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. Supongamos que $T_1(r) = O(f(r))$ y $T_2(r) = O(f(r))$. ¿Cuáles de las siguientes operaciones son ciertas?:
- **a)** $T_1(r) + T_2(r) = O(f(r))$.
- **b)** $T_1(r) - T_2(r) = O(f(r))$.
- **c)** $T_1(r) / T_2(r) = O(1)$.
- **d)** $T_1 = O(T_2(r))$.

3. Para cada uno de los seis 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;
```

(4)
```
sum=0;
for(i=1; i<=n; i++)
        for(j=1; j<=i; j++)
                sum=sum + 1;
```

(5)
```
sum=0;
for(i=1; i<=n; i++)
        for(j=1; j<=i * 2; j++)
                for(k=1; k<=j; k++)
                        sum=sum + 1;
```

(6)
```
sum=0;
for(i=1; i<=n; i++)
        for(j=1; j<=i * 2; j++)
                if(j % i = = 0)
                        for(k=1; k<=j; k++)
                                sum=sum + 1;
```

4. 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:

4.1. 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].
4.2. 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.
4.3. 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 (4.1) para n = 250, 500, 1000, 2000; el programa (4.2) para n = 2500, 5000, 10000, 20000, 40000, 80000, y el programa (4.3) 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?.

5. Considere el siguiente algoritmo (conocido como *regla de Horner*) para evaluar:

$$f(x) = \sum_{i=0}^{n} a_i x^i:$$

```
poli=0;
for(i=n; i>=0; i--)
        poli=x * poli + ai;
```

a) Muestre cómo se realizan los pasos para este algoritmo con x = 3, $f(x) = 4x^4 + 8x^3 + x + 2$.
b) Explique por qué funciona el algoritmo.
c) ¿Cuál es el tiempo de ejecución de este 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.

7. La criba de Eratóstenes es un método para obtener todos los primos menores que n. Se empieza haciendo una tabla de enteros entre 2 y n. Se encuentra el entero más pequeño i, que no esté marcado, se imprime i, y se marcan i, 2i, 3i, …. El algoritmo termina cuando i > $\sqrt{n}$. ¿Cuál es el tiempo de ejecución de este algoritmo?.

8. Escriba la rutina de la exponenciación rápida sin recursión.

9. 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?.

10. 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 $(0 \leq i < n: vector[i]=x) > n / 2$, 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 algoritmo, de los vistos en clase, 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.

11. Suponga que las líneas {6} y {7} del algoritmo 3 (página 94 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?.

12. Supongamos que la línea {9} de la rutina de la búsqueda binaria (página 96 del apunte de la cátedra) tiene la proposición *ultimo=medio* en vez de *ultimo=medio - 1*. ¿Todavía funcionará la rutina?. ¿Por qué?.

13. El objetivo de este ejercicio es la comparación de distintos tiempos de ejecución en algoritmos que resuelven el mismo problema. El problema planteado consiste en calcular el n-ésimo término de la sucesión de Fibonacci. Esta serie se describe en la forma siguiente:

$f_0 = 0$
$f_1 = 1$
$f_n = f_{n-1} + f_{n-2}$ para n>=2

Los términos de la serie son: 0, 1, 1, 2, 3, 5, 8, ...

Los diversos algoritmos a utilizar y comparar son los siguientes:

a) Algoritmo 1: Se basa en la definición directa, y sería el siguiente:

```
int fibonacci1(int n)
    {
    if(n < 2)
            return n;
    else
            return fibonacci1(n-1) + fibonacci1(n-2);
    }
```

Para este algoritmo, crear una tabla de tiempos de ejecución para n=10, 11, 12, 13, ... hasta donde sea posible. Nótese que para valores de n no muy elevados, será impracticable el efectuar el cálculo, además el valor del elemento carecerá de sentido por su tamaño. Compárese el tiempo de ejecución en el cálculo de los diversos valores de n. Compruébese que la relación T(n) / T(n-1) es constante. ¿Qué valor tiene?. ¿Qué conclusiones se extraen?.

b) Algoritmo 2: Este algoritmo está basado en la traducción iterativa del algoritmo previo, evitando el recálculo innecesario de diversos términos de la serie. Cada elemento será calculado una sola vez y utilizado en los siguientes cálculos.

```
int fibonacci2(int n)
    {
    int i, j, k;
    i=0;
    j=1;
    for(k=1; k<n; k++)
            {
            j=i + j;
            i=j - i;
            }
    return j;
    }
```

Crear una tabla de tiempos similar a la anterior, pero en este caso con los valores de n doblándose cada vez en lugar de incrementarse en una unidad. Es decir, para n= 2, 4, 8, 16, 32, 64, 128, ... Los tiempos empiezan a ser significativos para valores muy altos de n. Calcúlese la relación entre T(n) y T(2n). Compruébense los valores, y extráiganse conclusiones. Es posible que sea necesario cambiar algún tipo para poder realizar los cambios. Téngase en cuenta que no nos interesa el cálculo del valor en sí (el término número 1000 tendrá un número de dígitos descomunal), sino el tiempo empleado en realizar dicho cálculo.

c) Algoritmo 3: Este algoritmo es más artificioso, y utiliza una técnica Divide y Vencerás

```
int fibonacci3(int n)
    {
    int i, j, k, t, h;
    i=1;
    j=0;
    k=0;
    h=1;
    while(n>0)
            {
            if(n % 2 = = 1)
                    {
                    t=j * h;
                    j=i * h + j * k + t;
                    i=i * k + t;
                    }
            t=h * h;
            h=2 * k * h + t;
            k=k * k + t;
```

```
            n=n / 2;
            }
return j;
}
```

Inténtese realizar la misma experimentación realizada en el caso del Algoritmo 2. No obstante, los tiempos no son significativos por grande que sea el tamaño del problema, obteniéndose en cualquier caso tiempos insignificantes. Compruébese por la teoría la complejidad del algoritmo para poder comparar la situación.