**Órdenes**

**Tiempo de Ejecución de un Programa**

Al resolver un problema, debemos elegir el algoritmo a utilizar, siguiendo los siguientes objetivos:

Algoritmo fácil de entender, codificar y depurar,

Uso eficiente de los recursos y ejecución veloz.

**Medición del tiempo de ejecución de un programa**

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

* Los datos de entrada al programa,
* La calidad del código generado por el compilador,
* La naturaleza y rapidez de las instrucciones usadas en el programa,
* La complejidad de tiempo del algoritmo base del programa.

Muchas veces el tiempo depende del tamaño de la entrada más que de la entrada en sí misma.

Se denomina ***T(n)*** el tiempo de ejecución de un programa con entrada de tamaño ***n***.

Cuando el tiempo de ejecución es función de una entrada específica y no sólo del tamaño, entonces se define ***T(n)*** como el tiempo de ejecución del *peor caso*.

También se podría calcular el ***Tprom(n)***, o sea, el tiempo promedio de ejecución, pero suele ser bastante más difícil.

El hecho de que el tiempo de ejecución dependa del compilador y de la máquina impide que se pueda expresar en unidades de tiempo normales.

Se dirá entonces, que, por ejemplo, “el algoritmo es proporcional a ***n2***”.

**Notación asintótica – Orden del tiempo de ejecución**

Se dice que el tiempo de ejecución de un programa es de ***orden n***, (***T(n)*** es ***O(n)***), si existen dos constantes enteras positivas ***c*** y ***no*** tales que,

Para ***todo n >= no***, ***T(n) <= c \* n***

Se dice que ***T(n)*** es ***O(f(n))*** si existen dos constantes enteras positivas ***c***  y ***no*** tales que,

Para ***todo n >= no, T(n) <= c \* f(n)***

En este caso, se dice que el programa tiene *velocidad de crecimiento* ***f(n)***.

Si ***T(n)*** es ***O(f(n)), f(n)*** es una cota superior para la velocidad de crecimiento de ***T(n)***.

La cota inferior se especifica notando ***T(n)*** es Ω***(g(n))***. Esto significa que existe una constante **c** tal que ***T(n) >= c \* g(n)*** para un número infinito de valores de n.

Suponemos que podemos evaluar programas comparando sus funciones de tiempo de ejecución, sin considerar las constantes de proporcionalidad. De acuerdo a esto, un programa con tiempo de ejecución O(n2) es mejor que uno con tiempo de ejecución O(n3).

Además de los factores del compilador y la máquina, existe un factor debido al programa mismo.

Ej: 100 \* n2 milisegundos vs 5 \* n3 milisegundos.

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Tabla

Descripción generada automáticamente

Tabla

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

**Tiempo de ejecución**

Si un programa se va a ejecutar poco, el costo de su desarrollo es el dominante.

Si el programa se ejecuta sólo con entradas pequeñas, la velocidad de crecimiento puede ser menos importante que el factor constante.

Un algoritmo eficiente pero complicado aún puede no ser adecuado, por razones de mantenimiento.

Algunos algoritmos eficientes requieren demasiado espacio, debiéndose utilizar almacenamiento secundario, con lo cual pierden su eficiencia.

En los algoritmos numéricos, la precisión y la estabilidad son tan importantes como la eficiencia.

**Cálculo del tiempo de ejecución de un programa**

Reglas de la suma y producto en notación asintótica.

El tiempo de ejecución de cada proposición de asignación puede tomarse como O(1).

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

El tiempo de ejecución de una proposición condicional if es el costo de las proposiciones que se ejecutan condicionalmente, más el tiempo para evaluar la condición (se toma O(1)).

El tiempo para ejecutar un ciclo es la suma sobre todas las iteraciones del ciclo, del tiempo de ejecución del cuerpo y del empleado para evaluar la condición de terminación. Normalmente este tiempo es el producto del número de iteraciones del ciclo por el mayor tiempo posible para una ejecución del cuerpo.

**Analizar ciclos (tiene que ir por el peor caso):**

* **O(1) à** Toda instrucción que se considere tiempo constante (menos recursión y ciclos).
* **O(n) à** cuando el ciclo itere en base a una entrada, en caso de ser ciclos anidados, por ejemplo un for dentro de otro es O(n2)
* **O(log(n)) à** Cuando el ciclo en vez de incrementar por un num constante, se multiplica o se divide.
* **O(log(log(n))) à** cuando incrementa de forma exponencial.

**(Al sumar todo, solo se toma lo más significativo.)**

La cantidad de tiempo que tardan los algoritmos en ejecutarse dependen del tamaño de la entrada.

Ejemplo: se descarga un archivo que tarda 2segs en establecer conexión y la descarga va a 160 K/seg. Entonces el tiempo de descarga esta dado por la fórmula T(n) = n / 160 + 2.

Este sería un algoritmo ideal, directamente proporcional al tamaño de la entrada. Es el algoritmo más eficiente.

Regla de la suma: T1(n) + T2(n) es max (F1(n), F2(n)).

Regla del producto: T1(n) \* T2(n) es O(F1(n) \* F2(n)).

**Recursión à** Cuando un método se llama a sí mismo.

* Es la forma en la cual se especifica un proceso basado en su propia definición.

Definición de un problema en instancias más pequeñas de sí mismo.

Conociendo la solución explicita de las instancias más simples (casos base) se puede aplicar inducción para resolver el caso general.

Un objeto es recursivo si en parte está formado por si mismo, o está definido en función de si mismo.

Una función es recursiva cuando se invoca a sí misma.

Si P tiene referencia explicita a si mismo, es directamente recursivo.

Si P tiene referencia a M que tiene referencia a P, es indirectamente recursivo.

* Probar los casos base, debe haber por lo menos uno.
* Cada cadena de llamadas recursivas potencial debe de alcanzar un caso base y el manejo del caso base no debe usar recursividad.
* Realizar una sola llamada recursiva.
* Definir cada posible llamada recursiva de forma tal que progrese a un caso base.

Recursión de cola à ocurre cuando un método linealmente recursivo hace su llamada recursiva como último paso. Estos métodos pueden ser fácil convertidos en métodos no recursivos.

Recursión binaria à ocurre cada vez que hay 2 llamadas recursivas para cada caso no-base.

Rutina de preparación à ejecuta condiciones y luego la rutina recursiva.

Java utiliza una pila de registros de activación con todos los métodos recursivos llamados y los ejecuta en orden inverso.

**Jamás debe utilizarse recursión como sustituto de un bucle sencillo.**

Guardar en memoria las llamadas, con algoritmos costosos y que vuelven a llamar la misma función con mismos parámetros.)

Programación dinámica.

Optimización à encontrar una solución que maximiza o minimiza una función.

Requieren prueba de que devuelven la mejor solución.

Técnica para implementar eficientemente un algoritmo recursivo almacenando resultados parciales. Cuando el algoritmo recursivo computa los mismos subproblemas varias veces, almacenar los resultados para reusarlos.

Propiedades:

* Subproblemas superpuestos o que se solapan
* Subestructuras óptimas
* Reusar resultados (memorización)

Árboles de busca óptimos: árbol binario de búsqueda.

**Divide y vencerás à** (Algoritmos con al menos 2 llamadas recursivas.)

à Técnica que hace uso de la recursión.

* Algoritmos con 2 llamadas recursivos internas.

Plantear el problema de forma que sea posible descomponerlo en k-subproblemas.

Resolver los subproblemas, con sus respectivas llamadas recursivas, y combinarlos en una solución.

Algoritmos “ávidos” à dado c (entradas), el algoritmo ávido devuelve en cada iteración un conjunto S tal que S está incluido en C. Es como cambiar $50 por monedas.

S “prometedor”

Elementos de la técnica:

* C conjunto de candidatos (entradas).
* Función solución
* Función de selección.
* Función de factibilidad.
* Función objetivo.

Funcionamiento básico:

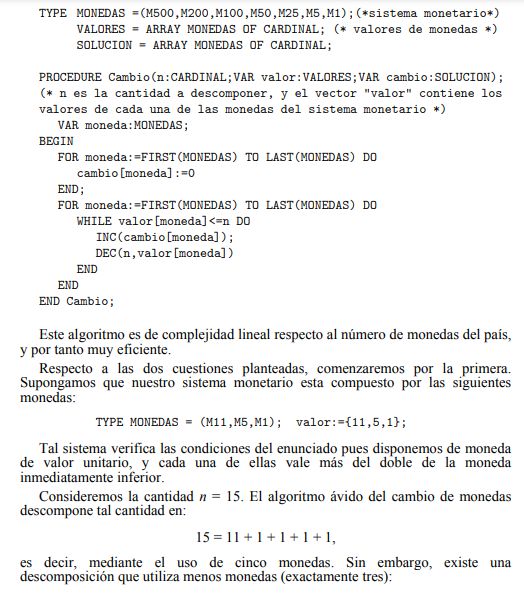
* Elegir el mejor elemento de C posible (más prometedor).
* Retirarlo del conjunto C de candidatos.
* Comprobar si produce una solución factible, en tal caso lo incluye en S.
* Si no, descartar.
* Repetir hasta alcanzar la función objetivo o agotar elementos de C.

Resumiendo, los algoritmos ávidos construyen la solución en etapas sucesivas, tratando siempre de tomar la decisión óptima para cada etapa.

**Regla más básica à** El tiempo de ejecución de un bucle es, como máximo, el tiempo de ejecución de las instrucciones contenidas dentro del bucle (incluyendo las comprobaciones) multiplicado por el número de iteraciones.

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza baja



Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente