Soluciones

Práctico 2 y Práctico 2 Complemento

Análisis de algoritmos

Práctico 2

Ejercicio 1

Haga un análisis del tiempo de ejecución para el siguiente fragmento de programa:

1. en forma exacta (cantidad de operaciones)
2. definiendo `sum++ como Op básica`
3. dando el orden de ejecución

```
(6)   sum = 0;                                  (1)
      for (i = 1; i <= n; i++)                  (2)
         for (j = 1; j <= i * i; j++)           (3)
               if (j % i == 0)                  (4)
                     for (k = 1; k <= j; k++)   (5)
                           sum++;               (6)
```

1.

Considerando a las operaciones con ++ como de costo 1, se tiene:

$$T(n) = 1+1+\sum_{i=1}^{i=n}\left(3+\sum_{j=1}^{j=i^2}(5)+\sum_{h=1}^{h=i}\left(1+\sum_{k=1}^{k=h.i}(3)+1\right)+2\right)+1$$

El primer sumando referencia a la asignación de la línea (1). El segundo marca la inicialización de la variable i en la línea (2). El tercero cuenta la sentencia for de esa línea, de la siguiente manera:

Se suma de 1 hasta n (índices del for (2))

- 1 por comparación i <= n (2)
- 1 por asignación i++ (2)
- 1 por inicialización j = 1 (3)
- Suma de 1 hasta i*i (for de la línea (3) + evaluación de la condición del if (4))*
  - 1 por multiplicación i * i (3)
  - 1 por comparación j <= i * i (3)
  - 1 por asignación j++ (3)
  - Condición del if ya que la misma siempre se evalúa:
    - 1 por operación módulo j % i (4)
    - 1 por comparación j % i == 0 (4)
- Suma de 1 hasta i (for de la línea (3) cuando el de la línea (5) se ejecuta). Los índices varían entre 1 e i porque son los valores en los que la comparación (4) da *true*.†
  - 1 por la inicialización k = 1 (5)
  - Suma de 1 hasta h*i (for de la línea (5)).
    - 1 por comparación k <= j (5)
    - 1 por asignación k++ (5)
    - 1 por asignación sum++ (6)
  - 1 por último chequeo de condición del for de la línea (5)
- 2 por último chequeo de condición del for de la línea (3). Este último chequeo tiene costo 2 ya que se tiene el costo del producto i*i y el costo de la comparación j <= i * i

1 por último chequeo de condición del for de la línea (2)

Desarrollando esta suma se obtiene:

$$T(n) = 3 + \sum_{i=1}^{i=n}\left(5 + 5\sum_{j=1}^{j=i^2}(1) + \sum_{h=1}^{h=i}\left(2 + 3\sum_{k=1}^{k=h.i}(1)\right)\right)$$

$$T(n) = 3 + \sum_{i=1}^{i=n}\left(5 + 5i^2 + \sum_{h=1}^{h=i}(2) + 3\sum_{h=1}^{h=i}\sum_{k=1}^{k=h.i}(1)\right)$$

$$T(n) = 3 + \sum_{i=1}^{i=n}\left(5 + 5i^2 + 2i + 3\sum_{h=1}^{h=i} hi\right)$$

$$T(n) = 3 + \sum_{i=1}^{i=n}\left(5 + 5i^2 + 2i + 3i\sum_{h=1}^{h=i} h\right)$$

$$T(n) = 3 + 5\sum_{i=1}^{i=n}(1) + 5\sum_{i=1}^{i=n}(i^2) + 2\sum_{i=1}^{i=n}(i) + 3\sum_{i=1}^{i=n}\left(i\left(\frac{i.(i+1)}{2}\right)\right)$$

* El análisis de la ejecución del for de la línea (3) se realiza en dos bloques. En el primero se estudia la evaluación de la condición del if de la línea (4) y en el segundo la ejecución del for de la línea (5).

† Si *j* es múltiplo de *i* (necesario para que se cumpla condición del if y se ejecute el for de la línea (5)) entonces *j* se puede escribir como j = *h* * *i*. Por otro lado, se sabe que *j* toma los valores desde 1 a i^2 (for de la línea (2)). Por lo tanto, se tienen *i* múltiplos (j = 1*i, 2*i, 3*i, …, i*i). Es decir, *h* tomará los valores desde 1 hasta *i*.

$$T(n) = 3 + 5\sum_{i=1}^{i=n}(1) + 5\sum_{i=1}^{i=n}(i^2) + 2\sum_{i=1}^{i=n}(i) + \frac{3}{2}\sum_{i=1}^{i=n}(i^3 + i^2)$$

$$T(n) = 3 + 5n + 5\left(\frac{n(n+1)(2n+1)}{6}\right) + 2\left(\frac{n(n+1)}{2}\right) + \frac{3}{2}\left(\frac{n(n+1)}{2}\right)^2 + \frac{3}{2}\left(\frac{n(n+1)(2n+1)}{6}\right)$$

$$T(n) = 3 + 5n + \frac{5}{6}\left((n^2+n)(2n+1)\right) + n^2 + n + \frac{3}{2}\left(\frac{(n^2+n)^2}{4}\right) + \frac{3}{12}\left((n^2+n)(2n+1)\right)$$

$$T(n) = 3 + 5n + \frac{5}{6}(2n^3 + 3n^2 + n) + n^2 + n + \frac{3}{8}(n^4 + 2n^3 + n^2) + \frac{3}{12}(2n^3 + 3n^2 + n)$$

$$T(n) = 3 + n\left(5 + \frac{5}{6} + 1 + \frac{3}{12}\right) + n^2\left(\frac{5}{2} + 1 + \frac{3}{8} + \frac{3}{4}\right) + n^3\left(\frac{5}{3} + \frac{3}{4} + \frac{3}{6}\right) + n^4\left(\frac{3}{8}\right)$$

$$T(n) = 3 + n\left(\frac{85}{12}\right) + n^2\left(\frac{37}{8}\right) + n^3\left(\frac{35}{12}\right) + n^4\left(\frac{3}{8}\right)$$

$$T(n) = \frac{3n^4}{8} + \frac{35n^3}{12} + \frac{37n^2}{8} + \frac{85n}{12} + 3$$

2.

Considerando sum++ como la operación básica la expresión se simplifica de la siguiente manera:

$$T(n) = \sum_{i=1}^{i=n}\left(\sum_{h=1}^{h=i}\left(\sum_{k=1}^{k=h.i}(1)\right)\right)$$

3.

A partir del resultado de la parte 1: $T(n) = \frac{3n^4}{8} + \frac{35n^3}{12} + \frac{37n^2}{8} + \frac{85n}{12} + 3$

Se puede decir que: $T(n) = O(n^4)$

Esta información se puede extraer también de la parte 2, teniendo en cuenta que su resultado da una expresión que contiene n^4 como término de mayor grado.

Ejercicio 2

$t(n) = 3seg - 18.\text{n miliseg} + 27.n^2 \text{ microseg} \Rightarrow t(n) = 3.10^6 - 18.10^3.n + 27.n^2 \text{ microseg}$

Si se estudian los ceros de dicho polinomio se obtiene que el polinomio tiene raíz doble n = 333,33. A su vez, si se estudia la derivada $t'(n) = 54.n - 18.10^3$ se observa que la función dada es creciente para $n \geq 333$.
Encontrar f(n) tal que $t(n) \in O(f(n))$ significa encontrar $(c \in R^+)$ y $(n_0 \in N)$ tal que $\forall n > no$, $t(n) \leq c.f(n)$.

Por lo anteriormente visto, se puede considerar $n_0 = 333$. Falta encontrar c y f que cumplan la condición planteada. El hecho de que t(n) sea un polinomio de segundo grado, sugiere considerar $f(n) = n^2$. Para verificar que $t(n) \in O(n^2)$ se debe encontrar c real positivo tal que $3.10^6 - 18.10^3.n + 27.n^2 \leq c * n^2$ para $n > 333$. Pero para $n > 333$, $3.10^6 \leq 18.10^3.n$ de donde alcanza con encontrar c tal que $27.n^2 \leq c * n^2$.

Tomando entonces c = 27, $t(n) \in O(n^2)$ por def.

Ejercicio 4

Considerando:

- f y g son funciones que retornan únicamente valores positivos
- $f > g$ si $\exists n_0 \in N, \forall n \in N, n > n_0, f(n) > g(n)$

Demostrar:

1. **$f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$.**

   Utilizando las definiciones de O y Ω:

   i) $f(n) \in O(g(n))$ si y solo si $\exists\ c_1 \in R^+, n_1 \in N / \forall\ n > n_1\ f(n) \leq c_1.g(n)$

   ii) $g(n) \in \Omega(f(n))$ si y solo si $\exists\ c_2 \in R^+, n_2 \in N / \forall\ n > n_2\ g(n) \geq c_2.f(n)$

   (→)

   A partir de i) se tiene que: $\forall\ n > n_1\ \frac{1}{c_1} * f(n) \leq g(n)$

   Si se elige $n_2 = n_1$ y $c_2 = 1/c_1$:

   $g(n) \geq c_2.f(n)\ \forall\ n > n_2$ entonces $g(n) \in \Omega(f(n))$

   (←)

   La demostración en el otro sentido es análoga.

Ejercicio 10

Escribir un programa C/C++ de ordenación por selección que ordene un arreglo de enteros en orden ascendente mediante la localización del valor más pequeño en cada iteración y el intercambio de éste con el componente apropiado.

Calcule el orden del tiempo de ejecución, tomando como operación básica la comparación de 2 elementos del arreglo, en el mejor caso, caso promedio y peor caso, dando un ejemplo de entrada para cada caso. Por último, compare los resultados obtenidos.

Una posible implementación es la siguiente:

```
#include <stdio.h>
#include <stdlib.h>

void SelectionSort (int* a, int n){
    int min, tmp;
(1) for (int i = 0; i < n-1; i++){
        min = i;
(2)     for (int j = i + 1; j < n; j++){
(3)         if (a[j] < a[min])
                min = j;
        }
       // Intercambio de elementos
        tmp = a[i];
        a[i] = a[min];
        a[min] = tmp;
    }
}
```

Para analizar el tiempo de ejecución del algoritmo se quiere contar únicamente la cantidad de veces que se evalúa condición del `if` (3). Es decir, el tiempo de ejecución está dado por la siguiente sumatoria:

$$T(n) = \sum_{i=0}^{i=n-2}\left(\sum_{j=i+1}^{j=n-1}(1)\right) = \sum_{i=0}^{i=n-2}(n-i-1) = \sum_{i=0}^{i=n-2}(n-1) - \sum_{i=0}^{i=n-2}(i) =$$

$$(n-1)^2 - \frac{(n-2)(n-1)}{2} = (n-1)\left(n-1-\left(\frac{n-2}{2}\right)\right) =$$

$$(n-1)\left(\frac{2n-2-n+2}{2}\right) = \frac{n(n-1)}{2}$$

Notar que *T* depende únicamente del largo del arreglo de entrada y no de la información que éste contenga. Por tanto, mejor caso, caso promedio y peor caso tienen el mismo tiempo de ejecución para un largo dado.

$$T(n) = \frac{n^2 - n}{2} \in O\left(n^2\right)$$

A continuación se muestra una rutina para probar esta función.

```
#include <iostream.h>
#include <limits.h>
#include <assert.h>
#include <time.h>

int main (){
    const int N = 100;
    const float MAX = 1000;
    int *a = new int[N];
    int valor = INT_MIN;

    srand (time(0));
    // asigna la semilla para la funcion que obtiene numeros aleatorios
    int i;
    for (i = 0; i < N; i++){
        a[i] = (MAX * rand ()) / (RAND_MAX + 1.0);
        /*
         * se asigna a la posicion i del vector a un valor aleatorio
         * entre 1 y MAX. Por mas informacion sobre la funcion rand
         * ejecutar "man rand" en la consola de cygwin
         */
        cout << a[i] << " ";
    }
    cout << "\n\n";

    SelectionSort (a, N);

    for (i = 0; i < N; i++){
        assert (valor <= a[i]);
        valor = a[i];
        cout << a[i] << " ";
    }
    cout << "\n\n";
    delete a;
    return 0;
}
```

Ejercicio 16

Suponga $T_1(n) \in \Omega(f(n))$ y $T_2(n) \in \Omega(g(n))$, $f(n), g(n) \geq 0 \ \forall n$. Cuales de las siguientes sentencias son verdaderas?

1. $T_1(n) + T_2(n) \in \Omega(\max(f(n), g(n)))$.

   La afirmación es **VERDADERA**.

   (a) Como $T_1(n) \in \Omega(f(n))$ entonces existen $c_1 \in R^+, n_1 \in N / \forall\ n > n_1\ T_1(n) \geq c_1.f(n)$.

   (b) Como $T_2(n) \in \Omega(g(n))$ entonces existen $c_2 \in R^+, n_2 \in N / \forall\ n > n_2\ T_2(n) \geq c_2.g(n)$.

   Por (a), como $c_1 \in R^+$ y $f(n) \in R^{*\ddagger} \Rightarrow T_1(n) \geq 0$, entonces $T_2(n) + T_1(n) \in \Omega(g(n))$

   Por (b) análogamente $T_2(n) \geq 0$, entonces $T_1(n) + T_2(n) \in \Omega(f(n))$

   Como consecuencia se tiene que $T_1(n) + T_2(n) \in \Omega(\max(f(n), g(n)))$

2. $T_1(n) / T_2(n) \in \Omega(f(n).g(n))$.

   La afirmación es **FALSA**.

   En particular cuando $T_1(n) \in \Omega(n)$ y $T_2(n) \equiv T_1(n)$ se tiene:

   $T_1(n) / T_2(n) \equiv 1$, y $T_1(n) / T_2(n) \notin \Omega(n^2)$

‡ Recuerde que $R^* = R^+ \cup \{\emptyset\}$

Práctico 2 Complemento

Ejercicio 2

Dada la función: f(x) = $\sum_{i=0}^{n} a_i x^i$

Analice el tiempo de ejecución en forma exacta (cantidad de operaciones) y compare los órdenes en los siguientes casos (considere n potencia de 2):

a. Utilizando una rutina simple para realizar la exponenciación.
b. Utilizando la siguiente rutina:

```
int potencia (int x, int n){
  int resultado;

  if (n == 0)
        resultado = 1;
  else if (n == 1)
        resultado = x;
  else if (n % 2 == 0)
        resultado = potencia (x * x, n / 2);
  else
        resultado = potencia (x * x, n / 2) * x;
  return (resultado);
}
```

Parte a:

Una rutina simple que puede ser implementada para resolver la exponenciación puede ser de la siguiente forma:

```
(1)     int potencia(int x, int n){
(2)           int pot;
(3)           pot = 1;
(4)           for (int i = 1; i <= n; i++)
(5)                 pot = pot * x;
(6)           return pot;
        }
```

Si se considera la cantidad de operaciones en forma exacta[**], se obtiene:

- 1 por inicialización de pot (3)
- 1 por inicialización de i (4)
- Suma de 1 a n (4)
  - 1 por la comparación 1<=n (4)
  - 1 por la asignación i++ (4)
  - 1 por la asignación pot = pot * x (5)
  - 1 por la multiplicación pot * x (5)
- 1 última comparación i <= n (4)

[**] Las operaciones que se tienen en cuenta en esta oportunidad son: suma, resta, multiplicación, módulo, asignación y comparación. El retorno del resultado no se contabilizará como operación.

$$T(n)=1+1+\sum_{i=1}^{n}(4)+1$$
$$T(n)=3+4n$$

Por lo tanto la cantidad de operaciones en forma exacta es $T(n)=3+4n$ y es de orden O(n).

Parte b:

Cantidad de operaciones en forma exacta para la función *potencia*:

```
     int potencia(int x,int n){
        int resultado;
(1)     if (n == 0)
(2)        resultado = 1;
(3)     else  if (n == 1)
(4)                resultado = x;
(5)           else  if (n % 2 == 0)
(6)                   resultado = potencia (x * x, n / 2);
(7)                else
(8)                   resultado = potencia (x * x, n / 2) * x;
(9)     return (resultado);
        }
```

A continuación se muestra la cantidad de operaciones realizadas para cada "camino":

(1) Comparación (n==0) → 1 operación
Si la comparación es
VERDADERA → asignación (resultado = 1) → 1 operación.
FALSA →(3) Comparación (n == 1) → 1 operación.
Si la comparación es
VERDADERA → asignación (resultado = x) → 1 operación.
FALSA → (5) Comparación (n % 2 == 0) → 1 operación
(5) módulo (n % 2) → 1 operación
Si la comparación es
VERDADERA → asignación →1 operación
multiplicación → 1 operación
división → 1 operación
#{operaciones(potencia(x^2,n/2))}
FALSA → asignación →1 operación
multiplicación → 1 operación
división → 1 operación
#{operaciones(potencia(x^2,n/2))}
multiplicación → 1 operación

Por lo tanto se puede definir *T(n)* como:

$$T(n)=\begin{cases}7+T(n/2) & \text{si } n>1 \ \wedge \ n=\dot{2}\\ 8+T(n/2) & \text{si } n>1 \ \wedge \ n\neq\dot{2}\\ 2 & \text{si } n=0\\ 3 & \text{si } n=1\end{cases}$$

Para analizar la cantidad de operaciones se va a analizar el peor caso. Se puede observar que el peor caso se da en el camino **F→F→F→...→F→...-->V** dónde se hace verdadero solamente en el caso base (cuando *n* = 1). Para que esto ocurra *n* debe ser impar, y a su vez, cada llamada recursiva debe ser impar.

Si se toma *n* impar de la forma: $2^k - 1$ se cumple que cada nueva invocación a la función *potencia* se hace con un *n* impar.

$$T_W(n) = 1 + 1 + 2 + 3 + T_W\left(\tfrac{n}{2}\right) + 1$$

Es decir, $T_W(n) = 8 + T_W\left(\tfrac{n}{2}\right)$

Si se expande esta fórmula se obtiene:

$$T_W(n) = 8 + T_W\left(\tfrac{n}{2}\right)$$

$$T_W\left(\tfrac{n}{2}\right) = 8 + T_W\left(\tfrac{n}{2^2}\right)$$

$$T_W\left(\tfrac{n}{2^2}\right) = 8 + T_W\left(\tfrac{n}{2^3}\right)$$

...

$$T_W\left(\tfrac{n}{2^{k-2}}\right) = 8 + T_W\left(\tfrac{n}{2^{k-1}}\right)$$

Si se suma la expansión anterior, se cancelan la mayoría de los términos, y se obtiene la siguiente ecuación:

$$T_W(n) = 8(k-1) + T_W\left(\tfrac{n}{2^{k-1}}\right)$$

Si se considera la observación hecha anteriormente sobre *n* se tiene que:

$$\frac{n}{2^{k-1}} = \frac{2^k - 1}{2^{k-1}} = 2 - \frac{1}{2^{k-1}}$$

Por lo tanto, como la invocación se hace con la división entera se obtiene:

$$T_W(n) = 8(k-1) + T_W(1) \text{ y } T_W(1) = 3$$

Entonces, $T_W(n) = 8(k-1) + 3$

A su vez, se condicionó *n* a ser de la forma: $n = 2^k - 1$

Entonces, si se despeja y aplica logaritmo en base dos, se obtiene: $\log(n+1) = k$.

Si se sustituye en $T_W(n)$:

$$T_W(n) = 8(\log(n+1) - 1) + 3$$

Por lo tanto la cantidad de operaciones en forma exacta para el peor caso es $T_W(n) = 8\log(n+1) - 5$ y es de orden $O(\log(n))$

Si se considera la siguiente rutina para calcular *f(x):*

```
      int eval_pol (int a[], int x, int n){
          int sum;
(1)       sum = 0;
(2)       for (int i = 0; i <= n; i++)
(3)             sum = sum + potencia (x, i) * a[i];
          return sum;
      }
```

la cantidad de operaciones en forma exacta, se obtiene de la siguiente manera:

- 1 por la asignación sum = 0 (1)
- 1 por la inicialización i = 0 (2)
- Suma de 0 a n (2)
  - 1 por la comparación (i <= n) (2)
  - 1 por la asignación i++ (2)
  - 1 por la asignación (3)
  - 1 por la suma (3)
  - 1 por la multiplicación (3)
  - Cantidad de operaciones de la subrutina potencia. (3)
- 1 última comparación i <= n) (2)

La cantidad de operaciones en forma exacta para la rutina principal aplicando el algoritmo de la parte a es:

$$T(n) = 3 + \sum_{i=0}^{i=n} (5 + 3 + 4i)$$

Desarrollando esta ecuación: $T(n) = 3 + \sum_{i=0}^{i=n} 8 + 4\sum_{i=0}^{i=n} i = 3 + 8(n+1) + 4\frac{n.(n+1)}{2}$

Por lo tanto para el primer caso la cantidad exacta de operaciones es: $T(n) = 2n^2 + 10n + 11$ y la función es de **O(n^2)**.

La cantidad de operaciones en forma exacta para la rutina principal aplicando el algoritmo de la parte b es:

$$T(n) = 3 + \sum_{i=0}^{i=n} (5 + 8\log(i+1) - 5)$$

Si se desarrolla esta ecuación:

$$T(n) = 3 + \sum_{i=0}^{i=n} 8\log(i+1) = 3 + 8\sum_{i=0}^{i=n} \log(i+1)$$

$$T(n) = 3 + 8\log\left(\prod_{i=1}^{i=n} (i+1)\right) = 3 + 8\log((n+1)!)^{\dagger\dagger}$$

Por lo tanto para el segundo caso la cantidad exacta de operaciones es: $T(n) = 8\log((n+1)!) + 3$ y la función es de $O(n.\log n)^{\ddagger\ddagger}$

[††] La productoria comienza en *i* = 1 ya que log(1) = 0.

[‡‡] Aquí se aplicó propiedades de logaritmos, acotando log ((n-1)!) por log (n^n).