Instituto Politécnico Nacional

Escuela Superior de Cómputo

Análisis de Algoritmos

Profesor: Edgardo Adrián Franco Martínez

Alumno: Ortega Victoriano Ivan

Fecha: 24 de Marzo 2017

Análisis de algoritmos no recursivos.
Ejercicio 1: Encuentre el orden O de complejidad temporal y espacial del algoritmo de ordenamiento por Burbuja Simple.

```
Procedimiento BurbujaSimple(A,n)
  para i=1 hasta (i<n) hacer
    para j=0 hasta (j<n-1) hacer
      si (A[j]>A[j+1]) hacer
        temp = A[j]
        A[j] = A[j+1]
        A[j+1] = temp
      fin si
    fin para
  fin para
fin Procedimiento
```

Sol: Para el análisis temporal se considerarán como operaciones básicas las 3 asignaciones dentro del *if*, la comparación del *if* y los incrementos en los loops. Además, se tomará el peor caso para encontrar la cota O.
Para este algoritmo, el peor caso se da cuando el arreglo esta ordenado de forma descendente, es decir, de mayor a menor, ya que así, siempre estará entrando a la condición del if, ejecutando las 3 asignaciones. Del código, vemos lo siguiente:

```
para i=1 hasta (i<n) hacer          //Compara n veces
  para j=0 hasta (j<n-1) hacer      //Compara n veces
    si (A[j]>A[j+1]) hacer          //1 comparacion
      temp = A[j]                   //3 asignaciones
      A[j] = A[j+1]
      A[j+1] = temp
    fin si
  fin para
fin para
```

De tal forma que la función de complejidad temporal del algoritmo, estará dada por:

$$f_t(n) = (n)(n)(3) = 3n^2$$

Ahora para encontrar el orden de complejidad temporal, de acuerdo con la notación de Landau:

Notación de Landau. *Se dice que* $f(n)$ *es de orden* $O(g(n))$ *si*

$$\exists c \geq 0 \ y \ n_0 \geq 0 \mid |f(n)| \leq c|g(n)| \ , \forall n \geq n_0.$$

Ahora, sea $g(n) = n^2$ y $c = 4$, vemos que:
Si n=0,

$$f(0) = 3(0)^2 = 0 \leq 0 = 4(0)^2 = g(0)$$

Si n=1,

$$f(1) = 3(1)^2 = 3 \leq 4 = 4(1)^2 = g(1)$$

Si n=2,

$$f(2) = 3(2)^2 = 12 \leq 16 = 4(2)^2 = g(2)$$

En general, se cumple que

$$|f(n)| \leq c|g(n)| \ , \forall n \geq n_0.$$

De lo que podemos decir que $f_t(n)$ es de orden $O(n^2)$.

Por otro lado, para el análisis espacial, tenemos

```
para i=1 hasta (i<n) hacer      // 1 variable i
  para j=0 hasta (j<n-1) hacer // 1 variable j
    si (A[j]>A[j+1]) hacer
      temp = A[j]   // 1 variable temp
      A[j] = A[j+1] // n variables del arreglo "A"
      A[j+1] = temp
    fin si
  fin para
fin para
```

De tal forma que la función de complejidad espacial del algoritmo estará dada por:

$$f_e(n) = 3 + n$$

Para encontrar la cota O de $f_e(n)$, se hace al igual que en el caso anterior.
Sea $g(n) = n$, $c = 4$ y $n_0 = 1$, tenemos que
Si n=1,

$$f(1) = 3 + 1 = 4 \leq 4 = 4(1) = g(1)$$

Si n=2,

$$f(2) = 3 + 2 = 5 \leq 8 = 4(2) = g(2)$$

Si n=3,

$$f(3) = 3 + 3 = 6 \leq 12 = 4(3) = g(3)$$

En general, se cumple que

$$|f(n)| \leq c|g(n)| \text{ , } \forall n \geq n_0.$$

De lo que podemos decir que $f_e(n)$ es de orden $O(n)$.

Ejercicio 2: Encuentre el orden O de complejidad temporal y espacial del algoritmo de ordenamiento por Inserción.

```
Procedimiento Insercion(A,n)
  para i=1 hasta i<n hacer
    temp=A[i]
    j=i-1
    mientras((A[j]>temp)&&(j>=0)) hacer
      A[j+1]=A[j]
      j--
    fin mientras
    A[j+1]=temp
  fin para
fin Procedimiento
```

Sol: Para este algoritmo, al igual que en el caso del ordenamiento burbuja, nos basaremos en el análisis del peor caso para encontrar la cota O. De tal forma que para el ordenamiento por inserción el peor caso se da de igual forma cuando el arreglo está ordenado de forma descendente, ya que así, siempre estará ejecuntando las operaciones del while, de no serlo así, habría ocasiones en las que simplemente no entraría al ciclo, ya que no se cumpliría que $A[j] > temp$.
Del código, vemos lo siguiente:

```
Procedimiento Insercion(A,n)
  para i=1 hasta i<n hacer    // compara n veces, sin embargo,
    temp=A[i]         // las operaciones  realizadas
    j=i-1           // dependen del while
    mientras((A[j]>temp)&&(j>=0)) hacer // su analisis se hace
      A[j+1]=A[j]           // a continuacion
      j--
    fin mientras
    A[j+1]=temp
  fin para
fin Procedimiento
```

Analizando el caso para el ciclo *while*, y tomando como operación base únicamente la asignación, tenemos que para el contador $i = 1$ el ciclo se ejecutará una sola vez, ya que $j = 0$, al decrementarla para la siguiente iteración tendríamos $j = -1$, lo cual ya no cumpliría con la condición del *while* que pide que $j <= 0$. Así para $i = 2$, tendríamos para las iteraciones del *while* que para la primera $j = 1$, se decrementa, luego $j = 0$, se decrementa, y así sucesivamente.

De tal forma que para la función de complejidad temporal tenemos lo siguiente:

$$f_t(n) = 1 + 2 + 3 + ... + (n-1) + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Si observamos la función, nos damos cuenta que es de orden cuadrático, ahora obtengamos la cota O de $f_t(n)$.
Sea $g(n) = n^2$, $c = 1$ y $n_0 = 1$, se tiene que
Si n=1,

$$f(1) = \frac{1^2+1}{2} = 1 \leq 1 = 1^2 = g(1)$$

Si n=2,

$$f(2) = \frac{2^2+2}{2} = 3 \leq 4 = 2^2 = g(2)$$

Si n=3,

$$f(1) = \frac{3^2+3}{2} = 6 \leq 9 = 3^2 = g(3)$$

En general, se cumple que

$$|f(n)| \leq c|g(n)| \ , \forall n \geq n_0.$$

De lo que podemos decir que $f_t(n)$ es de orden $O(n^2)$.
Ahora para el análisis de complejidad espacial tenemos que:

```
Procedimiento Insercion(A,n)
  para i=1 hasta i<n hacer    // 1 variable i y 1 variable temp
    temp=A[i]          // n variables del arreglo "A"
    j=i-1            // 1 variable j
    mientras((A[j]>temp)&&(j>=0)) hacer
      A[j+1]=A[j]
      j--
    fin mientras
    A[j+1]=temp
  fin para
fin Procedimiento
```

De tal forma que la función de complejidad espacial queda de la siguiente manera:

$$f_e(n)) = n + 3$$

Para encontrar su cota O, propongamos $g(n) = n$, $c = 4$ y $n_0 = 1$
Si n=1,

$$f(1) = 1 + 3 = 4 \leq 4 = 4(1) = g(1)$$

Si n=2,

$$f(2) = 2 + 3 = 5 \leq 8 = 4(2) = g(2)$$

Si n=3,

$$f(3) = 3 + 3 = 6 \leq 12 = 4(3) = g(3)$$

En general, se cumple que

$$|f(n)| \leq c|g(n)| \text{ , } \forall n \geq n_0.$$

De lo que podemos decir que $f_e(n)$ es de orden $O(n)$.

Ejercicio 3: Encuentre el orden O de complejidad temporal y espacial del algoritmo de ordenamiento por Seleccion.

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

Sol: Al igual que en los casos anteriores, el peor de los casos de este algoritmo se da cuando el arreglo esta ordenado de forma descendente, ya que de esta forma siempre estará entrando en las 2 sentencias *if* del *for* mas anidado, ejecutando un mayor número de operaciones. De tal forma que tendremos lo siguiente:

```
Procedimiento Seleccion(A,n)
  para k=0 hasta k<n-1 hacer
    p=k;
    para i=k+1 hasta i>n-1 hacer // se ejecuta parecido
      si A[i]<A[p] hacer     // al ordenamiento por insercion
        p = i        // el analisis se hace a continuacion
      fin si
      si p!=k hacer
        temp = A[p]     // las operaciones dentro  de las
        A[p] = A[k]     // sentencias if se cuentan, siendo un
        A[k] = temp     // total de 4
      fin si
    fin para
  fin para
fin Procedimiento
```

Al igual que en caso del ordenamiento por inserción, el número de operaciones va a regirse por el ciclo mas anidado, que es este caso es el for que va de $i = k + 1$ hasta $i > n - 1$, para la primera iteración de este arreglo

con $k = 0$, tendríamos $i = 1$ se harían $n - 1$ comparaciones para validar la condición del for por el número de operaciones dentro de él; para $i = 2$, se harían $n - 2$ comparaciones, y así hasta $i = n - 2$ (que es el último valor en entrar a la condición) se haría una sola comparación.
Si nos damos cuenta, se da una situación parecida al algoritmo de ordenamiento por inserción, solo que el número de operaciones realizadas por cada iteración va de forma descendente, empezando con $n - 1$ y bajando así hasta una sola operación. De tal forma que la función de complejidad temporal estará dada por:

$$f_t(n) = 4\sum_{i=1}^{n-1} i = \frac{4(n-1)n}{2} = 2n(n-1)$$

Ahora para encontrar su orden de complejidad, sean $g(n) = n^2$, $c = 2$ y $n_0 = 0$
Si n=0,

$$f(0) = 2(0)(0-1) = 0 \leq 0 = 2(0)^2 = g(0)$$

Si n=1,

$$f(1) = 2(1)(1-1) = 0 \leq 2 = 2(1)^2 = g(1)$$

Si n=2,

$$f(2) = 2(2)(2-1) = 4 \leq 8 = 2(2)^2 = g(2)$$

En general, se cumple que

$$|f(n)| \leq c|g(n)| \ , \forall n \geq n_0.$$

De lo que podemos decir que $f_t(n)$ es de orden $O(n^2)$
En cuanto al análisis temporal, tenemos lo siguiente:

```
Procedimiento Seleccion(A,n)
  para k=0 hasta k<n-1 hacer      // 1 variable k
    p=k;                 // 1 variable p
    para i=k+1 hasta i>n-1 hacer  // 1 variable i
      si A[i]<A[p] hacer      // n variables del arreglo "A"
        p = i
      fin si
      si p!=k hacer
        temp = A[p]       // 1 variable temp
```

```
        A[p] = A[k]
        A[k] = temp
      fin si
    fin para
  fin para
fin Procedimiento
```

Siendo entonces la función de complejidad espacial:

$$f_e(n) = n + 4$$

De lo que, si tomamos $g(n) = n$, $c = 5$ y $n_0 = 1$
Si n=1,

$$f(1) = 1 + 4 = 5 \leq 5 = 5(1) = g(1)$$

Si n=2,

$$f(2) = 2 + 5 = 7 \leq 10 = 5(2) = g(2)$$

Si n=3,

$$f(3) = 3 + 5 = 8 \leq 15 = 5(3) = g(3)$$

En general, se cumple que

$$|f(n)| \leq c|g(n)| \text{ , } \forall n \geq n_0.$$

De lo que podemos decir que $f_e(n)$ es de orden $O(n)$.

Ejercicio 4: Encuentre el orden O de complejidad temporal y espacial del algoritmo de ordenamiento Shell.

```
Procedimiento Shell(A,n)
  k = n / 2;
  mientras k >= 1 hacer
    para i=k hasta i>=n hacer
      v = A[i]
      j = i - k;
      mientras j >= 0 && A[j] > v hacer
        A[j + k] = A[j];
        j -= k;
      fin mientras
      A[j + k] = v;
    fin para
  k/=2;
  fin mientras
fin Procedimiento
```

Sol: El análisis de este algoritmo resulta un poco más complejo debido a que depende mucho de la manera en que se realizan los saltos. En esta ocasión se realizará el análisis sobre el mejor caso, es decir donde el arreglo esta ordenado.
Haciendo esa acotación podemos ver que si el arreglo está ordenado, la condición del ciclo *while* mas anidado, nunca se ejecutará, de tal forma que el número de operaciones estará dado por el ciclo *for*, para la primera iteración serán $\frac{n+1}{2}$ veces, para la seguna $\frac{3(n+1)}{4}$, para la tercera $\frac{7(n+1)}{8}$ y así sucesivamente. Visto de otra forma podemos hacer $n - \frac{n+1}{2}$ para la primera iteración, para la segunda $n - \frac{n+1}{3}$, para la tercera $n - \frac{n+1}{3}$. Lo anterior puede representarse mediante la siguiente expresión:

$$\sum_{i=1}^{m} n - \sum_{i=1}^{m} \frac{1}{2^i}(n-1)$$

Conociendo la serie geométrica, la segunda sumatoria puede cambiarse, teniendo así la siguiente expresión:

$$\sum_{i=1}^{m} n - \sum_{i=0}^{m-1} (\frac{1}{2})\frac{1}{2^i}(n-1)$$

Del resultado de la serie geométrica, suponiendo que $m \to \infty$ obtenemos así la siguiente expresión:

$$mn - (n-1)$$

Donde m representa la cantidad de veces que el arreglo de tamaño n puede dividirse sucesivamente en dos, es decir $log_2 n$.
Reduciendo la expresión y haciendo un cambio de variable $m-1 = b$, tenemos lo siguiente:

$$\sum_{i=1}^{m} n - \sum_{i=1}^{m} \frac{1}{2^i}(n-1) = bn + 2$$

Finalmente, es fácil ver que para el primer ciclo *while* tendremos que se ejecutará $log_2 n$ veces. Así finalmente la función de complejidad temporal:

$$f_t(n) = (bn + 2)\log_2(n)$$

Donde $b = log_2 n - 1$.
Encontrando una cota para esta función, como en los algoritmos anteriores, sean $g(n) = n\log_2(n)$, $c = 6$ y $n_0 = 2$
Si n=2,

$$f(2) = (\log_2(2) - 1)\log_2(2) = 0 \leq 12 = 6(2)\log_2(2) = g(2)$$

Si n=3,

$$f(3) = (\log_2(3) - 1)\log_2(3) = 0.927 \leq 28.52 = 6(2)\log_2(2) = g(3)$$

Si n=4,

$$f(4) = (\log_2(2) - 1)\log_2(2) = 2 \leq 48 = 6(4)\log_2(4) = g(4)$$

En general, se cumple que

$$|f(n)| \leq c|g(n)| \ , \forall n \geq n_0.$$

De lo que podemos decir que $f_t(n)$ es de orden $O(n\log_2(n))$.
Por otro lado, para el analisis de complejidad espacial, tenemos lo siguiente:

```
Procedimiento Shell(A,n)
  k = n / 2;                          // 1 variable k
  mientras k >= 1 hacer
    para i=k hasta i>=n hacer         // 1 variable i
      v = A[i]                        // 1 variable v
      j = i - k;                      // 1 variable j
      mientras j >= 0 && A[j] > v hacer
        A[j + k] = A[j];              // n variables de A
        j -= k;
      fin mientras
```

```
      A[j + k] = v;
    fin para
  k/=2;
  fin mientras
fin Procedimiento
```

Siendo entonces la función de complejidad espacial:

$$f_e(n) = n + 4$$

De lo que, si tomamos $g(n) = n$, $c = 5$ y $n_0 = 1$
Si n=1,

$$f(1) = 1 + 4 = 5 \leq 5 = 5(1) = g(1)$$

Si n=2,

$$f(2) = 2 + 5 = 7 \leq 10 = 5(2) = g(2)$$

Si n=3,

$$f(3) = 3 + 5 = 8 \leq 15 = 5(3) = g(3)$$

En general, se cumple que

$$|f(n)| \leq c|g(n)| \text{ , } \forall n \geq n_0.$$

De lo que podemos decir que $f_e(n)$ es de orden $O(n)$.

Ejercicio 5: El máximo común divisor de dos enteros positivos n y m; denotado por MCD(n,m); es el único entero positivo k tal que k divide a m y n y todos los demás enteros que dividen a m y n son menores que k. Encuentre el orden O de complejidad temporal y espacial del algoritmo.

```
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;
}
```

Sol: