

Tema 4: Diseño de algoritmos recursivos

Clara María Segura Díaz
Fundamentos de Algoritmia, Curso 2020-21

Dpto. de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

- **Diseño de programas. Formalismo y abstracción;** R. Peña. Tercera edición. Prentice Hall, 2005.

Capítulo 3

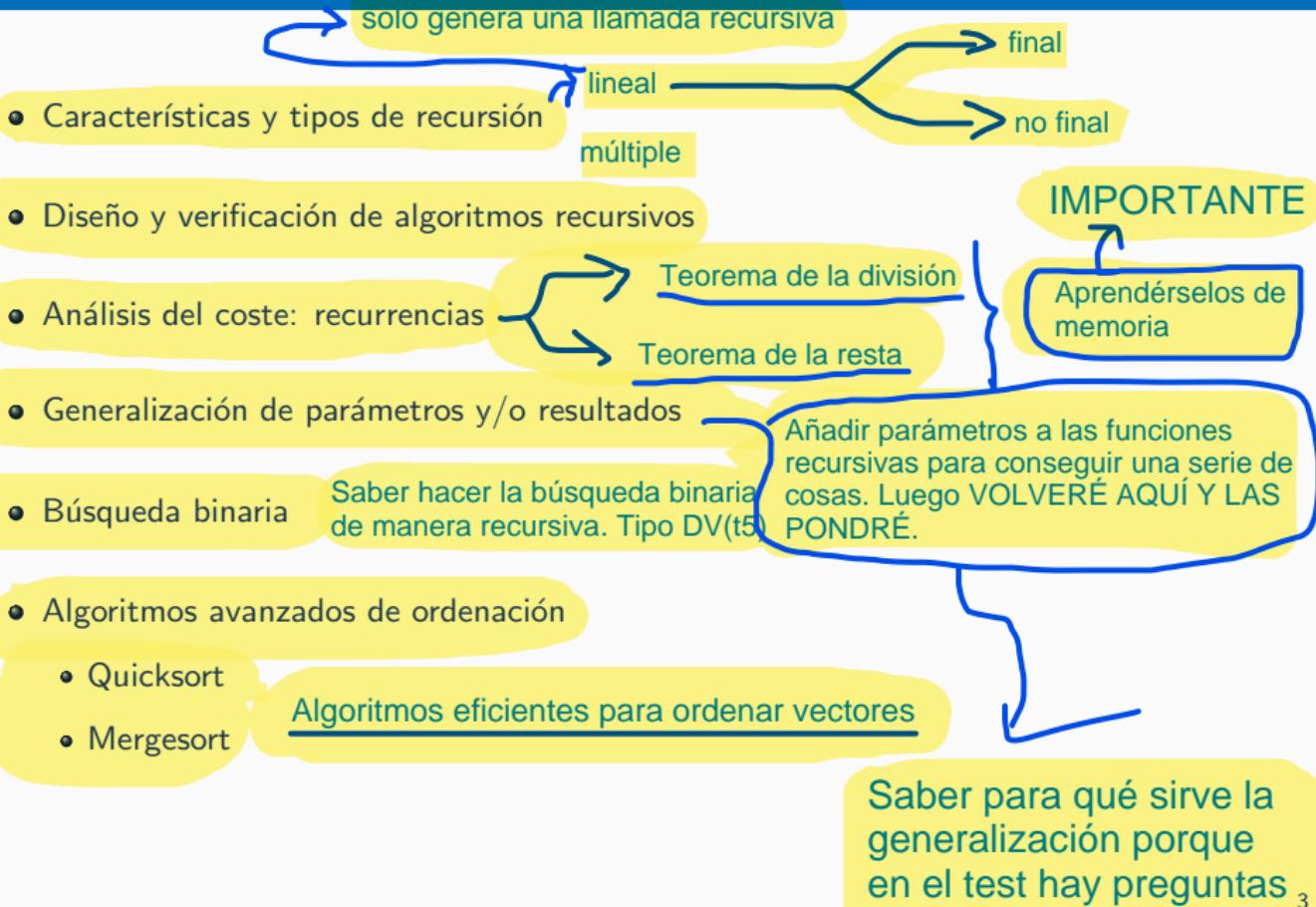
- **Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios.** N. Martí, C. Segura, J.A. Verdejo. Ibergarceta Publicaciones, 2012.

Capítulo 5

- **Programación Metódica.** J. L. Balcázar. McGraw-Hill, 1993.

Capítulos 3, 4, 5 y 6

Diseño de algoritmos recursivos



Características y tipos de recursión

Características de la recursión

- La recursividad es una característica de la mayoría de los lenguajes de programación que permite que un procedimiento o función haga referencia a sí mismo dentro de su definición.
- La recursividad se utiliza, al igual que la iteración, para describir cálculos que han de repetirse un cierto número de veces. Permite describir los mismos cómputos que la programación iterativa, en muchos casos mediante programas más compactos.
Misma función que los programas iterativos pero a veces quedan más compactos usando recursión
- Razonar recursivamente permite resolver un problema P sobre unos datos D suponiendo que P ya está resuelto para otros datos D' , del mismo tipo que D y en algún sentido más sencillos.

Para que el razonamiento tenga sentido se requiere que P pueda ser resuelto directamente para unos datos suficientemente sencillos.

Recursión lineal

Lineal solo hace una llamada recursiva en todo el programa

Un algoritmo es **recursivo lineal** si su ejecución genera una única llamada recursiva. El esquema general de un algoritmo recursivo lineal es:

```
{P( $\bar{x}$ )}  
fun fRec( $T_1 \bar{x}$ ) dev  $T_2 \bar{y}$  {  
    if  $B_t(\bar{x})$   
         $\bar{y} = \text{triv}(\bar{x});$   
    else  
         $\bar{y} = \text{comp}(f\text{-rec}(s(\bar{x})), \bar{x});$   
    return  $\bar{y};$   
}  
{Q( $\bar{x}, \bar{y}$ )}
```

Es decir, en cada ejecución del programa como mucho se llama a una.

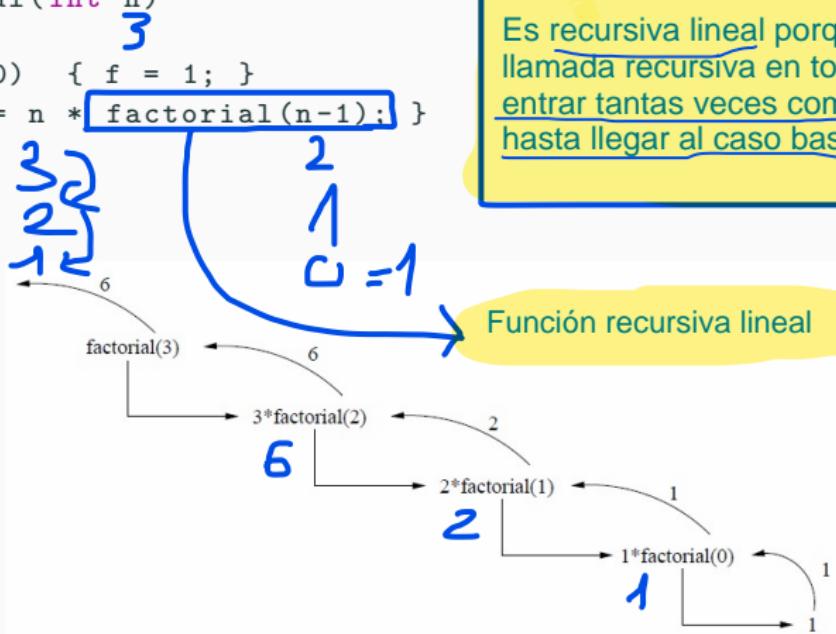
Recursivo lineal implica que SOLO hay una llamada recursiva. En este caso (rec). El caso triv() es el caso trivial o caso base.

Una única llamada recursiva

En nuestros problemas de recursión podemos tener una única llamada recursiva o bien varias como es en el caso del DV divide y vencerás, que por tanto será recursivo múltiple.

Factorial

```
{n ≥ 0}  
//fun factorial(int n) dev int f  
int factorial(int n)  
    int f;  
    if (n == 0) { f = 1; }  
    else { f = n * factorial(n-1); }  
    return f;  
}  
{f = n!}
```



Factorial

.....
3
.....

prog ppal

.....
3
.....

factorial(m)

.....
3
.....

factorial(2)

.....
2
.....

factorial(1)

.....
3
.....

factorial(0)

.....
2
.....

.....
1
.....

.....
1
.....

.....
0
.....

.....
1
.....

.....
1
.....

.....
2
.....

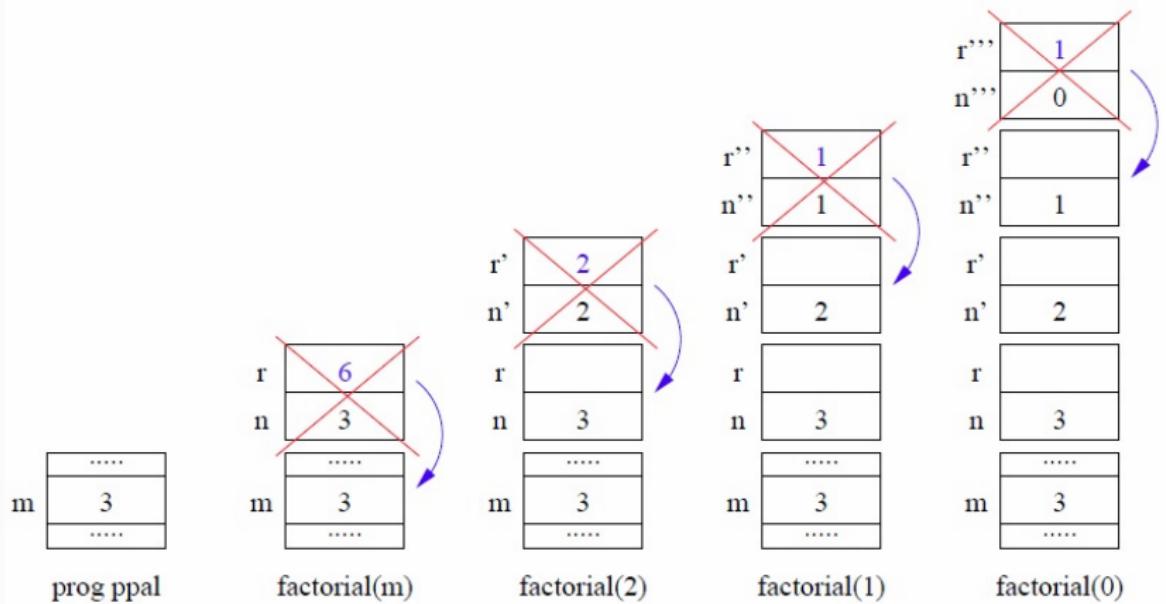
.....
6
.....

.....
3
.....

.....
3
.....

Registros
de activacion

Factorial



Potencia

No se si esta sería NO FINAL porque lo último que hace es multiplicar la a por el resultado de la recursión

$\{n \geq 0\}$

```
//fun potencia(int a, int n) dev int p  
int potencia(int a, int n)  
    int p;  
    if (n == 0) { p = 1; }  
    else { p = a * potencia(a,n-1); }  
    return p;  
}  
 $\{p = a^n\}$ 
```

Si queremos hacer 5^4 el 5 es la a, que es lo que multiplicamos siempre y la n es el exponente. De forma que $p = 5 * 5 * 5 * 5 * 1$

Caso base

Es el exponente

Según el chat gpt es final. No final implica que después de la llamada recursiva se hace algo aieno a la misma

Una única llamada recursiva

Los algoritmos **recursivos lineales** pueden ser:

- **Recursivo final:** la función comp no realiza ninguna acción (devuelve el primer argumento).
- **Recursivo no final:** la función comp realiza alguna acción.

la función comp no se que es, no se si es la función de la llamada recursiva.

Máximo común divisor

A pesar de que hay 2 llamadas recursivas, en cada llamada solo se accede a una de ellas. Es por eso que sigue siendo recursivo lineal. Y es final porque no se hace nada después de las llamadas recursivas.

Entre 7 y 14

El mcd sería 7

Ejemplo de función recursiva final.

```
{a > 0 ∧ b > 0}      a=7, b=14
//fun euclides(int a,int b) dev int mcd
int euclides(int a, int b){
    int mcd;
    if (a == b) {mcd = a;}
    else if (a > b) { mcd = euclides(a-b,b); }
    else { mcd = euclides(a,b-a); }
    return mcd;
}
{mcd = mcd(a,b)}
```

7

7, 14-7

Y es lineal porque en cada llamada solo se realiza un función recursiva

Como no hay nada y no hace, solo la llamada recursiva es recursiva final

Es final si lo último que hace según el chat gpt es la función recursiva.
(En teoría no hace operaciones adicionales).

Fibonacci

Un algoritmo es **recursivo múltiple** si una misma invocación genera más de una llamada recursiva.

Tipos de algoritmos recursivos

Recursivo lineal

Recursivo final: Euclides
Recursivo no final

Recursivo múltiple

$$fib_0 = 0$$

$$fib_1 = 1$$

$$fib_n = fib_{n-1} + fib_{n-2} \text{ si } n \geq 2$$

```
{n ≥ 0}  
//fun fibonacci(int n) dev int f  
int fibonacci(int n){  
    int f;  
    if (n==0) { f = 0; }  
    else if (n==1) { f = 1; }  
    else { f = fibonacci(n-1) + fibonacci(n-2); }  
    return f;  
}  
{f = fibn}
```

Una misma invocación a esta función genera más de una llamada recursiva.

Dos llamadas a funciones recursivas.

En una misma llamada se accede a dos llamadas recursivas.

Fibonacci

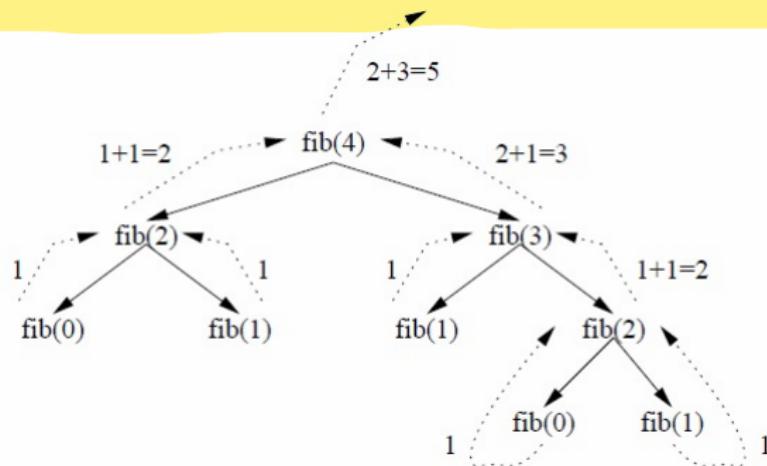
Resumen: Recursión = iterativos pero programas más compactos. Dos tipos de recursión:

1.Lineal: Si al invocar la función esta solo genera una llamada recursiva. Dos tipos:

1.1.Final: Si lo último que hace es lo de la llamada recursiva

1.2.No final: Lo último que hace es por ejemplo una multiplicación, suma...

2.Múltiple: Si En una invocación hay más de una llamada recursiva.



Lo último que se hace es cualquier otra cosa que no sea la llamada recursiva.

Diseño y verificación de algoritmos recursivos

Diseñar algoritmos y cómo saber si son correctos

Los pasos a seguir son:

- ① Especificación formal del algoritmo.
- ② Análisis por casos, descomposición.
- ③ Composición de resultados.
- ④ Verificación formal de la corrección.
- ⑤ Estudio del coste.

Análisis por casos y composición

Hay que pensar cómo se pueden descomponer los datos de forma que se pueda calcular fácilmente la solución pedida a partir de la solución al mismo problema para datos más pequeños.

Analizar cómo se resuelven los casos:

- triviales o básicos → caso base, al que tienes que llegar después de toda la ristra de llamadas
- no triviales o recursivos → Hasta llegar al caso base.

Análisis por casos y composición

Hay que pensar cómo se pueden descomponer los datos de forma que se pueda calcular fácilmente la solución pedida a partir de la solución al mismo problema para datos más pequeños.

Analizar cómo se resuelven los casos:

- triviales o básicos
- no triviales o recursivos

Ejemplo: cálculo de la potencia

$n = 0$	$a^0 = 1$	caso base
$n > 0$	$a^n = a * a^{n-1}$	caso recursivo

Análisis por casos y composición

Hay que pensar cómo se pueden descomponer los datos de forma que se pueda calcular fácilmente la solución pedida a partir de la solución al mismo problema para datos más pequeños.

Analizar cómo se resuelven los casos:

- triviales o básicos
- no triviales o recursivos

Ejemplo: cálculo de la potencia

$n = 0$	$a^0 = 1$
$n > 0$	$a^n = a * a^{n-1}$

Puede haber varios casos básicos y varios casos recursivos.

Puede ser una pregunta de test

- Asegurarse informalmente de que se reducen los problemas en los casos recursivos y de que estos nos acercan al caso trivial. **trivial = sencillo** siempre buscar el caso trivial.
- Asegurarse de que se cubren todos los casos permitidos por la **precondición**.

Como en cualquier ejercicio, asegurarse que la precondición se cumple.

Corrección

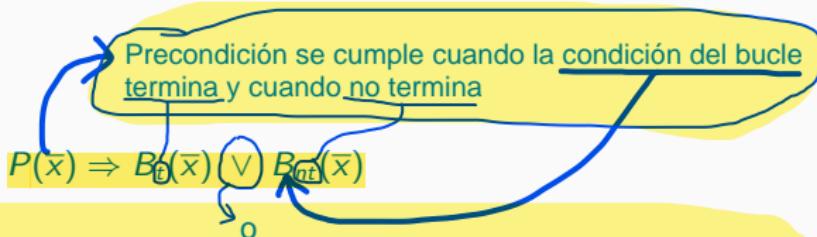
P = Precondición y Q = Postcondición.

Tenemos que probar que para todo posible dato de entrada \bar{x} que cumpla $P(\bar{x})$, se cumple $Q(\bar{x}, f\text{-rec}(\bar{x}))$.

Corrección

Tenemos que probar que para todo posible dato de entrada \bar{x} que cumpla $P(\bar{x})$, se cumple $Q(\bar{x}, f\text{-rec}(\bar{x}))$.

- ① Se cubren todos los casos:



En teoría, que la precondición se cumpla, según esto es que se cumple cuando la condición del bucle termino O cuando la condición del bucle no termina.

Corrección

Tenemos que probar que para todo posible dato de entrada \bar{x} que cumpla $P(\bar{x})$, se cumple $Q(\bar{x}, \text{f-rec}(\bar{x}))$.

- ① Se cubren todos los casos:

$$P(\bar{x}) \Rightarrow B_t(\bar{x}) \vee B_{nt}(\bar{x})$$

- ② El caso base es correcto:

$$P(\bar{x}) \wedge B_t(\bar{x}) \Rightarrow Q(\bar{x}, \text{triv}(\bar{x}))$$

Corrección

Tenemos que probar que para todo posible dato de entrada \bar{x} que cumpla $P(\bar{x})$, se cumple $Q(\bar{x}, f\text{-rec}(\bar{x}))$.

- ① Se cubren todos los casos:

Condición del bucle termina Condición del bucle no termina

$$P(\bar{x}) \Rightarrow B_t(\bar{x}) \vee B_{nt}(\bar{x})$$

Precondicion se cumple: $B_t =$ termina. $B_{nt} =$ no termina.

Precondición se cumple en las llamadas recursivas y en el caso trivial.

- ② El caso base es correcto:

si se cumple pre y se llega al caso base, se cumple pos
 $P(\bar{x}) \wedge B_t(\bar{x}) \Rightarrow Q(\bar{x}, \text{triv}(\bar{x}))$

La precondición y el caso base(caso base es cuando termina), la postcondición se cumple

- ③ La función recursiva es invocada siempre en estados que satisfacen su precondición:

$$P(\bar{x}) \wedge B_{nt}(\bar{x}) \Rightarrow P(s(\bar{x}))$$

sucesor, eso quiere decir que se cumple para las llamadas recursivas siguientes.

Los datos de entrada siempre satisfacen la precondición

- ④ El paso de inducción es correcto:

$$P(\bar{x}) \wedge B_{nt}(\bar{x}) \wedge Q(s(\bar{x}), \bar{y'}) \Rightarrow Q(\bar{x}, c(\bar{y'}, \bar{x}))$$

Corrección

- ④ El paso de inducción es correcto:

$$P(\bar{x}) \wedge B_{nt}(\bar{x}) \wedge Q(s(\bar{x}), \bar{y}') \Rightarrow Q(\bar{x}, c(\bar{y}', \bar{x}))$$

- ⑤ Existe una función de tamaño t tal que

$$P(\bar{x}) \Rightarrow t(\bar{x}) \geq 0$$

Esto era en el caso del tema 3
del invariante para hallar la
función de cota.

- ⑥ El valor de t decrece al hacer la llamada recursiva

$$P(\bar{x}) \wedge B_{nt}(\bar{x}) \Rightarrow t(s(\bar{x})) < t(\bar{x})$$

en la llamada sucesora o siguiente la t decrece, pero cumpliendo que siempre es mayor o igual que cero.

Ejemplo: potencia

Vamos a ver que se cumpla todo lo anterior utilizando el ejemplo de la potencia.

precondición

$\{n \geq 0\}$ negativos no aceptados para el exponente.

```
//fun potencia(int a, int n) dev int p
```

```
int potencia(int a, int n)
```

```
    int p;
```

```
    if (n == 0) { p = 1; } si exponente es 0, p = 1.
```

```
    else { p = a * potencia(a,n-1); }
```

```
    return p;
```

```
}
```

$\{p = a^n\}$ postcondición

$$\textcircled{1} \quad n \geq 0 \Rightarrow n = 0 \vee n > 0$$

la precondición implica que se cumple para el caso base/trivial ($n=0$) y para los casos recursivos. Se cumple

Ejemplo: potencia

```
{n ≥ 0}  
//fun potencia(int a, int n) dev int p  
int potencia(int a, int n)  
    int p;  
    if (n == 0) { p = 1; }  
    else { p = a * potencia(a,n-1); }  
    return p;  
}  
{p = an}
```

- ① $n \geq 0 \Rightarrow n = 0 \vee n > 0$ P(x) => Bt(x) Bnt(x) Si, la precondición se cumple p.t la entrada
- ② $n \geq 0 \wedge n = 0 \Rightarrow 1 = a^n$ P(x) \wedge Bt(x) => Q(x, triv(x)). El caso base es correcto

Cuando B termina

Ejemplo: potencia

```
{n ≥ 0}  
//fun potencia(int a, int n) dev int p  
int potencia(int a, int n)  
    int p;  
    if (n == 0) { p = 1; }  
    else { p = a * potencia(a,n-1); }  
    return p;  
}  
{p = an}
```

① $n \geq 0 \Rightarrow n = 0 \vee n > 0$

② $n \geq 0 \wedge n = 0 \Rightarrow 1 = a^n$ esto se cumple, ya que cualquier número⁰ = 1.

③ $n \geq 0 \wedge n > 0 \Rightarrow n - 1 \geq 0$

Ejemplo: potencia

```
{n ≥ 0}  
//fun potencia(int a, int n) dev int p  
int potencia(int a, int n)  
    int p;  
    if (n == 0) { p = 1; }  
    else { p = a * potencia(a,n-1); }  
    return p;  
}  
{p = an}
```

$$\textcircled{1} \quad n \geq 0 \Rightarrow n = 0 \vee n > 0$$

$$\textcircled{2} \quad n \geq 0 \wedge n = 0 \Rightarrow 1 = a^n$$

$$\textcircled{3} \quad n \geq 0 \wedge n > 0 \Rightarrow n - 1 \geq 0$$

$$\textcircled{4} \quad n \geq 0 \wedge n > 0 \wedge p' = a^{n-1} \Rightarrow a * p' = a^n$$

Los datos que pasemos a "trabajadores" tienen que cumplir la precondición. Nos garantiza que la llamada recursiva hace lo que debe.

Esto es la hipótesis de inducción.

Ejemplo: potencia

```
{n ≥ 0}
//fun potencia(int a, int n) dev int p
int potencia(int a, int n)
    int p;
    if (n == 0) { p = 1; }
    else { p = a * potencia(a,n-1); }
    return p;
}
{p = an}
```

- ① $n \geq 0 \Rightarrow n = 0 \vee n > 0$
- ② $n \geq 0 \wedge n = 0 \Rightarrow 1 = a^n$
- ③ $n \geq 0 \wedge n > 0 \Rightarrow n - 1 \geq 0$
- ④ $n \geq 0 \wedge n > 0 \wedge p' = a^{n-1} \Rightarrow a * p' = a^n$
- ⑤ $n \geq 0 \Rightarrow n \geq 0$

Ejemplo: potencia

```
{n ≥ 0}  
//fun potencia(int a, int n) dev int p  
int potencia(int a, int n)  
    int p;  
    if (n == 0) { p = 1; }  
    else { p = a * potencia(a,n-1); }  
    return p;  
}  
{p = an}
```

Función de cota, porque siempre es positiva.
Acaba siendo 0

$$① n \geq 0 \Rightarrow n = 0 \vee n > 0$$

Tanto caso base como recursivo
 $P(x) \Rightarrow Bt(x)$ $Bnt(x)$ Si, la precondition se cumple p.t la entrada

$$② n \geq 0 \wedge n = 0 \Rightarrow 1 = a^n$$

$P(x) \wedge Bt(x) \Rightarrow Q(x, \text{triv}(x))$. El caso base es correcto

$$③ n \geq 0 \wedge n > 0 \Rightarrow n - 1 \geq 0$$

$P(x) \wedge Bnt(x) \Rightarrow P(s(x))$ La función recursiva es invocada siempre en estados que satisfacen su precondition

$$④ n \geq 0 \wedge n > 0 \wedge p' = a^{n-1} \Rightarrow a * p' = a^n$$

$P(x) \wedge Bnt(x) \wedge Q(s(x), y_0) \Rightarrow Q(x, c(y_0, x))$ El paso de inducción es correcto.

$$⑤ n \geq 0 \Rightarrow n \geq 0$$

$P(x) \Rightarrow t(x) \geq 0$ Existe una función de tamaño t

$$⑥ n \geq 0 \wedge n > 0 \Rightarrow n - 1 < n$$

$P(x) \wedge Bnt(x) \Rightarrow t(s(x)) < t(x)$ El valor de t decrece al hacer la llamada recursiva. En este caso la n en cada llamada decrece para llegar al caso base/trivial. Y acaba siendo ≥ 0 .

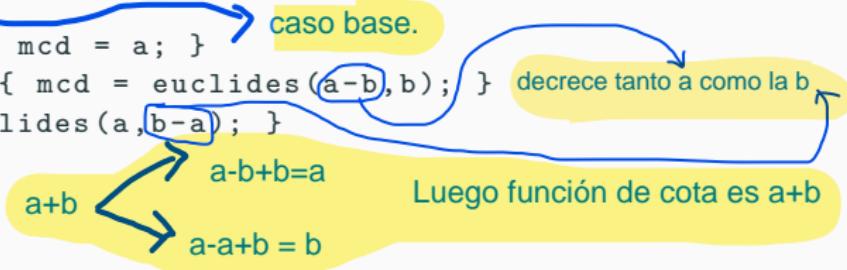
s(x) es como la llamada recursiva

Ejemplo: máximo común divisor

Tipo de recursión: lineal final.

Es final porque lo último que hace es la llamada recursiva, no hay ni multiplicación ni división.

```
{a > 0 ∧ b > 0}  
//fun euclides(int a,int b) dev int mcd  
int euclides(int a, int b){  
    int mcd;  
    if (a == b) { mcd = a; }  
    else if (a > b) { mcd = euclides(a-b,b); }  
    else { mcd = euclides(a,b-a); }  
    return mcd;  
}  
{mcd = mcd(a,b)}
```



Se utilizan las propiedades del máximo común divisor:

- $mcd(a, b) = mcd(a - b, b)$ si $a > b$.
- $mcd(a, b) = mcd(a, b - a)$ si $a < b$.
- $mcd(a, b) = a$ si $a = b$ caso base

dos casos recursivos.

Ejemplo: máximo común divisor

① $P(\bar{x}) \Rightarrow B_t(\bar{x}) \vee B_{nt}(\bar{x})$ Precondición se cumple cuando termina y cuando no termina

$$a > 0 \wedge b > 0 \Rightarrow a = b \vee a > b \vee a < b$$

$P(x)$

$Bt(x)$

$Bnt(x)$

Ejemplo: máximo común divisor

① $P(\bar{x}) \Rightarrow B_t(\bar{x}) \vee B_{nt}(\bar{x})$

$$a > 0 \wedge b > 0 \Rightarrow a = b \vee a > b \vee a < b$$

② $P(\bar{x}) \wedge B_t(\bar{x}) \Rightarrow Q(\bar{x}, \text{triv}(\bar{x}))$

$$a > 0 \wedge b > 0 \wedge a = b \Rightarrow a = \text{mcd}(a, b)$$

Ejemplo: máximo común divisor

① $P(\bar{x}) \Rightarrow B_t(\bar{x}) \vee B_{nt}(\bar{x})$

$$a > 0 \wedge b > 0 \Rightarrow a = b \vee a > b \vee a < b$$

② $P(\bar{x}) \wedge B_t(\bar{x}) \Rightarrow Q(\bar{x}, \text{triv}(\bar{x}))$

$$a > 0 \wedge b > 0 \wedge a = b \Rightarrow a = \text{mcd}(a, b)$$

③ $P(\bar{x}) \wedge B_{nt}(\bar{x}) \Rightarrow P(s(\bar{x}))$

$$a > 0 \wedge b > 0 \wedge a > b \Rightarrow a - b > 0 \wedge b > 0$$

$$a > 0 \wedge b > 0 \wedge a < b \Rightarrow a > 0 \wedge b - a > 0$$

Ejemplo: máximo común divisor

④ $P(\bar{x}) \wedge B_{nt}(\bar{x}) \wedge Q(s(\bar{x}), \bar{y'}) \Rightarrow Q(\bar{x}, c(\bar{y'}, \bar{x}))$

$$a > 0 \wedge b > 0 \wedge a > b \wedge mcd' = mcd(a - b, b) \Rightarrow mcd' = mcd(a, b)$$

$$a > 0 \wedge b > 0 \wedge a < b \wedge mcd' = mcd(a, b - a) \Rightarrow mcd' = mcd(a, b)$$

Ejemplo: máximo común divisor

④ $P(\bar{x}) \wedge B_{nt}(\bar{x}) \wedge Q(s(\bar{x}), \bar{y'}) \Rightarrow Q(\bar{x}, c(\bar{y'}, \bar{x}))$

$$a > 0 \wedge b > 0 \wedge a > b \wedge mcd' = mcd(a - b, b) \Rightarrow mcd' = mcd(a, b)$$

$$a > 0 \wedge b > 0 \wedge a < b \wedge mcd' = mcd(a, b - a) \Rightarrow mcd' = mcd(a, b)$$

⑤ $P(\bar{x}) \Rightarrow t(\bar{x}) \geq 0$ ya que hemos visto que puede decrecer tanto la a como la b

Elegimos como función $t = a + b$.

$$a > 0 \wedge b > 0 \Rightarrow \{a + b \geq 0\}$$

Ejemplo: máximo común divisor

④ $P(\bar{x}) \wedge B_{nt}(\bar{x}) \wedge Q(s(\bar{x}), \bar{y'}) \Rightarrow Q(\bar{x}, c(\bar{y'}, \bar{x}))$

$$a > 0 \wedge b > 0 \wedge a > b \wedge mcd' = mcd(a - b, b) \Rightarrow mcd' = mcd(a, b)$$

$$a > 0 \wedge b > 0 \wedge a < b \wedge mcd' = mcd(a, b - a) \Rightarrow mcd' = mcd(a, b)$$

⑤ $P(\bar{x}) \Rightarrow t(\bar{x}) \geq 0$

Elegimos como función $t = a + b$.

$$a > 0 \wedge b > 0 \Rightarrow \{a + b \geq 0\}$$

⑥ $P(\bar{x}) \wedge B_{nt}(\bar{x}) \Rightarrow t(s(\bar{x})) < t(\bar{x})$

$$a > 0 \wedge b > 0 \wedge a > b \Rightarrow a - b + b < a + b$$

$$a > 0 \wedge b > 0 \wedge a < b \Rightarrow a + b - a < a + b$$

Corrección de la recursión múltiple

Recursión múltiple.

Supongamos que una invocación de la función genera k llamadas recursivas con argumentos $s_1(\bar{x}), \dots, s_k(\bar{x})$. Hemos de adaptar algunos pasos de verificación:

- Todas las llamadas recursivas deben ser invocadas en estados que satisfagan su precondición. Para cada $i \in \{1..k\}$ debemos demostrar:

$$P(\bar{x}) \wedge B_{nt}(\bar{x}) \Rightarrow P(s_i(\bar{x}))$$

Para el número de llamadas recursivas que haya.

- En el paso de inducción podemos asumir que todas las llamadas recursivas son correctas:

$$P(\bar{x}) \wedge B_{nt}(\bar{x}) \wedge \bigwedge_{i=1}^k Q(s_i(\bar{x}), \bar{y}'_i) \Rightarrow Q(\bar{x}, c(\bar{y}'_1, \dots, \bar{y}'_k, \bar{x}))$$

- El valor de t decrece en todas las llamadas recursivas. Para cada $i \in \{1..k\}$ debemos demostrar:

Para cada llamada recursiva, la cota decrece.

$$P(\bar{x}) \wedge B_{nt}(\bar{x}) \Rightarrow t(s_i(\bar{x})) < t(\bar{x})$$

Si esto se cumple podemos asumir que la postcondición se cumple.

Ejemplo

```
{n ≥ 0}  
// fun frm(int n) dev int r  
int frm(int n) {  
    int r;  
    if (n == 0) { r = 0; }  
    else if (n == 1) { r = 1; }  
    else { r = 5 * frm(n - 1) - 6 * frm(n - 2); }  
    return r;  
}
```

$$\{r = 3^n - 2^n\}$$

Recursión múltiple. (2 llamadas recursivas)

Va a comprobar las reglas anteriores modificando las necesarias al ser múltiple.

- ① $n \geq 0 \Rightarrow (n = 0) \vee (n = 1) \vee (n \geq 2)$ P(x)=> Bt(x) \vee Bnt(x)

Ejemplo

```
{n ≥ 0}
// fun frm(int n) dev int r
int frm(int n) {
    int r;
    if (n == 0) { r = 0; }
    else if (n == 1) { r = 1; }
    else { r = 5 * frm(n - 1) - 6 * frm(n - 2); }
    return r;
}
{r = 3n - 2n}
```

① $n \geq 0 \Rightarrow (n = 0) \vee (n = 1) \vee (n \geq 2)$

② Tenemos dos casos básicos:

- $n \geq 0 \wedge n = 0 \Rightarrow 0 = 3^n - 2^n$
- $n \geq 0 \wedge n = 1 \Rightarrow 1 = 3^n - 2^n$

Ya que $1-1=0$;
Ya que $3-2=1$

Ejemplo

```
{n ≥ 0}
// fun frm(int n) dev int r
int frm(int n) {
    int r;          Bt
    if (n == 0) { r = 0; }
    else if (n == 1) { r = 1; } Bt
    else { r = 5 * frm(n - 1) - 6 * frm(n - 2); }
    return r;
}
{r = 3n - 2n}
```

① $n \geq 0 \Rightarrow (n = 0) \vee (n = 1) \vee (n \geq 2)$

$P(x) \Rightarrow Bt(x) \quad Bnt(x)$ Si, la precondición se cumple p.t la entrada

② Tenemos dos casos básicos:

- $n \geq 0 \wedge n = 0 \Rightarrow 0 = 3^n - 2^n$
- $n \geq 0 \wedge n = 1 \Rightarrow 1 = 3^n - 2^n$

$P(x) \wedge Bt(x) \Rightarrow Q(x, \text{triv}(x))$. El caso base es correcto

③ Las dos llamadas recursivas sobre argumentos que cumplen la precondición:

- $n \geq 0 \wedge n \geq 2 \Rightarrow n - 1 \geq 0$
- $n \geq 0 \wedge n \geq 2 \Rightarrow n - 2 \geq 0$

$P(x) \wedge Bnt(x) \Rightarrow P(s(x))$ La función recursiva es invocada siempre en estados que satisfacen su precondición

Ejemplo

- ④ Suponemos que las llamadas recursivas son correctas,

$$\text{frm}(n-1) = 3^{n-1} - 2^{n-1}$$

$$\text{frm}(n-2) = 3^{n-2} - 2^{n-2},$$

de donde obtenemos

$$\begin{aligned} &= \frac{5 * (3^{n-1} - 2^{n-1}) - 6 * (3^{n-2} - 2^{n-2})}{(5 * 3) * 3^{n-2} - (5 * 2) * 2^{n-2}} + \frac{6}{6 * 3^{n-2} + 6 * 2^{n-2}} \\ &= \frac{15-6}{9} * 3^{n-2} - \frac{10-6}{4} * 2^{n-2} \\ &= 3^n - 2^n. \end{aligned}$$

Inducción se cumple.

Ejemplo

- ④ Suponemos que las llamadas recursivas son correctas,

$$\text{frm}(n-1) = 3^{n-1} - 2^{n-1}$$

$$\text{frm}(n-2) = 3^{n-2} - 2^{n-2},$$

de donde obtenemos

$$\begin{aligned}& 5 * (3^{n-1} - 2^{n-1}) - 6 * (3^{n-2} - 2^{n-2}) \\&= (5 * 3) * 3^{n-2} - (5 * 2) * 2^{n-2} - 6 * 3^{n-2} + 6 * 2^{n-2} \\&= 9 * 3^{n-2} - 4 * 2^{n-2} \\&= 3^n - 2^n.\end{aligned}$$

- ⑤ $t = n$. $n \geq 0 \Rightarrow n \geq 0$

- ⑥ Las dos llamadas recursivas hacen decrecer la cota:

- $n \geq 0 \wedge n \geq 2 \Rightarrow n-1 < n$
- $n \geq 0 \wedge n \geq 2 \Rightarrow n-2 < n$

La función de cota diría que es la n ya que es lo que va decreciendo en cada llamada recursiva.

Saber cual es el coste de los algoritmos recursivos.

Análisis del coste: recurrencias

Como hacer el análisis de coste de las recursivas.

FORMA MANUAL

APRENDERSE LAS DOS FÓRMULAS

Siempre
piden esta
forma.

Análisis del coste

Cuando se analiza la complejidad de un algoritmo recursivo es frecuente que aparezcan funciones de coste también recursivas, llamadas **recurrencias**.

```
{n ≥ 0}  
//fun factorial(int n) dev int f  
int factorial(int n) n=7  
    int f;  
    if (n == 0) { f = 1; }  
    else { f = n * factorial(n-1); }  
    return f;  
}  
{f = n!}
```

Esto tendría el siguiente coste y la siguiente recurrencia:

$$T(n) \begin{cases} K_0 & \text{si } n == 0 \\ T(n-1) + k_1 & \text{si } n > 0 \end{cases}$$



k1= La multiplicación.

Cuanto más grande sea el número más me va a costar calcular el factorial

Análisis del coste

Cuando se analiza la complejidad de un algoritmo recursivo es frecuente que aparezcan funciones de coste también recursivas, llamadas **recurrencias**.

```
{n ≥ 0}  
//fun factorial(int n) dev int f  
int factorial(int n)  
    int f;  
    if (n == 0) { f = 1; }  
    else { f = n * factorial(n-1); }  
    return f;  
}  
{f = n!}
```

$$T(n) = \begin{cases} k_1 & \text{constante.} \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases}$$

1
Lo que me cuesta meterme en la llamada, más constantes.

Según el chat gpt la k_2 muestra el tiempo constante de hacer la multiplicación y la resta mientras que $T(n-1)$ es lo que cuesta la llamada recursiva. Todo el rato es de coste constante luego el coste del algoritmo es lineal.

Mi explicación: como $a=1$ entonces el coste es $a^k + 1$ aplicando el teorema de la resta. Como $k=0$ el coste $O(n)$ (lineal creo)

Análisis del coste

Cuando se analiza la complejidad de un algoritmo recursivo es frecuente que aparezcan funciones de coste también recursivas, llamadas **recurrencias**.

```
{n ≥ 0}  
//fun factorial(int n) dev int f  
int factorial(int n)  
    int f;  
    if (n == 0) { f = 1; }  
    else { f = n * factorial(n-1); }  
    return f;  
}  
{f = n!}
```

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n - 1) + k_2 & \text{si } n > 0 \end{cases}$$

Queremos obtener el *orden de complejidad* de $T(n)$. Dos formas:

- Mediante despliegue: obtener fórmula explícita de $T(n)$.
No vamos a usarlo salvo que ella nos lo pida explicitamente.
- Utilizando los teoremas que veremos: disminución del tamaño del problema por sustracción o por división.
Desarrollar la expresión varias veces.
Importantes

No lo haremos a menos que nos lo pida la profesora.

Despliegue de recurrencias

El objetivo es conseguir una fórmula explícita (en función de n) de $T(n)$.

Seguimos tres pasos:

Despliegue Sustituimos T por la parte derecha de la ecuación tantas veces como sea necesario hasta encontrar una fórmula que dependa del número de despliegues (o llamadas recursivas) i .

NO SE SUELE PEDIR NUNCA, APRENDERSE LAS FÓRMULAS.

Despliegue de recurrencias

El objetivo es conseguir una fórmula explícita (en función de n) de $T(n)$.

Seguimos tres pasos:

Despliegue Sustituimos T por la parte derecha de la ecuación tantas veces como sea necesario hasta encontrar una fórmula que dependa del número de despliegues (o llamadas recursivas) i .

Postulado Obtenemos el valor de i que hace que T desaparezca (caso básico), y en la fórmula sustituimos i por ese valor, obteniendo la fórmula explícita T^* (que solo depende de n).

Despliegue de recurrencias

El objetivo es conseguir una fórmula explícita (en función de n) de $T(n)$.

Seguimos tres pasos:

Despliegue Sustituimos T por la parte derecha de la ecuación tantas veces como sea necesario hasta encontrar una fórmula que dependa del número de despliegues (o llamadas recursivas) i .

Postulado Obtenemos el valor de i que hace que T desaparezca (caso básico), y en la fórmula sustituimos i por ese valor, obteniendo la fórmula explícita T^* (que solo depende de n).

Demostración Demostramos (por inducción) que $T = T^*$.

Explicacion

Ejemplo: factorial

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n - 1) + k_2 & \text{si } n > 0 \end{cases}$$

$$T(n) \stackrel{1}{=} T(n - 1) + k_2$$

Ejemplo: factorial

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} T(n-1) + k_2 \\ &\stackrel{2}{=} T(n-2) + k_2 + k_2 \end{aligned}$$

Ejemplo: factorial

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} T(n-1) + k_2 \\ &\stackrel{2}{=} T(n-2) + k_2 + k_2 &= & T(n-2) + 2 \cdot k_2 \end{aligned}$$

Ejemplo: factorial

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} T(n-1) + k_2 \\ &\stackrel{2}{=} T(n-2) + k_2 + k_2 = T(n-2) + 2 \cdot k_2 \\ &\stackrel{3}{=} T(n-3) + k_2 + 2 \cdot k_2 \end{aligned}$$

Ejemplo: factorial

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} T(n-1) + k_2 \\ &\stackrel{2}{=} T(n-2) + k_2 + k_2 = T(n-2) + 2 \cdot k_2 \\ &\stackrel{3}{=} T(n-3) + k_2 + 2 \cdot k_2 = T(n-3) + 3 \cdot k_2 \end{aligned}$$

Ejemplo: factorial

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} T(n-1) + k_2 \\ &\stackrel{2}{=} T(n-2) + k_2 + k_2 = T(n-2) + 2 \cdot k_2 \\ &\stackrel{3}{=} T(n-3) + k_2 + 2 \cdot k_2 = T(n-3) + 3 \cdot k_2 \\ &\vdots \\ &\stackrel{i}{=} T(n-i) + i \cdot k_2 \end{aligned}$$

Ejemplo: factorial

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} T(n-1) + k_2 \\ &\stackrel{2}{=} T(n-2) + k_2 + k_2 = T(n-2) + 2 \cdot k_2 \\ &\stackrel{3}{=} T(n-3) + k_2 + 2 \cdot k_2 = T(n-3) + 3 \cdot k_2 \\ &\vdots \\ &\stackrel{i}{=} T(n-i) + i \cdot k_2 \\ &\vdots \\ &\stackrel{n}{=} T(0) + n \cdot k_2 \end{aligned}$$

Ejemplo: factorial

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} T(n-1) + k_2 \\ &\stackrel{2}{=} T(n-2) + k_2 + k_2 = T(n-2) + 2 \cdot k_2 \\ &\stackrel{3}{=} T(n-3) + k_2 + 2 \cdot k_2 = T(n-3) + 3 \cdot k_2 \\ &\vdots \\ &\stackrel{i}{=} T(n-i) + i \cdot k_2 \\ &\vdots \\ &\stackrel{n}{=} T(0) + n \cdot k_2 = k_2 n + k_1 = T^*(n) \end{aligned}$$

Ejemplo: factorial

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} T(n-1) + k_2 \\ &\stackrel{2}{=} T(n-2) + k_2 + k_2 = T(n-2) + 2 \cdot k_2 \\ &\stackrel{3}{=} T(n-3) + k_2 + 2 \cdot k_2 = T(n-3) + 3 \cdot k_2 \\ &\vdots \\ &\stackrel{i}{=} T(n-i) + i \cdot k_2 \\ &\vdots \\ &\stackrel{n}{=} T(0) + n \cdot k_2 = k_2 n + k_1 = T^*(n) \in \Theta(n) \end{aligned}$$

Es más sencillo aprenderse el teorema de la resta que hacer esto. La explicación por el teorema de la resta está arriba

Ejemplo: factorial

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n-1) + k_2 & \text{si } n > 0 \end{cases} \quad T(0) = k_1$$

$$\begin{aligned} T(n) &\stackrel{1}{=} T(n-1) + k_2 \\ &\stackrel{2}{=} T(n-2) + k_2 + k_2 = T(n-2) + 2 \cdot k_2 \\ &\stackrel{3}{=} T(n-3) + k_2 + 2 \cdot k_2 = T(n-3) + 3 \cdot k_2 \\ &\vdots \\ &\stackrel{i}{=} T(n-i) + i \cdot k_2 \quad \text{Lo desarrollamos } i \text{ veces} \\ &\vdots \\ &\stackrel{n}{=} T(0) + n \cdot k_2 = k_2 n + k_1 = T^*(n) \in \Theta(n) \end{aligned}$$

Demostramos que $T(n) = T^*(n)$ para todo $n \geq 0$.

Base: $T(0) = k_1 = T^*(0)$

H.I.: $T(n) = T^*(n)$

Para $n+1$: $T(n+1) = T(n) + k_2 \stackrel{\text{h.i.}}{=} k_2 n + k_1 + k_2 = (n+1)k_2 + k_1 = T^*(n+1)$

Si se cumple para n , por inducción comprobar que se cumple para $n+1$.

Ejemplos

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T(n - 1) + 2 & \text{si } n \geq 2 \end{cases}$$

Para T(1)=1

Ejemplos

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T(n - 1) + 2 & \text{si } n \geq 2 \end{cases}$$

$$T(n) \stackrel{1}{=} 3T(n - 1) + 2$$

Ejemplos

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T(n-1) + 2 & \text{si } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} 3T(n-1) + 2 \\ &\stackrel{2}{=} 3(3T(n-2) + 2) + 2 \end{aligned}$$

Ejemplos

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T(n - 1) + 2 & \text{si } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} 3T(n - 1) + 2 \\ &\stackrel{2}{=} 3(3T(n - 2) + 2) + 2 = 3^2T(n - 2) + 3 \cdot 2 + 2 \end{aligned}$$


Ejemplos

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T(n-1) + 2 & \text{si } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} 3T(n-1) + 2 \\ &\stackrel{2}{=} 3(3T(n-2) + 2) + 2 = 3^2T(n-2) + 3 \cdot 2 + 2 \\ &\stackrel{3}{=} 3^2(3T(n-3) + 2) + 3 \cdot 2 + 2 \end{aligned}$$

Ejemplos

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T(n-1) + 2 & \text{si } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} 3T(n-1) + 2 \\ &\stackrel{2}{=} 3(3T(n-2) + 2) + 2 = 3^2T(n-2) + 3 \cdot 2 + 2 \\ &\stackrel{3}{=} 3^2(3T(n-3) + 2) + 3 \cdot 2 + 2 = 3^3T(n-3) + 3^2 \cdot 2 + 3 \cdot 2 + 2 \end{aligned}$$

Ejemplos

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T(n-1) + 2 & \text{si } n \geq 2 \end{cases}$$

•

$$\begin{aligned} T(n) &\stackrel{1}{=} 3T(n-1) + 2 \\ &\stackrel{2}{=} 3(3T(n-2) + 2) + 2 = 3^2T(n-2) + 3 \cdot 2 + 2 \\ &\stackrel{3}{=} 3^2(3T(n-3) + 2) + 3 \cdot 2 + 2 = 3^3T(n-3) + 3^2 \cdot 2 + 3 \cdot 2 + 2 \\ &\vdots \\ &\stackrel{i}{=} 3^i T(n-i) + \sum_{j=0}^{i-1} 3^j \cdot 2 \end{aligned}$$

Ejemplos

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T(n-1) + 2 & \text{si } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} 3T(n-1) + 2 \\ &\stackrel{2}{=} 3(3T(n-2) + 2) + 2 = 3^2T(n-2) + 3 \cdot 2 + 2 \\ &\stackrel{3}{=} 3^2(3T(n-3) + 2) + 3 \cdot 2 + 2 = 3^3T(n-3) + 3^2 \cdot 2 + 3 \cdot 2 + 2 \\ &\vdots \\ &\stackrel{i}{=} 3^i T(n-i) + \sum_{j=0}^{i-1} 3^j \cdot 2 \\ &\vdots \quad \bullet \\ &\stackrel{n-1}{=} 3^{n-1} T(1) + \sum_{j=0}^{n-2} 3^j \cdot 2 \end{aligned}$$

Ejemplos

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T(n-1) + 2 & \text{si } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} 3T(n-1) + 2 \\ &\stackrel{2}{=} 3(3T(n-2) + 2) + 2 = 3^2T(n-2) + 3 \cdot 2 + 2 \\ &\stackrel{3}{=} 3^2(3T(n-3) + 2) + 3 \cdot 2 + 2 = 3^3T(n-3) + 3^2 \cdot 2 + 3 \cdot 2 + 2 \\ &\vdots \\ &\stackrel{i}{=} 3^i T(n-i) + \sum_{j=0}^{i-1} 3^j \cdot 2 \\ &\vdots \\ &\stackrel{n-1}{=} 3^{n-1} T(1) + \sum_{j=0}^{n-2} 3^j \cdot 2 = 3^{n-1} + 2 \cdot \frac{3 \cdot 3^{n-2} - 3^0}{3 - 1} \end{aligned}$$

Ejemplos

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T(n-1) + 2 & \text{si } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} 3T(n-1) + 2 \\ &\stackrel{2}{=} 3(3T(n-2) + 2) + 2 = 3^2 T(n-2) + 3 \cdot 2 + 2 \\ &\stackrel{3}{=} 3^2(3T(n-3) + 2) + 3 \cdot 2 + 2 = 3^3 T(n-3) + 3^2 \cdot 2 + 3 \cdot 2 + 2 \\ &\vdots \\ &\stackrel{i}{=} 3^i T(n-i) + \sum_{j=0}^{i-1} 3^j \cdot 2 \\ &\vdots \\ &\stackrel{n-1}{=} 3^{n-1} T(1) + \sum_{j=0}^{n-2} 3^j \cdot 2 = 3^{n-1} + 2 \cdot \frac{3 \cdot 3^{n-2} - 3^0}{3 - 1} \\ &= 2 \cdot 3^{n-1} - 1 \quad \text{Función exponencial.} \end{aligned}$$

Todos estos procesos no son necesarios aprendérselos de memoria, no lo vamos a hacer de esta forma

Ejemplos

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 3T(n-1) + 2 & \text{si } n \geq 2 \end{cases}$$

$$\begin{aligned} T(n) &\stackrel{1}{=} 3T(n-1) + 2 \\ &\stackrel{2}{=} 3(3T(n-2) + 2) + 2 = 3^2 T(n-2) + 3 \cdot 2 + 2 \\ &\stackrel{3}{=} 3^2(3T(n-3) + 2) + 3 \cdot 2 + 2 = 3^3 T(n-3) + 3^2 \cdot 2 + 3 \cdot 2 + 2 \\ &\vdots \\ &\stackrel{i}{=} 3^i T(n-i) + \sum_{j=0}^{i-1} 3^j \cdot 2 \\ &\vdots \\ &\stackrel{n-1}{=} 3^{n-1} T(1) + \sum_{j=0}^{n-2} 3^j \cdot 2 = 3^{n-1} + 2 \cdot \frac{3 \cdot 3^{n-2} - 3^0}{3 - 1} \\ &= 2 \cdot 3^{n-1} - 1 \in \Theta(3^n) \end{aligned}$$

Orden exponencial

Ejemplos

ESTE EJEMPLO ES MUY IMPORTANTE SE NOS TIENE QUE QUEDAR EN LA CABEZA.

$$T(n) = \begin{cases} 4 & \text{si } n = 1 \\ 2T(n/2) + 3n + 2 & \text{si } n \geq 2, \text{ potencia de 2} \end{cases}$$

Divide y vencerás.

si $n = 1$
si $n \geq 2$, potencia de 2

En este caso p. ej k = 1. Es decir, además de las llamadas recursivas se realiza una función de coste lineal.

Ejemplos

$$T(n) = \begin{cases} 4 & \text{si } n = 1 \\ 2T(n/2) + 3n + 2 & \text{si } n \geq 2, \text{ potencia de 2} \end{cases}$$

Haciendo sucesivos desplegados obtenemos

$$T(n) \stackrel{1}{=} 2T\left(\frac{n}{2}\right) + 3n + 2$$

Ejemplos

$$T(n) = \begin{cases} 4 & \text{si } n = 1 \\ 2T(n/2) + 3n + 2 & \text{si } n \geq 2, \text{ potencia de 2} \end{cases}$$

Haciendo sucesivos desplegados obtenemos

$$\begin{aligned} T(n) &\stackrel{1}{=} 2T\left(\frac{n}{2}\right) + 3n + 2 \\ &\stackrel{2}{=} 2\left(2T\left(\frac{n/2}{2}\right) + 3\frac{n}{2} + 2\right) + 3n + 2 \end{aligned}$$

Ejemplos

$$T(n) = \begin{cases} 4 & \text{si } n = 1 \\ 2T(n/2) + 3n + 2 & \text{si } n \geq 2, \text{ potencia de 2} \end{cases}$$

Haciendo sucesivos desplegados obtenemos

$$\begin{aligned} T(n) &\stackrel{1}{=} 2T\left(\frac{n}{2}\right) + 3n + 2 \\ &\stackrel{2}{=} 2\left(2T\left(\frac{n/2}{2}\right) + 3\frac{n}{2} + 2\right) + 3n + 2 = 2^2 T\left(\frac{n}{2^2}\right) + 2 \cdot 3n + 2^2 + 2 \end{aligned}$$

Ejemplos

$$T(n) = \begin{cases} 4 & \text{si } n = 1 \\ 2T(n/2) + 3n + 2 & \text{si } n \geq 2, \text{ potencia de 2} \end{cases}$$

Haciendo sucesivos desplegados obtenemos

$$\begin{aligned} T(n) &\stackrel{1}{=} 2T\left(\frac{n}{2}\right) + 3n + 2 \\ &\stackrel{2}{=} 2\left(2T\left(\frac{n/2}{2}\right) + 3\frac{n}{2} + 2\right) + 3n + 2 = 2^2 T\left(\frac{n}{2^2}\right) + 2 \cdot 3n + 2^2 + 2 \\ &\stackrel{3}{=} 2^2 \left(2T\left(\frac{n/2^2}{2}\right) + 3\frac{n}{2^2} + 2\right) + 2 \cdot 3n + 2^2 + 2 \end{aligned}$$

Ejemplos

$$T(n) = \begin{cases} 4 & \text{si } n = 1 \\ 2T(n/2) + 3n + 2 & \text{si } n \geq 2, \text{ potencia de 2} \end{cases}$$

Haciendo sucesivos desplegados obtenemos

$$\begin{aligned} T(n) &\stackrel{1}{=} 2T\left(\frac{n}{2}\right) + 3n + 2 \\ &\stackrel{2}{=} 2\left(2T\left(\frac{n/2}{2}\right) + 3\frac{n}{2} + 2\right) + 3n + 2 = 2^2 T\left(\frac{n}{2^2}\right) + 2 \cdot 3n + 2^2 + 2 \\ &\stackrel{3}{=} 2^2 \left(2T\left(\frac{n/2^2}{2}\right) + 3\frac{n}{2^2} + 2\right) + 2 \cdot 3n + 2^2 + 2 \\ &= 2^3 T\left(\frac{n}{2^3}\right) + 3 \cdot 3n + 2^3 + 2^2 + 2 \end{aligned}$$

Ejemplos

$$T(n) = \begin{cases} 4 & \text{si } n = 1 \\ 2T(n/2) + 3n + 2 & \text{si } n \geq 2, \text{ potencia de 2} \end{cases}$$

Haciendo sucesivos desplegados obtenemos

$$\begin{aligned} T(n) &\stackrel{1}{=} 2T\left(\frac{n}{2}\right) + 3n + 2 \\ &\stackrel{2}{=} 2\left(2T\left(\frac{n/2}{2}\right) + 3\frac{n}{2} + 2\right) + 3n + 2 = 2^2 T\left(\frac{n}{2^2}\right) + 2 \cdot 3n + 2^2 + 2 \\ &\stackrel{3}{=} 2^2 \left(2T\left(\frac{n/2^2}{2}\right) + 3\frac{n}{2^2} + 2\right) + 2 \cdot 3n + 2^2 + 2 \\ &= 2^3 T\left(\frac{n}{2^3}\right) + 3 \cdot 3n + 2^3 + 2^2 + 2 \\ &\stackrel{4}{=} 2^3 \left(2T\left(\frac{n/2^3}{2}\right) + 3\frac{n}{2^3} + 2\right) + 3 \cdot 3n + 2^3 + 2^2 + 2 \end{aligned}$$

Ejemplos

$$T(n) = \begin{cases} 4 & \text{si } n = 1 \\ 2T(n/2) + 3n + 2 & \text{si } n \geq 2, \text{ potencia de 2} \end{cases}$$

Haciendo sucesivos desplegados obtenemos

$$\begin{aligned} T(n) &\stackrel{1}{=} 2T\left(\frac{n}{2}\right) + 3n + 2 \\ &\stackrel{2}{=} 2\left(2T\left(\frac{n/2}{2}\right) + 3\frac{n}{2} + 2\right) + 3n + 2 = 2^2 T\left(\frac{n}{2^2}\right) + 2 \cdot 3n + 2^2 + 2 \\ &\stackrel{3}{=} 2^2 \left(2T\left(\frac{n/2^2}{2}\right) + 3\frac{n}{2^2} + 2\right) + 2 \cdot 3n + 2^2 + 2 \\ &= 2^3 T\left(\frac{n}{2^3}\right) + 3 \cdot 3n + 2^3 + 2^2 + 2 \\ &\stackrel{4}{=} 2^3 \left(2T\left(\frac{n/2^3}{2}\right) + 3\frac{n}{2^3} + 2\right) + 3 \cdot 3n + 2^3 + 2^2 + 2 \\ &= 2^4 T\left(\frac{n}{2^4}\right) + 4 \cdot 3n + 2^4 + 2^3 + 2^2 + 2^1 \\ &\vdots \end{aligned}$$

Ejemplos

$$\begin{aligned} &= 2^i T\left(\frac{n}{2^i}\right) + i \cdot 3n + \sum_{j=1}^i 2^j \\ &= 2^i T\left(\frac{n}{2^i}\right) + i \cdot 3n + 2^{i+1} - 2. \end{aligned}$$

Ejemplos

$$\begin{aligned} &= 2^i T\left(\frac{n}{2^i}\right) + i \cdot 3n + \sum_{j=1}^i 2^j \\ &= 2^i T\left(\frac{n}{2^i}\right) + i \cdot 3n + 2^{i+1} - 2. \end{aligned}$$

Al caso básico se llega cuando $\frac{n}{2^i} = 1$, es decir, cuando $i = \log n$.

Ejemplos

$$\begin{aligned} &= 2^i T\left(\frac{n}{2^i}\right) + i \cdot 3n + \sum_{j=1}^i 2^j \\ &= 2^i T\left(\frac{n}{2^i}\right) + i \cdot 3n + 2^{i+1} - 2. \end{aligned}$$

Al caso básico se llega cuando $\frac{n}{2^i} = 1$, es decir, cuando $i = \log n$.

Por tanto para n potencia de 2,

$$\begin{aligned} T(n) &\stackrel{\log n}{=} 2^{\log n} T(1) + 3n \log n + 2^{1+\log n} - 2 \\ &= 4n + 3n \log n + 2n - 2 \\ &= 3n \log n + 6n - 2 \end{aligned}$$

Ejemplos

$$\begin{aligned} &= 2^i T\left(\frac{n}{2^i}\right) + i \cdot 3n + \sum_{j=1}^i 2^j \\ &= 2^i T\left(\frac{n}{2^i}\right) + i \cdot 3n + 2^{i+1} - 2. \end{aligned}$$

Al caso básico se llega cuando $\frac{n}{2^i} = 1$, es decir, cuando $i = \log n$.

Por tanto para n potencia de 2,

$$\begin{aligned} T(n) &\stackrel{\log n}{=} 2^{\log n} T(1) + 3n \log n + 2^{1+\log n} - 2 \\ &= 4n + 3n \log n + 2n - 2 \\ &= 3n \log n + 6n - 2 \\ &\in \Theta(n \log n). \end{aligned}$$


Más sencillo aplicar, en este caso, el teorema de la división. Y en el anterior caso aplicar el teorema de la resta.

Teorema de la resta

Está el teorema de la resta y el de la división. Siempre hay preguntas sobre esto. Hay que saberse el teorema. Tanto el de la resta como el de la división. Van a caer esas preguntas, tanto en el tipo test, como en el práctico para saber el coste de una recurrencia

Cuando

Lo mejor es aprendérselo de memoria.

- la descomposición recursiva se obtiene restando una cantidad constante,
- el caso directo tiene coste constante,
- la preparación de las llamadas y la combinación de los resultados tiene coste polinómico, o también exponencial

tenemos

$$T(n) = \begin{cases} k_0 & \text{si } 0 \leq n < b \\ a \cdot T(n - b) + k_1 \cdot n^k & \text{si } n \geq b \end{cases}$$

Teorema de la resta

Cuando

Para aplicar este teorema, se deben cumplir esos 3 enunciados

- la descomposición recursiva se obtiene restando una cantidad constante,
- el caso directo tiene coste constante, el caso trivial
- la preparación de las llamadas y la combinación de los resultados tiene coste polinómico,

tenemos

$$T(n) = \begin{cases} k_0 & \text{si } 0 \leq n < b \\ a T(n - b) + k_1 \cdot n^k & \text{si } n \geq b \end{cases}$$

Coste de lo ajeno a lo recursivo

Constante entre esos valores

Y de esa forma cuando $n \geq b$

Entonces

Coste $T(n) \in \begin{cases} \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^n / b) & \text{si } a > 1 \end{cases}$

O es es polinómico o es exponencial.

$a^{n/b}$

Aprendérnoslo de memoria

Teorema de la división

Segunda que nos tenemos que saber.

Es el caso de los ejercicios de divide y vencerás del siguiente tema, tema 5.

Si la descomposición se obtiene dividiendo por una cantidad constante $b \geq 2$,

$$T(n) = \begin{cases} k_0 & \text{si } 0 \leq n < b \\ a \cdot T\left(\frac{n}{b}\right) + k_1 \cdot n^k & \text{si } n \geq b \end{cases}$$

Teorema de la división

El ejemplo que hemos visto de $n/2$ encaja en esta transparencia.

Si la descomposición se obtiene dividiendo por una cantidad constante $b \geq 2$,

$$T(n) = \begin{cases} k_0 & \text{si } 0 \leq n < b \\ a \cdot T\left(\frac{n}{b}\right) + k_1 \cdot n^k & \text{si } n \geq b \end{cases}$$

Entonces

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log n) & \text{si } a = b^k \\ \Theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

búsqueda binaria.

El coste es menor cuanto más pequeñas son a y k y más grande es b .

El teorema es el mismo solo que uno hace $(T(n-b))$ y el otro hace $T(n/b)$). Pero sigue de la misma forma ($T(n) = a^*T(n-b)+k1*n^k$) ($T(n) = a*T(n/b)+k1*n^k$).

Lo importante es que dependiendo del tipo de algoritmo que se nos plantee Hay que saber el coste de ambos.

Generalización de parámetros y/o resultados

Importante ya que lo vamos a necesitar. Aquí vamos a ver técnicas recursivas que nos servirán en estructura de datos.

Generalización

La generalización de una función consiste en añadir parámetros y/o resultados adicionales de forma que la nueva función calcula lo mismo que la original para determinados valores iniciales de los parámetros adicionales, y quizá resultados adicionales.

Una forma de generalización es añadir una variable acumuladora

La generalización de una función consiste en añadir parámetros y/o resultados adicionales de forma que la nueva función calcula lo mismo que la original para determinados valores iniciales de los parámetros adicionales, y quizás resultados adicionales.

La generalización tiene diversas aplicaciones:

- Diseño de algoritmos recursivos.
- Mejorar la eficiencia de los algoritmos recursivos.
- Generar versiones recursivas finales, para las cuales algunos compiladores pueden generar código más eficiente.
- Hacer que el algoritmo sea más general.

Ejemplo

```
f 5  
f 4  
f 3  
f 2  
f 1  
f 0  
  
{n ≥ 0}  
int factorial(int n)  
    int f;  
    if (n == 0) { f = 1; }  
    else  
        { f = n * factorial(n-1); }  
    return f;  
}  
  
{f = n!}
```

versión no final del factorial.

```
factorial(n) = facFinal(n,1)
```

La recursión final es equivalente a un bucle.

Veremos más ejemplos de estos,
La de la izquierda es no final porque lo último que hacemos es una multiplicación de $n * \text{el factorial}$ acumulado. Claro entonces lo del chat gpt que decía arriba está mal. Es no final porque lo último que hace es la multiplicación.

```
5  
5*4  
5*4*3  
...*2  
...  
f 5  
f 4  
f 3  
f 2  
f 1  
f 0  
  
{n ≥ 0}  
int facFinal(int n, int ac)  
    int f;  
    if (n == 0) { f = ac; }  
    else  
        { f = facFinal(n-1, n * ac); }  
    return f;  
}  
  
{f = ac * n!}
```

Esta va acumulando el factorial

Esto en teoría es mejor que lo de la izquierda.

Ejemplo

Decimos que un vector de naturales es *pareado* si la diferencia entre el número de pares de su mitad izquierda y su mitad derecha no excede la unidad, y además ambas mitades son a su vez pareadas.

es decir, si $\text{abs}(\text{nParesIzq}-\text{nParesDcha}) \leq 1$

```
bool pareado (vector<int> const& v) { . . . }
```

Que ambas mitades sean pareadas implica que el número de pares de la primera mitad sea ≤ 1 y el número de pares de la segunda mitad sea ≤ 1 .

Ejemplo

Decimos que un vector de naturales es *pareado* si la diferencia entre el número de pares de su mitad izquierda y su mitad derecha no excede la unidad, y además ambas mitades son a su vez pareadas.

```
bool pareado (vector<int> const& v) { . . . }
```

Tenemos que tener cuidado porque la primera llamada siempre la hacen de esta forma.

La cuestión es cómo hacer referencia en la llamada recursiva a una mitad del vector: necesitamos saber cuándo un segmento válido cualquiera del vector es pareado.

```
bool pareado (vector<int> const& v, int c, int f) { . . . }
```

comienzo y final del vector

Los parámetros adicionales c y f nos dicen qué segmento de v , $[c..f]$, queremos ver si es pareado. En la precondición exigiremos $0 \leq c \leq f \leq v.size()$.

Va a ser un divide y vencerás. Analizaremos cada mitad por separado y luego en conjunto.

Ejemplo

Decimos que un vector de naturales es *pareado* si la diferencia entre el número de pares de su mitad izquierda y su mitad derecha no excede la unidad, y además ambas mitades son a su vez pareadas.

```
bool pareado (vector<int> const& v) { . . . }
```

La cuestión es cómo hacer referencia en la llamada recursiva a una mitad del vector: necesitamos saber cuándo un segmento válido cualquiera del vector es pareado.

```
bool pareado (vector<int> const& v, int c, int f) { . . . }  
comienzo (c) y fin (f) del vector que queremos saber si es pareado
```

Los parámetros adicionales c y f nos dicen qué segmento de v , $[c..f]$, queremos ver si es pareado. En la precondición exigiremos $0 \leq c \leq f \leq v.size()$.

La función original se puede implementar:

```
bool pareado (vector<int> const& v) {  
    return pareado(v, 0, v.size()); el tamaño del vector es v.size()  
}
```

c f

Aquí importante. En estos ejercicios no nos decía la c y la f al principio de la función pero lo tenemos que poner en la primera llamada recursiva.

Ejemplo: parámetros adicionales

Diseñamos ahora recursivamente la nueva función: necesitamos contar los pares de cada una de las dos mitades.

Calcula el número de pares del segmento del vector que le digamos

```
int contarPares(vector<int> const& v, int c, int f){  
    int pares=0;  
    for (int i=c; i<f; i++) { if (v[i]%2 ==0) { pares+=1; } }  
    return pares; }  
  
bool pareado (vector<int> const& v, int c, int f) {  
    bool b;  
    if (f <= c+1) { b = true; } caso base  
    else primera mitad y segunda mitad del vector  
        b = (abs(contarPares(v,c,m)-contarPares(v,m,f)) <= 1)  
            && pareado(v,c,m) && pareado(v,m,f);  
    return b; que ambos segmentos del vector sean pareados  
}
```



El resto del número div entre 2 ==0

Bucle for que recorre el vector y halla el número de pares del segmento



Importante que la $m = (c+f)/2$

ANALIZAREMOS MITAS

LA B LA PODRÍAMOS PONER COMO UN PARÁMETRO ADICIONAL, QUE PUEDA VARIAR.

Ejemplo: parámetros adicionales

Diseñamos ahora recursivamente la nueva función: necesitamos contar los pares de cada una de las dos mitades.

```
int contarPares(vector<int> const& v, int c, int f) {
    int pares=0;
    for (int i=c; i<f; i++) { if (v[i]%2 ==0) { pares+=1; } }
    return pares;
}

bool pareado (vector<int> const& v, int c, int f) {
    bool b;
    if (f <= c+1) { b = true; }
    else
        b = (abs(contarPares(v,c,m)-contarPares(v,m,f)) <= 1)
            && pareado(v,c,m) && pareado(v,m,f);
    return b;
}
```

Sea $n = f - c$

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ 2 * T(n \text{ div } 2) + k_2 * n & \text{si } n > 0 \end{cases}$$

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

Pero lo queremos hacer de coste constante $T(n) = O(n)$.

La llamada recursiva se realiza en dos ocasiones

En cada llamada recursiva el rango original se divide en dos mitades

si $n = 0$
si $n > 0$

La n es por la función contar pares

Como $a=2$ y $b=2$ entonces tiene ese coste.

Ejemplo: resultados adicionales

Estamos contando muchas veces el número de pares, cuando podríamos aprovechar los cálculos de las mitades de un segmento para saber el número de pares del segmento. Para mejorar la eficiencia de la función anterior añadiremos un resultado adicional: el número de pares del segmento $[c..f]$:

```
void pareado (vector<int> const& v, int c, int f, int &pares, bool &b)  
{...}
```

Es decir, en el caso base al que lleguemos veremos si el número es par o no y actualizaremos el número de pares

Ejemplo: resultados adicionales

Estamos contando muchas veces el número de pares, cuando podríamos aprovechar los cálculos de las mitades de un segmento para saber el número de pares del segmento. Para mejorar la eficiencia de la función anterior añadiremos un resultado adicional: el número de pares del segmento $[c..f]$:

```
void pareado (vector<int> const& v, int c, int f, int &pares, bool &b)  
{...}
```

Versión más general, diseño bueno de algoritmo recursivo, algoritmo recursivo final, y más general

Con esta nueva función, la función original se implementa:

```
bool pareado (vector<int> const& v) {  
    int pares; bool b;  
    pareado(v,0,v.size(),pares,b); //No usamos pares para nada  
    return b;  
}
```

Debe comprobar cuando llegue al caso base si el número en la posición $v[c]$ es par. Es decir, que $v[c]\%2==0$. Si es par, aumentar el número de pares.

Ejemplo: resultados adicionales

```
void pareado (vector<int> const& v, int c, int f, int &pares, bool &b){  
    //0<=c<=f<=v.size()  
    if (c == f) {b = true; pares = 0;} // Segmento vacio  
    else if (f == c+1){ // Segmento unitario  
        b = true;  
        if (v[c]%2 == 0) { pares =1; } else {pares = 0; }  
    }  
    else { // Segmento de longitud >= 2  
        int m = (c+f)/2; //punto medio  
        //Llamadas recursivas  
        bool b1,b2; int p1,p2;  
        pareado(v,c,m,b1,p1);  
        pareado(v,m,f,b2,p2);  
  
        //Calculo de los resultados  
        b = b1 && b2 && (abs(p1-p2) <= 1);  
        pares = p1+p2; No usamos el n mero de pares total, pero lo calculamos por si acaso se  
        necesita en un futuro.  
    }  
}
```

Por convenio as  lo debemos de poner.

Ejemplo: resultados adicionales

El coste de la nueva función:

Sea $n = f - c$

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ 2 * T(n \text{ div } 2) + k_2 & \text{si } n > 0 \end{cases}$$

$T(n) \in \Theta(n)$

Como en la llamada inicial $c = 0$ y $f = v.size()$, el coste de la función original está en $\Theta(v.size())$.

No hay función contar pares, se calcula durante la llamada recursiva directamente

del orden de n .

Claro porque aplicando el teorema de la división tenemos que $a = 2$, $b = 2$ y que $k = 0$; por tanto como $a > b^k$ $2 > 1$ entonces coste $n^{\log_b a}$. Como log en base 2 de 2 es 1 entonces $n^1 = n$. Por tanto el coste es n

Acordarnos siempre de que la búsqueda binaria se hace sobre un vector ordenado.

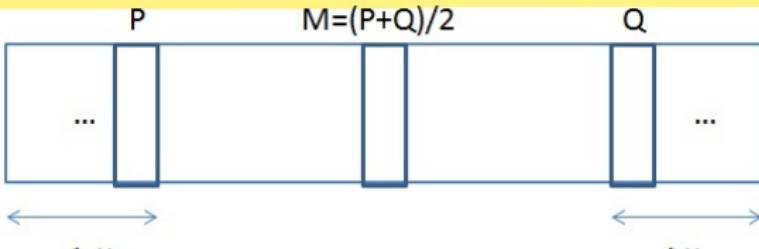
Búsqueda binaria

Algoritmo de búsqueda que trata de buscar un elemento sobre un vector ordenado. Para ello divide el vector en mitades continuamente hasta encontrar el elemento.

RECORDAR: El vector debe estar ordenado, no se hace como lo hacíamos en FP2, el algoritmo debe usar recursión y en este caso utilizará DV. Pero en cada llamada recursiva solo entrará en una llamada. Sería una llamada recursiva de tamaño la mitad.

Implementación recursiva de la búsqueda binaria

- Partimos de un vector ordenado, donde puede haber elementos repetidos, y un valor x que pretendemos encontrar en el vector.
- Buscamos la aparición más a la derecha del valor x , o, si no se encuentra en el vector, buscamos la posición anterior a donde se debería encontrar, por si queremos insertarlo.
- Estamos buscando el punto del vector donde las componentes pasan de ser $\leq x$ a ser $> x$. Si no se ha encontrado el valor entonces devolver la posición anterior a la que debería estar para poder insertarlo.

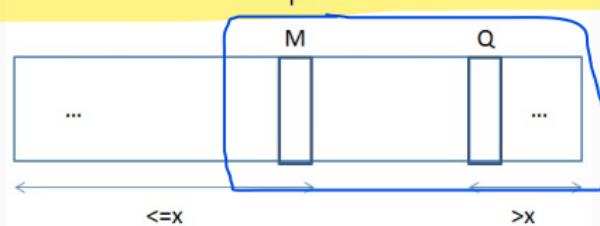


Esto es importante porque habrá veces donde no encontraremos el elemento que queremos. En ese caso hallamos la posición anterior a la que debería estar para, en caso de necesitarlo, poder insertarlo.

- La idea es que en cada comparación se reduce a la mitad el tamaño del subvector donde puede estar el elemento buscado.

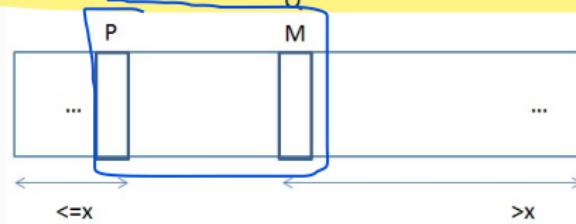
- Si $v[m] \leq x$ entonces debemos buscar a la derecha de m

Si el valor que buscamos es mayor que el valor que hay en la mitad, entonces buscar en la mitad derecha.



- Y si $v[m] > x$ entonces debemos buscar a la izquierda de m

Si el valor de la mitad es mayor que el valor que buscamos, entonces buscar en la mitad izquierda.



Ejemplo: búsqueda binaria

En este caso vamos a considerar que la búsqueda se realiza en un segmento cerrado $[c..f]$ por lo que exigiremos en la precondición $0 \leq c \leq f + 1 \leq v.size()$.

Ejemplo: búsqueda binaria

Recordamos que todo esto es sobre un vector ORDENADO.

En este caso vamos a considerar que la búsqueda se realiza en un segmento cerrado $[c..f]$ por lo que exigiremos en la precondición $0 \leq c \leq f + 1 \leq v.size()$.

Para el razonamiento sobre la corrección es fundamental saber sobre los elementos que hemos descartado:

- A la izquierda de c hemos dejado elementos \leq que el elemento buscado.
- A la derecha de f hemos dejado elementos $>$ que el elemento buscado.

Ejemplo: búsqueda binaria

Segmento cerrado por lo que corchetes.

En este caso vamos a considerar que la búsqueda se realiza en un segmento cerrado $[c..f]$ por lo que exigiremos en la precondición $0 \leq c \leq f + 1 \leq v.size()$.

Para el razonamiento sobre la corrección es fundamental saber sobre los elementos que hemos descartado:

- A la izquierda de c hemos dejado elementos \leq que el elemento buscado.
- A la derecha de f hemos dejado elementos $>$ que el elemento buscado.

Cuando $c = f + 1$ el segmento es vacío, el elemento no se encuentra en el vector, deberemos devolver $c - 1$, que es la última posición con un elemento $<$ que el elemento buscado.

Hay veces que el elemento que buscamos no se encuentra en el vector. En ese caso debemos devolver $c - 1$, lo cual es LA ÚLTIMA POSICIÓN MENOR QUE EL ELEMENTO BUSCADO.

Ejemplo: búsqueda binaria

Donde x recorre el segmento, c es el elemento inicial y f es el elemento final.

```
int buscaBin(vector<int> const& v, int x, int c, int f) {  
    int p, m;  
  
    if (c == f+1) Esto es si el elemento no se ha encontrado.  
    { p = c - 1; }  
    else { // c <= f En esa posición insertaríamos el elemento que NO hemos  
        m = (c+f) / 2; Esto es, si el elemento que buscamos es mayor o igual que el de la mitad,  
        if (v[m] <= x) entonces estará en la mitad de la derecha.  
            { p = buscaBin( v, x, m+1, f ); }  
        else Llamadas recursivas  
            { p = buscaBin( v, x, c, m-1 ); } Si no estará en la primera mitad.  
    return p;  
}  
  
int buscaBin(vector<int> const& v, int x)  
{ return buscaBin(v,x,0,v.size()-1); } ya que f es el último elemento del vector que es v.size()-1.
```

V.Si

Ejemplo: búsqueda binaria

Sea $n = f - c + 1$.

$$T(n) = \begin{cases} k_1 & \text{si } n = 0 \\ T(n \text{ div } 2) + k_2 & \text{si } n > 0 \end{cases}$$

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

Por tanto, la función original tiene coste en $\Theta(\log v.size())$

Como $a = 1$ y $b = 2$, la $k = 0$ porque no hay n (no hay otra llamada a una función externa) entonces $a = b^k \Rightarrow 1 = 2^0 \Rightarrow 1 = 1$ Entonces estamos en el caso de:

Coste = $O(n^k \log(n))$ como $k=0$ entonces es = $O(1 * \log(n))$

QUICKSORT

MERGESORT

Algoritmos avanzados de ordenación

En FP vimos algunos algoritmos de ordenación de vectores. En FAL veremos otros algoritmos de ordenación más avanzados y que son más eficientes que los que nosotros vimos (Eso creo).

Algoritmos avanzados de ordenación

Tenemos dos algoritmos de ordenación

Quicksort (ordenado rápido)

Creo que es más eficiente

Mergesort (ordenado por mezcla)

- La ordenación rápida (quicksort) y la ordenación por mezcla (mergesort) son dos algoritmos de ordenación de complejidad cuasilineal, $O(n \log n)$.
- Las ideas recursivas son similares en los dos algoritmos: para ordenar un vector se procesan por separado la mitad izquierda y la mitad derecha.

quicksort

- En la ordenación rápida, se colocan los elementos pequeños a la izquierda y los grandes a la derecha, y luego se sigue con cada mitad por separado. Elegimos un pivote.
- En la ordenación por mezcla, se ordena la mitad de la izquierda y la mitad de la derecha por separado y luego se mezclan los resultados.

Los algoritmos que utilizábamos nosotros en FP eran del orden de n^2 mucho más costosos que estos dos.

Algoritmos avanzados de ordenación

Quicksort

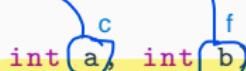
RECORDAR QUE ESTO SON ALGORITMOS DE ORDENACIÓN, Y QUE POR TANTO ES DISTINTO A LA BÚSQUEDA BINARIA YA QUE TRATA DE BUSCAR UN ELEMENTO SOBRE UN VECTOR ORDENADO DE VALORES.

Quicksort

- Como en el caso de la búsqueda binaria, la necesidad de encontrar un planteamiento recursivo nos lleva a definir un procedimiento auxiliar con los parámetros a y b , para así poder indicar el subvector del que nos ocupamos en cada llamada recursiva.

```
void quickSort ( vector<int> & v) {  
    quickSort(v, 0, v.size() - 1);  
} Ordenación de todo el vector
```

```
void quickSort(vector<int> & v, int a, int b)  
//0<=a<=b+1<=v.size()  
{...}
```



a: primer elemento en este caso
b: último elemento en este caso

Quicksort

- Como en el caso de la búsqueda binaria, la necesidad de encontrar un planteamiento recursivo nos lleva a definir un procedimiento auxiliar con los parámetros a y b , para así poder indicar el subvector del que nos ocupamos en cada llamada recursiva.

```
void quickSort ( vector<int> & v) {  
    quickSort(v, 0, v.size() - 1);  
}  
  
void quickSort(vector<int> & v, int a, int b)  
//0<=a<=b+1<=v.size()  
{...}
```

- El planteamiento recursivo consiste en:

- Elegir un pivote: un elemento cualquiera del subvector $v[a..b]$. Normalmente se elige $v[a]$ como pivote. primer elemento del vector, el vector puede o no estar ya ordenado.

El primer elemento de cada subvector.

Quicksort

- Como en el caso de la búsqueda binaria, la necesidad de encontrar un planteamiento recursivo nos lleva a definir un procedimiento auxiliar con los parámetros a y b , para así poder indicar el subvector del que nos ocupamos en cada llamada recursiva.

```
void quickSort ( vector<int> & v) {  
    quickSort(v, 0, v.size() - 1);  
}  
  
void quickSort(vector<int> & v, int a, int b)  
//0<=a<=b+1<=v.size()  
{...}
```

- El planteamiento recursivo consiste en:

- Elegir un pivote: un elemento cualquiera del subvector $v[a..b]$. Normalmente se elige $v[a]$ como pivote. $v[c]$, primer elemento del vector.
- Partitionar el subvector $v[a..b]$, colocando a la izquierda los elementos menores que el pivote y a la derecha los mayores. Los elementos iguales al pivote pueden quedar indistintamente a la izquierda o a la derecha.

Luego será mejor distinguir entre menores mayores e iguales que el pivote.

Quicksort

- Como en el caso de la búsqueda binaria, la necesidad de encontrar un planteamiento recursivo nos lleva a definir un procedimiento auxiliar con los parámetros a y b , para así poder indicar el subvector del que nos ocupamos en cada llamada recursiva.

```
void quickSort ( vector<int> & v) {  
    quickSort(v, 0, v.size() - 1); --->Importante, no es igual que la primera llamada  
} de la función. Nos tenemos que acordar.  
  
void quickSort(vector<int> & v, int a, int b)  
//0<=a<=b+1<=v.size()  
{...}
```

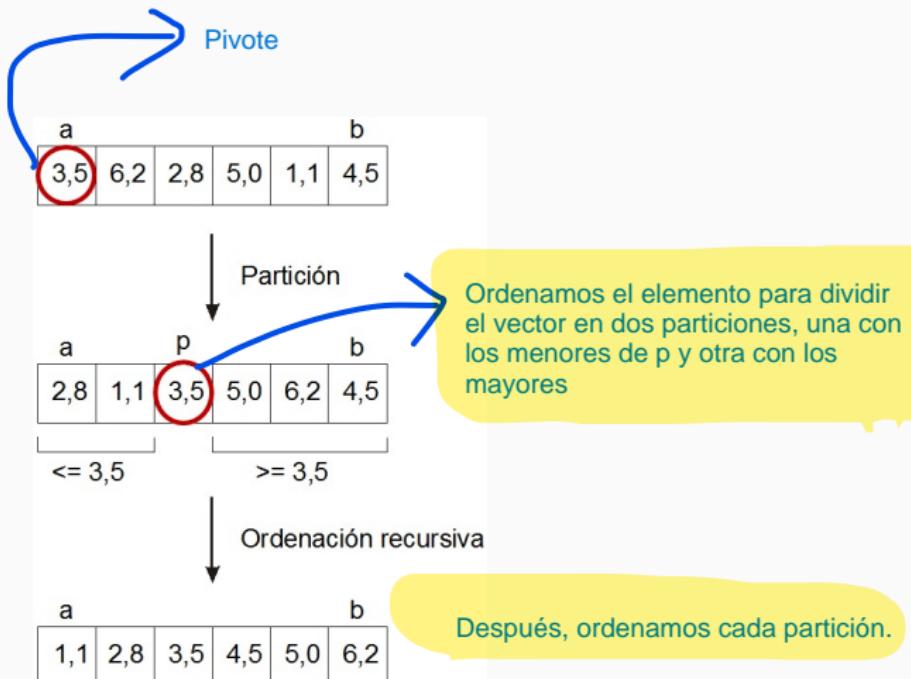
- El planteamiento recursivo consiste en:
 - Elegir un pivote: un elemento cualquiera del subvector $v[a..b]$. Normalmente se elige $v[a]$ como pivote. El primer elemento
 - Partitionar el subvector $v[a..b]$, colocando a la izquierda los elementos menores que el pivote y a la derecha los mayores. Los elementos iguales al pivote pueden quedar indistintamente a la izquierda o a la derecha.
Al final del proceso de partición, el pivote debe quedar en su posición definitiva p , separando los menores (situados en las posiciones $[a..p - 1]$) de los mayores (situados en las posiciones $[p + 1..b]$).

- Como en el caso de la búsqueda binaria, la necesidad de encontrar un planteamiento recursivo nos lleva a definir un procedimiento auxiliar con los parámetros a y b , para así poder indicar el subvector del que nos ocupamos en cada llamada recursiva.

```
void quickSort ( vector<int> & v) {  
    quickSort(v, 0, v.size() - 1);  
}  
  
void quickSort(vector<int> & v, int a, int b)  
//0<=a<=b+1<=v.size()  
{...}
```

- El planteamiento recursivo consiste en:
 - Elegir un pivote: un elemento cualquiera del subvector $v[a..b]$. Normalmente se elige $v[a]$ como pivote.
 - Partitionar el subvector $v[a..b]$, colocando a la izquierda los elementos menores que el pivote y a la derecha los mayores. Los elementos iguales al pivote pueden quedar indistintamente a la izquierda o a la derecha.
Al final del proceso de partición, el pivote debe quedar en su posición definitiva p , separando los menores (situados en las posiciones $[a..p - 1]$) de los mayores (situados en las posiciones $[p + 1..b]$).
 - Ordenar recursivamente los dos fragmentos que han quedado a la izquierda y a la derecha del pivote.

Quicksort



Quicksort

```
void quickSort( vector<int> & v, int a, int b ) {  
    int p;  
    p= v[a].  
    if ( a <= b ) {  
        particion(v, a, b, p); coge el elemento pivote y hace la partición.  
        quickSort(v, a, p-1); Ordenación de ambas mitades.  
        quickSort(v, p+1, b);  
    }  
}
```

Coste lineal.

ordenado de la mitad derecha

ordenado de la mitad izquierda

Esto es lo que tendremos que poner.

Coste de quicksort

- Ecuaciones de recurrencia en el caso peor: sea $n = b - a + 1$

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ T(n - 1) + c * n + c_0 & \text{si } n \geq 1 \end{cases}$$

En el caso peor :

Si solo ordena en menores y mayores.

$$T(n) \in \Theta(n^2)$$

Casos peores: vector ya ordenado, todos los elementos iguales.

Indica cuáles son casos peores del algoritmo quicksort que usa el primer elemento del segmento como pivote y que en la fase de partición reparte los elementos en menores, iguales y mayores el único caso es:

Los elementos ya están ordenados de forma estricta.

Correcta. El caso en que los elementos están ordenados a la inversa y son distintos es uno de los casos peores porque genera una única llamada recursiva con todos los elementos excepto el pivote, que queda en un extremo. El coste en este caso está en $O(n^2)$, siendo n el número de elementos del array.

Coste de quicksort

- Ecuaciones de recurrencia en el caso peor: sea $n = b - a + 1$

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ T(n-1) + c * n + c_0 & \text{si } n \geq 1 \end{cases}$$

En el caso peor :

$$T(n) \in \Theta(n^2)$$

a=1

k=1 O(n^k+1)

Casos peores: vector ya ordenado, todos los elementos iguales.

- Ecuaciones de recurrencia en el caso mejor:

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ 2 * T(n/2) + c * n & \text{si } n \geq 1 \end{cases}$$

En el caso mejor :

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

En el partición 2:

El caso mejor ocurre cuando el elemento pivote es igual a la mediana en cada llamada recursiva.
El mejor caso de PARTICIÓN2 ES que todos los elementos sean iguales.

En partición 1: coger la mediana

Coste de quicksort

- Ecuaciones de recurrencia en el caso peor: sea $n = b - a + 1$

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ T(n-1) + c * n + c_0 & \text{si } n \geq 1 \end{cases}$$

$T(0)$.

En el caso peor :

$T(n) \in \Theta(n^2)$
tanto creciente como decrecientemente

Casos peores: vector ya ordenado, todos los elementos iguales.

- Ecuaciones de recurrencia en el caso mejor:

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ 2 * T(n/2) + c * n & \text{si } n \geq 1 \end{cases}$$

coste de una particion.

En el caso mejor :

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

la llamada a particion

Se puede demostrar que en promedio se comporta como en el caso mejor.

Cambiando la política de elección del pivote se puede evitar que el caso peor sea un vector ordenado. Como por ejemplo lo que hemos dicho arriba, calcular la mediana

En este caso elegimos como pivote al primer elemento

O calcular la mediana.

Luego en el siguiente tema lo dividiremos en 3 partes: Menores estrictos, iguales y mayores estrictos.

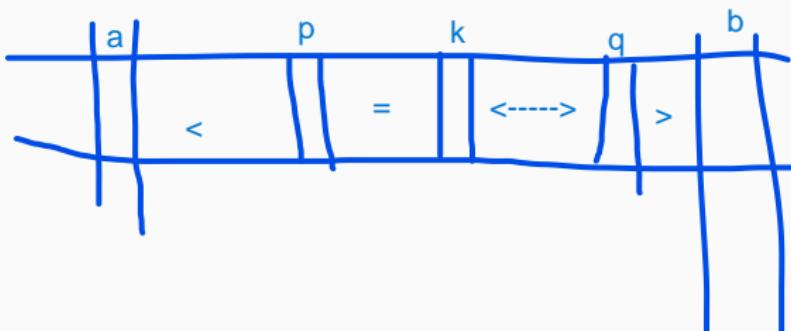
Partición mejorada

Particionamos el vector en tres partes que contienen los elementos estrictamente menores que el pivote, los iguales y los estrictamente mayores. El pivote se proporciona como argumento: caso peor: que los elementos estén ya ordenados de forma estricta.

```
//PRE: 0<=a<=b+1<=v.size()          c      f
void particion2(vector<int> & v, int a, int b, int pivote,
                 int& p, int& q) {
    int k; int aux;
    p=a; k=a; q=b;           p donde empiezan los iguales y q donde terminan los mismos.
    while (k<=q)           el k nos dice por donde vamos.
        //INV: a<=p<=k<=q+1<=b+1<=v.size() &&
        // menores(v,a,p-1) && iguales(v,p,k-1) && mayores(v,q+1,b)
    {
        if (v[k] == pivote) {k = k+1;}
        else if (v[k] < pivote)
            {aux = v[p]; v[p] = v[k]; v[k] = aux; p= p+1; k=k+1;}
        else {aux = v[q]; v[q] = v[k]; v[k] = aux; q=q-1;} no se si aquí faltaría el
    }                                k=k+1.
}                                v[k] > pivot.
//POST: a<=p<=q+1<=b+1<=v.size()&&
//menores(v,a,p-1) && iguales(v,p,q) && mayores(v,q+1,b)
```

Partición mejorada

```
void quickSort( vector<int> & v, int a, int b ) {  
    int p,q;  
  
    if ( a <= b ) {  
        particion(v, a, b, v[a], p, q);  
        quickSort(v, a, p-1);  
        quickSort(v, q+1, b);  
    }  
}
```



para distinguir menores,
iguales y mayores
particion2().

Algoritmos avanzados de ordenación

Mergesort

Mergesort

- Planteamos primero la generalización:

Siempre divide el vector por la mitad. Toma una mitad, toma otra y ordenalas.

```
void mergeSort ( vector<int> & v) {  
    mergeSort(v, 0, v.size() -1);  
}
```

```
void mergeSort(vector<int> & v, int a, int b)  
//0<=a<=b+1<=v.size()  
{...}
```

- Planteamos primero la generalización:

```
void mergeSort ( vector<int> & v) {  
    mergeSort(v, 0, v.size() -1);  
}  
  
void mergeSort(vector<int> & v, int a, int b)  
//0<=a<=b+1<=v.size()  
{...}
```

- Planteamiento recursivo. Para ordenar el subvector $v[a..b]$

- Obtenemos el punto medio m entre a y b , y ordenamos recursivamente los subvectores $v[a..m]$ y $v[(m + 1)..b]$. En vez de coger pivote, la posición del medio.
- Mezclamos ordenadamente los subvectores $v[a..m]$ y $v[(m + 1)..b]$ ya ordenados. ordenamos mitad izquierda y mitad derecha por separado y después ordenamos todo.

Mergesort

a	m	m+1	b
3,5	6,2	2,8	5,0

ordena la primera mitad

↓
Ordenación
recursiva de las
dos mitades

a	m	m+1	b
2,8	3,5	6,2	1,1

ordena la segunda mitad

↓
Mezcla

a	b
1,1	2,8

Mezclamos

Mergesort

```
void mergeSort(vector<int> & v, int a, int b ) {  
    int m;  
  
    if ( a < b ) {  
        m = (a+b) / 2;  
        mergeSort( v, a, m ); ordenamos izquierda  
        mergeSort( v, m+1, b );ordenamos derecha  
        mezcla( v, a, m, b ); ordenamos todo.  
    }  
}
```

Coste de mergesort

- Ecuaciones de recurrencia: tomamos $n = b - a + 1$

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ 2 * T(n/2) + c * n & \text{si } n \geq 1 \end{cases}$$

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

En todos los casos es de coste $n \log n$.