

Práctica No. 4 (Tiempo de Ejecución y Sorting)

- Utilizando las siguientes definiciones de *suma* y *suma1*:

```
/**
 * suma primeros n numeros.
 * @param n: ultimo numero a sumar
 * @return suma de los primeros n numeros.
 */
public static int suma1(int n){
    return (n * (n + 1) / 2);
}

/**
 * suma primeros n numeros.
 * @param n: ultimo numero a sumar
 * @return suma de los primeros n numeros.
 */
public static int suma(int n){
    if(n==0)
        return 0;
    return (n + suma(n - 1));
}
```

Realice experimentos para comparar los tiempos de corrida (en nanosegundos) de las funciones y complete la tabla para cada una de las entradas:

n	suma	suma1
0		
5000		
10000		
15000		
20000		
25000		

Ayuda: Utilizar el método **nanoTime** de la clase **System** (*System.nanoTime()*) para tomar el tiempo en nanosegundos antes y después de cada llamada.

- Sean $f(n), g(n), t(n), s(n)$ funciones crecientes no negativas, usando las definiciones de O (big oh), Ω (big omega), y Θ (big theta), probar las siguientes propiedades:
 - $f(n) \in O(g(n))$ y $t(n) \in O(s(n))$ entonces $f(n) * t(n) \in O(g(n) * s(n))$,
 - Probar que la relación $f(n) \equiv_{\Theta} g(n)$ definida como $f(n) \in \Theta(g(n))$ es una relación de equivalencia.
 - Probar que cualquier polinomio: $a_d * x^d + a_{d-1} * x^{d-1} + \dots + a_0 \in O(x^d)$.
- Probar que se cumple $\log_k n \in O(\log_t n)$, donde k y t son números positivos.
- Considere el siguiente programa:

```
public boolean cuantoTardo(int key, int[] data, int size) {
    int index = 0;
    while(index < size) {
        if(data[index] == key)
            return true;
        index++;
    }
    return false;
}
```

- Describa qué hace la función **cuantoTardo**.
- Calcule el tiempo de ejecución ($t(n)$) de la función **cuantoTardo**, en el peor caso, y la tasa de crecimiento $O(t(n))$.

5. Considere el siguiente fragmento de una implementación clásica de listas sobre arreglos:

```
public class ListaSobreArreglos<T extends Comparable>{
    private static final int MAX_LIST = 100;
    private T item[];
    private int numItems;
    ...
}
```

Además de las operaciones básicas se desea proveer una operación `public Integer buscar(T x)`, que retorna la posición del elemento en caso de que pertenezca, y una excepción en caso contrario. Esta operación debe ser eficiente, se requiere que sea $O(1)$ si el elemento buscado está al final de la lista, y que sea $O(\log N)$ en otro caso, manteniendo la inserción y la eliminación en $O(N)$.

6. Calcule, en función del tamaño de entrada, el tiempo de ejecución ($t(n)$) del algoritmo de ordenamiento *insertionSort*, en el peor caso, y la tasa de crecimiento $O(t(n))$ de acuerdo al siguiente programa definido en el lenguaje C:

```
#include <stdlib.h>
typedef int ElementType;
void InsertionSort( ElementType A[], int N ){
    int j, P;
    ElementType Tmp;
    for( P = 1; P < N; P++ ){
        Tmp = A[ P ];
        for( j = P; j > 0 && A[ j - 1 ] > Tmp; j-- )
            A[ j ] = A[ j - 1 ];
        A[ j ] = Tmp;
    }
}
```

7. Considere el siguiente programa, que retorna la multiplicación de matrices:

```
public static int[][] multiply(int[][] a, int[][] b) {
    int[][] c = new int[a.length][b[0].length];
    if (a[0].length == b.length) {
        for (int i = 0; i < a.length; i++) {
            for (int j = 0; j < b[0].length; j++) {
                for (int k = 0; k < a[0].length; k++) {
                    c[i][j] += a[i][k] * b[k][j];
                }
            }
        }
    }
    return c;
}
```

Calcule, en función del tamaño de entrada, el tiempo de ejecución ($t(n)$) de **multiply**, en el peor caso, y la tasa de crecimiento $O(t(n))$.

8. Dar las ecuaciones de recurrencias para las siguientes funciones en Haskell, y calcular su tiempo de ejecución.

```
maximum :: (Ord a) => [a] -> a
maximum [x] = x
maximum (x:xs)
    | x > maxTail = x
    | otherwise = maxTail
    where maxTail = maximum xs
```

```
isPalindrome xs = xs == reverse xs
```

```
reverse [] = []
reverse [x] = [x]
```

```
reverse (x:xs) = reverse xs ++ [x]
```

```
sort [] = []
```

```
sort (x:xs) = insert x (sort xs)
```

```
insert x [] = [x]
```

```
insert x (y:ys) = if x <= y then x : (y : ys) else y:(insert x ys)
```

9. Calcule el tiempo de ejecución y tasa de crecimiento de la operación *pow* definida en el lenguaje C de la siguiente manera:

```
#include <stdio.h>
#define IsEven( N ) ( ( N ) % 2 == 0 )
long int Pow( long int X, unsigned int N ){
    if( N == 0 )
        return 1;
    if( N == 1 )
        return X;
    if( IsEven( N ) )
        return Pow( X * X, N / 2 );
    else
        return Pow( X * X, N / 2 ) * X;
}
int main( ){
    printf( "2^21=%ld\n", Pow( 2, 21 ) );
    return 0;
}
```

10. Calcular una cota para la siguiente ecuación de recurrencia:

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

11. Demostrar $\log n \in O(n)$ y $\log n \in O(\sqrt{n})$

12. Resuelva las siguientes ecuaciones de recurrencia:

- $T(1) = 1, T(n) = 2 * T(n/2) + n.$
- $T(1) = 1, T(n) = 1 + T(n/2).$
- $T(1) = 1, T(n) = 2T(n - 1) - 1$

13. Considere la clase `ArraySorter` provista en el repositorio. Esta clase contiene métodos de ordenamiento, complete las implementaciones de métodos incompletos.

- Cuál es el tiempo de ejecución del QuickSort cuando le pasamos un arreglo ordenado en forma decreciente?
- Agregue una versión random del quickSort a la clase `ArraySorter`. En este caso en vez de tomar el primer elemento del arreglo como pivot, utilice un generador de números random para obtener una posición aleatoria.
- Utilizando la la clase `ArrayGenerator`, realice experimentos para comparar los métodos de ordenación definidos.

14. Utilizando el template de la clase `ArraySorter`, agregar la implementación del método de ordenación **Radix Sort** con el siguiente perfil:

```
public static Integer[] radixSort(Integer[] entrada)
```