Instituto Politécnico Nacional
Escuela Superior de Computo

Ejercicio 07

"Análisis de Algoritmos No Recursivos"

Mora Ayala José Antonio

Análisis de Algoritmos

INSTRUCCIONES

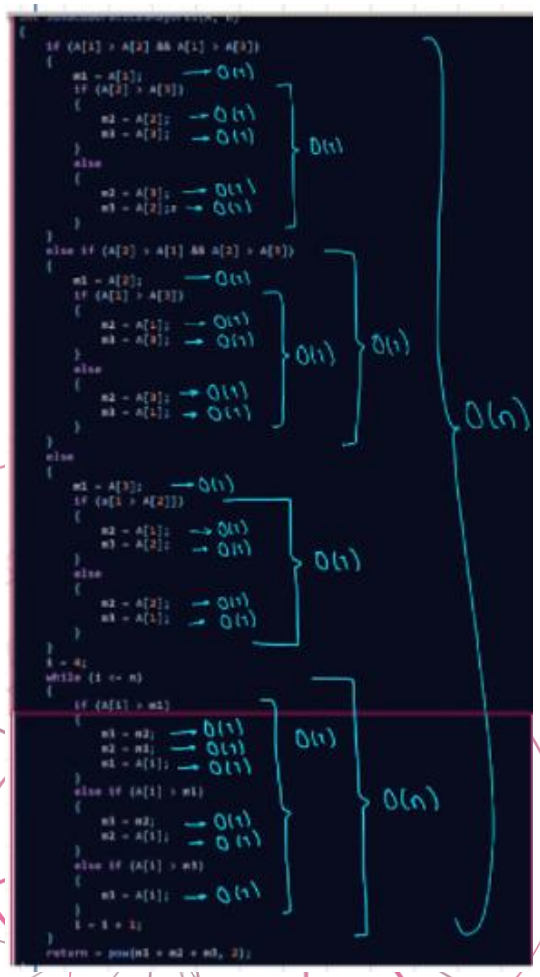
Para los siguientes algoritmos determine la cota de complejidad O (cota superior ajustada) bajo el principio del peor caso, de cada algoritmo. Indique a detalle el procedimiento de obtención de la cota.

CONTENIDO

EJERCICIO 1

```
int SumaCuadratica3Mayores(A, n)
{
    if (A[1] > A[2] && A[1] > A[3])
    {
        m1 = A[1];
        if (A[2] > A[3])
        {
            m2 = A[2];
            m3 = A[3];
        }
        else
        {
            m2 = A[3];
            m3 = A[2];z
        }
    }
    else if (A[2] > A[1] && A[2] > A[3])
    {
        m1 = A[2];
        if (A[1] > A[3])
        {
            m2 = A[1];
            m3 = A[3];
        }
        else
        {
            m2 = A[3];
            m3 = A[1];
        }
    }
    else
    {
        m1 = A[3];
        if (a[1 > A[2]])
        {
            m2 = A[1];
            m3 = A[2];
        }
        else
        {
            m2 = A[2];
            m3 = A[1];
        }
    }
    i = 4;
    while (i <= n)
    {
        if (A[i] > m1)
        {
```

```
            m3 = m2;
            m2 = m1;
            m1 = A[i];
        }
        else if (A[i] > m1)
        {
            m3 = m2;
            m2 = A[i];
        }
        else if (A[i] > m3)
        {
            m3 = A[i];
        }
        i = i + 1;
    }
    return = pow(m1 + m2 + m3, 2);
}
```



Para comenzar con el análisis de este código debemos partir de las operaciones primitivas que podemos encontrar dentro de el, las cuales no son difíciles de determinar, pues este se basa mayormente en las asignaciones o returns

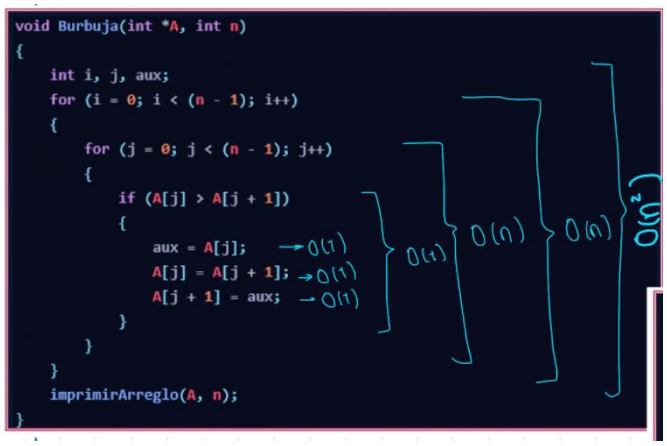
Dado que en la mayoría de los condicionales podemos observar este caso podemos determinar que son de orden constante contando con $O(1)$, siendo este su grado de complejidad, dado que se encuentran dentro de una comparativa y esta es considerada del mismo grado de complejidad, podremos determinar que el bloque completo es de orden O(1)

Como ya tenemos los bloques que están anidados dentro del if principal ahora procederemos a obtener la cota de este bloque, como todos los bloques son de orden O(1) y tendemos que elegir el de mayor orden, no hay problema en este caso todos son del mismo orden así que todo el bloque será de orden O(1)

Finalmente llegamos a la parte de la ejecución del while, el cual internamente presenta la misma situación que ya se ha descrito con anterioridad con respecto a las asignaciones y comparaciones, contando con un orden O(1), pero al observar el comportamiento de este podemos observar que es de orden O(n) y como el orden del algoritmo esta determinado por el mayor orden encontrado este será: $\boldsymbol{O(n)}$

EJERCICIO 2

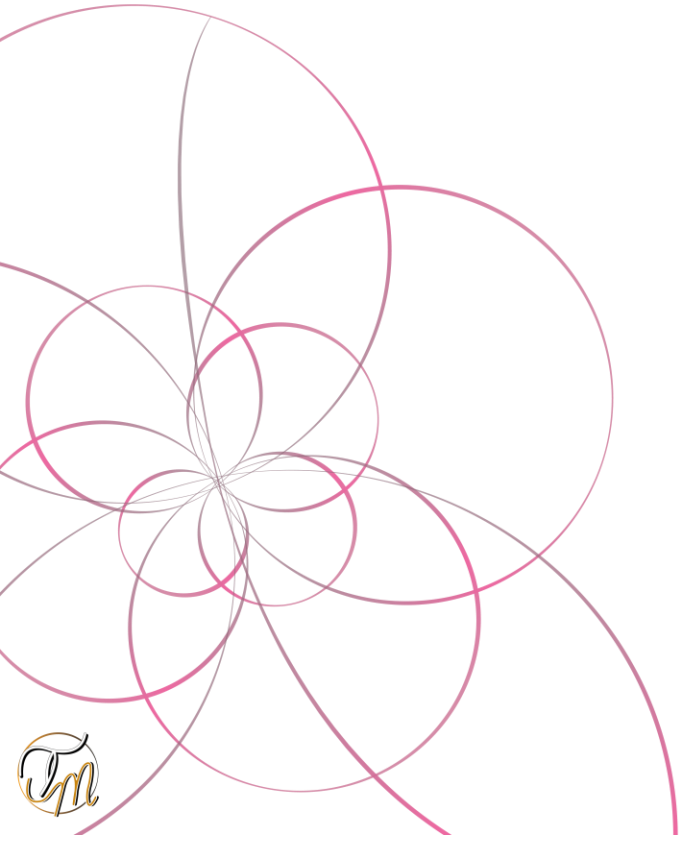
```
void Burbuja(int *A, int n)
{
    int i, j, aux;
    for (i = 0; i < (n - 1); i++)
    {
        for (j = i+1; j < (n ); j++)
        {
            if (A[j] < A[i])
            {
                aux = A[i];
                A[i] = A[j];
                A[j] = aux;
            }
        }
    }
}
```



En este algoritmo de ordenamiento, que es bastante conocido sabemos de antemano que por el comportamiento que este tiene y la forma en que este compuesto (dos bucles anidados) la cota que lo representa es de n^2 , pues analizando desde dentro podemos ver simples asignaciones y 1 operación aritmética, las cuales están consideradas con complejidad $O(1)$, por lo cual ya tenemos la complejidad del primer bloque que se encuentra en los mas inetrno del seguido ciclo for, hacemos una comparación que posee el mismo orden de complejidad y nos proporcionará un bloque en su totalidad de orden O(1), ahora el segundo bucle observamos que depende directamente de n, pues se realizará hasta que el valor sea menor a n iniciando en el valor de i+1, por lo cual depende mayormente de n, esto proporciona una complejidad lineal, ósea O(n), pues el incremento de la variable es lineal (incrementa de 1 en

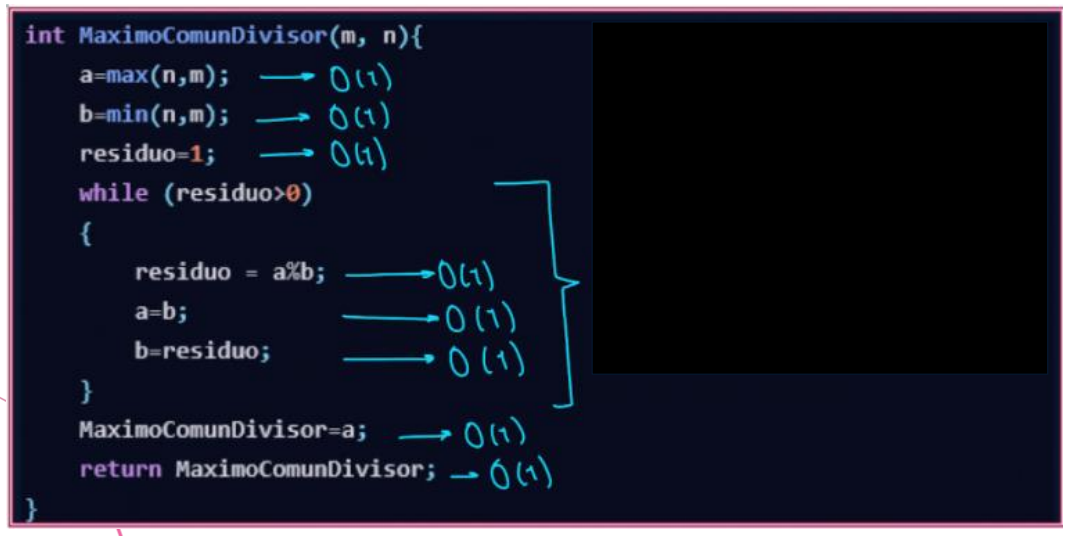
1), teniendo la complejidad del segundo bucle, solo nos queda determinar la complejidad del primero el cual presenta la misma situación, presentando la complejidad de O(n), finalmente multiplicamos estos niveles de complejidad y tendremos como resultado $n * n = n^2$

$$O(n^2)$$



EJERCICIO 3

```
int MaximoComunDivisor(m, n){
    a=max(n,m);
    b=min(n,m);
    residuo=1;
    while (residuo>0)
    {
        residuo = a%b;
        a=b;
        b=residuo;
    }
    MaximoComunDivisor=a;
    return MaximoComunDivisor;
}
```



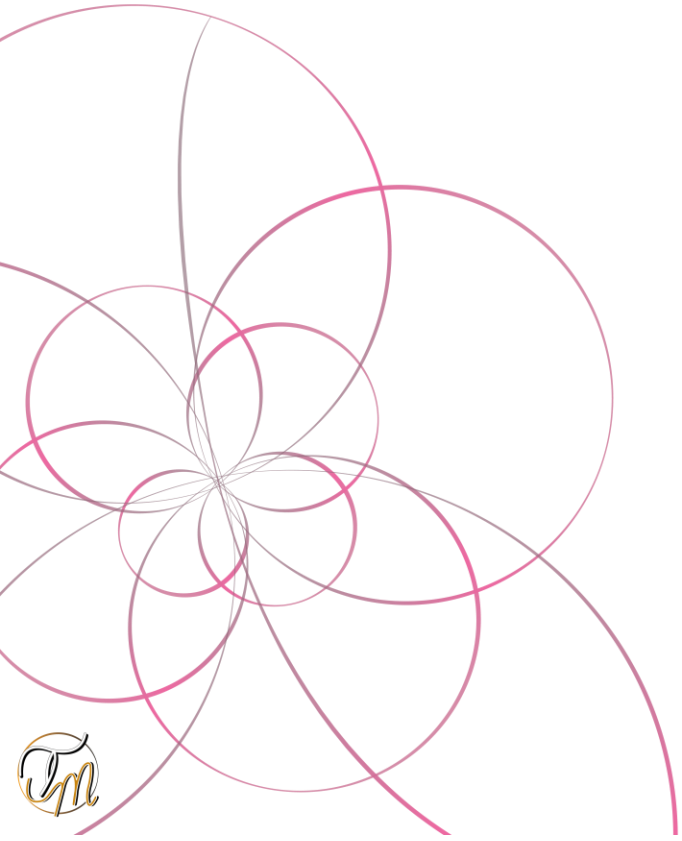
Como podemos observar tenemos varías asignaciones dentro de esta función, no contamos con recursividad o ciclos for anidados, pero podemos observar que las iteraciones que se realicen del bucle while, para este ciclo primero analizaremos las operaciones que tiene dentro, como podemos ver únicamente tenemos asignaciones tanto a las los valores de a, b y residuo, también tenemos la operación modulo, estas 3 líneas son primitivas por lo tanto son O(1)

Ya que nos estamos basando en el peor caso tenemos que para obtener el mayor número de operaciones modulo, o en otras palabras el número de veces donde mas se ejecuta el ciclo es cuando tenemos 2 números sucesivos de la serie de Fibonacci, entonces para obtener el total de número de operaciones modulo está dada por la formula siguiente:

$$f(a) = \frac{loga}{log\Phi} \; donde \; a \geq b \; y \; \phi \; 1 + \sqrt{52}$$

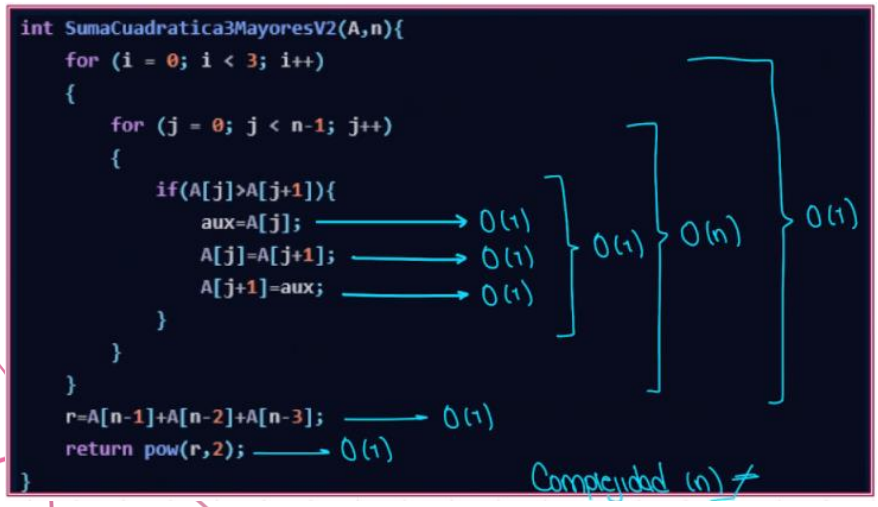
La fórmula nos dice que dado el numero a y b donde ambos son sucesivos de la serie de Fibonacci además de que el numero a es mayor a b, tenemos que la división entre el logaritmo de este numero de a entre el logaritmo de phi donde es mejor conocido por la formula de Binet definido como el rango de oro tenemos una función que a primera vista pensamos que es logarítmica pero lo cierto es que no, ya que estamos dividiendo un logaritmo entre otro, esto nos da una función de complejidad lineal por lo tanto, la cota de nuestro bloque while para el peor caso es de orden O(n)

$$O(n)$$



EJERCICIO 4

```
int SumaCuadratica3MayoresV2(A,n){
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < n-1; j++)
        {
            if(A[j]>A[j+1]){
                aux=A[j];
                A[j]=A[j+1];
                A[j+1]=aux;
            }
        }
    }
    r=A[n-1]+A[n-2]+A[n-3];
    return pow(r,2);
}
```



Para este algoritmo como en el resto procedemos a determinar las complejidades desde lo mas interno a lo mas externo, teniendo en cuenta las operaciones primitivas de asignación y las aritméticas que posee un grado de complejidad determinado por O(1) anidadas dentro de una comparativa que posee el mismo costo, tenemos como resultado que todo este primer bloque proporciona una complejidad O(1).

Ahora pasando a un nivel mas externo, tenemos el bucle for, el cual ira hasta n-1 con un incremento lineal de 1 en 1, por lo cual podemos observar que el limite de este se ve directamente afectado por el valor que tome n, por lo cual la complejidad que habrá aquí es de

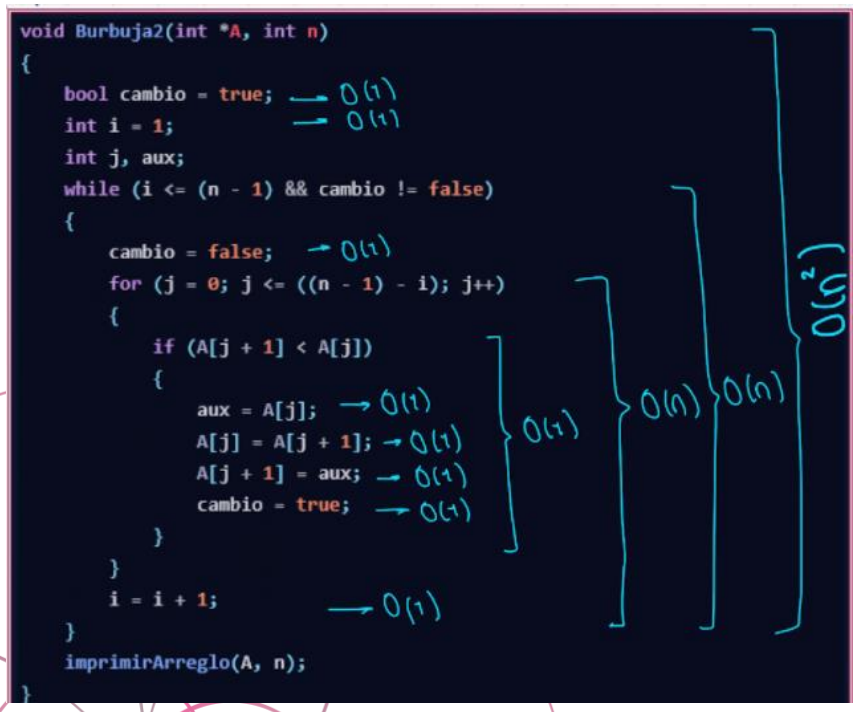
un orden n O(n), finalmente en el nivel mas externo tenemos el bucle i, el cual podríamos ensar que determinará otr complejidad de tipo n, pero no es así, pues tenemos que observar que el limite de este será cuando sea <3, lo cual no esta nunca determinado por n, de igual forma presenta un incremento de 1 en 1, por lo que la complejidad será O(1)

Este algoritmo puede verse de cierta forma algo engañoso, pues observamos un algoritmo que posee dos bucles anidados, por lo cual podríamos inferir de forma inmediata que será de una complejidad n^2, lo cual es incorrecto, pues debemos detenernos a ver como es que se comporta el primer bucle, y observamos que este tiene como límite el numero 3 no n iteraciones, por lo cual finalmente tendremos una complejidad de:

$$O(n)$$

EJERCICIO 5

```
void Burbuja2(int *A, int n)
{
    bool cambio = true;
    int i = 1;
    int j, aux;
    while (i <= (n - 1) && cambio != false)
    {
        cambio = false;
        for (j = 0; j <= ((n - 1) - i); j++)
        {
            if (A[j + 1] < A[j])
            {
                aux = A[j];
                A[j] = A[j + 1];
                A[j + 1] = aux;
                cambio = true;
            }
        }
        i = i + 1;
    }
    imprimirArreglo(A, n);
}
```

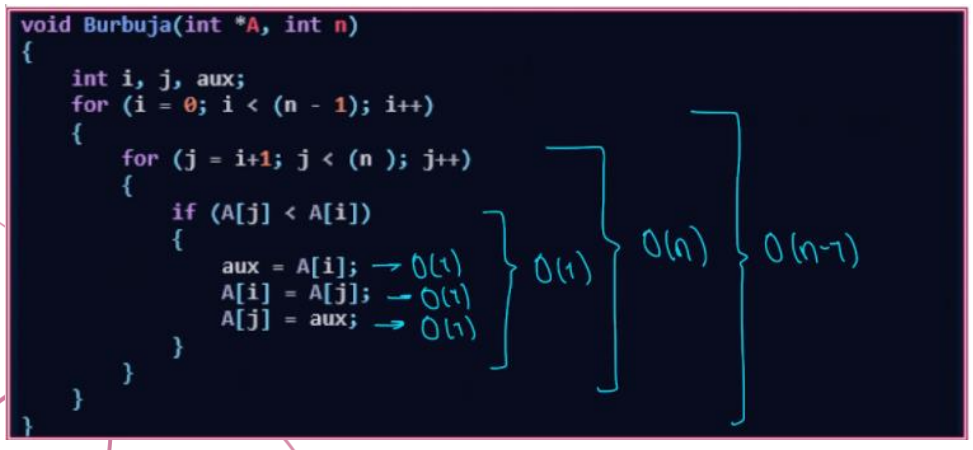


Como podemos observar tenemos nuevamente otra versión del ordenamiento de burbuja, donde una vez mas nos encontramos con varías asignaciones, un bucle while y un for, viendo el while podríamos pensar que habrá un algoritmo, pero nuevamente debemos fijarnos como es que incrementa la variable que controla al mismo, y esta no crece de forma exponencial, si no iterativa por lo cual presenta una complejidad n, así mismo el bucle for que hay dentro se ve controlado por el limite de n-1 por lo cual tendremos una complejidad resultante de:

$$O(n^2)$$

EJERCICIO 6

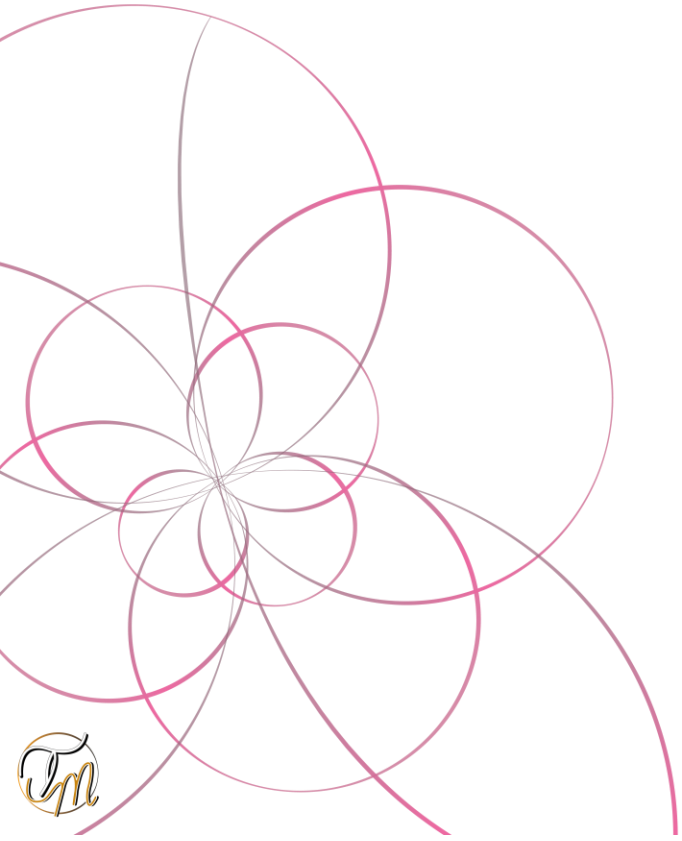
```
void Burbuja(int *A, int n)
{
    int i, j, aux;
    for (i = 0; i < (n - 1); i++)
    {
        for (j = 0; j < (n - 1); j++)
        {
            if (A[j] > A[j + 1])
            {
                aux = A[j];
                A[j] = A[j + 1];
                A[j + 1] = aux;
            }
        }
    }
    imprimirArreglo(A, n);
}
```



Podemos observar como en las situaciones anteriores que el método de burbuja simple, presenta en lo mas interno de su estructura 3 asignaciones, así como una comparativa, las cuales poseen un orden de O(n1), estas se encuentran dentro de un bucle for que esta determinado por n, incrementa de forma lineal, ósea que va de 1 en 1 por lo cual el orden de complejidad será de orden O(n), finalmente pasando a un bloque mas externo podemos observar el ultimo ciclo que va hasta n-1, por lo que e igual forma esta mayormente determinado por n, al tener un incremento lineal no hay presencia de exponenciales o logaritmos, por lo cual este ciclo se repetirá n-1 veces.

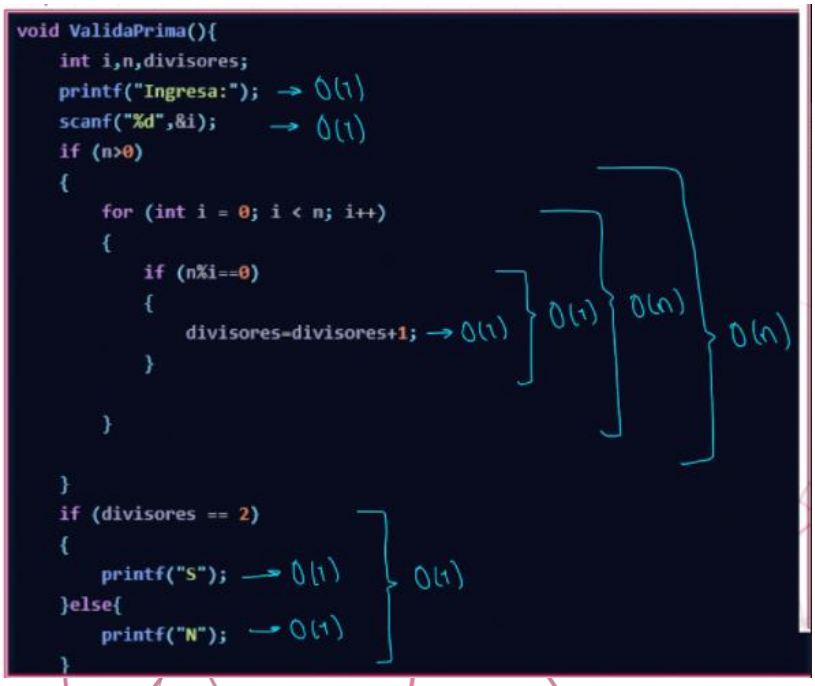
Ya que tenemos identificadas las operaciones primitivas, así como los ordenes de complejidad que nos va presentando cada bloque de instrucciones podemos determinar finalmente que este algoritmo nos esta presentando una complejidad final:

$$O(n^2)$$



EJERCICIO 7

```
void ValidaPrima(){
    int i,n,divisores;
    printf("Ingresa:");
    scanf("%d",&i);
    if (n>0)
    {
        for (int i = 0; i < n; i++)
        {
            if (n%i==0)
            {
                divisores=divisores+1;
            }
        }
    }
    if (divisores == 2)
    {
        printf("S");
    }else{
        printf("N");
    }
```



Para el siguiente algoritmo de validación si un numero es primo o no, tendremos diversos bloques que tienen anidaciones con ciclos, como siempre iniciamos encontrando todas las operaciones primitivas, siendo la primera la solicitud de un número.

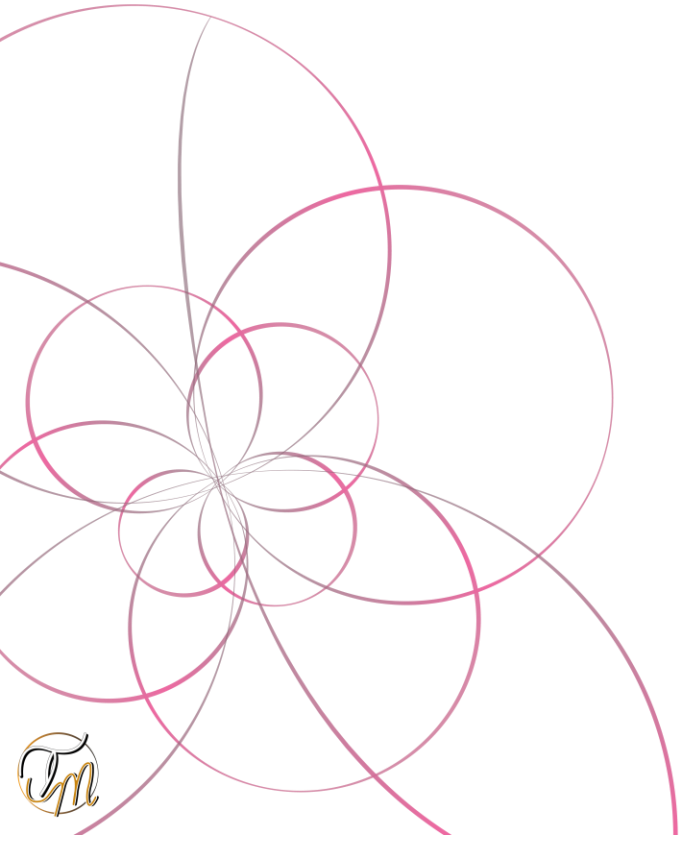
En lo mas interno del ciclo podemos observar una asignación, por lo que este primer bloque tiene una complejidad constante, este se encuentra dentro de una comparación, el cual proporciona el mismo tipo de complejidad

Posteriormente nos encontramos con un bucle que se ira realizando conforme el valor de la variable i sea menor que n,

esta variable de control va incrementando de 1 en 1 por lo cual vemos un crecimiento iterativo, lo cal nos proporciona una complejidad de (n).

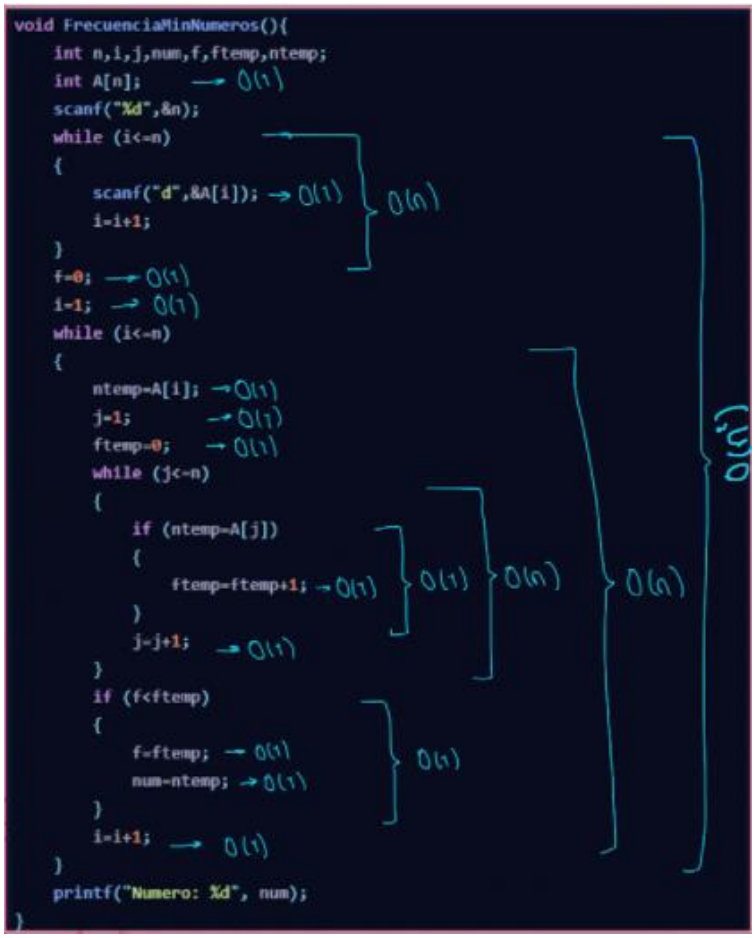
Para finalizar con esta serie de bloque analicemos el bloque que contiene a todos, el cual es la condicional que nos permite entrar al mismo, el cual tiene un comportamiento de oren lineal, lo cual nos proporciona como resultado

$$O(n)$$



EJERCICIO 8

```
void FrecuenciaMinNumeros(){
    int n,i,j,num,f,ftemp,ntemp;
    int A[n];
    scanf("%d",&n);
    while (i<=n)
    {
        scanf("d",&A[i]);
        i=i+1;
    }
    f=0;
    i=1;
    while (i<=n)
    {
        ntemp=A[i];
        j=1;
        ftemp=0;
        while (j<=n)
        {
            if (ntemp=A[j])
            {
                ftemp=ftemp+1;
            }
            j=j+1;
        }
        if (f<ftemp)
        {
            f=ftemp;
            num=ntemp;
        }
        i=i+1;
    }
    printf("Numero: %d", num);
}
```



Como podemos ver este algoritmo tiene una gran cantidad de operaciones primitivas, teniendo entre ellas en su mayoría operaciones de asignación a variables.

Ahora procederemos a analizar los bloques correspondientes según se presente

Tenemos el primer bloque que es un ciclo que guardará el valor recorriendo todas las posiciones del arreglo por lo tanto será de orden lineal

El próximo bloque es el bloque de comparación más anidado dentro del ciclo de j, como solo tiene una operación de orden constante este bloque será de igual manera constante

El siguiente bloque que tendremos es la comparación en el ciclo principal donde tenemos 2 operaciones constantes, al ser las 2 operaciones iguales el bloque será de igual manera de orden constante

Ahora que ya tenemos todos lo bloques procedemos a obtener el orden del principal gran bloque que es el ciclo en términos de i

Ahora podemos observar el bloque que esta controlado por la variable j, el cual ira recorriendo el arreglo mientras sea menor que n, como podemos observar la forma de incrementar esta variable, es mediante una suma en solo una unidad, por lo cual presenta un orden lineal O(n)

Ya contando con la cota de este bloque podemos determinar la del bloque controlado por i, al tener 3 operaciones primitivas, un bloque de comparación de orden constante, y mediante la aplicación de regla de la suma el contenido del ciclo es de orden lineal.

Ahora que tenemos todos los bloques analizamos podemos terminar con el segundo bucle while que de igual forma posee el mismo comportamiento que el while que este tiene dentro, pues de igual forma se realizara durante el cumplimiento de i<=n y con un incremento de 1 en 1 por lo cual tenemos un orden lineal. Finalmente procedemos a realizar la aplicación de la regla de la suma teniendo como resultado una complejidad total determinada por:

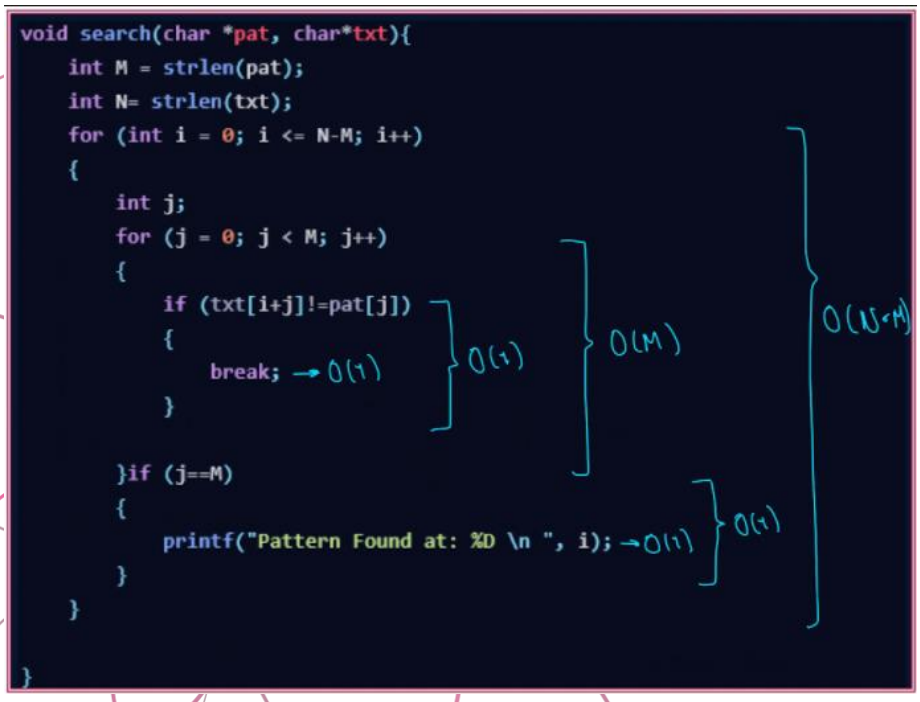
$$O(n^2)$$

EJERCICIO 9

```
void search(char *pat, char*txt){
    int M = strlen(pat);
    int N= strlen(txt);
    for (int i = 0; i <= N-M; i++)
    {
        int j;
        for (j = 0; j < M; j++)
        {
            if (txt[i+j]!=pat[j])
            {
                break;
            }

        }if (j==M)
        {
            printf("Pattern Found at: %D \n ", i);
        }
    }

}
```



Para comenzar con este algoritmo, podemos observar que esta emplea funciones pertenecientes a la librería de stingh.h, y debido a que no sabemos la forma exacta en que operan estas funciones las determinaremos con una complejidad de tipo 1

La siguiente operación primitiva es la declaración de j, dentro del bloque condicional dentro del ciclo de j tenemos un break esta es una operación que termina la función por ende es una operación de orden constante,

Finalmente, en el último bloque de comparación que se presenta tenemos un printf el cual consideraremos como orden 1

Como ya tenemos todas las operaciones primitivas podemos obtener las cotas de los bloques, la comparación más anidada es de orden constante (la comparación y una respectiva operación aritmética), después tenemos el ciclo en términos de j que iremos desde 0 hasta m, de uno en uno por ende es de orden lineal en términos de m $O(M)$

Ya que tenemos el orden del interior podemos obtener el orden del ciclo principal, como podemos ver estamos en términos de n y de m pero aun así estamos recorriendo el arreglo desde 0 hasta N, que será mayor al valor de M debido a que es donde contiene nuestro texto por lo tanto nuestro ciclo es de orden lineal en términos de n $O(N)$

Ya que tenemos estas 2 cotas podemos decir que todo el ciclo esta determinado por una cota de del tipo O(N*M) debido a que son diferente tamaños no podemos decir que son cuadráticos aplicando la regla de la multiplicación, por lo tanto, la cota de este ciclo es de este orden

EJERCICIO 10

```
stack<int> sortStack(stack<int> &input)
{
    stack<int> tmpStack;

    while (!input.empty())
    {
        // pop out the first element
        int tmp = input.top();
        input.pop();

        // while temporary stack is not empty and top
        // of stack is greater than temp
        while (!tmpStack.empty() && tmpStack.top() > tmp)
        {
            // pop from temporary stack and push
            // it to the input stack
            input.push(tmpStack.top());
            tmpStack.pop();
        }

        // push temp in temporary of stack
        tmpStack.push(tmp);
    }

    return tmpStack;
}
```

O(1)
O(1)
O(1)
O(1)
O(1)
O(1)
O(n
O(n)
$O(n^2)$

Para este algoritmo en especial tendremos una estructura de control que, mediante las operaciones para las pilas, push, pop y empty tendremos que ordenar una serie de números almacenados en la pila con una pila temporal o auxiliar, para este algoritmo tomaremos todas las operaciones primitivas siguientes

Tomaremos la creación de la pila temporal que tendrá nuestros números ordenados

Seguido de esta operación comenzamos con el primer ciclo while donde tendremos 2 operaciones primitivas:

- Creación de una variable que almacenara el valor del top de la pila para su ordenamiento
- Operación de pop que eliminara este elemento que se encuentra en tmp

Dentro del ciclo más anidado nos encontramos con 2 operaciones primitivas más:

- Insertaremos el valor en el top de nuestra pila temporal en nuestra pila original
- Eliminación con la operación pop

Saliendo de este ciclo nos encontramos con otra operación donde insertaremos el valor de temporal dentro de nuestra pila temporal

Una vez terminado este ciclo tendremos que devolver la pila temporal que tendrá nuestra pila con los valores ya ordenados

Todas estas operaciones son representantas mediante el orden de tipo 1

ya que tenemos todas nuestras operaciones primitivas procederemos a obtener la cota de nuestro ciclo más anidado, este ciclo se repetirá mientras nuestra pila temporal no este vacía y el top sea mayor que nuestro valor temporal, es decir ejecutaremos este ciclo mientras la operación no este vacía es decir que la haremos un aproximado de n veces donde n es el tamaño de la pila, por lo tanto, nuestro ciclo es de orden lineal $O(n)$

Ya que tenemos la cota del ciclo interno podemos realizar la obtención de todo el bloque que tiene a este contenido, primero determinamos que todas las operaciones internas de este ciclo tienen un orden 1 debido a que hablamos de asignaciones primitivas las cuales se realizaran la cantidad de veces determinada por la cota O(n), siendo este el orden de complejidad de este ciclo, por lo cual tendremos un orden final de O(n^2)

BIBLIOGRAFIA

[1] E. A. F. Martinez, «EaFranco.Eakdemy,» [En línea]. Available: https://eafranco.eakdemy.com/mod/scorm/player.php. [Último acceso: 07 Octubre 2021].

[2] «After Academy,» 2020 Abril 6. [En línea]. Available: https://afteracademy.com/blog/sort-a-stack-using-another-stack. [Último acceso: 2021 Octubre 8].

[3] «FacePrep,» 2019 Febrero 28. [En línea]. Available: https://www.faceprep.in/c/sort-a-stack-using-temporary-stack-and-recursion/. [Último acceso: 2021 Octubre 8].

[4] «GeeksforGeeks,» 2021 Septiembre 13. [En línea]. Available: https://www.geeksforgeeks.org/sort-stack-using-temporary-stack/. [Último acceso: 2021 Octubre 8].