1) Para cada una de las funciones siguientes indique su orden de crecimiento con la notación O(f(n)) y ordene por velocidad de crecimiento (en sentido creciente)

**a) 6 + n2; 4 + n + 2 log(n); n2 + 3 \* 2n; 5n3   
  
O(n) < O(n2) < O(n3) < O(2n)**

**b) 5n; n2 + 5 \* 2n; 9; 7\*n   
  
O(n) < O(n) < O(n2)**

**c) 2; log(n) + 2n; 5n + 2n; n  
  
O(1) < O(log(n)) < O(n)**

2) Cuales de las siguientes afirmaciones son verdaderas. Justifique.

a) n ∈ O(1)   
  
Falso. O(1) es un tiempo constante, n, en cambio, es un tiempo que aumenta en base a la entrada.

b) 500 ∈ O(n)   
  
Falso. 500 es una constante, n no.

c) 4n ∈ O(4n)  
  
Falso. 4n sería O(n), tiempo polinómico. O(4n) es tiempo exponencial.

d) n2 + 1/5n3 ∈ O(n3)

Verdadero, ya que tomamos el peor de los casos (O(n3)).

e) n\*(n+2)2/3 ∈ O(n2)  
  
Falso, al desarrollar la multiplicación vemos que es O(n3).

f) 2 log**2**n ∈ O(log**4**n)   
  
Verdadero. Por propiedades de logaritmos, podemos expresar cualquier logaritmo de una base en otra base (cambio de base).

g) 3n ∈ O(2n)  
  
Verdadero. Si tomamos 3n como (2+1)n y desarrollamos el binomio, nos da que el término de “mayor peso” es 2n.

3) a) O(a2\*log(a))   
b) O(n2)  
  
4) Suponiendo que se sabe que el tiempo de ejecución de un algoritmo pertenece a O(NlogN) y que el de otro algoritmo pertenece a O(n2), se pide:

a) ¿Que se puede decir sobre el rendimiento relativo de estos algoritmos?  
  
El primero crece más lento que el segundo, por lo que en casos en los que N sea muy grande, O(nlog(n)) será mejor.  
  
b) Suponiendo que ambos algoritmos resuelvan el mismo problema, enumere en cuales caso implementaría uno y en cuales implementaría el otro.   
  
Depende de las constantes que acompañan a la expresión original. La notación Big-O nos esconde dichas constantes, y es posible que, para ciertos valores, el algoritmo de complejidad O(n2) sea mejor que el algoritmo de O(NlogN). No podemos saber en qué casos utilizar cada uno. A lo sumo, podríamos decir, que si N es muy grande, lo correcto sería utilizar el algoritmo con complejidad O(NlogN).

c) En el caso de que ambos algoritmos sean de ordenación interna, enumere situaciones donde sería válido usar el algoritmo cuadrático.

Cuando la cantidad de elementos a ordenar sea pequeña podríamos utilizar el algoritmo cuadrático. Una de las mejoras de QuickSort es utilizar el algoritmo de inserción cuando la cantidad de elementos es pequeña, ya que resulta más rápido de esta forma ordenar el conjunto.

5) Sea la siguiente tabla:

|  |  |  |  |
| --- | --- | --- | --- |
| T(n)  Tiempo de ejecución  proporcional a: | Tamaño máximo del problema para 103seg | Tamaño máximo del problema para 104seg | Incremento |
| 100 n | 10 | 100 | 90 |
| 5n2 | 14 | 44 | 30 |
| n3 / 2 | 12 | 27 | 15 |
| 2n | 10 | 13 | 3 |

Se pide:

a) completar los datos que faltan.

b) saque alguna conclusión observando los respectivos incrementos.  
  
Podemos notar que los incrementos se hacen menores a medida que la complejidad aumenta. Si bien en un tiempo 103 todos comparten un tamaño similar, cuando observamos un tiempo 104 las discrepancias entre los tamaños empiezan a notarse fuertemente.

6)

Ejemplo de resolución:

T(n) = C \* n1  
T(n) = C/100 \* ni100\*(C\*n1) = C\*ni100n1 = C\*ni

|  |  |  |  |
| --- | --- | --- | --- |
| O(n) | 1 | 100  veces más rápido | 1000 veces  más rápido |
| n | N1 | 100N1 | 1000N1 |
| n2 | N2 | 10N2 | 31.6N2 |
| n3 | N3 | 4.64N3 | 10N3 |
| n5 | N4 | 2.51N4 | 3.98N4 |
| 2n | N5 | Log2(100N5) | Log2(1000N5) |
| 3n | N6 | Log3(100N6) | Log3(1000N5) |

7)

Ejemplo de resolución:

1 = T(N) = C\*O(n)  
1 = T(1000) = C\*1000  
1/1000 = C

|  |  |  |  |
| --- | --- | --- | --- |
|  | Ni / seg | Ni / minuto | Ni / hs. |
| n | 1000 | 60000 | 3.600.00 |
| n log n | 140 | 18000 | 1.080.00 |
| n2 | 31 | 62sqrt(15) | 1860 |
| n3 | 10 | 39 | 153 |
| 2n | 9 | 15 | 21 |

60 = C\*n  
n = 60000

|  |  |  |
| --- | --- | --- |
|  | Ni / seg | 10 veces más rápido |
| n | 1000 | 10000 |
| n log n | 140 | 3000 |
| n2 | 31 | 95 |
| n3 | 10 | 21 |
| 2n | 9 | 12 |

8)

10) Ambas formas de calcular el factorial (recursivo e iterativo) conllevan una complejidad O(n). Por más que la recursiva implique, valga la redundancia, recursividad, esta solo se utiliza para llamarse a sí misma.

**public** **static** **long** factorialRecursivo(**int** n) {

**if** (n == 1)

**return** 1;

**return** n \* *factorialRecursivo*(n-1);

}

**public** **static** **long** factorialIterativo(**long** n) {

**long** sum = n;

**for** (**int** i = 1; i < n; i++) {

sum \*= i;

}

**return** sum;

}

Sin embargo, al comprobar el tiempo, el factorial recursivo en varias ocasiones arrojó un tiempo de ejecución menor.

**long** time = System.*currentTimeMillis*(), endTime;

System.***out***.println(*factorialIterativo*(25));

**for** (**int** i = 0; i < 100000000; i++) {}

System.***out***.println(System.*currentTimeMillis*()-time);

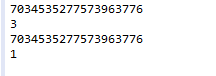
time = System.*currentTimeMillis*();

System.***out***.println(*factorialRecursivo*(25));

**for** (**int** i = 0; i < 100000000; i++) {}

endTime = System.*currentTimeMillis*();

System.***out***.println(endTime-time);



14) En el triple de tiempo ordena N = 17320.

Con el triple de elementos, tarda 90 segundos.

Con el triple de elementos en una máquina 3 veces más rápida, tarda 30 segundos.

15) Con n = 5000, se tarda 250 segundos en ordenarlo.

Si t = 40, se puede ordenar un vector de máximo de 2000 elementos.

16) El algoritmo de búsqueda binaria tiene una complejidad de O(log(n)), ya que descarta la mitad de elementos en cada búsqueda, es decir, se realiza una suma sucesiva de n/2 +n/4 + n/8… por lo que genera un logaritmo de base 2.

El algoritmo de búsqueda lineal tiene una complejidad O(n), ya que recorremos todos los elementos hasta hallar el que buscamos. Esto lo hace más lento.

17) El algoritmo de burbujeo tardaría 40 segundos en ordenar 8000 elementos en dicha PC.   
  
El algoritmo de selección, al tener la misma complejidad, tardaría el mismo tiempo para ambos casos.

18) En 20 segundos, en dicha PC, ordenaría un vector de n = 20.000 elementos.

En una máquina 5 veces más rápida, ordena un vector de n = 10.000 elementos en 1 segundo, y uno de n = 20.000 en 4 segundos.

19) Si la segunda prueba se realizó sobre una máquina dos veces más rápida.

20) 12 millones de elementos.