Estructura de Datos y Algoritmos

Introducción al análisis de algoritmos

1. Eficiencia en tiempo de Ejecución

Cuando resolvemos un problema nos vemos frecuentemente enfrentando una elección de programas, es usual tener más de un programa para resolver un mismo problema, por ejemplo, ordenamiento. En base a que elegimos? Usualmente hay dos objetivos contradictorios:

1. podemos querer un algoritmo facil de entender, codificar y poner a punto.
2. podemos querer un algoritmo que haga un uso eficiente de los recursos de máquina (como ser tiempo y espacio), en particular uno que se ejecute lo más rápido posible.

Cuando escribimos un programa que utilizaremos pocas veces, el objetivo 1) es más importante. Nos importa el tiempo que le lleva al programador codificar el programa luego el costo a optimizar es el costo de escribir el programa.

Cuando nos enfrentamos a un problema cuya solución será utilizada muchas veces, el costo de ejecutar el programa es más importante que el costo de escribirlo. Luego vale la pena implementar un algoritmo complicado dado que el programa resultante se ejecutará más rapido.

2. Midiendo el tiempo de ejecución de un programa

El tiempo de ejecución de un programa depende de factores tales como:

1. tamaño o valor de la entrada al programa
2. calidad del código generado por el compilador
3. naturaleza y rapidez de las instrucciones en la máquina utilizada para ejecutar el programa
4. complejidad en el tiempo del algoritmo representado por el programa

El hecho de que el tiempo de ejecución dependa de la entrada nos dice que el tiempo de ejecución de un programa debe definirse como una función de la entrada (o su tamaño). Un buen ejemplo es el de ordenamiento, en este caso tenemos como entrada una lista de elementos a ser ordenados y producimos como salida los mismos elementos en orden. Por ejemplo dados $2, 1, 3, 1, 5, 8$ como entrada produciremos $1, 1, 2, 3, 5, 8$ como salida. La medida natural de la entrada en este caso es el largo de la entrada. En general consideraremos el tamaño de la entrada para medir la complejidad de los programas a menos que se diga lo contrario.

Hablaremos de $T(n)$ que representa el tiempo de ejecución de programas en entradas de tamaño n. Pensaremos en $T(n)$ como indicando el número de instrucciones ejecutadas en la computadora.

Definiremos $T(n)$ como el tiempo de ejecución **en el peor caso** esto es el máximo entre las entradas de tamaño n del tiempo de ejecución del programa.

Podemos considerar también $T_{avg}(n)$, el promedio entre las entradas de tamaño n del tiempo de ejecución en esa entrada. En la práctica es mucho más dificil determinar el tiempo promedio que el tiempo en el peor caso, por lo cual, utilizaremos este último como principal medida de complejidad.

Consideremos ahora 2) y 3): que el tiempo de ejecución depende del compilador y de la máquina. Estos hechos implican que no podemos expresar el tiempo en medidas estandar como segundos. En cambio diremos por ejemplo : el tiempo de ejecución de tal programa es proporcional a n^2.

3. Notaciones O y "Ω"

Para hablar sobre tiempos de ejecución de programas utilizaremos lo que se conoce como notación ."O"mayuscula. Por ejemplo cuando decimos que el tiempo de ejecución de un programa es $O(n^2)$ lo que queremos decir es que:

existen constantes positivas c y n_0 tal que para todo n mayor o igual que n_0 se cumple $T(n) \leq cn^2$

En general definimos:

Definición 3.1 (O grande) *$T(n) = O(f(n))$ si existen constantes positivas c Y n_0 tal que $T(n) \leq c.f(n)$ para todo $n \geq n_0$.*

Ejemplo 3.2 *Supongamos $T(0) = 1$, $T(1) = 4$ y en general $T(n) = (n+1)^2$. Vemos entonces que $T(n)$ es $O(n^2)$. Podemos tomar $n_0 = 1$ y $c = 4$. Luego tenemos que para $n \geq 1$ se cumple $(n+1)^2 \leq 4n^2$. Observar que no podemos tomar $n_0 = 0$ pues $T(0) = 1$ no es menor que $c0^2 = 0$ cualquiera sea c.*

En lo que sigue asumiremos las funciones del tiempo de ejecución se definen en los enteros no negativos y que sus tiempos de ejecución son no negativos aunque no necesariamente enteros.

Decimos $T(n)$ es $O(f(n))$ si existen constantes c y n_0 tal que $T(n) \leq cf(n)$ para todo $n \geq n_0$. Cuando un programa tiene tiempo de ejecución $f(n)$ decimos tiene velocidad de crecimiento $f(n)$.

Ejemplo 3.3 *Consideremos la función $T(n) = 3n^3 + 2n^2$. Es $O(n^3)$. Para ver esto consideremos $n_0 = 0$ y $c = 5$. Podemos ver que para $n \geq 0$ $3n^3 + 2n^2 \leq 5n^3$. Podriamos tambien decir $T(n)$ es $O(n^4)$ pero esto es más debil que decir es $O(n^3)$.*

Ejemplo 3.4 *Probemos que la función 3^n no es $O(2^n)$. Supongamos existen constantes n_0 y c tal que para todo $n \geq n_0$ se cumple $3^n leq c2^n$, entonces $c \geq (3/2)^n$ para todo $n \geq n_0$. Pero $(3/2)^n$ crece cuando n crece, luego no hay ninguna constante c que sea mayor que $(3/2)^n$ para todo n mayor o igual que un n_0.*

Cuando decimos $T(n)$ es $O(f(n))$ sabemos que $f(n)$ es una cota superior de la rapidez de crecimiento de $T(n)$. Para especificar una cota inferior utilizamos la notación $T(n)$ es $\Omega(g(n))$ que significa:

Definición 3.5 (Ω) *$T(n) = \Omega(f(n))$ si existen constantes positivas c y n_0 tal que $T(n) \geq c.f(n)$ para todo $n \geq n_0$.*

Ejemplo 3.6 *Verifiquemos que la función $T(n) = n^3 + 2n^2$ es $\Omega(n^3)$. Consideremos $c = 1$, luego $T(n) \geq cn^3$ para $n = 0, 1, \ldots$.*

Ejemplo 3.7 *Sea $T(n) = n$ para n impar y $T(n) = n^2/100$ para $n \geq 0$ y par. Para verificar $T(n)$ es $\Omega(n)$ consideramos $c = 1/100$ y $n = 0$.*

4. Comparando programas

Comparemos programas considerando sus funciones de rapidez sin considerar las constantes de proporcionalidad. Bajo esta suposición un programa con tiempo $O(n^2)$ es mejor que un programa con tiempo $O(n^3)$.

Consideremos las constantes de proporcionalidad. Supongamos el primer programa tiene constante 100 y el segundo 5. Para entradas de tamaño $n < 20$ el segundo programa será más rápido que el primero. Luego, si el programa se ejecutará en entradas de pequeño tamaño podemos preferir el programa $O(n^3)$.

5. Calculando el tiempo de ejecución de un programa

Regla de la suma

Supongamos que $T_1(n)$ y $T_2(n)$ son los tiempos de ejecución de dos fragmentos de programa P_1 y P_2. Supongamos T_1 es $O(f(n))$ y T_2 es $O(g(n))$. Luego el tiempo de ejecución de P_1 seguido por P_2 es $O(max(f(n), g(n)))$. veamos porque:

Por hipótesis, hay constantes c_1, c_2, n_1 y n_2 tales que $T_1(n) \leq c_1 f(n)$ para $n \geq n_1$ y $T_2(n) \leq c_2 g(n)$ para $n \geq n_2$. Consideremos $n_0 = max(n_1, n_2)$. Luego $T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n)$ para $n \geq n_0$ luego $T_1(n) + T_2(n) \leq (c_1 + c_2) max(f(n), g(n))$ para $n \geq n_0$ luego$T_1(n) + T_2(n)$ es $O(max(f(n), g(n)))$.

En general, el tiempo de ejecución de una secuencia fija de instrucciones es el tiempo de ejecución de la instrucción con mayor tiempo de ejecución.

Listemos reglas generales para el analisis de programas. En general el tiempo de ejecución de una sentencia o grupo de sentencias se parametrizará en el tamaño de la entrada u otras variables. Las reglas son las siguientes:

1. El tiempo de ejecución de cada asignación, entrada, salida se tomará como $O(1)$.

2. El tiempo de ejecución de una secuencia de instrucciones se determina por la regla de la suma.

3. El tiempo de ejecución de una instrucción if (sin else) es el costo de evaluar la condición más el costo de la instrucción. El tiempo de una instrucción if (con else) es el tiempo de evaluar la condición más el mayor de los tiempos de las instrucciones en el then y el else.

4. El tiempo para ejecutar un loop es la suma considerando todas las veces que se ejecuta el loop del tiempo de ejecución del cuerpo más el tiempo de evaluación de la condición. Usualmente este tiempo es a menos de un factor constante el producto del número de veces que se ejecuta el loop por el tiempo más largo correspondiente a ejecución del cuerpo.

Ejemplo 5.1 *Consideremos el programa de ordenamiento* **burbuja** *que ordena un array de enteros en orden creciente:*

```
void burbuja(int a[],int n)
{
        int i,j,temp;

        for(i=0;i<n-1;i++)
            for(j=0;j<n-1;j++)
                if (a[j]>a[j+1])
                   intercambio(&a[j],&a[j+1]);
}


void intercambio(int *a, int *b)
{
            int aux;

            aux=*a;
            *a = *b;
            *b=aux;
}
```

El tamaño de la entrada es n, el tamaño del array. Por la regla de la suma el tiempo de intercambio es $O(1)$, luego el tiempo del if y su cuerpo es $O(1)$.

El for interior es la suma de n veces 1 luego este es $O(n)$. El for exterior es la suma n veces de n luego el programa es $O(n^2)$.

5.1. Llamadas de Procedimientos

Si tenemos un programa que llama funciones, ninguna de las cuales es recursiva, podemos calcular el tiempo de ejecución de las funciones de a una por vez, comenzando por aquellas funciones que no invocan otras. Continuamos evaluando el tiempo de ejecución de las funciones para las cuales el tiempo de ejecución de las funciones que estas llaman fue deteminado anteriormente.

Si hay funciones recursivas, plantearemos una recurrencia que satisface el tiempo de ejecución de las mismas.

Ejemplo 5.2 *Consideremos la función recursiva factorial:*

```
int fac (int n)
{
        if (n <= 1) return 1
        else return n*fac(n-1)
}
```

planteamos la recurrencia que satisface su tiempo:

$T(n) = c_0 \qquad$ *si* $n \leq 1$
$T(n) = c_1 + T(n-1) \quad$ *si* $n > 1$

resolvemos la recurrencia del siguiente modo:

$T(0) = c_0$
$T(1) = c_0$
$T(2) = c_1 + T(1)$
$T(3) = c_1 + T(2)$
$\vdots$
$T(n-1) = c_1 + T(n-2)$
$T(n) = c_1 + T(n-1)$
$- -$ *sumemos ambos lados*
$T(n) + T(0) = c_1(n-1) + 2c_0$
luego
$T(n) = c_1(n-1) + c_0$

concluimos $T(n)$ *es* $O(n)$.