Instituto Politecnico Nacional

ESCOM “ESCUELA SUPERIOR DE CÓMPUTO”

ANÁLISIS DE ALGORITMOS

COMPLEJIDAD DE LOS ALGORITMOS

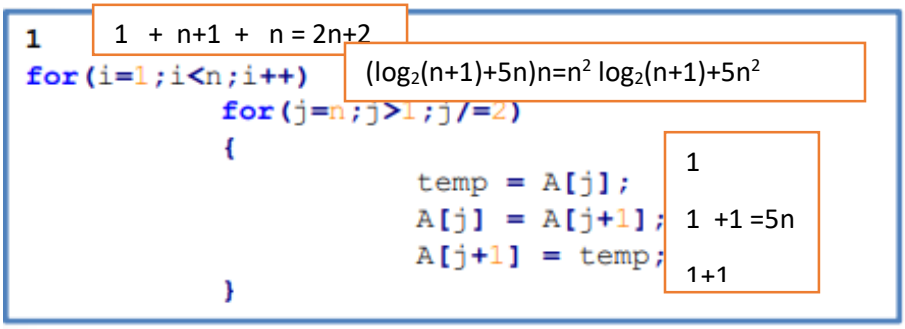
PROFE: Edgardo Adrian Franco Martínez

ALUMMNO: Rojas Alvarado Luis Enrique

GRUPO: 3CM4

Para los siguientes 5 algoritmos se determinan la función de complejidad temporal y espacial en términos de n. Se va a considerar las operaciones de: asignación, aritméticas y condicionales.

PARTE 1: FUNCION COMPLEJIDAD ESPACIAL Y TEMPORAL



n A[j]
i
j
temp

4

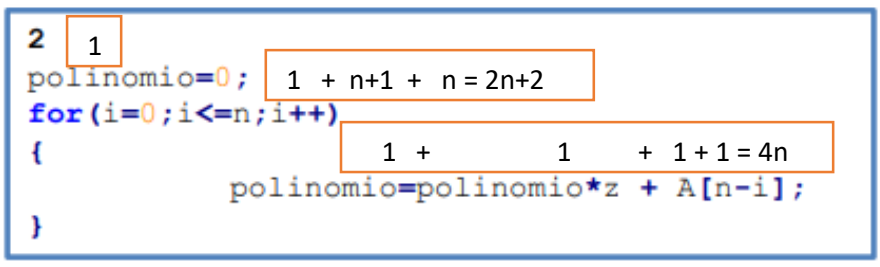
4n

$F_t(n)= n^2 \log_2(n+1)+5n^2+2n+2=n^2(\log_2(n+1)+5)+2n+2$

$F_e(n)=4n+4$

Se analiza de abajo hacia arriba, contando las 3 asignaciones que se hacen del arreglo a la variable temp, más 2 operaciones que se hacen en la dimensión del arreglo, esto multiplicado por n veces que entrará en el ciclo. Posteriormente se cuenta en el ciclo for una asignación, y la inicialización en n, vemos que los incrementos están dividiéndose entre 2, por lo que se vuelve una logarítmica de base 2, multiplicado por n veces que se entrará en el ciclo y por esto queda $n^2 \log_2(n+1)+5n^2$, finalmente se vuelve a tomar 1 asignación del ciclo for y una comparación n+1 por que hará n comparaciones y al final hará 1 más para cuando no se cumpla la condición, y se repetirá n veces. Al final se suma lo anterior y se obtiene que la función temporal es: $n^2(\log_2(n+1)+5)+2n+2$

Para la función espacial, se toman en cuenta las variables que participan en todo el algoritmo a analizar (4) y como en el arreglo se está usando sólo una posición en una variable, y se hacen 4 asignaciónes con éste arreglo unidimensional, se puede decir que su complejidad espacial es de 4n+4.



n A[n+1]
i
z
polinomio

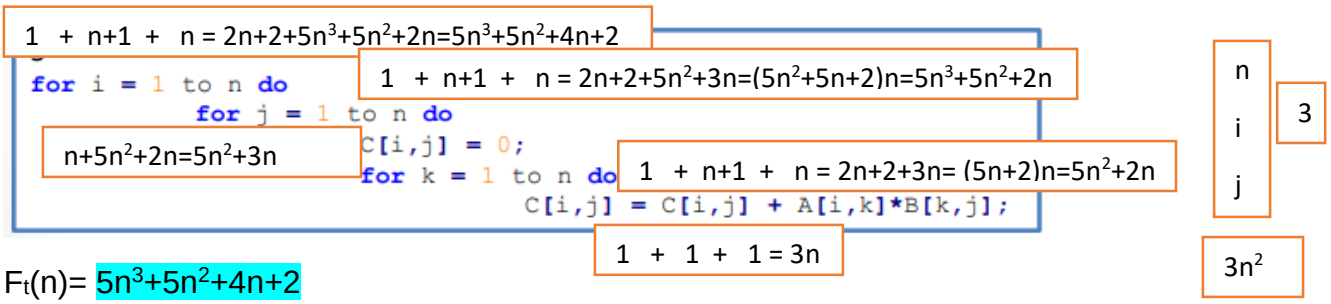
4

n

$F_t(n)= 4n+2n+2+1=6n+3$

$F_e(n)=n+4$

Analizando de dentro hacia afuera, se tienen 3 operaciones aritméticas incluyendo la que se hace en el arreglo, y 1 asignación y multiplicado por las veces que se repetirá el ciclo (n) queda 4n. Analizando el ciclo for, una asignación para la variable i, una condicional para que se pueda entrar al ciclo (n+1) y un incremento n, por lo que la suma queda 2n+2 y sumando las instrucciones dentro del for como las que se le asignan al propio ciclo, adicionando la asignación que está hasta arriba del programa: 4n+2n+2+1=6n+3

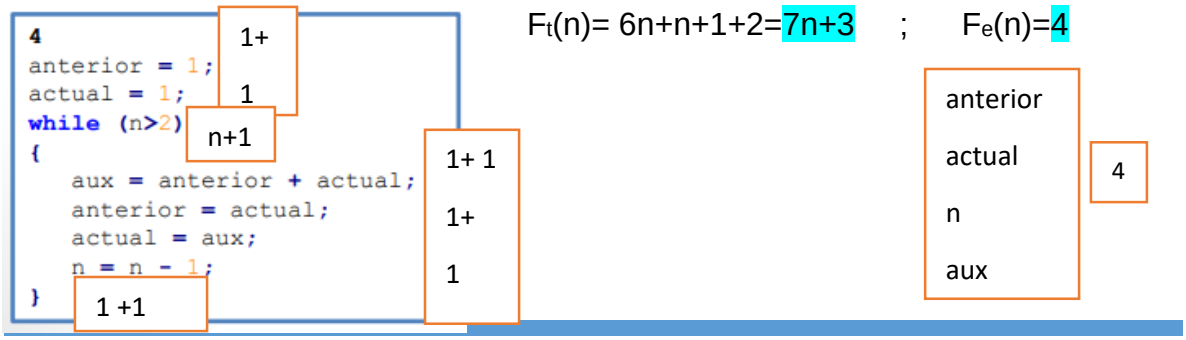


$F_t(n)= 5n^3+5n^2+4n+2$

$F_e(n)=3n^2+3$

Analizando desde abajo hacia arriba, existen 1 asignación y 2 operaciones aritméticas multiplicado por las veces que se repetirá el ciclo (n), lo que nos da 3n, siguiendo hacia arriba, se puede ver en el seudocódigo que hay una asignación y una condición que se repetirá n+1 y en n incrementos, sumando lo anterior queda: 2n+2 y sumando con lo que está dentro de su ciclo: $2n+2+3n= (5n+2)n=5n^2+2n$, se multiplica por n puesto que se repetirá n veces que entra al ciclo. Posteriormente se tiene una simple asignación multiplicada por las veces que se repite ese ciclo (n) y sumando lo anterior obtenido queda: $n+5n^2+2n=5n^2+3n$. Hacemos lo mismo para el for que está conteniendo todo lo anterior y vemos que tiene una asignación, una comparación n+1 y n incrementos si se suman con todo lo anterior obtenido$2n+2+5n^2+3n= (5n^2+5n+2)n=5n^3+5n^2+2n$, multiplicado por n porque entrará n veces. Y hacemos lo mismo para el último for, sumando lo que ya teníamos anteriormente para su asignación, condición n+1 y n incrementos. Para que al final quede $1 + n+1 + n = 2n+2+5n^3+5n^2+2n=5n^3+5n^2+4n+2$.

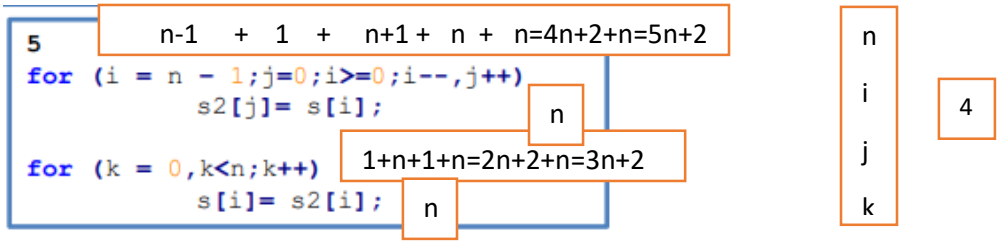
En cuanto la complejidad espacial se toman en cuenta las variables que interactúan en todo el programa siendo 3, sólo n, i y j, y ya que tenemos los arreglos A, B, C, son de n*n, se puede deducir que la función es $3n^2+3$



$F_t(n)= 6n+n+1+2=7n+3$; $F_e(n)=4$

Para éste algoritmo se analiza desde dentro hasta afuera, teniendo en la parte de abajo, una asignación, y una asignación y una operación aritmética, seguido de 2 asignaciones y al final una asignación y una operación aritmética, sumando tenemos que son 6 instrucciones multiplicadas por n veces que se entrará a ese ciclo. En la parte de la condición del ciclo while tenemos que se entrará n+1 veces (de igual manera que con las condiciones del ciclo for) y sumando se tiene que: 6n+n+1 y sumando las 2 asignaciones que están al inicio del programa queda 7n+3.

En la función espacial, se toman las 4 variables que interactúan con el algoritmo (4), y como no hay un arreglo de n dimensiones, se queda la función como 4.



4

$F_t(n)= 3n+2+5n+2=8n+4$

S1[n]+s[n]=n

$F_e(n)=2n+4$

Para éste algoritmo se tiene que son 2 for separados, por lo que solo se van a sumar sin multiplicarse por n como anteriormente se hizo. En el primer for se nota que va contando en reversa por así decirlo, pero su complejidad es la mismaEn el segundo ciclo for se tiene una asignación dentro de él, y en sus condiciones se tiene una asignación y las veces que se espera que se cumpla el ciclo que es n+1 y su n incremento. Al sumarse se tiene que es 3n+2+4n+4=7n+6.

Para la función espacial se colocan las 4 variables que interactúan dentro del algoritmo, y como es de orden n se tiene que la función es igual a n+4.

En los algoritmos que se usan arreglos, no se tomaron en cuenta operaciones dentro del arreglo puesto que no está especificado que se tome para el análisis de éstos.

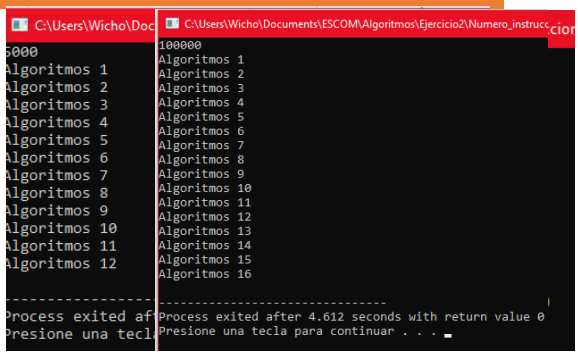
PARTE II: NÚMERO DE INSTRUCCIONES

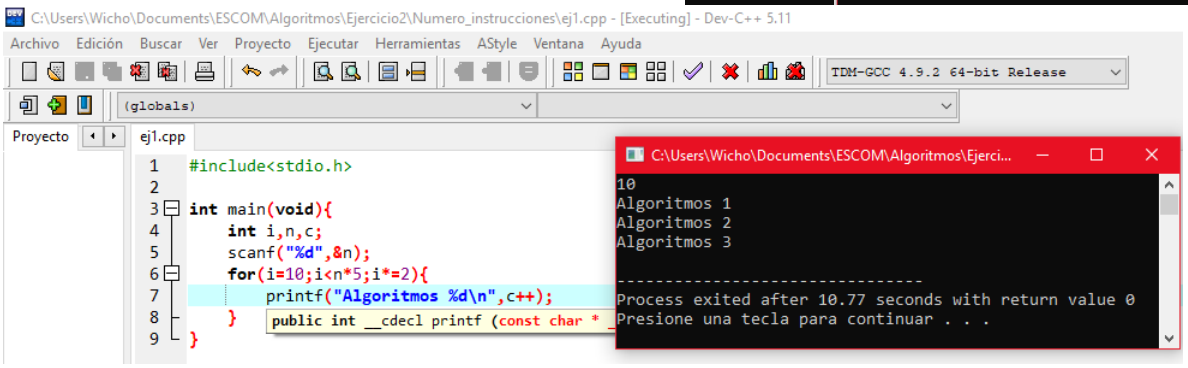
```
6
for(i=10;i<n*5;i*=2)
        printf("\"Algoritmos\n\"");
```

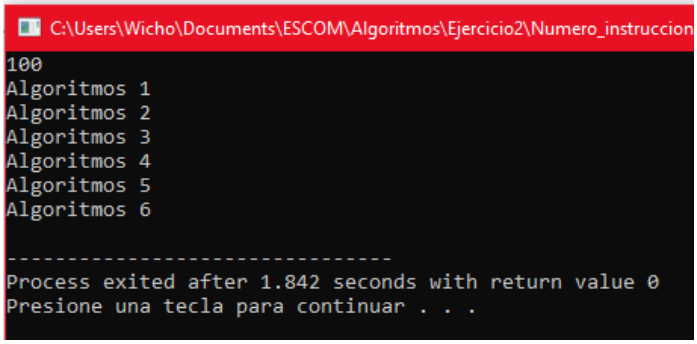
$F(n)=\text{Log}_2(\frac{\widetilde{n*5}}{10})$

n	#impresiones reales	#impresiones F(n)
10	3	2.3≈3

100	6	5.6≈6
1,000	9	8.9≈9
5,000	12	11.28≈12
100,000	16	15.6≈16





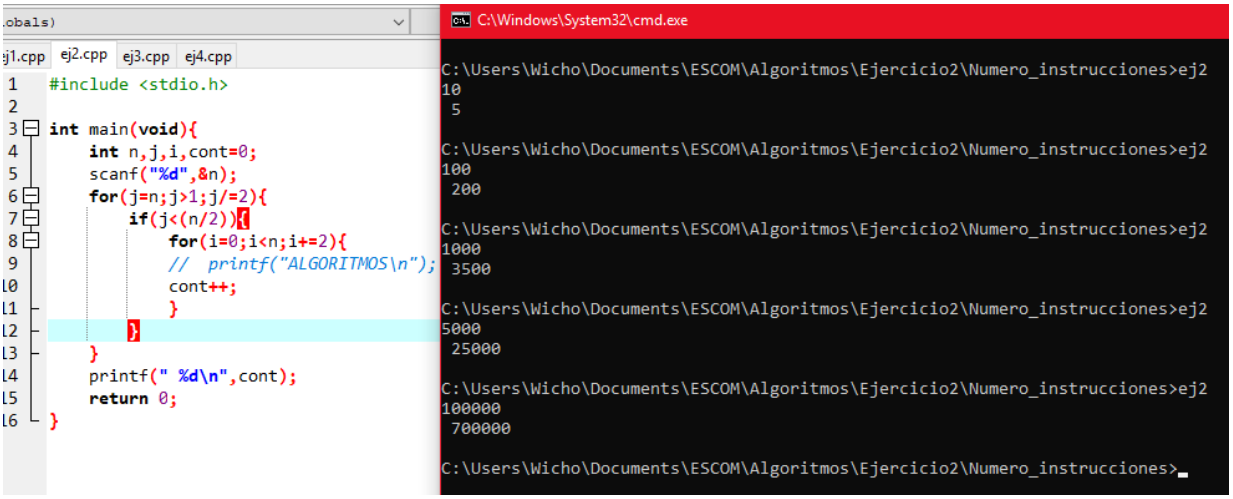


En el análisis de éste algoritmo se dedujo que los incrementos serán de forma exponencial de la forma 2^{n*5} pero para que sea más claro los incrementos se tomó como $\log_6(n*5)$, y como el índice empieza en 10, se tiene que compensar dividiendo la condición entre éste número para obtener su incremento. Por lo que por eso al final queda $\text{Log}_2(\frac{n*5}{10})$ y se le pone techo para que tome el siguiente entero.

```
7
for(j=n;j>1;j/=2)
{
        if(j<(n/2))
        {
                for(i=0;i<n;i+=2)
                {
                        printf("\"Algoritmos\n\"");
                }
        }
}
```

$F(n)= (\frac{n}{2} - 1)Log_2(\frac{n}{4})$

n	#impresiones reales	#impresiones F(n)
10	5	5.28≈5
100	200	227.5≈227
1,000	3500	3,974.9≈3,974
5,000	25,000	25,708.9≈25,708
100,000	700,000	730,467.4≈730467



Analizando de dentro hacia afuera, el ciclo for que imprime la palabra algoritmos se repite $\frac{n}{2}$ debido a que tiene incrementos de 2 en 2, y teniendo en cuenta que en el condicional if se repetirá las mismas veces, menos una (por que habrá una vez que no se cumpla) sólo se resta esa única instrucción al número de instrucciones hechas por el for que está dentro del condicional. Al final podemos ver que en el ciclo for que está al inicio, la función que se genera es una logarítmica de base 2 si analizamos el tamaño del problema de atrás hacia adelante (a la inversa de que si hubiera incrementos como: i*=2), si analizamos que la primera vez que entra el for no entra al condicional y por lo tanto hay que dividirlo entre 2 cada iteración, se puede decir que la primera vez siempre se va a dividir la n entre 2 para que entre

en el if, entonces en el argumento del algoritmo la n se divide entre 4. Por lo tanto la aproximación queda como: F(n)= $(\frac{n}{2}-1)Log_2(\frac{n}{4})$ y aun así no es exacta.

```
8
i=n;
while(i>=0)
        {
                for (j=n;i<j;i-=2,j/=2)
                {

                printf("\"Algoritmos\n\"");
                }
        }
        return 0;
```

F(n)=0

n	#impresiones reales	#impresiones F(n)
10	0	0
100	0	0
1,000	0	0
5,000	0	0
100,000	0	0

ej1.cpp | ej2.cpp | ej3.cpp | ej4.cpp

```
1  #include <stdio.h>
2
3  int main(void){
4      int n,j,i,cont=0;
5      scanf("%d",&n);
6      i=n;
7      while(i>=0){
8          for(j=n;i<j;i-=2,j/=2){
9              printf("ALGORITMOS %d\n",cont++);
10         }
11     }
12     return 0;
13 }
```

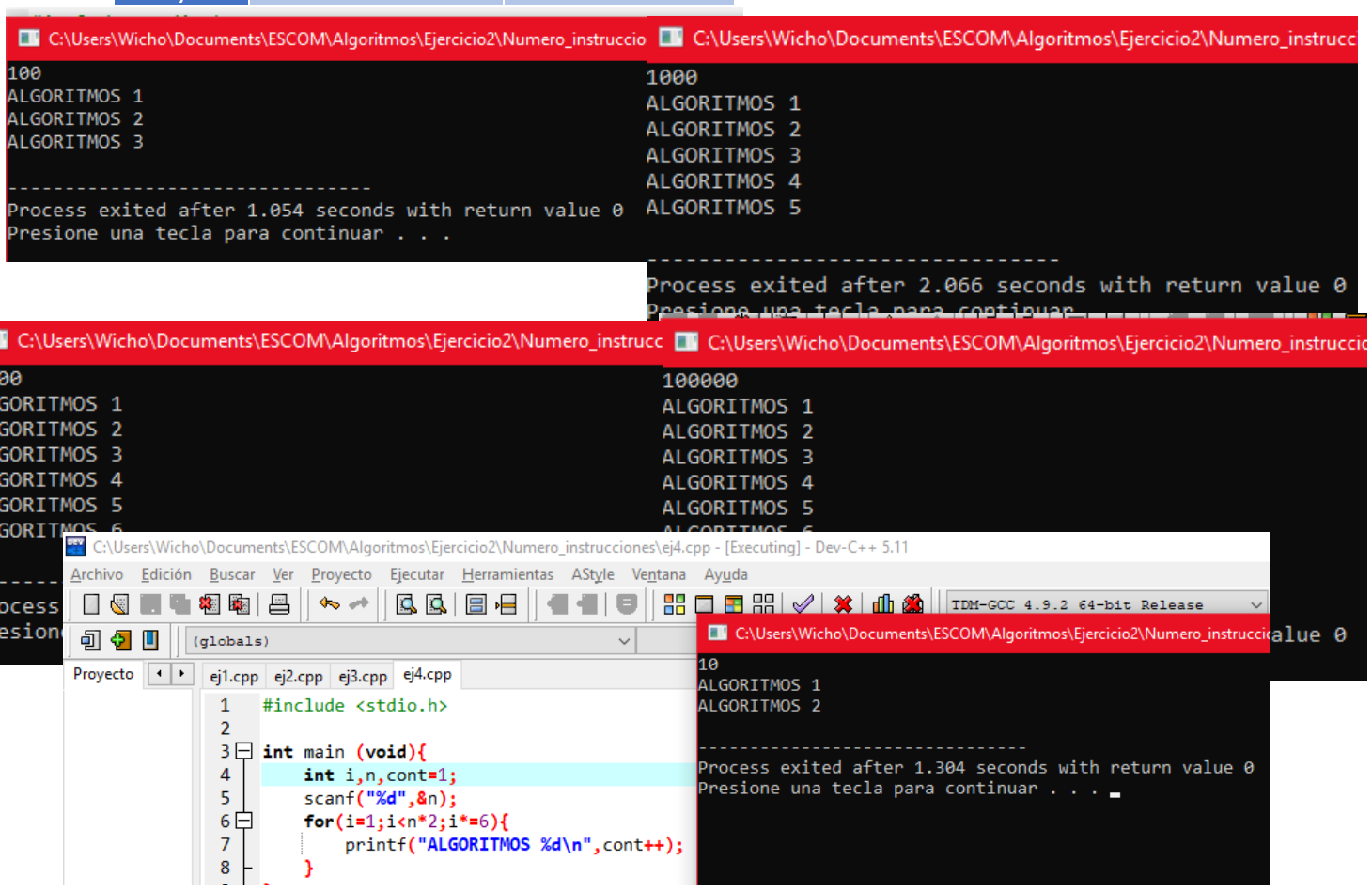
```
C:\Windows\System32\cmd.exe
Microsoft Windows [Versión 10.0.16299.967]
(c) 2017 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Wicho\Documents\ESCOM\Algoritmos\Ejercicio2\Numero_instrucciones>ej3
10
^C
C:\Users\Wicho\Documents\ESCOM\Algoritmos\Ejercicio2\Numero_instrucciones>ej3
100
^C
C:\Users\Wicho\Documents\ESCOM\Algoritmos\Ejercicio2\Numero_instrucciones>ej3
1000
^C
C:\Users\Wicho\Documents\ESCOM\Algoritmos\Ejercicio2\Numero_instrucciones>ej3
5000
^C
C:\Users\Wicho\Documents\ESCOM\Algoritmos\Ejercicio2\Numero_instrucciones>ej3
100000
^C
C:\Users\Wicho\Documents\ESCOM\Algoritmos\Ejercicio2\Numero_instrucciones>
```

Por lo que se puede observar, nunca entra en el for que está dentro del while y por lo tanto nunca imprimirá la palabra Algoritmos. Dado que es un ciclo infinito se tiene que interrumpir al momento de la ejecución del programa, ya que si lo dejas ejecutar, jamás podrá salir de ese ciclo.

```
9
for (i =1;i<n*2;i*=6)
    printf("\"Algoritmos\n\"");
```

$F(n)=\lceil Log_6(n*2) \rceil$

n	#impresiones reales	#impresiones F(n)
10	2	1.6≈2
100	3	2.9≈3
1,000	5	4.2≈5
5,000	6	5.1≈6
100,000	7	6.8≈7



De la misma manera que en el primer problema se tiene que el incremento es de 2^{n*2} y para que quede más claro se coloca en su forma logarítmica con $Log_2(n*2)$ como el índice empieza en 1 no se necesita dividir para obtener sus incremento real, y se le pone techo para obtener el siguiente entero.

PARTE III: FUNCIÓN COMPLEJIDAD TEMPORAL PARA EL MEJOR, PEOR Y CASO MEDIO.

```
10
func Producto2Mayores(A,n)
if(A[1] > A[2])
            mayor1 = A[1];
            mayor2 = A[2];
else
            mayor1 = A[2];
            mayor2 = A[1];
i = 3;

while(i<=n)
            if(A[i] > mayor1)
                        mayor2 = mayor1
                        mayor1 = A[i];
            else if (A[i] > mayor2)
                        mayor2 = A[i];
            i = i + 1;

return = mayor1 * mayor2;
fin
```

3 Operaciones si tomamos éste camino

3 Operaciones si tomamos éste camino

n-2 repeticiones si en los caminos anteriores encuentra los números en los primeros 2.

Y 3 operaciones si se toma éste camino.

3 Operaciones si tomamos éste camino

2 Operaciones para retornar el valor

Para el análisis se usaron como operaciones básicas las comparaciones entre elementos del arreglo y asignaciones a mayor1 y mayor2.

- Mejor caso: Se da cuando los números se encuentran en la primera pasada uno seguido de otro y así tendremos 3 operaciones iniciales y sumadas con la multiplicación de 2 instrucciones para retornar el valor y como se empieza en n-2 puesto que la inicialización de i=3, por lo que 3+2(n-2) dentro del while. Y la complejidad para el mejor caso es 2n-1.
- Peor caso: Cuando el arreglo está ordenado de menor a mayor. Entonces se tienen 3 operaciones iniciales multiplicadas por n-2 por la inicialización de i en 3 y siempre se va a cumplir que A[i]>mayor1, por lo que la función para el peor caso: $F_n(t)=3+3(n-2)=3n-3$.
- Caso medio: hay 3 caminos posibles, si vamos por el primer if $3+3(n-2)=3n-3$ y es la misma función para cuando vallamos por el segundo if, pero si no entramos a ningún if $3+2(n-2)=2n-1$ y asumiendo que los 3 casos son igual de probables, la complejidad para el caso medio quedaría como: $F_n(t)=\frac{1}{3}(3n-3)+\frac{1}{3}(3n-3)+\frac{1}{3}(2n-1)=\frac{8n-7}{3}$

```
11
func OrdenamientoIntercambio(a,n)
for (i=0; i<n-1; i++)
            for (int j=i+1; j<n;j++)
                        if (a[ j ] < a[ i ])
                        {
                                    temp=a[ i ];
                                    a[ i ]=a[ j ];
                                    a[ j ]=temp;
                        }
fin
```

Éste for se ejecuta n-1 veces

El siguiente ciclo n-i-1

4 instrucciones dentro del if

1 instrucción si no se entra al if

Se usaron como operaciones básicas las comparaciones entre elementos del arreglo A, asignaciones a temp y al arreglo A.

- Mejor caso: El arreglo está ordenado ascendentemente y el condicional nunca se ejecuta. Por lo que su función es: $F_n(t)= \frac{n2-n}{2}$
- Peor caso: Se da cuando el arreglo está ordenado descendentemente, por lo que el condicional siempre se ejecuta. Así, la complejidad temporal del peor caso es 4 veces la del mejor caso: $ft(n) = 2n^2-2n$
- Caso medio: dentro del condicional podemos decidir si ejecutar las líneas que asignan a temp y el arreglo o no. Pero ambas opciones son igual de probables. Por lo que $ft(n)= \frac{5n2-5n}{2}$

```
12
func MaximoComunDivisor(m, n)
{
            a=max(n,m);
            b=min(n,m);
            residuo=1;
            mientras (residuo > 0)
            {
                        residuo=a mod b;
                        a=b;
                        b=residuo;
            }
            MaximoComunDivisor=a;
            return MaximoComunDivisor;
}
```

*Recomendación: Usar como operación básica la operación de modulo de a y b

*Notar de dos números consecutivos de la serie de Fibonacci. ¿Qué propiedad tiene?

- Mejor caso: cuando n=0, así nunca entramos al ciclo while, por lo que la complejidad ft(n)=0.
- Peor caso: Cuando m y n son 2 numeros consecutivos de la serie de Fibonacci, es decir: Fk+1 y n= fk para alguna k perteneciente a N (números naturales). Por definición sabemos que Fk+1 = Fk + Fk-1 y 0 <= Fk-1 < Fk, de esa forma el residuo de dividir Fk+1 entre Fk será Fk-1 y el cociente será 1 en cada iteración. Entonces, estamos transformando (Fk+1; Fk) ! (Fk; Fk-1), los cuales siguen siendo números de Fibonacci consecutivos. Como el

cociente es al menos 1 para cualquier entrada y con esta entrada siempre obtenemos 1, estamos reduciendo al mínimo los dos números en cada iteración, obteniendo el peor caso. Así, nos tardaremos k divisiones llegar a que Fk = 0, y como $Fk \approx \phi^k$ la complejidad temporal aproximada del peor caso será $ft(n) \approx \log_\phi(n)$, donde $\phi = \frac{1+\sqrt{5}}{2}$.

- Caso medio: Si m y n no son números consecutivos de Fibonacci, al menos uno de los cocientes obtenidos será mayor a uno. Informalmente, podemos aumentar la base del logaritmo, de ϕ a 2, y decir que la complejidad temporal del caso medio es $ft(n) \approx \log_2(n)$.

```
13
Procedimiento BurbujaOptimizada(A,n)
        cambios = "No"
        i=0
        Mientras i< n-1 && cambios != "No" hacer
                cambios = "No"
                Para j=0 hasta (n-2)-i hacer
                        Si(A[i] < A[j]) hacer
                                aux = A[j]
                                A[j] = A[i]
                                A[i] = aux
                                cambios = "Si"
                        FinSi
                FinPara
                 i= i+1
        FinMientras
fin Procedimiento
```

El while se ejecutará a lo mucho n-1 veces

El for se ejecutará n-i-1 veces

4 instrucciones si se cumple el condicional

1 instrucción si no se cumple el condicional

Se usaron como operaciones básicas las comparaciones entre elementos del arreglo A, asignaciones a aux y al arreglo A.

- Mejor caso: Se da cuando A está ordenado ascendentemente, pues nunca entraremos al if y entraremos al while 1 vez pues la variable de cambios queda en “No”. Por lo que la complejidad del mejor caso es igual a $f_t(n)=n-1$.
- Peor caso: Cuando A está ordenado descendentemente. En cada while habrá un cambio, por lo tanto, la complejidad es: $f_t(n)=2n^2-2n$.
- Caso medio: Sólo tenemos 2 opciones en el if, entonces la complejidad temporal queda como: $f_t(n)=\frac{5n2-5n}{2}$

```
14
Procedimiento BurbujaSimple(A,n)
        para i=0 hasta n-2 hacer
                para j=0 hasta (n-2)-i hacer
                        si (A[j]>A[j+1]) entonces
                                aux = A[j]
                                A[j] = A[j+1]
                                A[j+1] = aux
                        fin si
                fin para
        fin para
fin Procedimiento
```

El for se ejecutará a lo mucho n-1 veces

El for se ejecutará a lo n-i-1

4 instrucciones si se entra al if

1 instrucción si no entra

Se usaron como operaciones básicas las comparaciones entre elementos del arreglo A, asignaciones a aux y al arreglo A.

- Mejor caso: Cuando está ordenado ascendentemente y el if se cumplirá como: $f_t(n)=\frac{n2-n}{2}$
- Peor caso:ocurre cuando A está ordenado descendentemente. Esto implica que el if siempre se cumplirá, causando que se ejecuten las 4 instrucciones. Entonces se multiplica por 4 la complejidad anterior. $f_t(n)=2n^2-2$
- Caso medio: será el promedio de las 2 anteriores: $f_t(n)=\frac{5}{2}n2-\frac{5}{2}n$

```
15
Procedimiento Ordena(a,b,c)
{
        if(a>b)
                if(a>c)
                        if(b>c)
                                salida(a,b,c);
                        else
                                salida(a,c,b);
                else
                        salida(c,a,b);
        else
                if(b>c)
                        if(a>c)
                                salida(b,a,c);
                        else
                                salida(b,c,a);
                else
                        salida(c,b,a);
}
```

3 ins
3 ins
2 ins
3 ins
3 ins
2 ins

Se usaron como operaciones básicas las comparaciones entre a,b,c

- Mejor caso: cuando solo entra en el primer if o en el primer else. Por lo que la función simplemente es : $f_t(n)=2$.
- Peor caso: Cuando se cumplen todos los condicionales posibles. Por lo que la función queda: $f_t(n)=3$.
- Caso medio: Como todas las salidas de la función tienen la misma probabilidad $\frac{1}{6}$ la complejidad es: $f_t(n)=\frac{3*4+2*2}{6}=\frac{8}{3}$.

```
16
Procedimiento Seleccion(A,n)
        para k=0 hasta n-2 hacer
                p=k
                para i=k+1 hasta n-1 hacer
                        si A[i]<A[p] entonces
                                p=i
                        fin si
                fin para
                temp = A[p]
                A[p] = A[k]
                A[k] = temp
        fin para
fin Procedimiento
```

n-1 veces para el for
n-k-1 veces para el for
1 operacion si se cumple
1 operacion si no se cumple
3 operaciones

Usaremos como operaciones básicas las comparaciones entre elementos del arrelgo A a temp y al arreglo A.

- Mejor caso: Ocurre cuando A está ordenado ascendentemente. Esto implica que la condición if nunca se cumple y teniendo 4 operaciones en el for interno, la complejidad es: $f_t(n)=\frac{n2+5n-6}{2}$
- Peor caso: Cuando A está ordenado descendentemente. Esto implica que la condición if siempre se cumple, pero como no se cuenta la asignación de p=i como operación básica tenemos exactamente la misma complejidad que la anterior $f_t(n)=\frac{n2+5n-6}{2}$
- El caso medio: Como la complejidad del peor caso es la misma que la del mejor caso, la del medio tendrá que ser la misma: $f_t(n)=\frac{n2+5n-6}{2}$