Departamento de Computación
FCEFQyN, Universidad Nacional de Río Cuarto
Asignatura: Estructuras de Datos y Algoritmos - Algoritmos y Estructuras de Datos II
Primer Cuatrimestre de 2025

Práctica Nº 8

Acceder al código de base de los ejercicios de esta práctica aceptando el siguiente assignment: `https://classroom.github.com/a/AIQ-NmOV`.

1. Utilizando las siguientes definiciones de Potencia de dos *powTwo0* y *powTwo1*:

```
    /**
 * @post calcula potencias de dos.
 */
public static long powTwo0(long n){
  if (n == 0)
    return 1;
  else
    return 2 * powTwo0(n-1);
}

/**
 * @post calcula potencias de dos.
 */
public static long powTwo1(long n){
  if (n == 0)
    return 1;
  else
    return powTwo1(n-1) + powTwo1(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	powTwo0	powTwo1
10		
15		
20		
25		
30		
35		

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

2. Considere el siguiente fragmento de una implementación de listas de elementos *int* sobre arreglos:

```
public class ListaSobreArreglos{
        private static final int MAX_LIST = 100;
        private int item[];
        private int numItems;
...
```

   Se desea proveer una operación `public Integer buscar(int 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 $O(N)$.

3. 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: $a_d * x^d + a_{d-1} * x^{d-1} + \cdots + a_0 \in O(x^d)$.
- Probar que cualquier algoritmo de sorting que utilice comparaciones para ordenar es $\Omega(n * log\ n)$

4. 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. 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;
        }
}
```

6. 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)). Considere en principio matrices cuadradas. ¿Qué cambia si las matrices no son cuadradas?

7. 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 xs)
```

8. 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;
}
```

9. Analizar el tiempo de ejecución del siguiente algoritmo en el peor caso, escriba y desarrolle la ecuación del tiempo de ejecución del algoritmo y determine la tasa de crecimiento.

```
public static int[] complexity(int[] array) {
        int[] aux = new int[0];
        for (int value : array) {
                aux = complexityAux(aux, value);
        }
        return aux;
}
public int[] complexityAux(int[] array, int value) {
        int[] other = new int[array.length + 1];
        for (int i = 0; i < array.length; i++) {
                other[i] = array[i];
        }
        other[other.length - 1] = value;
        return other;
}
```

   - qué hace la función *complexity*?

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

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

11. Resuelva las siguentes ecuaciones de recurrencia:

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

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

12. Considere el proyecto sorting provisto en el repositorio. Este proyecto encapsula algoritmos de sorting, agregue las clases e implementaciones de los algoritmos de ordenamiento: *bubbleSort*, *insertionSort*, *mergeSort* y *quickSort*

    - ¿Cuál es el tiempo de ejecución del algoritmo de ordenamiento **quickSort** cuando le pasamos un arreglo ordenado en forma decreciente?
    - Agregue una versión random de quickSort al proyecto sorting. 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.
    - Utilice los test definidos y agregue otros para comparar los algoritmos de ordenamiento.

13. Cree una implementación para el algoritmo de ordenamiento **radix Sort**.