Complejidad algorítmica

- Materia Algoritmos y Estructuras de Datos – CB100
- Cátedra Schmidt
- Complejidad algorítmica

Complejidad algorítmica

El concepto de eficiencia algorítmica hace referencia al **consumo de recursos**. Se dice que un algoritmo es más eficiente, si utiliza menos recursos que ese otro.

La eficiencia de un programa se determina sobre la base del consumo de tiempo (de ejecución) y espacio (de memoria utilizado por el algoritmo codificado durante su ejecución).

La complejidad temporal se refiere al el tiempo necesario para ejecutar un algoritmo

La complejidad espacial hace referencia a la **cantidad de espacio de memoria** (estática + dinámica) necesaria para ejecutar un algoritmo.

Habitualmente se busca una solución de compromiso considerando la **complejidad temporal y la espacial**

Los estudios sobre el consumo de tiempo pueden ser: **a posteriori**, si se mide el tiempo real de ejecución para determinados valores de entrada y en determinado procesador, o **a priori**, si de manera teórica se determina una función que indique el tiempo de ejecución para determinados valores de entrada.

Al trabajar con el cálculo del consumo de tiempo o coste temporal se hace referencia al **tamaño de la entrada del problema, o tamaño del problema**. Este tamaño depende de la naturaleza del problema, y corresponde a aquel o aquellos elementos que produzcan, al crecer, un aumento de tiempo de ejecución.

Ejemplo: en el caso del cálculo del factorial el tamaño del problema es el valor del número del cual se quiere el factorial, ya que, cuanto mayor sea este número, mayor será el tiempo consumido por la ejecución del algoritmo, pero en el caso de la búsqueda binaria, el tamaño del problema será el número de elementos que tenga el vector en el cual se realiza la búsqueda.

Tratamiento de la complejidad temporal algorítmica

El tiempo de ejecución de un programa depende, entre otros, de los siguientes factores: Los datos de entrada del algoritmo.

La calidad del código objeto generado por el compilador.

Las características de las instrucciones de máquina empleadas en la ejecución del programa.

Pero al analizar la complejidad se hace abstracción de cuestiones tales como velocidad y número de los microprocesadores, lenguaje usado, características del compilador, etc. Esto significa que decidimos ignorar las diferencias de tiempo provenientes de características como las mencionadas antes.

Esto se indica mediante el principio de invarianza.

Principio de invarianza:

Dado un algoritmo y dos máquinas M_1 y M_2, que tardan $T_1(n)$ y $T_2(n)$ respectivamente, existe una cte. real $c > 0$ y un $n_0 \in N$ tales que $\forall\, n \geq n_0$ se verifica que:

$$T_1(n) \leq cT_2(n)$$

Esto significa que **dos ejecuciones distintas del mismo algoritmo solo difieren, para nuestro cálculo e**n cuanto a eficiencia **en un factor constante** para valores de las entradas suficientemente grandes.

El número de pasos requeridos por un algoritmo para resolver un problema específico es denominado ‘running time’ o tiempo de ejecución del algoritmo y se analizará en función de la entrada del mismo. **T(n)** es la función cuyo valor es proporcional al tiempo de ejecución de un programa con una entrada de tamaño **n**.

Casos a considerar:

Para un mismo algoritmo se puede establecer tres situaciones o casos diferentes referidos al consumo temporal: el mejor caso, el peor caso y el caso promedio.

El **mejor caso** corresponde a la secuencia de sentencias del algoritmo (traza) en la cual se ejecute una secuencia de instrucciones para un determinado tamaño de la entrada que corresponda al tiempo mínimo.

El **peor caso** corresponde a la secuencia de sentencias o traza en la cual se ejecute una secuencia de instrucciones para un determinado tamaño de la entrada que corresponda al tiempo máximo.

El **caso promedio** corresponde al promedio entre todas las trazas posibles ponderadas según su probabilidad para un determinado tamaño de la entrada. Este último caso es, en general el que presenta mayores dificultades al análisis.

Aproximación práctica a la idea de complejidad temporal :

Considerar la búsqueda secuencial de un dato en un vector de n componentes. La función retorna la posición en la cual está el dato, o –1 si no está.

```
int posicion(int [] vec,int n , int dato) {
        for(int i = 0; i < sizeof(vec); i++) {
                if (vec[pos] == dato) {
                        return pos;
                }
        }
        return –1;
}
```

En este problema el tamaño de la entrada es n.

Si llamamos T(n) al coste temporal para una entrada de tamaño n, se puede observar que:
En el mejor caso, T(n) corresponde al tiempo de ejecución cuando el dato buscado está en la primera posición del array, el ciclo se ejecuta una sola vez.

En el peor caso, T(n) corresponde al tiempo de ejecución cuando el dato no está en el array, el ciclo se ejecuta n veces.
En el caso promedio, T(n) corresponde a la búsqueda promedio (el dato estará en el punto medio del array)

T(n) es la función que medirá el número de ***operaciones elementales*** según n.

En la práctica se trabaja generalmente con el peor caso. Es el más importante porque determina cual sería el tiempo de corrida del algoritmo con la peor entrada posible. Al obtener este tiempo, nos damos una idea de **cual es el tiempo máximo que puede llegar a tardar nuestro algoritmo.**

Operaciones elementales (OE):

En el cálculo de la complejidad temporal, hay operaciones a las cales se les asigna una duración de una unidad de tiempo. Son las operaciones elementales. En esos casos se verifica que el procesador las lleva a cabo en un tiempo acotado por una constante.

Consideraremos operaciones elementales (OE) a las operaciones aritméticas básicas , comparaciones lógicas, transferencias de control, asignaciones a variables de tipos básicos, etc.

Como en particular nos interesará determinar el tiempo correspondiente al “peor caso”, es decir, establecer cuál es el tiempo máximo que un algoritmo puede necesitar para ser ejecutado, tendremos en cuenta lo siguiente:

1) En una secuencia de sentencias, se suma la cantidad de OE correspondientes a cada sentencia para obtener el número total de OE.

Ejemplo 1:

```
int a, b=5;   //1 O.E.
a=b+1;        //2 O.E.
a++;          //2 O.E.

Total:  T(n) = 5 O.E.
```

En este caso observamos, además, que el número de OE del mejor caso coincide con el número de OE del peor caso.

2) En el caso de que haya bucles se debe sumar el número de OE del cuerpo del bucle con las OE del análisis de la condición, multiplicado por el número de veces que se lleve a cabo la misma, más las OE correspondientes a la condición de salida del bucle.

Ejemplo 2:

```
int  n,  i=0;  //1  O.E.
cin>>n;    //1    O.E.
while (i<n)
     {
      a=a*2;    //2 O.E.
     }
 //el bloque de 2 O.E. se ejecuta n veces
```

T(n) = 2 O.E. +(n+1) O.E. +2*n O.E. = 3n +3 O.E.

3) En el caso de encontrarnos con una bifurcación, para el cálculo del peor caso debemos considerar cuántas OE le corresponden a cada rama de la bifurcación, para así determinar cuál es la “peor rama”.

Entonces, el número de OE para el peor caso en una bifurcación se calcula se sumando el número de OE de la condición de la bifurcación con el número de OE de ‘la peor rama’, es decir aquella rama de la bifurcación que contenga más OE.

Análogamente si se trata de una sentencia de opción múltiple (switch)

Ejemplo 3:

```
int n, b=2,i; //1 O.E.
cin>>n;     //1 O.E.
if (n%5==0) // 2 O.E.
   {
    n=n+3;
    b=n+1;
    }
else
  {
     for(i=0; i<n;i++)
         {
            b=b+1
         }
   }

// la rama ‘true’ de la bifurcación tiene un coste de 4 OE
//la rama ‘false’ tiene 5 n + 2 operaciones elementales
```

Observamos que en el peor caso T (n) = 5 n +6 OE

Ejercicios

Ejercicio a)

```
int a, int b= 0;
a= b+1;
```

Ejercicio b)

```
int h = 0;
h++;
```

Ejercicio c)

```
int i, a = 0;
for (i=0;i<10;i++) {
	a=a+i;
}
```

Ejercicio d)

```
int i,k = 0;
i=2;
if (i >5) {
	k=i+30;
} else {
	k=2;
}
```

Medidas Asintóticas:

Como podemos intuir, el cálculo de la expresión del número de operaciones elementales puede ser muy engorroso.

Ahora bien, daremos un paso más en cuanto a los elementos de los cuales hacemos abstracción.

Si consideráramos dos algoritmos A1 y A2, cuyos tiempos fueran (para el peor caso), respectivamente:

TA1(N) = 3 N2 + 5N +6
TA2(N) = 12 N2+ 1

Podríamos acordar que el tiempo es similar ¿por qué? Porque en ambos casos la forma de la función que expresa el modo de crecer del número de OA (o *tiempo requerido para la ejecución*) es una cuadrática.

Esta idea se generaliza en el concepto de Medidas Asintóticas. Partimos de que, en general, no interesa calcular el número exacto de OE de un algoritmo, sino **acotarlo apropiadamente**.
La cota *superior,* por ejemplo, será una función no superada por aquella que expresa el número de OA, a partir de cierto valor de N.
Por ejemplo, para el caso de las dos funciones TA1(N) y TA2(N) mostradas antes, ambas son acotadas, por ejemplo, por f1(N)=13N2, o bien por f2(N)=20N2, o por fi(N)=5N3, o bien por….hay infinitas funciones que acotan a ambas superiormente, pero en particular algunas nos interesan por estar “más cerca” de FA1(N) y FA2(N), como las cuadráticas.

De este modo se puede indicar que el coste temporal, al aumentar el tamaño del problema no superará dicha función. Suele indicarse que esa cota establece el orden de complejidad temporal, o coste temporal.

El orden (logarítmico, lineal, cuadrático, exponencial, etc. según las características de la función que acota a la que expresa el número de OE del algoritmo) de la función **T(n),** mide la **complejidad temporal** de un algoritmo, y **expresa el comportamiento dominante cuando el tamaño de la entrada es grande.**

Para determinarlo, se analizará el comportamiento asintótico de la función, para valores de la entrada suficientemente grandes.

Medidas del comportamiento asintótico de la complejidad:

Cuando analizamos un algoritmo con un tamaño de entrada N, puede interesarnos estudiar el peor tiempo, el mejor tiempo o el tiempo promedio, como ya habíamos señalado antes.

Para cualquiera de los tres casos, puede interesarnos:

- Encontrar una función que, multiplicada por una constante, acote ese tiempo “por arriba” o superiormente. Como ya podemos inferir, no será única la función que acote superiormente a otra.

- Encontrar una función que, multiplicada por una constante, acote ese tiempo “por abajo” o inferiormente. Tampoco será única la función que acote inferiormente a otra.

- Encontrar una función que, multiplicada por dos constantes distintas permita acotar el tiempo tanto superior como inferiormente.

Las *medidas asintóticas* definen las características de funciones que verifican lo que nos interesa. Estas son:

O (omicron mayúscula) Ω (omega mayúscula) Θ (theta mayúscula)

En principio puede decirse, de modo muy simplificado, que

O hace referencia a una función que acote superiormente el T(n) , es decir indica **qué función no será superada por T(n) para n suficientemente grande.**

Ω hace referencia a una función que acote inferiormente el T(n) , es decir indica que función **será siempre superada por T(n) para n suficientemente grande**.

Θ hace referencia a una función que acote tanto inferiormente como superiormente el T(n) , es decir indica **qué función acotará superior e inferiormente a T(n) para n suficientemente grande.**

O, Ω y Θ se pueden calcular para el mejor caso, para el peor caso y para el caso promedio, pero **la medida asintótica que más nos interesa en este curso es O para el peor caso.**

Concepto de O Grande (también llamado Big O, Big Omicron, Big Oh)

Definición:

f(n) es de (o pertenece a) O(g) si y solo si existen constantes positivas c y n_0 , tales que se verifica, para todo $n>n_0$

$$0 \leq f(n) \leq c * g(n) \text{ para todo } n \geq n_0$$

(los valores de c y n_0 no dependen de n, sino de f)

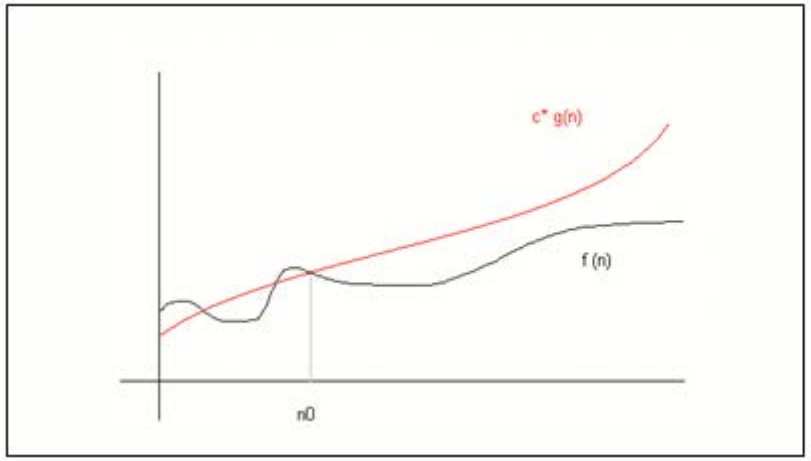
Observar que O(g(n)) es un conjunto (set) de funciones. Cuando se dice que f(n) es de O(g(n)) se está indicando que f(n) ***pertenece*** al conjunto O(g(n)).

$$O(g(n)) = \{f: \mathbf{N}+ \rightarrow \mathbf{R}+: (\text{existen } c \in R^+, n_0 \in N^+ (\text{para todo } n > n_0: f(n) <= c * g(n))) \}$$

f es de O(g) significa que: f no 'crece más rápido' que g.
f crece 'más lentamente, o a lo sumo tan rápido' que g. g
no crece 'más lentamente' que f.

Nota: A menudo se expresa f(n) = O(g(n)); esta forma de indicar la complejidad, es bastante frecuente, y es un abuso de notación, ya que no se trata de una verdadera igualdad (la igualdad enunciada arriba debe ser tratada como 'unidireccional').

Gráficamente:



f(n) es O(g(n))

Se dice que g acota superiormente a f si existe una constante real positiva c tal que para n mayor que cierto n0 (es decir, cuando n->∞), se verifica f(n)<= c * g(n).

Es por eso que en castellano se dice ‘f es del orden de g’, o ‘f pertenece a O(g)’ cuando f es O(g). Si f(n) es O(g(n)), se dice que g expresa el orden de complejidad del algoritmo, o el orden de f. La notación O es una relación reflexiva, **no simétrica** y transitiva.

Ejemplo:

Son de orden O(N2): N2 , 17 x N2 , 8 x N2 + 17 x N + 3.. y también 6 x N1.5 , 2 x N+1, etc

También 6 N1.5 es también O(N1.5)

Luego: 2 N+1 es también O(N) y también O(N1.5)

Como se ve, **los O están relacionados inclusivamente**. **Pero para cada función se elegirá el O más apropiado**, es decir el "**más cercano**". Así,

para 8 N2 + 17 N + 3 se indica O(N2)

para 6 N1.5 se indica O(N1.5)

para 2 N+1 se indica O(N)

Ejemplo

Si T(n) se expresa como 3 n3 + 2 n + 4, entonces T(n) es O(n3) puesto que se puede calcular una constante k que, multiplicada por n3 supere a f para n mayor que un cierto n0.

En este caso, basta con considerar como constante a 4.
3n3 + 2n + 4 < 4 n3

En los casos en que T(n) se exprese como polinomio, el término de mayor grado corresponderá al O de la función en cuestión.

Así, si T(n) =100 n6 + 2n3 +5 es O(n6)

Propiedades de Big O

1. f es O(f) (significa que f esta acotada por su orden)
2. O(f) es O(g) => O(f) está incluido en O(g)
3. O(f) = O(g) sii f es O(g) y g es O(f)
4. Si f es O(g) y g es O(h) =>f es O (h)
5. Si f es O(g) y f es O(h) =>f es O(min (g,h))
6. Regla de la suma: Si f_1 es O(g) y f_2 es O(h) => $f_1 + f_2$ es O(max (g, h))
7. Regla del producto: Si f_1 es O(g) y f_2 es O(h) => $f_1 * f_2$ es O(g * h)
8. Si existe $\lim_{n->\infty} (f(n)) / (g(n)) = k$, se verifica:
    - Si k<> 0 y k finito, O(f) = O(g)
    - Si k=0 , f es O(g), pero g no es O(f)

Los O se pueden ordenar de forma creciente, lo cual nos permite comparar la eficiencia temporal de los algoritmos mediante su O.

En este sentido, se verifica:

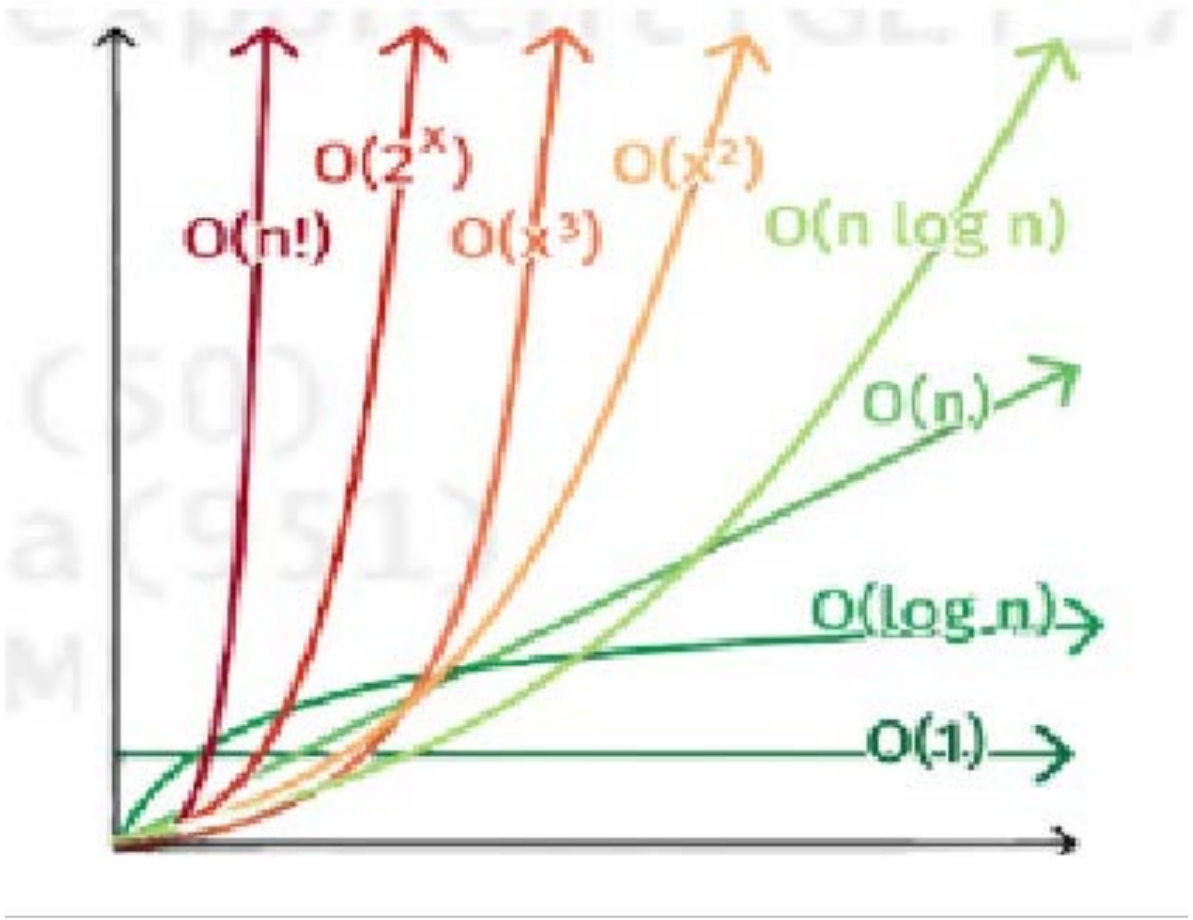
Cualquier función exponencial de n domina sobre otra función polinomial de n. Cualquier función polinomial de n domina sobre una función logarítmica de n. Cualquier función de n domina sobre un término constante.

Un polinomio de grado k domina sobre un polinomio de grado l sii k>l Resumiendo:

Los comportamientos asintóticos para O de más frecuente aparición se pueden ordenar de menor a mayor según su crecimiento de la siguiente forma:

$O(1) \subset O(\log N) \subset O(N) \subset O(N * \log N) \subset O(N^2) \subset O(N^3) \subset ... \subset O(2^N) \subset O(N!)$.

Ordenes de complejidad algoritmos



¿Cómo calcular O para el peor caso de forma sencilla y rápida?

Tener en cuenta:

1) Las O.E. son O(1)

2) En secuencias se suman las complejidades individuales **aplicando la regla de la suma**. Por ejemplo, una secuencia finita de bloques de O(1) es O(1)

3) Bucles en los cuales el número de iteraciones es fijo: es el O del bloque ejecutado Ejemplo:

```
int i, a, b=6, c=1;    //O(1)
for (i=0; i<1000; i++) {
      a=a*b+c-5   //O(1)
}
```

En este ciclo, 1000 veces, que es O(1), se ejecuta algo de O(1), por lo tanto este código es O(1)*O(1) = O(1) **por aplicación de la regla del producto.**

4) Bucles en los cuales el número de iteraciones corresponde al tamaño n del problema : es el O del bloque multiplicado por n

```
int i, a, b=6, c=1, n;    //O(1)
cin>>n;                    // O(1)
for (i=0; i<n; i++) {
     a=a*b+c-5   //O(1)
}
```

En este ciclo, n veces se ejecuta algo de O(1), por lo tanto este código, **por aplicación de la regla del producto** es n veces O(1), o bien O(n) * O(1) = O(n)

5) Bucles anidados: multiplicar los O correspondientes a cada anidamiento.

Ejemplo:

```
int i, j,n, a, b=2; //O(1)
cin>>n,             //O(1)
for (i=0; i<n;i++) {
  for (j=2; j>n;j++){
     a=a+b,
     b++;
  }
}
```

El bloque interior que es O(1), depende de un par de bucles anidados. Ambos iteran dependiendo de n, por lo tanto es O(n)*O(n)*O(1) = O(n2) **por aplicación de la regla del producto**.

6) En bifurcaciones se suma el O correspondiente al análisis de la condición más el de la peor rama.

Ejercicios:

1)

a)
```
int i=3;
int a,b;
b=c*a+5;
```

b)
```
int i, a=1, k=100;
cin>>n;
for(i=0;i<k;i++)
a=a+i;
```

c)
```
int i, a=1;
cin>>n;
for(i=0;i<n;i++)
a=a+i;
```

d)
```
 int i, j, a=1;
cin>>n;
for(i=0;i<n;i++)
for(j=2;j<n;j++)
    a=a+i;
```

e)
```
int i, a=1;
cin>>n;
for(i=0;i<n;i++)
a=n+i;
```

f)
```
int i=1, n;
cin>>n;
while (i<n)
i=i*2;
```

g)
```
int i=1, n;
cin>>n;
while (i<100)
i=i*2;
```

2) Analizar y determinar el coste temporal, en términos de O, para el peor caso de cada uno de estos algoritmos planteados de forma iterativa:

   a) Búsqueda secuencial en un array no ordenado de N elementos

   b) Ordenamiento por selección de un array de tamaño N (y pensar también en el O del mejor caso ¿cuál es el mejor caso?)

   c) Ordenamiento por burbujeo de un array de tamaño N (y pensar también en el O del mejor caso ¿cuál es el mejor caso?)

   d) Ordenamiento por inserción de un array de tamaño N (y pensar también en el O del mejor caso ¿cuál es el mejor caso?)

   e) Cálculo del factorial de un número X

3) Intuitivamente, determinar el O del peor caso de cada una de estas operaciones sobre un array:

   a) Alta final de los elementos que se hayan incorporado previamente, conociendo la posición del último elemento.
   b) Alta en posición 0, realizando un corrimiento de los elementos que hubiera previamente, una posición "hacia la derecha" (comenzando desde el último)

4) Considere diversas implementaciones del TDA Pila. Describa las mismas e indique coste de push y del pop, para el peor caso.

5) Idem 4 para un TDA Cola.

Tratamiento del análisis de complejidad en los algoritmos recursivos:

En el caso de algoritmos recursivos, el cálculo de O da origen a expresiones de recurrencia, que tienen diversas formas de resolución.

Básicamente, esos modos de resolución son tres:

a) Usando el método de expansión. En este curso mostraremos cómo puede aplicarse este método apelando también a la intuición en algunos pasos.

b) Usando los métodos de resolución de ecuaciones de recurrencia que nos provee el Análisis Numérico (excede los objetivos de este curso).

c) Usando algún teorema cuando se den las condiciones para su aplicación.

Consideremos algunos casos de algoritmos recursivos:

1) Análisis de la Búsqueda Binaria recursiva:

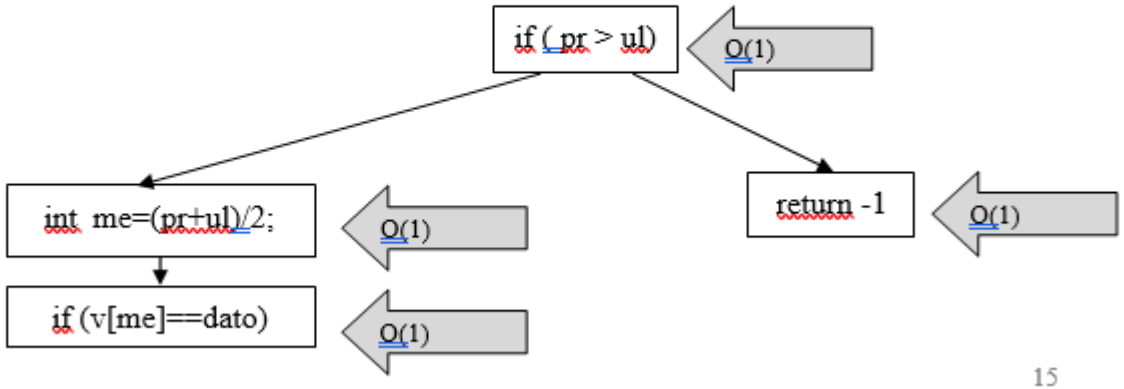
Como ya sabemos, el algoritmo codificado indica:

//retorna el subíndice de la posición en donde está el dato, o bien -1 si no está

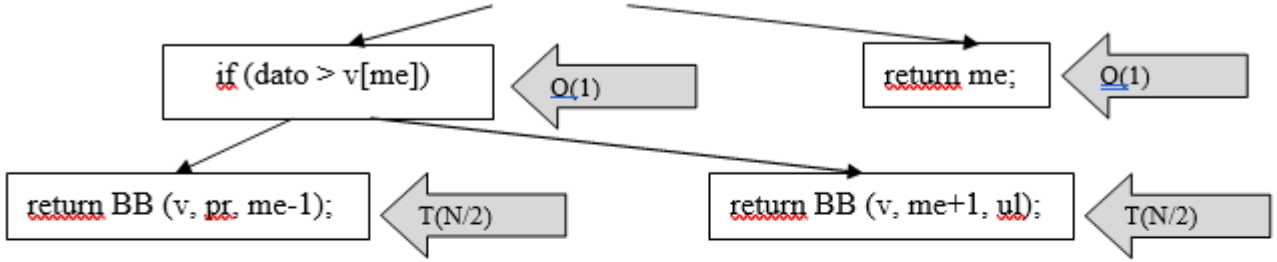
```
int BB (int v[], int pr, int tul, int dato) {
  if ( pr > ul)
  return -1;
  else
      {
        int  me=(pr+ul)/2;
        if (v[me]==dato) return me;
         else
           if (dato > v[me]) return BB (v, me+1, ul);
             else return BB (v, pr, me-1);
       }
}
```

Analicemos paso a paso lo que indica este código:

Para ello hemos realizado un diagrama que permite ver más claramente la estructura del código:



15



Como vemos, las dos peores ramas tienen un coste de

T(N)= O(1)+O(1)+ O(1)+O(1)+T(N/2), es decir,

T(N)= O(1) + T(N/2)

Como O(1) significa que el tiempo es acotado por una función constante, por ejemplo por f(N)=1, puede , y es conveniente para el cálculo, expresarlo como:

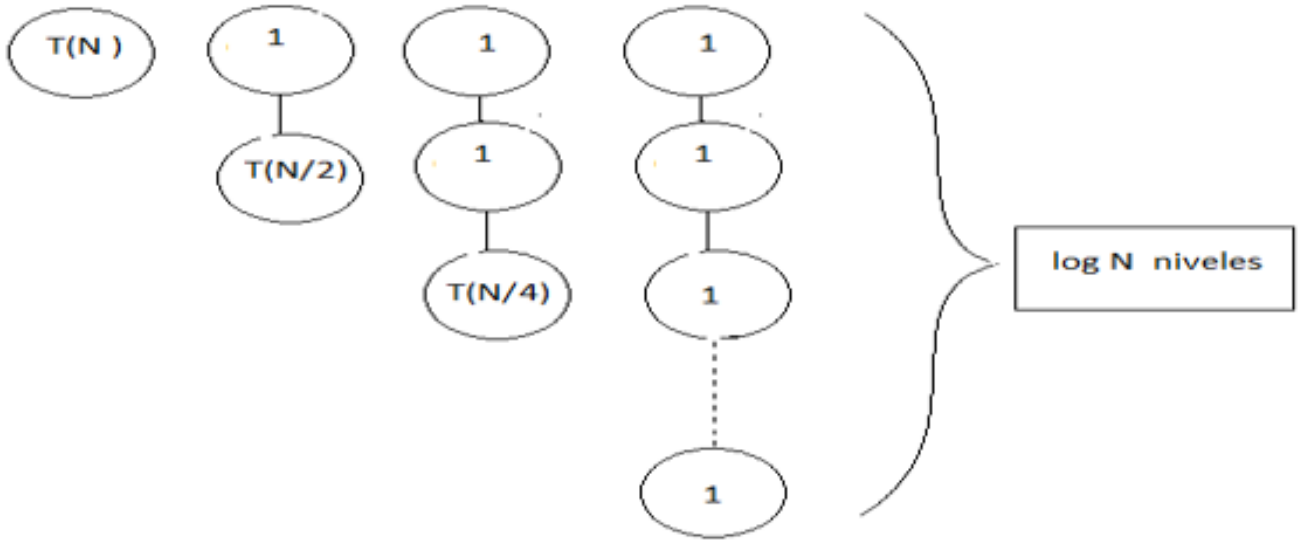
T(N) = T(N/2) +1

Esta ecuación de recurrencia expresa el coste temporal del algoritmo recursivo de búsqueda binaria en un array.

Sabemos además que se verifica O(0)=1 (que significa que si el tamaño del array es 0, el coste es 1, que es el coste del return -1), y que O(1)=1 (si el array tiene tamaño 1, también el coste de determinar si el dato está o no es 1).

Vamos a graficar el "árbol de llamadas recursivas" correspondiente a la ecuación.
Un árbol de llamadas grafica lo que sucede al realizar invocaciones a funciones. Eventualmente puede usarse para mostrar lo que pasa cuando se trabaja con funciones que se llaman a si mismas, es decir recursivas.

Considerando la situación de invocación inicial, y llamados posteriores, podemos graficar esta evolución del árbol de llamadas:

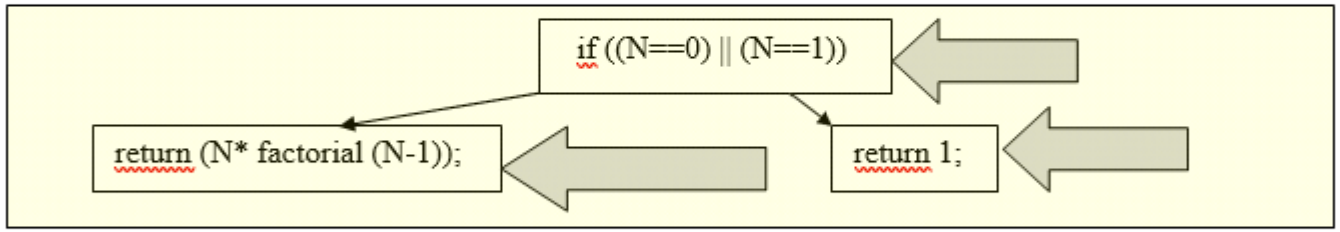


Entonces: sabemos que cada nodo del árbol de llamadas debe ser ejecutado y tiene asociado un tiempo de ejecución, que en el gráfico anterior está indicado dentro del nodo mismo. El tiempo de ejecución del algoritmo será la sumatoria de los tiempos de los nodos. Tenemos nodos de coste 1, y hay una cantidad de ellos que se puede contabilizar como $\log_2 N$.
Entonces $T(N) \in O(\log N)$ (¿habían leído que la búsqueda binaria tenía "coste logarítmico"?, acá tenemos la explicación)

2) Análisis del algoritmo recursivo de cálculo de factorial:

```
int factorial (int N)
{
 if ((N==0) || (N==1)) return 1;
  else return (N* factorial (N-1));
}
```

Haciendo un gráfico como en el caso anterior, es

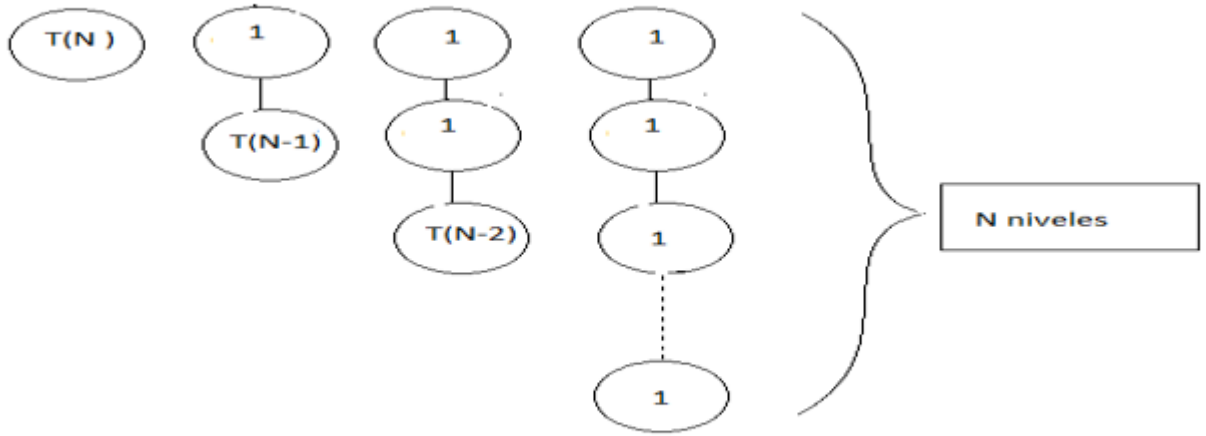


Observar que el return de la izquierda realiza una operación más, de coste O(1), luego de recibir el resultado de la invocación recursiva, que es la multiplicación por N.

Entonces, el tiempo se puede expresar como:

$T(N) = T(N-1) + 1$ y $T(1) = T(0) = 1$

Si realizamos, como en el ejemplo anterior, el árbol de llamadas, queda:



Tenemos N niveles de coste 1 cada uno. La sumatoria de los costes indica que
T (N) pertenece a O(N).

Otra forma: planteando el método de expansión podemos hacer este desarrollo:

$T(N) = T(N-1) + 1$

$T(N-1) = T(N-2) + 1$

⇨ $T(N) = T(N-2) + 2$

$T(N-2) = T(N-3) + 1$

⇨ $T(N) = T(N-3) + 3$

En el iésimo paso, es

$T(N) = T(N-i) + i$ (forma general)

Tomando i=N, es

$T(N) = T(0) + N$ y como $T(0) = 1$,
resulta $T(N)$ pertenece $O(N)$

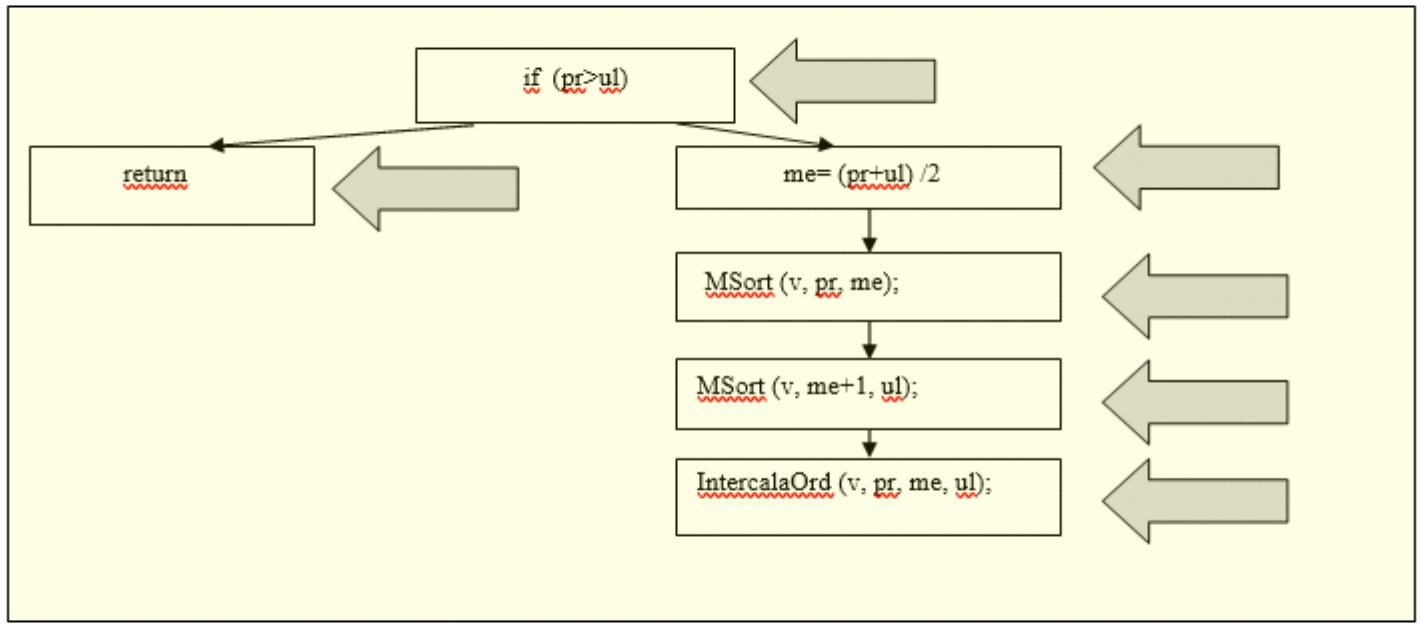
3) Análisisis del Merge Sort (ordenamiento por mezcla)

El algoritmo del del merge sort se puede codificar de este modo:

```
void IntercalaOrd (int v[], int i, int m, int f) {
        int *aux = new int[m-i+1];
        for(int j=i; j<=m; j++)
                aux[j-i] = v[j];
        int c1=0, c2=m+1;
        for(int j=i; j<=f; j=j+1) {
                if(aux[c1] < v[c2]) {
                        v[j] = aux[c1];
                          c1=c1+1;
                        if(c1==m-i+1)
                                for(int k=c2; k<=f; k++)
                                        {
                                          j=j+1;
                                      v[j] = v[k];

                                        }
         }
         else {
                  v[j] = v[c2];
                    c2=c2+1;
                  if(c2==f+1)
                          for(int k=c1; k<=m-i; k=k+1)
                                  {
                                    J=j+1;
                                 v[j] = aux[k];
                                  }
         }
}
```

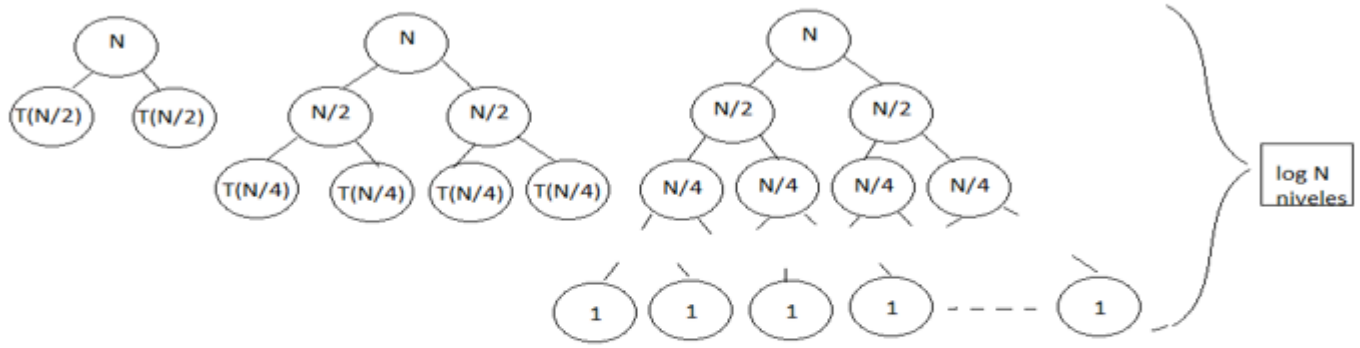
En el análisis de este algoritmo tenemos que considerar que el tiempo de la intercalación de dos arrays ordenados que tienen entre ambos N elementos, pertenece a O(N). Podemos intuirlo si pensamos que el algoritmo de intercalación ordenada realiza una única pasada por ambos arrays, para, mientras haya elementos en ambos que no hayan sido trasladados al tercer array, comparar un elemento de uno de ellos con uno del otro, y, cuando en uno de los dos dos arrays se hayan agotado los elementos, trasladar los que hayan quedado en el otro array al tercero. Se muestra un gráfico de análisis del algoritmo:



T(N) para el peor caso, se puede expresar como:

$T(N) = 2*T(N/2) + N$ y $T(0) = T(1) = 1$

Entonces, si hacemos como en los casos anteriores el árbol de llamadas, es:



Ahora bien, podemos observar que en cada nivel, la suma de los costes de los nodos de ese nivel es N.
Por lo tanto, tenemos log N niveles, con un coste total por nivel de N.
Es decir,

$T(N) \in O(N * \log N)$

Ahora bien, podemos observar que en cada nivel, la suma de los costes de los nodos de ese nivel es N.
Por lo tanto, tenemos log N niveles, con un coste total por nivel de N.
Es decir,

$T(N) \in O(N * \log N)$

Si lo resolvemos planteando el método de expansión, es:

$T(N) = 2\,T(n/2) + N$

$T(N/2) = 2\,T(N/4) + N/2$
⇨ $T(N) = 2(2T(N/4) + N/2) + N$
⇨ $T(N) = 4\,T(N/4) + 2N$

$T(N/4) = 2T(N/8) + N/4$
⇨ $T(N) = 4(2T(N/8) + N/4) + 2N$
⇨ $T(N) = 8T(N/8) + 3N$

En el iésimo paso

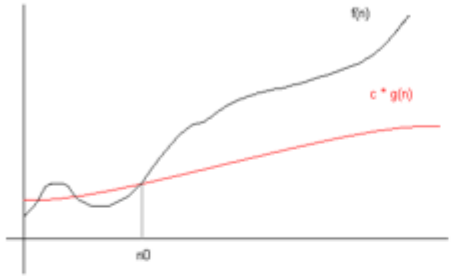
$T(N) = 2^i\,T(N/2^i) + i * N$

Tomando $i = \log_2 N$ (este reemplazo amerita una discusión…)

$T(N) = N\,T(1) + (\log_2 N) * N$
Resulta T(N) pertenece a O(N * log N)

Otras medidas asintóticas

Concepto de Ω grande (Big Ω, Big Omega, Omega Grande)



f(n) es Ω (g(n))

Definición:

f(n) es Ω (g(n)) si y solo si existen constantes positivas c y n_0, tales que se verifica, para todo $x>n_0$
$0 \leq c * g(n) \leq f(n)$ for all $n \geq k$. (los valores de c y n_0 no dependen de n, sino de f)

$\Omega (g(n)) = \{f: \mathbf{N}^+ \rightarrow \mathbf{R}^+ ;$ (existen $c \in R^+$, $n_0 \in N^+$ (para todo $n > n_0 : c * g(n) <= f(n)$)) }

Es el conjunto de funciones que 'acotan por abajo'.

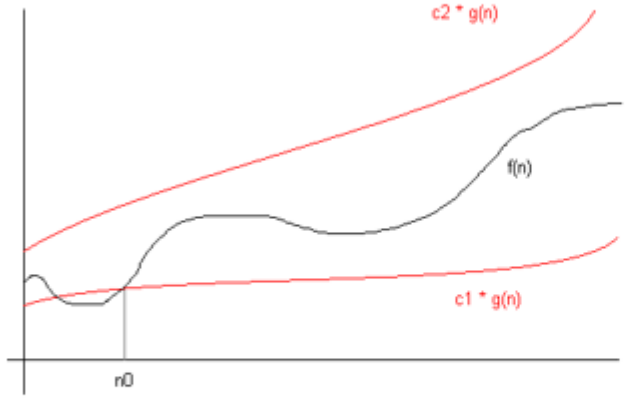
Ejemplo:
Son $\Omega (N^2)$: N^2, $17 * N^2$, $N^2 + 17 * N$, N^3, $100 * N^5$, . . .

Observar que f(n) es Ω (g(n)) sii g(n) es O(f(n))

En textos en inglés, suele decirse que la función f es ***'lowly bounded'*** por g

La expresión f(n) = (g(n)) se acepta como abuso de notación, debería escribirse como f(n) es (o pertenece a) Ω(g(n)).

Concepto de Θ Grande (Big Θ , Big Theta)



f(n) es Θ (g(n))

Cuando se acota la complejidad de un algoritmo tanto superior como inferiormente por la misma función se usa la notación Theta.

En estos casos se dice que f(n) está acotado por "arriba" y por "abajo", para *n* suficientemente grande.

Θ(g) es el conjunto de funciones que crecen exactamente al mismo ritmo que g.

f pertenece a Θ(g) si g *es a la vez cota superior e inferior* de f.

f(n) es (o pertenece a) Θ (g(n)) significa que existen constantes reales positivas c_1, c_2, y k, tales que se verifica
$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ para todo $n \geq n_0$.

Ejemplo:
Son $\Theta(x^2)$: x^2 , $3x^2 + 2x$, , $54\,x^2 + 20\,x + 90$, ...

Se dice que g y f poseen el mismo orden de crecimiento.
Si f es Θ(g) g es el **orden exacto** de f. Se verifica $\Theta(f) = O(f) \cap \Omega(f)$

Si f(n) es Θ (g(n)) se dice que f **es de orden (de magnitud) g**, o que f es de (o tiene) **coste** (o complejidad) g.

Teorema maestro

Teorema maestro (a veces llamado T.M. de reducción por división):

Esta propiedad suele aplicarse a algoritmos que provienen de aplicar la estrategia “Divide y Vencerás”.

No la demostraremos en este curso, pero podemos mostrar que lo que indica para cada caso es válido, haciendo un análisis de la propiedad.

El teorema maestro (por sustracción) es una forma de recurrencia que se da en algoritmos donde se reduce el tamaño del problema en una cantidad constante, en lugar de dividirlo. La forma general de la recurrencia es:

$$T(n) = a \cdot T(n-b) + O(n^k)$$

o

$$T(n) = a \cdot T(n-b) + O(n^k)$$

Donde:

- T(n) es el tiempo de ejecución para un problema de tamaño n.
- a es el número de subproblemas en los que se reduce.
- T(n–b) representa el costo de resolver un subproblema de tamaño n–b, donde b es una constante.
- O(n^k) es el costo de resolver el problema actual (en términos de n) fuera de la llamada recursiva.

Casos del teorema maestro por sustracción:

Este tipo de recurrencia es útil en algoritmos donde el problema se reduce por una cantidad **constante** en cada paso, como algunos algoritmos de búsqueda o eliminación (por ejemplo, búsqueda secuencial, problemas de escalera, etc.).

Para resolver este tipo de recurrencias, hay una técnica diferente a la del teorema maestro de **divide y vencerás**. Aquí se pueden aplicar las siguientes reglas, dependiendo de los valores de a y k:

1. Si $a > 1$:

   La recurrencia crece exponencialmente y la solución tiene la forma:

   $$T(n) = O(a^{n/b} \cdot n^k)$$

2. Si $a = 1$:

   El tamaño del problema disminuye linealmente en cada paso, y la solución es:

   $$T(n) = O(n^{k+1})$$

3. Si $a < 1$:

   En este caso, el problema se resuelve rápidamente, y la solución es:

   $$T(n) = O(n^k)$$

Ejemplo 1

Ejemplo:

Supongamos la recurrencia:

$$T(n) = T(n-1) + O(n)$$

Aquí, $a = 1$, $b = 1$ y $k = 1$. Esto corresponde a la reducción de tamaño en 1 en cada paso, con un costo lineal en cada iteración. Por lo tanto, utilizando la segunda regla (donde $a = 1$), la solución será:

$$T(n) = O(n^2)$$

Este tipo de recurrencia es común en algoritmos donde la reducción del problema ocurre en pasos constantes, como en el cálculo de problemas de tipo escalera, algoritmos de fuerza bruta, o problemas de búsqueda secuencial.

Ejemplo 2

$$T(n) = 2T(n-1) + n$$

La fórmula que mencionas se refiere a una solución más detallada cuando el teorema maestro por sustracción se aplica en un contexto particular. La relación general es:

$$T(n) = O\left(\frac{a}{b} \cdot n^k\right)$$

Ahora, aplicamos esto a nuestra ecuación:

$$T(n) = 2T(n-1) + n$$

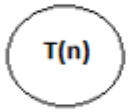
Sabemos que $a = 2$, $b = 1$, y $k = 1$. Para este caso, la solución es del tipo **exponencial** porque $a > 1$. Sin embargo, el comportamiento exacto de la complejidad también toma en cuenta el término no recursivo n^k.

La complejidad final de la recurrencia $T(n) = 2T(n-1) + n$, aplicando el teorema maestro por sustracción, es:

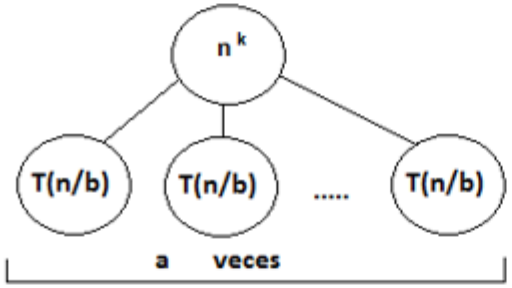
$$T(n) = O(n \cdot 2^n)$$

Análisis de lo que dice el TM de reducción por división:

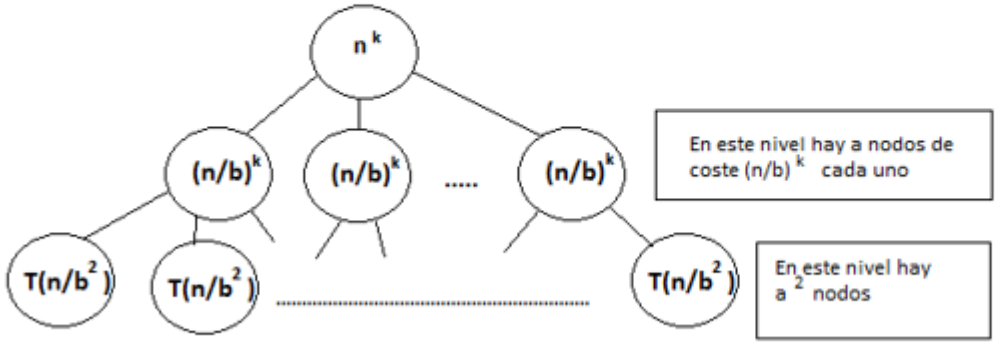
Consideremos los árboles de llamadas correspondientes al enunciado del TM:



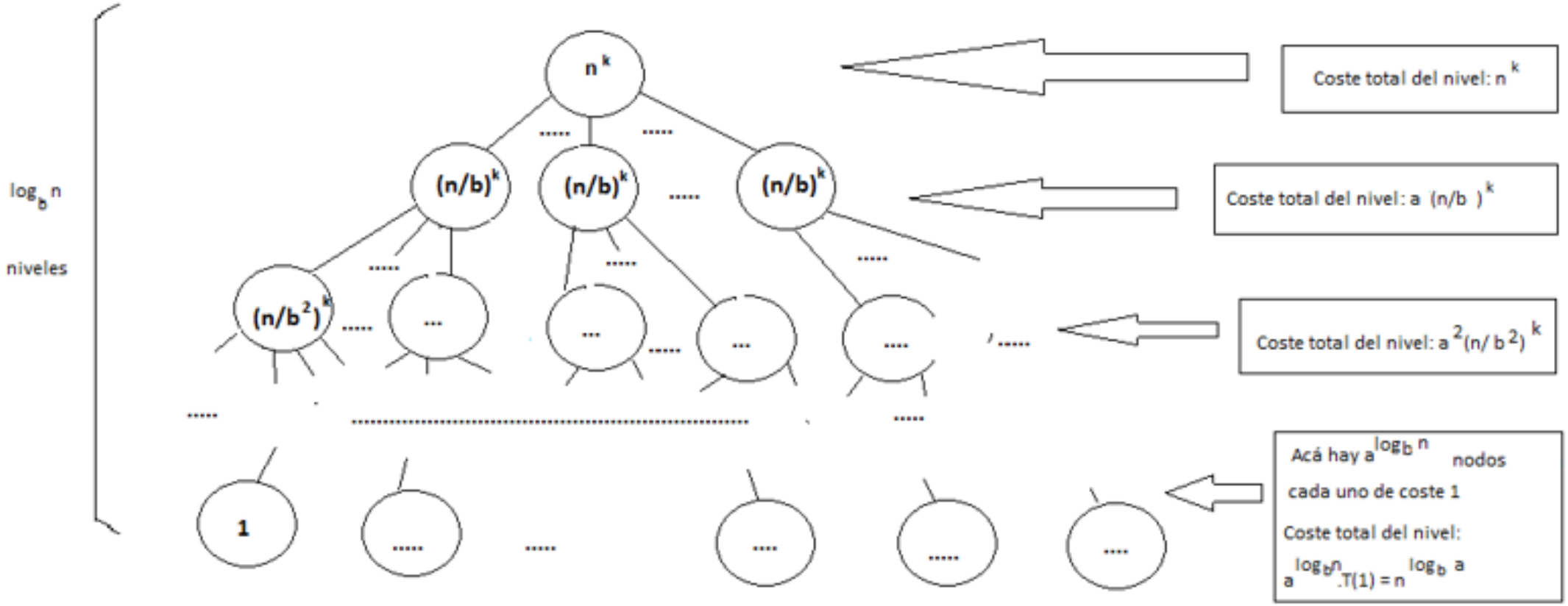
Paso 1



Paso 2



Paso 3


$\log_b n$
niveles
n^k
$(n/b)^k$
$(n/b)^k$
$(n/b)^k$
$(n/b^2)^k$
1
Coste total del nivel: n^k
Coste total del nivel: $a\ (n/b\ )^k$
Coste total del nivel: $a^2(n/b^2)^k$
Acá hay $a^{\log_b n}$ nodos
cada uno de coste 1
Coste total del nivel:
$a^{\log_b n}.T(1) = n^{\log_b a}$

Consideremos los siguientes casos:

1. $a < b^k$: entonces el tiempo de ejecución disminuye en cada nivel, predomina el tiempo de la raíz.
   El TM indica $O(n^k)$

2. $a = b^k$: entonces el tiempo de ejecución es el mismo en cada nivel.
   El TM indica $O(n^k \log n)$

3. $a > b^k$: el tiempo de ejecución aumenta en cada nivel, predomina el tiempo del nivel de las hojas
   El TM indica $O(n^{\log_b a})$

Otra propiedad útil (a veces llamada T.M. de resolución por sustracción):

- Reducción por sustracción:

  La ecuación de la recurrencia es la siguiente:

$$T(n) = \begin{cases} c * n^k & \text{si } 0 \leq n < b \\ a * T(n-b) + c * n^k & \text{si } n \geq b \end{cases}$$

  La resolución de la ecuación de recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < 1 \\ \theta(n^{k+1}) & \text{si } a = 1 \\ \theta\left(a^{n \text{ div } b}\right) & \text{si } a > 1 \end{cases}$$

Calculo de Orden de complejidad algorítmica de la ecuación de recurrencia

T(n)= 2 T(n/2)+ n , con T(1)=1

Paso 1: Expansión de la recurrencia

Vamos a expandir la recurrencia varias veces para identificar un patrón general.

1. **Primera expansión:**

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

2. **Segunda expansión:**

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

3. **Tercera expansión:**

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$$
$$T(n) = 4\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n$$
$$T(n) = 8T\left(\frac{n}{8}\right) + 3n$$

Paso 3: Determinación del valor de k

La expansión termina cuando $\frac{n}{2^k} = 1$. Esto ocurre cuando $n = 2^k$, por lo tanto $k = \log_2 n$.

Paso 4: Sustitución de k en la expresión general

Sustituimos $k = \log_2 n$ en la expresión:

$$T(n) = 2^{\log_2 n} T(1) + (\log_2 n) n$$

Sabemos que $2^{\log_2 n} = n$ y $T(1) = 1$:

$$T(n) = n \cdot 1 + n \log_2 n$$
$$T(n) = n + n \log_2 n$$

Paso 5: Determinación del término dominante

Para el análisis de complejidad asintótica, nos quedamos con el término dominante:

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

Por lo tanto, la complejidad algorítmica de la ecuación de recurrencia $T(n) = 2T(n/2) + n$ es $\boxed{\Theta(n \log n)}$, incluso sin utilizar el Teorema Maestro.

Si lo hiciéramos por el teorema maestro

Reducción por división:

La ecuación de la recurrencia es la siguiente:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

La resolución de la ecuación de recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

T(n)= 2 T(n/2)+ n , con T(1)=1

Los valores seria: a = 2, b = 2, c = 1 y k = 1

Entonces como $\text{si } a = b^k$

El orden es O(n) = n log (n)

Ejercicios propuestos:

1. Determine la expresión del coste temporal para el peor caso y calcule el O correspondiente para cada una de estas funciones:

```
int suma(int v[], int N)
{
if (N==0) return v[0];
 else return (v[N] + suma (v, N-1)) ;
}
```

2. función que resuelve el problema de las "Torres de Hanoi"

3. La siguiente función calcula la media de los números de un vector v que tiene un tamaño que es potencia de 2.

```
float Media1 (int v[], int pr, int ul)
{
  if (pr ==ul)
     {return v[pr];}
  else
     {return (Media1(v, pr, (pr+ul)/2 ) + Media1(v, (pr+ul)/2+1, ul))/2.0}

};
```

4. Para cada una de las siguientes ecuaciones básicas de recurrencia, indicar su O para el pero caso (utilizar expansión) y dar un ejemplo de algoritmo que responda a esa forma:
   a. $T(n)=T(n-1)+n$, con $T(0)=1$
   b. $T(n)= 2\ T(n-1)+n$, con $T(0)=1$
   c. $T(n)=T(n-1)+1$, con $T(0)=1$
   d. $T(n)= 3T(n-1)+1$, con $T(0)=1$
   e. $T(n)=T(n/2)+1$, con $T(1)=1$
   f. $T(n)= 2\ T(n/2)+1$, con $T(1)=1$
   g. $T(n)= 3\ T(n/2)+1$, con $T(1)=1$
   h. $T(n)=T(n/2)+n$, con $T(1)=1$
   i. $T(n)= 2\ T(n/2)+ n$, con $T(1)=1$

5. Se tiene un vector de valores enteros no repetidos y ordenados de forma creciente. Desarrollar un algoritmo que establezca si algún elemento del vector tiene un valor que coincida con su índice. El algoritmo debe tener coste logarítmico.

6. Determinar el O para el peor caso del máximo común divisor por el algoritmo de Euclides en su forma iterativa y en su forma recursiva.

Bibliografia

Donald E. Knuth, 'Big Omicron and Big Omega and Big Theta', SIGACT News, 8(2):18-24, April-June 1976.

Knuth, 'The Art of Computer Programming', Addison Wesley, vol I, II, III.

R. Sedgewick y P. Flajolet. 'An Introduction to Analysis of Algorithms', Addison-Wesley, 1997.

Cormen Gilles Brassard. 'Crusade for a Better Notation', SIGACT News Vol 17, 1985

A. Aho, J. Hopcrof y J. Ullman. 'The Design and Analysis of Computer Algorithms', Addison-Wesley, 1974.

Xavier Franch Gutierrez. 'Estructuras de Datos, especificación, diseño e implementación', Alfaomega, 2002

R. Sedgewick. 'Algorithms in C', Addison-Wesley, 1990

R. Guerequeta , A Vallecillo, 'Técnicas de Diseño de Algoritmos', Servicio de Publicaciones de la Universidad de Málaga, 1998

Complejidad amortizada

La complejidad amortizada es una técnica de análisis que se utiliza para determinar el tiempo promedio por operación de un algoritmo en el peor de los casos, a lo largo de una secuencia de operaciones. En lugar de analizar el tiempo de ejecución de una sola operación en particular, el análisis amortizado proporciona una estimación del costo total de una serie de operaciones, dividiendo este costo total por el número de operaciones. Este tipo de análisis es especialmente útil cuando ciertas operaciones pueden ser muy costosas individualmente, pero esas operaciones costosas ocurren con poca frecuencia.

Métodos para Análisis Amortizado

Existen varios métodos comunes para realizar análisis amortizado:

1. **Método del Costo Agregado (Aggregate Method):**
   - Se calcula el costo total de una secuencia de operaciones y se divide por el número de operaciones para obtener el costo amortizado por operación.
   - Ejemplo: Si una secuencia de n operaciones tiene un costo total de $T(n)$, entonces el costo amortizado por operación es $\frac{T(n)}{n}$.
2. **Método del Análisis de Contabilidad (Accounting Method):**
   - Se asigna un "crédito" a cada operación. Las operaciones baratas pueden acumular crédito extra, que luego se puede usar para pagar las operaciones más costosas.
   - Cada operación se paga con el crédito asignado, y el análisis asegura que siempre haya suficiente crédito para cubrir los costos.
3. **Método del Potencial (Potential Method):**
   - Se define una función de potencial que mide la "energía almacenada" en el sistema. El costo amortizado de una operación es el costo real más el cambio en el potencial.
   - Este método es similar al método de contabilidad, pero usa una función matemática para formalizar la acumulación y el uso de crédito.

Ejemplo de Complejidad Amortizada

Consideremos el ejemplo clásico de un vector dinámico (array dinámico) que se duplica en tamaño cuando se llena. Supongamos que tenemos un vector que inicialmente tiene capacidad 1 y que cada vez que se llena, su capacidad se duplica:

1. **Insertar el primer elemento:**
   - Costo: $O(1)$
2. **Insertar el segundo elemento (requiere redimensionar):**
   - Costo de redimensionar: $O(1)$
   - Costo de copiar elementos: $O(1)$
   - Costo total: $O(2)$
3. **Insertar el tercer elemento:**
   - Costo: $O(1)$
4. **Insertar el cuarto elemento (requiere redimensionar):**
   - Costo de redimensionar: $O(1)$
   - Costo de copiar elementos: $O(2)$
   - Costo total: $O(3)$

Y así sucesivamente.

Para n inserciones, el costo total es la suma de todas las operaciones de inserción y redimensionamiento. A través del método del costo agregado, se puede demostrar que, aunque algunas operaciones individuales son costosas, el costo total dividido por el número de operaciones da un costo amortizado de $O(1)$ por inserción.

Ejemplo: pila con eliminación múltiple

Este es quizás uno de los ejemplos más comunes e importantes de la idea de análisis amortizado.

Supongamos que tenemos una pila, como podría ser por ejemplo stack de la STL de C++. Comenzando con una pila vacía, vamos a realizarle **dos tipos de operaciones**:

1. Agregar un cierto elemento x a la pila.
2. Sacar una cierta cantidad k de elementos de la pila.

Tanto los x como los k de las operaciones anteriores son arbitrarios, y por ejemplo podemos pensar que los elige un usuario, o se leen de la entrada, sin que tengamos control sobre ellos.

La pregunta que nos hacemos es, ¿Cuál es la complejidad de estas dos operaciones? La operación 1, de agregar, es la más simple, pues consistirá de un único `.push()` en el stack, que ya sabemos que tiene complejidad $O(1)$ [1)].

Si hacemos un análisis de cuánto tarda la operación de sacar, en el peor caso podrían tener que sacarse muchos números, pues la cantidad que se saca depende del k ingresado. En particular, k podría ser tan grande como todos los elementos que se han agregado en la pila hasta el momento. Por lo tanto en el peor de los casos la operación 2 tiene un costo $O(N)$, siendo N la cantidad de elementos que se agregan en total (es decir, la cantidad de operaciones de tipo 1).

Entonces por ejemplo, si se hacen 1000 operaciones de tipo 1, y 1000 operaciones de tipo 2, como las operaciones de tipo 2 pueden costar hasta 1000 según nuestro análisis previo, el costo total de estas operaciones será como mucho $1000 \cdot 1000$ en el peor de los casos.

Si bien este análisis es correcto, es **pesimista**: En realidad el costo total será siempre **muchísimo menor** que $1000 \cdot 1000$. Observemos que, para que el costo total sea tan grande, **todas** las operaciones de tipo 2 deberían ser caras, es decir, en **todas** ellas se deberían eliminar **todos** los números. Pero esto es imposible: Una vez que un número fue eliminado, ya no está en la pila, y no puede "ser eliminado de nuevo". En otras palabras, **cada número que se agrega con una operación tipo 1, puede ser borrado a lo sumo una vez, en una sola operación tipo 2**.

Con eso en mente, podemos pensar el costo **total** de otra manera: En lugar de pensar cuánto cuesta hacer los borrados en **una** operación particular, pensamos cuántas veces se borra un cierto **elemento** particular. La observación es entonces que, sin importar en qué operación se borre, cada elemento que se agrega se borra como máximo **una única vez**, y entonces el costo total de los borrados es proporcional a la cantidad de elementos agregados, es decir, a la cantidad de operaciones de tipo 1.

Si llamamos N al total de operaciones (tanto de tipo 1 como de tipo 2), tenemos entonces resumiendo:

1. Las operaciones de tipo 1 son $O(1)$, y por lo tanto en total ejecutarlas todas tiene un costo $O(N)$
2. Las operaciones de tipo 2 son en total $O(N)$, y lo único "caro" que hacen es borrar elementos, pero **en total**, el costo de los borrados es $O(N)$. Por lo tanto el costo total de hacer **todas** las operaciones de este tipo será también $O(N)$.
3. El costo total de ejecutar todas las operaciones es, por los dos anteriores, $O(N)$.
4. Entonces, como N operaciones siempre tienen un costo proporcional a N, el **costo promedio por operación** resulta ser $\frac{N}{N} = 1$. Decimos entonces que ambas operaciones tienen un costo $O(1)$ **amortizado**: podemos pensar que ambas cuestan $O(1)$, y el costo final que obtendremos al calcular será el correcto.

Importancia de la Complejidad Amortizada

La complejidad amortizada es importante porque proporciona una visión más realista del rendimiento de un algoritmo en situaciones prácticas. Permite diseñar estructuras de datos y algoritmos que, aunque puedan tener operaciones costosas ocasionalmente, ofrecen un rendimiento eficiente en promedio, lo cual es crucial para aplicaciones donde se manejan grandes volúmenes de datos y operaciones.

Fin