1. Eficiencia

1.1 Introducción

Cuando tenemos un problema real, lo primero que tenemos que preguntarnos es si se puede resolver con la ayuda de un ordenador, en caso afirmativo, debemos modelizar el problema. Para ello, diseñamos un algoritmo siguiendo alguna técnica de diseño. Sobre cada una de las técnicas de diseño deberemos analizar: 1) Recursos que consume, 2) Obtiene la solución óptima o aproximada. Con respecto al estudio de los recursos que consume deberemos estudiar la eficiencia del algoritmo medida como tiempo de computación y necesidades de memoria. Dependiendo de estos factores podremos filtrar si el problema puede resolverse con un ordenador. A lo largo del curso veremos que dependiendo de la estrategia que siga nuestro algoritmo los recursos necesarios pueden variar y las soluciones a las que llegan son las óptimas o aproximadas. Las posibles técnicas nos pueden producir situaciones donde :

- El algoritmo necesita mucha memoria pero es rápido y además da la solución óptima. Estas características son frecuentes en problemas resueltos con Programación Dinámica.
- El algoritmo necesita poca memoria y es rápido, pero depende del problema que se obtenga una solución óptima. Esta situación aparecen en algoritmos que siguen la estrategia voraz.
- El algoritmo se describe muy intuitivamente. La estrategia que se puede calificar como evidente es la técnica divide y venceras, pero normalmente va a gastar muchos recursos, tanto en tiempo de computación como en memoria.
- El algoritmo necesita muchos recursos, y se formula todas las posibles salidas. Estos algoritmos son los que siguen una estrategia de Ramificación, y deberemos estudiar mecanismos para que estos algoritmos no se disparen en tiempo y en memoria. Por otro lado estos algoritmos encontrarán siempre la solución óptima si existe.

Por lo tanto en este primer tema tenemos que estudiar todas las técnicas que nos permitan medir los recursos que gasta un algoritmo.

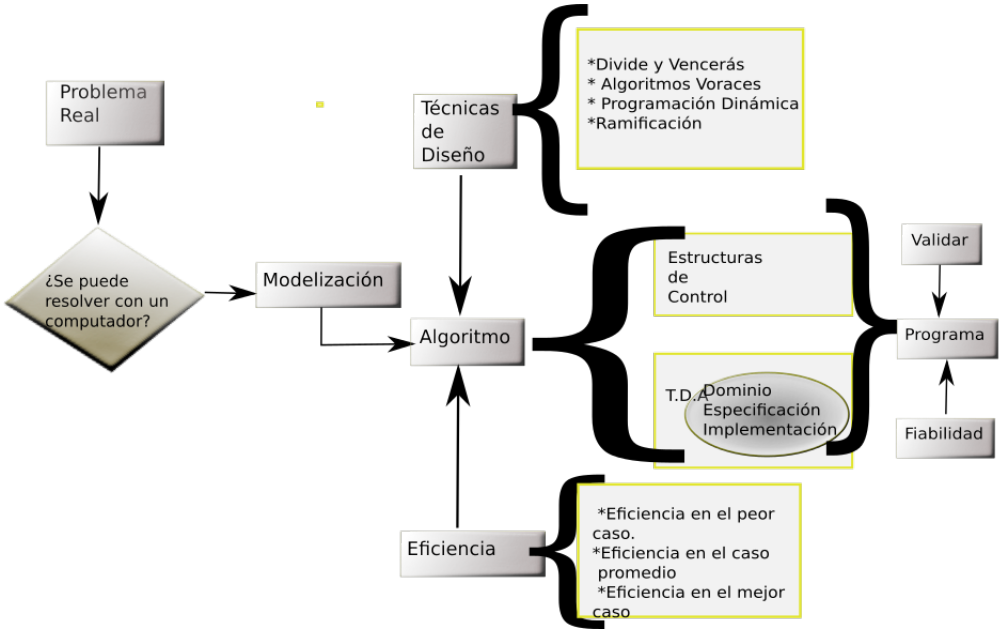


Figura 1.1: **Esquema**: Pasos a seguir para resolver un problema en un ordenador. Detalles a llevar a cabo para resolver el problema con un algoritmo

1.2 Algoritmo

El concepto de algoritmo se puede definir como una secuencia ordenada de instrucciones capaz de resolver un problema dado. Un algoritmo puede tener diferentes implementaciones. La implementación es la traslación de la secuencia de pasos ordenada -definidos por el algoritmo- a un lenguaje. En nuestro caso será a un lenguaje de programación (p.e $C++$). Diferentes implementaciones de un algoritmo deben tener el mismo orden de eficiencia. Por otro lado si tenemos diferentes algoritmos que resuelven un mismo problema no tienen porque tener el mismo orden de eficiencia.

Ejemplo 1.2.1
Obtener *Fibonacci* de un numero natural n.
Podemos plantear un método recursivo $f(n) = f(n-1) + f(n-2)$ de tal forma que tendriamos el siguiente código:

```
1  int fibonacci(int n){
2   if (n==0 ) return 0;
3   else if (n==1) return 1;
4   else return (fibonacci(n-1)+fibonacci(n-2));
5  }
```

O podemos plantear un método iterativo que recuerde cada $f(n)$ ya calculado.

```
int fibonacci(int n){
 vector<int> v(n+1);
 for (i=0;i<=n;i++){
   if (i==0 ) v[i]=0;
   else if (i==1) v[i]=1;
   else v[i]=v[i-1]+v[i-2];
 }
 return v[n];
}
```

Ambos algoritmos resuelven el mismo problema. Pero desde el punto de vista de la eficiencia medida como tiempo de ejecución el segundo es mucho más rápido que el primero. El cálculo de esta eficiencia se verá en el tema.

□

Las consideraciones que vamos a tratar sobre un algoritmo son:

1. Eficiencia. Son los recursos necesarios por el algoritmo. Normalmente serán memoria y tiempo de ejecución.
2. Diseño del algoritmo.- Se define como la técnica o método a seguir para definir los pasos que resuelven el problema. En este curso se verán las técnicas: Divide y Vencerás, Voraces o Greedy, Programación Dinámica y Ramificación.

1.3 Eficiencia de un Algoritmo

Cuando queremos medir la eficiencia -fijemonos en el tiempo de ejecución- de un algoritmo, el primer planteamiento puede ser ejecutarlo y medir cuanto tiempo tarda. Este tipo de eficiencia es la experimental o posteriori. Medir así la eficiencia de un algoritmo es dependiente de varios factores:

1. Datos de entrada.- Dependiendo de la naturaleza de los datos el algoritmo puede ejecutarse más o menos rápdio.
2. Calidad del código y nivel de optimización del compilador.
3. Rapidez de las instrucciones máquina.
4. Complejidad en si del algoritmo

Para que podamos medir la eficiencia de un algoritmo si tener en cuenta todos estos factores se supone que el algoritmo se ejecuta en un máquina ideal. Y a partir de la medición en esta máquina ideal podemos obtener la eficiencia a priori. Esta eficiencia obtiene una función que acota, inferior o superiormente el tiempo de ejecución del algoritmo para cualquier conjunto de datos de entrada.
Para llevar a cabo el estudio de la eficiencia vamos a presentar los siguiente conceptos:

- Tamaño de la entrada o talla. Número de elementos sobre los que se va a ejecutar el algoritmo (n).
- Tiempo de ejecución $T(n)$. Indica el número de instrucciones elementales ejecutadas por un ordenador idealizado. Esta medida es independiente del ordenador donde se ejecute.

1.3.1 Principio de Invarianza

Un algoritmo puede tener más de una implementación pero la diferencia en eficiencia de las distintas implementaciones no es mayor que una constate multiplicativa c. Así si tenemos un algoritmo A con

dos implementaciones I_1 e I_2 y tienen respectivamente tiempos de ejecución $T_1(n)$ y $T_2(n)$ siguiendo el *principio de invarianza* debe cumplirse que:

$$T_1(n) \leq cT_2(n) \quad \forall n \geq n_0 \text{ y } c > 0$$

Ejemplo 1.3.1
Como escoger entre dos Algoritmos.
Supongamos que tenemos dos algoritmos que tienen tiempo de ejecución $T_1(n) = 10^6 n^2$ y $T_2(n) = 5n^3$. Intuimos que $T_1(n) \leq T_2(n)$, ¿ pero a partir de que tamaño de problema (n) $T_1 \leq T_2$?.
Para contestar a esta pregunta igualamos los dos tiempos de ejecución multiplicanto T_2 por la constante c.

$$10^6 n^2 \leq c5n^3$$

ahora igualamos

$$10^6 n^2 = c5n^3$$

Igualando a cero

$$10^6 n^2 - c5n^3 = 0$$

Sacando factor común n^2

$$n^2(10^6 - c5n) = 0$$

Una solución trivial es $n = 0$ y la otra es que $(10^6 - c5n) = 0$. Asi que despejando de esta última n obtenemos:

$$10^6 - c5n = 0 \rightarrow n_0 = \frac{10^6}{5c}$$

escogiendo $c = 1$ obtenemos $n_0 = 2 * 10^5$.
Con estos datos podemos concluir que el algorimtmo con tiempo $T_2(n) = 5n^3$ se comporta mejor (es más eficiente) para tamaño de datos menores a n_0. Pero a partir de n_0 hasta el infinito es mejor el algoritmo con tiempo de ejecución $T_1(n) = 10^6 n^2$.

□

1.4 Reglas Generales

A continuación daremos las reglas generales para obtener el tiempo que tarda en ejecutarse las diferentes operaciones elementales (OE) en un ordenador ideal.

- Tiempo de ejecución de una O.E cuesta 1.

- Tiempo de ejecución de una secuencia de instrucciones se calcula sumando los tiempos de ejecución de cada una de las instrucciones
- Tiempo de ejecución de la sentencia switch como la siguiente

```
switch (c)
 case 1:
   Acciones 1: S1
   break;
 case 2:
    Acciones 2: S2
    break;
 ...
 case n:
    Acciones n:Sn
```

se obtiene como $T(n) = T(c) + max\{T(S1), T(S2), \ldots, T(Sn)\}$ siendo $T(c)$ el tiempo para evaluar c que se compara con las diferentes posibilidades.

- Tiempo de ejecución de **if-else**

```
if (C)
   S1
else
  S2
```

se obtiene como $T(n) = T(C) + max\{T(S1), T(S2)\}$

- Tiempo de ejecución de un bucle **while**

```
while (C)
 S1
```

se obtiene como $T(n) = T(C) + numero_i teraciones \times (T(S1) + T(C))$

- Tiempo de ejecución de un bucle **for**

```
for (int i=0;i<n;i++)
   S
```

se obtiene como $T(n) = \underbrace{1}_{i=0} + \underbrace{1}_{i<n} + n \times (T(S) + \underbrace{1}_{i++} + \underbrace{1}_{i<n}) = 2 + (T(S)+2) \times n$

- Llamada a una funcion $f(P_1, \ldots, P_n)$ se obtiene como:

$$T(n) = \underbrace{1}_{llamada} + T(P_1) + \cdots + T(P_n) + T(Funcion)$$

siendo $T(P_i)$ el tiempo para pasar el parámetro (p.e puede que haya que hacer cálculos para obtener P_i) y $T(Funcion)$ es lo que cuesta ejecutar la función, que se obtiene usando las reglas vistas.

1.5 Tiempos de Ejecución: Casos

A continuación vamos a ver como obtener el tiempo de ejecución de un algoritmo. Para ello vamos a presentar tres tipos de tiempo de ejecución:

- Tiempo de ejecución en el mejor caso.
- Tiempo de ejecución en el peor caso.
- Tiempo de ejecución en el caso promedio

Para estudiar los diferentes tiempos de ejecución vamos a usar el algoritmo que busca secuencialmente un elemento en un vector.

```
/**
  @brief Busca un elemento en un vector.
  @param a: array con los elementos. Los elementos estan ordenados de forma creciente
  @param n: numero de elementos
  @param x: el elemento a buscar
  @return la posicion en a donde esta x. En caso de que no este se devuelve -1.
  */
int Buscar(int a[],int n,int x){
  int j;
  j=0;
  while (a[j]<x && j<n){
    j=j+1;
  }
  if (a[j]==x)
    return j;
  else
    return -1;
}
```

1.5.1 Caso Mejor

El tiempo de ejecución en el caso mejor se hace un seguimiento en el código, o traza, con el menor número de operaciones elementales. Es importante resaltar que no tiene que ver el mejor caso con que el tamaño del problema sea de menor tamaño. Los tiempos de ejecución se describen independientes del tamaño del problema.

Ejemplo 1.5.1
Obtener el tiempo de ejecución en el mejor caso de la búsqueda secuencial.

```
/**
  @brief Busca un elemento en un vector.
  @param a: array con los elementos
  @param n: numero de elementos
  @param x: el elemento a buscar
  @return la posicion en a donde esta x. En caso de que no este se devuelve -1.
  */
int Buscar(int a[],int n,int x){
  int j;
```

```
10    j=0;  //  ← 1
11    while (a[j]<x && j<n){//  ← 4
12      j=j+1;
13    }
14    if (a[j]==x)//  ← 2
15      return j;//
16    else
17      return -1;//  ← 1
18  }
```

Hemos añadido a las líneas que se ejecutan un comentario con el número de operaciones elementales que se ejecutan. Para notar el tiempo de ejecución en el mejor caso lo haremos con $T_m(n)$. En este caso se define como

$$T_m(n) = 1+4+2+1 = 8. \tag{1.1}$$

1.5.2 Caso Peor

El tiempo de ejecución en el caso peor se hace un seguimiento en el código, o traza, con el mayor número de operaciones elementales.

Ejemplo 1.5.2
Obtener el tiempo de ejecución en el peor caso de la búsqueda secuencial.

```
1   /**
2     @brief Busca un elemento en un vector.
3     @param a: array con los elementos
4     @param n: numero de elementos
5     @param x: el elemento a buscar
6     @return la posicion en a donde esta x. En caso de que no este se devuelve -1.
7     */
8   int Buscar(int a[],int n,int x){
9     int j;
10    j=0;  //  ← 1
11    while (a[j]<x && j<n){//  ← 4
12      j=j+1;//  ← 2
13    }
14    if (a[j]==x)//  ← 2
15      return j;//
16    else
17      return -1;//  ← 1
18  }
```

Hemos añadido a las lineas que se ejecutan un comentario con el numero de operaciones elementales que se ejecutan. Para notar el tiempo de ejecución en el peor caso lo haremos con $T_p(n)$. En este caso

se define como

$$T_p(n) = 1+4+(\sum_{j=0}^{n-1}(2+4))+2+1 = 8+6n \tag{1.2}$$

□

1.5.3 Caso Medio

El tiempo de ejecución en el caso medio traza el algoritmo que realiza un número de instrucciones igual a la esperanza matemática de la variable aleatoria definida por todas las trazas del algoritmo para un tamaño de la entrada, con las posibilidades de que estas ocurran con la entrada. Para obtener este tiempo en primer lugar debemos determinar cual es la esperanza matemática (media) del número de instrucciones medio que se realiza. Así si por ejemplo tenemos un while como el siguiente:

```
1 int cnt=0;
2 vector<int > a (5);
3 int i=0;
4 ....
5 while (a[i]!=3 && i<5){
6   cnt++;
7   i++;
8 }
```

Las trazas están determinadas por la probabilidad de que $a[i] == 3$. Si existe la misma probabilidad de que el 3 esté en la posición 0,1,2,3,4 entonces esta probabilidad es $\frac{1}{5}$. Y por lo tanto el numero de iteraciones que dará el *while* será:

$$\sum_{i=0}^{4} \frac{1}{5} i$$

De forma que si en la posición 0 si está 3 se ejecuta 0 veces con probabilidad $0/n$ con $n = 5$. Si esta en la posición 1 se ejecuta el while 1 vez con probabilidad de que no esté en 0 y que si este en 1, esto suma $1/n$. Si está en la posición 2 se ejecutará 2 iteraciones con probabilidad $1/n+2/n$ si está en la posición j-ésima se ejecutrá $\underbrace{1/n}_{\text{no está en 0}} + \underbrace{2/n}_{\text{no está en 1}} + \cdots + \underbrace{j/n}_{\text{no está en j-1}}$.

Ejemplo 1.5.3
Obtener el tiempo de ejecución en el caso medio de la búsqueda secuencial.
Antes de pasar a estudiar el tiempo de ejecución nos debemos plantear cuantas veces se ejecuta el while

```
1 /**
2   @brief Busca un elemento en un vector.
3   @param a: array con los elementos
```

```
4    @param n: numero de elementos
5    @param x: el elemento a buscar
6    @return la posicion en a donde esta x. En caso de que no este se devuelve -1.
7    */
8  int Buscar(int a[],int n,int x){
9    int j;
10   j=0;  //  ← 1
11   while (a[j]<x && j<n){//  ← 4 Cuantas veces se ejecuta en el caso medio???
12     j=j+1;//  ← 2
13   }
14   if (a[j]==x)//  ← 2
15     return j;//
16   else
17     return -1;//  ← 1
18 }
```

Suponiendo que el dato x tiene la misma probabilidad de que esté en la posición 1,2,...,n-1 y siendo equiprobables, es decir con probabilidad $\frac{1}{n}$. Resulta que el número medio de veces que se ejecuta el bucle for es :

$$\sum_{i=0}^{n-1} \frac{1}{n} i = \frac{1}{n} \sum_{i=0}^{n-1} i = \frac{1}{n}(0+1+\cdots+n-1) = \frac{1}{n}\frac{n-1}{2}n = \frac{n-1}{2} \tag{1.3}$$

Por lo tanto el tiempo de ejecución en el caso medio es :

$$T_{\frac{1}{2}}(n) = 1+4+(\sum_{j=0}^{\frac{n-1}{2}} 2+4)+3 = 11+3n \tag{1.4}$$

Ejemplo 1.5.4
Obtener los tiempos de ejecución de la función *Máximo*, que busca la posción del elemento máximo en un vector.

```
1  int Maximo(int a[],int i,int j){
2   int pmax;
3   pmax = i;
4   for (int k=i+1;k<=j;k++)
5    if (a[k]>a[pmax])
6      pmax =k;
7   return pmax;
8  }
```

CASO PEOR. En este caso la condición if siempre es verdadera. Por lo tanto el tiempo de ejecución será:

$$T_p(n) = T_p(j-i+1) = 1+3+(\sum_{k=i+1}^{j}(3+2+1))+\underbrace{1}_{return}$$

$$= 5+6(j-i) = 5+6(n-1) = 6n-1$$

CASO MEJOR. En este caso la condición if es siempre false. Por lo tanto el tiempo de ejecución será:

$$T_m(n) = T_m(j-i+1) = 1+3+(\sum_{k=i+1}^{j}(3+2))+\underbrace{1}_{return}$$

$$= 5+6(j-i) = 5+5(n-1) = 5n$$

CASO MEDIO. En este caso la condición if se ejecuta el 50 % de las veces. Por lo tanto el tiempo de ejecución será:

$$T_{\frac{1}{2}}(n) = T_{\frac{1}{2}}(j-i+1) = 1+3+(\sum_{k=i+1}^{j}(3+2+\tfrac{1}{2}))+\underbrace{1}_{return}$$

$$= 5{,}5n-0{,}5$$

□

Ejemplo 1.5.5
Obtener los tiempos de ejecución en el caso mejor,peor y medio del algoritmo de ordenación por inserción.

```
1    void Insercion(int a[],int n){
2      for (int i=1;i<n;i++){
3        int x=a[i]; int j=i-1;
4        while (j>=0 && x<a[j]){
5           a[j+1]=a[j];
6           j=j-1;
7        }
8        a[j+1]=x;
9      }
10   }
```

CASO MEJOR. Se da cuando el vector está ordenado. En este caso el while no se ejecuta.

$$T_m(n) = \underbrace{2}_{i=1\ \ i<n} + \sum_{i=1}^{n-1} 2 + 2 + \underbrace{4}_{\text{while}} + 3 + \underbrace{2}_{i++,i<n} = 13n - 11$$

CASO PEOR. Se da cuando el vector está ordenadode forma inversa. En este caso siempre se entra en el while y se recorre de $i-1$ hasta 0.

$$T_p(n) = \underbrace{2}_{i=1\ \ i<n} + \sum_{i=1}^{n-1} 4 + \underbrace{4}_{\text{while}} + (\sum_{j=0}^{i-1} 4 + 2 + 4) + 3 + \underbrace{2}_{i++,i<n}$$

$$= 2 + \sum_{i=1}^{n-1} 13 + \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 10 =$$

$$2 + 13(n-1) + \sum_{i=1}^{n-1} 10i =$$

$$2 + 13n - 13 + 10(\tfrac{n}{2}(n-1)) =$$

$$5n^2 + 8n - 11$$

CASO MEDIO. Supondremos equiprobables la posición de cada elemento dentro del vector. Por tanto para cada i, la probabilidad de que el elemento se sitúe en la posición k de las i primeras posiciones será $\frac{1}{i}$. Esto da lugar a:

$$\sum_{j=0}^{i-1} \frac{1}{i} j = \frac{1}{i}(i-1)\frac{i}{2} = \frac{i-1}{2} \leftarrow \text{Número promedio de veces que entra en el while}$$

Con este dato ya podemos definir el tiempo de ejecución en el caso medio:

$$T_{\frac{1}{2}}(n) = 2 + 13(n-1) + \sum_{i=1}^{n-1} \sum_{j=0}^{\frac{i-1}{2}} 10 =$$

$$2 + 13(n-1) + 10 \sum_{i=1}^{n-1} \tfrac{i+1}{2} =$$

$$\tfrac{1}{2}(5n^2 + 31n) - 16$$

□

Ejemplo 1.5.6
Obtener los tiempos de ejecución en el caso mejor,peor y medio del algoritmo de ordenación por selección

```
1  void Seleccion(int a[],int n){
2   for  (int i=0;i<n-1;i++){
3     int pmin=i;
4     for (int j=i+1;j<n;j++){
5        if (a[pmin]>a[j])
6          pmin=j;
7     }
8     swap(a[i],a[pmin];
9    }
10 }
```

CASO MEJOR El vector ya está ordenado y por lo tanto nunca

$$T_m(n) = \underbrace{3}_{i=0 i<n-1} + (\sum_{i=0}^{n-2} 1 + 3 + \sum_{j=i+1}^{n-1} \underbrace{3}_{if} + \underbrace{1}_{j++} + \underbrace{1}_{j<n}) + \underbrace{7}_{swap}$$

$$= 3 + 11(n-1) + \sum_{i=0}^{n-2} 5(n-i-1) = \\ = \frac{1}{2}(5n^2 + 17n - 16)$$

CASO PEOR El vector está ordenado pero en sentido inverso y por lo tanto siempre se cumple la condición $a[pmin] > a[j]$.

$$T_m(n) = \underbrace{3}_{i=1 i<n-1} + \sum_{i=0}^{n-2} 1 + 3 + (\sum_{j=i+1}^{n-1} \underbrace{3}_{if} + \underbrace{1}_{j++} + \underbrace{1}_{j<n} + 1) + \underbrace{7}_{swap}$$

$$= 3n^2 + 8n - 8$$

CASO MEDIO El if es verdader el 50 % de las veces.

$$T_m(n) = \underbrace{3}_{i=1 i<n-1} + \sum_{i=0}^{n-2} 1 + 3 + (\sum_{j=i+1}^{n-1} \underbrace{3}_{if} + \underbrace{1}_{j++} + \underbrace{1}_{j<n} + \frac{1}{2}) + \underbrace{7}_{swap}$$

$$= \frac{5}{2}n^2 + \frac{39}{4}n + 8$$

□

1.6 Cotas de Complejidad.

En esta sección vamos a describir un conjunto de medidas asíntoticas que permiten acotar superior e inferiormente , y en el caso promedio los tiempos de ejecución. Son medidas asíntoticas porque se establecen estas acotaciones cuando la talla del problema se hace muy grande, podríamos decir cuando $n \to \infty$

1.6.1 Cota Superior. Notación O-grande

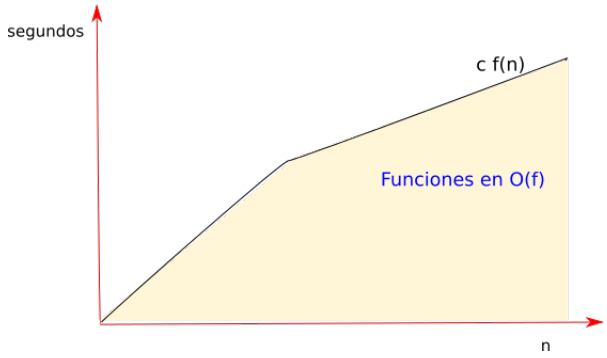
Al conjunto de funciones g que a lo sumo crecen tan deprisa como f se denominan $\mathbf{O}(f)$. Si conocemos la cota superior de un algoritmo siempre aseguraremos que el tiempo de ejecución nunca será peor que

esta cota. Podemos decir que si $g \in \mathbf{O}(f)$, el tiempo de ejecución g será siempre menor o igual que el tiempo de ejecución f cuando lo analizamos para tamaños de problema muy grandes, es decir cuando $n \to \infty$.

Definición 1.6.1 O-grande Sea $f : \mathbb{N} \to [0,\infty]$. Se define el conjunto de funciones de orden $\mathbf{O}(f)$ como:

$$\mathbf{O}(f) = \{g : \mathbb{N} \to [0,\infty] | \exists\, c \in R, c > 0, \exists n_0 \in \mathbb{N} \text{ tal que } g(n) \leq cf(n) \;\; \forall n \geq n_0\}$$

En la siguiente imagen se muestra la región donde se localizan las funciones que pertenecen en $O(f)$



♠

Ejemplo 1.6.1
Suponed que $f(n) = n^2$, dar ejemplo de funciones en $\mathbf{O}(f)$
Un ejemplo de funciones en $\mathbf{O}(f)$ sería:
$\mathbf{O}(f) = \{1, 5, log_2(n), \sqrt{n}, 5n+10, 30n^2+20, \dots\}$
En general todas las funciones que quedan por debajo de $f(n)$ si la pintamos en una gráfica. Puede que para un numero finito de puntos la función quede por enciam pero cuando $n \to \infty$ como máxmo la iguala. Como ejemplo de una función que como máximo la iguala sería $\mathbf{O}(f)$ la función $30n^2+20$. Para comprobar que es cierto debemos buscar según la definición 1.6.1 una constate c y n_0 a partir del cual siempre se cumple que :

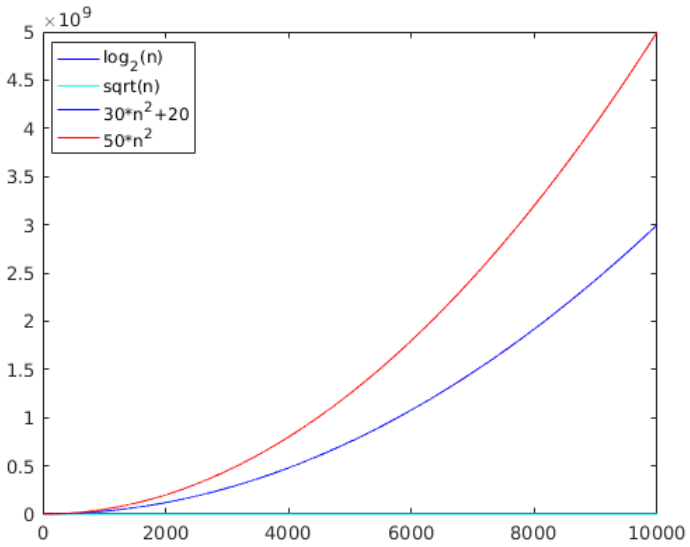
$$30n^2 + 20 \leq cn^2 \tag{1.5}$$

Para encontrar estos valores, c y n_0, igualamos los dos términos

$$\begin{gathered} 30n^2+20 = cn^2 \Longrightarrow \\ \text{no llevamos todo a la izquierda} \\ 30n^2+20-cn^2 = 0 \Longrightarrow \\ n^2(30-c)+20 = 0 \Longrightarrow \\ n = \sqrt{\frac{20}{c-30}} \end{gathered} \tag{1.6}$$

Para valores de $c > 30$ p.e $c = 50 \Longrightarrow n_0 = 1$

Lo cual nos indica que para $c = 50$ y $n_0 = 1$ demostramos que $30n^2+20 \in \mathbf{O}(f)$. En la siguiente imagen se muestra ejemplos de funciones g y la señal f.



□

Ejemplo 1.6.2
Sea $g(n) = 5n^2 + 9n$ ¿cuál es el orden de g?.
Siguiendo la definición 1.6.1, claramente se puede ver $g(n) \in \mathbf{O}(n^2)$. Para ello, te quedas con el monomio mayor, y eliminas la constante.

$$
\begin{gathered}
g(n) \leq cn^2 \Longrightarrow \\
5n^2 + 9n \leq cn^2 \Longrightarrow \\
5n^2 + 9n - cn^2 = 0 \Longrightarrow \\
\text{sacando factor común } n \Longrightarrow \\
n(n(c-5)-9) = 0 \Longrightarrow \\
\text{una solución es que se haga } (n(c-5)-9) = 0 \Longrightarrow \\
n = \tfrac{9}{c-5} \Longrightarrow \text{ para } c > 5 \text{ se cumple} \\
\text{escogiendo } c = 6 \text{ obtenemos } n_0 = 9
\end{gathered}
$$

□

Propiedades de la Notación O-grande

A continuación se describe un conjunto de propiedades que cumple la notación **O**-grande:

1. Para cualquier función $f \Longrightarrow f \in \mathbf{O}(f)$.
2. Si $f \in \mathbf{O}(g) \Longrightarrow$O(f)$\subset \mathbf{O}(g)$
3. Si $\mathbf{O}(f) = \mathbf{O}(g) \Longleftrightarrow f \in \mathbf{O}(g) \wedge g \in \mathbf{O}(f)$
   Dem-
   - Supongamos que $\mathbf{O}(f) = \mathbf{O}(g)$
     Por la propiedad 1) $f \in \mathbf{O}(f) \Longrightarrow f \in \mathbf{O}(g)$
     Igualmente por la propiedad 1)$f \in \mathbf{O}(f) \Longrightarrow f \in \mathbf{O}(g)$

- Supongamos que $f \in \mathbf{O}(g) \wedge g \in \mathbf{O}(f)$
  - $f \in \mathbf{O}(g) \xrightarrow{Prop2} \mathrm{O(f)} \subset \mathbf{O}(g)$
  - $g \in \mathbf{O}(f) \xrightarrow{Prop2} \mathrm{O(g)} \subset \mathbf{O}(f)$
  - Podemos concluir que $\mathbf{O}(f) = \mathbf{O}(g)$.

4. Si $f \in \mathbf{O}(g) \wedge g \in \mathbf{O}(h) \Longrightarrow f \in \mathbf{O}(h)$
   Dem-
   - Si $f \in \mathbf{O}(g) \Longrightarrow f(n) \leq c_1 g(n) \quad \forall n_0$
   - Si $g \in \mathbf{O}(h) \Longrightarrow g(n) \leq c_2 h(n) \quad \forall n_1$
   - Entonces $f(n) \leq c_1 c_2 h(n) \forall n \geq max(n_0, n_1$
5. Si $f \in \mathbf{O}(g) \wedge f \in \mathbf{O}(h) \Longrightarrow f \in \mathbf{O}(min\{g,h\})$
6. **Regla de la Suma**.
   Si $f_1 \in \mathbf{O}(g)$ y $f_2 \in \mathbf{O}(h) \Longrightarrow f_1 + f_2 \in \mathbf{O}(max\{g,h\})$
7. **Regla del Producto**. Si $f_1 \in \mathbf{O}(g)$ y $f_2 \in \mathbf{O}(h) \Longrightarrow f_1 \cdot f_2 \in \mathbf{O}(\{g \cdot h\})$
8. Si existe $lim_{n\to\infty} \frac{f(n)}{g(n)} = k$ entonces se debe cumplir que:
   a) Si $k \neq 0$ y $k < \infty$ entonces $\mathbf{O}(f) = \mathbf{O}(g)$
   b) Si $k = 0$ entonces $f(n) \in \mathbf{O}(g)$ y $g(n) \notin \mathbf{O}(g) \Longrightarrow O(f) \subset O(g)$

Ejemplo 1.6.3
Dadas las siguiente funciones
- $f(n) = 5n^2 + log_2(n)$ y
- $g(n) = 8n^2$

Establecer el orden entre ellas dos.

Para resolver este ejercicio vamos a usar la **Propiedad 8**. Así tenemos que :

$$lim_{n\leftarrow\infty} \frac{5n^2 + log_2(n)}{8n^2} = \frac{\infty}{\infty}$$

Aplicando la Regla de L'Hopital: derivamos numerador y denominador y obtenemos de nuevo el límite:

$$lim_{n\leftarrow\infty} \frac{10n + \frac{1}{n}}{16n} = lim_{n\leftarrow\infty} \frac{10n^2+1}{16n^2} \Longrightarrow$$
$$lim_{n\leftarrow\infty} \frac{10n^2}{16n^2} = \frac{10}{16} > 0 \Longrightarrow \mathbf{O}(f) = \mathbf{O}(g)$$

□

Ejercicio 1.1
Sea $f(n) = 16n^3$ y $g(n) = 2n^2 + 5n$ ¿ Es $\mathbf{O}(f) = \mathbf{O}(g)$? □

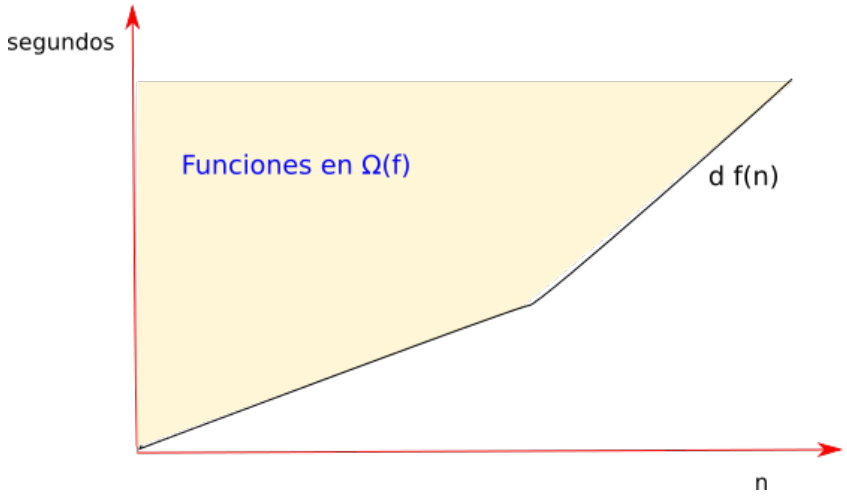
1.6.2 Cota Inferior. Notación Ω

En esta sección vamos a estudiar aquellas funciones g que a lo sumo crecen tan lentamente como f. En ese caso diremos que $g \in \Omega(f)$.

Definición 1.6.2 $\Omega(f)$ Sea $f : \mathbb{N} \to [0,\infty)$. Se define el conjunto de funciones de orden $\Omega(f)$ como:

$$\Omega(f) = \{g : \mathbb{N} \to [0,\infty) | \exists\ c \in R, c > 0, \exists n_0 \in \mathbb{N} \text{ tal que } g(n) \geq cf(n)\ \ \forall n \geq n_0\}$$

Son todas las funciones que quedan por encima de f, excepto por un número finito de puntos, incluyendo a ella misma. A continuación se muestra la región donde se encuentran las funciones en Ωf.



♠

Ejemplo 1.6.4
$f(n) = 5n^2 + log_2(n)$. Se cumplen las siguientes sentencias.

- $f(n) \in \Omega(n^2)$ es cierto ya que $5n^2 + log_2(n) \geq n^2\ \ \forall n_0$ con $n_0 = 1$ y $c = 1$
- $f(n) \in \Omega(log_2(n))$ es cierto ya que $5n^2 + log_2(n) \geq log_2(n) \forall n_0$ con $n_0 = 1$ y $c = 1$

Pero entre esas dos cotas inferiores escogeremos la mayor que es $\Omega(n^2)$.

□

Propiedades de la Notación Ω

A continuación se describe un conjunto de propiedades que cumple la notación Ω:

1. Para cualquier función $f \Longrightarrow f \in \Omega(f)$.
2. Si $f \in \Omega(g) \Longrightarrow \Omega(f) \subset \Omega(g)$
3. Si $\Omega(f) = \Omega(g) \Longleftrightarrow f \in \Omega(g) \wedge g \in \Omega(f)$
4. Si $f \in \Omega(g) \wedge g \in \Omega(h) \Longrightarrow f \in \Omega(h)$
5. Si $f \in \Omega(g) \wedge f \in \Omega(h) \Longrightarrow f \in \Omega(max\{g,h\})$
6. **Regla de la Suma**.
   Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Longrightarrow f_1 + f_2 \in \Omega(g+h) = \Omega(max(g,h))$
7. **Regla del Producto**. Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Longrightarrow f_1 \cdot f_2 \in \Omega(\{g \cdot h\})$
8. Si existe $lim_{n\to\infty} \frac{f(n)}{g(n)} = k$ entonces se debe cumplir que:

a) Si $k \neq 0$ y $k < \infty$ entonces $\Omega(f) = \Omega(g)$
b) Si $k = 0$ entonces $g(n) \in \Omega(f)$ y $f(n) \notin \Omega(g) \Longrightarrow \Omega(g) \subset \Omega(f)$

1.6.3 Orden Exacto Θ

El orden exacto se refiere al conjunto de funciones que crecen asíntoticamente de la misma forma

Definición 1.6.3 $\Theta(f)$

Sea $f : \mathbb{N} \to [0,\infty)$. Se define el conjunto de funciones de orden $\Theta(f)$ como:

$$\Theta(f) = \{g : \mathbb{N} \to [0,\infty) | \exists\, c,d \in R, c,d > 0, \exists n_0 \in \mathbb{N} \text{ tal que } cf(n) \leq g(n) \leq df(n) \ \ \forall n \geq n_0\}$$

Son todas las funciones que están tanto en $\mathbf{O}(f)$ y en $\Omega(f)$:

$$\Theta(f) = \mathbf{O}(f) \wedge \Omega(f)$$

Por ejemplo $\Theta(n)$ es el conjunto de todas las funciones que tiene un crecimiento asintótico es exactamente igual al polinonio n. De esta forma en $\Theta(n)$ estarian todas las funciones cuyo mayor término, eliminando constantes es n. Por ejmplo: $\Theta(n) = \{100n + \frac{1}{n}, 4n + 2\sqrt{n} + 3, 20, 3n + 2, 1log_2(n), \cdots\}$. Si miramos $\mathbf{O}(n)$ un ejemplo de funciones que estarían serán:

$$\mathbf{O}(n) = \{1, 5, \frac{1}{n}, log_2(n), 1{,}5log_2(n), 2^log_2(n), \cdots, \underbrace{100n + \frac{1}{n}, 4n + 2\sqrt{n} + 3, 20, 3n + 2, 1log_2(n), \cdots}_{\Theta(n)}\}$$

y un ejemplo de funciones en $\Omega(n)$ sería:

$$\Omega(n) = \{100n^2, 30nlog_2(n) + n, n^3, 2^n \cdots, \underbrace{100n + \frac{1}{n}, 4n + 2\sqrt{n} + 3, 20, 3n + 2, 1log_2(n), \cdots}_{\Theta(n)}\}$$

Claramente se puede observar que $\Theta(n) = \mathbf{O}(n) \wedge \Omega(n)$.

♠

Ejemplo 1.6.5

Suponed que $g(n) = 5n^2 + log_2(n)$ ¿ cuál es su orden exacto?

Sabemos que $g(n) \in \mathbf{O}(n^2)$ y que $g(n) \in \Omega(n^2)$ entonces podemos decir que $g(n) \in \Theta(n^2)$.

□

Propiedades de la Notación Θ

A continuación se describe un conjunto de propiedades que cumple la notación Θ:

1. Para cualquier función $f \Longrightarrow f \in \Theta(f)$.
2. Si $f \in \Theta(g) \Longrightarrow \Theta(f) = \Theta(g)$
3. Si $\Theta(f) = \Theta(g) \Longleftrightarrow f \in \Theta(g) \wedge g \in \Theta(f)$
4. Si $f \in \Theta(g) \wedge g \in \Theta(h) \Longrightarrow f \in \Theta(h)$

5. **Regla de la Suma**.
   Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h) \Longrightarrow f_1 + f_2 \in \Theta(max\{g,h\})$
6. **Regla del Producto**. Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h) \Longrightarrow f_1 \cdot f_2 \in \Theta(\{g \cdot h\})$
7. Si existe $lim_{n\to\infty} \frac{f(n)}{g(n)} = k$ entonces se debe cumplir que:
   a) Si $k \neq 0$ y $k < \infty$ entonces $\Theta(f) = \Theta(g)$
   b) Si $k = 0$ entonces los órdenes exactos de $f(n)$ y $g(n)$ son distintos.

1.6.4 Casos de los tiempo de ejecución:Acotación

La asociación que haremos para acotar los tiempos de ejecución en cada uno de sus casos: peor , mejor y promedio se realizará de la siguiente forma:

- *Tiempo de ejecución en el peor caso* $T_p(n)$. Lo acotaremos con **O**-grande.
- *Tiempo de ejecución en el mejor caso* $T_m(n)$. Lo acotaremos con Ω
- *Tiempo de ejecución en el caso medio* $T_{\frac{1}{2}}(n)$. Lo acotaremos con Θ.

1.7 Ejercicios

Ejercicio 1.2

Encontrar dos funciones $f(n)$ y $g(n)$ tales que $f \in O(g)$ y $g \notin O(f)$
Por ejemplo sea $f(n) = n$ y

$$g(n) = \begin{cases} n^2 & \text{si n es par} \\ 1 & \text{si n es impar} \end{cases}$$

□

Ejercicio 1.3

$T_1 \in \mathbf{O}(f)$ y $T_2 \in \mathbf{O}(f)$. Indicar cuales de las siguientes afirmaciones son ciertas.

1. $T_1 + T_2 \in \mathbf{O}(f)$. Es cierta
   - Si $T_1 \in \mathbf{O}(f) \Longleftrightarrow \exists c_1$ y $\forall n_0 T_1(n) \leq c_1 f(n)$
   - Si $T_2 \in \mathbf{O}(f) \Longleftrightarrow \exists c_2$ y $\forall n_1 T_2(n) \leq c_2 f(n)$
   - sea $c = c_1 + c_2$ y $n_2 = max\{n_0, n_1\} \Longleftrightarrow T_1(n) + T_2(n) \leq c f(n) \forall n \geq n_2$
2. $T_1 + T_2 \in \mathbf{O}(f)$. Es cierta

$$lim_{n\to\infty} \frac{T_1(n)}{f(n)} = k_1 < \infty$$
y
$$lim_{n\to\infty} \frac{T_2(n)}{f(n)} = k_2 < \infty$$
entonces
$$lim_{n\to\infty} \frac{T_1(n) - T_2(n)}{f(n)} = lim_{n\to\infty} \frac{T_1(n)}{f(n)} - lim_{n\to\infty} \frac{T_2(n)}{f(n)} = k_1 - k_2 < \infty$$

3. $\frac{T_1}{T_2} \in O(1)$ es falso. Por ejemplo sea $T_1(n) = n^2$ y $T_2(n) = n$ y $f(n) = n^3$ entonces $\frac{T_1(n)}{T_2(n)} = n$
4. $T_1 \in \mathbf{O}(T_2)$ es falso. Por ejemplo sea $T_1(n) = n^2$ y $T_2(n) = n$ y $f(n) = n^3$

□

Ejercicio 1.4

Dado el siguiente algoritmo obtener los tiempos de ejecución en el mejor, peor y promedio. Y acotar cada tiempo asintóticamente.

```
1   int Algoritmo2(int a[], int n, int c){
2     int inf=0;int sup=n-1;
3     while (sup>=inf){
4      i=(inf+sup)/2;
5      if (a[i]==c) return i;
6      else
7       if (c<a[i]) sup = i-1;
8       else inf=i+1;
9      }
10     return 0;
11  }
```

Claramente este algoritmo representa la búsqueda binaria, implementada de forma iterativa.

CASO MEJOR. El caso mejor ocurre cuando se ejecutan las instrucciones (2-5), y se debe a que el elemento que buscamos está en la mitad.

$$T_m(n) = \underbrace{3}_{linea2} + \underbrace{1}_{linea3} + \underbrace{3}_{linea4} + \underbrace{3}_{linea5} = 10$$

En este caso podemos decir que $T_m(n) \in \Omega(1)$.

CASO PEOR.El caso peor ocurre cuando elemento c no está en el vector. En este caso el numero de veces que se ejecuta el while se corresponde con el numero de veces que podemos dividir por dos n, que equivale al $log_2(n)$.

$$T_p(n) = \underbrace{3}_{linea2} + \underbrace{1}_{linea3} +$$
$$(\sum_{i=1}^{log_2(n)} 3+2+2+1)+1 = 5+10log_2(n)$$

Por lo tanto $T_p(n) \in \mathbf{O}(log_2(n))$.

CASO MEDIO.Para obtener el tiempo de ejecución $T_{\frac{1}{2}(n)}$ se necesita calcular el número medio de veces que se repite el bucle while, definiendo qué probabilidad tiene cada iteración de suceder. El bucle puede repetirse desde 1 hasta $log_2(n)$ veces puesto que en cada iteración se divide por dos el número de elementos considerados. Si se repite una sola vez el elemento ocuparía la posición $\frac{n}{2}$, lo que ocurre con probabilidad $\frac{1}{n+1}$ (dividimos por $n+1$ para contabilizar tambien que no esté). Si se repite dos veces entonces el elemento está en $n/4$ o $3n/4$ lo que ocurre con probabilidad $\frac{2}{n+1}$

En general para i-iteaciones el elemento podría estar en 2^{i-1} posiciones. Luego la probabilidad de hacer i iteraciones sería $\frac{2^{i-1}}{n+1}$

Por lo tanto el número medio de veces que se repite el ciclo es:

$$\sum_{i=1}^{log_2(n)} i\frac{2^{i-1}}{n+1} =$$

$$\frac{1}{n+1}(1\cdot 2^0 + 2\cdot 2^1 + 3\cdot 2^2 + \cdots + log_2(n)\cdot 2^{log_2(n)-1}) =$$

$$\frac{1}{n+1}(nlog_2(n) - n + 1)$$

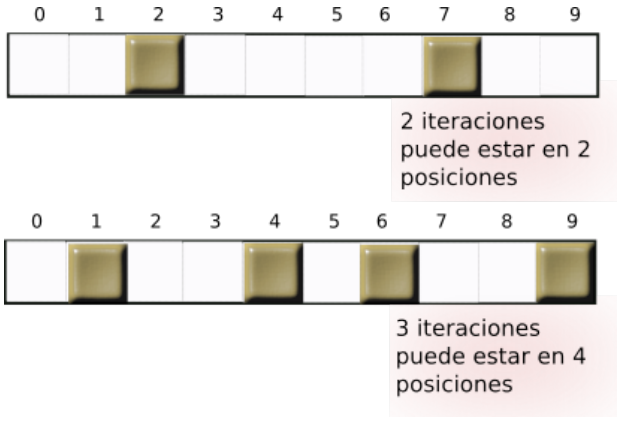


Figura 1.2: Ejemplo de las posiciones candidatas donde puede estar el elemento que se busca en 2 y 3 iteraciones

Por lo tanto el número medio de veces que se repite el bucle es $\frac{nlog_2(n)-n+1}{n+1}$. Así

$$T_{\frac{1}{2}}(n) = 3+1 + \underbrace{\frac{nlog_2(n)-n+1}{n+1}}_{num\_iteraciones} \times (1+3+2+2) + \underbrace{1+3+3}_{ultima\ iteracion} =$$

$$11+8\,\frac{nlog_2(n)-n+1}{n+1}$$

Con esto podemos concluir que $T_{1/2}(n) \in \Theta(log_2(n))$.

□

1.8 Resolución con ecuaciones recurrentes.

En esta sección estudiaremos como expresar la eficiencia de algoritmos recursivos. Por ejemplo los algoritmos para obtener el factorial de un entero, el algoritmo de fibonacci o el algoritmo de Hanoi [1] , como se ve a continuación, son ejemplos dealgoritmos recursivos:

```
1  int fibonacci(int n){
2   if (n==0 ) return 0;
3   else if (n==1) return 1;
4   else return (fibonacci(n-1)+fibonacci(n-2));
5  }
```

[1] **El algoritmo de Hanoi** consiste en dados n discos de difentes diámetros, y todos inicialmente puestos en un palo de forma descendente por diámetro, el objetivo es pasarlos todos a otro de los palos, siempre manteniendo que los discos se coloquen de forma descendente en diámetro. Se puede usar para ello otro palo intermedio.

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

```
//suponiendo 3 palos que se etiquetan como 1,2,3
 int Hanoi(int n,int p1,int p2){
  if (n>0 ){
   Hanoi(n-1,p1,6-p1-p2);
   cout<<''Mover 1 de ''<<p1<<'a'<<p2<<endl;
   Hanoi(n-1,6-p1-p2,p2);

 }
}
```

Para resolver la eficiencia de estos algoritmos tenemos que distinguir el tipo de ecuación que se deduce, distinguiendo entre:

- Recurrencia Homogenea: por ejemplo el algoritmo de Fibonacci
- Recurrencia no Homogenea: por ejemplo los algoritmos de Hanoi, o el factorial.

1.8.1 Recurrencia Homogenea

Se caracterizan por ser de la forma:

$$a_0 T(n) + a_1 T(n-1) + \cdots + a_k T(n-k) = 0 \tag{1.7}$$

Siendo $a_i \in \mathbb{R}$ y $k \in [1,n]$. Y $T(n)$ es el tiempo de ejecución del algoritmo para un talla n.
Forma de Resolver Se hace el siguiente cambio de variable $x^k = T(n)$, de forma que la ecuación 1.7 queda como:

$$a_0 x^k + a_1 x^k - 1 + \cdots + a_k x^0 = 0; \tag{1.8}$$

La solución a la ecuación 1.8 tiene como solucion k raices:

$$(r_1, r_2, \cdots, r_k)$$

Por lo tanto la ecuación 1.8 podemos expresarla como:

$$(x - r_1)(x - r_2)\cdots(x_k - r_k) = 0$$

De esta forma el tiempo de ejecución se expresa como:

$$T(n) = \sum_{i=1}^{k} c_i p_i(n) r_i^n \tag{1.9}$$

siendo r_i números reales correspondientes a las raices, $p_i(n)$ polinomios asociados a las raices y ci numeros reales. Dentro de las ecuaciones recurrentes podemos distinguir varios casos:

1. Todas las raices r_i son diferentes
2. Exiten raices con multiplicidad mayor que 1.

1.8.2 Caso 1. Recurrencias Homogeneas con k raíces distintas

La ecuación que tenemos que revolcer en este caso es:

$$T(n) = \sum_{i=1}^{k} c_i r_i^n \tag{1.10}$$

Los coeficientes c_i se calculan usando las condiciones iniciales (casos bases de la recurrencia).

Ejemplo 1.8.1
Resolvamos la eficiencia del algoritmo de Fibonacci.

$$T(n) = \begin{cases} n & sin = 0 \text{ o } n = 1 \\ T(n-1)+T(n-2) & n > 1 \end{cases} \tag{1.11}$$

El tiempo de ejecución para $n = 0$ o $n = 1$ tiene tiempos de ejecución $T(0) = 0$ o $T(1) = 1$. En otro caso el tiempo de ejecución es la suma de los tiempos de ejecución para $n-1$ y $n-2$. Así que vamos a resolver esta segunda parte:

$$T(n) = T(n-1)+T(n-2) \qquad n > 1$$

no lo llevamos todo a la parte izquierda

$$T(n) - T(n-1) - T(n-2) = 0 \qquad n > 1$$

como $k = 2$ hacemos el cambio de variable $T(n) = x^2$

$$x^2 - x - 1 = 0$$

$$x = \frac{1 \pm \sqrt{1+4}}{2} \qquad \text{con soluciones}$$

$$x_1 = \frac{1+\sqrt{5}}{2} \text{ y } x_2 = \frac{1-\sqrt{5}}{2}$$

de esta forma ahora $T(n)$ se puede escribir como

$$T(n) = c_1(\frac{1+\sqrt{5}}{2})^n + c_2(\frac{1-\sqrt{5}}{2})^n$$

A continuación tenemos que obtener los valores para c_1 y c_2 usando las condiciones iniciales (casos bases en la ecuación). Como tenemos dos constantes nos hará falta saber dos valores concretos para $T(n)$.

$$(1)T(0) = c_1 + c_2 = 0$$
$$(2)T(1) = c_1(\frac{1+\sqrt{5}}{2})^n + c_2(\frac{1-\sqrt{5}}{2})^n = 1$$

De (1) obtenemos que $c_1 = -c_2 \Longrightarrow c_2 = -c1$, sustituyendo en (2):

$$T(1) = c_1(\tfrac{1+\sqrt{5}}{2})^n + c_1(\tfrac{-1+\sqrt{5}}{2})^n = 1 \Longrightarrow$$
$$\sqrt{5}c_1 = 1 \Longrightarrow c_1 = \tfrac{1}{\sqrt{5}} \text{ y } c_2 = -\tfrac{1}{\sqrt{5}}$$

Por lo tanto el tiempo de eficiencia $T(n)$ lo podemos expresar como:

$$T(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Cuando $n \to \infty$ el término $\left(\frac{1-\sqrt{5}}{2}\right)^n$ tiende a 0. Por lo tanto podemos decir que $T(n) \in \mathbf{O}\left(\frac{1-\sqrt{5}}{2}\right)^n$

□

Ejemplo 1.8.2
Dado el siguiente código

```
1  int Suma(int *v, int n){
2   if (n==1)
3    return v[0];
4
5    else
6    return Suma(v,n/2)+Suma(v+n/2,n-n/2);
7   }
```

Obtener el tiempo de ejecución en términos de **O**-grande.

A continuación vamos a rescribir el código con cometarios acerca del numero de operaciones elementales que realiza:

```
1  int Suma(int *v, int n){// ← T(n)
2   if (n==1)
3    return v[0];// ← 1
4    else
5    return Suma(v,n/2)+Suma(v+n/2,n-n/2);//← T(n/2)+T(n/2)
6   }
```

Hemos simplificado el tiempo de ejecución cuando $n == 1$ con un coste de 1. Con este detalle en el código escribimos la ecuación de recurrencia para el tiempo de ejecución:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(\frac{n}{2}) & n > 1 \end{cases}$$

Empezamos a resolver la ecuación:

$$T(n) = 2T(\tfrac{n}{2})\ n > 1$$
$$T(n) - 2T(\tfrac{n}{2}) = 0 \leftarrow \text{ Ecuación recurrente homogenea}$$

Hacemos el cambio de variable $n = 2^m$ y sustituimos

$$T(2^m) - 2T(\tfrac{2^m}{2}) = 0 \Longrightarrow T(2^m) - 2T(2^{m-1}) = 0$$

teniendo en cuenta que $k = 1$ hacemos el cambio de variable $x = T(2^m)$

$$x - 2 = 0 \text{ la raíz de este monomio es } x_1 = 2 \Longrightarrow T(2^m) = c_1 2^m$$
deshaciendo el cambio de variable $n = 2^m$

$$T(n) = c_1 n$$

A continuación usando las condiciones iniciales calculamos el valor de c_1:

$$T(1) = c_1 = 1 \Longrightarrow T(n) = n$$

Por lo tanto $T(n) \in \mathbf{O}(n)$

□

1.8.3 Caso 2. Recurrencias Homogeneas con k raíces, algunas repetidas

En este caso se estudia como resolver la ecuación de recurrencia cuando tenemos k raíces pero algunas tienen multiplicidad mayor que 1. Por ejemplo si r_1 tuviese multiplicidad m, tenemos que:

$$(x - r_1)^m (x - r_2) \cdots (x_k - r_{m+1}) = 0$$

y por lo tanto el tiempo de ejecución se puede expresar como

$$T(n) = \underbrace{\sum_{i=1}^{m} c_1 n^{i-1} r_1^n}_{\text{por } r_1 \text{ que tiene multiplicidad } m} + \sum_{i=m+1}^{k} c_i r_{i-(m+1)}^n$$

Como se puede observar el primer sumatorio incluye polinomios del tipo n^{i-1} para incluir que el hecho de que r_1 aparece con multiplicidad.

Ejemplo 1.8.3

$$T(n) = \begin{cases} 5T(n-1) - 8T(n-2) + 4T(n-3) & n > 2 \\ n & n \leq 2 \end{cases}$$

Resolviendo para $n > 2$

$$\begin{aligned} T(n) &= 5T(n-1) - 8T(n-2) + 4T(n-3) \\ T(n) - 5T(n-1) + 8T(n-2) - 4T(n-3) &= 0 \end{aligned} \tag{1.12}$$

haciendo el cambio de variable $x^3 = T(n)$

$$x^3 - 5x^2 + 8x - 4 = 0$$

Para resolver esta ecuación podemos usar Ruffini:

	1	-5	8	-4
1		1	-4	4
	1	-4	4	0
2		2	-4	
	1	-2	0	

Tenemos por lo tanto como raices $r_1 = 1$ con multiplicidad 1 y $r_2 = 2$ con multiplicidad 2:

$$T(n) = (x-1)(x-2)^2$$

$$T(n) = c_1 1^n + c_2 2^n + \underbrace{c_3 n 2^n}_{\text{por la multiplicidad } n^{2-1}}$$

A continuación usando las condiciones iniciales obtenemos las constantes.

$$\begin{aligned} T(0) &= c_1 + c_2 = 0 && (1) \\ T(1) &= c_1 + 2c_2 + 2c_3 = 1 && (2) \\ T(2) &= c_1 + 4c_2 + 8c_3 = 2 && (3) \end{aligned}$$

de (1) podemos llegar a $c_1 == -c_2$ sutituyendo en (2)

$c_2 + 2c_3 = 1 \Longrightarrow c_2 = 1 - 2c_3$ y $c_1 = 2c_3 - 1$ sutituyendo en (3)

$2c_3 + 3 = 2 \Longrightarrow c_3 = -\frac{1}{2}$, $c_1 = -2$ y $c_2 = 2$

luego $T(n)$ queda como $T(n) = -2 \cdot 1^n + 2 \cdot 2^n - \frac{1}{2} n \cdot 2^n \Longrightarrow$

$$T(n) = 2^{n+1} - n2^n - 1 - 2$$

La cuestión ahora es ¿ quién es más grande: $2^n + 1$ o $n2^n - 2$?. Para verlo vamos a resolver :

$$\begin{aligned} 2^{n+1} &= cn2^{n-1} \\ cn = \tfrac{2^n+1}{2^{n-1}} &= 4 \Longrightarrow n_0 = 4 \text{ y } c = 1 \end{aligned}$$

Luego se cumple que $n \cdot 2^{n-1} > 2^{n+1}$ por lo tanto $T(n)$ es negativo !!. Como conclusión la ecuación de partida $T(n)$ no se corresponde con el tiempo de ejecución de un algoritmo. Otra alternativa es que la

recurrencia ha sido mal calculada pero este no es el caso. Podemos decir que tenemos una función recurrente (matemáticamente correcta) que no representa la función de ejecución de un algoritmo (el término $8T(n-2)$ de la función recurrente original no se puede obtener a partir de un algoritmo).

□

1.8.4 Recurrencias No Homogeneas

La forma de la recurrencia no homogenea es la siguiente:

$$a_0T(n)+a_1T(n-1)+\cdots+a_kT(n-k)=b^n\cdot p(n)$$

La ecuación característica sería:

$$(a_0x^k+a_1x^{k-1}+\cdots+a_k)(x-b)^{d+1}=0$$

Ejemplo 1.8.4
Supongamos que tenemos la siguiente ecuación recurrente:

$$T(n)=\begin{cases}0 & \text{si } n=0\\ 2T(n-1)+1 & \text{en otro caso}\end{cases}$$

Veamos el desarrollo de la ecuación $T(n)=2T(n-1)+1$ cuando $n\geq 1$

$$\begin{gathered}T(n)=2T(n-1)+1\\ T(n)-2T(n-1)=1 \text{ haciendo el cambio de variable } x=T(n)\\ (x-2)=1\Longrightarrow b=1 \text{ y } p(n)=1 \text{ luego el grado } d=0\\ (x-2)(x-1)=0\Longrightarrow \text{ ecuación característica}\\ T(n)=c_12^n+c_21^n\end{gathered}$$

Obtenemos las constantes c_1 y c_2 usando la condición inicial

$$T(0)=c_1+c_2=0\Longrightarrow c_1=-c_2$$

Nos hace falta otra ecuación para poder despejar c_1 y c_2, que la obtenemos desde la forma general:

$$\begin{gathered}T(1)=2T(0)+1=2\cdot 0+1=1\Longrightarrow\\ \text{sustityendo en } T(n)=c_12^n+c_21^n\\ T(1)=c_1*2+c_2=1\Longrightarrow\\ \text{usando que } c_1=-c_2\\ T(1)=c_1*2-c_1=1\Longrightarrow c_1=1 \text{ y } c_2=-1\\ T(n)=2^n-1^n\in O(2^n)\end{gathered}$$

Esta recurrencia se puede ver en algoritmos basados en la técnica divide y vencerás en el que el problema original se divide en dos partes. Cada una de esas partes se resuelven y para obtener la solución a nuestro problema original, se fusiona las dos soluciones mitad en tiempo constate.

□

Ejemplo 1.8.5
Supongamos que tenemos la siguiente ecuación recurrente:

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 2T(n-1)+n & \text{en otro caso} \end{cases}$$

La diferencia con el anterior algoritmo es en el término general (en otro caso) que se ha sustituido el 1 por n. Al igual que la anterior recurrencia esta recurrencia puede aparecer en algoritmos basados en la técnica divide y vencerás en el que el problema original se divide en dos partes. Cada una de esas partes se resuelven y para obtener la solución a nuestro problema original, se fusiona las dos soluciones mitad en tiempo lineal n. Por ejemplo el algoritmo de ordenación *Mergesort* parte de esta recurrencia. Para resolverla sigamos los siguientes pasos:

$$T(n) - 2T(n-1) = n \text{ podemos deducir los términos} de la parte de derecha b = 1 \quad p(n) = n \quad d = 1$$
$$\text{haciendo el cambio de variable } x = T(n)$$
$$(x-2)(x-1)^2 = 0 \Longrightarrow$$
$$T(n) = c_1 \cdot 2^n + c_2 \cdot 1^n + c_3 \cdot n \cdot 1^n = 0$$

Usando las condiciones iniciales podemos despejar las constantes c_1, c_2 y c_3:

$$\begin{array}{l} (1)\ \ T(0) = c_1 + c_2 = 0 \Longrightarrow c_1 = -c_2 \\ (2)\ \ T(1) = 1 = 2c_1 + c_2 + c_3 \\ (3)\ \ T(2) = 4 = 4c_1 + c_2 + c_3 \end{array}$$

$$\begin{array}{c} \text{de (2) haciendo } c_1 = -c_2 \Longrightarrow T(1) = 1 = c_1 + c_{33} = 1 - c_1 \\ T(2) = 4 = c_1 + 2 \Longrightarrow c_1 = 2, c_2 = -2, c_3 = -1 \\ \text{por lo tanto} \\ T(n) = 2 \cdot 2^n + (-2) \cdot 1^n + (-1) \cdot n \cdot 1^n \in \mathbf{O}(2^n) \end{array}$$

□

1.8.5 Generalizacion de las Recurrencias No Homogeneas

En general las recuencias no homogenes en la parte derecha puede tener diferentes sumando de términos $b_i^n p_i(n)$, de la siguiente forma:

$$a_0 T(n) + a_1 T(n-1) + \cdots + a_k T(n-k) = b_1^n \cdot p_1(n) + b_2^n \cdot p_2(n) + \cdots + b_s^n \cdot p_s(n)$$

con ecuación característica:

$$(a_0 x^k + a_1 x^{k-1} + \cdots + a_k) + (x-b_1)^{d_1+1}(x-b_2)^{d_2+1} \cdots (x-b_s)^{d_s+1} = 0$$

Ejemplo 1.8.6
Supongamos la siguiente ecuación de recurrencia:

$$T(n) = 2T(n-1) + n + 2^n \quad n \geq 1 \quad T(0) = 1$$

Esta ecuación es una ecuación no homogenea con más de un término no homogeneo n y 2^n.

$$T(n)-2T(n-2)=n+2^n \text{ tenemos dos polinomios: } \begin{cases} p_1(n)=n \text{ con } d_1=1 \text{ y } b_1=1 \\ p_2(n)=1 \text{ con } d_2=0 \text{ y } b_2=2 \end{cases}$$

$$\text{haciendo el cambio de variable } x=T(n) \Longrightarrow$$
$$(x-2)(x-1)^2(x-2)=0 \Longrightarrow (x-2)^2(x-1)^2=0$$
$$T(n)=c_1\cdot 1^n+c_2\cdot n\cdot 1^n+c_3\cdot 2^n+c_4\cdot n\cdot 2^n$$

Para poder despejar las constantes necesitamos 4 valores: $T(0)=1, T(1)=5, T(2)=16, T(3)=43$, con estos valores planteamos las ecuaciones correspondientes:

$$\begin{aligned} (1)\ & 1=c_1+c_3 \\ (2)\ & 5=c_1+c_2+2c_3+2c_4 \\ (3)\ & 16=c_1+2c_2+4c_3+8c_4 \\ (4)\ & 43=c_1+3c_2+8c_3+24c_4 \end{aligned}$$

resolviendo para despejar las c's tenemos $c_1=-2, c_2=-1, c_3=3, c_4=1$. Por lo tanto el tiempo de ejecución:

$$T(n)=2+3n+(-2)\cdot 2^n+n\cdot 2^{n+1} \in \mathbf{O}(n\cdot 2^n).$$

□

1.8.6 Recurrencias: Ejemplos

Cambio de Variable

A continuación vamos a ver un conjunto de ejemplo en el que necesitamos estudiar bien el cambio de variable para desencadenar una recurrencia que se exprese de forma lineal con la nueva variable, y poder aplicar los pasos vistos en las secciones anteriores.

Ejemplo 1.8.7
Resolver la siguiente recurrencia:

$$T(n)=\begin{cases} 0 & n=0 \\ 3 & n=1 \\ 5T(n-1)+6T(n-2)+4\cdot 3^n & n>1 \end{cases}$$

Esta ecuación se corresponde con una ecuación recurrente no homogenea:

$$T(n)-5T(n-1)-6T(n-2)=4\cdot 3^n \Longrightarrow \begin{cases} p(n)=4 & d=0 \\ b=3 \end{cases}$$

haciendo el cambio de variable $x^2=T(n) \Longrightarrow$

$$(x^2-5x-6)(x-3)=0$$

resolviendo la ecuación de segundo grado $\Longrightarrow$

$$x=\frac{5\pm\sqrt{25+24}}{2} \Longrightarrow \begin{cases} x_1=6 \\ x_2=-1 \end{cases}$$

$$(x-6)(x+1)(x-3)=0$$
$$T(n)=c_1\cdot 6^n+c_2\cdot(-1)^n+c_3\cdot 3^n$$

Formulamos el sistema de ecuaciones para despejar las constantes c's:

$$(1)\ \ T(0)=0=c_1+c_2+c_3$$
$$(2)\ \ T(1)=36=6c_1-c_2+3c_3$$
$$(3)\ \ T(2)=216=5T(1)+6T(0)+4\cdot 9 \Longrightarrow T(2)=216=c_1\cdot 36+c_2+9\cdot c_3$$

Despejando el sistema de ecuaciones obtenemos que $c_1=\frac{48}{7}, c_2=-\frac{27}{7}, c_3=-3$ Por lo tanto el tiempo de ejecución:

$$T(n)=\frac{48}{7}\cdot 6^n+-\frac{27}{7}\cdot(-1)^n+(-3)\cdot 3^n$$

Podemos afirmar que $T(n)\in \mathbf{O}(6^n)$.

□

Ejemplo 1.8.8
En este ejemplo queremos afianzar el concepto de cambio de variable. Para ello vamos a tener en cuenta la siguiente propiedad del logaritmo:

$$log_a x^y = y log_a x$$

Resolver la ecuación recurrente:

$T(n)=4T(\frac{n}{2})+n$ para $n\geq 2$ y n es potencia de 2

Este tipo de recurrencia nos indica que realiza cuatro llamadas recursivas de tamaño la mitad de n y la fusion de los resultados obtenidos le cuesta n.

$$T(n) - 4T(\tfrac{n}{2}) = n$$
hacemos el cambio de variable $n = 2^m$
$$T(2^m) - 4T(2^{m-1}) = 2^m \Longrightarrow \begin{cases} x = T(2^m) \\ b = 2 \\ p(m) = 1 \\ d = 0 \end{cases}$$

$$(x-4)(x-2) = 0$$
$$T(2^m) = c_1 4^m + c_2 \cdot 2^m$$
deshaciendo el cambio de variable $n = 2^m$ y sabiendo que $4^m = (2^m)^2$
$$T(n) = c_1 \cdot n^2 + c_2 \cdot n$$

En este ejemplo como no tenemos caso base debemos suponer que $T(n)$ representa una función de tiempo y tiene que ser positiva para n. Por lo tanto c_1 debe ser positivo y asintóticamente $T(n) \in \mathbf{O}(n^2)$ siendo n potencia de dos.

□

Ejemplo 1.8.9
Suponed la modificación de la anterior recurrencia a :

$$T(n) = 4T(\frac{n}{2}) + n^2 \text{ para } n \geq 2 \text{ y } n \text{ es potencia de 2}$$

La diferencia es que fusionar los resultados de las llamadas recursivas le cuesta n^2.

$$T(n) - 4T(\tfrac{n}{2}) = n^2$$
hacemos el cambio de variable $n = 2^m$
$$T(2^m) - 4T(2^{m-1}) = 4^m \Longrightarrow \begin{cases} x = T(2^m) \\ b = 4 \\ p(m) = 1 \\ d = 0 \end{cases}$$

$$(x-4)(x-4) = 0 \Longrightarrow (x-4)^2 = 0$$
$$T(2^m) = c_1 4^m + c_2 \cdot m \cdot 4^m$$
deshaciendo el cambio de variable $n = 2^m$ y sabiendo que $4^m = (2^m)^2$
$$T(n) = c_1 \cdot n^2 + c_2 \cdot n^2 \cdot log_2(n)$$

Suponiendo que $T(n)$ es una función de tiempo (positiva) implica que c_2 debe ser positiva y por lo tanto $T(n) \in \mathbf{O}(n^2 log_2(n))$

□

Ejemplo 1.8.10
Resolved la siguiente ecuación de recurrencia:

$$T(n) = 2T(\frac{n}{2}) + n \cdot log_2(n) \text{ para } n \geq 2 \text{ y } n \text{ es potencia de 2}$$

$T(n)-2T(\frac{n}{2})=n\cdot log_2(n)$hacemos el cambio de variable $n=2^m$

$$T(2^m)-2T(2^{m-1})=2^m\cdot m \Longrightarrow \begin{cases} x=T(2^m) \\ b=2 \\ p(m)=m \\ d=1 \end{cases}$$

$$(x-2)(x-2)^2=0 \Longrightarrow (x-2)^3=0$$
$$T(2^m)=c_1 2^m+c_2\cdot m\cdot 2^m+c_3\cdot m^2\cdot 2^m$$

deshaciendo el cambio de variable $n=2^m$ y sabiendo que

$$T(n)=c_1\cdot n+c_2\cdot n\cdot log_2(n)+c_3\cdot n\cdot log_2(n)^2$$

Por lo tanto podemos afirmar que $T(n)\in \mathbf{O}(n\cdot log_2(n)^2)$.

□

Ejemplo 1.8.11
Suponed que tenemos la siguiente recurrencia:

$$T(n)=2T(\sqrt{n})+log_2(n) \text{ para } n\geq 4 \text{ y } T(2)=1$$

Esta ecuación exige un cambio de variable del tipo $n=2^{2^m} \Longrightarrow \sqrt{n}=n^{\frac{1}{2}}=(2^{2^m})^{\frac{1}{2}}=2^{2^{m-1}}$

$$T(2^{2^m})=2T(2^{2^{m-1}})+2^m$$
$$T(2^{2^m})-2T(2^{2^{m-1}})=2^m \Longrightarrow \begin{cases} x=T(2^{2^m}) \\ b=2 \\ p(m)=1 \\ d=0 \end{cases}$$

$$(x-2)(x-2)=0 \Longrightarrow (x-2)^2=0$$
$$T(2^{2^m})=c_1 2^m+c_2 m\cdot 2^m$$

deshaciendo el cambio de variable $n=2^{2^m}$ y sabiendo que $log_2(n)=2^m$ y $log_2(log_2(n))=m$

$$T(n)=c_1\cdot log_2(n)+c_2 log_2(log_2(n))\cdot log_2(n)$$

En este caso como tenemos condiciones iniciales podemos obtener los valores para las c's:

$$T(2)=1=c_1$$
$$T(4)=2T(2)+2=4 \Longrightarrow T(4)=4=2\cdot c_1+2c_2$$
$$4=2+2c_2 \Longrightarrow c_2=1$$

Por lo tanto $T(n)=log_2(n)+log_2(log_2(n))\cdot log_2(n)\in O(log_2(log_2(n))\cdot log_2(n))$.

□

Ejemplo 1.8.12

Dada la siguiente recurrencia:

$$T(n) = 5T(\frac{n}{2}) + (n \cdot log_2(n))^2 \text{ para } n \geq 4 \text{ y } T(1) = 1$$

El cambio de variable que debemos adoptar es $n = 2^m \Longrightarrow log_2(n) = m$:

$$T(2^m) = 5T(2^{m-1}) + (2^m)^2 \cdot m^2$$

$$T(2^m) = 5T(2^{m-1}) + (4^m) \cdot m^2 \Longrightarrow \begin{cases} x = T(2^m) \\ b = 4 \\ p(m) = m^2 \\ d = 2 \end{cases}$$

$$(x-5)(x-4)^3 = 0$$
$$T(2^m) = c_1 \cdot 5^m + c_2 \cdot 4^m + c_3 \cdot m \cdot 4^m + c_4 \cdot m^2 \cdot 4^m$$

deshaciendo el cambio de variable
$$T(n) = c_1 \cdot 5^{log_2(n)} + c_2 \cdot n^2 + c_3 \cdot log_2(n) \cdot n^2 + c_4 \cdot log_2(n)^2 \cdot n^2$$
teniendo en cuenta que $5^{log_2(n)} = n^{log_2(5)}$podemos poner

$$T(n) = c_1 \cdot n^{log_2(5)} + c_2 \cdot n^2 + c_3 \cdot log_2(n) \cdot n^2 + c_4 \cdot log_2(n)^2 \cdot n^2$$

Para obtener las constantes:

$$T(1) = 1 = c_1 + c_2$$
$$T(2) = 9$$
$$T(4) = 5 \cdot 9 + 8^2 = 109 T(8) = 5 \cdot 109 + 24^2 = 1121$$
el sistema de ecuaciones sería

$$n = 1 \Longrightarrow 1 = c_1 + c_2 \Longrightarrow c_1 = 1 - c2$$
$$n = 2 \Longrightarrow 9 = 5c_1 + 4c_2 + 4c_3 + 4c_4$$
$$n = 4 \Longrightarrow 109 = 5^2 c_1 + 4^2 c_2 + 24^2 c_3 + 2^2 \cdot 4^2 c_4$$
$$n = 8 \Longrightarrow 1121 = 5^3 c_1 + 8^2 c_2 + 38^2 c_3 + 3^2 \cdot 8^2 c_4$$

Resolviendo el sistema de ecuaciones obtenemos $c_1 = 181, c_2 = -180, c_3 = -40, c_4 = -4$. Teniendo en cuenta que $log_2(5) = 2{,}32$ y es mayor que cualquier otro término. Podemos decir que:

$$T(n) \in \mathbf{O}(n^{2,32})$$

□

Ejemplo 1.8.13
Resolver la siguiente ecuación:

$$T(n) = 3T(\frac{n}{2}) - 2T(\frac{n}{4}) + log_2(n) \text{ para } n \geq 4 \text{ y } T(1) = T(2) = 3$$

El cambio de variable que debemos adoptar es $n = 2^m \Longrightarrow log_2(n) = m$:

$$T(2^m) = 3T(2^{m-1}) + 2T(2^{m-2}) + m$$

$$T(2^m) - 3T(2^{m-1}) - 2T(2^{m-2}) = m \Longrightarrow \begin{cases} x^2 = T(2^m) \\ b = 1 \\ p(m) = m \\ d = 1 \end{cases}$$

$$(x^2 - 3x + 2)(x-1)^2 = 0 \Longrightarrow (x^2 - 3x + 2) = 0 \Longrightarrow x = \frac{3 \pm \sqrt{9-8}}{2} \begin{cases} x_1 = 2 \\ x_2 = 1 \end{cases}$$

$$(x-2)(x-1)^3 = 0$$
$$T(2^m) = c_1 \cdot 2^m + c_2 \cdot 1^m + c_3 \cdot m \cdot 1^m + c_4 \cdot m^2 \cdot 1^m$$

deshaciendo el cambio de variable
$$T(n) = c_1 \cdot n + c_2 + c_3 \cdot log_2(n) + c_4 \cdot log_2(n)^2$$

ara obtener las constantes:

$$n = 1 \Longrightarrow 3 = c_1 + c_2 \Longrightarrow c_1 = 3 - c2$$
$$n = 2 \Longrightarrow 3 = 2c_1 + c_2 + c_3 + c_4$$
$$n = 4 \Longrightarrow 5 = 5c_1 + 2c_2 + 2c_3 + 4c_4$$
$$n = 8 \Longrightarrow 12 = 8c_1 + c_2 + 3c_3 + 9c_4$$

Resolviendo el sistema de ecuaciones obtenemos $c_1 = 3, c_2 = 0, c_3 = -2{,}5, c_4 = -0{,}5$. Podemos decir que:

$$T(n) \in \mathbf{O}(n)$$

□

Transformación de Rango

Los siguientes ejemplos muestran como realizar una transformación sobre no la variable sino sobre la función $T(n)$ para obtener otra función en la que podemos aplicar los anteriores resultados.

Ejemplo 1.8.14
Resolver la siguiente ecuación:

$$T(n) = \sqrt{n}T(\sqrt{n})) + n$$

El primer cambio que hacemos es el cambio de rango : $V(n) = \frac{T(n)}{n} \Longrightarrow T(n) = V(n) \cdot n$.

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

teniendo en cuenta que $V(n) = \frac{T(n)}{n}$ y $V(\sqrt{n}) = \frac{T(\sqrt{n})}{\sqrt{n}}$

$V(n) = V(\sqrt{n}) + 1$ sobre esta ecuación aplicamos un cambio de variable $n = 2^{2^m}$

$$V(2^{2^m}) = V(2^{2^{m-1}}) + 1 \Longrightarrow \begin{cases} x = V(2^{2^m}) \\ b = 1 \\ p(m) = 1 \\ d = 0 \end{cases}$$

$$(x-1)^2 = 0$$
$$V(2^{2^m}) = c_1 \cdot 1^m + c_2 \cdot m \cdot 1^m \Longrightarrow$$
deshaciendo el cambio de variable $m = loglogn$
$$V(n) = c_1 + c_2 log_2(log_2(n))$$
deshaciendo el cambio de rango $\frac{T(n)}{n} = c_1 + c_2 log_2(log_2(n))$
$$T(n) = c_1 n + c_2 n log_2(log_2(n))$$
y podemo concluir que $T(n) \in O(nlog_2(log_2(n))$

□

Ejemplo 1.8.15
Resolver la siguiente ecuación:

$T(n) = nT^2(\frac{n}{2})$ para $n > 1$ y $T(1) = 6$ y n potencia de dos

El primer cambio que hacemos es el cambio de rango : $V(n) = log_2(T(n)) \Longrightarrow T(n) = 2^{V(n)}$.

$$log_2(T(n) = log_2(n) + 2log(T(\tfrac{n}{2}))$$
$$V(n) = 2V(\tfrac{n}{2}) + log_2(n)$$
hacemos el cambio de variable $n = 2^m$
$$V(2^m) = 2V(2^{m-1}) + m \Longrightarrow \begin{cases} x = V(2^m) \\ b = 1 \\ p(m) = m \\ d = 1 \end{cases}$$

$$(x-2)(x-1)^2 = 0$$
$$V(2^m) = c_1 \cdot 2^m + c_2 \cdot 1^m + c_3 \cdot m \cdot 1^m \Longrightarrow$$
deshaciendo el cambio de variable $m = logn$
$$V(n) = c_1 n + c_2 + c3log_2(n)$$
deshaciendo el cambio de rango $log_2(T(n)) = c_1 n + c_2 + c_3 log_2(n)$
$$T(n) = 2^{c_1 n + c_2 + c_3 log_2(n)}$$

Para obtener las constantes se puede hacer desde $V(n)$ o desde $T(n)$. Si lo hacemos desde $V(n)$

Con este sistema despejamos y obtenemos que $c_1 = 3 + log_2(3) = 4{,}585, c_2 = -2, c_3 = -1$. Por lo tanto :

$$T(n) = 2^{(3+log_2(3))n-2-log_2(n)}) = 8^n . \frac{3}{4} \in \mathbf{O}(8^n)$$

□