INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Análisis de algoritmos no recursivos

Ejercicios 04

Luis Fernando Reséndiz Chávez

Análisis de Algoritmos

Franco Martínez Edgardo Adrián

3CM3

5 de diciembre de 2020

Código 01

Haciendo un análisis por cada línea de código, podemos vemos que en el programa hay muchas asignaciones y comparaciones fuera de algún ciclo, tomando en cuenta que nos basamos en el peor de los casos, el programa pasará por todas las comparaciones y por todas las instrucciones dentro de las estructuras condicionales if else, es decir, hasta este momento nuestra complejidad en Big-O notation es O(1), constante. Sin embargo, en la línea 30 nos encontramos con un ciclo while, donde la comparación es controlada por la variable i, la cual siempre tendrá un valor inicial de 4, ya que es asignado en la línea 28, sin embargo, la complejidad exacta sería n-4, por lo tanto, en notación Big-O o notación asintótica sería O(n), dentro de este ciclo tenemos únicamente asignaciones y comparaciones, las cuales siguen siendo constantes O(1), finalmente, la complejidad de todo el algoritmo es O(n).

```
1   func SumaCuadratica3Mayores(A,n)                    
2   {                                                   
3       if(A[1] > A[2] && A[1] > A[3])          O(1)    
4           m1 = A[1];                          O(1)    
5           if (A[2] > A[3])                    O(1)    
6               m2 = A[2];                      O(1)    
7               m3 = A[3];                      O(1)    
8           else                                        
9               m2 = A[3];                      O(1)    
10              m3 = A[2];                      O(1)    
11      else if(A[2] > A[1] && A[2] > A[3])     O(1)    
12          m1 = A[2];                          O(1)    
13          if (A[1] > A[3])                    O(1)    
14              m2 = A[1];                      O(1)    
15              m3 = A[3];                      O(1)    
16          else                                        
17              m2 = A[3];                      O(1)    
18              m3 = A[1];                      O(1)    
19      else                                    O(1)    
20          m1 = A[3];                          O(1)    
21          if (A[1] > A[2])                    O(1)    
22              m2 = A[1];                      O(1)    
23              m3 = A[2];                              
24          else                                        
25              m2 = A[2];                      O(1)    
26              m3 = A[1];                      O(1)    
27                                                      
28      i = 4;                                  O(1)    
29                                                      
29                                                      
30      while(i<=n)                             O(n)    
31          if(A[i] > m1)                               
32              m3 = m2;                        O(1)    
33              m2 = m1;                        O(1)    
34              m1 = A[i];                      O(1)    
35          else if (A[i] > m2)                 O(1)    
36              m3 = m2;                        O(1)    
37              m2 = A[i];                      O(1)    
38          else if (A[i] > m3)                 O(1)    
39              m3 = A[i]                       O(1)    
40                                                      
41          i = i + 1;                          O(1)    
42                                                      
43      return = pow(m1 + m2 + m3,2);           O(1)    
44  }                                                   
```

Código 02

En este algoritmo, podemos notar que comenzamos con un ciclo for el cual va desde 0 hasta n-2, por lo tanto, la complejidad asintótica de este ciclo for es O(n), ahora, por cada iteración se ejecuta otro ciclo for, el peor caso es cuando entra por primera vez, cuando el ciclo va desde 1 hasta n-1, por lo tanto la complejidad asintótica de este segundo ciclo for es O(n), entonces la complejidad total que llevamos hasta el momento es $O(n^2)$, dentro de este segundo ciclo for encontramos únicamente operaciones de comparación y de asignación, es decir, operaciones en tiempo constante, por lo tanto, nuestra complejidad final para este algoritmo es $O(n^2)$.

```
1   func OrdenamientoIntercambio(A,n)
2   for (i=0; i<n-1; i++)                    O(n)
3       for ( j=i+1; j<n;j++)                    O(n)
4           if (A[j] < A[i])                         O(1)
5           {
6               temp=A[i];                           O(1)
7               A[i]=A[j];                           O(1)
8               A[j]=temp;                           O(1)
9           }
10  fin
```

Código 03

Este algoritmo, ya es conocido y lo hemos trabajado anteriormente, es el algoritmo de Euclides para obtener el maximo comun divisor entre dos números, en este caso para encontrar la complejidad asintótica de dicho algoritmo, únicamente hay que pensar en el peor de los casos, es decir, cuales son los peores valores que podemos recibir en la función, estos numeros son numeros seguidos en la serie de fibonacci, por lo tanto la complejidad para este caso es O(log(n)), logaritmo base 2 de n, para todos los demás casos se podría decir que es casi constante, sin embargo, nos estamos basando en el peor de los casos para obtener la complejidad de forma asintótica. Dentro del ciclo while solo se hacen operaciones constantes, es decir O(1), por lo tanto, la complejidad final del algoritmo es O(log(n)).

```
1   func MaximoComunDivisor(m, n)
2   {
3       a=max(n,m);                 O(1)
4       b=min(n,m);                 O(1)
5       residuo=1;                  O(1)
6       mientras (residuo > 0)      O(log(n))
7       {
8           residuo=a mod b;        O(1)
9           a=b;                    O(1)
10          b=residuo;              O(1)
11      }
12      MaximoComunDivisor=a;       O(1)
13      return MaximoComunDivisor;
14  }
```

Código 04

En este algoritmo nos encontramos primero con un ciclo for, el cual siempre va de 0 a 2, es decir, es constante y no depende de ninguna N, por lo tanto se puede tomar como una instrucción constante. Posteriormente, tenemos un segundo ciclo for anidado, el cual si hace uso de la n dada, sin embargo se va restando el valor actual del primer for pero como nos estamos basando en el peor de los casos, la primera iteración hace n-1 comparaciones, por lo tanto se puede tomar como un ciclo con complejidad asintótica O(n), las instrucciones que se encuentran dentro de este segundo for, las cuales son asignaciones y operaciones aritméticas son constantes, de tal manera que hasta el momento la complejidad del algoritmo se mantiene en orden lineal.
Finalmente, fuera de los ciclos tenemos operaciones aritméticas y de asignación y una operación de return la cual manda a llamar a la función pow, la cual se puede tomar como constante. Podemos concluir que la complejidad temporal asintótica de este algoritmo es O(n).

```
1   func SumaCuadratica3MayoresV2(A,n)
2   {
3       for(i=0;i<3;i++)                    O(1)
4       {
5           for (j=0;j<n-1-i;j++)           O(n)
6           {
7               if(A[j]>A[j+1])             O(1)
8               {
9                   aux=A[j];               O(1)
10                  A[j]=A[j+1];            O(1)
11                  A[j+1]=aux;             O(1)
12              }
13          }
14      }
15      r=A[n-1] + A[n-2] + A[n-3];         O(1)
16      return pow(r,2);                    O(1)
17  }
```

Código 05

En este algoritmo, el cual ya hemos analizado con anterioridad, nos damos cuenta que tenemos dos ciclos for, los cuales están anidados, y ambos tiene complejidad asintótica lineal, ya que como habíamos visto antes, el peor caso que podemos recibir es un arreglo que este ordenado de manera descendente, y tendrá que realizar todas las comparaciones posibles. Dentro del segundo for tenemos cuatro instrucciones que siempre se ejecutarán en el peor caso, tienen complejidad constante ya que solo son asignaciones y operaciones aritméticas. Por lo tanto, al tener dos ciclos for anidados y ambos de complejidad O(n), la complejidad temporal asintótica de este algoritmo es $O(n^2)$.

```
1   Procedimiento BurbujaOptimizada(A,n)
2       cambios = “Si”                                  O(1)
3       i=0                                             O(1)
4       Mientras i< n-1 && cambios != “No” hacer        O(n)
5           cambios = “No”                                  O(1)
6           Para j=0 hasta (n-2)-i hacer                    O(n)
7               Si(A[j] < A[j]) hacer                           O(1)
8                   aux = A[j]                                  O(1)
9                   A[j] = A[j+1]                               O(1)
10                  A[j+1] = aux                                O(1)
11                  cambios = “Si”
12              FinSi
13          FinPara
14          i= i+1                                      O(1)
15      FinMientras
16  fin Procedimiento
```

Código 06

En este algoritmo, el cual ya hemos analizado con anterioridad, vemos que al igual que el algoritmo de burbuja optimizada consta de dos ciclos for anidados, los cuales tienen complejidad temporal asintótica basada en en el peor caso O(n), es decir, la complejidad de este algoritmo también es $O(n^2)$. Únicamente existen operaciones constantes dentro de estos ciclos.

```
1   Procedimiento BurbujaSimple(A,n)
2       para i=0 hasta n-2 hacer                O(n)
3           para j=0 hasta (n-2)-i hacer            O(n)
4               si (A[j]>A[j+1]) entonces
5                   aux = A[j]                      O(1)
6                   A[j] = A[j+1]                   O(1)
7                   A[j+1] = aux                    O(1)
8               fin si                              O(1)
9           fin para
10      fin para
11  fin Procedimiento
```

Código 07

En este algoritmo, el cual tiene como objetivo validar un numero primero utilizando fuerza bruta, en las primeras tres líneas hay operaciones de asignación, por lo tanto son de orden constante. Y posteriormente hay un ciclo for el cual itera desde 1 hasta el número n dado, por lo tanto este ciclo tiene complejidad asintótica O(n), dentro de él solo hay comparaciones y asignaciones que se ejecutan en orden constante. Finalmente tenemos una estructura de control if else que evalúa cuántos números dividen al número dado, por lo tanto, son operaciones de orden constante O(1), podemos concluir que el algoritmo tiene complejidad temporal asintótica O(n).

```
1   Proceso ValidaPrimo
2       Leer n                              O(1)
3       divisores<-0                        O(1)
4       si n>0 Entonces                     O(1)
5           Para i<-1 Hasta n Hacer         O(n)
6               si (n%i=0) Entonces             O(1)
7                   divisores=divisores+1       O(1)
8               FinSi
9           FinPara
10      FinSi
11                                          O(1)
12      si divisores=2                      O(1)
13          Escribir 'S'
14      SiNo                                O(1)
15          Escribir 'N'                    O(1)
16      FinSi
17  FinProceso
```

Código 08

Este algoritmo comienza con asignaciones y creación de un arreglo, las cuales son operaciones que se ejecutan en tiempo constante O(1), sin embargo, nos encontramos con un ciclo while el cual lee del teclado n valores, por lo tanto su complejidad temporal asintótica es O(n), más adelante, después de otras operaciones de asignación constantes, nos encontramos con un ciclo while el cual va desde 1 hasta n, por lo tanto, podemos decir que su complejidad temporal asintótica es O(n), dentro de este while no solo hay operaciones constantes, posteriormente hay otro ciclo for, el cual siempre empieza desde 1 y termina hasta n, por lo tanto su complejidad temporal asintótica es O(n) ya que dentro de este hay únicamente operaciones de comparación y asignación las cuales se ejecutan de form constante. Finalmente podemos concluir que la complejidad de este algoritmo es $O(n^2 + n)$.

```
1   Algoritmo FrecuenciaMinNumeros
2       Leer n                                  O(1)
3       Dimension A[n]                          O(1)
4       i=1                                     O(1)
5       Mientras i<=n                           O(n)
6           Leer A[i]                               O(1)
7           i=i+1                                   O(1)
8       FinMientras
9
10      f=0                                     O(1)
11      i=1                                     O(1)
12      Mientras i<=n                           O(n)
13          ntemp=A[i]                              O(1)
14          j=1                                     O(1)
15          ftemp=0
16          Mientras j<=n                           O(n)
17
18              si ntemp=A[j]                           O(1)
19                  ftemp=ftemp+1                       O(1)
20              FinSi                                   O(1)
21              j=j+1
22          FinMientras
23                                                  O(1)
24          si f<ftemp                              O(1)
25              f=ftemp                             O(1)
26              num=ntemp
27          FinSi
28
29          i=i+1
30      FinMientras
31      Escribir num
32  FinAlgoritmo
```

Código 09

En este algoritmo comenzamos a evaluar en las líneas tres y cuatro, las cuales son asignaciones que se ejecutan en tiempo constante, obtiene las longitudes de ambos arreglos de caracteres recibidos por la función. Después tenemos un primer ciclo for, el cual va desde 0 hasta la diferencia de n y m, por lo tanto, la complejidad temporal asintótica de este algoritmo es O(n-m) , dentro del mismo tenemos un ciclo for dentro, es decir un ciclo anidado, el cual se ejecuta desde 0 hasta m, por lo tanto su complejidad temporal asintótica para el peor caso es O(m), pero como es un ciclo anidado, tenemos que la complejidad que llevamos hasta el momento es O((n-m)*(m)), finalmente también tenemos un if en el primer for, sin embargo no afecta en nuestra complejidad temporal actual

ya que se ejecuta en tiempo constante. Por lo tanto, la complejidad temporal asintótica final del algoritmo es O((n-m)*(m)).

```
1   void search(char* pat, char* txt)                    
2   {                                                    
3       int M = strlen(pat);                  O(1)       
4       int N = strlen(txt);                  O(1)       
5                                                          
6       for (int i = 0; i <= N - M; i++)      O(n-m)     
7       {                                                  
8           int j;                            O(1)       
9                                                          
10          for (j = 0; j < M; j++)           O(m)       
11              if (txt[i + j] != pat[j])     O(1)       
12                  break;                                 
13                                                         
14          if (j == M)                       O(1)       
15              printf("Pattern found at index %d \n", i);
16      }                                                  
17  }                                                    
```

Código 10

En este algoritmo el cual es un ordenamiento de números utilizando dos pilas implementado en C++ comenzamos a analizar la línea número 3, la cual crea una pila de números enteros, la cual podemos contar como una instrucción que se ejecuta en tiempo constante. Posteriormente nos encontramos con un ciclo while el cual se va a ejecutar mientras la pila recibida en la función tenga al menos un elemento dentro, si tomamos como N el número de elementos dentro de dicha pila, podemos decir que este ciclo while tiene como complejidad temporal asintótica O(n), dentro del mismo tenemos dos instrucciones que se ejecutan en tiempo constante, una asignación y sacar un elemento de la pila, sin embargo más adelante nos encontramos con otro ciclo while, el cual tiene la

condición de que se va a ejecutar mientras la pila auxiliar que se creó al principio del algoritmo no esté vacía, por lo tanto cuando recibamos el peor de los casos que es una pila con los números ordenados de mayor a menor, resulta que en el peor de los casos vamos a tener que eliminar todos los elementos de la pila, sin embargo sabemos que el número máximo de elementos que puede tener dentro esta pila auxiliar son n, por lo tanto la complejidad temporal asintótica de este ciclo while anidado es de O(n), por lo tanto podemos concluir que la complejidad temporal de este algoritmo es $O(n^2)$.

```
1   stack<int> sortStack(stack<int> &input)
2   {
3       stack<int> tmpStack;                                    O(1)
4
5       while (!input.empty())                                  O(n)
6       {                                                       O(1)
7           int tmp = input.top();                              O(1)
8           input.pop();
9
10          while (!tmpStack.empty() && tmpStack.top() > tmp)   O(n)
11          {                                                   O(1)
12              input.push(tmpStack.top());                     O(1)
13              tmpStack.pop();
14          }
15
16          tmpStack.push(tmp);                                 O(1)
17      }                                                       O(1)
18      return tmpStack;
```