Capitulo 10

1. Eficiencia algorítmica- Concepto
2. Factibilidad
3. Objetivos
4. Tiempo de Ejecución

1.Concepto

La eficiencia de un algoritmo se basa en la cantidad de recursos informáticos que consume:
- Espacio de memoria
- Tiempo de cómputo

Muchas veces el espacio y el tiempo requerido por un algoritmo están inversamente relacionados,
En la búsqueda de economizar estos aspectos, no se debe perder de vista lo más importante, la eficacia, o sea que produzca el resultado correcto para todos los posibles valores de entrada.

Se focaliza la optimización en el tiempo de ejecución, ya que en la tecnología actual el espacio de memoria no es en general un problema, aunque no implica que este aspecto sea insignificante, ya que en muchos dispositivos y sistemas empotrados (automóviles, electrodomésticos) la memoria asociada es limitada.

Es posible medir la eficiencia de un algoritmo, independientemente del tamaño de la entrada. Para ello existe una medida que permite comparar diferentes versiones y seleccionar la mejor.
Se consideran dos operaciones elementales: las comparaciones de valores y las asignaciones.
Analizamos dichos aspectos al calcular el mínimo de tres números a, b y c. Se presentan cuatros formas de resolverlo determinando la cantidad de operaciones del peor caso.

min := a;
If b < min then
min:= b;
If c < min then
min:= c;
 	
2 comparaciones – 3 asignaciones
	If a <= b then
	If a <= c then
	min:= a
	else
	min:= c
	else
	If b <= c then
	min:= b
	else
	min:= c;
	
	2 comparaciones – 1 asignación

If (a<=b) and (a<=c) then
min:= a;
If (b<=a) and (b<=c) then
min:= b;
If (c<=a) and (c<=b) then
min:= c;
 	
6 comparaciones – 1 asignación
	If (a<=b) and (a <=c) then
	min:=a
	else
	If b <= c then
	min:= b
	else
	min:=c;
	
	3 comparaciones – 1 asignación

En general se tiene interés en el comportamiento del algoritmo en el peor caso o en el caso promedio. En la mayoría de las aplicaciones no es importante cuan rápido puede resolver el problema frente a los datos organizados favorablemente, sino ocurre frente a datos adversos (caso peor) o en el caso estadístico (caso promedio).

Otro aspecto importante es evitar la repetición de cálculos innecesarios

R:=1/(2*x*t-1) + 1/(2*x*t-2) + 1/(2*x*t-3) + 1/(2*x*t-4)	a:= 2 * x * t; …….. R:= 1/(a-1) + 1/(a-2) + 1/(a-3) + 1/(a-4)

Si el cálculo asignado a R en el código de la izquierda está dentro de un ciclo de 1000 repeticiones, la ejecución reiterada de 2 * x * t, significa 4000 operaciones redundantes.

2. Factibilidad

Optimizar un algoritmo implica construirlo lo más correctamente posible, con estilo, transparente, sin errores y eficiente.
El intento de optimización puede provocar escribirlo incorrectamente, puede resultar más difícil de entender y más costoso su mantenimiento, como así también más propenso a incorporar errores.
Por lo tanto si se justifica se debe comenzar con un diseño no óptimo, esto en pro de la claridad y simplicidad, y a posteriori optimizarlo.

2.1. Formas de Optimización
2.1.1. Por afinación
Esto implica no modificar la estructura del algoritmo sino utilizar factores de bloque, segmentación de programas, asignación de memorias intermedias, etc.
2.1..2. Por algoritmos
La optimización por algoritmos se realiza a través de recursos como las Estructuras de datos (arreglos, pilas, colas, árboles, etc).
2.1.3. Matemáticos (por ejemplo para determinar pares e impares se utilizan los recursos: parte entera, resto, potencia de menos uno (-1), etc).
2.1.4. Parámetros (por ejemplo para determinar la longitud de los arreglos, tope de una pila, frente y final de una cola, tasa de interés, etc.).

3. Objetivos

En función de optimizar el tiempo de ejecución, los objetivos principales son que el algoritmo debe ser construido lo más pequeño posible y lo más rápido en su ejecución, o de ser viable, ambos a la vez.

- Lo más pequeño posible: significa que tenga la menor cantidad de instrucciones posibles
- Lo más rápido posible: significa economizar el tiempo de ejecución del algoritmo en máquina.

Para ilustrar la optimización por algoritmos se analizan dos ejemplos:

- determinación de tipos de triángulos (lo más pequeño posible).
- clasificación de intercambio directo o "burbuja" (lo más rápido posible)

3.1. Optimización haciendo el algoritmo más pequeño

Dado un conjunto de triplas de valores A, B, C (mayores que cero) determinar,
cuales forman triángulo, de que tipo (escaleno, isósceles, equilátero) y si son rectángulo

Se debe tener en cuenta lo siguiente:
a) la condición matemática de triángulo (un lado sea menor que la suma de los otros dos) se transforma en suficiente si el lado que se compara es el mayor de todos
b) deben compararse todos contra todos los lados para determinar el tipo de triángulo, el tener juntos los posibles lados iguales, implica una simplificación del algoritmo.
c) la verificación de triangulo rectángulo, solo se realiza para los tipos escaleno e isósceles.
e) si los lados están ordenados se toma el mayor como hipotenusa.
Por lo tanto tener los lados ordenados (ascendente o descendente) simplifica el algoritmo y lo hace más eficiente.

Los tres lados desordenados:

```
If  a< b+c then
   If  b<a+c  then
      If  c<a+b  then
         If  a=b   then
            If  b=c  then
               Writeln('equilatero')
            Else
               begin
               Writeln('isosceles');
               If  c*c=a*a+b*b then
               Writeln('rectangulo');
               End
         Else
            Begin
            If  a =c then
               begin
               Writeln('isosceles');
               If    b*b=a*a+c*c then
                  Writeln('rectangulo');
               End
            Else
               If  b=c then
                  begin
                  Writeln('isosceles');
                  If  a*a= b*b+c*c then
                     Writeln('rectangulo');
                  End
               else
                  begin
                  Writeln('escaleno');
                  If a*a=b*b+c*c then
                     Writeln('rectangulo')
                  else
                     If b*b=a*a+c*c then
                        Writeln('rectangulo')
                     else
                        If c*c=a*a+b*b then
                           Writeln('rectangulo');
                  End;
```

Los tres lados ordenados:

```
If (a< b+c)  then
  If a=b then
    If  b=c  then
       Writeln('equilatero')
    Else
       Begin
       Writeln('isosceles');
       If  a*a= b*b+c*c then
          Writeln('rectangulo')
       end
   else
      If  b=c then
         Begin
         Writeln('isosceles');
         If  a*a= b*b+c*c then
            Writeln('rectangulo')
         end
      Else
         begin
         Writeln('escaleno');
         If (a*a= b*b+c*c then
            Writeln('rectangulo');
         End;
```

3.2. Optimización haciendo el algoritmo más rápido.

A continuación se desarrollan dos versiones de uno de los métodos de clasificación más conocido: intercambio directo o burbuja:

```
Procedure Burbuja (Var V : TV; N : byte);
Var    i, j    : byte;
       Aux     : real;
Begin
 For i := 1 to N - 1 Do
     For j := 1 to N-I Do
         If V[j] > V[j+1] Then
              Begin
               Aux   := V[j];
               V[j]  := V[j+1];
               V[j+1] := Aux;
               End;
End;
```

```
Procedure BurbujeoMejorado (Var V : TV ; N :
byte);
Var
  i,K, Tope : byte;
  Aux : real;
Begin
Tope:= N ;
Repeat
   K : = 0:
   For i := 1 to  Tope -1 do
       If V[i] > V[i+1] then
           Begin
           Aux:= V[i];
           V[i]:= V[i+1];
           V[i+1]:=Aux;
           K:= i ;
           End;
    Tope:= K;
Until K <=1;
End:
```

El procedimiento Burbujeo tiene menos instrucciones, sin embargo es ineficiente pues repite incondicionalmente las acciones que contienen los ciclos for anidados. En el caso de que el vector que ingresa este total o parcialmente ordenado no se detecta dicha situación, acelerando la finalización del proceso.
El proceso Burbuja Mejorado tiene más instrucciones pero va reduciendo el espacio a ordenar en función de los cambios que realiza (y no en un elemento). Por lo tanto si el arreglo esta ordenado o parcialmente ordenado hace menos recorridas.

De todas formas la BurbujaMejorado no es un buen método, ya que MergeSort o QuickSort basados en el principio “Divide y vencerás” son más eficientes.

4.Tiempo de Ejecución:

Para estimar dicho tiempo se tienen dos enfoques:

- Análisis teórico permite obtener una estimación del tiempo de ejecución a partir de la cantidad de comparaciones y asignaciones de los diferentes algoritmos, y así poder determinar el más eficiente.

- Análisis empírico se basa en utilizar diferentes juegos de datos para medir los tiempos de respuesta. Aunque es fácil de implementar no tiene una medida de referencia, lo que puede ocasionar conclusiones erróneas si los datos empleados son favorables para el algoritmo.

Medimos el tiempo de esta función:

```
Function SumaN(n: integer):integer;
var
   j, Suma integer;
Begin
{1} Suma := 0;
{2} for j := 1 to n do
{3}       Suma := Suma + j*j*j;
{4} SumaN := Suma;
End;
```

Las declaraciones no consumen tiempo.
Las líneas {1} y {4} implican una unidad de tiempo c/u.
La línea {2} tiene un costo encubierto: inicializar (1); testear si j <=n (n+1) e incrementar j (n).
Lo que nos da: 2n+2 (1+ n + 1 + n).
La línea {3} consume 4 unidades de tiempo y se ejecuta n veces. Da un total de 4n unidades.
El costo total es 6n +4

Regla 1: Para lazos incondicionales.
El tiempo de ejecución de un lazo incondicional es, a lo sumo, el tiempo de ejecución de las sentencias que están dentro del lazo, incluyendo testeos, multiplicada por cantidad de iteraciones que se realizan.

Regla 2: Para lazos incondicionales anidados.
Se debe realizar el análisis desde adentro hacia afuera. El tiempo total de unbloque dentro de lazos anidados es el tiempo de ejecución del bloque multiplicadopor el producto de los tamaños de todos los lazos incondicionales.

Regla 3: If then else
Dado un fragmento de código de la forma

```
 If condición then
     S1
 else
     S2
```

El tiempo de ejecución no puede ser superior al tiempo del testeo mas el max (t1,t2) donde t1 es el tiempo de ejecución de S1 y t2 es el tiempo de ejecución de S2

Regla 4: Para sentencias consecutivas.
Si un fragmento de código está formado por dos bloques de uno con tiempo t1(n) y otro t2(n), el tiempo total es la suma de los mismos.