UNIDAD 2 – ANÁLISIS DE ALGORITMOS.

1- Introducción

Un objetivo natural en el desarrollo de un programa computacional es mantener tan bajo como sea posible el consumo de los diversos recursos, aprovechándolos de la mejor manera que se encuentre. Se desea un buen uso, eficiente, de los recursos disponibles, sin desperdiciarlos.

Para que un programa sea práctico, en términos de requerimientos de almacenamiento y tiempo de ejecución, debe organizar sus datos en una forma que apoye el procesamiento eficiente.

Siempre que se trata de resolver un problema, puede interesar considerar distintos algoritmos, con el fin de utilizar el más eficiente. Pero, ¿cómo determinar cuál es "el mejor"?. La estrategia empírica consiste en programar los algoritmos y ejecutarlos en un computador sobre algunos ejemplares de prueba. La estrategia teórica consiste en determinar matemáticamente la cantidad de recursos (tiempo, espacio, etc.) que necesitará el algoritmo en función del tamaño del ejemplar considerado.

El tamaño de un ejemplar x corresponde formalmente al número de dígitos binarios necesarios para representarlo en el computador. Pero a nivel algorítmico consideraremos el tamaño como el número de elementos lógicos contenidos en el ejemplar.

2- Concepto de Eficiencia

Un algoritmo es **eficiente** cuando logra llegar a sus objetivos planteados utilizando la menor cantidad de recursos posibles, es decir, minimizando el uso memoria, de pasos y de esfuerzo humano.

Un algoritmo es **eficaz** cuando alcanza el objetivo primordial, el análisis de resolución del problema se lo realiza prioritariamente.

Puede darse el caso de que exista un algoritmo eficaz pero no eficiente, en lo posible debemos de manejar estos dos conceptos conjuntamente.

La eficiencia de un programa tiene dos ingredientes fundamentales: espacio y tiempo.

- La **eficiencia en espacio** es una medida de la cantidad de memoria requerida por un programa.
- La **eficiencia en tiempo** se mide en términos de la cantidad de tiempo de ejecución del programa.

Ambas dependen del tipo de computador y compilador, por lo que no se estudiará aquí la eficiencia de los programas, sino la eficiencia de los algoritmos. Asimismo, este análisis dependerá de si trabajamos con máquinas de un solo procesador o de varios de ellos. Centraremos nuestra atención en los algoritmos para máquinas de un solo procesador que ejecutan una instrucción y luego otra.

3- Medidas de Eficiencia

Inventar algoritmos es relativamente fácil. En la práctica, sin embargo, no se pretende sólo diseñar algoritmos, si no más bien que buenos algoritmos. Así, el objetivo es inventar algoritmos y probar que ellos mismos son buenos.

La calidad de un algoritmo puede ser avalada utilizando varios criterios. Uno de los criterios más importantes es el tiempo utilizado en la ejecución del algoritmos. Existen varios aspectos a considerar en cada criterio de tiempo. Uno de ellos está relacionado con el tiempo de ejecución requerido por los diferentes algoritmos, para encontrar la solución final de un problema o cálculo particular.

Normalmente, un problema se puede resolver por métodos distintos, con diferentes grados de

eficiencia. Por ejemplo: búsqueda de un número en una guía telefónica.

Cuando se usa un computador es importante limitar el consumo de recursos.

Recurso Tiempo:
- Aplicaciones informáticas que trabajan "en tiempo real" requieren que los cálculos se realicen en el menor tiempo posible.
- Aplicaciones que manejan un gran volumen de información si no se tratan adecuadamente pueden necesitar tiempos impracticables.

Recurso Memoria:
- Las máquinas tienen una memoria limitada.

4- Análisis A Priori y Prueba A Posteriori

El análisis de la eficiencia de los algoritmos (memoria y tiempo de ejecución) consta de dos fases: **Análisis A Priori** y **Prueba A Posteriori**.

El Análisis A Priori (o teórico) entrega una función que limita el tiempo de cálculo de un algoritmo. Consiste en obtener una expresión que indique el comportamiento del algoritmo en función de los parámetros que influyan. Esto es interesante porque:

- La predicción del costo del algoritmo puede evitar una implementación posiblemente laboriosa.
- Es aplicable en la etapa de diseño de los algoritmos, constituyendo uno de los factores fundamentales a tener en cuenta.

En la Prueba A Posteriori (experimental o empírica) se recogen estadísticas de tiempo y espacio consumidas por el algoritmo mientras se ejecuta. La estrategia empírica consiste en programar los algoritmos y ejecutarlos en un computador sobre algunos ejemplares de prueba, haciendo medidas para:
- una máquina concreta,
- un lenguaje concreto,
- un compilador concreto y
- datos concretos

La estrategia teórica tiene como ventajas que no depende del computador ni del lenguaje de programación, ni siquiera de la habilidad del programador. Permite evitar el esfuerzo inútil de programar algoritmos ineficientes y de despediciar tiempo de máquina para ejecutarlos. También permite conocer la eficiencia de un algoritmo cualquiera que sea el tamaño del ejemplar al que se aplique.

5- Análisis de algoritmos

Cuando un programa se va a usar repetidamente resulta importante que los algoritmos implicados sean eficientes. Generalmente, asociaremos eficiencia con el tiempo de ejecución del programa, y más raramente con la utilización de espacio de memoria.

Vamos a concentrarnos en la definición de eficiencia como el tiempo de ejecución de un algoritmo en función del tamaño de su entrada. A la eficiencia de un algoritmo también se le denomina costo, rendimiento o complejidad del algoritmo.

Para medir el costo de un algoritmo se pueden emplear dos enfoques:
- Pruebas
- Análisis

El primer enfoque (llamado benchmarking en inglés) consiste en elaborar una muestra

significativa o típica de los posibles datos de entrada del programa, y tomar medidas cuidadosas del tiempo de ejecución del programa para cada uno de los elementos de la muestra.

Una vez se entra en posesión de estos datos, se aplican técnicas estadísticas para inferir el rendimiento del programa para datos de entrada no presentes en la muestra. Debemos tener en cuenta que en determinadas circunstancias las conclusiones derivadas no siempre son fiables.

El otro enfoque, el análisis, emplea procedimientos matemáticos para determinar el costo del algoritmo; sus conclusiones son fiables, pero a veces su realización es difícil o imposible a efectos prácticos.

El análisis de algoritmos mediante el uso de herramientas como por ejemplo la evaluación de costos intenta determinar que tan eficiente es un algoritmo para resolver un determinado problema. En general el aspecto más interesante a analizar de un algoritmo es su costo.

Consideremos por un momento un programa que juega al ajedrez, el programa funcionar en base a un único algoritmo que se llama *SearchBestMove*. Que devuelve para una posición de piezas dada cual es la mejor movida para un cierto bando (blancas o negras). Con este algoritmo el programa es sencillo, dependiendo de quien empiece lo único que debe hacer es aplicar el algoritmo cada vez que le toque, esperar la movida del contrario y luego aplicar el algoritmo a la nueva posición. Pensemos ahora en el algoritmo SearchBestMove, este algoritmo para ser óptimo debería analizar todas las posibles jugadas, todas las posibles respuestas a dichas jugadas y así sucesivamente, formando un árbol de jugadas posibles del cual selecciona la jugada que produce mejor resultado final. Este aunque funcione en forma ultra-veloz tiene un grave problema de espacio, no hay memoria suficiente en ninguna máquina construida por el hombre para almacenar todas las posibles combinaciones de jugadas.

Importante: *El análisis del costo de un algoritmo comprende el costo en tiempo y en espacio.*

El primer paso que debe llevarse a cabo para analizar un algoritmo es definir con precisión lo que entendemos por tamaño de los datos de entrada.

El tamaño de los datos es una variable o expresión en función de la cual intentaremos medir la complejidad del algoritmo.

Es claro que para cada algoritmo la cantidad de recurso (tiempo, memoria) utilizados depende fuertemente de los datos de entrada. En general, la cantidad de recursos crece a medida que crece el tamaño de la entrada.

El análisis de esta cantidad de recursos no es viable de ser realizado instancia por instancia.

Se definen entonces las funciones de cantidad de recursos en base al **tamaño de la entrada**. Suele depender del número de datos del problema. Este tamaño puede ser la cantidad de dígitos para un número, la cantidad de elementos para un arreglo, la cantidad de caracteres de una cadena, en problemas de ordenación es el número de elementos a ordenar, en matrices puede ser el número de filas, columnas o elementos totales, en algoritmos recursivos es el número de recursiones o llamadas propias que hace la función.

Se debe elegir la misma variable para comparar algoritmos distintos aplicados a los mismos datos.

Normalmente tal definición resulta natural y no es complicado dar con ella. Por ejemplo,

1. En un algoritmo de ordenación, el tamaño de la entrada es el número de elementos a ordenar.
2. En un algoritmo de búsqueda, el tamaño de la entrada es el número de elementos entre los que hay que buscar uno dado.
3. En un algoritmo que actúa sobre un conjunto, el tamaño de la entrada es el número de elementos que pertenecen al conjunto.

Queremos saber la eficiencia de los algoritmos, no del computador. Por ello, en lugar de medir el tiempo de ejecución en microsegundos o algo por el estilo, nos preocuparemos del número de veces que se ejecuta una operación primitiva (de tiempo fijo).

Para estimar la eficiencia de este algoritmo, podemos preguntarnos, "si el argumento es una frase de **N** números, ¿cuántas multiplicaciones realizaremos?" La respuesta es que hacemos una multiplicación por cada número en el argumento, por lo que hacemos **N** multiplicaciones. La cantidad de tiempo que se necesitaría para el doble de números sería el doble.

Una vez definido el tamaño resulta conveniente usar una función T(n) para representar el número de unidades de tiempo (segundos, milisegundos,...) que un algoritmo tarda en ejecutarse cuando se le suministran unos datos de entrada de tamaño n.

Cualquier fórmula **T**(**N**) incluye referencias al parámetro **N** y a una serie de constantes"**Ti**" que dependen de factores externos al algoritmo como pueden ser la calidad del código generado por el compilador y la velocidad de ejecución de instrucciones del computador que lo ejecuta.

Dado que es fácil cambiar de compilador y que la potencia de los computadores crece a un ritmo vertiginoso (en la actualidad, se duplica anualmente), intentaremos analizar los algoritmos con algún nivel de independencia de estos factores; es decir, buscaremos estimaciones generales ampliamente válidas.

No se puede medir el tiempo en segundos porque no existe un computador estándar de referencia, en su lugar medimos el número de ***operaciones básicas o elementales***.

Las operaciones básicas son las que realiza el computador en tiempo acotado por una constante, por ejemplo:

- Operaciones aritméticas básicas
- Asignaciones de tipos predefinidos
- Saltos (llamadas a funciones, procedimientos y retorno)
- Comparaciones lógicas
- Acceso a estructuras indexadas básicas (vectores y matrices)

Es posible realizar el estudio de la complejidad de un algoritmo sólo en base a un conjunto reducido de sentenc ias, por ejemplo, las que más influyen en el tiempo de ejecución.

Como el tiempo de ejecución de un programa puede variar sustancialmente según el computador concreto en que se lleve a cabo la ejecución (no es lo mismo usar un PC que un IBM 709), resulta preferible que T(n) sea el número de instrucciones simples (asignaciones, comparaciones, operaciones aritméticas, etc.) que se ejecutan o equivalentemente, el tiempo de ejecución del algoritmo en un computador idealizado, donde cada asignación, comparación, lectura, etc. consume 1 unidad de

tiempo.

Por esta razón suele dejarse sin especificar las unidades de tiempo empleadas en T(n). Proceder de este modo no resulta restrictivo, porque si T(n) es el tiempo de ejecución del algoritmo en el modelo idealizado, entonces el tiempo de ejecución del algoritmo en el computador real X será aproximadamente igual a cX * T(n) para una cierta constante cX dependiente del computador. La constante cX puede ser determinada usando las técnicas de benchmarking y estadísticas convencionales.

Por otra parte, si T(n) es el costo de un algoritmo es perfectamente razonable que asumamos que n >= 0 y que T (n) es positiva para cualquier valor n.

Con mucha frecuencia el costo de un algoritmo depende de los datos de entrada particulares sobre los que opere, y no sólo del tamaño de esos datos. En tales casos, T(n) será a el costo en caso peor, es decir, el costo máximo del algoritmo para datos de entrada de tamaño n. En otras ocasiones interesa calcular el costo promedio del algoritmo para datos de entrada de tamaño n. Conocer el costo promedio de un algoritmo puede resultar en muchas ocasiones más útil que conocer el costo en caso peor (demasiado pesimista), pero el más difícil de calcular y se basa en la hipótesis de equiprobabilidad de las posibles entradas, hipótesis que no siempre es cierta.

6- Diseño de algoritmos

El diseño de algoritmos se encarga de encontrar cual es el mejor algoritmo para un problema determinado, en general existen algunos paradigmas básicos que pueden aplicarse para encontrar un buen algoritmo. Es claro que esta es una tarea difícil que requiere de conocimientos específicos y de una habilidad particular. Algunas de las técnicas mas utilizadas en el diseño de algoritmos son las siguientes:

- Dividir para conquistar.
- Algoritmos aleatorizados.
- Programación dinámica.
- Algoritmos golosos (Greedy).
- Algoritmos de heurísticos.
- Reducción a otro problema conocido.
- Uso de estructuras de datos que solucionen el problema.

7- Modelos computacionales

Para poder estudiar en detalle un algoritmo debemos fijar un marco en el cual podamos probar y analizar un algoritmo, así como también que permita comparar dos algoritmos entre sí. Este ambiente necesario para el estudio de los algoritmos se conoce como **"modelo computacional"**.

Maquina RAM de costo fijo

La máquina RAM proviene de Random Access Memory. Y es una máquina ideal muy similar a una computadora actual aunque con algunas simplificaciones. Los programas de la máquina RAM se almacenan en memoria.

Puede suponerse que todos los accesos a memoria tienen el mismo costo (en tiempo) y que todas las instrucciones tienen un costo constante e idéntico (en tiempo). El set de instrucciones de la máquina RAM está a compuesto por la gran mayoría de las instrucciones que podemos encontrar en un lenguaje de alto nivel.

Un programa para la máquina RAM se escribe en un pseudocódigo especial en el cual vamos a adoptar algunas convenciones básicas. Las llaves solo son necesarias si su ausencia afecta la claridad del código.

Los pasajes de parámetros a una función se hacen por valor. El acceso a un elemento de un

arreglo cuesta lo mismo que el acceso a una variable.

Algoritmo 1 - Este es un ejemplo
```
y <- 1              /* Asignación
A[2] <- 1           /* Asignación a un elemento de un vector
si n < 1 entonces
        X <- w
sino
        X <- y
fin_si
mientras n >= 0 hacer
        n  <- n -1
fin_mientras
para i desde 0 hasta 10 hacer
        A[i] <- 0
fin_para
```

Algoritmo 2 - Ejemplo simple
```
a  <- 3
b  <-  a * 5
si b = 5 entonces
        a  <- 2
sino
        a  <- 1
fin_si
```

El **algoritmo 2** tiene 5 instrucciones de las cuales se ejecutan únicamente 4. Por lo tanto el costo del programa en tiempo es de C*4, siendo C una constante que indica cuanto tiempo tarda en ejecutarse una instrucción. El espacio que necesita el programa es el espacio ocupado por las variables A y B. Es decir 2*Ec siendo Ec el espacio que ocupa una variable.

La máquina RAM de costo fijo es muy realista ya que puede verse como, claramente, a partir de un algoritmo escrito para la máquina RAM podemos desarrollar un algoritmo escrito en C, Pascal u otro lenguaje para una computadora actual.

Este acercamiento a la realidad en el código de la máquina RAM no se evidencia en cuanto al costo ya que el modelo planteado en el cual todas las instrucciones tienen un costo fijo no es real ya que hay algunas instrucciones que son claramente mas costosas que otras, por ejemplo una multiplicación insume mas tiempo que una suma en cualquier computadora.

Maquina RAM de costo variable

En la máquina RAM de costo variable cada instrucción Ii tiene asociado un costo Ci que le es propio y que depende del costo de implementar dicha instrucción.

Algoritmo 3 - Ejemplo simple II
```
a  <- 3
b  <-  a * 5
si b = 5 entonces
        a  <- 2
sino
        a  <- 1
fin_si
```

Este programa cuesta ahora 3*C1+1*C2+1*C3

Donde:

C1 = costo de una asignación.
C2 = costo de una multiplicación.
C3 = costo de comparar dos números.

El costo del espacio en la máquina RAM de costo variable es igual al de la máquina RAM de costo fijo.

Claramente observamos que pese a ganar claridad en cuanto a la escritura de programas perdemos precisión y se hace más complejo el cálculo de costos. Mas adelante veremos algunas herramientas que permitan simplificar el cálculo de dichos costos.

Importante: *A partir de este momento adoptaremos como base para el análisis de los algoritmos que estudiaremos la máquina RAM de costo variable.*

8- Algoritmos iterativos

Los algoritmos iterativos son aquellos que se basan en la ejecución de ciclos; que pueden ser de tipo for,while,repeat,etc. La gran mayoría de los algoritmos tienen alguna parte iterativa y muchos son puramente iterativos. Analizar el costo en tiempo de estos algoritmos implica entender cuantas veces se ejecuta cada una de las instrucciones del algoritmo y cual es el costo de cada una de las instrucciones.

Tipos de análisis

- Peor caso: indica el mayor tiempo obtenido, teniendo en consideración todas las entradas posibles.
- Mejor caso: indica el menor tiempo obtenido, teniendo en consideración todas las entradas posibles.
- Caso Medio: indica el tiempo medio obtenido, considerando todas las entradas posibles.

Como no se puede analizar el comportamiento sobre todas las entradas posibles, va a existir para cada prob lema particular un análisis en él:

- peor caso
- mejor caso
- caso promedio (o medio)

El caso promedio es la medida más realista de la performance, pero es más difícil de calcular pues establece que todas las entradas son igualmente probables, lo cual puede ser cierto o no. Trabajaremos específicamente con el "**peor caso"**.

Ordenamiento Uno contra Todos

Uno de los temas importantes del curso es el análisis de algoritmos de ordenamiento, por lo que vamos a empezar con uno de los más simples: el de uno contra todos.

```
Algoritmo Uno_contra_todos (A,n). Ordena el vector A.
para i desde 1 hasta (n -1) hacer
        para  j desde (i + 1) hasta n hacer
                si A[j] < A[i] entonces
                        Swap(A[i], A[j])
                fin_si
        fin_para
fin_para
```

Ejemplo:

3 4 1 5 2 7 6 4 Compara A[1] con A[2]
3 4 1 5 2 7 6 4 Compara A[1] con A[3]
1 4 3 5 2 7 6 4 Compara A[1] con A[4]
1 4 3 5 2 7 6 4 Compara A[1] con A[5]
1 4 3 5 2 7 6 4 Compara A[1] con A[6]
1 4 3 5 2 7 6 4 Compara A[1] con A[7]
1 4 3 5 2 7 6 4 Compara A[1] con A[8]
1 4 3 5 2 7 6 4 Compara A[2] con A[3]
1 3 4 5 2 7 6 4 Compara A[2] con A[4]
1 3 4 5 2 7 6 4 Compara A[2] con A[5]
1 2 4 5 3 7 6 4 Compara A[2] con A[6]
1 2 4 5 3 7 6 4 Compara A[2] con A[7]
1 2 4 5 3 7 6 4 Compara A[2] con A[8]
1 2 4 5 3 7 6 4 Compara A[3] con A[4]
1 2 4 5 3 7 6 4 Compara A[3] con A[5]
1 2 3 5 4 7 6 4 Compara A[3] con A[6]
1 2 3 5 4 7 6 4 Compara A[3] con A[7]
1 2 3 5 4 7 6 4 Compara A[3] con A[8]
1 2 3 5 4 7 6 4 Compara A[4] con A[5]
1 2 3 4 5 7 6 4 Compara A[4] con A[6]
1 2 3 4 5 7 6 4 Compara A[4] con A[7]
1 2 3 4 5 7 6 4 Compara A[4] con A[8]
1 2 3 4 5 7 6 4 Compara A[5] con A[6]
1 2 3 4 5 7 6 4 Compara A[5] con A[7]
1 2 3 4 5 7 6 4 Compara A[5] con A[8]
1 2 3 4 4 7 6 5 Compara A[6] con A[7]
1 2 3 4 4 6 7 5 Compara A[6] con A[8]
1 2 3 4 4 5 7 6 Compara A[7] con A[8]
1 2 3 4 4 5 6 7 FIN.

Para un vector de 8 elementos el algoritmo insumió 28 comparaciones. Antes de analizar en forma genérica el costo en tiempo del algoritmo veamos ejemplos más simples.

Costo de una instrucción en ciclos simples

```
para i desde 1 hasta n hacer
        instrucción
fin_para
```

El cálculo en costo de una instrucción en un ciclo simple puede hacerse utilizando una sumatoria.

$$\sum_{i=1}^{n} C1$$

Donde C1 es el costo de la instrucción que se ejecuta en el ciclo. El resultado de la sumatoria es n * C1, lo cual es evidente porque la instrucción dentro del ciclo se ejecuta n veces.

Costo de una instrucción en ciclos anidados independientes

```
para i desde 1 hasta n hacer
        para j desde 3 hasata m hacer
                instrucción
        fin_para
fin_para
```

Podemos calcular el costo de la instrucción nuevamente usando sumatorias de la forma:

$$\sum_{i=1}^{n}\sum_{j=3}^{m} C1 = \sum_{i=1}^{n}(m-2)*C1 = n*(m-2)*C1$$

Costo de una instrucción en ciclos anidados dependientes

Cuando los ciclos dependen uno del otro podemos aplicar la misma técnica.

```
para i desde 1 hasta n hacer
        para j desde i hasata n hacer
                instrucción
        fin_para
fin_para
```

$$\sum_{i=1}^{n}\sum_{j=i}^{n} C1 = \sum_{i=1}^{n}(n-i+1)*C1 = C1*\sum_{i=1}^{n}(n-i+1) =$$

$$= C1*(\sum_{i=1}^{n} n - \sum_{i=1}^{n} i + \sum_{i=1}^{n} 1) = C1*(n*n - \frac{n*(n+1)}{2} + n) = C1*(n^2 - \frac{n^2+n}{2} + n) =$$

$$= C1*(\frac{2n^2 - n^2 - n + 2n}{2}) = C1*\frac{n^2+n}{2}$$

Antes de continuar vamos a recordar algunas sumatorias útiles.

$$\sum_{i=1}^{n} c = c*n$$

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

$$\sum_{i=1}^{n} \frac{1}{i} = \ln n + 1$$

Análisis del algoritmo "Uno contra Todos"

Pasemos nuevamente al algoritmo.

```
Algoritmo Uno_contra_todos (A,n). Ordena el vector A.
para i desde 1 hasta (n -1) hacer
        para  j desde (i + 1) hasta n hacer
                si A[j] < A[i] entonces
                        Swap(A[i], A[j])
                fin_si
        fin_para
fin_para
```

El costo del algoritmo lo vamos a calcular suponiendo que C1 es el costo de efectuar la comparación y que C2 es el costo de efectuar el SWAP. Sin embargo el SWAP no se hace siempre sino que se hace únicamente cuando la comparación da A[j] > A[i], por ello debemos particionar el análisis en tres casos: el peor caso, el mejor caso y el caso medio.

Nota: *No se tiene en cuenta en este ejemplo el costo de las instrucciones* ***Para_Fin_para****, solo se tienen en cuenta los costos de las instrucciones de comparación y Swap.*

Peor caso

En el peor caso el algoritmo siempre hace el Swap. Esto ocurre por ejemplo cuando el vector viene ordenado en el orden inverso al que utilizamos.

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (C1 + C2)$$

$$T(n) = \sum_{i=1}^{n-1} (C1 + C2) * (n - (i+1) + 1)$$

$$T(n) = \sum_{i=1}^{n-1} (C1 + C2) * (n - i)$$

$$T(n) = (C1 + C2) * (\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i)$$

$$T(n) = (C1 + C2) * ((n-1) * n - \frac{(n-1) * n}{2})$$

$$T(n) = (C1 + C2) * (n^2 - n - \frac{n^2 - n}{2})$$

$$T(n) = (C1 + C2) * (\frac{2n^2 - 2n - n^2 + n}{2})$$

$$T(n) = (C1 + C2) * (\frac{n^2 - n}{2})$$

$$T(n) = (C1 + C2) * (\frac{1}{2} n^2 - \frac{1}{2} n)$$

Por ejemplo para N = 8 el peor caso nos da: (C1+C2)*(32–4)=(C1+C2)*28 un total de 28 comparaciones y 28 Swaps.

Mejor caso

En el mejor caso el vector ya viene ordenado por lo que nunca efectúa ningún Swap, el costo total es entonces el costo de las comparaciones que es igual a lo calculado antes.

$$T(n) = C1 * (\frac{1}{2} n^2 - \frac{1}{2} n)$$

Caso medio

En el caso medio el algoritmo efectúa todas las comparaciones y solo la mitad de los Swaps.

$$T(n) = C1 * (\frac{1}{2} n^2 - \frac{1}{2} n) + C2 * \frac{1}{2} (\frac{1}{2} n^2 - \frac{1}{2} n)$$

Importante: *Al analizar el tiempo de un algoritmo se debe analizar el mejor caso, el peor caso y el caso medio. Habitualmente el que más interesa es el peor caso.*

Otro ejemplo

Considérese el siguiente algoritmo de localización del mínimo elemento de una tabla que almacena m>0 elementos enteros. El tamaño de la entrada de la función será a m.

Nota: *En este ejemplo SI se tiene en cuenta el costo de las instrucciones* ***Para_Fin_para****, además de las instrucciones de comparación y asignación.*

```
1. tipo tabla enteros = tabla [1..MAX] de entero
2. fin_tipo
3.
4. función localización_mínimo (ent A: tabla_enteros; ent m: entero) retorna entero
5.         var j, min : entero
6.         fin_var
7.         min := 1
8.         para j desde 2 hasta m hacer 10
9.                   si  A[j] < A[min] entonces
10.                            j := min
11.                  fin_si
12.        fin_para
13.        retorna min
14. fin_función
```

Al tiempo de ejecución del algoritmo sólo contribuyen ciertas líneas de este algoritmo, no todas. Por ejemplo, las declaraciones del tipo y de la función no consumen tiempo porque sólo son consideradas cuando el algoritmo es compilado y no al ser ejecutado.

Empecemos las cuentas. En la línea (7) efectuamos una asignación, por lo que cargamos una unidad de tiempo a la ”factura".

En la línea (8) cargamos dos unidades de tiempo por cada iteración que se haga, ya que en cada iteración se le da valor a j (se incrementa) y se comprueba si hemos llegado al final de la iteración o no; más dos unidades de tiempo de la iteración final, en la que j:=m+1, la comparación entre j y m fracasa, y se abandona el bucle para sin ejecutar el cuerpo del mismo.

La iteración para realiza (m–1) iteraciones. El cuerpo de la iteración (líneas (9) a (11)) puede tener un costo de una o de dos unidades.

La evaluación de las condiciones consume una unidad siempre, pero en función del resultado consumiremos una unidad más por la asignación j:=min (línea (10)) o no habrá a nada más que hacer.

Como vamos a evaluar el costo en caso peor, adoptamos la hipótesis pesimista y suponemos que siempre habrá a que gastar dos unidades de tiempo en el cuerpo de la iteración, en todas las iteraciones. Si la tabla estuviera ordenada decrecientemente, ocurriría justamente esto.

Para finalizar el contabilización de operaciones realizadas, añadimos al total una unidad de tiempo por el retorno de la expresión min.

Resumiendo, si T (m) es el costo (en caso peor) de localización mínimo para una entrada de tamaño m tenemos:

$$T(m) = \underset{(7)}{1} + \underset{(8)}{2\,(m-1)+2} + \underset{(9)\text{ y }(10)}{2\,(m-1)} + \underset{(13)}{1} = 4\,(m-1) + 4 = 4m$$

9- <u>Orden de un algoritmo</u>

El análisis de los algoritmos que hemos realizado hasta aquí es muy preciso pero resulta incómodo para entender que tan eficiente es un algoritmo o para poder compararlo contra otro algoritmo.

Supóngase que para cierto problema hemos hallado dos posibles algoritmos que lo resuelven, A y B.

Evaluamos con cuidado sus costos respectivos y obtenemos que $T_A(n) = 100n$ y $T_B(n) = 2n^2$.

¿Cuál será conveniente usar, basándose en el criterio de eficiencia? Si n < 50, resulta que B es más eficiente que A, pero no mucho más; para n>50 el algoritmo A es mejor que B, y la ventaja de A sobre B se hace mucho mayor cuánto mayor es n.

Para entradas de tamaño n=100, A es el doble de rápido que B, pero para entradas de tamaño n = 1000, A es veinte veces más rápido que B.

Este pequeño ejemplo muestra que es mucho más importante la forma funcional (n vs. n^2) de los costos que no las constantes que intervienen(2 vs. 100).

Como además el tiempo de ejecución del algoritmo en un computador real requiere que multipliquemos el costo por una cierta constante sólo mensurable a través de la experimentación, no tiene caso que nos preocupemos de calcular los factores multiplicativos a la hora de analizar un algoritmo.

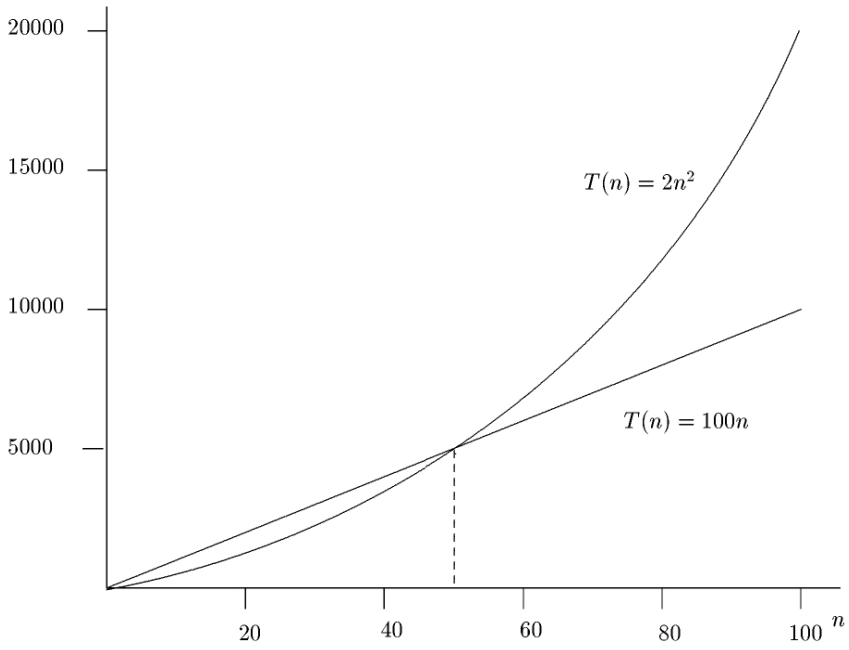


Figura 1: Comparación de los costes $100n$ y $2n^2$

10- <u>Notación "O"</u>

Esta notación sirve para clasificar el crecimiento de las funciones. Así en vez de decir que el costo T(k) del algoritmo de localización del mínimo es T(k) = 4k, diremos que es O(k), lo que informalmente significa que el costo es "alguna constate multiplicada por k" para entradas de tamaño k.

Habitualmente no interesa cuanto tarda exactamente un algoritmo sino que nos interesa saber cual es la tasa de crecimiento del algoritmo en función de los datos de entrada.

Para el estudio de algoritmos existe una notación muy práctica y utilizada universalmente en este tipo de problemas conocida como la gran "O" (Big-Oh notation) que define el orden de un algoritmo.

<u>Importante</u>: *Orden de un algoritmo: Tasa de crecimiento del tiempo que insume el algoritmo en función de la cantidad o tamaño de los datos de entrada.*

La definición formal es la siguiente: Sea *f(n)* una función definida sobre los números enteros positivos n. Diremos que *T(n)* es *O(f(n))* si *T(n)* es menor o igual que *f(n)* multiplicada por cierta

constante, excepto para ciertos valores de n.

Más formalmente, *T(n)* es *O(f(n))* si existe una constante c > 0 y un número n_o tales que para todo n >= n_o se cumple que

$$T(n) <= c * f(n)$$

Por ejemplo, si $T(n)=(n+1)^2$ entonces T(n) es $O(n^2)$, ya que para c=4 y $n_o=1$ se cumple la definición anterior, $(n + 1)^2 < 4n^2$ para n >= 1.

Puede parecer un contrasentido decir que $(n + 1)^2$ es $O(n^2)$ ya que $(n + 1)^2$ es mayor que n^2 para toda n, pero al fin y al cabo, ambas funciones crecen a ritmos equivalentes y eso es lo que expresamos al decir que $(n + 1)^2$ es $O(n^2)$.

De hecho, también es cierto que $(n+1)^2$ es $O(n^2/100)$ (tómese $n_o = 1$ y c = 400).

Se dice que una función f(n) es del mismo orden o de orden inferior que g(n) si f(n) es O(g(n)). Si g(n) es O(f(n)) entonces f(n) y g(n) son del mismo orden y, si g(n) no es O(f(n)) entonces f(n) es de orden inferior a g(n).

Proposiciones

1. Para cualesquiera constantes 0 <= a <= b, se cumple que n^a es $O(n^b)$. además n^a es de orden inferior a n^b si a < b, y son del mismo orden si a = b.

2. Para cualesquiera constantes a >= 0, b > 0 y c > 1, se cumple que $(\log_c n)^a$ es $O(n^b)$ y es de orden inferior.

3. Para cualesquiera constantes a >= 0 y b > 1, se cumple que n^a es $O(b^n)$ y de orden inferior.

4. Para cualquier constante b > 1, se cumple que b^n es O(n!) y es de orden inferior.

Recapitulando sobre lo que ya hemos visto, los principios generales que podemos extraer son los siguientes:

- Los factores constantes no importan. Si T(n) es O(f(n)) entonces para cualesquiera constantes d1, d2 > 0, d1 * T(n) es O(d2 * f(n)).

- Los términos de orden inferior no importan. Por ejemplo, si T(n) es un polinomio

  $$T(n) = a_k n^k + a_{k-1} n^{k-1} + ... + a_o$$

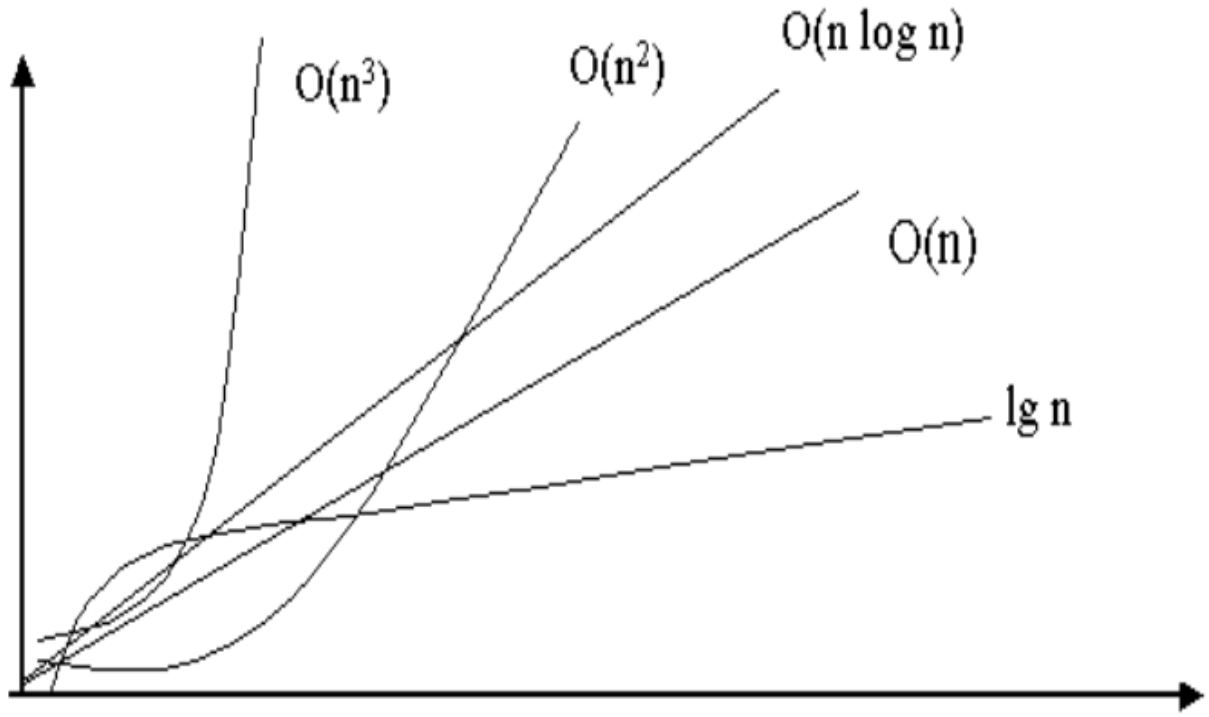
  siendo $a_k > 0$, podemos concluir que T(n) es $O(n^k)$. De hecho, si T(n) es una suma de funciones $f_i(n)$ positivas entonces T (n) es $O(f_k(n))$, donde $f_k(n)$ es la función de mayor orden entre las distintas funciones $f_i(n)$. Otra forma de expresar esto mismo, es decir que si f(n) es O(g(n)) entonces max{f(n), g(n)} es O(g(n)).

- La base de los logaritmos no importa. Puesto que $\log_b n = \log_c n * \log_b c$, para cualesquiera constantes b, c > 1 y puesto que $\log_b c$ es una constante positiva, dos funciones logarítmicas son siempre del mismo orden. Usaremos la expresión O(log n), sin especificar la base, ya que carece de importancia.

- Las expresiones usando la notación O son transitivas. Si f(n) es O(g(n)) y g(n) es O(h(n)) entonces f(n) es O(h(n)).

Orden	Nombre informal	Comentario
O(1)	Orden constante	Todo aquel algoritmo que responde en un tiempo constante. Son los que aplican alguna fórmula sencilla.
O(log n)	Orden logarítmico	El tiempo crece con un criterio logarítmico independientemente de cuál sea la base mientras que esta sea mayor que 1. Esto implica que un bucle realiza menos iteraciones que el tamaño de los datos del problema.
O(n)	Orden lineal	Es un orden bueno
O(n log n)	Orden linearítmico	Es un orden relativamente bueno
$O(n^2)$	Orden cuadrático	
$O(n^3)$	Orden cúbico	
$O(n^k)$	Orden polinómico (de grado k>3)	Incluye al cuadrático y al cúbico. Se puede decir que este orden es el último aceptable. A partir del siguiente, los algoritmos son complicados de tratar
$O(C^n)$	Orden exponencial (c>1)	Crecen mucho más rápido que el polinomial. Es un problema intratable.
O(n!)	Orden factorial	El algoritmo prueba todas las combinaciones posibles.
$O(n^n)$	Orden combinatorio	Tan intratable como el anterior. A menudo no se hace distinción entre ellos.

Nombres informales para algunos órdenes comunes



Curvas de crecimientos de algunos órdenes comunes

Reglas

Antes de dar unas cuántas reglas para calcular el costo de un algoritmo veamos un par de convenios útiles al usar la notación O. Si el costo de un algoritmo o de una de sus partes es independiente del tamaño de la entrada, es decir, constante, diremos que su costo es O(1). Por otra parte, en vez de decir T(n) es O(f(n)) escribiremos T(n) = O(f(n)), aunque esta convención encierra

algún peligro porque "=" no equivale a "es" y no es cierto que O(f(n)) =T (n).

A la hora de analizar un algoritmo, debemos tener en cuenta que cualquier asignación, lectura de variables, escritura, operación aritmética o comparación tiene costo O(1). Si en alguna de las expresiones evaluadas o las comparaciones interviene una función entonces se añade el costo de la función.

Regla de las sumas: Consideremos un algoritmo que consta de k partes compuestas secuencialmente, cada una con un costo $T_i(n)$ y donde $T_i(n) = O(f_i(n))$ para i = 1, .., k. Entonces

$$T(n) = T_1 (n) + ... + T_k(n) = O(max\{ f_1(n), ... f_k(n)\}).$$

Será muy conveniente que la regla de las sumas se emplee usando las expresiones $O(f_i(n))$ más ajustadas posibles y con arreglo a los criterios que exponíamos antes.

Supóngase que un algoritmo consta de dos partes. Si la primera tiene costo $O(n^2)$ y la segunda tiene costo $O(n^3)$, entonces el costo del algoritmo sería

$$T(n) = O(n^2) + O(n^3) = O(n^2 + n^3) = O(max\{n^2, n^3\}) = O(n^3)$$

Para obtener el costo de una estructura alternativa se han de contabilizar los costos de evaluar cada una de las condiciones (típicamente todas tienen el mismo costo) y los costos de las acciones emprendidas en cada uno de los casos.

Como pretendemos obtener el costo en el caso peor, supondremos que siempre se ejecuta la acción más costosa (salvo que podamos demostrar lo contrario) y se aplica la regla de las sumas.

Para las estructuras iterativas debemos evaluar el costo del cuerpo y el número de iteraciones que se producirán en función del tamaño de la entrada. En iteraciones para el número de iteraciones se puede obtener fácilmente, pero para las iteraciones mientras puede que no resulte sencillo y nos hayamos de contentar encontrando una cota superior al número de iteraciones expresada con la notacióon

O. Si el número de iteraciones es nulo, el costo total de la estructura iterativa será a nulo, pero salvo en este caso trivial el costo total de la estructura iterativa se obtiene usando la llamada regla de los productos.

Regla de los productos: Si el costo del cuerpo de una iteración es O(f(n)) y el número de iteraciones es O(g(n)) entonces el costo total de la iteración es O(f(n) * g(n)).

Si en un algoritmo aparecen bucles anidados se analizan uno por uno desde el más interno al más externo. Por ejemplo, el siguiente segmento de un algoritmo inicializa las componentes de una matriz n x n a 0.

```
para i desde 1 hasta n hacer
        para j desde 1 hasta n hacer
                C[i, j] : = 0
        fin_para
fin_para
```

Si tomamos como tamaño de la entrada la dimensión de la matriz, n, el costo del segmento completo es $O(n^2)$.

En efecto, para el bucle más interno (donde se usa j), el costo del cuerpo es O(1) y el número de iteraciones que se efectúan es O(n); luego el costo del bucle es O(n).

Para el bucle externo, el número de iteraciones realizadas es O(n), y el costo de su cuerpo es el costo previamente calculado: O(n).

Aplicando la regla del producto llegamos a la respuesta buscada: el costo total es O(n * n) = $O(n^2)$.

Aplicando esta notación a los algoritmos estudiados podemos ver que el mejor caso Uno contra Todos es $O(n^2)$. El peor caso es $O(n^2)$ por lo tanto el algoritmo es $O(n^2)$. Es decir que es un algoritmo de orden cuadrático.

Cuanto mas rápido sea el crecimiento de una función peor será el rendimiento del algoritmo. Por eso analizando el orden del algoritmo podemos determinar cual es más eficiente.

Algunas funciones típicas ordenadas por orden de crecimiento.

$$\sqrt{n} < \log n < n < n \log n < n^c < x^n < n! < n^n$$

Ejemplos de algoritmos de distintos Orden

O(1)
En O(1) están los algoritmos que se ejecutan siempre en la misma cantidad de pasos, sin importar el tamaño de la entrada:

```
bool IsFirstElementNull(IList<string> elements)
{
   return elements[0] == null;
}
```

O(n)
En O(n) están los algoritmos cuyo tiempo de ejecución crece de forma lineal y en proporción directa al tamaño de su entrada:

```
bool ContainsValue(IList<string> elements, string value)
{
   foreach (var element in elements)
   {
      if (element == value) return true;
   }
   return false;
}
```

O(n^2)
En O(n^2) están los algoritmos cuyo tiempo de ejecución crece de forma proporcional al cuadrado del tamaño de su entrada:

```
bool ContainsDuplicates(IList<string> elements)
{
   for (var outer = 0; outer < elements.Count; outer++)
   {
      for (var inner = 0; inner < elements.Count; inner++)
      {
         // Don't compare with self
         if (outer == inner) continue;
         if (elements[outer] == elements[inner]) return true;
      }
   }
   return false;
}
```

$O(2^n)$

En $O(2^n)$ están los algoritmos cuyo tiempo de ejecución se duplica cada vez que el tamaño de la entrada se incrementa de una unidad:

```
int Fibonacci(int number)
{
   if (number <= 1) return number;
   return Fibonacci(number - 2) + Fibonacci(number - 1);
}
```

11- Ejemplo de cálculo de costo y orden de un algoritmo

Para resolver los problemas vamos a considerar una MT (Máquina de Túring) de costo fijo. Todas las operaciones (asignaciones, lecturas, comparaciones, sumas, productos, etc.) van a tener un costo de valor 1 c/u.

Si en una instrucción hay por ejemplo una asignación y una suma, se sumarán los valores de c/u y su valor sería de 2.

Las estructuras repetitivas como el Para (For) van a tener un costo mayor ya que deben realizar comparaciones, incrementar el valor de la variable de control y comparar.

Ejemplo

```
i = 0                          C1 = 1 (costo de la asignación)                          Línea 1
A = 1                          C2 = 1 (costo de la asignación)                          Línea 2
B = 5                          C3 = 1 (costo de la asignación)                          Línea 3
Mientras i <= 5 Hacer          C4 = ? (costo del Mientras)                              Línea 4
    Para j = A hasta B Hacer   C5 = ? (costo del Para)                                  Línea 5
        Leer Dato(j)           C6 = 1 (costo de la lectura)                             Línea 6
        Si Dato(j) < 100 Entonces  C7 = 1 (costo de comparar)                           Línea 7
            i = i + 1          C8 = 2 (costo de la suma + asignación)                   Línea 8
        Fin_Si                                                                          Línea 9
    Fin_Para                                                                            Línea 10
    D = j                      C9 = 1 (costo de la asignación)                          Línea 11
    E = D + 5                  C10 = 2 (costo de la suma + asignación)                  Línea 12
    i = i + 1                  C11 = 2 (costo de la suma + asignación)                  Línea 13
Fin_Mientras                                                                            Línea 14
```

El Costo Total de este algoritmo será

$T(n) = C1 + C2 + C3 + C4 = 1 + 1 + 1 + C4 = 3 + C4$ (1)

Las líneas 9, 10 y 14 no tendrán costo.

Necesitamos saber cuál es el valor de C4.

Al ser una estructura repetitiva Mientras, vamos a considerar su costo como 1, ya que sólo tiene una comparación, pero ese costo de 1 debe ser multiplicado por la cantidad de repeticiones que se den en el Mientras. ¿Cómo sé cuántas repeticiones serán?

La única forma que tengo es hacer una prueba de escritorio, ya que uno al ver la estructura podría decir que repite 5 veces, pero veamos:

Valor de i	i <= 5	Valor comparación
0	0 <= 5	Verdadero
1	0 <= 5	Verdadero
2	0 <= 5	Verdadero
3	0 <= 5	Verdadero
4	0 <= 5	Verdadero
5	0 <= 5	Verdadero
6	0 <= 5	Falso

Como podemos ver, la estructura Mientras se repite 6 veces, las 6 veces que la condición i <= 5 fue verdadera

Los costos de las estructuras repetitivas los vamos a representar con la sumatoria

Costo del Mientras

C4 = $\sum_{i=1}^{6} (1 + C5 + C9 + C10 + C11) + 1$ (la sumatoria de 1 a 6 es por las 6 repeticiones)

El valor de 1 es el costo del mientras en cada repetición, la comparación que hace el mientras de i <= 5. Los valores de C9, C10 y C11 son los costos de las asignaciones que están dentro del Mientras, tanto el valor del mientras como el de las asignaciones están afectados por la cantidad de repeticiones del Mientras (6).

El 1 que está al final es el valor del mientras en la última comparación, i <= 5 cuando i vale 6 y esa comparación posee el valor falso que hace que se salga de la estructura Mientras.

Reemplazamos los valores que conocemos y tenemos:

C4 = $\sum_{i=1}^{6} (1 + C5 + 1 + 2 + 2) + 1$

C4 = $\sum_{i=1}^{6} (1 + C5 + 5) + 1$

C4 = $\sum_{i=1}^{6} (6 + C5) + 1$

Aplicando propiedades de Sumatoria

C4 = $\sum_{i=1}^{6} 6 + \sum_{i=1}^{6} C5 + 1$

C4 = $6 * 6 + 6 * C5 + 1 = 37 + 6 * C5$ (2)

Costo del Para

La cantidad de repeticiones del Para de este ejemplo es de 5, ya que son 5 repeticiones desde A=1 hasta B=5 (queda para los alumnos realizar la prueba de escritorio para confirmar las 5 repeticiones), más un costo adicional que es el que corresponde cuando A toma el valor 6 y se compara contra el valor de B y sale del bucle.

En el Para vamos a tomar un costo de 3 (1 por asignar el valor a la variable de control j, otro 1 por comparar si j llegó al valor B y otro 1 por incrementar esa variable de control j).

C5 = $\sum_{j=1}^{5} (3 + C6 + C7 + C8) + 3$ (los valores de 3 son del Para y de su adicional)

Consideramos que C7, la comparación, será verdadera siempre, ya que estamos considerando el Peor Caso, por lo tanto siempre se va a ejecutar C8 también.

C5 = $\sum_{j=1}^{5} (3 + 1 + 1 + 2) + 3$

C5 = $\sum_{j=1}^{5} 7 + 3$

C5 = $5 * 7 + 3 = 35 + 3 = 38$ (3)

Reemplazando (3) en (2)

C4 = $37 + 6 * 38 = 37 + 228 = 265$ (4)

Reemplazando (4) en (1)

T(n) = 3 + C4 = 3 + 265 = **268**

En el caso del Orden sería ***O*(*n*²)**, ya que tenemos una estructura repetitiva dentro de la otra; pero cuidado, ***siempre*** hay que realizar las pruebas de escritorio para ver si las estructuras repetitivas se repiten más de una vez.

En el algoritmo del ejemplo llegamos a un único valor, porque sabemos las cantidades de repeticiones, pero si cambiamos el algoritmo por el siguiente no podremos saber el valor exacto del costo sino un valor en función a la cantidad de datos ***n*** (realice el cálculo del costo y orden como tarea).

```
i = 0                          C1 = 1 (costo de la asignación)                          Línea 1
A = 1                          C2 = 1 (costo de la asignación)                          Línea 2
Mientras i <= n Hacer          C4 = ? (costo del Mientras)                              Línea 3
    Para j = A hasta n Hacer   C5 = ? (costo del Para)                                  Línea 4
        Leer Dato(j)           C6 = 1 (costo de la lectura)                             Línea 5
        Si Dato(j) < 100 Entonces  C7 = 1 (costo de comparar)                           Línea 6
            i = i + 1          C8 = 2 (costo de la suma + asignación)                   Línea 7
        Fin_Si                                                                          Línea 8
    Fin_Para                                                                            Línea 9
    E = j + 5                  C10 = 2 (costo de la suma + asignación)                  Línea 10
    i = i + 1                  C11 = 2 (costo de la suma + asignación)                  Línea 11
Fin_Mientras                                                                            Línea 12
```

Ayuditas

- Si el algoritmo posee hay una estructura repetitiva vamos a decir que es de ***O*(*n*),**
- Si el algoritmo posee hay una estructura repetitiva dentro de otra vamos a decir que es de ***O*(*n*²).**
- Si el algoritmo posee hay una estructura repetitiva dentro de otra, que a su vez está dentro de otra estructura repetitiva vamos a decir que es de ***O*(*n*³).**
- Siempre verificar que las estructuras repetitivas se repiten más de una vez, ya que si no lo hacen el Orden variará y los puntos anteriores no se cumplirán.

12- Algoritmos recursivos

Una técnica bastante poderosa a la hora de realizar algoritmos consiste en programar en forma recursiva. Un algoritmo recursivo es aquel que en algún momento durante su ejecución se invoca a si mismo. Por ejemplo el siguiente programa sirve para calcular el factorial de un número.

Algoritmo Fact(n). Calcula el factorial de n en forma recursiva.
si n < 2 entonces
    devolver 1
sino
    devolver n*Fact(n-1)
fin_si

En general el programar en forma recursiva permite escribir menos código y algunas veces nos permite simplificar un problema difícil de solucionar en forma iterativa. En cuanto a la eficiencia un programa recursivo puede ser mas, menos o igual de eficiente que un programa iterativo.

En primer lugar hay que señalar que los programas recursivos consumen espacio en memoria para la pila ya que cada una de las llamadas recursivas al algoritmo son apiladas una sobre otras para preservar el valor de las variables locales y los parámetros utilizados en cada invocación.

Este aspecto lo vamos a considerar un costo en espacio ya que lo que estamos consumiendo es memoria. En cuanto al tiempo de ejecución del programa debemos calcularlo de alguna forma para poder obtener el orden del algoritmo. Para ello se recurre a plantear el tiempo que insume un programa recursivo de la siguiente forma.

Tiempo de ejecución del factorial

$$T(n) = \begin{cases} 1 & \text{si n < 2} \\ 1 + T(n-1) & \text{sino} \end{cases}$$

Las ecuaciones planteadas definen una recurrencia. Una recurrencia es un sistema de ecuaciones en donde una o más de las ecuaciones del sistema están definidas en función de si mismas. Para calcular el orden de un algoritmo recursivo es necesario resolver la recurrencia que define el algoritmo.

Recurrencias

Tomemos por ejemplo la siguiente recurrencia muy común en algoritmos recursivos.

$$T(n) = \begin{cases} 1 & \text{si n = 1} \\ 2T(\frac{n}{2}) + n & \text{sino} \end{cases}$$

Estudiaremos tres técnicas que suelen utilizarse para resolver recurrencias.

- El método de sustitución
- El método iterativo
- El teorema maestro de las recurrencias.

El método de sustitución

En método de sustitución se utiliza cuando estamos en condiciones de suponer que el resultado de la recurrencia es conocido. En estos casos lo que hacemos es sustituir la solución en la recurrencia y luego demostrar que el resultado es valido por inducción.

Para nuestro ejemplo sabemos que

$$T(n) = (n \log n) + n$$

Veamos que pasa cuando n = 1

Si n = 1 => T(1)=1

1 log 1 + 1 = 0 + 1 = 1 Verifica

Hipótesis inductiva

si n > 1 => T (n') = (n' log n') + n' si n' < n

Demostración

$$T(n) = 2T(\frac{n}{2}) + n \quad (1)$$

Como $\frac{n}{2} < n$

Por inducción

$$T(\frac{n}{2}) = (\frac{n}{2}) * \log(\frac{n}{2}) + \frac{n}{2} \quad (2)$$

Reemplazando (2) en (1)

$$T(n) = 2((\frac{n}{2}) * \log(\frac{n}{2}) + \frac{n}{2}) + n$$

$$T(n) = (n\log(\frac{n}{2}) + n) + n$$

$$T(n) = n(\log n - \log 2) + 2n$$

$$T(n) = n\log n - n + 2n$$

$$T(n) = n\log n + n$$

$$T(n) = \Theta(n\log n)$$

Por lo que queda demostrado.

El método de sustitución es simple pero requiere que uno ya conozca cual es la solución de la recurrencia. A veces sin embargo en recurrencias muy complejas para resolver por otros métodos puede llegar a ser conveniente intentar adivinar la solución y luego demostrarla usando este método.

El método iterativo

El método iterativo es un forma bastante poderosa de resolver una recurrencia sin conocer previamente el resultado. En este método se itera sobre la recurrencia hasta que se deduce un cierto patrón que permite escribir la recurrencia usando sumatorias. Luego realizando una sustitución apropiada y resolviendo las sumatorias se llega al resultado de la recurrencia. Utilizando el método iterativo en nuestro ejemplo observamos lo siguiente.

$$T(1) = 1$$

$$T(n) = 2T(\frac{n}{2}) + n$$

$$T(n) = 2(2T(\frac{n}{4}) + \frac{n}{2}) + n = 4T(\frac{n}{4}) + n + n$$

$$T(n) = 4(2T(\frac{n}{8}) + \frac{n}{4}) + n + n = 8T(\frac{n}{8}) + n + n + n$$

$$T(n) = 2^k * T(\frac{n}{2^k}) + kn$$

Como T(1) = 1 hacemos $\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$

Reemplazando

$$T(n) = 2^{\log_2 n} T(\frac{n}{2^{\log_2 n}}) + n \log_2 n$$

$$Sabemosque: 2^{\log_2 n} = n^{\log_2 2} = n^1 = n$$

$$T(n) = nT(\frac{n}{n}) + n \log_2 n = nT(1) + n \log_2 n$$

$$T(n) = n + n \log_2 n$$

$$T(n) = \Theta(n \log n)$$

Como podemos ver el método iterativo requiere de varios cálculos y puede llegar a requerir de ciertas manipulaciones algebraicas que resultan molestas pero en general puede aplicarse prácticamente a cualquier tipo de recurrencia con resultado favorable.

El teorema maestro de las recurrencias

Mediante este teorema podemos contar con una poderosa herramienta para resolver algunas recurrencias. El teorema dice lo siguiente.

Teorema: Sea:

$T(n) = aT(\frac{n}{b}) + n^k$

Si $a \geq 1$ y $b > 1$

Entonces

- Caso 1 Si $a > b^k \Rightarrow T(n) = \in \Theta(n^{\log_b a})$
- Caso 2 Si $a = b^k \Rightarrow T(n) = \in \Theta(n^k \log n)$
- Caso 3 Si $a < b^k \Rightarrow T(n) = \in \Theta(n^k)$

Por ejemplo para $T(n) = 2T(\frac{n}{2}) + n$ tenemos que a = 2, b = 2, k = 1 luego a = b^k y estamos en el caso 2. Por lo que el algoritmo es O(n^1 log n)

Lamentablemente no todas las recurrencias tienen la forma que requiere el teorema maestro por lo que a veces no queda más remedio que aplicar el método iterativo.

Otra herramienta

Además de los tres métodos nombrados que son los básicos en cuanto a la resolución de recurrencias hay otra herramienta que pueden resultar útil al resolver recurrencias.

Cambio de variables

En algunas recurrencias puede ser conveniente realizar un cambio de variables de forma tal de resolver una recurrencia ya conocida, luego aplicando la inversa del cambio podemos obtener la solución de la recurrencia original.

Supongamos que tenemos la siguiente recurrencia:

$$T(n) = \begin{cases} 1 & \text{si n = 1} \\ T(\sqrt{n}) + 1 & \text{sino} \end{cases}$$

Sea m log n. Entonces n = 2^m. Reemplazando 2^m por n en la recurrencia de T (n) tenemos:

T (2^m) = T ($2^{m/2}$) + 1

Llamamos S(m) = T (2^m). Entonces la recurrencia queda:

S(m) = S(m/2) + 1

Usando el teorema maestro sabemos que S(m) = O(log m). Por lo tanto

T (n) = T (2^m) = S(m) = O(lgm)

T (n) = O (log log n)

Ejemplos

Ejemplo I

Sea la siguiente recurrencia:

$$T(n) = \begin{cases} 1 & \text{si n = 1} \\ 3T(\frac{n}{4}) + n & \text{sino} \end{cases}$$

De acuerdo al teorema maestro a=3, b=4, k=1. a < b^k. Por lo tanto se aplica el caso 3. O(n)

Verifiquemos el resultado del teorema maestro usando el método iterativo.

$$T(n) = 3T(\frac{n}{4}) + n$$

$$T(n) = 3(3T(\frac{n}{16}) + \frac{n}{4}) + n = 9T(\frac{n}{16}) + \frac{3}{4}n + n$$

$$T(n) = 9(3T(\frac{n}{64}) + \frac{n}{16}) + \frac{3}{4}n + n = 27T(\frac{n}{64}) + \frac{9}{16}n + \frac{3}{4}n + n$$

$$T(n) = 3^k * T(\frac{n}{4^k}) + \frac{3^{k-1}}{4^{k-1}}n + ... +$$

$$T(n) = 3^k * T(\frac{n}{4^k}) + \sum_{i=0}^{k-1}(\frac{3}{4})^i n$$

Como T(1) = 1 hacemos $\frac{n}{4^k} = 1 \Rightarrow n = 4^k \Rightarrow k = \log_4 n$

Reemplazando

$$T(n) = 3^{\log_4 n} T(\frac{n}{4^{\log_4 n}}) + \sum_{i=0}^{\log_4 n - 1} (\frac{3}{4})^i n$$

$$Sabemos que: 4^{\log_4 n} = n^{\log_4 4} = n^1 = n$$

$$y\, que: a^{\log b} = b^{\log a}$$

$$T(n) = n^{\log_4 3} + \sum_{i=0}^{\log_4 n - 1} (\frac{3}{4})^i n$$

$$T(n) = n^{\log_4 3} + n \sum_{i=0}^{\log_4 n - 1} (\frac{3}{4})^i n$$

$$Sabemos que: \sum_{i=a}^{b} x^i = \frac{x^{b+1} - x^a}{x-1}$$

$$T(n) = n^{\log_4 3} + n * \frac{(\frac{3}{4})^{\log_4 n} - (\frac{3}{4})^0}{(\frac{3}{4}) - 1}$$

$$\Rightarrow (\frac{3}{4})^{\log_4 n} = n^{\log_4 (\frac{3}{4})} = n^{\log_4 3 - \log_4 4} = n^{\log_4 3 - 1} = \frac{n^{\log_4 3}}{n}$$

$$T(n) = n^{\log_4 3} + n * (\frac{\frac{n^{\log_4 3}}{n} - 1}{-\frac{1}{4}}) = n^{\log_4 3} + n * (\frac{\frac{n^{\log_4 3} - n}{n}}{-\frac{1}{4}})$$

$$T(n) = n^{\log_4 3} - n * (\frac{4(n^{\log_4 3} - n)}{n}) = n^{\log_4 3} - 4(n^{\log_4 3} - n)$$

$$T(n) = n^{\log_4 3} - 4n^{\log_4 3} + 4n = 4n - 3n^{\log_4 3} = 4n - 3n^{0.79}$$

$$T(n) = \Theta(n)$$

Como vemos la solución final tiene un término que es O(n) y otro que es $O(n^k)$, cuando $k < 1$ ocurre que la función lineal crece más rápido que la exponencial y por eso la recurrencia pertenece a O(n).

Ejemplo II

Sea la siguiente recurrencia.

$$T(n) = \begin{cases} 1 & \text{si n = 1} \\ 2T(\frac{n}{2}) + n \log n & \text{sino} \end{cases}$$

Esta recurrencia también es bastante común en algunos algoritmos recursivos y lamentablemente no podemos escribirla en la forma en la cual vale el teorema maestro.

Como queremos averiguar la solución tenemos que aplicar el método iterativo.

$$T(n) = 2T(\frac{n}{2}) + n\log n$$

$$T(n) = 2(2T(\frac{n}{4}) + (\frac{n}{2})\log\frac{n}{2}) + n\log n = 4T(\frac{n}{4}) + n\log\frac{n}{2} + n\log n$$

$$T(n) = 4(2T(\frac{n}{8}) + (\frac{n}{4})\log\frac{n}{4}) + n\log\frac{n}{2} + n\log n = 8T(\frac{n}{8}) + n\log\frac{n}{4} + n\log\frac{n}{2} + n\log n$$

$$T(n) = 2^k * T(\frac{n}{2^k}) + n\sum_{i=0}^{k-1}\log(\frac{n}{2^i})$$

Como T(1) = 1 hacemos $\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log n$

Reemplazando

$$T(n) = 2^{\log n}T(\frac{n}{2^{\log n}}) + n\sum_{i=0}^{\log n-1}\log(\frac{n}{2^i})$$

$$T(n) = n + n(\sum_{i=0}^{\log n-1}\log n - \log 2^i) = n + n(\sum_{i=0}^{\log n-1}\log n - \sum_{i=0}^{\log n-1}\log 2^i)$$

$$T(n) = n + n(\log n * \log n - \sum_{i=0}^{\log n-1} i^{\log 2}) = n + n(\log^2 n - \sum_{i=0}^{\log n-1} i)$$

$$T(n) = n + n(\log^2 n - (\frac{(\log n - 1) * \log n}{2}) = n + n(\log^2 n - (\frac{(\log^2 n - \log n}{2})$$

$$T(n) = n + n(\frac{2\log^2 n - \log^2 n + \log n}{2}) = n + \frac{n}{2}(\log^2 n + \log n)$$

$$T(n) = n + \frac{n}{2}\log^2 n + \frac{n}{2}\log n$$

$$T(n) = \Theta(n\log^2 n)$$

En este caso el término de mayor crecimiento es n $\log^2$ n.

Resumen y aspectos clave

El propósito de ésta unidad fue introducir el estudio de los algoritmos, explicando en qué consiste el análisis y el diseño de los algoritmos. Dentro del análisis de algoritmos explicamos la necesidad de contar con un modelo computacional que permita calcular el costo de un algoritmo.

Vimos que el costo de un algoritmo se puede calcular en tiempo (cuánto tarda el algoritmo) o en espacio (cuánta memoria requiere un algoritmo). En general en el curso vamos a trabajar estudiando el costo en tiempo de los algoritmos, vigilando que el costo espacial no sea excesivo.

El diseño de algoritmos es un área que requiere de habilidad especial ya que no siempre es claro como encontrar un algoritmo eficiente para un determinado problema. Mencionamos algunos paradigmas básicos mediante los cuales podremos diseñar mejores algoritmos.

Entre los modelos computacionales revisados la máquina RAM que nos permite una escritura clara de los algoritmos pero introduce algunas complicaciones en el cálculo de costos, más adelante veremos cómo resolver estos problemas.

Hemos visto cuales son las herramientas utilizadas para analizar algoritmos iterativos, estos son los más comunes y todos estamos familiarizados con escribir este tipo de algoritmos. Para analizar el costo en tiempo de los algoritmos iterativos vimos que en realidad nos interesa conocer la tasa de crecimiento u orden del algoritmo por lo que podemos dejar de lado los factores constantes o de menor crecimiento.

Introducimos la notación O(n) que utilizaremos a lo largo de todo el curso para indicar el orden de un algoritmo. Para comparar dos algoritmos utilizaremos el orden de los mismos, cuanto menor sea la tasa de crecimiento más eficiente será el algoritmo. El considerar mejor, peor y caso medio es importante en el análisis de los algoritmos.

Se presentaron las herramientas necesarias para el estudio de algoritmos recursivos. Vimos como el costo en tiempo de un algoritmo recursivo puede expresarse mediante una recurrencia. Para resolver recurrencias explicamos tres técnicas.

El método de sustitución permite demostrar la solución de una recurrencia aplicando inducción. Esto es útil cuando ya conocemos la solución o cuando estamos en condiciones de adivinarla. El método iterativo nos permite reducir una recurrencia a una serie de sumatorias que luego resolvemos para resolver la recurrencia. El teorema maestro de las recurrencias nos proporciona una "receta" para resolver recurrencias que responden a una forma determinada.