UNIDAD 7

Análisis de Algoritmos

7.1 Concepto de complejidad de Algoritmos

- La complejidad de un algoritmo es aquella función que da el tiempo y/o el espacio utilizado por el algoritmo en función del tamaño de la entrada.
- Uno de los principales objetivos es el de desarrollar algoritmos que manipulen eficientemente los datos.
- El tiempo y el espacio utilizados para ello miden la mayor o menor eficacia de un algoritmo.

7.1.1 Complejidad de Algoritmos y estructura de datos

- Cada algoritmo guarda una estrecha relación con una estructura de datos.
- Se debe buscar un equilibrio entre tiempo y espacio.
- La elección de la correcta estructura de datos depende de varios factores como el tipo de datos a manejar y la frecuencia con que se realizan diferentes operaciones sobre ellos.
- En general, si aumentamos el espacio necesario para almacenar datos, conseguiremos un mejor rendimiento en el tiempo y viceversa.

7.1.2 Eficiencia de un algoritmo

- El análisis de algoritmos estudia, desde el punto de vista teórico, los recursos computacionales que necesita la ejecución de un programa: su eficiencia. Por ejemplo:

  Tiempo de CPU
  Uso de memoria
  Ancho de banda …

- En el desarrollo real de software, existen otros factores que, a menudo, son más importantes que la eficiencia: funcionalidad, corrección, robustez, usabilidad, modularidad, mantenibilidad, fiabilidad, simplicidad y el tiempo del programador.

- La eficiencia de un algoritmo nos sirve para establecer la frontera entre lo factible y lo imposible.

7.2 Complejidad

Supongamos que **M** es un algoritmo y que **n** es el tamaño de los datos de entrada:

- El tiempo de ejecución y el espacio utilizados por **M** son los parámetros principales que miden la eficiencia de **M**.
- El tiempo suele medirse contando el número de operaciones claves realizadas (ejem. # comparaciones).
- El espacio es evaluado contando el máximo de memoria necesaria por el algoritmo.

La complejidad de un algoritmo **M** es una función **f(n)** que da el tiempo y/o el espacio de almacenamiento necesario en la ejecución del algoritmo, en función del tamaño de los datos de entrada **n**.

7.2.1 Tiempo de ejecución de un algoritmo

- La complejidad en cuanto al tiempo de un programa es la cantidad de tiempo que se necesita para ejecutarlo.
- La evaluación del tiempo de computación resulta más importante que la del espacio de almacenamiento.
- El tiempo no es posible expresarlo en segundos, pues entonces dependería de la máquina en la que estuviera siendo ejecutado o de otros factores como el compilador utilizado.
- E necesario ignorar las constantes dependientes del contexto
- Se considera el “principio de la invarianza”, el cual establece que al cambiar la máquina donde se ejecuta un algoritmo o el lenguaje en el que se implementan los tiempos de ejecución no difieren más que en una constante multiplicativa.

7.2.1 Tiempo de ejecución de un algoritmo (cont.)

- El tiempo de ejecución depende del tamaño del conjunto de datos.
- Se debe parametrizar el tiempo de ejecución en función del tamaño del conjunto de datos, intentando buscar una cota superior que nos sirva de garantía.
- Hay que considerar que en algunos casos el tiempo de ejecución depende también de una entrada específica.
- Si T(n) es el tiempo de ejecución de un programa con entrada de tamaño n, será posible valorar T(n) como el número de sentencias o instrucciones ejecutadas por el programa.

7.2.2 Tipos de análisis

- Peor caso:

T(n) = Tiempo máximo necesario para un problema de tamaño n. Adecuado para algoritmos cuyo tiempo de respuesta sea crítico.

- Caso medio (a veces)

T(n) = Tiempo esperado para un problema cualquiera de tamaño n. Requiere establecer una distribución estadística. Adecuado para algoritmos cuya respuesta no es critica y de uso frecuente.

- Mejor caso (engañoso)

T(n) = Indica el tiempo para el mejor caso.

7.2.3 Complejidad en espacio

- La complejidad en cuanto al espacio de un programa es la cantidad de memoria que se necesita para ejecutarlo.
- La memoria estática se calcula sumando la que ocupan las variables simples, los campos de los registros y los componentes de los vectores.
- La memoria dinámica depende de la cantidad de datos y del funcionamiento del programa y se calcula con técnicas similares a las que se utilizan para la evaluación del tiempo.

7.3 Aritmética de la notación O-grande

El tiempo de ejecución de un programa se expresa normalmente utilizando la notación "O-grande" que esta diseñada para expresar factores constantes, tales como:

- El número medio de instrucciones máquina que genera un compilador determinado.
- El número medio de instrucciones máquina por segundo que ejecuta una computadora específica.

Así, se dirá que un algoritmo determinado emplea un tiempo O(n) que se lee "O grande de n" o bien "O de n" y que informalmente significa: "algunos tiempos constantes n".

7.3.1 Supuestos notación O-grande

- La noción de “algunos tiempos constantes n” permite ignorar la constante desconocida asociada con el compilador y la máquina.
- Sea *f(n)=T(n)* el tiempo de ejecución de algún programa o algoritmo, medido como una función de la entrada de tamaño *n*.
- Sea *f(n)* una función definida sobre enteros no negativos.
- Se dice *“f(n) es O(g(n))”* , si *f(n)* es como máximo una constante de tiempo *g(n)*, excepto posiblemente para algunos valores pequeños de *n*.

7.3.1 Supuestos notación O-grande(cont)

- Se dice *f(n)* es *O(g(n))* si existe un entero n_0 y una constante c>0 tal que para todos los enteros $n>=n_0$ se tendrá que *f(n)<=c*g(n)* (cota superior).
- La notación *f(n)=O(g(n))* significa que |*f(n)*|<=*c*|*g(n)*| para $n>=n_0$.
- Por consiguiente, |*g(n)*| es un límite o cota superior para |*f(n)*|.
- Formalmente: *f(n)=O(g(n))* si existen constantes *c>=0* y n_0 tales que para $n>=n_0$, es decir si:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c \geq 0$$

7.3.2 Propiedades de la notación O

1. $c*O(f(n))=O(f(n))$
2. $O(f(n))+O(f(n))=O(f(n))$
3. $O(f(n))+O(g(n))=O(f(n)+g(n))$
4. $O(O(f(n)))=O(f(n))$
5. $Max(O(f(n)),O(g(n)))=O(Max(f(n),g(n)))$
6. $O(\log_a(n))=O(\log_b(n))$ para $a,b>1$
7. $O(\log(n!))=O(n*\log(n))$

7.3.2 Propiedades de la notación O

8. Para k>1 $O\left(\sum_{i=1}^{n} i^{k}\right) = O\left(n^{k+1}\right)$

9. $O\left(\sum_{i=1}^{n} i^{-1}\right) = O\left(\sum_{i=1}^{n} \frac{1}{i}\right) = O(\log(n))$

10. Si a es diferente de b entonces: $\left(O\left(\sum_{i=1}^{n} i^{K}\right) = O\left(n^{k+1}\right)\right)$

11. En general para todo c>0 y a diferente de b:

$$O\left(c^{\log_a(n)}\right) \neq O\left(c^{\log_b(n)}\right)$$

12. $O\left(\sum_{i=2}^{n} \log(i)\right) = O(n\log(n))$

7.3.3 Complejidad de las distintas sentencias para algoritmos iterativos.

- Las asignaciones, lecturas, escrituras y comparaciones son todas de orden 1, excepto que sean tipos estructurados de datos.

$$T(n) = O(1)$$

- La complejidad de una secuencia es la suma de las complejidades de cada una de las sentencias que la forman.

$$T(n) = T_1(n) + T_2(n) + \ldots = \max \{ O(T_1(n)), O(T_2(n)), \ldots \}$$

- La complejidad de una selección es igual a 1 más el máximo de la complejidad de cada una de las partes que forman la selección.

$$T(n) = O\left(T_{condición}(n)\right) + \max \{ O(T_{then}(n)), O(T_{else}(n)) \}$$

7.3.3 Complejidad de las distintas sentencias para algoritmos iterativos.

- Para a complejidad de un bucle se considera:
  a) Complejidad fija para cada iteración=Producto del número de iteraciones por la complejidad de cada iteración.
  b) Si la complejidad varía en función de la iteración, obtenemos una sumatoria de las complejidades para cada iteración de las instrucciones de dentro del bucle.
  c) Si no conocemos con exactitud el número de iteraciones (bucles while y do-while), se estima este número para el peor caso.
  d) Esto supondrá sumar series, aritmeticas, geométricas, etc.
- La llamada a un procedimiento o una función será de orden 1 siempre que no exista recursividad, y que los parámetros sean todos simples. En caso de que los parámetros sean estructurados, y se transmitan por valor, habrá que considerar el tiempo de transmisión de los parámetros.

7.3.4 Complejidad para algoritmos recursivos.

Para estudiar la complejidad de los algoritmos recursivos se emplean sobre todo los tres métodos siguientes:

1. La inducción matemática.
2. Expansión de recurrencias. Consiste en sustituir la recurrencia por su igualdad hasta llegar a un caso conocido (más utilizado por su sencillez).
3. Usar soluciones generales ya conocidas y compararlas con las que se quieren obtener. (Ecuaciones en diferencias finitas.)

7.3.5 Funciones de complejidad más comunes

- O(1) Complejidad Constante. La más deseada. Aparece en algoritmos sin bucles.
- O(log(n)) Complejidad logarítmica. Es una complejidad óptima. Aparece en la búsqueda binaria. Este tiempo de ejecución es normal en programas que resuelven un problema de gran tamaño transformándolo en uno más pequeño, dividiéndolo mediante una fracción constante.
- O(n) Complejidad lineal. Es muy buena y muy usual. Aparece en los bucles simples.
- O(n*log(n)) Aparece en algoritmos recursivos con un bucle simple y dos llamadas recursivas de tamaño mitad.
- $O(n^2)$ Complejidad cuadrática. Aparece en los bucles anidados dobles.

7.3.5 Funciones de complejidad más comunes (cont.)

- $O(n^3)$ Complejidad cúbica. Aparece en los bucles anidados triples. Para n grande crece excesivamente.
- $O(n^k)$ Complejidad polinómica. Para k>=3 crece demasiado rápidamente.
- $O(2^n)$ Complejidad exponencial. Aparece en algoritmos recursivos, cuyo tamaño del ejemplar disminuye en sólo una unidad en cada llamada, y que tienen dos llamadas recursivas (Torres de Hanoi).
- $O(k^n)$ para k>2. Aparece en algoritmos recursivos, cuyo tamaño del ejemplar disminuye en sólo una unidad en cada llamada, y que tienen k llamadas recursivas (caso del problema del caballo k=8).

7.3.6 Eficiencia asintótica

Cuando n es lo suficientemente grande…

un algoritmo $O(1)$ es más eficiente que

un algoritmo $O(\log n)$ es más eficiente que

un algoritmo $O(n)$ es más eficiente que

un algoritmo $O(n \log n)$ es más eficiente que

un algoritmo $O(n^2)$ es más eficiente que

un algoritmo $O(n^3)$ es más eficiente que

un algoritmo $O(2^n)$

7.3.7 Inconvenientes de la notación O-grande

- Simplemente proporciona un límite superior para la función.
- Siempre que sea posible, se elige el elemento más pequeño de la siguiente jerarquía de órdenes:

$$O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3), \ldots O(2^n), O(n^n)$$

- Sólo aproxima el comportamiento de una función para argumentos arbitrarios grandes.
- En el análisis de algoritmos se considera usualmente el caso peor, si bien a veces conviene analizar igualmente el caso mejor y hacer alguna estimación sobre un caso promedio.
- Los órdenes de complejidad sólo son importantes para grandes problemas.

7.4 Selección de un algoritmo

Para evaluar un algoritmo debemos considerar:

1. Cumpla los objetivos y funciones con cualquier posible valor de los datos que maneja.
2. Se fácil de codificar y depurar.
3. No exista otro algoritmo que resuelva el problema utilizando menos recursos (tiempo en ejecutarse y memoria consumida).

Nota: Los puntos 2 y 3 pueden ser contradictorios.