TRABAJO PRACTICO N° 1
Tema: Análisis de Algoritmos

1. Ordene las siguientes funciones por tasa de crecimiento: n, $\sqrt{n}$, $n^{1.5}$, n^2, n log n, n log (log n), n $\log^2$ n, n log (n^2), 2/n, 2^n, $2^n/2$, 37, n^2 log n, n^3. Indique cuáles crecen con la misma tasa.

2. Para cada uno de los tres fragmentos de programas siguientes:
- **a)** Haga un análisis del tiempo de ejecución (O grande).
- **b)** Implante el código y dé el tiempo de ejecución para diferentes valores de n.
- **c)** Compare sus análisis con los tiempos de ejecución reales.

(1)
```
sum=0;
for(i=1; i<=n; i++)
        sum=sum + 1;
```

(2)
```
sum=0;
for(i=1; i<=n; i++)
        for(j=1; j<=n; j++)
                sum=sum + 1;
```

(3)
```
sum=0;
for(i=1; i<=n; i++)
        for(j=1; j<=n * 2; j++)
                sum=sum + 1;
```

3. Dado el siguiente programa con 3 métodos y un main, se pide obtener mediante las notaciones asintóticas, el T(n) e indicar su tipo de complejidad. La solución debe estar acompañado de su desarrollo matemático.

Programa :
```
void inicializarMatriz (int[][] matriz, int N) (
   int i, j;
   for (i=0; i < N; i++)
      for (j=0; j < N; j++)
         if( (i+j) % 2 == 0)
            matriz[i][j] = 1;
         else
            matriz[i][j] = i + j;
}

void mostrarMatriz (int[][] matriz, int N) {
   int i, j;
   System.out.println("Matriz: ");
   for (i=0; i < N; i++) {
      for (j = 0; j < N; j++)
         System.out.print(matriz[il[j]);
      System.out.println(" ");
   }
}

void traspuesta(int[][] matriz, int N) {
   int i, j, aux;
   for (i = 0; i < N; i++)
      for (j = 0; j < N; j++) {
         if (i < j) {
            aux = matriz[i][j];
            matriz[i][j] = matriz[j][i];
            matriz[j][i] = aux;
         }
      }
}
// programa principal
public static void main(String[] args){
   int[][] matriz=new int[10][10];
   inicializarMatriz (matriz, 10);
   mostrarMatriz (matriz, 10);
   traspuesta (matriz, 10);
   mostrarMatriz (matriz, 10);
}
```

TRABAJO PRACTICO N° 1
Tema: Análisis de Algoritmos

4. De acuerdo al siguiente programa en lenguaje C, analizar el siguiente código y obtener el T(n), e indicar su tipo de complejidad. La solución debe estar acompañado de su desarrollo matemático.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
   int arreglo[100], i, N, pos=0, mayor;
   printf("Ingrese el valor de N: ");
   scanf("%d", &N);
   if (N < 1 || N > 100){
      printf("Error, tiene que ser entre 1 y 100.");
      return 0;
   } else {
      for (i=0; i < N; i++) {
         arreglo[i]= rand()% 10;
         printf("%d", arreglo[i]);
      }
      mayor = arreglo[pos];
      for (i=1; i < N; i++)
         if(arreglo[i] > mayor){
            pos = i;
            mayor = arreglo[pos];
         }
      printf("\nEl %d, en la posición: %d", mayor, pos);
   }
   return 0;
}
```

5. Supongamos que necesitamos generar una permutación *aleatoria* de los primeros n enteros. Por ejemplo, [4, 3, 1, 5, 2] y [3, 1, 4, 2, 5] son permutaciones legales, pero [5, 4, 1, 2, 1] no lo es, porque un número (1) está duplicado y otro (3) no está. Esta rutina se usa a menudo en simulación de algoritmos. Suponemos la existencia de un generador de números aleatorios, *generadorEnterosAleatorio(i, j)*, el cual genera enteros entre i y j con una probabilidad igual. Aquí se presentan tres algoritmos:

5.1. Llenar el arreglo desde vector[0] hasta vector[n-1] como sigue: para llenar vector[i] generar números aleatorios hasta que se tiene uno que no está ya en vector[0], vector[1], vector[2], …, vector[i - 1].
5.2. Igual que el algoritmo anterior, pero mantener un arreglo adicional llamado *usado*. Cuando un número aleatorio, *aleatorio*, se coloca en el arreglo, a *usado[aleatorio]* se le asigna 1 o true. Esto significa que cuando se llena vector[i] con un número aleatorio, se puede comprobar en un paso si el número ya ha sido utilizado, en vez de los (posiblemente) i - 1 pasos del primer algoritmo.
5.3. Llenar el arreglo de forma que vector[i] = i. Después,

```
for(i=2; i<=n; i++)
        intercambiarValores(vector[i], vector[generadorEnterosAleatorio(0, i)]);
```

a) Dar un análisis (O grande) tan preciso como sea posible del tiempo de ejecución *esperado* de cada algoritmo.
b) Escribir programas (separados) para ejecutar cada algoritmo 10 veces, para obtener un buen promedio. Ejecutar el programa (5.1) para n = 250, 500, 1000, 2000; el programa (5.2) para n = 2500, 5000, 10000, 20000, 40000, 80000, y el programa (5.3) para n = 10000, 20000, 40000, 80000, 160000, 320000, 640000.
c) Comparar el análisis con los tiempos de ejecución reales.
d) ¿Cuál es el tiempo de ejecución en el peor caso para cada algoritmo?

6. Proporcione algoritmos eficientes (junto con los análisis del tiempo de ejecución) para:
a) Encontrar la suma de la subsecuencia mínima.
b) Encontrar la suma de la subsecuencia mínima positiva.
c) Encontrar el producto de la subsecuencia máxima.

TRABAJO PRACTICO N° 1
Tema: Análisis de Algoritmos

7. Se analizan los programas A y B y se encuentra que tienen un tiempo de ejecución en el peor caso no mayor que 150 n $\log_2$ n y n^2, respectivamente. Responda las siguientes preguntas, si es posible:

- **a)** ¿Qué programa tiene la mejor garantía en el tiempo de ejecución, para valores grandes de n (n > 10000)?.
- **b)** ¿Qué programa tiene la mejor garantía en el tiempo de ejecución, para valores pequeños de n (n < 100)?.
- **c)** ¿Qué programa se ejecutará más rápidamente en promedio para n = 1000?.
- **d)** ¿Es posible que el programa B se ejecute más rápidamente que el programa A para todas las entradas posibles?.

8. Se dice que un vector es mayoritario si tiene un elemento mayoritario y se dice que un elemento *x* es mayoritario en el vector cuando más de la mitad de los elementos del vector son iguales a *x*. Formalmente, si existe un *x* tal que (*0 ≤ i < n: vector[i]=x) > n / 2*, entonces *x* es elemento mayoritario y, por tanto, el vector es mayoritario. Como es evidente, si el vector es mayoritario, sólo existe en él un único elemento mayoritario.

- **a)** Supongamos que existe una relación de orden entre los elementos que figuran en el vector. Queremos determinar si un vector dado es mayoritario o no y, caso de que lo sea, saber cuál es ese elemento. ¿Qué tipo de estructura iterativa habría que utilizar para conseguir resolver este problema con un coste O(*n*)?. Identificar claramente el algoritmo final y justificar la respuesta indicando, por ejemplo, porqué hace falta que exista una relación de orden entre los elementos.

- **b)** Supongamos ahora que ya no tenemos una relación de orden entre los elementos y que la única operación disponible para comparar los elementos del vector es la igualdad. Diseñar un algoritmo que, con coste O(*n log n*), resuelva el mismo problema que se plantea en (a) (Obviamente no se puede ordenar el vector). Indicar el esquema empleado, justificar la corrección y determinar su coste.

9. Suponga que las líneas {6} y {7} del algoritmo 3 (página 10 del apunte de la cátedra) se sustituyen por:

```
{6}   sumaMaxIzq=sumaSubsecuenciaMaxima(a, izq, centro - 1) y;
{7}   sumaMaxDer=sumaSubsecuenciaMaxima(a, centro, der).
```

¿Todavía funcionará la rutina?. ¿Por qué?.