Análisis y diseño de software

dit
UPM

Tema 1: Algoritmos /complejidad /java

José A. Mañas
http://jungla.dit.upm.es/~pepe/doc/adsw/index.html

21.2.2018

ETSIT UPM

referencias

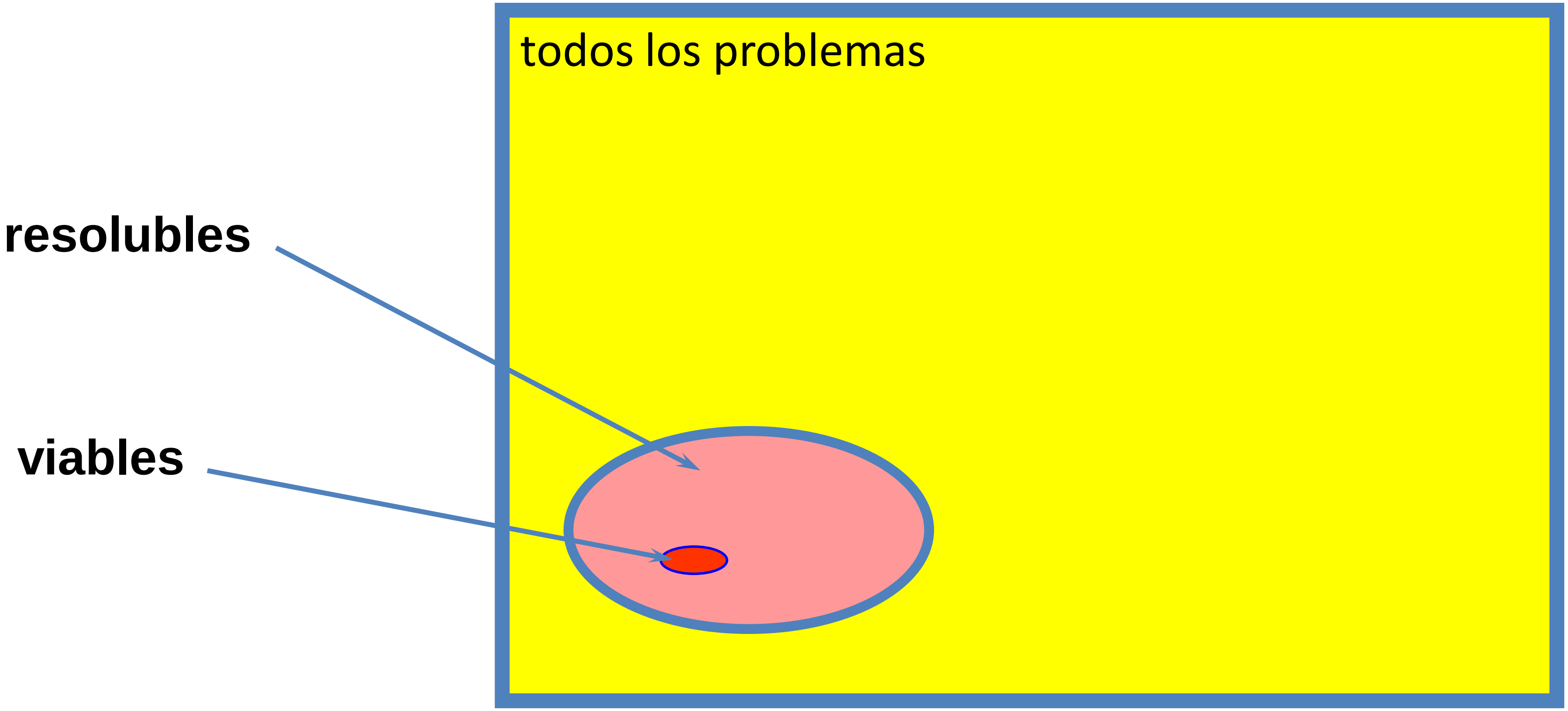
- http://www.dit.upm.es/~pepe/doc/adsw/tema1/Complejidad.pdf

motivación

- no todos los algoritmos son iguales
- unos requieren más recursos que otros
  - **más tiempo**
  - más memoria
  - ➢ un mal algoritmo no tiene remedio
- no todos los programas son iguales
  - hay mejores programadores
  - hay mejores compiladores
  - ➢ un mal programa se puede arreglar

problemas y algoritmos

- No se conocen algoritmos para todos los problemas
- No todos los algoritmos son viables



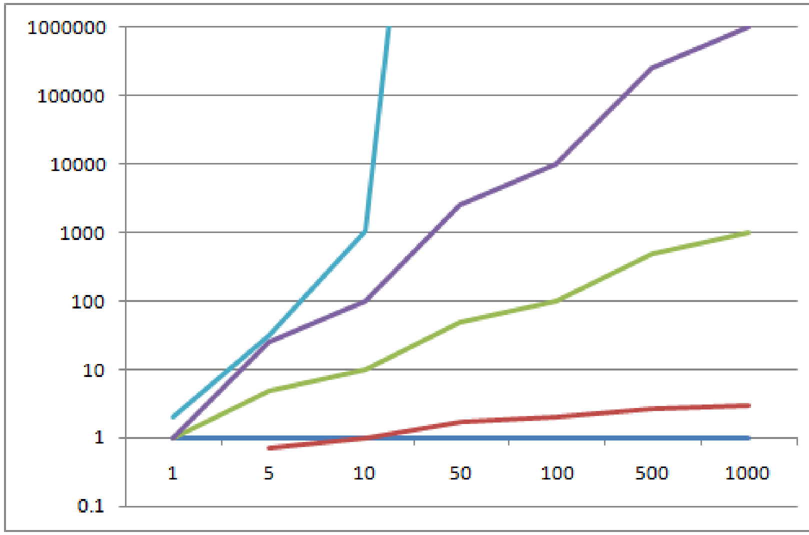
T(n)

- se mide cómo crece el tiempo t de ejecución en función del tamaño n del problema
- se analiza la función

  $t = T(n)$

- obviando los casos pequeños y estudiando la tendencia

  $\lim_{n \to \infty} T(n)$



complejidad

- Una relación de orden total entre funciones de consumo de recursos
- Calculamos

$$\lim_{n \to \infty} \left(\frac{f(n)}{g(n)} \right) = K$$

  - K = ∞, decimos que f(n) > g(n) mayor complejidad
  - K = 0, decimos que f(n) < g(n) menor complejidad
  - K ≠ 0 y K ≠ ∞, decimos que f(n) = g(n), misma complejidad

funciones de referencia

función	nombre
$f(n) = 1$	constante
$f(n) = \log(n)$	logaritmo
$f(n) = n$	lineal
$f(n) = n \times \log(n)$	
$f(n) = n^2$	cuadrática
$f(n) = n^a$	polinomio (de grado $a > 2$)
$f(n) = a^n$	exponencial ($a > 1$)
$f(n) = n!$	factorial

orden de complejidad

- se definen conjunto de funciones de complejidad igual o menor

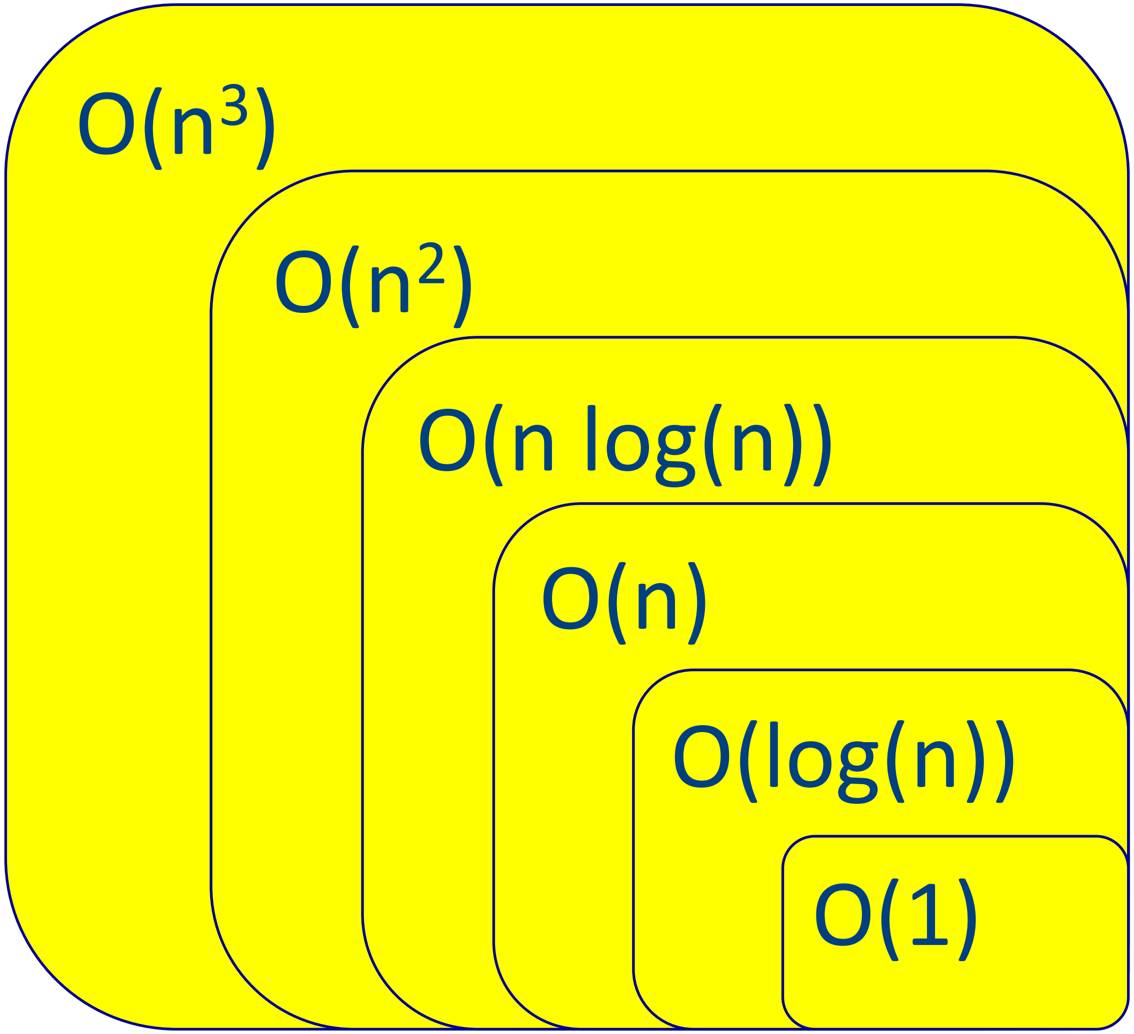
  O(f(n)) = { g(n), $\lim_{n \to \infty} \left(\frac{g(n)}{f(n)} \right) < \infty$ }

  – el conjunto incluye las funciones g(n) que son igual de complejas que la de referencia, f(n) y las que son menos complejas que f(n)

  – este truco nos permite no tener que calcular g(n) exactamente, bastando una cota superior

  – nunca será más complejo que una f(n) dada

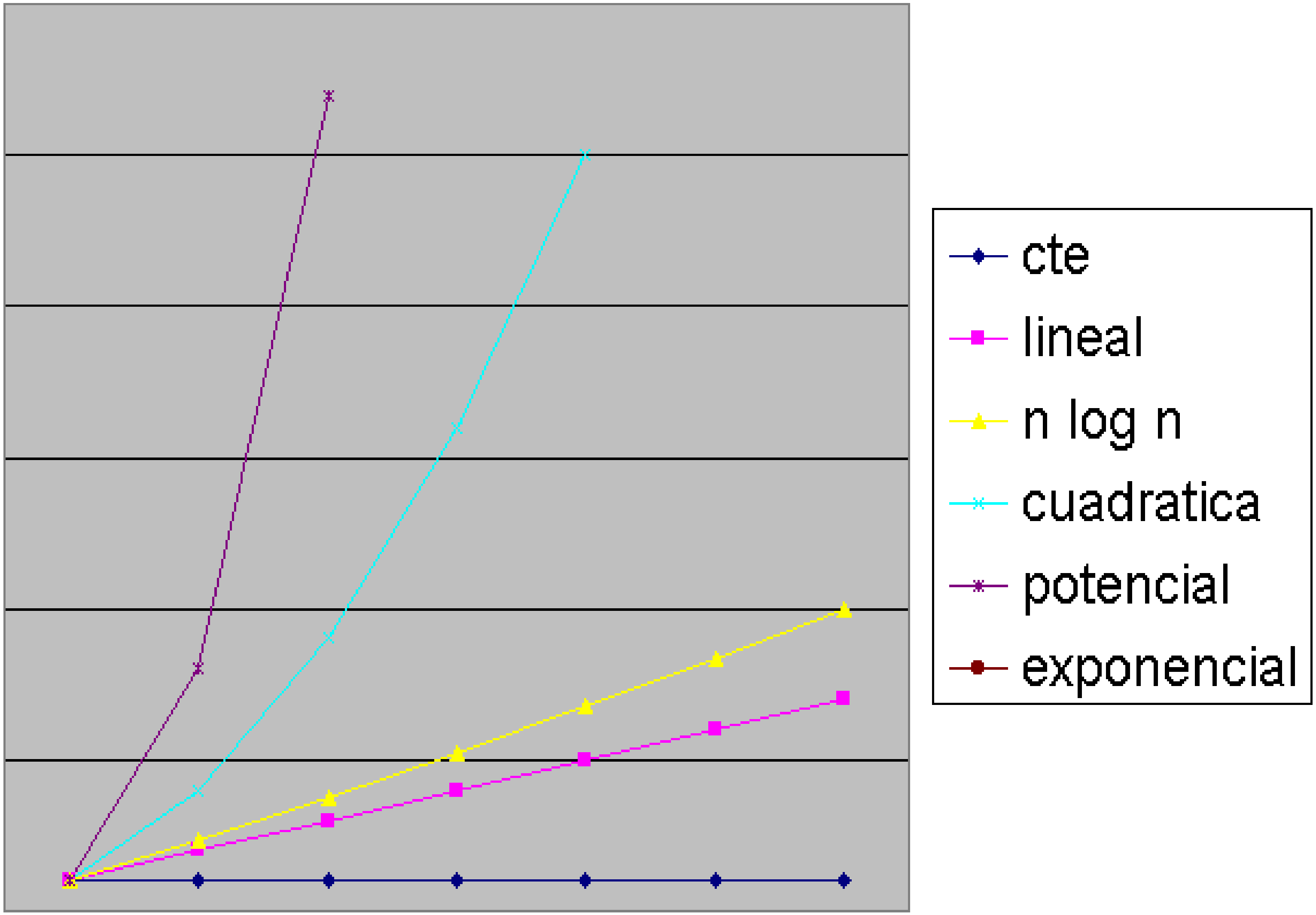
órdenes habituales



órdenes habituales

orden	nombre	comentario
O(1)	constante	ideal
O(log(n))	logarítmico	una maravilla
O(n)	lineal	lo normal
O(n log(n))		está bien
$O(n^2)$		tratable
$O(n^a)$	polinomial (a > 2)	"tratable"
$O(a^n)$	exponencial (a > 1)	no es práctico
O(n!)	factorial	inviable

impacto comparado



impacto comparado

n	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(n^5)	O(5^n)	O(n!)
10	1	2	10	23	100	1e+05	1e+07	4e+06
20	1	3	20	60	400	3e+06	1e+14	2e+18
30	1	3	30	102	900	2e+07	9e+20	3e+32
40	1	4	40	148	1,600	1e+08	9e+27	8e+47
50	1	4	50	196	2,500	3e+08	9e+34	3e+64
60	1	4	60	246	3,600	8e+08	9e+41	8e+81
70	1	4	70	297	4,900	2e+09	8e+48	1e+100
80	1	4	80	351	6,400	3e+09	8e+55	7e+118
90	1	4	90	405	8,100	6e+09	8e+62	1e+138
100	1	5	100	461	10,000	1e+10	8e+69	9e+157

impacto comparado

n	O(1)	O(lg n)	O(n)	O(n lg n)	O(n^2)	O(n^5)	O(5^n)	O(n!)
10	1µs	2µs	10µs	23µs	100µs	100ms	10s	4s
20	1µs	3µs	20µs	60µs	400µs	3s	3a	63 mil años
30	1µs	3µs	30µs	102µs	900µs	20s	28 millones de años	
40	1µs	4µs	40µs	148µs	1,6ms	100s		
50	1µs	4µs	50µs	196µs	2,5ms	300s		
60	1µs	4µs	60µs	246µs	3,6ms	800s		
70	1µs	4µs	70µs	297µs	4,9ms	33m		
80	1µs	4µs	80µs	351µs	6,4ms	50m		
90	1µs	4µs	90µs	405µs	8,1ms	1h40m		
100	1µs	5µs	100µs	461µs	10ms	2h46m		

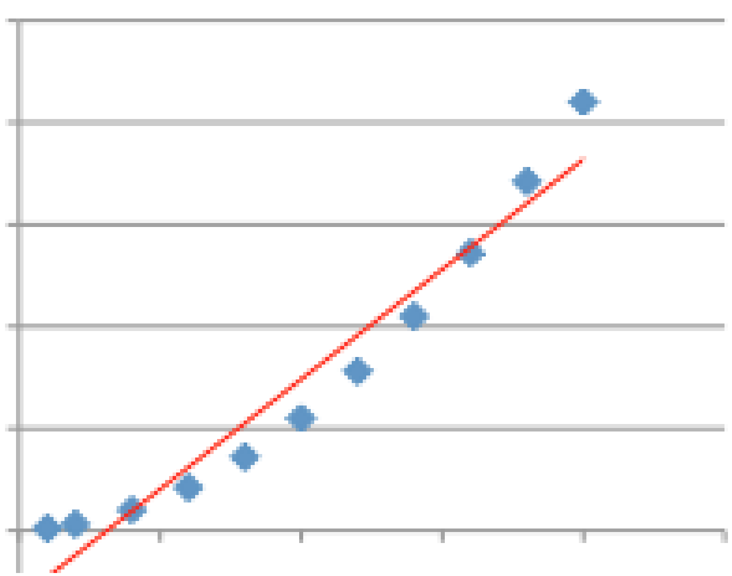
impacto práctico

- Si 100 datos se procesan en 1 hora
  - ¿Cuántos datos se procesan en 2 horas?
  - ¿Cuántas horas lleva procesar 200 datos?

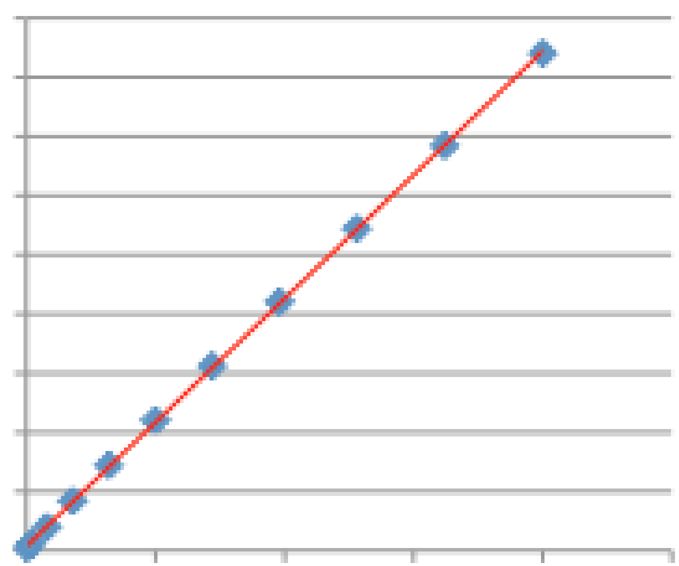
O(f(n))	N = 100	t = 2h	N = 200
log n	1	100.000	1,15
n	1	200	2,00
n log n	1	199	2,30
n^2	1	141	4,00
n^3	1	126	8,00
2^n	1	101	10^{30}

representación gráfica

- con un cambio de variable podemos transformar cualquier orden en una recta
- ojo: los datos medidos no son exactos
- ej: $O(n^2)$



$x = n$



$x = n^2$

reglas de cálculo

C. $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0 \Rightarrow f \in O(g)$

$\Rightarrow g \notin O(f)$

$\Rightarrow O(f) \subset O(g)$

D. $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = K \Rightarrow f \in O(g)$

$\Rightarrow g \in O(f)$

$\Rightarrow O(f) = O(g)$

E. $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \infty \Rightarrow f \notin O(g)$

$\Rightarrow g \in O(f)$

$\Rightarrow O(f) \supset O(g)$

F. Si $f, g \in O(h) \Rightarrow f + g \in O(h)$

G. Sea k una constante, $f(n) \in O(g) \Rightarrow k * f(n) \in O(g)$

H. Si $f \in O(h1)$ y $g \in O(h2) \Rightarrow f + g \in O(h1+h2)$

I. Si $f \in O(h1)$ y $g \in O(h2) \Rightarrow f * g \in O(h1*h2)$

J. Sean los reales $0 < a < b \Rightarrow O(n^a) \subset O(n^b)$

K. Sea P (n) un polinomio de grado k $\Rightarrow P(n) \in O(n^k)$

L. Sean los reales a, b > 1 $\Rightarrow O(\log\_a) = O(\log\_b)$

reglas de cálculo

- sentencias;

  → O(1)

- s; s; …

  → Σ O(s)

- if (x) s1; else S2;

  → O(x) + max(O(s1), O(s2))

reglas de cálculo: bucles

```
for (int i= 0; i < K; i++)
  algo_de_O(1);
```

$\Rightarrow K * O(1) = O(1)$

```
for (int i= 0; i < N; i++)
 for (int j= 0; j < N; j++)
   algo_de_O(1);
```

$\Rightarrow N * N * O(1) = O(n^2)$

```
for (int i= 0; i < N; i++)
 for (int j= 0; j < i; j++)
   algo_de_O(1);
```

$1 + 2 + 3 + ... + N = N * (1+N) / 2 \rightarrow O(n^2)$

reglas de cálculo: bucles

```
int c = 1;
while (c < N) {
   algo_de_O(1);
   c*= 2;
}
```

- El valor inicial de c es 1,
  siendo 2^k al cabo de k iteraciones
- El número de iteraciones es tal que
  - $2^k \geq N \Rightarrow k = \lceil \log_2(N) \rceil$
- $\lceil x \rceil$ es el entero inmediato superior a x

➔ O(log n)

reglas de cálculo: bucles

```
for (int i = 0; i < N; i++) {
    c = i;
    while (c > 0) {
        algo_de_O(1);
        c/= 2;
    }
}
```

- bucle interno: O(log n) que se ejecuta N veces,
- orden del conjunto: O(n log n)

ejercicio

```
int power1(int a, int n) {
    int r = 1;
    for (int i = 0; i < n; i++)
        r *= a;
    return r;
}
```

```
int power2(int a, int n) {
    if (n == 0)
        return 1;
    if (n % 2 == 0)
        return power2(a * a, n / 2);
    else
        return a * power2(a * a, (n - 1) / 2);
}
```

reglas de recurrencia

- $T(n) = c + T(n/2)$
- $T(1) = c$

- $T(n) = c + T(n/2) = c + c + T(n/4) = \dots$
  $= kc + T(n/2^k)$
- $T(n) = kc$, cuando $n/2^k = 1$
  $k = \log(n)$

➔ $T(n) \in O(\log n)$

relaciones de recurrencia

relación	complejidad	ejemplos
$T(n) = T(n/2) + O(1)$	$O(\log n)$	búsqueda binaria
$T(n) = T(n-1) + O(1)$	$O(n)$	búsqueda lineal factorial bucles for, while
$T(n) = 2\,T(n/2) + O(1)$	$O(n)$	recorrido de árboles binarios: preorden, en orden, postorden
$T(n) = 2\,T(n/2) + O(n)$	$O(n \log n)$	ordenación rápida (quick sort)
$T(n) = T(n-1) + O(n)$	$O(n^2)$	ordenación por selección ordenación por burbuja
$T(n) = 2\,T(n-1) + O(1)$	$O(2^n)$	torres de hanoi

ejemplo: números de fibonacci

- F(n) = F(n-1) + F(n-2)
- solución 1: recursiva
- solución 2: iterativa
- solución 3: fórmula

recursiva	iterativa	fórmula
$O(1{,}6^n)$	O(n)	O(1)

Fibonacci 1 - recursiva

```
int fibo(int n) {
    if (n < 2)
        return 1;
    return fibo(n - 1) + fibo(n - 2);
}
```

Fibonacci 2 – iterativa

```
int fibo(int n) {
    int n0 = 1;
    int n1 = 1;
    for (int i = 2; i <= n; i++) {
        int ni = n0 + n1;
        n0 = n1;
        n1 = ni;
    }
    return n1;
}
```

Fibonacci 5 – fórmula

$$fib(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$$

```
static int fibo(int n) {
    ops++;
    if (n < 2)
        return 1;
    n += 1;
    double t1 = Math.pow((1 + SQRT_5) / 2, n);
    double t2 = Math.pow((1 - SQRT_5) / 2, n);
    return (int) Math.round((t1 - t2) / SQRT_5);
}
```

fibonacci - tiempos

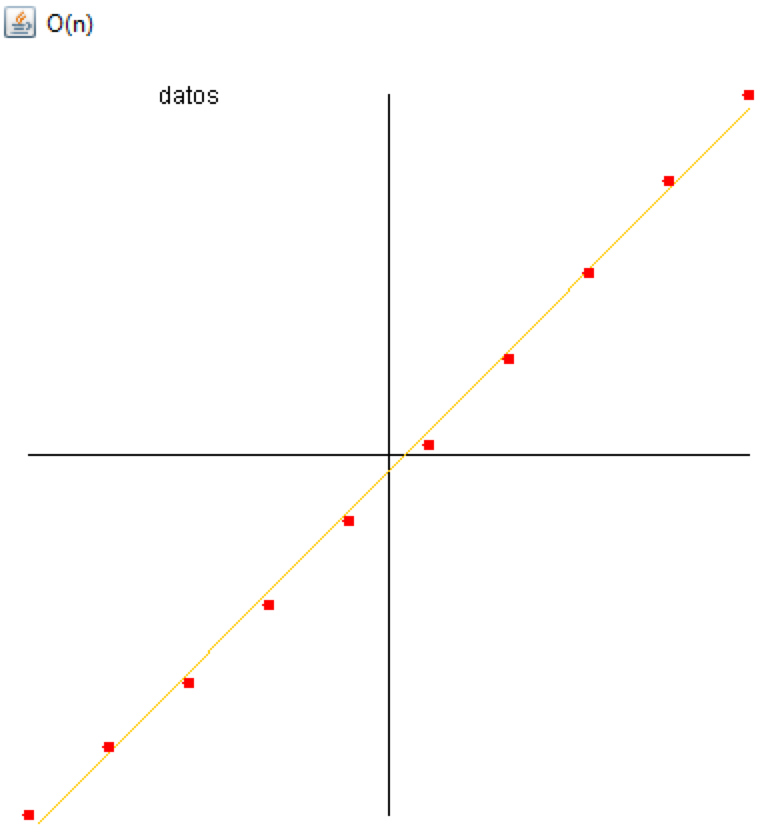
N	recursiva	iterativa	fórmula
10	812.001	6.569	158.460
20	3.397.022	6.979	154.766
30	392.534.843	6.979	153.945
40		7.800	153.944
50		8.621	153.945

$O(1{,}6^n)$ $O(n)$ $O(1)$

iterativa

```
public static void main(String[] args) {
      for (int n = 10; n < 105; n+= 10) {
         long t0 = System.nanoTime();
         for (int i = 0; i < 1000; i++)
            fibo1(n);
         long t2 = System.nanoTime();
         System.out.printf("%d: %d%n", n, t2 - t0);
      }
   }
```

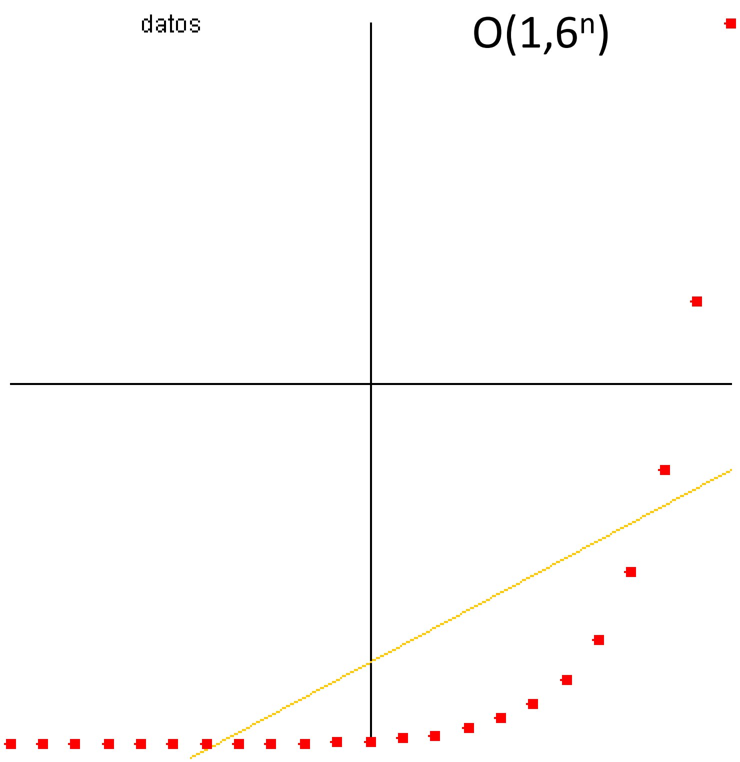
```
10 0.10469364955925486
20 0.1723654967412887
30 0.23442707079160435
40 0.31007614844406894
50 0.3912395735055524
60 0.46632809755211757
70 0.5485557849526885
80 0.632572506819411
90 0.721964227205798
100 0.8047836539410471
```



recursiva

```
public static void main(String[] args) {
      for (int n = 1; n < 25; n++) {
         long t0 = System.nanoTime();
         for (int i = 0; i < 1000; i++)
            fibo1(n);
         long t2 = System.nanoTime();
         System.out.printf("%d: %d%n", n, t2 - t0);
      }
   }
```

1	0.0003454115529308664
2	0.0009024123542856691
3	0.0014586236435194873
4	0.0014298064511035409
5	0.0024222231871814806
6	0.004271655330588462
7	0.006676904007168772
8	0.01159635403302529
9	0.01764559590401116
10	0.02863560462812005
11	0.04751560273329096
12	0.07462705421184979
13	0.14815155474674427
14	0.19916785422448202
15	0.3473805961606026
16	0.5550637333659665
17	0.8289885165462005
18	1.351826338913868
19	2.167616581333565
20	3.577065067641451
21	5.695772162592126
22	9.180202588810246
23	14.898625064641308



fibo recursiva

- $T(n) = T(n-1) + T(n-2)$
- hipótesis: $T(n) = x^n$
- $x^n = x^{n-1} + x^{n-2}$
- $x^2 - x - 1 = 0$
- $x = \frac{1 \pm \sqrt{1+4}}{2} = \{ 1.618, -0.61 \}$
- $T(n) \in O(1.6^n)$

corroboración experimental

fibo1

N	T(N)	1,618^n	T(N)/1,618^n
10	33.662	123	274
12	72.661	322	226
14	197.868	843	235
16	626.448	2.206	284
18	1.627.697	5.776	282
20	3.227.889	15.121	213
22	8.685.293	39.585	219
24	22.499.166	103.630	217
26	56.970.218	271.295	210
28	151.812.614	710.229	214

conclusiones

- Para problemas de tamaño pequeño
  - hay que optimizar el código
- Para problemas de tamaño grande ($n \rightarrow \infty$)
  1. hay que elegir un buen algoritmo
  2. hay que optimizar el código
- Algunos algoritmos muy buenos para n grande son muy malos para n pequeño
  - ejemplo: *quicksort*
    - solución: híbrido (entre *quick* e inserción)

conclusiones

- Si en un programa combinamos 2 algoritmos, el algoritmo peor impone su ley
  – porque cuando $n \to \infty$, el algoritmo de baja complejidad tiene una contribución despreciable
  – $\lim_{n \to \infty} (ax^2 + bx + c) \approx \lim_{n \to \infty} (ax^2)$
  – $O(x^2) \cup O(x) \cup O(1) = O(x^2)$
- En consecuencia hay que empezar cambiando el algoritmo peor

Problemas NP

- Aquellos para los que encontrar la solución requiere
  1. probar todas las formas posibles
  2. en tiempo polinómico sabemos si es la solución buscada

Problema: suma de subconjuntos

- Dado un conjunto de números enteros, ¿existe un subconjunto tal que la suma de sus elementos sea 0?
- Ejemplo

  { -2, -3, 15, 14, 7, -10 }

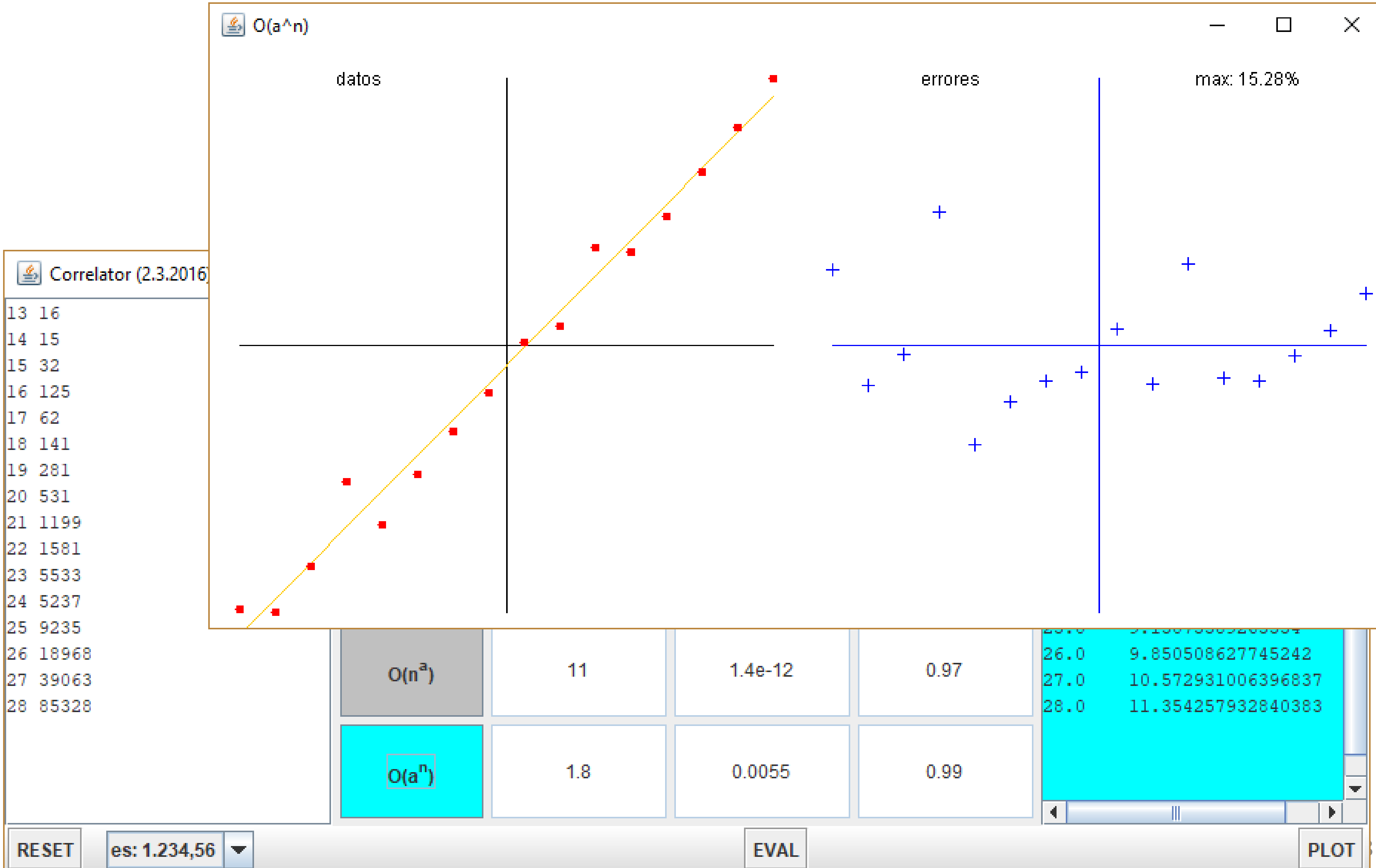
  – solución: sí existe, pero hay ir probando una a una
    { -2, -3, 15, -10 }

The travelling salesman problem (TSP) asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?
It is an NP-hard problem in combinatorial optimization, ...

Problema: suma de subconjuntos

```
List<Integer> suma0(List<Integer> set) {
    BigInteger max = BigInteger.ONE.shiftLeft(set.size());
    BigInteger mask = BigInteger.ONE;
    while (mask.compareTo(max) < 0) {
        List<Integer> subset = new ArrayList<>();
        int s = 0;
        for (int i = 0; i < set.size(); i++)
            if (mask.testBit(i)) {
                s += set.get(i);
                subset.add(set.get(i));
            }
        if (s == 0)
            return subset;
        mask = mask.add(BigInteger.ONE);
    }
    return Collections.emptyList();
}
```

Problema: suma de subconjuntos



ejemplo : proof of work

- https://blockchain.info/

Block #509271

Summary	
Number Of Transactions	224
Output Total	1,127.86613546 BTC
Estimated Transaction Volume	67.1644822 BTC

Hashes	
Hash	00000000000000000053dd9d775650e933eeb9d51446efade2cd20a05bcc3e18
Previous Block	0000000000000000003b205d94e92633b68e234d7f13ffcd6444988324a2a0de
Next	

- http://blockchain.mit.edu/blockchain/

encontrar N tal que

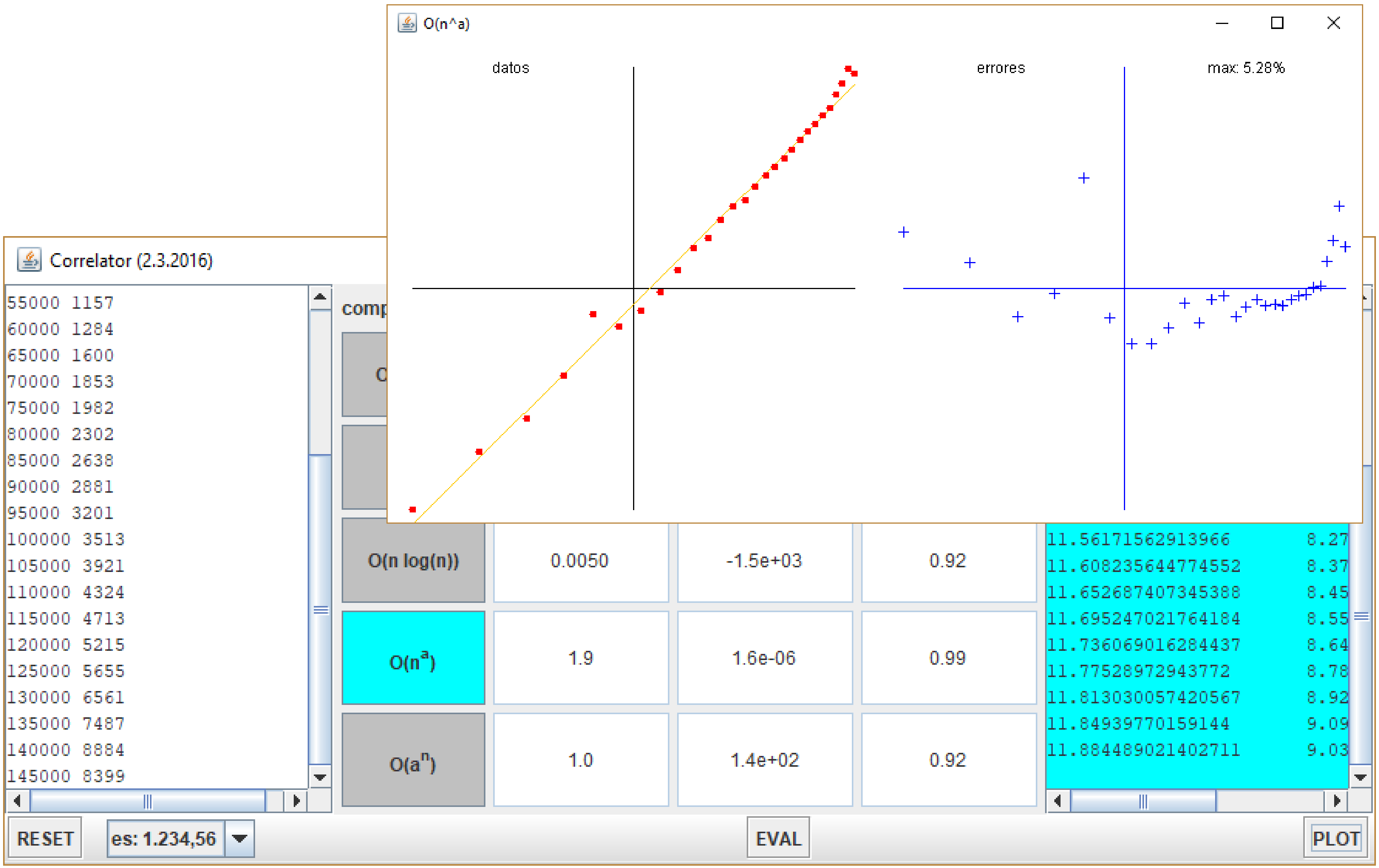
h(N, bloque anterior, transacciones)

tenga un mínimo número de ceros iniciales

ejercicio – suma 2

- Given a set (unsorted) of n numbers;
- and a number x;
- what is the fastest algorithm to determine if the set contains two numbers whose sum exactly equals x?
- ej.
  - set: { -7, 5, 4, 33, 12, -4 }
  - x: 1
  - [5, -4 }

ejercicio – suma 2 /algo 1



ejercicio – suma 2 /medidas

algo 1	algo 2	algo 3
10000 46	10000 16	10000 16
20000 204	110000 110	110000 31
30000 404	210000 94	210000 15
40000 867	310000 125	310000 32
50000 969	410000 188	410000 32
60000 1422	510000 219	510000 78
70000 1972	610000 406	610000 78
80000 2566	710000 265	710000 78
90000 3266	810000 234	810000 79
100000 4208	910000 269	910000 63
110000 5341	1010000 281	1010000 94
120000 6512	1110000 312	1110000 94
130000 8874	1210000 359	1210000 109
140000 10100	1310000 406	1310000 110
	1410000 390	1410000 133
$O(n^2)$	**$O(n \log n)$**	**$O(n)$**