Instituto Politécnico Nacional
Escuela Superior de Cómputo

Problemario: Análisis de Algoritmos

Profesor: Dr. Benjamín Luna Benoso.

Grupo:__________________

Alumno:__________________________________

1. Calcular el costo computacional temporal (O) de las funciones *main* de cada uno de los mostrados en la figura 1, a partir del costo computacional temporal de línea por línea.

producto.c

```c
#include <stdio.h>
int main(void)
{
  int m, n;
  scanf("%d", &n);
  m = n * n;
  printf("%d\n", m);
  return 0;
}
```

suma.c

```c
#include <stdio.h>
int main(void)
{
  int m, n, i;
  scanf("%d", &n);
  m = 0;
  for (i=0; i<n; i++)
    m = m + n;
  printf("%d\n", m);
  return 0;
}
```

incremento.c

```c
#include <stdio.h>
int main(void)
{
  int m, n, i, j;
  scanf("%d", &n);
  m = 0;
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      m++;
  printf("%d\n", m);
  return 0;
}
```

Figura 1

2. Demostrar mediante la definición formal (encontrar valores constantes que cumplan las condiciones correspondientes o demostrar mediante reducción al absurdo) cada una de las siguientes afirmaciones.

i) $3n + 2 \in O(n)$ ii) $10n^2 + 4n - 6 \in O(n)$ iii) $10n^2 + 4n - 6 \notin O(n)$
iv) $6 \cdot 2^n + n^2 \in O(2^n)$ v) $6 \cdot 2^n + n^2 \notin O(n^{100})$ vi) $10n^2 + 4n + 2 \in \Omega(n^2)$

3. Demuestre las siguientes propiedades:

i) $f(n) \in \Theta(f(n))$ ii) $O(cf(n)) = O(f(n))$
iii) $t_1 \in O(t_2(n))$, entonces $t_1 + t_2 \in O(t_2)$.
iv) $t_1 \in \Omega(t_2(n))$, entonces $t_1 + t_2 \in \Omega(t_2)$.
v) $t_1 \in \Theta(t_2(n))$, entonces $t_1 + t_2 \in \Theta(t_2)$.

4. Demostrar las siguientes propiedades:

i) $f(n) \in \Theta(g(n))$ si y solo si $g(n) \in \Theta(f(n))$
ii) $f(n) \in O(g(n))$ si y solo si $g(n) \in \Omega(f(n))$
iii) $f(n) \in \Theta(g(n))$ si y solo si $f(n) \in O(g(n))$ y $f(n) \in \Omega(g(n))$

5. Demostrar que si $t_1(n) \in \Theta(f_1(n))$ y $t_2(n) \in \Theta(f_2(n))$, entonces $t_1(n) \cdot t_2(n) \in \Theta(f_1(n) \cdot f_2(n))$.

6. Demostrar que si $O(g(n)) \subseteq O(f(n))$, entonces $O(f(n) + g(n)) = O(f(n))$.

7. Calcular el costo computacional temporal en el peor de los casos de la función *main* mostrada en la figura 2 (transpuesta de una matriz), a partir del costo computacional temporal de línea por línea.

```
#define n 10
void trasponer(double m[n][n])
{
  int i, j;
  double aux;
  for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++) {
      aux = m[i][j];
      m[i][j] = m[j][i];
      m[j][i] = aux;
    }
}
```

Figura 2

8. De las siguientes afirmaciones, indicar cuales son ciertas y cuales no. Argumentar sus resultados.

i) $n^2 \in O(n^3)$

ii) $n^3 \in O(n^2)$

iii) $2^{n+1} \in O(2^n)$

iv) $(n+1)! \in O(n!)$

v) $f(n) \in O(n) \Rightarrow 2^{f(n)} \in O(2^n)$

vi) $3^n \in O(2^n)$

vii) $logn \in O(n^{1/2})$

viii) $n^{1/2} \in O(logn)$

ix) $n^2 \in \Omega(n^3)$

x) $n^3 \in \Omega(n^2)$

xi) $2^{n+1} \in \Omega(2^n)$

xii) $(n+1)! \in \Omega(n!)$

xiii) $f(n) \in \Omega(n) \Rightarrow 2^{f(n)} \in \Omega(2^n)$

xiv) $3^n \in \Omega(2^n)$

xv) $logn \in \Omega(n^{1/2)}$

xvi) $n^{1/2} \in \Omega(logn)$

9. Calcular el orden de complejidad de los siguientes algoritmos a partir del costo computacional de línea por línea.

```
Algoritmo1(a[0,...,n-1])
for i=n-1 to i>0 do
    for j=0 to j<i
        if a[j]>a[j+1] then
           temp=a[j]
           a[j]=a[j+1]
           a[j+1]=temp
```

```
Funcion Fibo(int n)

i=1
j=0
for k=1 to n do
     j=i+j
     i=j-i
return j
```

10. Implementar una función Iterativa que encuentre el máximo de un arreglo de enteros y calcular su orden de complejidad mediante el conteo temporal de línea por línea.

11. Calcular el orden de complejidad mediante el costo computacional de línea por línea de los siguientes algoritmos de ordenamiento.

Select-Sort$(A[0,\ldots,n-1])$

for $j \leftarrow 0$ to $j \leq n-2$ do
  $k \leftarrow j$
  for $i \leftarrow j+1$ to $i \leq n-1$ do
    if $A[i] < A[k]$ then
      $k \leftarrow i$
  Intercambia $(A[j], A[k])$

```
BubbleSort(A[1,...,n-1])

for i= n-1 downto i>0 do
    for j=0 to j<i do
        if A[j]>A[j+1]
            AUX=A[j]
            A[j]=A[j+1]
            A[j+1]=AUX
```

12 a) Demostrar que $f \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
b) Dar un ejemplo de funciones f y g tales que $f \in O(g)$ pero $f \notin \Omega(g)$
c) Demostrar que $\forall a, b > 1$ se tiene que $log_a n \in \Theta(log_b n)$

13. Demostrar que para cualquier constante k se verifica que $log^k n \in O(n)$

14. Considere el costo de un algoritmo como la función $t(n)$ definida como:

$$t(n) = \begin{cases} 1 & si \quad n = 1 \\ 1 + t(n-1) & si \quad n > 1 \end{cases}$$

Demuestre que $t(n) \in \Theta(n)$, demostrando por inducción matemática que $t(n) = n \ \forall n \geq 1$.

15. Calcular el orden de complejidad del algoritmo de búsqueda binaria de manera recursiva.

16. Considere el costo de un algoritmo como la función $t(n)$ definida como:

$$t(n) = \begin{cases} 4 + t(n/2) & si \quad n > 1 \\ 3 & si \quad n = 1 \end{cases}$$

Demuestre que $t(n) \in \Theta(log_2 n)$ cuando n es potencia de 2.

17. Use inducción matemática para demostrar que cuando n es potencia de 2, la solución de la recurrencia:

$$T(n) = \begin{cases} 2 & si \quad n = 2 \\ 2T(n/2) + n & si \quad n = 2^k, \ para \ k > 1 \end{cases}$$

es $T(n) = nlogn$.

18. Sean $f(n)$ y $g(n)$ funciones asintóticamente no negagtivas. Usando la definición de la notación Θ, demostrar que $max(f(n), g(n)) = \Theta(f(n) + g(n))$.

19. Demostrar mediante la definición de Θ que si a y b son números reales constantes con $b > 0$, entonces $(n + a)^b = \Theta(n^b)$.

20. Demuestre mediante la definición de O si cada una de las siguientes proposiciones son falsas o verdaderas:

i) $2^{n+1} = O(2^n)$.
ii) $2^{2n} = O(2^n)$.

21. Demostrar que $nlogn = O(n^{1+a})$, donde $0 < a < 1$.

22. Simplificar $O(3m^3 + 2mn^2 + n^2 + 10m + m^2)$.

23. Demostrar que si $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0$ con $a_k \neq 0$ es un polinomio, entonces $p(n) = O(n^k)$.

24. Calcule el orden de complejidad temporal del siguiente algoritmo en el mejor y en el pero de los casos (Calculando línea por línea).

Algorithm $Enigma(A_{n-1\times n-1})$
Input: Una matriz $A_{n-1\times n-1}$ de tamaño $n-1\times n-1$.

for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i][j] \neq A[j][i]$
      return false
return true

25.En los siguientes algoritmos, calcular el orden de complejidad en el peor y mejor caso. Argumentar sus resultados.

26. Resolver las siguientes ecuaciones de recurrencia:

i) $x(n) = x(n-1) + 5\ \forall n > 1,\ x(1) = 0$.
ii) $x(n) = 3x(n-1)\ \forall n > 1,\ x(1) = 4$.
iii) $x(n) = x(n/2) + n\ \forall n > 1,\ x(1) = 1$ (considerar para $n = 2^k$).
iv) $x(n) = x(n/3) + 1\ \forall n > 1,\ x(1) = 1$ (considerar para $n = 3^k$).

27. Calcular el orden de complejidad del siguiente algoritmo:

Algorithm $Q(n)$
Input: Un entero positivo n

if n=1 **return** 1
else return $Q(n-1) + 2n - 1$

28. Demostrar que la solución de:
i) $T(n) = T(n-1) + n$ es $O(n^2)$.
ii) $T(n) = T(\lceil n/2 \rceil) + 1$ es $O(log\ n)$.
iii) $T(n) = 2T(\lfloor n/2 \rfloor) + n$ es $O(nlog\ n)$.

29. Probar que si $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + log\ n$, entonces $T(n) = O(log\ nloglog\ n)$.

30. Demostrar que la recurrencia $T(n) = 4T(n/3) + n$ tiene orden de complejidad temporal $T(n) = \Theta(n^{log_3 4})$.

```
int minimo1(int *A, int ini, int fin)
  if(ini==fin)
     return A[ini]
  else
     medio=(ini+fin)/2
     min1=minimo1(A,ini,medio)
     min2=minimo1(A,medio+1,fin)
     if(min1>min2)
       return min1
     else
       return min2
```

```
int minimo2(A[0,...,n])
   if n==0
      return A[0]
   else
      aux=minimo2(A,n-1)
      if A[n]>aux
         return aux
      else
         return A[n]
```

```
int minimo3(int *A, int ini, int fin)
  if(ini==fin)
     return A[ini]
  else
     medio=(ini+fin)/2
     if(minimo3(A,ini,medio)>minimo3(A,medio+1,fin))
       return minimo3(A,ini,medio)
     else
       return minimo3(A,medio+1,fin)
```

```
int suma(int A[], int n)
    if n==0
       return A[0]
    else
       return A[n]+suma(A,n-1)
```

```
int encuentra(char *A, char car, int n)
{
    if(n==0 && A[n]!=car)
      return -1;
    else if(n==0 && A[n]==car)
      return 0;
    else if(A[n]==car)
      return n;
    else
      return encuentra(A,car,n-1)
}
```

```
int prod(int a, int b)
   if b==1
      return a
   else
      return a+prod(a,b-1)
```

```
int mcd(int a, int b)
   if a==b
      return a
   else if a>b
      return mcd(a-b,b)
   else
      return mcd(a,b-a)
```

```
binario(int n)
  if n<2
     imprime(n)
  else
     binario(n/2)
     imprime(n/2)
```

31. Encontrar el orden de complejidad temporal para la recurrencia $T(n) = 3T(\sqrt{n}) + log\ n$.

32. Utilice árboles de recursión para determinar una cota superior asintótica para la recurrencia $T(n) = 4T(\lfloor n/2 \rfloor) + cn$ con $c > 0$ constante.

33. Resolver las siguientes recurrencias mediante método de sustitución:

i) $T(n) = T(n/2) + n^2$.
ii) $T(n) = 2T(n-1) + 1$

34. Use el método maestro para resolver las siguientes recurrencias:

i) $T(n) = 2T(n/4) + 1$.
ii) $T(n) = 2T(n/4) + \sqrt{n}$.
iii) $T(n) = 2T(n/4) + n$.
iv) $T(n) = 2T(n/4) + n^2$.

35. Calcular la complejidad temporal del algoritmo de búsqueda binaria.

36. Mostrar la fórmula general de las siguientes ecuaciones de recurrencia:

i) $F(n) = 3F(n-1) + 6F(n-2)$.
ii) $F(n) = 2F(n-1) + 4F(n-2)$.

37. Resolver la recurrencia:

$$T(n) = \begin{cases} 3T(\lfloor n/4 \rfloor) + n & si \quad n > 1 \\ 1 & si \quad n = 1 \end{cases}$$

38. Resolver la recurrencia $T(n) = 2T(n/2) + nlogn$

39. Resolver la recurrencia $T(n) = T(\lceil n/2 \rceil) + 7$

40. Resolver la recurrencia

$$T(n) = \begin{cases} 1 & si \quad n = 1 \\ T(n-1) + n(n-1) & si \quad n > 1 \end{cases}$$

41. Encontrar la complejidad de la recurrencia $T(n) = T(\sqrt{n}) + 1$

42. Encontrar la complejidad de la recurrencia $T(n) = 2T(\sqrt{n}) + 1$

43. Calcular el orden de complejidad de cada uno de los siguientes algoritmos:

```
function (int n) {
        if(n <= 1)  return;
        for(int i = 1; i < n; i + +)
                printf(“ ∗ ”);
        function ( 0.8n ) ;
}
```

a)

```
void function(int n) {
        if( n < 2 ) return;
        else counter = 0;
        for i = 1 to 8 do
                function (n/2);
        for i =1 to n^3 do
                counter = counter + 1;
 }
```

b)

```
function(int n) {
        for(int i = 1 ; i <= n ; i + + )
                for(int j = 1 ; j <= n ; j * = 2 )
                        printf( “ ∗ ” );
}
```

c)

```
function(int n) {
        for(int i = 1 ; i <= n/3 ; i + + )
                for(int j = 1 ; j <= n ; j += 4 )
                        printf( “ ∗ ” );
}
```

d)

```
function(int n) {
        int i=1;
        while (i < n) {
                int j=n;
                while(j > 0)
                    j = j/2;
                i=2*i;
        } // i
}
```

e)

```
A es un array
void Binary(int n){
  if(n<1)
    printf("%s", A);
  else
     A[n-1]=0
     Binary(n-1)
     A[n-1]=1
     Binary(n-1)
}
```

f)

```
void function(int n) {
    if(n <=  1) return;
    if(n > 1) {
            printf (" ∗ ");
            function( n/2 );
            function( n/2 );
      }
}
```

g)