INF 263 – Algoritmia

Complejidad Computacional

MG. JOHAN BALDEON

MG. RONY CUEVA

2021-2

CAPITULO I – Complejidad Computacional

Cálculo de T(n)

En lugar de evaluar el tiempo de ejecución, evaluamos la cantidad de operaciones elementales

Entonces para el cálculo de T(n) es importante tomar en cuenta lo siguiente:

- Considerar que el tiempo de una OE es, por definición, de orden O(1).
- El tiempo de ejecución de una secuencia consecutiva de instrucciones se calcula sumando los tiempos de ejecución de cada una de las instrucciones

Reglas generales de cálculo

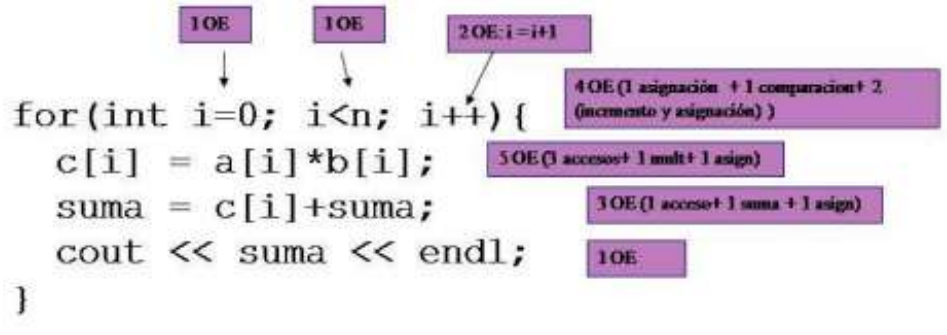
Regla 1 – ciclos

El tiempo de ejecución de un ciclo, es a lo mas el tiempo de ejecución de las instrucciones que están en el interior del ciclo (incluyendo las condiciones) por el número de iteraciones.

Reglas generales de cálculo

Regla 1 – ciclos

Veamos el siguiente ejemplo:



Tiempo total en el peor caso: $1+1+\sum_{i=0}^{n-1}(\#\text{OE de instrucciones}+2+1)$

$$1+1+\sum_{i=0}^{n-1}(9+2+1)=2+12n \in \mathbf{O}(n)$$

Reglas generales de cálculo

Regla 2 – ciclos anidados

Analizarlos de adentro hacia fuera. El tiempo de ejecución total de una proposición dentro del grupo de ciclos anidados es el tiempo de ejecución de la proposición multiplicado por el producto de los tamaños de todos los ciclos.

Como un ejemplo, el siguiente fragmento de programa es $O(n^2)$.

```
for i := 1 to n do
      for j := 1 to n do
            k := k + 1;
```

Reglas generales de cálculo

Regla 2 – ciclos anidados

La complejidad de dos ciclos anidados cuya condición de control depende de *n* es ***O***(n^2) mientras que la complejidad de ciclos for separados es ***O***(*n*). Así es que cuando *n* es grande, hay que procurar poner los ciclos for por separado cuando sea posible. A continuación se muestra el cálculo del tiempo total en el peor caso para el for anidado:

Número de operaciones elementales para el *for* anidado en el peor caso**:**

$$1 + 1 + \sum_{i=0}^{n-1} (\#OE\ for\ interno + 2 + 1)$$

Reglas generales de cálculo

Regla 2 – ciclos anidados

Sea *x* el número de instrucciones dentro del ciclo interno y el número de operaciones elementales es, en el peor caso:

$$1 + 1 + \sum_{i=0}^{n-1} (x + 2 + 1)$$

Entonces el número de OE del ciclo for anidado, en el peor caso, es:

$$\mathbf{1} + 1 + \sum_{i=0}^{n-1} \left[1 + 1 + \sum_{j=0}^{n-1} (x + 2 + 1) + 2 + 1 \right] = 2 + n(2 + n(3 + x) + 3)$$

$$= 2 + 5n + (3 + x)n^2 \in O(n^2)$$

Reglas generales de cálculo

Regla 2 – ciclos anidados

Ahora, si calculamos el número de OE para dos ciclos for consecutivos, tenemos que si *x1* es el número de instrucciones dentro del primer ciclo for, y *x2* es el número de instrucciones dentro del segundo ciclo for, el número de operaciones elementales en el peor caso es, para el primer ciclo**:**

$$1 + 1 + \sum_{i=0}^{n-1} (x_1 + 2 + 1)$$

Y para el segundo ciclo::

$$1 + 1 + \sum_{i=0}^{n-1} (x_2 + 2 + 1)$$

Reglas generales de cálculo

Regla 2 – ciclos anidados

En total:

$$1+1+\sum_{i=0}^{n-1}(x_1+2+1)+\ 1+1+\sum_{i=0}^{n-1}(x_2+2+1)$$

$$=\ 4+n(\ 3+x_1)+2+n(3+x_2)\ \in O(n)$$

Reglas generales de cálculo

Regla 2 – ciclos anidados

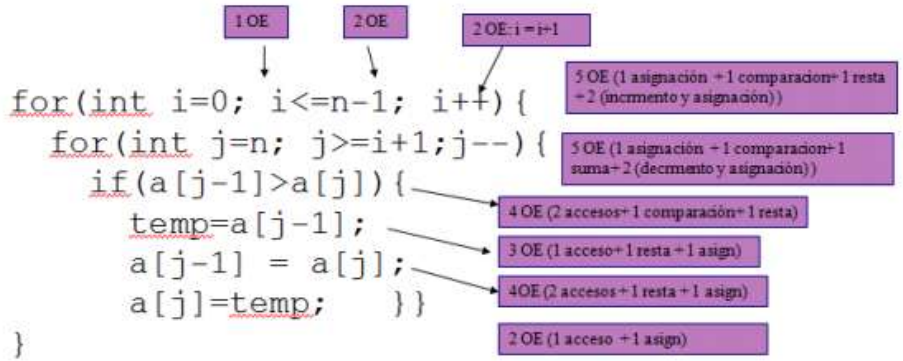


Figura 2.10: Tiempo de ejecución de un algoritmo que ordena un arreglo de menor a mayor

Para calcular el número total de instrucciones:

$$1 + 2 + \sum_{i=0}^{n-1} (\#forinterno + 2 + 2)$$

Reglas para calcular complejidad

Parte 2

Reglas generales de cálculo

Regla 3 – proposiciones consecutivas

Simplemente se suman, lo cual significa que el máximo es el único que cuenta; *véase la Regla 1*.

Como ejemplo, el siguiente fragmento de programa, que tiene trabajo $O(n)$ seguido de trabajo $O(n^2)$, también es $O(n^2)$:

```
for i := 1 to n do
  a[i] := 0;
for i := 1 to n do
  for j := 1 to n do
    a[i] := a[i] + a[j] + i + j;
```

Reglas generales de cálculo

Regla 4 – llamados a funciones

El tiempo de ejecución de una llamada a una función F(P1, P2,..., Pn) es 1 (por la llamada), más el tiempo de evaluación de los parámetros P1, es decir, el tiempo que se emplea en obtener el valor de cada parámetro, P2,..., Pn, más el tiempo que tarde en ejecutarse F, esto es, T = 1 + T(P1) + T(P2) + ... + T(Pn) + T(F).

Reglas generales de cálculo

Regla 5 – Condicional “if”

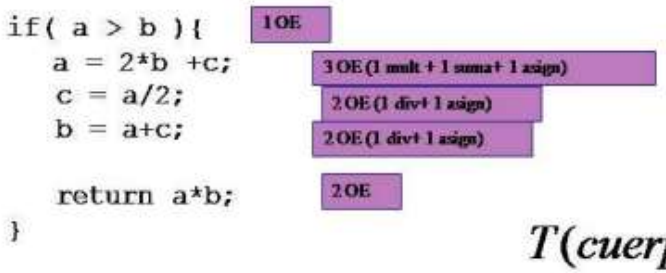
El tiempo de ejecución de una instrucción condicional “if” es el costo de las instrucciones que se ejecutan condicionalmente, más el tiempo para evaluar la condición.

$$\mathbf{O}(\text{Tiempo(if)}) = \mathbf{O}(\text{Tiempo(condición)}) + \mathbf{O}(\text{Tiempo(cuerpo)}).$$

Reglas generales de cálculo

Regla 5 – Condicional "if"

Veamos el siguiente ejemplo:



$$T(cuerpo) = 9 \in \mathbf{O}(1)$$

$$\mathbf{O}(\text{Tiempo}(if)) = \mathbf{O}(1) + \mathbf{O}(1) = \mathbf{O}(1)$$

Reglas generales de cálculo

Regla 6 – Condicional "if-else"

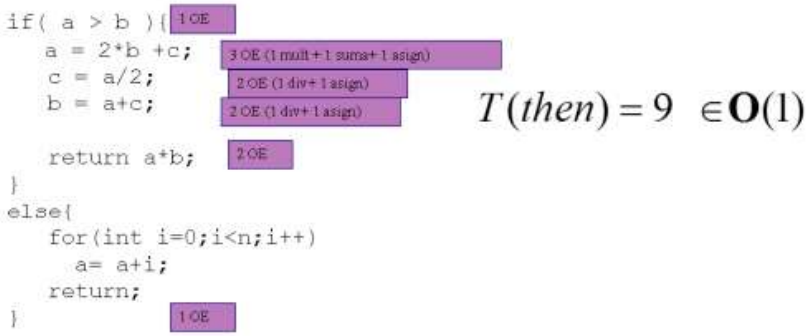
El tiempo de ejecución es el tiempo para evaluar la condición más el mayor entre los tiempos necesarios para ejecutar las proposiciones cuando la condición es verdadera y el tiempo de ejecución de las proposiciones cuando la condición es falsa.

$$\mathbf{O}(\text{Tiempo(if-else)}) = \mathbf{O}(\text{Tiempo(condición)}) + \text{Max}(\mathbf{O}(\text{Tiempo(then)}), \mathbf{O}(\text{Tiempo(else)}))$$

Reglas generales de cálculo

Regla 6 – Condicional "if-else"

Veamos el siguiente ejemplo:



$$T(then) = 9 \quad \in \mathbf{O}(1)$$

El tiempo de ejecución del else es el siguiente (se incluye el 1 del return):

$$Tiempo(else) = 1 + 1 + \left[\sum_{i=0}^{n-1} 2 + 2 + 1\right] + 1 = 3 + 5n$$

$$O\big(Tiempo(if - else)\big) = O(1) + \max\big(O(1), O(n)\big) = O(n)$$

Reglas generales de cálculo

Veamos el siguiente ejercicio :

```
if(n%2 == 0){
  for(int i=1; i<= n; i++)
     x++;
}
```

Podemos calcular el orden de complejidad de este algoritmo calculando el número de operaciones elementales, y así tenemos que:

$$T(n) = 2 + \left(1 + 1 + \left(\sum_{i=1}^{n} 2 + 2 + 1\right)\right) = 4 + 5n \in \mathbf{O}(n)$$

Reglas generales de cálculo

Calcular *T(n)* para el algoritmo "intercambiar":

```
void intercambiar (int *x, int *y){
    int *aux;
    *aux = *x;
    *x = *y;
    *y = *aux;
}
```

En este caso se trata de una función (subrutina) la cual recibe y regresa dos parámetros, entonces hay que tomar en cuenta la *regla 4*:

T(*n*) = 1 + *Tiempo*(*P*1) + *Tiempo*(*P*2) + ... + *Tiempo*(*Pn*) + *Tiempo*(*F*).

Como solo se tienen dos parámetros y *Tiempo(F)* = 3, tenemos que:

$$Tiempo(Intercambiar) = 1 + 1+1+3 = 6 \in \mathbf{O}(1)$$

Reglas generales de cálculo

Calcular *T(n)* para el algoritmo Sumatoria:

```
float Sumatoria (int n){
   float suma=0;
   for( int i=1; i<=n; i++ )
      suma = suma+i;
    return suma;
}
```

Podemos calcular el orden de complejidad de este algoritmo calculando el número de operaciones elementales y así tenemos que:

$$T(n) = 1 + 1 + 1 + 1 + 1 + \left[\sum_{i=1}^{n} 2 + 2 + 1\right] + 1 = 6 + 5n \in \mathbf{O}(n)$$

Llamado

Parámetro

suma=0

Regreso

Reglas para calcular complejidad

Parte 3

Reglas generales de cálculo

Regla 7 – Algoritmos recursivos

Un algoritmo ***recursivo*** consiste en obtener la solución total de un problema a partir de la solución de casos más sencillos de ese mismo problema. Estos casos se resuelven invocando al mismo algoritmo, el problema se simplifica hasta llegar al caso más sencillo de todos llamado *caso base*, cuya solución se da por definición.

El aspecto de un algoritmo recursivo es el siguiente:

```
ALGORITMO Recursivo(caso)
INICIO
 if caso = casoBase
    regresa solucionPorDefinicion
 else
    .....
    s1 = Recursivo(casoReducido1)
    s2 = Recursivo(casoReducido2)
    ...
    regresa SolucionDelCaso(s1,s2,...)
FIN
```

Reglas generales de cálculo

Regla 7 – Algoritmos recursivos

Por ejemplo, usando un algoritmo recursivo para obtener el factorial de un número natural tenemos la siguiente expresión, a la cual se le llama *ecuación de recurrencia*, porque la función está dada en términos de la propia función.

$$factorial(n) = \begin{cases} 1, & si\ n = 0 \\ n * factorial(n-1), & si\ n > 0 \end{cases}$$

```
long int factorial( long int n ) {
   long int s1;
   if  (n <= 1)
      return 1;
   else {
      s1= factorial(n-1);
      return n*s1;
   };
};
```

La función se llama a sí misma.

Reglas generales de cálculo

Regla 7 – Algoritmos recursivos

El tiempo de ejecución de un algoritmo recursivo se expresa por medio de una *ecuación de recurrencia,* la cual es una función $T(n)$. En las *recurrencias* se separa el tiempo de ejecución del caso básico del tiempo de ejecución del caso recursivo. En este último caso, se utiliza la misma función $T(n)$ para representar el tiempo de ejecución de la llamada recursiva, por ejemplo:

En el caso del factorial, si consideramos que el caso base tiene un tiempo de ejecución k_1 y que el resto de los casos tienen un tiempo de ejecución $T(n-1) + k_2$ podemos substituir n por $n-1$ para obtener $T(n-1)$. Así, el factorial queda expresado con la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} k_1 & n = 0 \\ T(n\text{-}1) + k_2 & n > 0 \end{cases}$$

Reglas generales de cálculo

Ejercicio 1.- Dada la ecuación de recurrencia de la sección anterior, correspondiente al algoritmo que obtiene el factorial de manera recursiva, encontrar su solución y su cota superior de complejidad (notación **O**).

$$T(n) = T(n-1) + k_2$$

Ahora usemos la misma definición para hallar $T(n-1)$

$$T(n-1) = T(n-2) + k_2$$

Siguiendo con esta lógica resolveremos $T(n)$

Reglas generales de cálculo

$$T(n) = T(n-1) + k_2$$
$$T(n-1) = T(n-2) + k_2$$
$$T(n-2) = T(n-3) + k_2$$
$$\vdots \qquad \vdots \qquad \vdots$$
$$T(n-i+1) = T(n-i) + k_2$$

Si sumamos todos los lados izquierdos, debería ser iguala la suma de todos los lados derechos. Por lo tanto expresiones iguales a ambos lados se cancelan

$$T(n) = T(n-i) + ik_2 \Rightarrow T(n) = k_1 + nk_2 \Rightarrow \boxed{T(n) = O(n)}$$

Para poder resolver $T(n-i)$ nos conviene escoger un valor de i que facilite la solución. De la ecuación de recurrencia sabemos que $T(0) = k_1$. Por lo tanto nos conviene $i = n$

Reglas generales de cálculo

Ejercicio 2.- Si tenemos la siguiente función recursiva, encontrar la solución a su ecuación de recurrencia y la cota superior de la complejidad de esta función:

```
int Recursiva1( int n ) {
   if (n <= 1)
     return (1);
   else
     return ( 2*Recursiva1(n/2) );
}
```

Reglas generales de cálculo

Solución: La ecuación de recurrencia para la función Recursiva1 es:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n/2) + 1 & n > 1 \end{cases}$$

Aplicaremos la técnica de reemplazo para solucionar esta ecuación de recurrencia

Reglas generales de cálculo

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

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

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

$$\vdots \qquad \vdots \quad \vdots$$

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

$$T(n) = T\left(\frac{n}{2^i}\right) + i \Rightarrow T(n) = T(1) + \log_2 n \Rightarrow \boxed{T(n) = O(\log n)}$$

Nos conviene $\frac{n}{2^i} = 1$, para poder usar el caso base de la recurrencia

Por lo tanto, $n = 2^i$, entonces $i = \log_2 n$

Reglas generales de cálculo

Ejercicio 3.- Encontrar la solución a su ecuación de recurrencia y la cota superior de la complejidad de esta función:

```
int Rec3 (int n){
   if (n <= 1)
     return (1);
   else
     return ( Rec3(n-1) + Rec3(n-1) );
}
```

Reglas generales de cálculo

Solución: La ecuación de recurrencia para la función Rec3 es:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(n\text{-}1) + 1 & n > 1 \end{cases}$$

Usaremos la técnica de reemplazo para este problema

Reglas generales de cálculo

$$T(n) = 2T(n-1) + 1$$

$$T(n-1) = 2T(n-2) + 1$$

Para poder eliminar el término $T(n-1)$ de la izquierda, con el de la derecha, tenemos que multiplicar por 2 a toda la ecuación de abajo

Reglas generales de cálculo

$$T(n) = 2T(n-1) + 1$$
$$2T(n-1) = 2^2T(n-2) + 2$$
$$2^2T(n-2) = 2^3T(n-3) + 2^2$$
$$\vdots \qquad \vdots \qquad \vdots$$
$$2^{i-1}T(n-i+1) = 2^iT(n-i) + 2^{i-1}$$

Hacemos $i = n - 1$, de esa forma $T(n-i) = 1$

Resolviendo la sumatoria de la progresión geométrica, tenemos:

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

$$\boxed{T(n) = 2^n - 2 = O(2^n)}$$

Bibliografía recomendada

- LEVITIN, A. **Introduction to The Design and Analysis of Algorithms**. 3ra edición. USA: Pearson, 2012. ISBN 0-13-231681-1.
- Mark Allen Weiss. **Estructuras de Datos y Algoritmos, Addison-Wesley Iberoamericana,** 1995, pp. 17-44.