Juan Menduiña

Trabajo Práctico N° 4:
Tiempos de Ejecución.

Ejercicio 1.

Debido a un error en la actualización de sus sistemas, el banco AyED perdió la información del estado de todas sus cuentas. Afortunadamente, logran recuperar un backup del día anterior y, utilizando las transacciones registradas en las últimas 24hrs, podrán reconstruir los saldos. Hay poco tiempo que perder, el sistema bancario debe volver a operar lo antes posible.

Las transacciones se encuentran agrupadas en consultas, una consulta cuenta con un valor y un rango de cuentas consecutivas a las que hay que aplicar este cambio, por ejemplo la consulta (333..688= 120) implica sumar $120 a todas las cuentas entre la número 333 y la número 688 (inclusive). Entonces, la recuperación de los datos consiste en aplicar todas las consultas sobre el estado de las cuentas recuperadas en el backup del día anterior.

El equipo de desarrollo se pone manos a la obra y llega a una solución rápidamente (Algoritmo procesarMovimientos). Toman cada consulta y recorren el rango de cuentas aplicando el valor correspondiente, como muestra el siguiente algoritmo.

```
Consultas.comenzar()
While(!consultas.fin()){
        Consulta = consultas.proximo();
        for(i = consulta.desde; i < consulta.hasta; i++){
                cuenta[i] = cuenta[i] + consulta.valor;
        }
}
```

Escriben la solución en pocos minutos y ponen en marcha el proceso de recuperación. Enseguida se dan cuenta que el proceso va a tardar muchas horas en finalizar, son muchas cuentas y muchos movimientos, la solución aunque simple es ineficiente. Luego de discutir varias ideas llegan a una solución (Algoritmo procesarMovimientosOptimizado) que logra procesar toda la información en pocos segundos. Ambos algoritmos se encuentran en el archivo Ejercicio 1 - rsq_tn_ayed.zip del material adicional.

(a) *Para que se pueda experimentar el tiempo que demora cada uno de los dos algoritmos en forma empírica, se debe ejecutar cada uno de ellos, con distintas cantidades de elementos y completar la tabla. Luego, hacer la gráfica para comparar los tiempos de ambos algoritmos. Tener en cuenta que el algoritmo posee dos constantes CANTIDAD_CUENTAS y CANTIDAD_CONSULTAS, sin embargo, por simplicidad, ambas toman el mismo valor. Sólo se necesita modificar CANTIDAD_CUENTAS.*

N° Cuentas (y consultas)	procesarMovimientos	procesarMovimientosOptimizado
1.000	0,034	0,001
10.000	0,036	0,003
25.000	0,034	0,006
50.000	0,033	0,011
100.000	0,046	0,019

(b) *¿Por qué procesarMovimientos es tan ineficiente? Tener en cuenta que pueden existir millones de movimientos diarios que abarquen gran parte de las cuentas bancarias.*

El método tiene una complejidad temporal de O(N*M), donde N es la cantidad de consultas y M es el promedio del tamaño de los rangos de cuentas afectadas por cada consulta. En el caso que haya muchas consultas o los rangos sean grandes, este algoritmo sería muy ineficiente.

(c) *¿En qué se diferencia procesarMovimientosOptimizado? Observar las operaciones que se realizan para cada consulta. Aunque los dos algoritmos se encuentran explicados en los comentarios, no es necesario entender su funcionamiento para contestar las preguntas.*

En lo que se diferencia *procesarMovimientosOptimizado* es en que hace un único recorrido tanto en el arreglo de consultas como el de cuentas. Esto es así gracias a la ayuda de un arreglo auxiliar en donde se registran los cambios que deben aplicarse a las cuentas sumando los valores correspondientes (en el anterior algoritmo, los cambios se realizaban, directamente, dentro del bucle anidado).

Ejercicio 2.

La clase BuscadorEnArrayOrdenado del material adicional (Ejercicio 2 - Tiempo.zip) resuelve el problema de buscar un elemento dentro de un array ordenado. El mismo problema lo resuelve de dos maneras diferentes: búsqueda lineal y búsqueda dicotómica.

Se define la variable cantidadElementos, la cual se va modificando para determinar una escala (por ejemplo, de a 100.000 o 1.000.000, dependiendo de la capacidad de cada equipo). Realizar una tabla con el tiempo que tardan en ejecutarse ambos algoritmos, para los distintos valores de la variable.

Por ejemplo:

N	Lineal	Dicotómica
100.000	0,002	0,000
200.000	0,005	0,000
300.000	0,006	0,000
400.000	0,007	0,000
500.000	0,008	0,000
600.000	0,011	0,000

Ejercicio 3.

En la documentación de la clase ArrayList que se encuentra en el siguiente link https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html, se encuentran las siguientes afirmaciones:

- *"The size, isEmpty, get, set, iterator, and listIterator operations run in constant time."*
- *"All of the other operations run in linear time (roughly speaking)."*

Explicar por qué se cree que algunas operaciones se ejecutan en tiempo constante y otras en tiempo lineal.

Las operaciones como *size*, *isEmpty*, *get*, *set*, *Iterator* y *listIterator* se ejecutan en tiempo constante porque acceden a posiciones de memoria o propiedades simples. En cambio, el resto de las operaciones se ejecutan en tiempo lineal porque requieren mover o procesar varios elementos.

Ejercicio 4.

Determinar si las siguientes sentencias son verdaderas o falsas, justificando la respuesta utilizando notación Big-Oh.

(a) 3^n *es de O(*2^n*).*

¿Existen constantes c $>$ 0 y $n_0 > 0$ tales que f (n) $\leq$ c g (n) para todo n $\geq n_0$?

$3^n \leq c2^n$
$\frac{3^n}{2^n} \leq c$
$(\frac{3}{2})^n \leq c.$

La sentencia es FALSA, ya que no existen c $>$ 0 y $n_0 > 0$ tales que f (n) $\leq$ c g (n) para todo n $\geq n_0$.

(b) *n +* log_2 *(n) es de O(n).*

¿Existen constantes c $>$ 0 y $n_0 > 0$ tales que f (n) $\leq$ c g (n) para todo n $\geq n_0$?

n + log_2 (n) $\leq$ cn
log_2 (n) $\leq$ cn - n
log_2 (n) $\leq$ n (c - 1)
log_2 (n) $\leq$ n (2 - 1)
log_2 (n) $\leq$ n * 1
log_2 (n) $\leq$ n
n $\geq$ 1.

La sentencia es VERDADERA, ya que existen c= 2 y n_0= 1 tales que f (n) $\leq$ c g (n) para todo n $\geq n_0$.

(c) $n^{\frac{1}{2}} + 10^{20}$ *es de O(*$n^{\frac{1}{2}}$*).*

¿Existen constantes c $>$ 0 y $n_0 > 0$ tales que f (n) $\leq$ c g (n) para todo n $\geq n_0$?

$n^{\frac{1}{2}} + 10^{20} \leq cn^{\frac{1}{2}}$
$10^{20} \leq cn^{\frac{1}{2}} - n^{\frac{1}{2}}$
$10^{20} \leq n^{\frac{1}{2}}$ (c - 1)
$10^{20} \leq n^{\frac{1}{2}}$ (2 - 1)
$10^{20} \leq n^{\frac{1}{2}}$ * 1
$10^{20} \leq n^{\frac{1}{2}}$

$n \geq (10^{20})^2$
$n \geq 10^{40}$.

La sentencia es VERDADERA, ya que existen c= 2 y $n_0 = 10^{40}$ tales que f (n) ≤ c g (n) para todo $n \geq n_0$.

(d) $\begin{cases} 3n + 17, n < 100 \\ 317, n \geq 100 \end{cases}$ *tiene orden lineal.*

La sentencia es VERDADERA, ya que el orden de mayor grado es O(n).

(e) *Mostrar que p(n)= $3n^5 + 8n^4 + 2n + 1$ es O(n^5).*

¿Existen constantes $c > 0$ y $n_0 > 0$ tales que p (n) ≤ c g (n) para todo $n \geq n_0$?

$3n^5 + 8n^4 + 2n + 1 \leq cn^5$.

Primer término:

$3n^5 \leq cn^5$
$3n^5 \leq 3n^5$.

$c_1 = 3$; $n_1 = 0$.

Segundo término:

$8n^4 \leq cn^5$
$8n^4 \leq 8n^5$.

$c_2 = 8$; $n_2 = 0$.

Tercer término:

$2n \leq cn^5$
$2n \leq 2n^5$.

$c_3 = 2$; $n_3 = 0$.

Cuarto término:

$1 \leq cn^5$
$1 \leq 1n^5$
$1 \leq n^5$.

$c_4 = 1$; $n_4 = 1$.

Entonces:

$3n^5 + 8n^4 + 2n + 1 \leq c_1 n^5 + c_2 n^5 + c_3 n^5 + c_4 n^5$
$3n^5 + 8n^4 + 2n + 1 \leq (c_1 + c_2 + c_3 + c_4)\, n^5$
$3n^5 + 8n^4 + 2n + 1 \leq (3 + 8 + 2 + 1)\, n^5$
$3n^5 + 8n^4 + 2n + 1 \leq 14n^5$, con c= 14 para todo n $\geq n_0$, con n_0= 1.

Por lo tanto, p (n) es O(n^5), ya que existen constantes c= 14 y n_0= 1 tales que p (n) ≤ c g (n) para todo n ≥ n_0.

(f) *Si p(n) es un polinomio de grado k, entonces, p(n) es O(n^k).*

¿Existen constantes c > 0 y $n_0 > 0$ tales que p (n) ≤ c g (n) para todo n ≥ n_0?

La sentencia es VERDADERA, ya que el término de mayor grado k domina a medida que n ⟶ ∞, siendo los términos de orden inferior (n^{k-1}, n^{k-2}, …) despreciables frente a n^k.

Ejercicio 5.

Se necesita generar una permutación random de los n primeros números enteros. Por ejemplo, [4, 3, 1, 0, 2] es una permutación legal, pero [0, 4, 1, 2, 4] no lo es, porque un número está duplicado (el 4) y otro no está (el 3). Se presentan tres algoritmos para solucionar este problema. Se asume la existencia de un generador de números random, ran_int (i, j), el cual genera, en tiempo constante, enteros entre i y j inclusive con igual probabilidad (esto significa que puede retornar el mismo valor más de una vez). También se supone el mensaje swap() que intercambia dos datos entre sí.

```
public class EjercicioPermutaciones {
	private static Random rand = new Random();

	public static int[] randomUno(int n) {
		int i, x = 0, k;
		int[] a = new int[n];
		for (i = 0; i < n; i++) {
			boolean seguirBuscando = true;
			while (seguirBuscando) {
				x = ran_int(0, n - 1);
				seguirBuscando = false;
				for (k = 0; k < i && !seguirBuscando; k++)
					if (x == a[k])
						seguirBuscando = true;
			}
			a[i] = x;
		}
		return a;
	}

	public static int[] randomDos(int n) {
		int i, x;
		int[] a = new int[n];
		boolean[] used = new boolean[n];
		for (i = 0; i < n; i++) used[i] = false;
		for (i = 0; i < n; i++) {
			x = ran_int(0, n - 1);
			while (used[x]) x = ran_int(0, n - 1);
			a[i] = x;
			used[x] = true;
		}
		return a;
	}
```

```
	public static int[] randomTres(int n) {
		int i;
		int[] a = new int[n];
		for (i = 0; i < n; i++) a[i] = i;
		for (i = 1; i < n; i++) swap(a, i, ran_int(0, i - 1));
		return a;
	}


	private static void swap(int[] a, int i, int j) {
		int aux;
		aux = a[i]; a[i] = a[j]; a[j] = aux;
	}

	/** Genera en tiempo constante, enteros entre i y j con igual probabilidad.
	 */
	private static int ran_int(int a, int b) {
		if (b < a || a < 0 || b < 0) throw new IllegalArgumentException("Parametros
invalidos");
		return a + (rand.nextInt(b - a + 1));
	}

	public static void main(String[] args) {
		System.out.println(Arrays.toString(randomUno(1000)));
		System.out.println(Arrays.toString(randomDos(1000)));
		System.out.println(Arrays.toString(randomTres(1000)));
	}
}
```

(a) *Analizar si todos los algoritmos terminan o alguno puede quedar en loop infinito.*

Los primeros dos algoritmos podrían quedar en un *loop* infinito sólo cuando el generador de números aleatorios retorna siempre una secuencia de números repetidos. Ambos algoritmos están diseñados para que terminen una vez hallada una permutación legal y, en caso contrario, sigan iterando hasta hallarla.

El último de los algoritmos, en caso de que el generador de números aleatorios retorne siempre una secuencia de números repetidos, la posición intercambiada sería la misma, pero no generaría un *loop* infinito.

(b) *Describir, con palabras, la cantidad de operaciones que realizan.*

- *randomUno*: Para cada valor, compara con los anteriores hasta encontrar uno no repetido (con posibles repeticiones).
- *randomDos*: Usa *used[]* para evitar repeticiones y sólo realiza reintentos si hay colisiones.
- *randomTres*: Llena el arreglo y hace un intercambio aleatorio por posición.

Ejercicio 6.

(a) *Se supone que se tiene un algoritmo de O(log^2 n) y se dispone de 1 hora de uso de CPU. En esa hora, la CPU puede ejecutar el algoritmo con una entrada de tamaño n= 1.024 como máximo. ¿Cuál sería el mayor tamaño de entrada que podría ejecutar el algoritmo si se dispone de 4 horas de CPU?*

En 1 hora:

$(log_2\ n)^2 = (log_2\ 1024)^2$
$(log_2\ n)^2 = 10^2$
$(log_2\ n)^2 = 100$.

En 4 horas:

$(log_2\ n)^2 = 4 * 100$
$(log_2\ n)^2 = 400$
$(log_2\ n)^2 = 20^2$
log_2 n= 20
n= 2^{20}.

Por lo tanto, el mayor tamaño de entrada que podría ejecutar el algoritmo si se dispone de 4 horas de CPU es 2^{20}.

(b) *Considerando que un algoritmo requiere T(n) operaciones para resolver un problema y la computadora procesa 10.000 operaciones por segundo. Si T(n)= n^2, determinar el tiempo en segundos requerido por el algoritmo para resolver un problema de tamaño n= 2.000.*

T(n)= n^2.

T(2000)= 2000^2
T(2000)= 4000000.

10.000 operaciones por segundo.

Tiempo (en segundos)= $\frac{4000000}{10000}$
Tiempo (en segundos)= 400.

Por lo tanto, el tiempo en segundos requerido por el algoritmo para resolver un problema de tamaño n= 2.000 es 400.

Ejercicio 7.

Para cada uno de los siguientes fragmentos de código, calcular, intuitivamente, el orden del tiempo de ejecución.

`for(int i = 0; i< n; i++)` `  sum++;`	`for(int i = 0; i< n; i+=2)` `  sum++;`

`for(int i = 0; i< n; i++)`
`  for(int j = 0; j< n; j++)`
`  sum++;`
	`for(int i = 0; i < n + 100; ++i) {`
	`  for(int j = 0; j < i * n ; ++j){`
	`  sum = sum + j;`
	`  }`
	`  for(int k = 0; k < n + n + n; ++k){`
	`  c[k] = c[k] + sum;`
	`  }`
	`}`

`for(int i = 0; i< n; i++)`
`  for(int j = 0; j< n; j++)`
`  sum++;`
`for(int i = 0; i< n; i++)`
`  sum++;`
	`int i,j;`
	`int x = 1;`
	`for (i = 0; i <= n²; i=i+2)`
	`  for (j = n; j >= 1; j-= n/4)`
	`  x++;`

(a)

T(n)= $\sum_{i=1}^{n} cte$
T(n)= n cte.

Por lo tanto, para este fragmento de código, el orden de tiempo de ejecución es O(n).

(b)

T(n)= $\sum_{i=1}^{\frac{n}{2}} cte$
T(n)= $\frac{1}{2}$ n cte.

Por lo tanto, para este fragmento de código, el orden de tiempo de ejecución es O(n).

(c)

T(n)= $\sum_{i=1}^{n} \sum_{j=1}^{n} cte$
T(n)= $\sum_{i=1}^{n} n * cte$
T(n)= nn cte
T(n)= n^2 cte.

Por lo tanto, para este fragmento de código, el orden de tiempo de ejecución es O(n^2).

(d)

T(n)= $\sum_{i=1}^{n+100}(\sum_{j=1}^{in} cte_1 + \sum_{k=1}^{3n} cte_2)$
T(n)= $\sum_{i=1}^{n+100}(in * cte_1 + 3n * cte_2)$
T(n)= $\sum_{i=1}^{n+100} in * cte_1 + \sum_{i=1}^{n+100} 3n * cte_2$
T(n)= n cte_1 $\sum_{i=1}^{n+100} i$ + (n + 100) 3n cte_2
T(n)= n cte_1 $\frac{(n+100)(n+100+1)}{2}$ + (n + 100) 3n cte_2
T(n)= n cte_1 $\frac{(n+100)(n+101)}{2}$ + (n + 100) 3n cte_2
T(n)= $\frac{(n+100)(n+101)n}{2}$ cte_1 + ($3n^2$ + 300n) cte_2
T(n)= $\frac{n^3+101n^2+100n^2+10100n}{2}$ cte_1 + ($3n^2$ + 300n) cte_2
T(n)= $\frac{n^3+201n^2+10100n}{2}$ cte_1 + ($3n^2$ + 300n) cte_2
T(n)= ($\frac{1}{2}n^3 + \frac{201}{2}n^2$ + 5050n) cte_1 + ($3n^2$ + 300n) cte_2.

Por lo tanto, para este fragmento de código, el orden de tiempo de ejecución es O(n^3).

(e)

T(n)= $\sum_{i=1}^{n}\sum_{j=1}^{n} cte_1 + \sum_{i=1}^{n} cte_2$
T(n)= $\sum_{i=1}^{n} n * cte_1$ + n cte_2
T(n)= nn cte_1 + n cte_2
T(n)= n^2 cte_1 + n cte_2.

Por lo tanto, para este fragmento de código, el orden de tiempo de ejecución es O(n^2).

(f)

T(n)= $cte_1 + \sum_{i=1}^{\frac{n^2}{2}}\sum_{j=1}^{4} cte_2$
T(n)= $cte_1 + \sum_{i=1}^{\frac{n^2}{2}} 4cte_2$
T(n)= $cte_1 + \frac{n^2}{2} * 4cte_2$
T(n)= $cte_1 + 2n^2$ cte_2.

Por lo tanto, para este fragmento de código, el orden de tiempo de ejecución es O(n^2).

Ejercicio 8.

Para cada uno de los algoritmos presentados, calcular el T(n).

(a) *Expresar, en función de n, el tiempo de ejecución.*

(b) *Establecer el orden de dicha función usando notación Big-Oh.*

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

```
2. int c = n;
   while ( c > 1 ) {
     algo_de_O(1);
     c = c / 2;
   }
```

```
3. public static void calcular(int n) {
        int i, j, r = 0;
        for ( i = 1; i < n; i= i+2)
           for (j = 1; j <= i; j++ )
                 r = r + 1;
        return r;
   }

  }
```

(1)

Iteraciones del *while*:

c= 1.
c= 2.
c= 4.
…
c= 2^{k-1}.

2^{k-1}= n - 1
$log_2\ 2^{k-1}$= log_2 (n - 1)
(k - 1) log_2 2= log_2 (n - 1)
(k - 1) * 1= log_2 (n - 1)
k - 1= log_2 (n - 1)
k= log_2 (n - 1) + 1.

Entonces:

T(n)= $cte_1 + \sum_{c=1}^{log_2(n-1)+1} cte_2$
T(n)= cte_1 + [log_2 (n - 1) + 1] $cte_2 \leq$ O(log_2 n).

¿Existen constantes c $>$ 0 y $n_0 > 0$ tales que f (n) $\leq$ c g (n) para todo n $\geq n_0$?

cte_1 + [log_2 (n - 1) + 1] $cte_2 \leq$ c log_2 n.

Primer término:

$cte_1 \leq$ c log_2 n
$cte_1 \leq cte_1\ log_2$ n.

c_1= cte_1; n_0= 2.

Segundo término:

cte_2 [log_2 (n - 1) + 1] $\leq$ c log_2 n
cte_2 [log_2 (n - 1) + 1] $\leq cte_2\ log_2$ n.

c_2= cte_2; n_0= 2.

Entonces:

cte_1 + [log_2 (n - 1) + 1] $cte_2 \leq c_1\ log_2$ n + $c_2\ log_2$ n
cte_1 + [log_2 (n - 1) + 1] $cte_2 \leq (c_1 + c_2)\ log_2$ n
cte_1 + [log_2 (n - 1) + 1] $cte_2 \leq (cte_1 + cte_2)\ log_2$ n, con c= cte_1 + cte_2 para todo n $\geq$ n_0, con n_0= 2.

Por lo tanto, f (n) es O(log_2 n), ya que existen constantes c= cte_1 + cte_2 y n_0= 2 tales que f (n) $\leq$ c g (n) para todo n $\geq$ n_0.

(2)

Iteraciones del *while*:

c= n.
c= $\frac{n}{2}$.
c= $\frac{n}{4}$.
…
c= $\frac{n}{2^{k-1}}$.

$\frac{n}{2^{k-1}}$= 1 + 1
$\frac{n}{2^{k-1}}$= 2
2 * 2^{k-1}= n
2^k= n
$log_2\ 2^k = log_2$ n
k log_2 2= log_2 n
k * 1= log_2 n
k= log_2 n.

Entonces:

T(n)= $cte_1 + \sum_{c=1}^{log_2 n} cte_2$
T(n)= $cte_1 + log_2$ n * $cte_2 \leq$ O(log_2 n).

¿Existen constantes c > 0 y $n_0 > 0$ tales que f (n) ≤ c g (n) para todo n ≥ n_0?

Primer término:

$cte_1 \leq$ c log_2 n
$cte_1 \leq cte_1\ log_2$ n.

c_1= cte_1; n_0= 2.

Segundo término:

$cte_2\ log_2$ n ≤ c log_2 n
$cte_2\ log_2$ n ≤ $cte_2\ log_2$ n.

c_2= cte_2; n_0= 2.

Entonces:

$cte_1 + log_2$ n * $cte_2 \leq c_1\ log_2$ n + $c_2\ log_2$ n
$cte_1 + log_2$ n * $cte_2 \leq (c_1 + c_2)\ log_2$ n
$cte_1 + log_2$ n * $cte_2 \leq (cte_1 + cte_2)\ log_2$ n, con c= $cte_1 + cte_2$ para todo n ≥ n_0, con n_0= 2.

Por lo tanto, f (n) es O(log_2 n), ya que existen constantes c= $cte_1 + cte_2$ y n_0= 2 tales que f (n) ≤ c g (n) para todo n ≥ n_0.

(3)

Iteraciones del primer *for*:

i= 1.
i= 3.
i= 5.
…
i= 2k - 1.

2k - 1= n - 1
2k= n - 1 + 1
2k= n
k= $\frac{n}{2}$.

Iteraciones del segundo *for*:

j= 1.
j= 2.
j= 3.
…
j= k.

k= i.

Entonces:

T(n)= $cte_1 + \sum_{i=1}^{\frac{n}{2}} \sum_{j=1}^{i} cte_2$
T(n)= $cte_1 + \sum_{i=1}^{\frac{n}{2}} i * cte_2$
T(n)= $cte_1 + cte_2 \sum_{i=1}^{\frac{n}{2}} i$
T(n)= $cte_1 + cte_2 \frac{\frac{n}{2}(\frac{n}{2}+1)}{2}$
T(n)= $cte_1 + \frac{\frac{n^2}{4}+\frac{n}{2}}{2} cte_2$
T(n)= $cte_1 + (\frac{1}{8} n^2 + \frac{1}{4}$ n) $cte_2 \leq$ O(n^2).

¿Existen constantes c $>$ 0 y $n_0 > 0$ tales que f (n) ≤ c g (n) para todo n ≥ n_0?

Primer término:

$cte_1 \leq cn^2$
$cte_1 \leq cte_1 n^2$.

c_1= cte_1; n_0= 1.

Segundo término:

cte_2 ($\frac{1}{8} n^2 + \frac{1}{4}$ n) ≤ cn^2
cte_2 ($\frac{1}{8} n^2 + \frac{1}{4}$ n) ≤ $cte_2 n^2$.

c_2= cte_2; n_0= 1.

Entonces:

$cte_1 + (\frac{1}{8} n^2 + \frac{1}{4}$ n) $cte_2 \leq c_1 n^2 + c_2 n^2$
$cte_1 + (\frac{1}{8} n^2 + \frac{1}{4}$ n) $cte_2 \leq (c_1 + c_2)\, n^2$
$cte_1 + (\frac{1}{8} n^2 + \frac{1}{4}$ n) $cte_2 \leq (cte_1 + cte_2)\, n^2$, con c= $cte_1 + cte_2$ para todo n ≥ n_0, con n_0= 1.

Por lo tanto, f (n) es O(n^2), ya que existen constantes c= $cte_1 + cte_2$ y n_0= 1 tales que f (n) ≤ c g (n) para todo n ≥ n_0.

Ejercicio 9.

(a) *Expresar la función del tiempo de ejecución de cada uno de los siguientes algoritmos, resolverla y calcular el orden.*

```
static public int rec2(int n){
	if (n <= 1)
		return 1;
	else
		return (2 * rec2(n-1));
}

static public int rec1(int n){
	if (n <= 1)
		return 1;
	else
		return (rec1(n-1) + rec1(n-1));
}

static public int rec3(int n){
	if  ( n == 0 )
		return  0;
	else {
		if ( n == 1 )
			return 1;
		else
			return (rec3(n-2) * rec3(n-2));
	}
}
static public int potencia_iter(int x, int n){
	int potencia;
	if  (n == 0)
		potencia = 1;
	else {
		if  (n == 1)
			potencia  =  x;
		else{
			potencia  = x;
			for (int  i  = 2 ; i <=  n ; i++) {
				potencia *= x ;
			}
		}
	}
	return  potencia;
}
```

```
static public int potencia_rec( int x, int n){
        if( n == 0 )
                return 1;
        else{
                if( n == 1)
                        return x;
                else{
                        if ( (n  %  2 ) == 0)
                                return  potencia_rec (x * x, n / 2 );
                        else
                                return potencia_rec (x * x, n / 2) * x;
                }
        }
}
```

Algoritmo *rec2*:

$T(n)=\begin{cases} cte_1, n \leq 1 \\ T(n-1)+cte_2, n > 1 \end{cases}$.

Paso 1:

T(n)= T(n - 1) + cte_2, si n > 1.

Paso 2:

T(n)= T(n - 1 - 1) + cte_2 + cte_2
T(n)= T(n - 2) + $2cte_2$, si n > 2.

Paso 3:

T(n)= T(n - 2 - 1) + cte_2 + $2cte_2$
T(n)= T(n - 3) + $3cte_2$, si n > 3.

Paso i (Paso general):

T(n)= T(n - i) + icte_2, si n > i.

n - i= 1
i= n - 1.

Entonces:

T(n)= T(n - (n - 1)) + (n - 1) cte_2
T(n)= T(n - n + 1) + (n - 1) cte_2
T(n)= T(1) + (n - 1) cte_2
T(n)= cte_1 + (n - 1) cte_2
T(n)= cte_1 + (n - 1) cte_2 ≤ O(n).

¿Existen constantes c > 0 y $n_0 > 0$ tales que f (n) ≤ c g (n) para todo n ≥ n_0?

Primer término:

$cte_1 \leq$ cn
$cte_1 \leq cte_1$n.

$c_1 = cte_1$; $n_0 = 1$.

Segundo término:

$cte_2 \leq$ cn
$cte_2 \leq cte_2$n.

$c_2 = cte_2$; $n_0 = 1$.

Entonces:

cte_1 + (n - 1) $cte_2 \leq c_1$n + c_2n
cte_1 + (n - 1) $cte_2 \leq (c_1 + c_2)$ n
cte_1 + (n - 1) $cte_2 \leq (cte_1 + cte_2)$ n, con c= $cte_1 + cte_2$ para todo n $\geq n_0$, con $n_0 = 1$.

Por lo tanto, f (n) es O(n), ya que existen constantes c= $cte_1 + cte_2$ y $n_0 = 1$ tales que f (n) $\leq$ c g (n) para todo n $\geq n_0$.

Por lo tanto, la función de tiempo de ejecución es T(n)= c_1 + (n - 1) c_2 y el orden es O(n).

Algoritmo *rec1*:

Algoritmo *rec3*:

Algoritmo *potencia_iter*:

Algoritmo *potencia_rec*:

(b) *Comparar el tiempo de ejecución del método "rec2" con el del método "rec1".*

El tiempo de ejecución del método "rec2" es menor al del método "rec1".

(c) *Implementar un algoritmo más eficiente que el del método "rec3" (es decir, que el T(n) sea menor).*

Ejercicio 10.

(a) *Resolver las siguientes recurrencias.*

(b) *Calcular el O(n). Justificar usando la definición de Big-Oh.*

1.

$$T(n) = \begin{cases} 2, n = 1 \\ T(n-1) + n, n \geq 2 \end{cases}$$

2.

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T\left(\frac{n}{4}\right) + \sqrt{n}, & n \geq 2 \end{cases}$$

2.

$$T(n) = \begin{cases} 2, n = 1 \\ T(n-1) + \frac{n}{2}, n \geq 2 \end{cases}$$

3.

$$T(n) = \begin{cases} 1, n = 1 \\ 4T\left(\frac{n}{2}\right) + n^2, n \geq 2 \end{cases}$$

(1)

(2)

(3)

(4)

Ejercicio 11.

Calcular el tiempo de ejecución de los métodos buscarLineal y buscarDicotomica de la clase BuscadorEnArrayOrdenado. Comparar el tiempo con los valores obtenidos, empíricamente, en el Ejercicio 2.

***buscarLineal*:**

T(n)= $cte_1 + \sum_{i=1}^{n} cte_2$
T(n)= cte_1 + n $cte_2 \leq$ O(n).

***buscarDicotomica*:**

T(n)= $\begin{cases} cte_1, n \leq 1 \\ T\left(\frac{n}{2}\right) + cte_2, n > 1 \end{cases}$.

Paso 1:

T(n)= T($\frac{n}{2}$) + cte_2, si n $>$ 1.

Paso 2:

T(n)= T($\frac{\frac{n}{2}}{2}$) + cte_2 + cte_2
T(n)= T($\frac{n}{4}$) + 2cte_2, si n $>$ 2.

Paso 3:

T(n)= T($\frac{\frac{n}{4}}{2}$) + cte_2 + 2cte_2
T(n)= T($\frac{n}{8}$) + 3cte_2, si n $>$ 3.

Paso i (Paso general):

T(n)= T($\frac{n}{2^i}$) + icte_2, si n $>$ i.

$\frac{n}{2^i}$= 1
1 * 2^i= n
2^i= n
log_2 2^i= log_2 n
i log_2 2= log_2 n
i * 1= log_2 n
i= log_2 n.

Entonces:

T(n)= T($\frac{n}{2^{log_2 n}}$) + log_2 n * cte_2
T(n)= T($\frac{n}{n}$) + log_2 n * cte_2
T(n)= T(1) + log_2 n * cte_2
T(n)= cte_1 + log_2 n * cte_2 ≤ O(log_2 n).

Ejercicio 12.

Calcular el tiempo de ejecución de procesarMovimientos y procesarMovimientosOptimizado del Ejercicio 1. Comparar el tiempo con los valores obtenidos empíricamente.

procesarMovimientos:

T(n)= $cte_1 + \sum_{i=1}^{n}(cte_2 + \sum_{i=1}^{n+1} cte_3)$
T(n)= $cte_1 + \sum_{i=1}^{n}[cte_2 + (n+1)\ cte_3]$
T(n)= cte_1 + n $[cte_2$ + (n + 1) $cte_3]$
T(n)= cte_1 + n cte_2 + n^2 cte_3 + n cte_3
T(n)= cte_1 + n $(cte_2 + cte_3)$ + n^2 $cte_3 \leq$ O(n^2).

procesarMovimientosOptimizado:

T(n)= $cte_1 + \sum_{i=1}^{n} cte_2 + \sum_{i=1}^{n} cte_3$
T(n)= cte_1 + n cte_2 + n cte_3
T(n)= cte_1 + n $(cte_2 + cte_3) \leq$ O(n).

Ejercicio 13.

Resolver las recurrencias y calcular el orden. Para cada recurrencia, se muestra, a modo de ejemplo, el código correspondiente.

<table>
<tr>
<td>

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

$$T(n) = \begin{cases} 1, n \leq 1 \\ T(n-1) + c, n \geq 2 \end{cases}$$

</td>
<td>

```
int recursivo(int n){
if (n == 1)
        return  1;
else
        return (recursivo (n/2));
}
```

$$T(n) = \begin{cases} 1, n = 1 \\ T(n/2) + c, n \geq 2 \end{cases}$$

</td>
</tr>
<tr>
<td>

```
int recur (int n){
if (n == 1)
        return  1;
else
        return(recur(n/2)+recur(n/2));
}
```

$$T(n) = \begin{cases} 1, n = 1 \\ 2T(n/2) + c, n \geq 2 \end{cases}$$

</td>
<td>

```
int recursivo(int n){
if (n <= 5)
        return  1;
else
        return (recursivo (n-5));
}
```

$$T(n) = \begin{cases} 1, n \leq 5 \\ T(n-5) + c, n \geq 6 \end{cases}$$

</td>
</tr>
<tr>
<td>

```
int recur (int n){
if (n == 1)
        return  1;
else
        return(recur(n-1)+recur(n-1));
}
```

$$T(n) = \begin{cases} 1, n = 1 \\ 2T(n-1) + c, n \geq 2 \end{cases}$$

</td>
<td>

```
int recursivo(int n){
if (n <= 7)
        return  1;
else
        return (recursivo (n/8));
}
```

$$T(n) = \begin{cases} 1, n \leq 7 \\ T(n/8) + c, n \geq 8 \end{cases}$$

</td>
</tr>
</table>

(a)

(b)

(c)

(d)

(e)

(f)

Ejercicio 14.

Considerar el siguiente fragmento de código:

```
int count = 0; int n = a.length;
        for (int i = 1; i <= n; i = i*2)   {
                for (int j = 0; j < n; j+=n/2)  {
                a[j]++;
                }
        }
```

Este algoritmo se ejecuta en una computadora que procesa 1.000 operaciones por segundo. Determinar el tiempo aproximado que requerirá el algoritmo para resolver un problema de tamaño n= 4.096.

Iteraciones del primer *for*:

i= 1.
i= 2.
i= 4.
…
i= 2^{k-1}.

2^{k-1}= n
(k - 1) log_2 2= log_2 n
(k - 1) * 1= log_2 n
k - 1= log_2 n
k= log_2 n + 1.

Iteraciones del segundo *for*:

j= 0.
j= $\frac{n}{2}$.

Entonces:

T(n)= $cte_1 + \sum_{i=1}^{log_2 n+1} \sum_{j=1}^{2} cte_2$
T(n)= $cte_1 + \sum_{i=1}^{log_2 n+1} 2cte_2$
T(n)= cte_1 + (log_2 n + 1) * $2cte_2$
T(n)= cte_1 + 2 (log_2 n + 1) cte_2.

T(4096)≅ log_2 4096
T(4096)≅ 12.

1.000 operaciones por segundo.

Tiempo (en segundos)$\cong \frac{12}{1000}$
Tiempo (en segundos)$\cong$ 0,012.

Por lo tanto, el tiempo aproximado que requerirá el algoritmo para resolver un problema de tamaño n= 4.096 es 0,012 segundos.