2.2. Problemas Tratables e Intratables

Tratabilidad

Los problemas denominados *intratables* son aquellos que, con entradas grandes, no pueden ser resueltos por ningún computador, no importa lo rápido que sea, cuanta memoria tenga o cuanto tiempo se le de para que lo resuelva. Lo anterior sucede debido a que los algoritmos que existen para solucionar estos problemas tienen una *complejidad* muy grande.

Complejidad

La *complejidad* de un algoritmo mide el grado u orden de crecimiento que tiene el tiempo de ejecución del algoritmo dado el tamaño de la entrada que tenga.

Existen dos maneras rápidas de hallar la complejidad de un algoritmo (métodos más profundos, formales y detallados pueden verse en [Cormen et al., 1990]):

1. por conteo, ó

2. por inspección o tanteo.

Para encontrar la complejidad de un algoritmo por conteo se debe tomar cada línea de código y determinar cuántas veces se ejecuta. Luego se suman las cantidades encontradas y la complejidad será del orden del resultado dado. Esta complejidad es una aproximación de cuánto se demoraría todo el algoritmo en ejecutarse.

Ejemplo 2.2.1
Un primer ejemplo sencillo es el algoritmo 1 para imprimir los 100 primeros números naturales.

```
void imprime100()
{
   int i = 1;
   while(i <= 100)
   {
      printf("%d",i);
      i++;
   }
}
```

Algoritmo 1: Imprime los números del 1 al 100

El conteo de líneas se puede realizar utilizando una tabla donde se numeren las líneas de código y se determine el número de veces que se ejecuta cada una:

Número de línea	Línea de código	Número de ejecuciones
1	`void imprime100()`	
2	`{`	
3	`int i = 1;`	1
4	`while(i <= 100)`	101
5	`{`	
6	`printf("%d",i);`	100
7	`i++;`	100
8	`}`	
9	`}`	

La línea 3 se ejecuta una sola vez. La guarda del `while` (línea 4) se ejecuta 101 veces debido a que verifica las 100 veces que se imprime el número más una vez adicional donde se determina que el ciclo ha terminado. Las líneas internas del ciclo se ejecutan 100 veces.
La suma de las cantidades encontradas es[3]:

$$\text{Numero Total de Ejecuciones} = 1 + 101 + 100 + 100 = 302$$

De allí que la complejidad del algoritmo es del orden de $\mathcal{O}(302)$. Por ser 302 una constante, la complejidad se puede aproximar a $\mathcal{O}(1)$, esto es, afirmar que es una complejidad constante.

⋆ ⋆ ⋆

Ejemplo 2.2.2
El ejemplo anterior se puede generalizar modificando la función de manera que tenga como parámetro la cantidad de números naturales a imprimir, es decir, hasta qué número natural se quiere escribir. Este nueva función se puede ver en el algoritmo 2.

```
void imprimeN(int n)
{
   int i = 1;
   while(i <= n)
   {
      printf("%d",i);
      i++;
   }
}
```

Algoritmo 2: Imprime los números del 1 al n

[3]La línea 1 no se tiene en cuenta ya que corresponde a los datos de referencia de la función (tipo de retorno, nombre y parámetros formales). Las líneas 2, 5, 8 y 9 tampoco se tienen en cuenta ya que son simplmente delimitadores de bloque.

Se numeran las líneas y se procede a contabilizar.

Número de línea	Línea de código	Número de ejecuciones
1	`void imprimeN(int n)`	
2	`{`	
3	`int i = 1;`	1
4	`while(i <= n)`	$n+1$
5	`{`	
6	`printf("%d",i);`	n
7	`i++;`	n
8	`}`	
9	`}`	

Al igual que el ejemplo 2.2.1, la línea 3 se ejecuta una sola vez, la guarda del `while` (línea 4) se ejecuta $n+1$ veces y las líneas internas del ciclo (líneas 6 y 7) se ejecutan n veces. Ahora la suma de las cantidades encontradas es:

$$\text{Numero Total de Ejecuciones} = 1 + (n+1) + n + n = 3n + 2$$

En el caso en que n fuera un número extremadamente grande, se puede ver que

$$3n + 2 \approx n$$

De esta manera la complejidad del algoritmo anterior es del orden de n, es decir, es $\mathcal{O}(n)$.

⋆ ⋆ ⋆

Ejemplo 2.2.3
Otro ejemplo cuyo código es muy simple pero su análisis es de cuidado es el algoritmo 3.

```
void imprime_mitad(int n)
{
   int i = n;
   while(i >= 0)
   {
      printf("%d",i);
      i = i / 2;
   }
}
```

Algoritmo 3: Imprime los números del n al 1 dividiendo por dos cada vez

En primera instancia se podría decir que el programa debería imprimir los números desde `n` hasta 0, pero dentro del ciclo el contador `i` se divide entre 2. Entonces en realidad se imprimirán los números $n, n/2, n/4, n/8, \ldots$. De allí que el número de veces que se ejecutan las instrucciones dentro del ciclo van disminuyendo exponencialmente:

- En la iteración 1 se disminuye en 2.
- En la iteración 2 se disminuye en 4, es decir, 2^2.
- En la iteración 3 se disminuye en 8, es decir, 2^3.
- En la iteración 4 se disminuye en 16, es decir, 2^4.

  $\vdots$

- En la iteración k se disminuye en 2^k.

Como se necesita saber cuántas veces se repite el ciclo y el contador cambia su valor desde 0 a `n`, entonces se requiere llegar al punto donde $n = 2^k$, siendo k el número que indica en qué iteración está la ejecución del algoritmo.

Para hallar el valor de k, se utilizan las propiedades de los logaritmos:

$$\begin{aligned} n &= 2^k \\ \log_2 n &= \log_2 2^k \\ \log_2 n &= k \end{aligned}$$

Luego el número de iteraciones que se realizan es $\log_2 n$.

Número de línea	Línea de código	Número de ejecuciones
1	`void imprime_mitad(int n):`	
2	`{`	
3	`int i = n;`	1
4	`while(i >= 0)`	$\log_2 n + 1$
5	`{`	
6	`cout << i;`	$\log_2 n$
7	`i = i / 2;`	$\log_2 n$
8	`}`	
9	`}`	

La suma de las cantidades encontradas es:

$$\begin{aligned} \text{Numero Total de Ejecuciones} &= 1 + (\log_2 n + 1) + \log_2 n + \log_2 n \\ &= 2 + 3\log_2 n \\ &\approx \log_2 n \end{aligned}$$

Por lo tanto, la complejidad del algoritmo es $\mathcal{O}(\log_2 n)$, que normalmente se expresa como $\mathcal{O}(\log\ n)$.

⋆ ⋆ ⋆

Ejemplo 2.2.4
Cuando analizamos algoritmos con condicionales hay que tener en cuenta que el conteo se hace considerando si las guardas de los condicionales se cumplen o no. La complejidad en estos algoritmos se halla en *el peor de los casos* (cuando se asume que las guardas de los condicionales siempre se cumplen), *el caso promedio* (cuando se asume que las guardas algunas veces se cumplen y otras veces no) y *el mejor de los casos* (cuando se asume que las guardas no se cumplen).

El algoritmo 4 suma los elementos impares de un vector de enteros.

```
int sumaVector(int *v, int n)
{
   int i = 0;
   int sum = 0;
   while(i < n)
   {
      if(v[i] % 2 != 0)
         sum = sum + v[i];
      i++;
   }
   return sum;
}
```

Algoritmo 4: Suma los elementos impares de un vector de enteros

Se numeran las líneas y se procede a contabilizar.

Número de línea	Línea de código	Número de ejecuciones
1	`int sumaVector(int *v, int n)`	
2	`{`	
3	`int i = 0;`	1
4	`int sum = 0;`	1
5	`while(i < n)`	$n+1$
6	`{`	
7	`if(v[i] % 2 != 0)`	n
8	`sum = sum + v[i];`	?
9	`i++;`	n
10	`}`	
11	`return sum;`	
12	`}`	

La cantidad de veces que se ejecuta la línea 8 es indefinida debido a que depende de si la guarda del condicional es verdadera o falsa. Por esta razón tenemos que analizar esta situación desde los tres casos:

- En el mejor de los casos ningún elemento del vector es impar por lo que la línea 8 no se ejecutaría nunca.
- En el caso promedio, aproximadamente la mitad de los elementos será impar y la otra mitad par. En este caso la línea se ejecutaría $n/2$ veces.
- En el peor de los casos todos los elementos del vector son impares por lo que siempre que se ejecute la línea 7 se ejecutará la línea 8. Luego esta línea se ejecutará n veces.

La suma de las cantidades encontradas es entonces:

- En el mejor de los casos:

$$\begin{aligned} \text{Numero Total de Ejecuciones} &= 1+1+(n+1)+n+\mathbf{0}+n \\ &= 3+3n \end{aligned}$$

- En el caso promedio:

$$\begin{aligned} \text{Numero Total de Ejecuciones} &= 1+1+(n+1)+n+\mathbf{n/2}+n \\ &= 3+7(n/2) \end{aligned}$$

- En el peor de los casos:

$$\begin{aligned} \text{Numero Total de Ejecuciones} &= 1+1+(n+1)+n+\mathbf{n}+n \\ &= 3+4n \end{aligned}$$

Por lo tanto, la complejidad del algoritmo es, en este algoritmo particular, $\mathcal{O}(n)$.

⋆ ⋆ ⋆

Ejemplo 2.2.5
Otro ejemplo, un poco más complejo, es un algoritmo de ordenamiento de un vector de enteros:

```
void burbuja(int *v, int n)
{
   int i = 0;
   while(i < n)
   {
      int j = i + 1;
      while(j < n)
      {
         if(v[i] > v[j])
         {
            int temp = v[i];
            v[i] = v[j];
            v[j] = temp;
         }
         j++;
      }
      i++;
   }
}
```

Algoritmo 5: Ordenamiento por el método burbuja

Se numeran las líneas y se procede a contabilizar.

Número de línea	Línea de código	Número de ejecuciones
1	`void burbuja(int *v, int n)`	
2	`{`	
3	`int i = 0;`	1
4	`while(i < n)`	$n+1$
5	`{`	
6	`int j = i+1;`	n
7	`while(j < n)`	$\sum_{k=1}^{n} k$
8	`{`	
9	`if(v[i] > v[j])`	$(\sum_{k=1}^{n} k) - n$
10	`{`	
11	`int temp = v[i];`	$(\sum_{k=1}^{n} k) - n$
12	`v[i] = v[j];`	$(\sum_{k=1}^{n} k) - n$
13	`v[j] = temp;`	$(\sum_{k=1}^{n} k) - n$
14	`}`	
15	`j++;`	$(\sum_{k=1}^{n} k) - n$
16	`}`	
17	`i++;`	n
18	`}`	
19	`}`	

La línea 3 se ejecutará una sola vez. Solamente asigna el valor 0 a la variable `i`.

Si el tamaño del vector es n, entonces la guarda del ciclo externo (línea 4) va ejecutarse $n+1$ veces, ya que la variable `i` comienza el ciclo con el valor 0 y se incrementa en 1 cada iteración hasta que llegue al valor n, cuando se termina el ciclo. Sin embargo las líneas 6 y 17 se ejecutan n veces, es decir, uno menos que la línea 4. Lo anterior debido a que se debe verificar la guarda del ciclo una vez adicional para saber que ya no debe entrar más al ciclo.

La guarda del ciclo interno (línea 7) se ejecutará un número de veces que depende de la variable i, que en el último ciclo tendrá el valor del tamaño del vector menos uno. Más en detalle, en la primera iteración del ciclo externo la variable `i` es igual a 0, por lo que la guarda del ciclo interno se ejecuta n veces; en la segunda iteración del ciclo externo la variable `i` es igual a 1, por lo que la línea 7 se ejecuta $n-1$ veces; así sucesívamente hasta la última iteración del ciclo externo, donde la variable `i` es igual a $n-1$ y la guarda del ciclo interno se ejecuta 1 vez. Todo esto da como resultado una sumatoria del número de ejecuciones de la línea 7, desde 1 hasta n.

Si consideramos el peor de los casos, al igual que pasó con la línea 6, la guarda del `if` y las asignaciones internas (líneas 11–13) se ejecutan una vez menos que la guarda el ciclo interno, es decir, la sumatoria del número de ejecuciones desde 1 hasta n menos n (en cada iteración dichas líneas se ejecutan 1 vez menos y en total son n iteraciones).

Por teoría matemática, se tiene que

$$\sum_{i=1}^{n} i = \frac{n \times (n+1)}{2}$$

De allí que es posible expresar el número de ejecuciones sólo en términos de n:

$$\begin{aligned} \text{Numero Total de Ejecuciones} &= (1) + (n+1) + (n) + \left(\tfrac{n\times(n+1)}{2}\right) + \ldots \\ &\quad \ldots + 5\left(\tfrac{n\times(n+1)}{2} - n\right) + (n) \\ &= 2 + n + 3n^2 \\ &\approx n^2 \end{aligned}$$

Y así se puede decir que la complejidad del algoritmo `burbuja` es del orden de n^2, es decir, es $\mathcal{O}(n^2)$.

⋆ ⋆ ⋆

Por otro lado, hallar la complejidad por inspección o tanteo, es más rápida pero imprecisa y, si no se cuenta con la suficiente experiencia, poco confiable.

Simplemente se mira la estructura del algoritmo y se siguen las tres siguientes reglas:

1. La complejidad de una asignación es $\mathcal{O}(1)$.
2. La complejidad de un condicional es 1 más el máximo entre la complejidad del cuerpo del condicional cuando la guarda es positiva y el cuerpo del condicional cuando la guarda es negativa.
3. La complejidad de un ciclo es el número de veces que se ejecuta el ciclo multiplicado por la complejidad del cuerpo del ciclo.

En el algoritmo 5 (`burbuja`) se puede observar que el ciclo externo tiene n iteraciones (siendo n el tamaño del vector), el ciclo interno, en la primera iteración del ciclo externo, tiene n iteraciones, y el condicional tiene como cuerpo tres asignaciones. Por todo esto, la complejidad del algoritmo sería

$$\mathcal{O}(1 + n \times (2 + n \times ((1+3)+1))) = \mathcal{O}(1 + 2n + 5n^2) \approx \mathcal{O}(n^2).$$

Ahora, si las asignaciones internas y la condición del `if` (las líneas 5-8 juntas) tomaran un segundo en ejecutarse, en el peor de los casos (cuando siempre se ejecute lo que está dentro del `if`) se tendría:

Tamaño del vector	**Tiempo de ejecución (seg.)**
10	100
20	400
50	2500
100	10000
1000	1000000

El cuadro anterior muestra que un algoritmo con complejidad $\mathcal{O}(n^2)$ puede ser rápido para tamaños de entrada pequeños, pero a medida que crece la entrada, se va volviendo ineficiente.

En el cuadro 2.2 se muestran algunos ejemplos de complejidades que pueden encontrarse en los análisis de algoritmos, para un problema de tamaño n, desde la más rápida hasta la más ineficiente. Se dice que un problema es *tratable* si su complejidad es polinomial o menor.

Complejidad	**Nombre**
$\mathcal{O}(1)$	Constante
$\mathcal{O}(\log\ n)$	Logarítmica
$\mathcal{O}(n^c), 0 > c > 1$	Fraccional
$\mathcal{O}(n)$	Lineal
$\mathcal{O}(n\ \log\ n)$	Lineal-logarítmica
$\mathcal{O}(n^2)$	Cuadrática
$\mathcal{O}(n^3)$	Cúbica
$\mathcal{O}(n^c), c > 3$	Polinomial
$\mathcal{O}(2^n)$	Potencia de dos
$\mathcal{O}(3^n)$	Potencia de tres
$\mathcal{O}(c^n), c > 3$	Exponencial
$\mathcal{O}(n!)$	Factorial
$\mathcal{O}(n^n)$	Potencia de n

Cuadro 2.2: Complejidades

A partir de una complejidad $\mathcal{O}(2^n)$, los problemas para los cuales hay un algoritmo con dicha complejidad son intratables.

NP-Completitud

Existen otros problemas llamados "NP-Completos", cuya complejidad es desconocida. Se dice que son "NP" ya que se presume que los algoritmos que los solucionen son *No-Polinomiales*; sin embargo, no existe ninguna prueba que demuestre lo contrario, es decir, que sean "P" (Polinomiales)[4]. Los científicos creen que los problemas NP-Completos son intratables, debido a que si existiera un algoritmo que resolviera alguno en un tiempo polinomial, entonces *todos* los problemas NP-Completos podrían ser resueltos en un tiempo polinomial. Lo anterior llevó a los investigadores a plantear una de las más importantes preguntas en las ciencias de la computación [Cook, 1971]:

$$\text{¿P} = \text{NP?}$$

Es decir, ¿las clases de complejidades Polinomiales y las No-Polinomiales son equivalentes? El instituto de Matemáticas Clay, en Cambridge, está ofreciendo 1 millón de dólares

[4] Aunque Vinay Deolalikar de los laboratorios de Hewlett Packard en el segundo semestre del 2010 compartió una versión preliminar de una prueba [Deolalikar, 2010].