Becerra Ramírez David Enrique

Análisis de Algoritmos

El análisis de algoritmo que hacemos toca únicamente el punto de vista temporal (tiempo de ejecución de un algoritmo) y utilizamos como herramienta el lenguaje de programación Java.
Es difícil realizar un análisis simple de un algoritmo que determine la cantidad exacta de tiempo requerida para ejecutarlo. La primera complicación es que la cantidad exacta de tiempo dependerá de la implementación del algoritmo y de la máquina real en que se ejecuta. El análisis normalmente debe ser independiente de la computadora (hardware y software) y del lenguaje o máquina que se utilice para implementar el algoritmo. La tarea de calcular el tiempo exacto requerido suele ser bastante pesado.
Un algoritmo es un conjunto de instrucciones ordenados de manera lógica que resuelven un problema. Estas instrucciones a su vez pueden ser: enunciados simples (sentencias) o enunciados compuestos (estructuras de control). El tiempo de ejecución dependerá de como esté organizado ese conjunto de instrucciones, pero nunca será constante.
Es conveniente utilizar una función T(n) para representar el número de unidades de tiempo (o tiempo de ejecución del algoritmo) tomadas por un algoritmo de cualquier entrada de tamaño n. La evaluación se podrá hacer desde diferentes puntos de vista:

Peor caso. Se puede hablar de T(n) como el tiempo para el peor caso. Se trata de aquellos ejemplares del problema en los que el algoritmo es menos eficiente (no siempre existe el caso peor). Ejemplos: insertar al final de una lista, ordenar un vector que está ordenado en orden inverso, etc. Nos interesa mucho.

Mejor caso. Se habla de T(n) como el tiempo para el mejor caso. Se trata de aquellos ejemplares del problema en los que el algoritmo es más eficiente; por ejemplo: insertar en una lista vacía, ordenar un vector que ya está ordenado, etc. Generalmente no nos interesa.

Caso medio. Se puede computar T(n) como el tiempo medio de ejecución del programa sobre todas las posibles ejecuciones de entradas de tamaño n. El tiempo de ejecución medio es aveces una medida más realista del rendimiento en la práctica, pero es, normalmente, mucho más difícil de calcular que el tiempo de ejecución en el pero caso, ya que requiere definir una distribución de probabilidades de todo el conjunto de datos de entrada, el cuál típicamente es una tarea difícil.

Función de complejidad

La función de complejidad de un algoritmo es el número de operaciones elementales que utiliza un algoritmo cualquiera para resolver un problema de tamaño n. Matemáticamente se define la Función de complejidad asi:
Sea A un algoritmo, la función de complejidad del algoritmo A T(n) se define como el número máximo de operaciones elementales que utiliza el algoritmo para resolver un problema de tamaño n.
T(n) = Max {nx: nx es el número de operaciones que utiliza A para resolver una instancia x de tamaño n}
Nota: Una operación elemental es cualquier operación cuyo tiempo de ejecución es acotado por una constante (que tenga tiempo constante). Por ejemplo: una operación lógica, una operación aritmética, una asignación, la invocación a un método

Cálculo del T(n)

Para hallar la función de complejidad (t(n)) de un algoritmo se puede evaluar el algoritmos desde tres puntos de vista:

Peor Caso: Se puede hablar de T(n) como el tiempo de ejecución para el peor de los casos,

en aquellos ejemplares del problema en el que el algoritmo es Menos Eficiente.
Caso Medio: Se puede comportar el T(n) como el tiempo medio de ejecución del programa sobre todas las posibles ejecuciones de entrada de tamaño n. Es una medida más realista del rendimiento del algoritmo en la práctica, pero es mucho más difícil del cálculo, ya que requiere una distribucion de probabilidades de todo el conjunto de entrada lo cual es una tarea difícil.
Mejor Caso: Se puede hablar de T(n) como el tiempo de ejecución para el mejor de los casos, en aquellos ejemplares del problema en el que el algoritmo es Más Eficiente.
Lo ideal sería poder evaluar el algoritmo en el caso promedio, pero por el nivel de operaciones y dificultad que conlleva este, el estudio de la complejidad de los algoritmos se evalúa en el peor de los casos.
Para calcular el T(n), se deben calcular el número de operaciones elementales.

Orden de Magnitud (Notación O Grande)
Cuando se trata de algoritmos, el cálculo detallado del tiempo de ejecución de todas las operaciones primitivas llevaría mucho tiempo. Además, ¿qué importancia tendría el número de instrucciones primitivas ejecutadas por un algoritmo? Es más útil en el análisis de algoritmos, ver la velocidad de crecimiento del tiempo de ejecución como una función del tamaño de la entrada n, en lugar de realizar cálculos detallados. Es más significativo saber que un algoritmo crece, por ejemplo, proporcionalmente a n, a razón de un factor constante pequeño que depende del hardware o software y puede variar en un cierto rango, dependiendo de una entrada n específica. Esto lo que se conoce como orden de magnitud “O(g(n))” o notación asintótica o notación “O grande”. El orden de magnitud se utiliza para comparar la eficiencia de los algoritmos.
La notación O grande es una técnica utilizada para el análisis de la complejidad computacional de un algoritmo, este análisis se centra en el término dominante (El término que más aumenta), permitiendo así ignorar constantes y términos de orden menor.
Por ejemplo:

T(n) = 1	**----** > O (1) ----> Orden Constante
T(n) = $log_2 n$	**----** > O $(log\ n)$ ----> Orden Logaritmicc
T(n) = $an + b$	**----** > O (n) ----> Orden Lineal
T(n) = $nlog_2 n$	**----** > O (n) ----> Orden $nlog_2 n$
T(n) = $an^2 + bn + c$	**----** > O (n^2) ----> Orden Cuadratico
T(n) = $an^3 + bn^2 + c$	**----** > O (n^3) ----> Orden Cubico
T(n) = $n^m,\ m = 0, 1, 2, 3\ldots$	**----** > O (n^2) ----> Orden Polinomial
T(n) = $c^n,\ c > 1$	**----** > O (n^3) ----> Orden Exponencial
T(n) = $n!$	**----** > O (n^2) ----> Orden Factorial