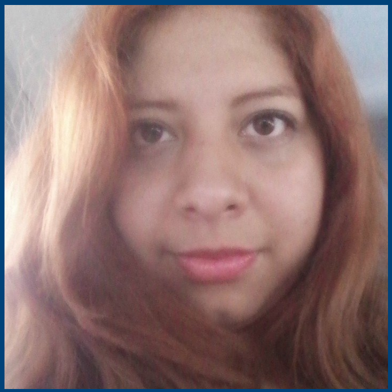
ESCOM-IPN

Ejercicio 3
Análisis de Complejidad

Análisis de Algoritmos

Laura Andrea Morales López

Abril 2018



Índice

1. Complejidad de los algoritmos

Para los siguientes 15 algoritmos determine la cota O().

1.1. Algoritmo 1:

```
for(i = 1; i < n; i++){                     //O(n)
    for(j = 0; j < n - 1; j++){             //O(n)
        temp = A[j];                        //O(1)
        A[j] = A[j + 1];
        A[j + 1] = temp;
    }
}
```

Son 2 ciclos anidados. Analizando el bloque de instrucciones dentro de ambos tenemos que es constante $O(1)$.

Siguiendo el analisis el ciclo que lo controla es un ciclo $O(n-1) = O(n)$ por lo tanto la complejidad de ese bloque sería $O(1n) = O(n)$.

Analizamos el ciclo exterior y notamos que se ejecuta $O(n)$ con lo cual tenemos una $O(n^2)$.

La complejidad de este algoritmo es $O(n^2)$

1.2. Algoritmo 2:

```
polinomio = 0;
for(i = 0; i <= n; i++){                        //O(n)
    polinomio = polinomio * z + A[n - i];   //O(1)
}
```

La primer instrucción es de $O(1)$.

Sigue con el ciclo for que tiene una instucción constante. El ciclo for es de $O(n)$.

Al ser mayor el ciclo usamos este para describir la complejidad del algoritmo siendo esta $O(n)$

1.3. Algoritmo 3:

```
for i = 1 to n do                               // O(n^3)
    for j = 1 to n do                           //O(n^2)
        C[i,j] = 0;                             //O(1)
        for k = 1 to n do                   //O(n)
C[i,j] = C[i,j] + A[i,k]*B[k,j]; //O(1)
```

En el bloque más interno tenemos una serie de operaciones $O(1)$ en un for lineal siento la complejidad de este algoritmo $O(n)$.

El siguiente bloque es de orden constante con lo cual vemos que el orden lineal será la complejidad de nuestro bloque completo.

Este bloque esta dentro de un for que tambien es lineal en terminos de n con lo cual tendremos una complejidad $O(n^2)$ por el momento.

Analizando el ultimo bloque podemos ver un for lineal tambien con lo cual concluímos una complejidad del algoritmo de $O(n^3)$

1.4. Algoritmo 4:

```
anterior = 1;        //O(1)
actual = 1;
while (n > 2){                //O(n)
    aux = anterior + actual;//O(1)
    anterior = actual;
    actual = aux;
    n = n - 1;
}
```

Tenemos el bloque de instrucciones interior constante así como el bloque del principio.

Con lo cual la complejidad dependerá exclusivamente del ciclo en cual es lineal.

Conlcuimos la complejidad $O(n)$

1.5. Algoritmo 5:

```
for (i = n - 1, j = 0; i >= 0; i--, j++)     //O(n)
    s2[j] = s[i];                      //O(1)
for (i = 0, i < n; i++)                       //O(n)
    s[i] = s2[i];                      //O(1)
```

Tenemos 2 ciclos separados cada uno de estos tienen bloques de operaciones constantes y ambos son se complejidad $O(n)$.

Por lo tanto la complejidad del algoritmo es $O(n)$.

1.6. Algoritmo 9:

```
func Producto2Mayores(A,n)
    if(A[1] > A[2])           //O(1)
        mayor1 = A[1];
        mayor2 = A[2];
    else
```

```
            mayor1 = A[2];      //O(1)
            mayor2 = A[1];
        i = 3;

        while(i <= n)                           //O(n)
            if(A[i] > mayor1)       //O(1)
                mayor2 = mayor1;
                mayor1 = A[i];
            else if (A[i] > mayor2) //O(1)
                mayor2 = A[i];
            i = i + 1;
        return = mayor1 * mayor2;
fin
```

En este algoritmo tenemos el primer bloque constante, y dentro del ciclo tambien tenemos un bloque constante por lo tanto el algoritmo depende del ciclo.

Este se ejecuta n-3 veces con lo cual el algoritmo es lineal. $O(n)$

1.7. Algoritmo 10:

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

La parte más interna de los ciclos es constante y el primer ciclo a analizar se comparta de la manera n-i-1 entonces es $O(n)$ y el siguiente ciclo es $O(n)$ por lo tanto el algoritmo es $O(n^2)$

1.8. Algoritmo 11:

```
func MaximoComunDivisor(m, n){
    a = max(n, m);
    b = min(n, m);          //O(1)
    residuo = 1;
    mientras (b > 0){       //O(log(n))
        residuo = a mod b;  //O(1)
        a = b;
        b = residuo;
```

```
    }
    MaximoComunDivisor = a;
    return MaximoComunDivisor;
}
```

El teorema de Lamé afirma que el caso peor para este algoritmo es cuando se le pide calcular el máximo común divisor de dos números consecutivos de la sucesión de Fibonacci.

Como los numeros de fibonacci son consecutivos si les realizamos el mod la resta será siempre de 1. Entonces tendremos la complejidad de $O(log(n))$

1.9. Algoritmo 12:

```
func BurbujaOptimizada(A, n)
    cambios = "Si"
    i = 0                                   //O(1)
    Mientras i < n - 1 && cambios != "No" hacer //O(n)
        cambios = "No"                      //O(1)
        Para j = 0 hasta (n - 2) - i hacer  //O(n)
            Si A[j + 1] < A[j] hacer    //O(1)
                aux = A[j]
                A[j] = A[j + 1]
                A[j + 1] = aux
                cambios = "Si"
            Fin Si
        Fin Para
        i = i + 1
    Fin Mientras
Fin func
```

El pequeño burbuja, aunque se dice una manera optimizada la realidad es que el algoritmo no se optimiza lo suficiente.

Tenemos un bloque de instrucciones $O(1)$ dentro de un ciclo $O(n)$, dentro de otro ciclo $O(n)$.

Por lo tanto el algoritmo es de complejidad $O(n^2)$

1.10. Algoritmo 13:

```
func BurbujaSimple(A, n)
    Para i = 0 hasta n - 2 hacer
   //O(n)
        Para j = 0 hasta (n - 2) - i hacer      //O(n)
            Si A[j + 1] < A[j] hacer    //O(1)
```

```
                aux = A[j]
                A[j] = A[j + 1]
                A[j + 1] = aux
            Fin Si
        Fin Para
    Fin Para
Fin func
```

Es casi igual que el anterior, solo no tiene la bandera entonces lo podemos analizar de la siguiente manera.

Tenemos un bloque de instrucciones $O(1)$ dentro de un ciclo $O(n)$, dentro de otro ciclo $O(n)$.

Por lo tanto el algoritmo es de complejidad $O(n^2)$

1.11. Algoritmo 14

```
func 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);            //O(1)
    else
        if(b > c)
            if(a > c)
                salida(b, a, c);
            else
                salida(b, c, a);
        else
            salida(c, b, a);
Fin func
```

Este algortimo simplemente son comparaciones que son unicamente operaciones constantes.

Por lo tanto la complejidad tambien será constante.

1.12. Algoritmo 15:

```
func Seleccion(A, n)
    Para k = 0 hasta n - 2 hacer                //O(n)
```

```
        p = k
        Para i = k + 1 hasta n - 1 hacer     //O(n)
            Si A[i] < A[p] entonces
                p = i       //O(1)
            Fin Si
        Fin Para
        temp = A[p]         //O(1)
        A[p] = A[k]
        A[k] = temp
    Fin Para
Fin func
```

Tenemos 2 ciclos anidados en los cuales el primero se realiza con una complejidad n y el siguiente tambien por lo tanto tenemos una complejidad $O(n^2)$

2. Pequeña Gran Conclusión

Este análisis es el más bonito de todos, como nos pudimos dar cuenta en unos cuantos pasos podiamos decidir la complejidad que tendrian los algoritmos sin rompernos la cabeza.

Claro que este es un análisis que tiende a ser mejor para tamaños de problema muy grandes, y ademas no es tan exacto como el que se usó al principio pero cuando vez los tiempos reales de los algoritmos te das cuenta que lo que realmente alenta los algoritmos son los ciclos y de estos prácticamente depende la complejidad de los mismos.

Para realizar una estimación de un algoritmo esta es la mejor manerapues realizarlo es muy intuitivo y además rápido y simple.

Si tomamos los algoritmos del Ejercicio 2 nos damos cuenta que en realidad son los mismos simplemente que tomamos la cota, y dejamos de lado los valores constantes que realmente no afectan mucho a nuetro algoritmo.

Así que concluyo que éste es el método que más me gusta.