24. Eficiencia. Complejidad temporal.

24.1 Introducción.

La eficiencia de un programa es una medida de la cantidad de recursos que se requieren para producir resultados correctos.

Hace algunos años atrás, los únicos recursos que se medían y trataban de optimizar eran los de máquina. Entre ellos: memoria ocupada, tiempo del procesador y espacio de almacenamiento secundario. Para la medición se empleaban un conjunto de programas estándares, denominados bancos de pruebas (benchmarks) y se analizaba, comparativamente, cuántos recursos ocupaban.

Actualmente, también se consideran recursos humanos en la eficiencia. Esto debido a un problema de costos; a medida que pasan los años, siguen disminuyendo los costos de los equipos y aumentando los costos del personal.

24.2 Eficiencia.

Puede clasificarse en cuatro componentes básicas la eficiencia de un programa:

a) Eficiencia como usuario.
Se intenta medir la cantidad de tiempo y esfuerzo que un usuario debe gastar en aprender cómo usar el programa, cómo preparar los datos y cómo interpretar y usar los resultados.

b) Eficiencia en la mantención.
Es la cantidad de tiempo, que un programador debe emplear para leer el texto del programa y su documentación técnica, hasta lograr entenderlo, lo suficientemente bien, como para efectuarle las modificaciones que sean necesarias. Un programa que se ejecute rápido, pero que sea difícil de mantener, producirá elevados costos en salarios. Los que seguramente excederán el ahorro por costo de máquina, a medida que pasa el tiempo.

c) Complejidad algorítmica.
Tiene que ver con la eficiencia del algoritmo propiamente tal, independiente de la máquina. Por ejemplo, en largas listas es preferible la búsqueda binaria que la secuencial.

d) Eficiencia en la codificación.

Este ha sido el punto de vista tradicional de la eficiencia. Y consiste en escoger el lenguaje más adecuado a la aplicación, y dentro de él, a las mejores instrucciones del repertorio, para lograr disminuir el tiempo de proceso y el espacio necesario.

Relacionado con los puntos anteriores existen una serie de conceptos como : portabilidad, generalidad, documentación, etc., que deberían desarrollarse, para lograr una visión más completa sobre la eficiencia. Sin desconocer la importancia de esos temas, se preferirá desarrollar, con mayor detalle, el punto sobre complejidad de un algoritmo.

Cuando un programa se ejecutará muy pocas veces, o muy poco frecuentemente, lo que importa es que sea de fácil codificación, de entendimiento rápido y sencillo de probar. En caso contrario, debe considerarse que se empleen eficientemente los recursos y que su tiempo de ejecución sea lo menor posible.

24.3 Complejidad temporal.

El tiempo de ejecución de un programa depende fundamentalmente de:

- El tamaño de los datos de entrada a procesar. Suele tomarse como unidad, la entidad que es sometida al proceso básico y característico del programa. Se habla da una entrada de tamaño "n", si se tienen "n" datos de tipo unidad.

- La calidad del código objeto compilado.

- La velocidad de la máquina. Tipo de instrucciones, largo de palabra, frecuencia del reloj, etc..

- La complejidad temporal del algoritmo.

El segundo y tercer punto no pueden modificarse, salvo seleccionando un compilador más eficiente, o un computador más poderoso.

Para medir la complejidad temporal, no sirve obtener una fórmula que de un resultado en segundos; pues evidentemente existirán incorporadas en ella, constantes de proporcionalidad que dependen del compilador y de la máquina. La forma usual es describir mediante una función de "n". Sea T(n) la complejidad temporal.

La obtención de esta función, no será una tarea simple en general. Más adelante veremos algunos ejemplos.

24.4 Orden de complejidad.

Sin embargo, en raras ocasiones suelen emplearse estas fórmulas; lo que interesa es encontrar una cota superior para el crecimiento de T(n). Esta cota es el orden de complejidad.

Ejemplo.-

Si: $T(n)=(n+1)^2$ es fácil de demostrar que:

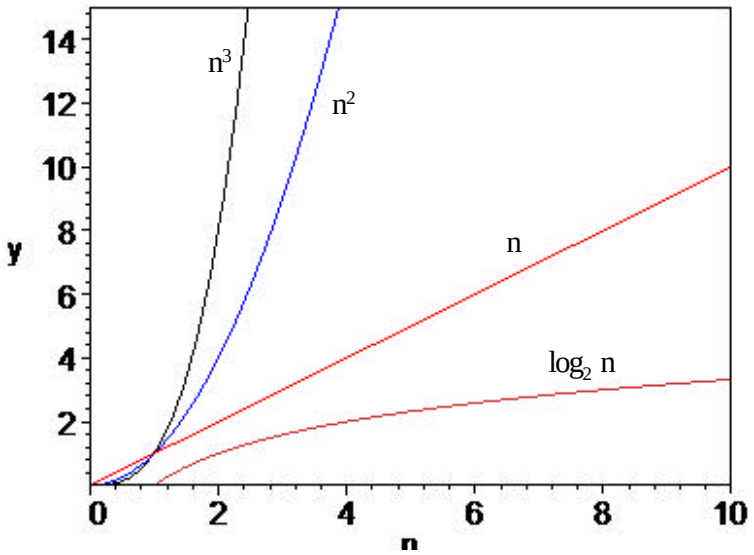
$$(n+1)^2 <= 4n^2 \text{ para } n>=1$$

Se dice que T(n) es de orden n^2.

En general se dice que T(n) es de orden n^i si existen constantes c y k tales que:

$$T(k) <= c*k^i$$

Ejemplos de órdenes de complejidad:



Puede observarse que un algoritmo de orden n^3 crece muy rápidamente. El tiempo de ejecución para uno de orden logarítmico será mucho menor que para uno de orden n^2.

En todos los casos, el orden de complejidad es una medida importante del tiempo de ejecución, sobre todo cuando n es elevado.

24.5 Cálculo de complejidad.

El siguiente ejemplo ilustra algunas de las técnicas empleadas para sumar el número de manipulaciones básicas.

Dado un arreglo de m caracteres, que constituyen una secuencia, determinar un procedimiento que verifique si existen o no sub-secuencias adyacentes iguales.

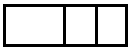
Análisis:

a) Largo par.

Si m es par, las subsecuencias a ser comparadas varían en largo desde 1 a m/2.

Una comparación de 2 secuencias de largo l, requiere de l comparaciones de caracteres.

Por ejemplo para m=4



Hay 3 comparaciones de subsecuencias de largo 1.
Hay 1 comparación de subsecuencias de largo 2.

En general, siempre habrá:

una comparación de largo m/2
tres comparaciones de largo (m/2)-1
...
m-1 comparaciones de largo 1

Entonces el número de comparaciones es:

$$N(m) = (m-1)*1 + (m-3)*2 + ... + 3*(m/2-1) + 1*(m/2)$$

$$N(m) = \sum_{i=1}^{i=m/2} (m-(2i-1))i = (m+1)\sum_{i=1}^{i=m/2} i - 2\sum_{i=1}^{i=m/2} i^2$$

Se tienen:

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

$$\sum_{i=1}^{i=n} i^2 = n(n+1)(2n+1)/6$$

que pueden verificarse por inducción.

Se logra:

N(m) = m(m+1)(m+2)/24 = (m^3+3m^2 +2m)/24

Nótese que también podría haberse sumado, en orden inverso:

$$N(m) = \sum_{i=1}^{i=m/2} (2i-1)(m/2-i+1)$$

y se obtiene igual resultado.

b) Largo impar:
En este caso, siempre habrán:

2 comparaciones de largo (m-1)/2
4 comparaciones de largo (m-3)/2
...
(m-3) comparaciones de largo 2
(m-1) comparaciones de largo 1

Entonces, el número total de comparaciones elementales es:

N(m) = (m-1)*1+(m-3)*2+ ... +4*((m-3)/2)+2*((m-1)/2)

$$N(m) = \sum_{i=1}^{i=(m-1)/2} (m-(2i-1))i$$

N(m) = (m-1)(m+1)(m+3)/24

N(m) = (m^3 +3m^2 -m-3)/24

Sumando en sentido inverso:

$$N(m) = \sum_{i=1}^{i=(m-1)/2} 2i(((m-1)/2)-i+1)$$

y se logra igual resultado.

Para un valor de m elevado, el número N varía con m^ 3.

c) En general: observamos que para una subsecuencia de largo l deben efectuarse nl comparaciones.

Con:

nl = m - (2*l-1)

También el máximo largo de una subsecuencia será:

ml = m div 2

d) Algoritmo.

Si se efectúan las comparaciones desde la cola hacia la cabeza del arreglo, puede plantearse que la expresión básica para comparar dos caracteres, de una subsecuencia de largo l es:

s[m-i-l] = s[m-i]

Donde i debe variar desde 0 hasta (l-1), para subsecuencias que contienen a s[m] como último elemento. Y desde 1 a l para subsecuencias que contienen a s[m-1] como último elemento de una de las subsecuencias.

En general i debe variar desde q hasta q+l-1 . A su vez q debe variar entre 0 y (nl-1).

Para comparar las subsecuencias de largo l, en general :

```
...
nl:=m-(2*l-1); q:=0;
repeat
  i:=q; igual:=true;
  repeat       {compara un par de subsecuencias}
    igual:=igual and (s[m-i-l]=s[m-i]);
    i:=i+1
  until (not igual) or (i=q+l);
```

```
    q:=q+1
until igual or (q=nl);
```

La variable igual se emplea para detener la repetición interna, si en una comparación de subsecuencias se encuentra, a lo menos, un caracter diferente. La repetición externa se detiene si dos subsecuencias completas de largo l son iguales. Si al segmento lo denominamos secl (secuencias de largo l), todas las subsecuencias son comparadas por:

```
...
ml:=m div 2; l:=0;
while (not igual) and (l<ml) do
begin
  l:=l+1;
  secl
end;
...
```

Si al segmento anterior lo llamamos compare; puede diseñarse la función hayigual, que detecta si existe o no, a lo menos un par de subsecuencias adyacentes iguales:

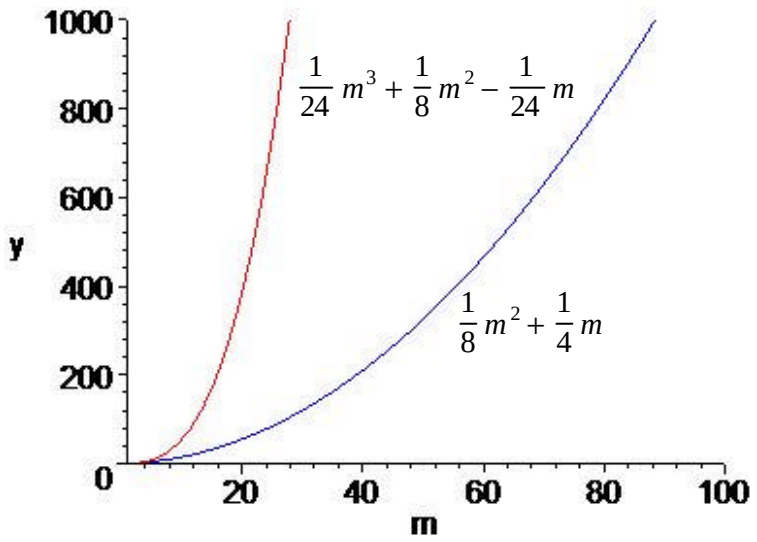
```
function hayigual(var s:secuencia; m:integer):boolean;
var    l,   {largo}
       ml,   {maximo largo}
       nl,   {numero subsec. largo l}
      i,q  : integer;
      igual: boolean;

procedure secl;
.
.

procedure compare;
.
.

begin
     igual:=false;
     compare;
     hayigual:=igual
end;
```

e) Caso particular.

Un caso particular de esta función, es determinar si existen subsecuencias adyacentes iguales, cuando una de las subsecuencias contiene al último elemento. Vemos que el lazo externo, controlado por q no será necesario; pues en este caso q es una constante con valor cero. En este caso se efectúa 1 comparación de largo 1; 1 comparación de largo 2; y hasta 1 comparación de largo m/2. El número total de comparaciones es:

$$N(m) = 1 + 2 + 3 + \ldots + m/2 = \sum_{i=1}^{i=m/2} i = (m^2 + 2m)/8$$

Con un costo menor que el algoritmo general. Se pasa de un número proporcional a m^3, a uno proporcional a m^2.



f) Observación.

Puede notarse que el cálculo de la complejidad, aún en casos sencillos, puede ser algebraicamente complicado. Este análisis se puede lograr en mejor forma si se tiene experiencia en el trabajo con sumatorias.

g) Programa de prueba.

En el texto del programa se han dejado algunos comentarios que fueron útiles en un análisis práctico del funcionamiento.

El programa principal contiene dos formas de generar los valores del arreglo de caracteres s. Se ha dejado como comentario una serie de instrucciones que permiten generar todas las posibles secuencias, en el caso que la constante n sea 3.

```
program subsec;
const  n = 6;
type   secuencia = array [1..n] of char;
var    s : secuencia;
       m : integer;
       i,x,y,z : 1..n;

function hayigual:boolean;
var    l,ml,nl,i,q : integer;
       igual       : boolean;
begin
  l:=0; ml:=m div 2; igual:=false;
  while not igual and (l<mitad) do
    begin
      l:=l+1; q:=0; nl:=m-(2*l-1); {write('l=',l)};
      repeat
        i:=q; igual:=true;
        repeat
          {write(' ',s[m-i-l],s[m-i],' ');}
          igual:=igual and (s[m-i-l]=s[m-i]);
          i:=i+1;
        until (i=q+1);
        q:=q+1;
      until q=nl
    end;
    hayigual:=igual
```

```
  end; {hayigual}

begin
 m:=n;
{prueba con todas las posibilidades para s}
{for x:=1 to n do
   begin
     s[1]:=chr(x+48);
     for y:=1 to n do
      begin
        s[2]:=chr(y+48);
        for z:=1 to n do
          begin
            s[3]:=chr(z+48);}
      {prueba con un solo valor de s}
      s[1]:='1'; s[2]:='2'; s[3]:='3';
      s[4]:='4'; s[5]:='5'; s[6]:='6';
      for i:=1 to n do write(s[i]);
      if hayigual then write('=') else write('<>'); writeln
         {end
      end
   end}
end.
```