(int vector[n]) {

    int i;

    for (i=0; i<n; i++) {
        vector[i] = vector[i] * 2;
        ordenaBombolla(vector);
    }

}
```

O(n)
O(1)
O(n^2)

Tenim un sol loop i té cost O(n^3)

O(c^n)

Cost exponencial

Cost exponencial O(c^n): Punts clau

- N es la mida
- C és una constant
- O(2^n),O(3^n)

Un exponent és una multiplicació.

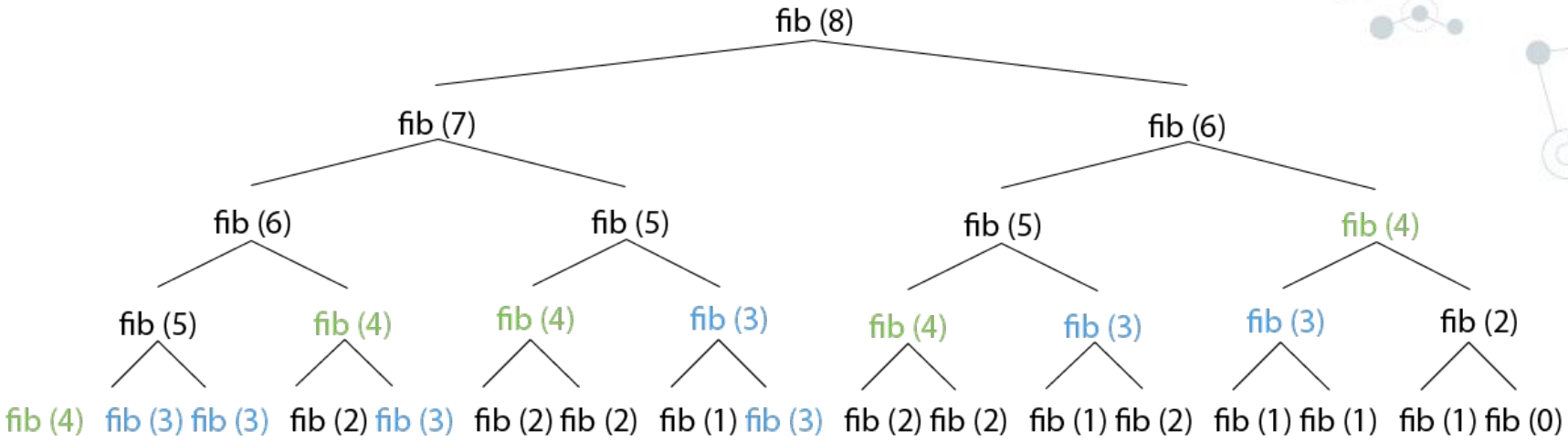
Heu sentit mai la frase “Els problemes es multipliquen?”

Cost exponencial: Fibonacci recursiu

```
int fibonacci(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

El fibonacci recursiu és O(2^n)

Cost exponencial: Crides fibonacci



Cada crida genera 2 crides: O(2^n)

Cost exponencial: TRIBonacci recursiu

```
int TRIbonacci(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1)+fib(n-2)+fib(n-3);
}
```

Quin cost assimptòtic tindria el tribonacci? ©

Cost exponencial: Passwords

5 caracteres (3 letras minúsculas, 2 números)	36^5=60.466.176	60.466.176 / 2.000.000.000 = **0,03 segundos**
7 caracteres (1 mayúscula, 6 minúsculas)	52^7= 1.028.071.702.528	1.028.071.702.528 / 2.000.000.000 = 514 Segundos = **aprox. 9 minutos**
8 caracteres (4 letras minúsculas, 2 caracteres especiales, 2 números)	68^8= 457.163.239.653.376	457.163.239.653.376 / 2.000.000.000 = 228.581 Segundos = **aprox. 2,6 días**
9 caracteres (2 mayúsculas, 3 minúsculas, 2 números, 2 caracteres especiales)	94^9= 572.994.802.228.616.704	572.994.802.228.616.704 / 2.000.000.000 = 286.497.401 Segundos = **aprox. 9,1 años**
12 caracteres (3 mayúsculas, 4 minúsculas, 3 caracteres especiales, 2 números)	94^{12}= 475.920.314.814.253.376.475.136	475.920.314.814.253.376.475.136 / 2.000.000.000 = 237.960.157.407.127 Segundos = **aprox. 7,5 millones de años**

36=26a+10d
52=26A+26a
68=26A+26a+10d+6e
94=26A+26a+10d+32e

Fixeu-vos el "poder" de l'exponent, de fer servir 9 a 12 caracters a la password

O(n!)

Cost factorial

Cost factorial O(n!): Punts clau

- N es la mida
- n!=(n)(n-1)(n-2)(n-3)...(2)(1)
- 4!=4*3*2*1

Cost factorial O(n!): Exemple

```
int exponencial (int n) {

      if (n == 1) return;

      for (int i=0; i<n; i++){
          exponencial(n-1);
      }

}
```

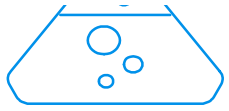
El factorial de 70 (70*69*68...1) és més gran que el número d'àtoms a l'univers

n	n!
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5 040
8	40 320
9	362 880
10	3 628 800
11	39 916 800
12	479 001 600
13	6 227 020 800
14	87 178 291 200
15	1 307 674 368 000
16	20 922 789 888 000
17	355 687 428 096 000
18	6 402 373 705 728 000
19	121 645 100 408 832 000
20	2 432 902 008 176 640 000
25	$1{,}551\,121\,004 \times 10^{25}$
50	$3{,}041\,409\,320 \times 10^{64}$
70	$1{,}197\,857\,167 \times 10^{100}$
100	$9{,}332\,621\,544 \times 10^{157}$

Algoritmes

Cost exponencial: taula de costos

mida	Operacions segons la complexitat					
	$\log_2 n$	$n\log_2 n$	n	n^2	n^3	2^n
10	3	33	10	100	1000	1024
100	7	664	100	10000	1000000	1,26E+30
1.000	10	9966	1000	1000000	1000000000	1,07E+301
10.000	13	132877	10000	100000000	1E+12	OMG
100.000	17	1660964	100000	10000000000	1E+15	OMG
1.000.000	20	19931569	1000000	1E+12	1E+18	OMG



Altres algoritmes d'ordenació : Cost

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Estructures de dades: Cost

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

B-Tree inventats al 1972 per Boeing. Rudolf Bayer & Ed McCreight

Exercicis

Ex 1: Plantejament problema

```
int funcio (int n) {
    int i,j = 0;
    while( i < n) {

        j = 0;
        while (j<3*n) {
            j++;}

        j = 0;
        while (j<2*n) {
            j++;}

        i++;
    }
}
```

Ex 1: Solució

```
int funcio (int n) {
    int i,j = 0;
    while( i < n) {

        j = 0;
        while (j<3*n) {
            j++;}

        j = 0;
        while (j<2*n) {
            j++;}

        i++;
    }
}
```

$f(n)=n*(3n+2n)$
$= 3n^2 + 2n^2 =$
$5n^2 = O(n^2)$

Ex 2: Plantejament problema

```
if x > 0
    // O(1)
else if x < 0
    // O(logn)
else
    // O(n²)
```

Ex 2: Solució

```
if x > 0
    // O(1)
else if x < 0
    // O(logn)
else
    // O(n²)
```

$f(n)=O(n^2)$

Ex 3: Plantejament problema

```
int funcio(int n)
{
  while i < 3*n do
       j:=10;
       while j <= 50 do
            j++;

       j:=0;
       while j < n*n*n do
            j = j + 2;

       i++;
}
```

Ex 3: Solució

```
int funcio(int n)
{
  while i < 3*n do
        j:=10;
        while j <= 50 do
            j++;

        j:=0;
        while j < n*n*n do
            j = j + 2;

        i++;
}
```

$f(n)=3n*(c+n^3/2) = 3n + 3/2n^4 = O(n^4)$

Ex 4: Plantejament problema

```
int funcio(int n)
{
   i := n
   while i > 0 do
        j:= 1;
        while j < n do
             j++;

        i = i / 2;
}
```

Ex 4: Solució problema

```
int funcio(int n)
{
   i := n
   while i > 0 do
        j:= 1;
        while j < n do
             j++;

        i = i / 2;
}
```

$f(n)=\log(n)*n$

Ex 5: Plantejament problema

```
int funcio(int n)
{
   i := 0
   while i < n do
        j:= 1;
        while j < n do
             j = j*2;

        i++;
}
```

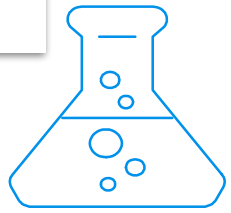
Ex 5: Solució problema

```
int funcio(int n)
{
   i := 0
   while i < n do
        j:= 1;
        while j < n do
             j = j*2;

        i++;
}
```

f(n)=n*log(n)

Ex 6: Plantejament problema

```
int factorial(int n)
{
    if (n<=1) {
        return 1;
    } else {
        return n * factorial(n-1);
    }

}
```



Ex 6: Solució problema

```
int factorial(int n)
{
    if (n<=1) {
        return 1;
    } else {
        return n * factorial(n-1);
    }

}
```

f(n)=n

Ex 7: Plantejament problema

```
int iteracions(int n)
{
    for (int i = 1; i <= pow(2, n); i++){
        printf("iteraccio " + i);
    }
}
```

Ex 7: Solució problema

```
int iteracions(int n)
{
    for (int i = 1; i <= pow(2, n); i++){
        printf("iteraccio " + i);
    }
}
```

f(n)= 2^n

exponencial

- 2^10 = 1024
- 2^128=3.4^38
- 2^266 = univers


Fins al
pròxim
laboratori!