
Fundamentos de Algoritmia



APUNTES DE CLASE

Ricardo Peña Mari
Marco Antonio Gómez Martín
Gonzalo Méndez Pozo
Manuel Freire Morán
Antonio Sánchez Ruiz-Granados
Isabel Pita Andreu
Ramón González del Campo
Miguel Valero Espada
Clara Segura Díaz
Miguel Gómez-Zamalloa

Facultad de Informática
Universidad Complutense de Madrid

7 de septiembre de 2023

Documento maquetado con T_EX_S v.1.0+.

Este documento está preparado para ser imprimido a doble cara.

Fundamentos de Algoritmia

Apuntes de clase

Grado en Ingeniería de Computadores
Grado en Ingeniería Informática
Grado en Ingeniería del Software
Doble grado de Matemáticas e Ingeniería Informática

Facultad de Informática
Universidad Complutense de Madrid

7 de septiembre de 2023

Copyright © Ricardo Peña Mari, Gonzalo Méndez Pozo, Isabel Pita Andreu, Ramón
González del Campo, Miguel Valero Espada, Clara Segura Díaz, Miguel Gómez-Zamalloa

ISBN 978-84-697-0852-1

Índice

1. Análisis de la eficiencia	1
1. Introducción	1
2. Medidas asintóticas de la eficiencia	3
3. Jerarquía de órdenes de complejidad	6
4. Propiedades de los órdenes de complejidad	8
Notas bibliográficas	9
Ejercicios	9
2. Especificación de algoritmos	15
1. Introducción	15
2. Predicados	17
3. Significado	19
4. Ejemplos de especificación	20
Notas bibliográficas	23
Ejercicios	23
3. Diseño de algoritmos iterativos	27
1. Introducción	27
2. Reglas prácticas para el cálculo de la eficiencia	28
2.1. Ejemplos de cálculo de complejidad de algoritmos iterativos	28
3. Verificación	32
3.1. Semántica de un lenguaje imperativo	32
3.2. Reglas específicas para el cálculo de la precondition más débil	33
3.3. Ejemplos	34
3.4. Bucles e invariantes	34
3.5. Ejemplos	35
4. Derivación	38
4.1. Ejemplos	39
5. Derivación de algoritmos iterativos típicos	41
5.1. Uso de variables acumuladoras para evitar bucles anidados	41
5.2. Cálculo de la moda	44
5.3. Segmento de longitud máxima	47
5.4. Algoritmo de partición	49
5.5. Mezcla de dos vectores ordenados	51

5.6.	Algoritmos de matrices	53
5.7.	Algoritmos numéricos	54
	Notas bibliográficas	56
	Ejercicios	56
4.	Diseño de algoritmos recursivos	63
1.	Introducción	63
1.1.	Recursión simple	64
1.2.	Recursión final	67
1.3.	Recursión múltiple	69
1.4.	Resumen de los distintos tipos de recursión	70
2.	Diseño de algoritmos recursivos	70
2.1.	Implementación recursiva de la búsqueda binaria	75
2.2.	Algoritmos avanzados de ordenación	81
3.	Análisis de la complejidad de algoritmos recursivos	86
3.1.	Ecuaciones de recurrencias	86
3.2.	Despliegue de recurrencias	87
3.3.	Resolución general de recurrencias	89
4.	Técnicas de generalización de algoritmos recursivos	91
4.1.	Planteamientos recursivos finales	93
4.2.	Generalización por razones de eficiencia	94
5.	Verificación de algoritmos recursivos	97
	Notas bibliográficas	98
	Ejercicios	99
5.	Divide y Vencerás	105
1.	Introducción	105
2.	Ejemplos de aplicación del esquema con éxito	107
3.	Problema de selección	108
4.	Organización de un campeonato	112
4.1.	Implementación	113
5.	El problema del par más cercano	114
5.1.	Corrección	115
5.2.	Implementación	116
6.	La determinación del umbral	120
	Notas bibliográficas	122
	Ejercicios	122
6.	Vuelta Atrás	127
1.	Motivación	127
2.	Introducción	128
2.1.	Esquema básico de la <i>vuelta atrás</i>	130
2.2.	Resolución del problema de las palabras	131
3.	Vuelta atrás con marcaje	132
4.	Ejemplo: problema de las n -reinas	133
5.	Ejemplo de búsqueda de una sola solución: Dominó	135
6.	Ejemplo que no necesita desmarcar: El laberinto	137

7.	Optimización	139
7.1.	Ejemplo: Problema del viajante	139
7.2.	Ejemplo: Problema de la mochila	141
8.	Para terminar...	142
	Notas bibliográficas	143
	Ejercicios	143
Bibliografía		147

Índice de figuras

1.	Crecimiento de distintas funciones de complejidad	7
2.	Jerarquía de órdenes de complejidad	8
1.	Diseño de <i>particion</i>	50
2.	Diseño de <i>mezcla</i>	52
1.	Ejecución de <i>factorial(3)</i>	65
2.	Llamadas recursivas de <i>factorial(3)</i>	65
3.	Vuelta de las llamadas recursivas de <i>factorial(3)</i>	66
4.	Ejecución de <i>fib(4)</i>	70
5.	Cálculo del punto medio	75
6.	Búsqueda en la mitad derecha	75
7.	Búsqueda en la mitad izquierda	76
8.	Planteamiento del algoritmo <i>quicksort</i>	82
9.	Planteamiento del algoritmo <i>mergesort</i>	84
1.	Solución gráfica del problema del torneo	113
2.	Razonamiento de corrección del problema del par más cercano	116

Análisis de la eficiencia de los algoritmos ¹

No puede haber orden cuando hay mucha prisa

Seneca

RESUMEN: En este tema se muestra la importancia de contar con algoritmos eficientes, se define qué se entiende por coste de un algoritmo y se enseña a comparar las funciones de coste de distintos algoritmos con el fin de decidir cuál de ellos es preferible. Se define una jerarquía de órdenes de complejidad que ilustra la separación entre algoritmos eficientes y los que no lo son.

1. Introducción

- ★ Aproximadamente cada año y medio se duplica el número de instrucciones por segundo que son capaces de ejecutar los computadores. Ello puede inducir a pensar que basta con esperar algunos años para que problemas que hoy necesitan muchas horas de cálculo puedan resolverse en pocos segundos.
- ★ Sin embargo hay algoritmos tan ineficientes que ningún avance en la velocidad de las máquinas podrá conseguir para ellos tiempos aceptables. El factor predominante que delimita lo que es soluble en un tiempo razonable de lo que no lo es, es precisamente el **algoritmo** elegido para resolver el problema.
- ★ En este capítulo enseñaremos a medir la **eficiencia** de los algoritmos y a comparar la eficiencia de distintos algoritmos para un mismo problema. Después de la **corrección**, conseguir eficiencia debe ser el principal objetivo del programador. Mediremos principalmente la eficiencia en **tiempo de ejecución**, pero los mismos conceptos son aplicables a la medición de la eficiencia en **espacio**, es decir a medir la memoria que necesita el algoritmo.
- ★ La siguiente función ordena los n primeros elementos del vector v que recibe como parámetro utilizando el método de seleccion:

¹Ricardo Peña es el autor principal de este tema.

```

1 void ordenaSeleccion(int v[], int n) {
2
3     for (int i = 0; i < n-1; i++) {
4         // pmin calcula la posición del mínimo de v[i..n-1]
5         int pmin = i;
6         for (int j = i+1; j < n; j++)
7             if (v[j] < v[pmin])
8                 pmin = j;
9
10        // Tenemos la posición del mínimo. Lo ponemos en v[i]
11        // Código equivalente a swap(v[i], v[pmin])
12        int temp = v[i];
13        v[i] = v[pmin];
14        v[pmin] = temp;
15    }
16 }

```

- ★ Una manera de medir la eficiencia en tiempo de esta función es **contar** cuántas instrucciones de cada tipo se ejecutan, multiplicar este número por el tiempo que emplea la instrucción en ejecutarse, y realizar la suma para los diferentes tipos. Sean

$$t_a = \text{tiempo de una asignación entre enteros} \quad (1.1)$$

$$t_c = \text{tiempo de una comparación entre enteros}$$

$$t_i = \text{tiempo de incrementar un entero}$$

$$t_v = \text{tiempo de acceso a un elemento de un vector}$$

- La línea (3) da lugar a una asignación, a $n-1$ incrementos y a n comparaciones, es decir, a un tiempo $t_a + (n-1)t_i + nt_c$.
- La línea (5) da lugar a un tiempo $(n-1)t_a$.
- El bucle interior `for` se ejecuta $n-1$ veces, cada una con un valor diferente de i . Para cada valor de i y siguiendo el cálculo hecho para la (3), la línea (6) da lugar a un tiempo $t_a + (n-i-1)t_i + (n-i)t_c$.
- La línea (7) da, para cada valor de i , un tiempo mínimo de $(n-i-1)(2t_v + t_c)$, suponiendo que la instrucción `pmin = j` nunca se ejecuta. A ello hay que sumar $(n-i-1)t_a$ en el caso más desfavorable en que dicha rama se ejecute todas las veces. El caso promedio tendrá un tiempo de ejecución entre estos dos.
- Finalmente, la línea (9) dará lugar a un tiempo $(n-1)(4t_v + 3t_a)$.
- Por tanto, el tiempo del bucle interior `for`, en el caso más desfavorable, se calcula mediante el siguiente sumatorio:

$$\sum_{i=0}^{n-2} (t_a + t_c + (n-i-1)(t_i + 2t_v + t_a + 2t_c)) = P(n-1) + \frac{1}{2}Qn(n-1)$$

siendo $P = t_a + t_c$ y $Q = t_i + 2t_v + t_a + 2t_c$.

Para no cansar al lector con tediosos cálculos, concluiremos que la suma de todos estos tiempos da lugar a dos polinomios de la forma:

$$\begin{aligned} T_{min} &= An^2 - Bn + C \\ T_{max} &= A'n^2 - B'n + C' \end{aligned}$$

donde A , A' , B , B' , C y C' son expresiones racionales positivas que dependen linealmente de los tiempos elementales descritos en 1.1.

- ★ En este sencillo ejemplo se observan claramente los tres factores de los que en general depende el tiempo de ejecución de un algoritmo:
 1. El **tamaño** de los datos de entrada, simbolizado aquí por la longitud n del vector.
 2. El **contenido** de los datos de entrada, que en el ejemplo hace que el tiempo para diferentes vectores del mismo tamaño esté comprendido entre los valores T_{min} y T_{max} .
 3. El código generado por el **compilador** y el **computador** concreto utilizados, que afectan a los tiempos elementales 1.1.
- ★ Como el objetivo es poder comparar algoritmos independientemente del valor de los datos de entrada, el segundo factor podemos eliminarlo de dos maneras:
 - O bien midiendo solo el **caso peor**, es decir la ejecución que tarde más tiempo de todos los ejemplares de tamaño n .
 - O bien midiendo todos los casos de tamaño n y calculando el tiempo del **caso promedio**.

En esta asignatura nos concentraremos en el caso peor por dos razones:

1. El caso peor establece una cota superior fiable para **todos** los casos del mismo tamaño.
2. El caso peor es más fácil de calcular.

El caso promedio es más difícil de calcular pero a veces es más informativo. Además exige conocer la probabilidad con la que se va a presentar cada caso. Muy raramente puede ser útil conocer el **caso mejor** de un algoritmo para un tamaño n dado. Ese coste es una *cota inferior* al coste de cualquier otro ejemplar de ese tamaño.

- ★ El tercer factor impediría comparar algoritmos escritos en diferentes lenguajes, traducidos por diferentes compiladores, o ejecutados en diferentes máquinas. El criterio que seguiremos es **ignorar** estos factores.
- ★ Por tanto solo mediremos la eficiencia de un algoritmo en función del **tamaño** de los datos de entrada. Este criterio está en la base de lo que llamaremos **medida asintótica** de la eficiencia.

2. Medidas asintóticas de la eficiencia

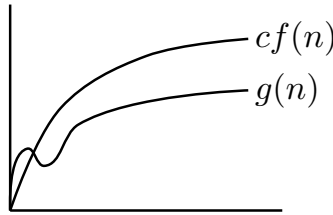
- ★ El **criterio asintótico** para medir la eficiencia de los algoritmos tiene como objetivo comparar algoritmos **independientemente** de los lenguajes en que están escritos, de las máquinas en que se ejecutan y del valor concreto de los datos que reciben como entrada. Tan solo considera importante el **tamaño** de dichos datos. Para cada problema habrá que definir qué se entiende por tamaño del mismo.
- ★ Se basa en tres principios:

1. El coste o eficiencia es una función que solo depende del tamaño de la entrada, e.g. $f(n) = n^2$.
 2. Las constantes multiplicativas o aditivas no se tienen en cuenta, e.g. $f(n) = n^2$ y $g(n) = 3n^2 + 27$ se consideran costes equivalentes.
 3. La comparación entre funciones de coste se hará para valores de n **suficientemente grandes**, es decir los costes para tamaños pequeños se consideran irrelevantes.
- ★ Sea \mathbb{N} el conjunto de los números naturales y \mathbb{R}^+ el conjunto de los reales estrictamente positivos.

Definición 1.1 Sea $f : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\}$. El conjunto de las funciones **del orden de** $f(n)$, denotado $\mathcal{O}(f(n))$, se define como:

$$\mathcal{O}(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\} \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} . \forall n \geq n_0 . g(n) \leq cf(n)\}$$

Asímismo, diremos que una función g es **del orden de** $f(n)$ cuando $g \in \mathcal{O}(f(n))$. También diremos que g **está en** $\mathcal{O}(f(n))$.



- ★ Generalizando, admitiremos también que una función negativa o indefinida para un número finito de valores de n pertenece al conjunto $\mathcal{O}(f(n))$ si eligiendo n_0 suficientemente grande, satisface la definición.
- ★ Esta garantiza que, si el tiempo de ejecución $g(n)$ de una implementación concreta de un algoritmo es del orden de $f(n)$, entonces el tiempo $g'(n)$ de cualquier otra implementación del mismo que difiera de la anterior en el lenguaje, el compilador, o/y la máquina empleada, también será del orden de $f(n)$. Por tanto, el coste $\mathcal{O}(f(n))$ expresa la eficiencia del algoritmo *per se*, no el de una implementación concreta del mismo.
- ★ Las clases $\mathcal{O}(f(n))$ para diferentes funciones $f(n)$ se denominan **clases de complejidad**, u **órdenes de complejidad**. Algunos órdenes tienen nombre propio. Así, al orden de complejidad $\mathcal{O}(n)$ se le llama **lineal**, al orden $\mathcal{O}(n^2)$, **cuadrático**, el orden $\mathcal{O}(1)$ describe la clase de las funciones **constantes**, etc. Eligiremos como representante del orden $\mathcal{O}(f(n))$ la función $f(n)$ **más sencilla** posible dentro del mismo.
- ★ Nótese que la definición 1.1 se puede aplicar tanto a un análisis en el caso peor, como a un análisis en el caso promedio. Por ejemplo, hay algoritmos cuyo coste en tiempo está en $\mathcal{O}(n^2)$ en el caso peor y en $\mathcal{O}(n \log n)$ en el caso promedio.
- ★ Nótese también que las unidades en que se mide el coste en tiempo (horas, segundos, milisegundos, etc.), o en memoria (octetos, palabras, celdas de longitud fija, etc.) **no**

son relevantes en la complejidad asintótica: dos unidades distintas se diferencian en una constante multiplicativa (e.g. $120 n^2$ segundos son $2 n^2$ minutos, ambos en $\mathcal{O}(n^2)$).

- ★ Aplicando directamente la definición de $\mathcal{O}(f(n))$, demostremos que $(n+1)^2 \in \mathcal{O}(n^2)$. Un modo de hacerlo es por inducción sobre n . Elegimos $n_0 = 1$ y $c = 4$, es decir demostraremos $\forall n \geq 1. (n+1)^2 \leq 4n^2$:

Caso base: $n = 1, (1+1)^2 \leq 4 \cdot 1^2$

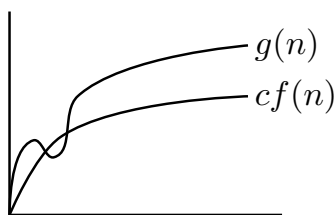
Paso inductivo: h.i. $(n+1)^2 \leq 4n^2$. Demostrémoslo para $n+1$:

$$\begin{aligned} (n+1+1)^2 &\leq 4(n+1)^2 \\ (n+1)^2 + 1 + 2(n+1) &\leq 4n^2 + 4 + 8n \\ (n+1)^2 &\leq 4n^2 + \underbrace{6n+1}_{\geq 0} \end{aligned}$$

- ★ También podemos probar que $3^n \notin \mathcal{O}(2^n)$. Si perteneciera, existiría $c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ tales que $3^n \leq c \cdot 2^n$ para todo $n \geq n_0$. Esto implicaría que $(\frac{3}{2})^n \leq c$ para todo $n \geq n_0$. Pero esto es falso porque dado un c cualquiera, bastaría tomar $n > \log_{1.5} c$ para que $(\frac{3}{2})^n > c$, es decir $(\frac{3}{2})^n$ no se puede acotar superiormente.
- ★ La notación $\mathcal{O}(f(n))$ nos da una cota superior al tiempo de ejecución $t(n)$ de un algoritmo. Normalmente estaremos interesados en la **menor** función $f(n)$ tal que $t(n) \in \mathcal{O}(f(n))$. Una forma de realizar un análisis más completo es encontrar además la **mayor** función $g(n)$ que sea una cota inferior de $t(n)$. Para ello introducimos la siguiente medida.

Definición 1.2 Sea $f : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\}$. El conjunto $\Omega(f(n))$, leído **omega de $f(n)$** , se define como:

$$\Omega(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\} \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}. \forall n \geq n_0. g(n) \geq cf(n)\}$$



- ★ Es frecuente confundir la medida $\mathcal{O}(f(n))$ como aplicable al caso peor y la medida $\Omega(f(n))$ como aplicable al caso mejor. Esta idea es **errónea**. Aplicaremos **ambas medidas al caso peor** (también podríamos aplicar ambas al caso promedio, o al caso mejor). Si el tiempo $t(n)$ de un algoritmo en el caso peor está en $\mathcal{O}(f(n))$ y en $\Omega(g(n))$, lo que estamos diciendo es que $t(n)$ no puede valer más que $c_1 f(n)$, ni menos que $c_2 g(n)$, para dos constantes apropiadas c_1 y c_2 y valores de n suficientemente grandes.
- ★ Es fácil demostrar (ver ejercicios) el llamado **principio de dualidad**: $g(n) \in \mathcal{O}(f(n))$ si y solo si $f(n) \in \Omega(g(n))$.

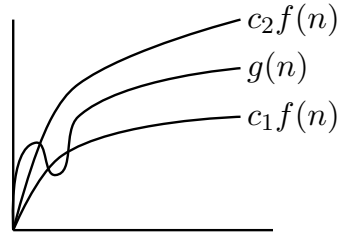
- ★ Sucede con frecuencia que una misma función $f(n)$ es a la vez cota superior e inferior del tiempo $t(n)$ (peor, promedio, etc.) de un algoritmo. Para tratar estos casos, introducimos la siguiente medida.

Definición 1.3 El conjunto de funciones $\Theta(f(n))$, leído del orden exacto de $f(n)$, se define como:

$$\Theta(f(n)) = \mathcal{O}(f(n)) \cap \Omega(f(n))$$

También se puede definir como:

$$\Theta(f(n)) = \{g : \mathbb{N} \longrightarrow \mathbb{R}^+ \cup \{0\} \mid \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}. \\ \forall n \geq n_0. c_1 f(n) \leq g(n) \leq c_2 f(n)\}$$



- ★ Siempre que sea posible, daremos el orden exacto del coste de un algoritmo, por ser más informativo que dar solo una cota superior.

3. Jerarquía de órdenes de complejidad

- ★ Es importante visualizar las implicaciones prácticas de que el coste de un algoritmo pertenezca a una u otra clase de complejidad. La Figura 1 muestra el crecimiento de algunas de estas funciones, suponiendo que expresan un tiempo en milisegundos (ms = milisegundos, s = segundos, m = minutos, h = horas, etc.).
- ★ Se aprecia inmediatamente la **extraordinaria eficiencia** de los algoritmos de coste en $\mathcal{O}(\log n)$: pasar de un tamaño de $n = 10$ a $n = 1\,000\,000$ solo hace que el tiempo crezca de 1 milisegundo a 6. La búsqueda binaria en un vector ordenado, y la búsqueda en ciertas estructuras de datos de este curso, tienen este coste en el caso peor.
- ★ En sentido contrario, los algoritmos de coste $\mathcal{O}(2^n)$ son **prácticamente inútiles**: mientras que un problema de tamaño $n = 10$ se resuelve en aproximadamente un segundo, la edad del universo conocido ($1,4 \times 10^8$ siglos) sería totalmente insuficiente para resolver uno de tamaño $n = 100$. Algunos algoritmos de *vuelta atrás* que veremos en este curso tienen ese coste en el caso peor.
- ★ Esta tabla confirma la afirmación hecha al comienzo de este capítulo de que para ciertos algoritmos es inútil esperar a que los computadores sean más rápidos. Es más productivo invertir esfuerzo en diseñar **mejores algoritmos** para ese problema.

n	$\log_{10} n$	n	$n \log_{10} n$	n^2	n^3	2^n
10	1 <i>ms</i>	10 <i>ms</i>	10 <i>ms</i>	0.1 <i>s</i>	1 <i>s</i>	1.02 <i>s</i>
10^2	2 <i>ms</i>	0.1 <i>s</i>	0.2 <i>s</i>	10 <i>s</i>	16.67 <i>m</i>	$4.02 \cdot 10^{20}$ <i>sig</i>
10^3	3 <i>ms</i>	1 <i>s</i>	3 <i>s</i>	16.67 <i>m</i>	11.57 <i>d</i>	$3.4 \cdot 10^{291}$ <i>sig</i>
10^4	4 <i>ms</i>	10 <i>s</i>	40 <i>s</i>	1.16 <i>d</i>	31.71 <i>a</i>	$6.3 \cdot 10^{3000}$ <i>sig</i>
10^5	5 <i>ms</i>	1.67 <i>m</i>	8.33 <i>m</i>	115.74 <i>d</i>	317.1 <i>sig</i>	$3.16 \cdot 10^{30093}$ <i>sig</i>
10^6	6 <i>ms</i>	16.67 <i>m</i>	1.67 <i>h</i>	31.71 <i>a</i>	317 097.9 <i>sig</i>	$3.1 \cdot 10^{301020}$ <i>sig</i>

Figura 1: Crecimiento de distintas funciones de complejidad

- ★ Para mejorar la intuición anterior, hagamos el siguiente experimento: supongamos seis algoritmos con los costes anteriores, tales que tardan todos ellos 1 hora en resolver un problema de tamaño $n = 100$. ¿Qué ocurre si duplicamos la velocidad del computador? O lo que es lo mismo, ¿qué ocurre si duplicamos el tiempo disponible?

$t(n)$	$t = 1h.$	$t = 2h.$
$k_1 \cdot \log n$	$n = 100$	$n = 10\,000$
$k_2 \cdot n$	$n = 100$	$n = 200$
$k_3 \cdot n \log n$	$n = 100$	$n = 178$
$k_4 \cdot n^2$	$n = 100$	$n = 141$
$k_5 \cdot n^3$	$n = 100$	$n = 126$
$k_6 \cdot 2^n$	$n = 100$	$n = 101$

- ★ Observamos que mientras el de coste logarítmico es capaz de resolver problemas 100 veces más grandes, el de coste exponencial resuelve un tamaño prácticamente igual al anterior. Obsérvese que los de coste $\mathcal{O}(n)$ y $\mathcal{O}(n \log n)$ se comportan de acuerdo a la intuición de un usuario no informático: al duplicar la velocidad del computador (o el tiempo disponible), se duplica aproximadamente el tamaño del problema resuelto. En los de coste $\mathcal{O}(n^k)$, al duplicar la velocidad, el tamaño se multiplica por un factor $\sqrt[k]{2}$.
- ★ En la Figura 2 se muestra la **jerarquía de órdenes de complejidad**. Las inclusiones estrictas expresan que se trata de clases distintas. Los algoritmos cuyos costes están en la parte izquierda resuelven problemas que se denominan **tratables**. Estos costes se denominan en su conjunto **polinomiales**. Hay problemas que solo admiten algoritmos de complejidad exponencial o superior. Se llaman **intratables**.
- ★ También hay muchos problemas interesantes cuyos mejores algoritmos conocidos son exponenciales en el caso peor, pero no se sabe si existirán para ellos algoritmos polinomiales. Se llaman **NP-completos** y se verán en el próximo curso. El más conocido de todos es el problema **SAT** que consiste en determinar si una fórmula de la lógica proposicional es satisfactible.

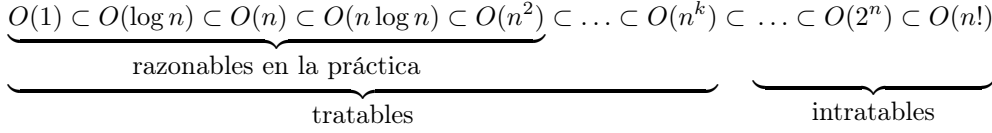


Figura 2: Jerarquía de órdenes de complejidad

4. Propiedades de los órdenes de complejidad

- ★ $O(a \cdot f(n)) = O(f(n))$ con $a \in \mathbb{R}^+$.

(\subseteq) $g \in O(a \cdot f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ tal que $\forall n \geq n_0, g(n) \leq c \cdot a \cdot f(n)$. Tomando $c' = c \cdot a$ se cumple que $\forall n \geq n_0, g(n) \leq c' \cdot f(n)$, luego $g \in O(f(n))$.

(\supseteq) $g \in O(f(n)) \Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ tal que $\forall n \geq n_0, g(n) \leq c \cdot f(n)$. Entonces tomando $c' = \frac{c}{a}$ se cumple que $\forall n \geq n_0, g(n) \leq c' \cdot a \cdot f(n)$, luego $g \in O(a \cdot f(n))$.

- ★ La base del logaritmo no importa: $O(\log_a n) = O(\log_b n)$, con $a, b > 1$. La demostración es inmediata sabiendo que:

$$\log_b n = \frac{\log_a n}{\log_a b}$$

- ★ Si $f \in O(g)$ y $g \in O(h)$, entonces $f \in O(h)$.

$$\begin{aligned} f \in O(g) &\Rightarrow \exists c_1 \in \mathbb{R}^+, n_1 \in \mathbb{N} \text{ tal que } \forall n \geq n_1, f(n) \leq c_1 \cdot g(n) \\ g \in O(h) &\Rightarrow \exists c_2 \in \mathbb{R}^+, n_2 \in \mathbb{N} \text{ tal que } \forall n \geq n_2, g(n) \leq c_2 \cdot h(n) \end{aligned}$$

Tomando $n_0 = \max(n_1, n_2)$ y $c = c_1 \cdot c_2$, se cumple

$$\forall n \geq n_0, f(n) \leq c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n)$$

Y por tanto $f \in O(h)$.

- ★ Regla de la suma: $O(f + g) = O(\max(f, g))$.

(\subseteq) $h \in O(f + g) \Rightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0, h(n) \leq c \cdot (f(n) + g(n))$. Pero $f \leq \max(f, g)$ y $g \leq \max(f, g)$, luego:

$$h(n) \leq c \cdot (\max(f(n), g(n)) + \max(f(n), g(n))) = 2 \cdot c \cdot \max(f(n), g(n))$$

Tomando $c' = 2 \cdot c$ se cumple que $\forall n \geq n_0, h(n) \leq c' \cdot \max(f(n), g(n))$ y por tanto $h \in O(\max(f, g))$.

(\supseteq) $h \in O(\max(f, g)) \Rightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0, h(n) \leq c \cdot \max(f(n), g(n))$. Pero $\max(f, g) \leq f + g$, luego $h \in O(f + g)$ trivialmente.

- ★ Regla del producto: Si $g_1 \in O(f_1)$ y $g_2 \in O(f_2)$, entonces $g_1 \cdot g_2 \in O(f_1 \cdot f_2)$. La demostración es similar.

★ Teorema del límite

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow f \in O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) = O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \text{ y } g \notin O(f) \Leftrightarrow O(f) \subset O(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow f \notin O(g) \text{ y } g \in O(f) \Leftrightarrow O(f) \supset O(g)$

★ Por el principio de dualidad, también tenemos:

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow g \in \Omega(f) \text{ y } f \in \Omega(g) \Leftrightarrow \Omega(f) = \Omega(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow g \in \Omega(f) \text{ y } f \notin \Omega(g) \Leftrightarrow \Omega(f) \supset \Omega(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow g \notin \Omega(f) \text{ y } f \in \Omega(g) \Leftrightarrow \Omega(f) \subset \Omega(g)$

★ Aplicando la definición de $\Theta(f)$, también tenemos:

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = m \in \mathbb{R}^+ \Rightarrow g \in \Theta(f) \text{ y } f \in \Theta(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g) \text{ pero } f \notin \Theta(g)$
- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow g \in O(f) \text{ pero } g \notin \Theta(f)$

Notas bibliográficas

Se recomienda ampliar el contenido de estas notas estudiando el Capítulo 1 de (Peña, 2005) en el cual se han basado. También la Sección 1.4 de (Rodríguez Artalejo et al., 2011).

El Capítulo 3 de (Martí Oliet et al., 2012) contiene material válido para este capítulo y material adicional que será utilizado en los Capítulos 3 y 4. También tiene numerosos ejemplos resueltos.

Ejercicios

1. Demostrar el Principio de dualidad, es decir $g(n) \in \mathcal{O}(f(n)) \Leftrightarrow f(n) \in \Omega(g(n))$.
2. Demostrar que todo polinomio $a_m n^m + \dots + a_1 n + a_0$, en n y de grado m , cuyo coeficiente a_m correspondiente al mayor grado sea positivo, está en $\mathcal{O}(n^m)$.

3. Demostrar

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow \mathcal{O}(f(n)) \subset \mathcal{O}(g(n))$$

Dar un ejemplo de que la implicación inversa puede no ser cierta.

4. Demostrar

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k > 0 \Rightarrow f(n) \in \Theta(g(n))$$

5. Usar el teorema del límite para demostrar las siguientes inclusiones estrictas (suponemos $k > 1$):

$$O(1) \subset O(\log n) \subset O(n) \subset O(n^k) \subset O(2^n) \subset O(n!)$$

6. Si tenemos dos algoritmos con costes $t_1(n) = 3n^3$ y $t_2(n) = 600n^2$, ¿cuál es mejor en términos asintóticos? ¿A partir de qué umbral el segundo es mejor que el primero?
7. Si el coste de un algoritmo está en $\mathcal{O}(n^2)$ y tarda 1 segundo para un tamaño $n = 100$, ¿de qué tamaño será el problema que puede resolver en 10 segundos?
8. Demostrar por inducción sobre $n \geq 0$ las siguientes igualdades:
 - a) $\sum_{i=1}^n i = n(n+1)/2$.
 - b) $\sum_{i=1}^n i^2 = n(n+1)(2n+1)/6$.
 - c) $\sum_{i=1}^n 2^i = (n-1)2^{n+1} + 2$.
9. Demostrar que $\sum_{i=1}^n i^k \in \Theta(n^{k+1})$.
10. Demostrar que $\log n \in O(\sqrt{n})$ pero que $\sqrt{n} \notin O(\log n)$.
11. ¿Verdadero o falso?
 - a) $2^n + n^{99} \in O(n^{99})$.
 - b) $2^n + n^{99} \in \Omega(n^{99})$.
 - c) $2^n + n^{99} \in \Theta(n^{99})$.
 - d) Si $f(n) = n^2$, entonces $f(n)^3 \in O(n^5)$.
 - e) Si $f(n) \in O(n^2)$ y $g(n) \in O(n)$, entonces $f(n)/g(n) \in O(n)$.
 - f) Si $f(n) = n^2$, entonces $3f(n) + 2n \in \Theta(f(n))$.
 - g) Si $f(n) = n^2$ y $g(n) = n^3$, entonces $f(n)g(n) \in O(n^6)$.
12. Comparar con respecto a O y Ω los siguientes pares de funciones:
 - a) 2^{n+1} , 2^n .
 - b) $(n+1)!$, $n!$.
 - c) $\log n$, \sqrt{n} .
 - d) Para cualquier $a \in \mathbb{R}^+$, $\log n$, n^a .
13. Supongamos que $t_1(n) \in \mathcal{O}(f(n))$ y $t_2(n) \in \mathcal{O}(f(n))$. Razonar la verdad o falsedad de las siguientes afirmaciones:
 - a) $t_1(n) + t_2(n) \in \mathcal{O}(f(n))$.
 - b) $t_1(n) \cdot t_2(n) \in \mathcal{O}(f(n^2))$.
 - c) $t_1(n)/t_2(n) \in \mathcal{O}(1)$.
14. (Martí Oliet et al. (2012)) Demostrar o refutar cada una de las siguientes afirmaciones:
 - a) $n^2 + n + \log n \in O(n^2)$.
 - b) $n^2 + n + \log n \in \Omega(n)$.
 - c) $n^2 + n + \log n \in \Theta(\log n)$.
 - d) $21n^3 + 7n^2 + n \log n - 17n + 8 \in O(n^4)$.
 - e) $2^n + 3^n + n^{59} \in O(n^{59})$.
 - f) $2^n + 3^n + n^{59} \in \Omega(2^n)$.

- g) Si $f(n) = n^2$, entonces $f(n)^3 \in \Theta(f(n)^3)$.
- h) $\Theta(2^n) = \Theta(2^{n+2}) = \Theta(4^n)$.
- i) $\log_2 n \in O(\log_3 n)$.
15. (Martí Oliet et al. (2012)) Para cada una de las siguientes funciones $f(n)$, obtener el menor número natural k tal que $f(n) \in O(n^k)$.
- a) $f(n) = 2n^3 + n^2 \log n$.
- b) $f(n) = 3n^5 + (\log n)^4$.
- c) $f(n) = (n^4 + n^2 + 1)/(n^3 + 1)$.
- d) $f(n) = (n^4 + n^2 + 1)/(n^4 + 1)$.
- e) $f(n) = (n^4 + 5 \log n)/(n^4 + 1)$.
- f) $f(n) = (n^3 + 5 \log n)/(n^4 + 1)$.
16. (Martí Oliet et al. (2012)) Para cada una de las siguientes funciones $f(n)$, encontrar una función $g(n)$ del menor orden posible tal que $f(n) \in O(g(n))$.
- a) $f(n) = (n \log n + n^2)(n^3 + 2)$.
- b) $f(n) = (2^n + n^2)(n^3 + 3^n)$.
- c) $f(n) = n \log(n^2 + 1) + n^2 \log n$.
- d) $f(n) = n^{2^n} + n^{n^2}$.
- e) $f(n) = (n! + 2^n)(n^3 + \log(n^2 + 1))$.
17. (Martí Oliet et al. (2012)) Comparar con respecto a O y Ω los siguientes pares de funciones:
- a) $n^2 + 3n + 7, n^2 + 10$.
- b) $n^2 \log n, n^3$.
- c) $n^4 + \log(3n^8 + 7), (n^2 + 17n + 3)^2$.
- d) $(n^3 + n^2 + n + 1)^4, (n^4 + n^3 + n^2 + n + 1)^3$.
- e) $\log(n^2 + 1), \log n$.
- f) $2^{n+3}, 2^{n+7}$.
- g) $2^{2^n}, 2^{n^2}$.
18. (Martí Oliet et al. (2012)) Indicar cuántas multiplicaciones realizan los siguientes algoritmos para calcular potencias en el caso peor. Indicar también sus ordenes asintóticos.

```

1 int potencial(int x, int y) {
2   int p = 1;
3   while (y > 0) {
4     p = p*x;
5     y--;
6   }
7   return p;
8 }
```

```

1 int potencia2(int x, int y){
2     int w = x; int p = 1;
3     while (y > 0){
4         if (y%2 == 1) p = p*w;
5         y = y/2;
6         w = w*w;
7     }
8     return p;
9 }

```

19. Sea la siguiente función que implementa el algoritmo de inserción de un elemento en una lista (tLista) ordenada de enteros con posibles repeticiones, representada como una estructura con dos campos, elems, un array de enteros y cont, un entero. Indicar el número de instrucciones que ejecuta esta función y su orden asintótico en los casos mejor y peor. Nota: Para simplificar, se considera una instrucción a una línea de programa distinta de llave o else.

```

1 bool insertar(tLista& lista, int x){
2     if (lista.cont == MAX){
3         return false;
4     } else{
5         int pos = lista.cont;
6         while (pos > 0 && x < lista.elems[pos-1]) {
7             lista.elems[pos] = lista.elems[pos-1];
8             pos--;
9         }
10        lista.elems[pos] = x;
11        lista.cont++;
12        return true;
13    }
14 }

```

20. Sea la siguiente función que implementa el algoritmo de inserción de un elemento en una lista ordenada de enteros sin repeticiones. Indicar el número de instrucciones que ejecuta (ver Ej. 19) en los casos mejor y peor. Indicar también su orden asintótico en los casos mejor y peor en los siguientes dos supuestos: a) buscar es una búsqueda lineal; b) buscar es una búsqueda binaria.

```

1 bool insertar(tLista& lista, int x) {
2     if (lista.cont == MAX)
3         return false;
4     else {
5         int pos = 0;
6         if (!buscar(lista, x, pos)) { // pos indica la posición en la
7             for (int i = lista.cont; i > pos; i--); // que debe colocarse x
8                 lista.elems[i] = lista.elems[i-1];
9                 lista.elems[pos] = x;
10                lista.cont++;
11                return true;
12            }
13        else return false;
14    }
15 }

```

21. Dada la siguiente función que implementa el algoritmo de ordenación por inserción directa. Indicar el número de instrucciones que ejecuta (ver Ej. 19) y su orden asintótico en los casos mejor y peor.

```
1 void ordenar(tLista& lista) {
2     int nuevo, pos;
3     for (int i = 1; i < lista.cont; i++) {
4         nuevo = lista.elems[i];
5         pos = 0;
6         while ((pos < i) && !(lista.elems[pos] > nuevo)) {
7             pos++;
8         }
9         // pos: índice del primer mayor; i si no lo hay
10        for (int j = i; j > pos; j--) {
11            lista.elems[j] = lista.elems[j - 1];
12        }
13        lista.elems[pos] = nuevo;
14    }
15 }
```

22. Dada la siguiente función que implementa el algoritmo de ordenación de la burbuja. Indicar el número de instrucciones que ejecuta (ver Ej. 19) y su orden asintótico en los casos mejor y peor.

```
1 void ordenar(tLista& lista) {
2     bool inter = true;
3     int i = 0; int tmp;
4     while ((i < lista.cont - 1) && inter) {
5         inter = false;
6         for (int j = lista.cont - 1; j > i; j--)
7             if (lista.elems[j] < lista.elems[j - 1]) {
8                 tmp = lista.elems[j];
9                 lista.elems[j] = lista.elems[j - 1];
10                lista.elems[j - 1] = tmp;
11                inter = true;
12            }
13        if (inter)
14            i++;
15    }
16 }
```

Especificación de algoritmos¹

*Las matemáticas son el alfabeto con el cual Dios
ha escrito el Universo.*

Galileo Galilei (1564-1642)

RESUMEN: En este tema se enseña a distinguir entre especificar e implementar algoritmos, se repasa la lógica de predicados, y se establecen convenios y notaciones para especificar funciones y procedimientos utilizando dicha lógica.

1. Introducción

- ★ **Especificar** un algoritmo consiste en contestar a la pregunta “**qué** hace el algoritmo”, sin dar detalles sobre cómo consigue dicho efecto. Una actitud correcta consiste en contemplar el algoritmo como una **caja negra** de la que solo podemos observar sus entradas y sus salidas.
- ★ **Implementar**, por el contrario, consiste en contestar a este segundo aspecto, es decir “**cómo** se consigue la funcionalidad pretendida”, suponiendo que dicha funcionalidad está perfectamente clara en la especificación.
- ★ Muchos programadores noveles tienen dificultades para distinguir entre ambas actividades, produciendo documentos donde se mezclan continuamente los dos conceptos.
- ★ La especificación de un algoritmo tiene un doble destinatario:
 - Los **usuarios** del algoritmo: debe recoger todo lo necesario para un uso correcto del mismo. En particular, ha de detallar las obligaciones del usuario al invocar dicho algoritmo y el resultado producido si es invocado correctamente.
 - El **implementador** del algoritmo: define los requisitos que cualquier implementación válida debe satisfacer, es decir, las obligaciones del implementador. Ha de dejar suficiente **libertad** para que este elija la implementación que estime más adecuada con los recursos disponibles.

¹Ricardo Peña es el autor principal de este tema.

- ★ Una vez establecida la especificación, los trabajos del usuario y del implementador pueden proseguir **por separado**. La especificación actúa pues como una **barrera o contrato** que permite independizar los razonamientos de corrección de los distintos componentes de un programa grande. Así, se descompone la tarea de razonar sobre el mismo en **unidades manejables** que corresponden a su estructura modular.

- ★ Propiedades de una buena especificación:

Precisión Ha de responder a cualquier pregunta sobre el uso del algoritmo sin tener que acudir a la implementación del mismo. El lenguaje natural no permite la precisión requerida si no es sacrificando la brevedad.

Brevedad Ha de ser significativamente más breve que el código que especifica. Los lenguajes formales ofrecen a la vez precisión y brevedad.

Claridad Ha de transmitir la intuición de lo que se pretende. A veces el lenguaje formal ha de ser complementado con explicaciones informales.

- ★ En este capítulo se presenta una técnica de especificación formal de funciones y procedimientos basada en el uso de la **lógica de predicados**, también conocida como técnica **pre/post**. Más adelante se presentará otra técnica formal diferente, las llamadas **especificaciones algebraicas**, más apropiadas para especificar tipos de datos.

- ★ Las ventajas de una **especificación formal** frente a una informal son bastantes:
 - **Eliminan ambigüedades** que el usuario y el implementador podrían resolver de formas distintas, dando lugar a errores de uso o de implementación que aparecerían en ejecución.
 - Son las únicas que permiten realizar una **verificación formal** del algoritmo. Este tema se tratará en los capítulos **3** y **4**, y consiste en esencia en razonar sobre la corrección del algoritmo mediante el uso de la **lógica**.
 - Permite generar **automáticamente** un conjunto de **casos de prueba** que permitirán probar la corrección de la implementación. La especificación formal se usa para **predecir** y **comprobar** el resultado esperado.

- ★ Supongamos declarado en alguna parte el siguiente tipo de datos:

typedef int vect [1000]; (2.1)

Queremos una función que, dado un vector a de tipo *vect* y un entero n , devuelva un valor booleano b que indique si el valor de alguno de los elementos $a[0], \dots, a[n-1]$ es igual a la suma de todos los elementos que le preceden en el vector. Utilizaremos la siguiente cabecera sintáctica:

fun *essuma* (*vect* a , **int** n) **return** (**bool** b)

- ★ Esta especificación informal presenta algunas ambigüedades:
 - No quedan claras todas las obligaciones del usuario: (1) ¿serían admisibles llamadas con valores negativos de n ?; (2) ¿y con $n = 0$?; (3) ¿y con $n > 1000$?
 - En caso afirmativo, ¿cuál ha de ser el valor devuelto por la función?

- Tampoco están claras las obligaciones del implementador. Por ejemplo, si $n \geq 1$ y $a[0] = 0$ la función, ¿ha de devolver **true** o **false**?

Tratar de explicar en lenguaje natural todas estas situaciones llevaría a una especificación más extensa que el propio código de la función *essuma*.

- ★ La siguiente especificación formal contesta a estas preguntas:

$$P \equiv \{0 \leq n \leq 1000\}$$

$$\text{fun } \textit{essuma} \text{ (vect } a, \text{ int } n) \text{ return bool } b$$

$$Q \equiv \{b = \exists w : 0 \leq w < n : a[w] = (\sum u : 0 \leq u < w : a[u])\}$$

No son correctas llamadas con valores negativos de n , ni con $n > 1000$; si son correctas llamadas con $n = 0$ y en ese caso ha de devolver **false**; las llamadas con $n \geq 1$ y $a[0] = 0$ han de devolver $b = \text{true}$. Para saber por qué, seguir leyendo.

- ★ La técnica pre/post, debida a C.A.R. Hoare (1969), se basa en considerar que un algoritmo es una función de estados en estados: comienza su ejecución en un **estado inicial** válido descrito por el valor de los parámetros de entrada y, tras un cierto tiempo cuya duración no es relevante a efectos de la especificación, termina en un **estado final** en el que los parámetros de salida contienen los resultados esperados.
- ★ Si S representa la función a especificar, P es un predicado que tiene como variables libres los **parámetros de entrada** de S , y Q es un predicado que tiene como variables libres los **parámetros de entrada y de salida** de S , la notación

$$\{P\}S\{Q\}$$

es la **especificación formal** de S y ha de leerse: “Si S comienza su ejecución en un estado descrito por P , S termina y lo hace en un estado descrito por Q ”.

- ★ P recibe el nombre de **precondición** y caracteriza el conjunto de estados iniciales para los que está garantizado que el algoritmo funciona correctamente. Describe las **obligaciones del usuario**. Si este llama al algoritmo en un estado no definido por P , no es posible afirmar qué sucederá.
- ★ Q recibe el nombre de **postcondición** y caracteriza la relación entre cada estado inicial, supuesto este válido, y el estado final correspondiente. Describe las **obligaciones del implementador**, supuesto que el usuario ha cumplido las suyas.

2. Predicados

- ★ Usaremos las letras mayúsculas P, Q, R, \dots , para denotar predicados de la lógica de primer orden. La sintaxis se define inductivamente mediante las siguientes reglas:
 1. Una expresión e de tipo **bool** es un predicado.
 2. Si P es un predicado, $\neg P$, leído “no P ”, también lo es.
 3. Si P y Q son predicados, $P \& Q$ también lo es, siendo $\&$ cualquiera de las conectivas lógicas $\{\wedge, \vee, \rightarrow, \leftrightarrow\}$, leídas respectivamente “y”, “o”, “entonces”, “si y solo si”.

4. Si P y Q son predicados, $(\forall w : Q(w) : P(w))$ y $(\exists w : Q(w) : P(w))$ también lo son, denominados respectivamente **cuantificación universal** y **cuantificación existencial**.

En lógica de primer orden, estas cuantificaciones se escriben con una sintaxis más simple (respectivamente $\forall w . R(w)$ y $\exists w . R(w)$, siendo R un predicado). La sintaxis edulcorada propuesta aquí se escribiría entonces $\forall w . (Q(w) \rightarrow P(w))$ y $\exists w . (Q(w) \wedge P(w))$ respectivamente. La hemos elegido porque facilitará la escritura de asertos sobre programas.

- ★ Algunos ejemplos de predicados:

$$\begin{aligned} n &\geq 0 \\ x &> 0 \wedge x \neq y \\ 0 &\leq i < n && \{\text{abreviatura de } 0 \leq i \wedge i < n\} \\ \forall w : 0 &\leq w < n : a[w] \geq 0 \\ \exists w : 0 &\leq w < n : a[w] = x \\ \forall w : 0 &\leq w < n : \text{impar}(w) \rightarrow (\exists u : u \geq 0 : a[w] = 2^u) \end{aligned}$$

donde $\text{impar}(x)$ es un predicado que determina si su argumento x es impar o no.

- ★ Es **muy importante** saber distinguir los dos tipos de variables que pueden aparecer en un predicado:

Variables ligadas. Son las que están afectadas por un cuantificador, es decir, se encuentran dentro de su ámbito. Por ejemplo, el ámbito del cuantificador en $(\forall w : Q(w) : P(w))$ son los predicados Q y P . Toda aparición de w en ellos que no esté ligada a otro cuantificador más interno, está ligada al $\forall w$ externo.

Variables libres. Son el resto de las variables que aparecen en el predicado. La intención es que estas variables se refieran a variables o parámetros del algoritmo que se está especificando.

- ★ En el predicado $\forall w : 0 \leq w < n : \text{impar}(w) \rightarrow (\exists u : u \geq 0 : a[w] = 2^u)$, las variables n y a son **libres**, mientras que w y u son **ligadas**.
- ★ Una variable ligada se puede **renombrar** de forma consistente sin cambiar el significado del predicado. Formalmente, si v no aparece en Q ni en P , entonces:

$$(\partial w : Q(w) : P(w)) \equiv (\partial v : Q(v) : P(v))$$

donde ∂ representa un cuantificador cualquiera (por ahora hemos presentado \forall y \exists , pero aparecerán más). Por ejemplo:

$$(\exists w : 0 \leq w < n : a[w] = x) \equiv (\exists v : 0 \leq v < n : a[v] = x)$$

- ★ Dado el uso tan diferente de las variables libres y ligadas y que el nombre de estas últimas siempre se puede cambiar, utilizaremos en este curso la siguiente regla:

*Las variables ligadas de un predicado **nunca tendrán el mismo nombre** que una variable del programa que está siendo especificado o verificado. Siempre que sea posible, usaremos las letras u, v, w, \dots para nombrar variables ligadas.*

- ★ Utilizaremos frecuentemente otras expresiones cuantificadas, **de tipo entero** en lugar de booleano, que se han demostrado útiles en la escritura de predicados breves y legibles. Son las siguientes (Q y P son predicados y e es una expresión entera):

$\sum w : Q(w) : e(w)$	suma de las $e(w)$ tales que $Q(w)$
$\prod w : Q(w) : e(w)$	producto de las $e(w)$ tales que $Q(w)$
$\text{máx } w : Q(w) : e(w)$	máximo de las $e(w)$ tales que $Q(w)$
$\text{mín } w : Q(w) : e(w)$	mínimo de las $e(w)$ tales que $Q(w)$
$\# w : Q(w) : P(w)$	número de veces que se cumple $Q(w) \wedge P(w)$

3. Significado

- ★ En el contexto de este curso, utilizaremos predicados para definir **conjuntos de estados**.
- ★ Un **estado** es simplemente una asociación de las variables del algoritmo con valores compatibles con su tipo. Por ejemplo, si x e y son variables de tipo entero, $\sigma = \{x \mapsto 3, y \mapsto 7\}$ representa un estado en el que la variable x vale 3 y la variable y vale 7. Un estado modeliza intuitivamente una “fotografía” de las variables de un algoritmo en un instante concreto.
- ★ Diremos que un estado σ **satisface** un predicado P si al sustituir en P las variables libres por los valores que esas variables tienen en σ , el predicado se evalúa a **true**. Por ejemplo, el estado σ anterior satisface $P \equiv y - x > 0$, pero no $Q \equiv x \bmod 2 = 0$.
- ★ Dado un predicado P , queda determinado con precisión el conjunto de todos los estados que satisfacen P . Adoptaremos entonces el punto de vista de **identificar** un predicado con el **conjunto de estados que lo satisfacen**. Este conjunto frecuentemente es infinito y a veces es vacío.
- ★ Suponiendo que x e y sean las únicas variables del algoritmo, $P \equiv y - x > 0$ define los infinitos pares (x, y) en los que y es mayor que x (es decir el semiplano encima de la diagonal principal del plano), mientras que $Q \equiv x \bmod 2 = 0$ define todos los pares (x, y) en los que x es un número par.
- ★ En particular, el predicado **true** define el conjunto de **todos** los estados posibles (i.e. cualquier variable del algoritmo puede tener cualquier valor de su tipo), y el predicado **false** define el conjunto **vacío**.
- ★ El significado del predicado $\forall w : Q(w) : P(w)$ es equivalente al de la **conjunción** $P(w_1) \wedge P(w_2) \wedge \dots$, donde w_1, w_2, \dots son todos los valores de w que hacen cierto $Q(w)$. Si este conjunto es **vacío**, entonces $\forall w : Q(w) : P(w) \equiv \text{true}$.
- ★ El significado del predicado $\exists w : Q(w) : P(w)$ es equivalente al de la **disyunción** $P(w_1) \vee P(w_2) \vee \dots$, donde w_1, w_2, \dots son todos los valores de w que hacen cierto $Q(w)$. Si este conjunto es **vacío**, entonces $\exists w : Q(w) : P(w) \equiv \text{false}$.
- ★ Hay predicados que se satisfacen en todos los estados (e.g. $x > 0 \vee x \leq 0$). Equivalen a **true**. Hay otros que no se satisfacen en ninguno (e.g. $\exists w : 0 < w < 1 : a[w] = 8$). Equivalen a **false**.

- ★ Por definición, el significado del resto de los cuantificadores cuando el rango $Q(w)$ al que se extiende la variable cuantificada es **vacío**, es el siguiente:

$$\begin{aligned}
 (\sum w : \mathbf{false} : e(w)) &= 0 \\
 (\prod w : \mathbf{false} : e(w)) &= 1 \\
 (\text{máx } w : \mathbf{false} : e(w)) &\quad \text{indefinido} \\
 (\text{mín } w : \mathbf{false} : e(w)) &\quad \text{indefinido} \\
 (\# w : \mathbf{false} : P(w)) &= 0
 \end{aligned}$$

- ★ Diremos que un predicado P es **más fuerte** (resp. **más débil**) que otro Q , y lo expresaremos $P \Rightarrow Q$ (resp. $P \Leftarrow Q$), cuando en términos de estados se cumpla $P \subseteq Q$ (resp. $P \supseteq Q$), es decir, todo estado que satisface P también satisface Q (resp. todo estado que satisface Q también satisface P).

$$\begin{aligned}
 x > 0 &\Rightarrow x \geq 0 \\
 P \wedge Q &\Rightarrow P \\
 P \wedge Q &\Rightarrow Q \\
 P &\Rightarrow P \vee Q \\
 \forall w : 0 \leq w < 10 : a[w] \neq 0 &\Rightarrow a[3] \neq 0
 \end{aligned}$$

- ★ El predicado más fuerte posible es **false** pues, para cualquier predicado P , $\mathbf{false} \Rightarrow P$. Simétricamente, el predicado más débil posible es **true**: para cualquier predicado P , $P \Rightarrow \mathbf{true}$.
- ★ La relación “ser más fuerte que” coincide con la noción lógica de **deducción**. Leeremos con frecuencia $P \Rightarrow Q$ como “de P se deduce Q ”, o “ P implica Q ”.
- ★ Si $P \Rightarrow Q$ y $Q \Rightarrow P$, diremos que son igual de fuertes, o igual de débiles, o simplemente que son **equivalentes**, y lo expresaremos como $P \equiv Q$. Dos predicados equivalentes definen el mismo conjunto de estados.

4. Ejemplos de especificación

- ★ Identificaremos un algoritmo con la noción de **función** en C++. En este lenguaje, las funciones tienen un tipo que corresponde al del resultado devuelto. Si no devuelven ningún valor, se usa el tipo **void**. Los mecanismos de paso de parámetros distinguen entre **paso por valor**, **paso por referencia** y **paso por referencia constante**. En esencia especifican si los parámetros formales de la función llamada son disjuntos o son los mismos que los parámetros reales del llamante. Se trata de un aspecto más relacionado con la eficiencia que con el flujo de información.

- ★ A efectos de especificación es más ilustrativo saber si los parámetros son de

entrada Su valor inicial es relevante para el algoritmo y este **no debe** modificarlo.

salida Su valor inicial es irrelevante para el algoritmo y este **debe** almacenar algún valor en él.

entrada/salida Su valor inicial es relevante para el algoritmo, y además este **puede** modificarlo.

- ★ Por ello, usaremos en la especificación dos tipos de **cabeceras virtuales** que el programador habrá de traducir después a la cabecera de función C++ que le parezca más apropiada.

$$\begin{array}{l} \mathbf{fun} \text{ nombre } (\mathbf{tipo}_1 p_1, \dots, \mathbf{tipo}_n p_n) \mathbf{return} \mathbf{tipo} r \\ \mathbf{proc} \text{ nombre } (\mathit{cualif} \mathbf{tipo}_1 p_1, \dots, \mathit{cualif} \mathbf{tipo}_n p_n) \end{array}$$

donde *cualif* será vacío si el parámetro es de entrada, **out** si es de salida, o **inout** si es de entrada/salida. Los parámetros de una cabecera **fun** se entienden siempre de entrada.

- ★ La primera se usará para algoritmos que devuelvan un solo valor, y la segunda para los que no devuelvan nada, devuelvan más de un valor, o/y modifiquen sus parámetros.
- ★ Especificar un algoritmo que calcule el cociente y el resto de la división de naturales. Primer intento:

$$\begin{array}{l} \{a \geq 0 \wedge b > 0\} \\ \mathbf{proc} \text{ divide } (\mathbf{int} a, \mathbf{int} b, \mathbf{out} \mathbf{int} q, \mathbf{out} \mathbf{int} r) \\ \{a = q \times b + r\} \end{array}$$

- ★ El especificador ha de imaginar que el implementador es un ser **malévolo** que trata de satisfacer la especificación del modo más simple posible, respetando la “letra” pero no el “espíritu” de la especificación. El siguiente programa sería correcto:

$$\{q = 0; r = a; \}$$

El problema es que la postcondición es demasiado **débil**.

- ★ Segundo intento:

$$\begin{array}{l} \{a \geq 0 \wedge b > 0\} \\ \mathbf{proc} \text{ divide } (\mathbf{int} a, \mathbf{int} b, \mathbf{out} \mathbf{int} q, \mathbf{out} \mathbf{int} r) \\ \{a = q \times b + r \wedge 0 \leq r < b\} \end{array}$$

Por conocimientos elementales de matemáticas sabemos que solo existen dos números naturales que satisfacen lo que exigimos a q y r .

- ★ El máximo de las primeras n posiciones de un vector:

$$\begin{array}{l} \{n > 0 \wedge longitud(a) \geq n\} \\ \mathbf{fun} \text{ maximo } (\mathbf{int} a[], \mathbf{int} n) \mathbf{return} \mathbf{int} m \\ \{(\forall w : 0 \leq w < n : m \geq a[w]) \wedge (\exists w : 0 \leq w < n : m = a[w])\} \end{array}$$

- La precondition $n > 0$ es necesaria para asegurar que el rango del existencial no es vacío. Una postcondición **false** solo la satisfacen los algoritmos que no terminan.
- La segunda conjunción de la precondition requiere que el vector parámetro real tenga una longitud de al menos n .
- Nótese que la segunda conjunción de la postcondición es necesaria. Si no, el implementador malévolo podría devolver un número muy grande pero no necesariamente en el vector. Una postcondición más sencilla sería.

$$\{m = \max w : 0 \leq w < n : a[w]\}$$

- ★ Devolver un número primo mayor o igual que un cierto valor:

$$\{n > 1\}$$

```

fun unPrimo (int n) return int p
  {p ≥ n ∧ (∀w : 1 < w < p : p mód w ≠ 0)}

```

- ¿Sería correcto devolver $p = 2$?
 - Nótese que la postcondición no determina un único p . El implementador tiene la libertad de devolver cualquier primo mayor o igual que n .
- ★ Devolver el **menor** número primo mayor o igual que un cierto valor:

$$\{n > 1\}$$

```

fun menorPrimo (int n) return int p
  {p ≥ n ∧ primo(p) ∧ (∀w : w ≥ n ∧ primo(w) : p ≤ w)}

```

donde $\text{primo}(x) \equiv (\forall w : 1 < w < x : x \text{ mód } w \neq 0)$

- Obsérvese que el uso del predicado auxiliar $\text{primo}(x)$ hace la especificación a la vez más modular y legible.
- Nótese que un predicado **no** es una implementación. Por tanto no le son aplicables criterios de eficiencia: $\text{primo}(x)$ es una propiedad y **no** sugiere que la comprobación haya de hacerse dividiendo por todos los números menores que x .
- La postcondición podría expresarse de un modo más conciso:

$$p = (\text{mín } w : w \geq n \wedge \text{primo}(w) : w)$$

sin que de nuevo esta especificación sugiera una forma de implementación.

- ★ Especificar un procedimiento que *positiviza* un vector. Ello consiste en reemplazar los valores negativos por ceros. Primer intento:

$$\{n \geq 0 \wedge \text{longitud}(a) = n\}$$

```

proc positivizar (inout int a[], int n)
  {∀w : 0 ≤ w < n : a[w] < 0 → a[w] = 0}

```

Lo que conseguimos es una postcondición equivalente a **false**.

- ★ Añadimos a la precondition la condición $a = A$ que nos sirve para poder dar un nombre al valor del vector a **antes** de ejecutar el procedimiento, ya que a en la postcondición se refiere a su valor **después** de ejecutarlo. Segundo intento:

$$\{n \geq 0 \wedge \text{longitud}(a) = n \wedge a = A\}$$

```

proc positivizar (inout int a[], int n)
  {∀w : 0 ≤ w < n : A[w] < 0 → a[w] = 0}

```

- ★ El implementador malévolo podría modificar también el resto de valores. Tercer intento:

$$\{n \geq 0 \wedge \text{longitud}(a) = n \wedge a = A\}$$

```

proc positivizar (inout int a[], int n)
  {∀w : 0 ≤ w < n : (A[w] < 0 → a[w] = 0) ∧ (A[w] ≥ 0 → a[w] = A[w])}


```


Notas bibliográficas

Se recomienda ampliar el contenido de estas notas estudiando el Capítulo 2 de (Peña, 2005) en el cual se han basado, si bien la notación para predicados es ligeramente diferente. También la Sección 1.1 de (Rodríguez Artalejo et al., 2011).

El Capítulo 1 de Martí Oliet et al. (2012) utiliza la misma notación de predicados empleada aquí y contiene numerosos ejemplos resueltos.



Ejercicios

Algunos de los ejercicios puedes probarlos en el portal y juez en línea “Acepta el reto” (<https://www.aceptaelreto.com>). Son los que aparecen marcados con el icono  seguido del número de problema dado en el portal.

1. Representar gráficamente la relación $P \Rightarrow Q$ para el siguiente conjunto de predicados:

- a) $P_1 \equiv x > 0$
- b) $P_2 \equiv (x > 0) \wedge (y > 0)$
- c) $P_3 \equiv (x > 0) \vee (y > 0)$
- d) $P_4 \equiv y \geq 0$
- e) $P_5 \equiv (x \geq 0) \wedge (y \geq 0)$

Indicar cuáles de dichos predicados son *incomparables* (P es incomparable con Q si $P \not\Rightarrow Q$ y $Q \not\Rightarrow P$).

2. (ACR247) Construir un predicado $ord(a, n)$ que exprese que el vector **int** $a[n]$ está ordenado crecientemente.
3. Generalizar el predicado anterior a otro $ord(a, i, j, n)$ que exprese que el subvector $a[i..j]$ de **int** $a[n]$ está ordenado crecientemente. ¿Tiene sentido $ord(a, 7, 6, 10)$?
4. Construir un predicado $perm(a, b, n)$, donde a y b son vectores de longitud n , que exprese que el vector b contiene una permutación de los elementos de a .
5. Especificar un procedimiento o función que ordene un vector de longitud n crecientemente.
6. Especificar un procedimiento o función que sustituya en un vector **int** $a[n]$ todas las apariciones del valor x por el valor y .
7. (ACR152) Dada una colección de valores, se denomina *moda* al valor que más veces aparece repetido en dicha colección. Queremos especificar una función que, dado un vector $a[n]$ de enteros con $n \geq 1$, devuelva la moda del vector.
8. Un vector **int** $a[n]$, con $n \geq 0$, se dice que es *gaspariforme* si todas sus sumas parciales son no negativas y la suma total es igual a cero. Se llama suma parcial a toda suma $a[0] + \dots + a[i]$, con $0 \leq i < n$. Especificar una función que, dados a y n , decida si a es o no gaspariforme. ¿Qué debe devolver la función cuando $n = 0$?
9. Especificar un algoritmo que calcule la imagen especular de un vector. Precisando, el valor que estaba en $a[0]$ pasa a estar en $a[n-1]$, el que estaba en $a[1]$ pasa a estar en $a[n-2]$, etc. Permitir el caso $n = 0$.

10. Dado un vector **int** $a[n]$, con $n \geq 0$, formalizar cada una de las siguientes afirmaciones:

- a) El vector a es estrictamente creciente.
- b) Todos los valores de a son distintos.
- c) Todos los valores de a son iguales.
- d) Si a contiene un 0, entonces a también contendrá un 1.
- e) No hay dos elementos contiguos de a que sean iguales.
- f) El máximo de a solo aparece una vez en a .
- g) El resultado l es la longitud máxima de un segmento constante en a .
- h) Todos los valores de a son números primos.
- i) El número de elementos pares de a es igual al número de elementos impares.
- j) El resultado p es el producto de todos los valores positivos de a .
- k) El vector a contiene un cuadrado perfecto.

11. Especificar una función que dado un natural n , devuelva la raíz cuadrada entera de n .

12. Especificar una función que dados $a > 0$ y $b > 1$, devuelva el logaritmo entero en base b de a .

13. Dado un vector **int** $a[n]$, con $n \geq 1$, expresar en lenguaje natural las siguientes postcondiciones:

- a) $b = (\exists w : 0 \leq w < n : a[w] = 2 \times w)$
- b) $m = (\# w : 1 \leq w < n : a[w-1] < a[w])$
- c) $m = (\text{máx } u, v : 0 \leq u < v < n : a[u] + a[v])$
- d) $l = (\text{máx } u, v : 0 \leq u \leq v < n \wedge (\forall w : u \leq w \leq v : a[w] = 0) : v - u + 1)$
- e) $r = (\text{máx } u, v : 0 \leq u \leq v < n : (\sum w : u \leq w \leq v : a[w]))$
- f) $p = (\prod u, v : 0 \leq u < v < n : a[v] - a[u])$

14. (Martí Oliet et al. (2012)) Dado x un vector de enteros, r y m dos enteros, y $n \geq 1$ un natural, formalizar cada una de las siguientes afirmaciones:

- $x[0..n)$ contiene el cuadrado de un número.
- r es el producto de todos los elementos positivos en $x[0..n)$.
- m es el resultado de sumar los valores en las posiciones pares de $x[0..n)$ y restar los valores en las posiciones impares.
- r es el número de veces que m aparece en $x[0..n)$.

15. (Martí Oliet et al. (2012)) Comparar la fuerza lógica de los siguiente pares de predi-
cados:

- a) $P = x \geq 0, \quad Q = x \geq 0 \rightarrow y \geq 0.$
- b) $P = x \geq 0 \vee y \geq 0, \quad Q = x + y = 0.$
- c) $P = x < 0, \quad Q = x^2 + y^2 = 9.$
- d) $P = x \geq 1 \rightarrow x \geq 0, \quad Q = x \geq 1.$

16. (Martí Oliet et al. (2012)) Especificar funciones que resuelvan los siguientes problemas:
- a) Decidir si un entero es el factorial de algún número natural.
 - b) Calcular el número de ceros que aparecen en un vector de enteros.
 - c) Calcular el producto escalar de dos vectores de reales.
 - d) Calcular la posición del máximo del vector no vacío de enteros $v[0..n)$.
 - e) Calcular la primera posición en la que aparece el máximo del vector no vacío $v[0..n)$.
17. (Martí Oliet et al. (2012)) Especificar una función que, dado un vector de números enteros, devuelva un valor booleano indicando si el valor de alguno de los elementos del vector es igual a la suma de todos los elementos que le preceden en el vector.
18. (Martí Oliet et al. (2012)) La fórmula de Taylor-Maclaurin proporciona la siguiente serie convergente para calcular el seno de un ángulo x expresado en radianes:

$$\text{seno}(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1}.$$

Por esta razón, para aproximar el seno de x con un error menor que un número real positivo ϵ basta encontrar un número natural k tal que $\left| \frac{(-1)^k}{(2k+1)!} x^{2k+1} \right| < \epsilon$ y entonces

$$\text{senoAprox}(x) = \sum_{n=0}^{k-1} \frac{(-1)^n}{(2n+1)!} x^{2n+1}.$$

Especificar formalmente una función que, dados los valores de x y $\epsilon > 0$, calcule el valor aproximado del seno de x con un error menor que ϵ .

19. Diremos que una posición de un vector de enteros es *fija* si el valor contenido en dicha posición coincide con la posición. Especificar una función *hayFija* que, dado un vector $v[0..n)$ de enteros estrictamente creciente, indique si hay alguna posición *fija* en el vector.
20. (🐘ACR171) Dado un vector de enteros v , el índice i es *mirador* si $v[i]$ es el mayor elemento de $v[i..n-1]$. Especifica una función que reciba un vector y calcule el número de miradores que tiene.
21. (🐘ACR221) Especifica una función que reciba un vector de enteros y devuelva un valor booleano indicando si se puede dividir en dos partes (alguna de ellas quizá vacía) donde la primera parte tenga sólo números pares y la segunda números impares.
22. (🐘ACR242) Especifica una función que reciba un vector de enteros y devuelva el resultado de sumar la multiplicación de todas las parejas de números $v[i]$, $v[j]$ (con i distinto de j).
23. (🐘ACR219) Especifica una función que reciba un vector y calcule cuántos números pares tiene.
24. (🐘ACR224) Especifica una función que reciba un vector y calcule, si existe, la posición del mismo que es igual a la suma de los elementos que tiene a la izquierda. Si hay más de uno, debe devolverse el situado más a la derecha.

Diseño de algoritmos iterativos¹

Cada cosa tiene su belleza, pero no todos pueden verla.

Confucio

RESUMEN: El tema comienza explicando el cálculo de la eficiencia de programas iterativos y a continuación se introducen los conceptos de verificación y derivación de algoritmos como técnicas que nos permiten razonar sobre la corrección de un algoritmo y su correcta construcción respectivamente. Por último se presentan diversos esquemas de diseño de programas iterativos que permiten obtener programas eficientes y se razona sobre su corrección.

1. Introducción

En este capítulo se estudia la implementación de algoritmos iterativos correctos y eficientes. En el capítulo anterior se ha visto la posibilidad de utilizar predicados para definir conjuntos de estados y cómo es posible especificar un algoritmo mediante dos predicados llamados precondición y postcondición. El primero describe el conjunto de estados válidos al comienzo del algoritmo y el segundo el conjunto de estados alcanzables con la ejecución del algoritmo.

Un algoritmo es correcto si cumple su especificación, es decir, si partiendo de un estado que cumple la precondición se llega a un estado que cumple la postcondición. Para comprobar si un algoritmo ya implementado es correcto se emplea la técnica de la *verificación*, consistente en representar los estados intermedios del algoritmo mediante predicados, generalmente denominados *aserciones* o *asertos*, y razonar con ellos formalmente hasta obtener la corrección.

Más interesante es razonar directamente sobre la especificación del programa de forma que se obtenga un algoritmo cuya corrección se garantiza por su propia construcción, evitando así el proceso de verificación posterior y los cambios en la implementación derivados de la detección de errores. La técnica que nos permite construir programas correctos se denomina *derivación*, y está basada en los mismos principios que la verificación.

En el apartado 2 se estudia cómo calcular la eficiencia de un programa iterativo usando los conceptos introducidos en el tema 1 de la asignatura. El apartado 3 está dedicado a la

¹Ramón González del Campo es el autor principal de este tema.

verificación de algoritmos y el apartado 4 a la derivación. Por último, en el apartado 5 se presentan varios algoritmos iterativos en cuya implementación se utiliza alguna técnica de resolución típica de muchos problemas y se razona sobre su corrección y su eficiencia.

2. Reglas prácticas para el cálculo de la eficiencia

1. Las instrucciones de asignación, de entrada-salida, los accesos a elementos de un vector y las expresiones aritméticas y lógicas (siempre que no involucren variables cuyos tamaños dependan del tamaño del problema) tendrán un coste constante, $\Theta(1)$. No se cuentan los *return*.
2. Para calcular el coste de una composición secuencial de instrucciones, $S_1; S_2$ se suman los costes de S_1 y S_2 . Si el coste de S_1 está en $\Theta(f_1(n))$ y el de S_2 está en $\Theta(f_2(n))$, entonces el coste de $S_1; S_2$ está en: $\Theta(f_1(n) + f_2(n)) = \Theta(\max(f_1(n), f_2(n)))$.
3. Para calcular el coste de una instrucción *condicional*:

if (B) { S_1 } **else** { S_2 }

Si el coste de S_1 está en $\mathcal{O}(f_1(n))$, el de S_2 está en $\mathcal{O}(f_2(n))$ y el de B en $\mathcal{O}(f_B(n))$, podemos señalar dos casos para el *condicional*:

- *Caso peor*: $\mathcal{O}(\max(f_B(n), f_1(n), f_2(n)))$.
- *Caso promedio*: $\mathcal{O}(\max(f_B(n), f(n)))$ donde $f(n)$ es el promedio de $f_1(n)$ y $f_2(n)$.

Se pueden encontrar expresiones análogas a éstas para la clase Omega.

4. Bucles con la forma:

while (B) { S }

Para calcular el coste de un bucle hay que calcular primero el coste de cada vuelta del bucle y después sumar los costes de todas las vueltas que se hagan en el bucle. El número de iteraciones dependerá de lo que tarde en hacerse falso B , teniendo en cuenta los datos concretos sobre los que se ejecute el programa y lo grandes que sean.

2.1. Ejemplos de cálculo de complejidad de algoritmos iterativos

De manera práctica, se analizará el código de los algoritmos presentados de arriba hacia abajo y de dentro hacia fuera.

- Búsqueda secuencial

```

1 bool buscaSec( int v[], int n, int x ) {
2   int j;
3   bool encontrado;
4
5   j = 0;
6   encontrado = false;
7   while ( (j < n) && ! encontrado ) {
8     encontrado = ( v[j] == x );
9     j++;

```

```

10 }
11 return encontrado;
12 }

```

- Caso peor: el elemento buscado x *condición* no está en el vector
 - Cuerpo del bucle *while*: 4 (3 en la línea 8 y 1 en la línea 9)
 - Bucle *while*: $\underbrace{n}_{j=0..n-1} * (\underbrace{4}_{\text{cuerpo}} + \underbrace{3}_{\text{condición}}) + \underbrace{3}_{\text{condición}} = 7n + 3$
 - Total: $7n + 3 + 2_{(\text{inicializaciones})} = 7n + 5$
- Caso mejor: el elemento buscado x está en la primera posición del vector; solo se entra una vez en el bucle *while*
 - Cuerpo del bucle *while*: 4
 - Bucle *while*: $(4 + 3) + 3 = 10$
 - Total: $10 + 2 = 12$

■ Búsqueda binaria

```

1 int buscaBin( int v[], int n, int x ) {
2   int izq, der, centro;
3
4   izq = -1;
5   der = n;
6   while ( der != izq+1 ) {
7     centro = (izq+der) / 2;
8     if ( v[centro] <= x )
9       izq = centro;
10    else
11      der = centro;
12  }
13  return izq;
14 }

```

- En esta ocasión no hay caso mejor ni peor, el bucle se ejecuta el mismo número de veces independientemente de la posición de x en el vector o incluso si x no aparece en el vector. Aunque el elemento buscado esté en el centro, el bucle sigue ejecutándose hasta acotar un subvector de longitud 1.
 - Cuerpo del bucle *while*: $\underbrace{3}_{\text{línea 7}} + \underbrace{3}_{\text{if}} = 6$
 - Bucle *while*: $\underbrace{\log(n)}_{\text{vueltas while}} * (\underbrace{6}_{\text{cuerpo}} + \underbrace{2}_{\text{condición}}) + \underbrace{2}_{\text{condición}} = 8 \log(n) + 2$
 - Total: $(8 \log(n) + 2) + 2_{(\text{inicializaciones})} = 8 \log(n) + 4$

En caso de que el elemento buscado estuviese repetido, ¿qué posición se devolvería?

■ Ordenación por inserción

```

1 void ordenaIns ( int v[], int n ) {
2   int i, j, x;
3
4   for ( i = 1; i < n; i++ ) {
5     x = v[i];

```

```

6     j = i-1;
7     while ((j >= 0) && (v[j] > x)) {
8         v[j+1] = v[j];
9         j = j-1;
10    }
11    v[j+1] = x;
12 }
13 }

```

- Caso peor: el vector está ordenado a la inversa; la segunda parte de la condición del *while* se cumple siempre

- Cuerpo del bucle *while*: 6 (4 de la línea 8 y 2 de la línea 9)

- Bucle *while*: $\underbrace{i}_{j=0..i-1} * (\underbrace{6}_{\text{cuerpo}} + \underbrace{4}_{\text{condición}}) + \underbrace{4}_{\text{condición}} = 10i + 4$

- Cuerpo del bucle *for*: $\underbrace{(10i + 4)}_{\text{while}} + \underbrace{2}_{l5} + \underbrace{2}_{l6} + \underbrace{3}_{l11} + \underbrace{1}_{i++} = 10i + 12$

- Bucle *for*:

$$\begin{aligned}
 & \sum_{i=1}^{n-1} (\underbrace{(10i + 12)}_{\text{cuerpo}} + \underbrace{1}_{\text{condición}}) + \underbrace{1}_{\text{condición}} + \underbrace{1}_{\text{ini}} = \sum_{i=1}^{n-1} (10i + 13) + 2 = \\
 & = 10 \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 13 + 2 = 10 \frac{n(n-1)}{2} + 13(n-1) + 2 = \\
 & = 5n^2 - 5n + 13n - 13 + 2 = 5n^2 + 8n - 11
 \end{aligned}$$

- Caso mejor: el vector está ordenado; la segunda parte de la condición del *while* no se cumple nunca

- Bucle *while*: $4_{(\text{condición})}$

- Cuerpo del bucle *for*: $4 + 2 + 2 + 3 + 1 = 12$

- Bucle *for*:

$$\sum_{i=1}^{n-1} (12 + 1) + 1 + 1 = \sum_{i=1}^{n-1} 13 + 2 = 13(n-1) + 2 = 13n - 11$$

■ Ordenación por selección

```

1 void ordenaSel ( int v[], int n ) {
2     int i, j, menor, aux;
3
4     for ( i = 0; i < n; i++ ) {
5         menor = i;
6         for ( j = i+1; j < n; j++ )
7             if ( v[j] < v[menor] )
8                 menor = j;
9         if ( i != menor ) {
10            aux = v[i];
11            v[i] = v[menor];
12            v[menor] = aux;
13        }
14    }
15 }

```

- Caso peor: vector desordenado

- Cuerpo del *for* interno: $\underbrace{3}_{if} + \underbrace{1}_{l8} + \underbrace{1}_{j++} = 5$

- Bucle *for* interno: $\underbrace{(n-i-1)}_{j=i+1..n-1} \left(\underbrace{5}_{cuerpo} + \underbrace{1}_{condición} \right) + \underbrace{1}_{condición} + \underbrace{2}_{ini} = 6n - 6i - 3$

- Cuerpo del *for* externo: $\underbrace{1}_{l5} + \underbrace{(6n - 6i - 3)}_{for} + \underbrace{1}_{if} + \underbrace{2}_{l10} + \underbrace{3}_{l11} + \underbrace{2}_{l12} + \underbrace{1}_{i++}$
 $= 6n - 6i + 7$

- Bucle *for* externo:

$$\sum_{i=0}^{n-1} \left(\underbrace{(6n - 6i + 7)}_{cuerpo} + \underbrace{1}_{condición} \right) + \underbrace{1}_{condición} + \underbrace{1}_{ini} = 6 \sum_{i=0}^{n-1} n - 6 \sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} 8 + 2 =$$

$$= 6n^2 - 6 \frac{n(n-1)}{2} + 8n + 2 = 6n^2 - \frac{6}{2}n^2 + \frac{6}{2}n + 8n + 2 = 3n^2 + 11n + 2$$

- Caso mejor: vector ordenado; la condición del *if* más interno no se cumple nunca, y como consecuencia la del más externo tampoco:

- Cuerpo del *for* interno: 4

- Bucle *for* interno: $(n-i-1)(4+1) + 1 + 2 = 5n - 5i - 2$

- Cuerpo del *for* externo: $1 + (5n - 5i - 2) + 1 + 1 = 5n - 5i + 1$

- Bucle *for* externo:

$$\sum_{i=0}^{n-1} ((5n - 5i + 1) + 1) + 1 + 1 = 5 \sum_{i=0}^{n-1} n - 5 \sum_{i=0}^{n-1} i + \sum_{i=0}^{n-1} 2 + 2 =$$

$$= 5n^2 - 5 \frac{n(n-1)}{2} + 2n + 2 = 5n^2 - \frac{5}{2}n^2 + \frac{5}{2}n + 2n + 2 = \frac{5}{2}n^2 + \frac{9}{2}n + 2$$

¿Qué sucede si el vector está ordenado al revés?

- Ordenación por el método de la burbuja modificado

```

1 void ordenaBur ( int v[], int n ) {
2     int i, j, aux;
3     bool modificado;
4
5     i = 0;
6     modificado = true;
7     while ( (i < num-1) && modificado ) {
8         modificado = false;
9         for ( j = n-1; j > i; j-- )
10             if ( v[j] < v[j-1] ) {
11                 aux = v[j];
12                 v[j] = v[j-1];
13                 v[j-1] = aux;
14                 modificado = true;
15             }
16         i++;
17     }
18 }
```

- Caso peor: El vector está ordenado al revés; se entra siempre en el *if*:
 - Cuerpo del bucle *for*: $\underbrace{4}_{if} + \underbrace{2}_{l11} + \underbrace{4}_{l12} + \underbrace{3}_{l13} + \underbrace{1}_{l14} + \underbrace{1}_{j--} = 15$
 - Bucle *for*: $\underbrace{(n-i-1)}_{j=i+1..n-1} (\underbrace{15}_{cuerpo} + \underbrace{1}_{condición}) + \underbrace{1}_{condición} + \underbrace{2}_{ini} = 16n - 16i - 13$
 - Cuerpo del bucle *while*: $\underbrace{1}_{l8} + \underbrace{16n - 16i - 13}_{for} + \underbrace{1}_{i++} = 16n - 16i - 11$
 - Bucle *while*:

$$\sum_{i=0}^{n-2} (\underbrace{(16n - 16i - 11)}_{cuerpo} + \underbrace{3}_{condición}) + \underbrace{3}_{condición} = 16 \sum_{i=0}^{n-2} n - 16 \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 8 + 3 =$$

$$= 16n(n-1) - 16 \frac{(n-2)(n-1)}{2} - 8(n-1) + 3 =$$

$$= 16n^2 - 16n - 8n^2 + 24n - 16 - 8n + 8 + 3 = 8n^2 - 5$$
 - Total: $\underbrace{8n^2 - 5}_{while} + \underbrace{2}_{ini} = 8n^2 - 3$
- Caso mejor: El vector está ordenado; nunca se entra en el *if* y solo se da una vuelta al *while*:
 - Cuerpo del bucle *for*: $4 + 1 = 5$
 - Bucle *for*: $(n-1)(5+1) + 1 + 2 = 6n - 3$
 - Cuerpo del bucle *while*: $1 + (6n - 3) + 1 = 6n - 1$
 - Bucle *while*:

$$((6n - 1) + 3) + 3 = 6n + 5$$
 - Total: $6n + 5 + 2 = 6n + 7$

3. Verificación

3.1. Semántica de un lenguaje imperativo

- Especificar un algoritmo: encontrar dos predicados P y Q tales que: $\{P\}A\{Q\}$.
- Verificar: introducir predicados intermedios entre cada par de instrucciones elementales:

$$\{P \equiv R_0\}A_1\{R_1\}; \dots; \{R_{n-1}\}A_n\{R_n \equiv Q\}$$

- Si se satisface $\{R_{k-1}\}A_k\{R_k\} \forall k$ entonces se satisface $\{P\}A\{Q\}$.
- Buscaremos reglas para determinar si se satisface $\{R_{k-1}\}A_k\{R_k\}$ (reglas de verificación).
- Conocer dichas reglas para cada instrucción del lenguaje es equivalente a conocer la *semántica* de éste.

Todas las instrucciones que se definen más adelante cumplen las siguientes reglas generales:

- Si se cumple $\{P\}A\{Q\}$ podemos conocer otras especificaciones de A . Si $P' \Rightarrow P$ entonces se cumple $\{P'\}A\{Q\}$:

$$\frac{P' \Rightarrow P \quad \{P\}A\{Q\}}{\{P'\}A\{Q\}} \quad (\text{fortalecimiento de la precondition})$$

- Si $Q \Rightarrow Q'$ entonces se cumple $\{P\}A\{Q'\}$:

$$\frac{\{P\}A\{Q\} \quad Q \Rightarrow Q'}{\{P\}A\{Q'\}} \quad (\text{debilitamiento de la postcondición})$$

- Conjunción en la postcondición:

$$\frac{\{P\}A\{Q_1\} \quad \{P\}A\{Q_2\}}{\{P\}A\{Q_1 \wedge Q_2\}}$$

- Disyunción en la precondition:

$$\frac{\{P_1\}A\{Q\} \quad \{P_2\}A\{Q\}}{\{P_1 \vee P_2\}A\{Q\}}$$

Acabamos de ver que la precondition de una especificación correcta se puede fortalecer. Esto implica que para demostrar la corrección de $\{P\}A\{Q\}$ es suficiente con demostrar $\{R\}A\{Q\}$ (donde R es la condición más débil de A con respecto a Q , $pmd(A, Q)$, que verifica la postcondición) y ver que $P \Rightarrow R$:

$$\frac{P \Rightarrow pmd(A, Q)}{\{P\}A\{Q\}}$$

3.2. Reglas específicas para el cálculo de la precondition más débil

Denotemos por Q_x^e la sustitución en Q de las apariciones libres de la variable x por la expresión e . Entonces:

- Instrucción *nada*:

$$pmd(nada, Q) = Q$$

- Asignación:

$$pmd(x = e, Q) = dom(e) \wedge Q_x^e$$

donde $dom(e)$ denota el conjunto de estados en los que la expresión e está definida.

- Composición secuencial:

Consideremos la siguiente composición de instrucciones $A_1; A_2$.

$$pmd(A_1; A_2, Q) = pmd(A_1, pmd(A_2, Q))$$

- Sentencias condicionales:

$$pmd(\text{if } (B) \{A_1\} \text{ else } \{A_2\}, Q) = (B \wedge pmd(A_1, Q)) \vee (\neg B \wedge pmd(A_2, Q))$$

3.3. Ejemplos

- Suponiendo x entero determina el predicado más débil que satisfaga la especificación: $\{P\}x = x + 2\{x \geq 0\}$:

$$pmd(x = x + 2, x \geq 0) \Leftrightarrow (x \geq 0)_x^{x+2}$$

$$\Leftrightarrow (x + 2) \geq 0$$

$$\Leftrightarrow x \geq -2$$

- Suponiendo x, y enteros, calcula en cada caso el predicado más débil que satisfaga la especificación: $\{P\}x = x * x; x = x + 1\{x > 0\}$

$$pmd(x = x * x; x = x + 1, x > 0)$$

$$\Leftrightarrow pmd(x = x * x; pmd(x = x + 1, x > 0))$$

$$\Leftrightarrow pmd(x = x * x; (x > 0)_x^{x+1})$$

$$\Leftrightarrow ((x > 0)_x^{x+1})_x^{x*x}$$

$$\Leftrightarrow (x + 1 > 0)_x^{x*x}$$

$$\Leftrightarrow (x * x) + 1 > 0$$

$$\Leftrightarrow \text{cierto}$$

- Suponiendo x, y enteros, determina el predicado más débil que satisfaga la especificación dada:

$$\{P\} \text{ if } (x \geq 0) \{y = x\} \text{ else } \{y = -x\} \{y = 4\}$$

$$pmd(\text{if } (x \geq 0)\{y = x\} \text{ else } \{y = -x\}, y = 4)$$

$$\Leftrightarrow (x \geq 0 \wedge pmd(y = x, y = 4)) \vee (x < 0 \wedge pmd(y = -x, y = 4))$$

$$\Leftrightarrow (x \geq 0 \wedge x = 4) \vee (x < 0 \wedge -x = 4)$$

$$\Leftrightarrow x = 4 \vee x = -4$$

3.4. Bucles e invariantes

- Consideremos la siguiente instrucción:

$$\begin{array}{l} \{P\} \\ \text{while } (B) \{ A \} \\ \{Q\} \end{array}$$

donde A es una instrucción y B es una expresión booleana.

- Si B es falso el cuerpo del bucle no se ejecuta ninguna vez \rightarrow Es como si no estuviera
 \Leftrightarrow Es equivalente a la instrucción *nada*.
- Para la verificación del bucle necesitamos un predicado llamado *invariante*, I , que describe los distintos estados por los que pasa el bucle:
 - I se satisface antes de ejecutarse el bucle: $P \Rightarrow I$.
 - I se satisface en cada iteración del bucle: $\{I \wedge B\}A\{I\}$.
 - I se satisface cuando finaliza la ejecución del bucle: $I \wedge \neg B \Rightarrow Q$.
- El invariante representa la relación entre las variables del bucle.

- En cada iteración, el cuerpo del bucle modifica los valores de las variables pero **no** su relación \Leftrightarrow Existen conjuntos de valores para las variables incompatibles entre sí.
- Es preciso que el programador proponga un predicado invariante que le permita probar la corrección. Normalmente dicho predicado se obtiene generalizando la post-condición del algoritmo.
- PROBLEMA:
 - Consideremos una especificación de un algoritmo y un algoritmo que satisface esta especificación.
 - Sustituyamos el cuerpo del bucle por la instrucción *nada*.
 - ¿Qué ocurre?
 - El algoritmo no funciona, se mete en un bucle infinito.
 - ¡NUESTRO INVARIANTE SIGUE VALIENDO! \Leftrightarrow Verifica las condiciones anteriores.
 - Nuestra verificación es defectuosa \rightarrow Es preciso exigir algo que cumpla nuestra primera implementación y **no** cumpla la segunda.
- Introducimos una función $cota : estado \rightarrow \mathbb{Z}$ que depende de las variables del cuerpo del bucle.
 - Es mayor o igual que cero cuando B se cumple: $I \wedge B \Rightarrow cota \geq 0$.
 - Decrece a ejecutarse el bucle: $\{I \wedge B \wedge cota = T\} A \{cota < T\}$, donde T es una constante.

3.5. Ejemplos

Verifica el siguiente algoritmo:

```

1  int suma(int V[], int N) {
2  int n, x;
3  n = N;
4  x = 0;
5  while (n != 0)
6  {
7      x = x+V[n-1];
8      n = n-1;
9  }
10 return x;
11 }
```

donde:

$$\{N \geq 0\}$$

fun suma(**int** V[N]) **return** **int** x

$$\{x = (\sum i : 0 \leq i < N : V[i])\}$$

utilizando como invariante: $I \equiv 0 \leq n \leq N \wedge x = (\sum i : n \leq i < N : V[i])$.

- $0 \leq n \leq N$: representa el rango de valores válidos de n .
- $x = (\sum i : n \leq i < N : V[i])$ es decir, x contiene la suma parcial de los elementos procesados.

- Relación entre x y n : x representa la suma parcial de los valores de V comprendidos entre n y $N - 1$.

Solución:

1. Es preciso demostrar que el invariante se cumple justo antes de comenzar el bucle \rightarrow
Es preciso demostrar:

$$\boxed{\{P\}n = N; x = 0\{I\}}$$

Utilizamos la regla de la asignación y la composición secuencial: $pm d(n = N; x = 0, I) \Leftrightarrow$

$$\begin{aligned} &\Leftrightarrow (I_x^0)_n^N \\ &\Leftrightarrow 0 \leq N \leq N \wedge \underbrace{(0 = \sum i : N \leq i < N : V[i])}_{\text{Certo}} \\ &\quad \text{(ya que el rango del sumatorio es vacío, por lo que es igual a 0)} \\ &\Leftrightarrow P \end{aligned}$$

2. El invariante se mantiene en cada iteración de bucle:

$$\boxed{\{I \wedge B\}A\{I\} \Leftarrow \{I \wedge n \neq 0\}x = x + V[n - 1]; n = n - 1\{I\}}$$

$$pm d(x = x + V[n - 1]; n = n - 1, I) \Leftrightarrow$$

$$\begin{aligned} &\Leftrightarrow (I_n^{n-1})_x^{x+V[n-1]} \\ &\Leftrightarrow 0 \leq n - 1 \leq N \wedge x + V[n - 1] = (\sum i : n - 1 \leq i < N : V[i]) \\ &\Leftrightarrow 0 \leq n - 1 \leq N \wedge x + V[n - 1] = V[n - 1] + (\sum i : n \leq i < N : V[i]) \\ &\Leftrightarrow I \wedge n \neq 0 \end{aligned}$$

3. Al salir del bucle se cumple la postcondición:

$$\boxed{\{I \wedge \neg B\} \Rightarrow \{Q\}}$$

$$\begin{aligned} I \wedge \neg(n \neq 0) &\Leftrightarrow (0 \leq n \leq N \wedge x = (\sum i : n \leq i < N : V[i])) \wedge n = 0 \\ &\Rightarrow x = (\sum i : 0 \leq i < N : V[i]) \equiv Q \end{aligned}$$

4. Consideramos la siguiente función cota: $cota(x, n) = n$. Es trivial ver que es positiva:

$$I \wedge n \neq 0 \Rightarrow n \geq 0$$

5. Cuando se ejecuta el bucle y decrece con cada iteración:

$$\boxed{\{I \wedge n \neq 0 \wedge n = T\}x = x + V[n - 1]; n = n - 1\{n < T\}}$$

$$pm d(x = x + V[n - 1]; n = n - 1, n < T) \Leftrightarrow$$

$$\begin{aligned} &\Leftrightarrow ((n < T)_n^{n-1})_x^{x+V[n-1]} \\ &\Leftrightarrow n - 1 < T \\ &\Leftrightarrow I \wedge n \neq 0 \wedge n = T \end{aligned}$$

Veamos otro ejemplo. Demostrar la corrección de la siguiente especificación suponiendo x, y y n : enteros.

$\{n \geq 0\}$

```

1  int x, y;
2  x = 0;
3  y = 1;
4  while (x != n)
5  {
6      x = x+1;
7      y = y+y;
8  }
```

$\{y = 2^n\}$

Solución:

- x representa el número de iteraciones del bucle.
- En cada iteración y se duplica.
- Por lo tanto, consideramos el siguiente invariante: $I \equiv 0 \leq x \leq n \wedge y = 2^x$

1. El invariante se cumple antes de comenzar el bucle:

$$\boxed{\{n \geq 0\}x = 0; y = 1\{I\}}$$

$$pmd(x = 0; y = 1, I) \Leftrightarrow$$

$$\Leftrightarrow 0 \leq 0 \leq n \wedge 1 = 2^0$$

$$\Leftarrow n \geq 0$$

2. El invariante se cumple dentro del bucle:

$$\boxed{\{I \wedge x \neq n\}x = x + 1; y = y + y\{I\}}$$

$$pmd(x = x + 1; y = y + y, I) \Leftrightarrow$$

$$\Leftrightarrow 0 \leq x + 1 \leq n \wedge y + y = 2^{x+1}$$

$$\Leftrightarrow 0 \leq x + 1 \leq n \wedge 2y = 2 \cdot 2^x$$

$$\Leftarrow I \wedge x \neq n$$

3. Al salir del bucle se cumple la postcondición:

$$I \wedge x = n \Rightarrow y = 2^n$$

4. Como función cota tomamos $n - x$ (número de iteraciones que quedan). Mientras se ejecuta el bucle es positiva:

$$\boxed{I \wedge x \neq n \Rightarrow n - x \geq 0}$$

5. $n - x$ decrece en cada iteración:

$$\boxed{\{I \wedge x \neq n \wedge n - x = T\}x = x + 1; y = y + y\{n - x < T\}}$$

$$pmd(x = x + 1; y = y + y, n - x < T) \Leftrightarrow$$

$$\Leftrightarrow n - (x + 1) < T$$

$$\Leftrightarrow (n - x) - 1 < T$$

$$\Leftarrow n - x = T$$

4. Derivación

- **Derivar:** construir las instrucciones a partir de la especificación asegurando su corrección.
- La postcondición dirige el proceso de verificación.
- Las igualdades de la postcondición se intentan satisfacer mediante las asignaciones correspondientes:
 - Si la precondition es más fuerte que el predicado más débil de estas asignaciones y la postcondición \rightarrow Proceso finalizado.
 - En caso contrario utilizaremos instrucciones iterativas. Intentaremos ceñirnos al siguiente esquema:

```

{P}
A0 (Inicialización)
{I, Cota}
while (B) {
    {I ∧ B}
    A1 (Restablecer)
    {R}
    A2 (Avanzar)
    {I}
}
{Q}

```

donde:

- A_0 es la instrucción que hace que el invariante se cumpla inicialmente.
- A_1 mantiene el invariante a cierto.
- A_2 hace que la cota decrezca.

Pasos para construir un algoritmo con bucle:

1. Diseñar el invariante y la condición del bucle sabiendo que se tiene que cumplir:
 $I \wedge \neg B \Rightarrow Q$
2. Diseñar A_0 para hacer el invariante cierto: $\{P\}A_0\{I\}$
3. Diseñar la función cota, C , de tal forma que: $I \wedge B \Rightarrow C \geq 0$.
4. Diseñar A_2 y el predicado $R \equiv pmd(A_2, I)$.
5. Diseñar A_1 para que se cumpla: $\{I \wedge B\}A_1\{R\}$.
6. Comprobar que la cota realmente decrece:

$$\{I \wedge B \wedge C = T\}A_1; A_2\{C < T\}$$

4.1. Ejemplos

Veamos como derivar un algoritmo para calcular la suma de los componentes de un vector. En el apartado de verificación vimos un algoritmo que resolvía el problema y demostramos que era correcto. En este caso se trata de obtener un algoritmo de forma que se garantice que es correcto sin necesidad de realizar una verificación posterior.

La especificación del problema es:

```
{N ≥ 0}
fun suma(int V[N]) return int x
{x = (Σi : 0 ≤ i < N : V[i])}
```

Para construir el algoritmo seguimos los pasos anteriores:

1. Obtenemos un invariante del bucle debilitando la postcondición. Elegimos el invariante que utilizamos para la verificación del algoritmo en la sección 3. $I \equiv 0 \leq n \leq N \wedge x = (\Sigma i : n \leq i < N : V[i])$.

Para que se cumpla la tercera condición de la verificación $I \wedge \neg B \Rightarrow Q$, la condición de parada debe ser: $n == 0$ y por lo tanto la condición del bucle $n \neq 0$ o $n > 0$ ya que el invariante asegura $n \geq 0$.

2. Instrucciones de inicialización. Debemos asignar valores a las variables libres del invariante n y x . Para que $\{P\}A_0\{I\}$ sea correcto, hacemos la asignación $n = N$ lo que hace el rango del sumatorio que define el valor de x en el invariante vacío y por lo tanto la asignación correcta a x es $x = 0$.
3. La función cota será la propia variable n : $t(n) = n$. Esta función decrece en cada vuelta del bucle.
4. La función de avance debe hacer que la cota disminuya. Por lo tanto podemos elegir $n = n - 1$. El predicado $R \equiv pmd(A_2, I)$, es $0 \leq n - 1 \leq N \wedge x = (\Sigma i : n - 1 \leq i < N : V[i])$.
5. Para restaurar el invariante y que $\{I \wedge B\}A_1\{R\}$ sea correcto, vemos que $\Sigma i : n - 1 \leq i < N : V[i] \equiv (\Sigma i : n \leq i < N : V[i] + V[n - 1])$. Por lo tanto la instrucción que necesitamos debe incrementar x en el valor de $V[n - 1]$, es decir $x = x + V[n - 1]$.
6. Por último comprobamos que con las instrucciones seleccionadas la cota disminuye al ejecutar el bucle, lo cual es cierto por la instrucción $n = n - 1$.

El algoritmo obtenido con el proceso de derivación coincide, como se espera, con el implementado en la sección 3.

```
1  int suma(int V[], int N) {
2  int n, x;
3  n = N;
4  x = 0;
5  while (n != 0)
6  {
7      x = x+V[n-1];
8      n = n-1;
9  }
10 return x;
11 }
```

Observemos que si elegimos un invariante distinto para guiar el proceso de derivación obtenemos un algoritmo distinto, aunque igualmente correcto.

1. Sea el invariante $I \equiv 0 \leq n \leq N \wedge x = (\sum i : 0 \leq i < n : V[i])$. Para que se cumpla la tercera condición de verificación la condición de parada debe ser: $n == N$ y por lo tanto la condición del bucle: $n \neq N$ o bien $n < N$.
2. Instrucciones de inicialización. Para que se cumpla el invariante seleccionamos un valor para las variables que hagan el rango del sumatorio vacío. En este caso tenemos: $n = 0$ y $x = 0$.
3. En este caso no nos sirve el valor de n como función cota, ya que su valor crecerá en cada vuelta del bucle para poder alcanzar el valor N . Observamos que n nunca superará el valor de N , por lo tanto podemos elegir como función cota $N - n$.
4. La función de avance debe hacer disminuir la función cota, de forma que el bucle avance hacia su finalización. Esto se consigue incrementando el valor de n : $n = n + 1$. El predicado intermedio que obtenemos es $I_n^{n+1} \equiv 0 \leq n + 1 \leq N \wedge x = (\sum i : 0 \leq i < n + 1 : V[i])$
5. Para restaurar el invariante observamos que $(\sum i : 0 \leq i < n + 1 : V[i]) \equiv (\sum i : 0 \leq i < n : V[i] + V[n])$. Por lo tanto la instrucción debe incrementar el valor de x en $V[n]$.

El algoritmo obtenido con este proceso de derivación es

```

1  int suma(int V[], int N) {
2  int n, x;
3  n = 0;
4  x = 0;
5  while (n < N)
6  {
7      x = x+V[n];
8      n = n+1;
9  }
10 return x;
11 }
```

Se podrían definir otros invariantes que darían lugar a algoritmos ligeramente diferentes de los dos presentados.

Observamos también que la instrucción de avance aparece siempre al final del bucle cuando seguimos el proceso de derivación. Un algoritmo en que se ejecutase la instrucción $n = n + 1$ antes que la instrucción $x = x + V[n]$ podría ser correcto, pero su construcción no garantizaría su corrección por lo que habría que verificarlo.

Ejemplo de bucle con condición de parada múltiple. Derivar un algoritmo lineal que dado un vector a de n enteros devuelva la primera posición en la que haya un 0, y n en caso de que no haya ninguno. La especificación del algoritmo se puede dar de la siguiente forma:

$$\{ 0 \leq n \leq longitud(a) \}$$

fun buscar-cero(**int** a[], **int** n) **return** **int** r

$$\{ r = (\text{máx } i : 0 \leq i \leq n \wedge (\forall j : 0 \leq j < i : a[j] \neq 0)) : i \}$$

La postcondición se puede escribir también de la siguiente forma:

$$0 \leq r \leq n \wedge (\forall j : 0 \leq j < r : a[j] \neq 0) \wedge (r = n \vee (0 \leq r < n \wedge_c a[r] = 0))$$

1. En este tipo de búsquedas se puede definir como invariante

$$0 \leq r \leq n \wedge (\forall j : 0 \leq j < r : a[j] \neq 0)$$

es decir, con él se representa el recorrido del vector de izquierda a derecha y los elementos por los que se ha ido avanzando son distintos de 0. El bucle terminará o bien cuando r llegue a n , en cuyo caso todos eran distintos de 0, o bien cuando $a[r] = 0$ y por tanto hemos encontrado el primer 0. Luego, formalmente, la condición del bucle es $r < n \wedge_c a[r] \neq 0$. La conjunción con cortocircuito es esencial para que no se produzcan accesos indebidos al vector: cuando $r = n$, si $n = longitud(a)$ se produciría un acceso fuera de rango.

2. La inicialización es $r = 0$, que satisface inicialmente el invariante, ya que el rango de la cuantificación universal es vacía.
3. Para avanzar basta con ir al siguiente elemento en el recorrido $r = r + 1$ siendo la cota $n - r$.
4. No es necesario restablecer ya que $I_r^{r+1} \equiv 0 \leq r+1 \leq n \wedge (\forall j : 0 \leq j < r+1 : a[j] \neq 0)$ y por tanto

$$I \wedge (r < n \wedge a[r] \neq 0) \Rightarrow I_r^{r+1}$$

El algoritmo por tanto queda de la siguiente forma:

```

1  int buscar-cero(int a[ ],int n){
2      int r=0;
3      while (r<n && a[r] != 0)
4          { r=r+1; }
5      return r;
6  }
```

5. Derivación de algoritmos iterativos típicos

5.1. Uso de variables acumuladoras para evitar bucles anidados

Veamos un ejemplo en el que se utiliza una variable acumuladora para evitar un bucle anidado.

Un número binario $b_n \dots b_0$ (con $n < longitud(v)$) se puede representar como un vector $v[]$ de 0s y 1s donde $v[i] = b_i$ para $i = 0, \dots, n$. Derivar un algoritmo lineal que, dado un número binario en forma de vector v y un entero n (indicando la posición de la cifra más significativa), calcule el número decimal correspondiente.

Primero planteamos la especificación de la función:

$$\{P \equiv 0 \leq n < longitud(v) \wedge \forall j : 0 \leq j \leq n : v[j] \in \{0, 1\}\}$$

fun decimal(**int** v[],**int** n) **return** **int** d

$$\{Q \equiv d = (\sum j : 0 \leq j \leq n : v[j] * 2^j)\}$$

1. En este caso el invariante se puede obtener sustituyendo n por una variable nueva i que comienza valiéndolo 0 y termina valiéndolo n , lo que representa un recorrido de izquierda a derecha en el vector. Así, por el momento, el invariante sería $I \equiv 0 \leq i \leq n \wedge d = (\sum j : 0 \leq j \leq i : v[j] * 2^j)$ y la condición del bucle $i < n$, es decir, cuando $i = n$ obtenemos la postcondición.
2. Para hacer cierto el invariante inicialmente asignamos $i = 0$, en cuyo caso d debe ser $(\sum j : 0 \leq j \leq 0 : v[j] * 2^j)$, es decir $v[0]$. Así:

$$P \Rightarrow (I_d^{v[0]})_i^0$$

3. Para avanzar hacia la derecha hacemos $i = i + 1$, luego en este caso una cota que decrece es $n - i$. Se cumple $I \wedge i < n \Rightarrow n - i \geq 0$ trivialmente.
4. Calculamos

$$I_i^{i+1} \equiv 0 \leq i + 1 \leq n \wedge d = \left(\sum j : 0 \leq j \leq i + 1 : v[j] * 2^j \right)$$

Para restablecer el invariante vemos que falta un sumando en d : $v[i + 1] * 2^{i+1}$ por lo que hacemos $d = d + v[i + 1] * 2^{i+1}$. Se cumple entonces que :

$$I \wedge i < n \Rightarrow (I_i^{i+1})_d^{d+v[i+1]*2^{i+1}}$$

ya que

$$(I_i^{i+1})_d^{d+v[i+1]*2^{i+1}} \equiv 0 \leq i+1 \leq n \wedge d+v[i+1]*2^{i+1} = \left(\sum j : 0 \leq j \leq i + 1 : v[j] * 2^j \right)$$

5. Fácilmente se comprueba que $I \wedge i < n \wedge n - i = T \Rightarrow n - (i + 1) < T$.

Así, el algoritmo queda:

```

1  int decimal(int v[ ],int n){
2  int i,d;
3  i=0;d=v[0];
4  while (i<n)
5  {
6      d=d+v[i+1]* 2^(i+1);
7      i=i+1;
8  }
9  return d;
10 }
```

Para calcular la potencia $2^{(i+1)}$ podríamos utilizar un bucle acumulando el producto

```

1  p=2; for (int l=0;l<i;l++) {p=2*p}
```

en cuyo caso el coste del algoritmo

```

1
2  i=0;d=v[0];
3  while (i<n)
4  {
5      p=2; for (int l=0;l<i;l++) {p=2*p};
6      d=d+v[i+1]* p;
7      i=i+1;
8  }
9  return d;
```

será cuadrático.

El invariante del bucle principal sería el mismo de antes y el del bucle *for* sería $0 \leq i \leq n \wedge d = (\sum j : 0 \leq j \leq i : V[j] * 2^j) \wedge 0 \leq l \leq i \wedge p = 2^{(l+1)}$. En este caso la variable p es local al cuerpo del bucle, su valor no se aprovecha entre dos vueltas del bucle.

Usando un algoritmo algo más inteligente para cálculo de potencias se podría conseguir un coste logarítmico y por tanto el cálculo total tendría coste $O(n \log n)$.

Para conseguir que el coste del algoritmo sea lineal, podemos aprovechar el bucle principal para ir acumulando la potencia de forma que cada vez sólo se hace una multiplicación. Para ello añadimos al invariante la variable $p = 2^i$ (puesto que en la asignación a d necesitamos $2^{(i+1)}$ y esto se tiene al sustituir i por $i + 1$). La inicialización debe incluir por tanto $p = 2^0 = 1$ y para restablecerla basta con multiplicar por 2:

```

1  int decimal(int v[ ],int n){
2  int i,d,p;
3  i=0;d=v[0];p=1;
4  while (i<n)
5  {   p=p*2;
6      d=d+v[i+1]*p;
7      i=i+1;
8  }
9  return d;
10 }
```

El orden en que se restablecen p y d es esencial para la corrección del algoritmo, si las intercambiamos sería incorrecto. La sustitución $((I_i^{i+1})_d^{d+v[i+1]*p})_p^{p*2}$ resulta:

$$0 \leq i + 1 \leq n \wedge d + v[i + 1] * p * 2 = \left(\sum j : 0 \leq j \leq i + 1 : V[j] * 2^j \right) \wedge p * 2 = 2^{i+1}$$

mientras que $((I_i^{i+1})_p^{p*2})_d^{d+v[i+1]*p}$ es

$$0 \leq i + 1 \leq n \wedge d + v[i + 1] * p = \left(\sum j : 0 \leq j \leq i + 1 : V[j] * 2^j \right) \wedge p * 2 = 2^{i+1}$$

lo cual quiere decir que estaríamos multiplicando $v[i + 1]$ por 2^i en lugar de $2^{(i+1)}$.

Obsérvese que en el caso de haber escrito una especificación distinta, como:

$\{P \equiv 0 < n \leq longitud(v) \wedge \forall j. 0 \leq j \leq n - 1. v[j] \in \{0, 1\}\}$
fun decimal(**int** v[],**int** n) **return** **int** d
 $\{Q \equiv d = (\sum j : 0 \leq j \leq n - 1 : V[j] * 2^j)\}$

aplicando los pasos anteriores y utilizando el invariante

$$0 < i \leq n \wedge d = \left(\sum j : 0 \leq j \leq i - 1 : V[j] * 2^j \right) \wedge p = 2^{(i-1)}$$

el algoritmo sería:

```

1  int decimal(int v[ ],int n){
2  int i,d,p;
3  i=1;d=v[0];p=1;
4  while (i<n)
5  {   p=p*2;
6      d=d+v[i]*p;
7      i=i+1;
8  }
9  return d;
10 }
```

y en caso de haber usado el invariante

$$0 \leq i \leq n-1 \wedge d = \left(\sum_{j: 0 \leq j \leq i} V[j] * 2^j \right) \wedge p = 2^i$$

el algoritmo sería:

```

1  int decimal(int v[ ],int n) {
2  int i,d,p;
3  i=0;d=v[0];p=1;
4  while (i<n-1)
5  { p=p*2;
6    d=d+v[i+1]*p;
7    i=i+1;
8  }
9  return d;
10 }
```

5.2. Cálculo de la moda

La moda es el valor que mas veces aparece repetido en una colección. En el tema 2 vimos una especificación del problema del cálculo de la moda.

$$\{1 \leq n \leq longitud(a)\}$$

fun moda (**int** a[],**int** n) **return** **int** m

$$\{frec(m, a, n) = \text{máx } w : 0 \leq w < n : frec(a[w], a, n)\}$$

donde el predicado $frec(x, v, n) \stackrel{\text{def}}{=} \#l : 0 \leq l < n : v[l] = x$.

Esta especificación es no determinista, ya que si existen varios valores que se repiten el mismo número de veces, cualquiera de ellos hace cierta la postcondición. Para simplificar la explicación no impondremos más requisitos a nuestro algoritmo. Si quisiésemos determinar un único valor, por ejemplo el que aparezca primero en el vector, habría que fortalecer la postcondición dada añadiendo más predicados.

La primera idea para resolver el problema es calcular la frecuencia de cada elemento y quedarnos con el máximo. Sin embargo dado que calcular la frecuencia de un elemento con un bucle tiene coste lineal en el número de elementos, este algoritmo tendrá complejidad cuadrática.

```

1 int m = a[0]; int fmax = 0;
2 for (int i = 0; i < n; i++) {
3   int f = 0;
4   for (int j = 0; j < n; i++) if (a[j] == a[i]) ++f;
5   if (fmax < f) { fmax = f; m = a[i]; }
6 }
```

Para mejorar el coste debemos contar la frecuencia de todos los elementos recorriendo el vector una única vez. Observamos que si el vector está ordenado, todos los elementos iguales aparecen juntos y podemos implementar el cálculo de la frecuencia de todos los elementos recorriendo el vector una única vez.

Derivamos el siguiente algoritmo, en cuya precondition se exige que el vector esté ordenado:

$$\{1 \leq n \leq longitud(a) \wedge ord(a)\}$$

fun moda (**int** a[],**int** n) **return** **int** m

$$\{frec(m, a, n) = \text{máx } w : 0 \leq w < n : frec(a[w], a, n)\}$$

La idea es que mientras se ejecuta el bucle la variable m tenga el valor de la moda correspondiente a la parte del vector que ya se ha recorrido.

$$\leftarrow m = 4 \rightarrow$$

1	1	4	4	4	6	7	7	10	11
---	---	---	---	---	---	---	---	----	----

k

Necesitamos dos variables auxiliares que indiquen la frecuencia del elemento que más veces se ha repetido: fm y la frecuencia del elemento $k - 1$ que denominamos f .

Cuando k avanza al siguiente elemento, puede ocurrir que este elemento sea igual al anterior, en cuyo caso debemos incrementar f y comprobar si el valor de fm sigue siendo mayor que el de f o si hemos encontrado un elemento que se repite más veces. Si el elemento es distinto, entonces debemos poner su frecuencia f a 1, ya que es la primera ocurrencia.

Derivemos un algoritmo correcto siguiendo estas ideas:

1. El invariante se obtiene generalizando la postcondición con una variable k que indica el avance de izquierda a derecha en el vector. Añadimos también la definición de las variables f y fm . $\{I \equiv \text{frec}(m, a, k) = \text{máx } w : 0 \leq w < k : \text{frec}(a[w], a, k) \wedge f = \text{frec}(a[k-1], a, k) \wedge fm = \text{frec}(m, a, k) \wedge f \leq fm\}$.
2. La condición de parada es $k == n$ y por lo tanto la condición del bucle $k < n$.
3. Inicializamos las variables libres m , k , f y fm de forma que el rango de la función max tenga un único elemento. Observar que la función máx está indefinida para rangos vacíos. Las instrucciones son $k = 1$, $m = a[0]$, $f = 1$ y $fm = 1$.
4. El valor de k debe incrementarse para alcanzar la condición de parada. Por lo tanto la instrucción de avance es $k = k + 1$ y la función cota $n - k$.
5. Para mantener el invariante hacemos una distinción de casos como indicamos en la explicación:
 - $a[k] == a[k-1] \wedge fm > f$. Para recuperar el invariante basta con incrementar el valor de f . El valor de fm no se modifica, ya que el obtenido en f es menor y por lo tanto no es necesario modificar el valor de m .
 - $a[k] == a[k-1] \wedge fm == f$. Para recuperar el invariante es necesario incrementar la variable f y modificar las variables fm y m , dándoles el valor de f y $a[k]$ respectivamente, ya que el elemento k es el que más veces se repite.
 - Observamos que no es posible el caso $fm < f$ ya que el invariante afirma $fm \geq f$.
 - $a[k] \neq a[k-1]$. Para recuperar el invariante es necesario modificar el valor de f para que tenga la frecuencia del valor $a[k]$ que es 1 ya que es la primera ocurrencia de este elemento. Recordad que el vector está ordenado.
6. El algoritmo garantiza que la función cota decrece, ya que las variables k y n sólo se modifican en la instrucción de avance.

El algoritmo obtenido es:

```

ordenar(a,n);
int k = 1; int m = a[0]; int f = 1; int fm = 1;
while (k < n) {
    if (a[k] == a[k-1])
        if (fm == f) {
            ++fm; ++f; m = a[k];
        }
        else if (fm > f)
            ++f;
        else f = 1;
        ++k;
}

```

El coste del algoritmo implementado es la suma del coste de ordenar el vector $\mathcal{O}(n \log n)$ más el coste de calcular el elemento que mas se repite: $\mathcal{O}(n)$, siendo n el número de elementos del vector. Por lo tanto el coste final es $\mathcal{O}(n \log n)$.

¿Podemos hacerlo mejor?. Para ello tenemos que evitar ordenar el vector. ¿Podemos calcular el número de veces que se repite cada elemento en tiempo lineal y sin ordenar el vector?. La idea en este caso es utilizar memoria auxiliar. Declaramos un nuevo vector con tantas componentes como posibles valores del vector de entrada. En él acumularemos las apariciones de cada elemento. Observad que este método no puede emplearse si el número de posibles valores del vector de entrada es muy grande, ya que entonces el vector auxiliar para acumular las apariciones podría sobrepasar la memoria disponible.

El algoritmo se deriva en dos etapas. Primero se crea el vector con las apariciones de cada elemento y después se busca el máximo de este vector. La especificación es:

$$\{1 \leq n \leq \text{longitud}(v) \wedge 1 \leq t \leq \text{longitud}(a) \wedge \forall i : 0 \leq i < n : 0 \leq v[i] < t\}$$

```

proc etapa1 (int v[], int n, out int a[], int t)
    { $\forall s : 0 \leq s < t : a[s] = \text{frec}(s, v, n)$ }
fun etapa2 (int a[], int t) return int m
    { $m = \text{máx } w : 0 \leq w < t : \text{frec}(a[w], a, t)$ }

```

Para derivar el algoritmo correspondiente a la primera etapa, generalizamos la post-condición con una variable k que indica el avance de izquierda a derecha en el vector v : $\{\forall s : 0 \leq s < t : a[s] = \text{frec}(s, v, k)\}$. La condición de parada del bucle será $k == n$ y por lo tanto la condición del bucle $k < n$. Las instrucciones de inicialización que hacen el invariante cierto son: $k = 0$ e inicializar todas las componentes del vector a a cero. La instrucción de avance para alcanzar la condición de parada será $k = k + 1$.

Para mantener el invariante en cada vuelta del bucle observamos que al considerar el elemento k se incrementa en uno la frecuencia con la que aparece $v[k]$ en el vector, por lo tanto debemos incrementar el valor de la componente $a[v[k]]$ del vector a .

Observamos que la función cota $n - k$ decrece en cada vuelta del bucle.

El algoritmo correspondiente a la etapa 1 queda:

```

int k = 0;
for (int i = 0; i < t; ++i) a[i] = 0;
while (k < n) {
    a[v[k]] = a[v[k]] + 1;
    k = k + 1;
}

```

El algoritmo correspondiente a la etapa 2 consiste en calcular el máximo de un vector. Su derivación no debería presentar ninguna dificultad para el alumno.

Cada una de las dos etapas tienen coste lineal en el número de elementos del vector, por lo que el algoritmo obtenido tiene coste $\mathcal{O}(n)$ siendo n el número de elementos del vector.

Para terminar, una observación sobre la verificación y derivación de la instrucción de asignación a componentes de un vector. La especificación/derivación de la instrucción de asignación debe considerar el vector completo, no solamente la componente a la que se le asigna el valor. Esto se debe a que el índice del vector de la componente que se modifica puede resultar también modificado p.e. $v[v[2]] = 1$, invalidando la regla de la asignación utilizada hasta ahora. Para considerar el vector completo se trata la asignación $v[i] = e$ como $v = \text{asig}(v, i, e)$ donde el vector *asig* queda definido como:

$$\text{asig}(v, i, e)[j] = \begin{cases} e & \text{si } i = j \\ v[j] & \text{si } i \neq j \text{ y } j \text{ está en el rango de los índices de } v \end{cases}$$

5.3. Segmento de longitud máxima

Vamos a implementar un algoritmo que resuelva el problema de calcular el mayor número de personas que han nacido en un periodo de p años si tenemos los años de nacimiento de cada una de ellas ordenados de menor a mayor (🐛ACR346). La especificación de este problema es:

```
{  $1 \leq n \leq \text{longitud}(v) \wedge p \geq 1 \wedge \text{ord}(v)$  }
fun seg-max(int v[ ], int n, int p) return int r
{  $r = (\text{máx } i, j : 0 \leq i \leq j < n \wedge (v[j] - v[i] < p) : j - i + 1)$  }
```

Observamos que nos piden el número de elementos del mayor subvector que cumpla que la diferencia entre el valor del último elemento del subvector y el primero sea menor que p . Por ejemplo, en el siguiente vector, si $p = 5$, el subvector de longitud máxima que cumple los requisitos es el comprendido entre las dos flechas y su longitud es $r = 6$.

1	1	4	4	5	6	6	7	10	11
		↑					↑		

La idea es mantener invariante el valor de r como la longitud del subvector de longitud máxima que cumple el requisito, en la parte del vector tratada hasta este momento en el bucle. En cada vuelta del bucle consideramos un elemento más del vector: k , como el vector está ordenado, este elemento sólo afecta al subvector que incluye este elemento. Necesitamos por lo tanto información en el programa sobre el subvector que incluye al último elemento. En nuestro caso mantendremos el índice del vector, s , en que comienza el último subvector que cumple la propiedad. Se podría optar por mantener la longitud del último subvector, ya que a partir de cualquiera de los dos valores podemos obtener el otro.

$\leftarrow \quad r \quad = \quad 5 \quad \rightarrow$									
1	1	4	4	5	6	6	7	10	11
s				k					

Cuando k avanza al siguiente elemento, puede ocurrir que el último subvector siga cumpliendo la propiedad, en cuyo caso si su longitud es mayor que la que teníamos almacenada en r debemos cambiar el valor de r al nuevo máximo.

$$\leftarrow r = 6 \rightarrow$$

1	1	4	4	5	6	6	7	10	11
s				k					

Por el contrario, si al avanzar el subvector deja de cumplir la propiedad entonces debemos avanzar el índice s hasta que el subvector incluyendo el nuevo elemento vuelva a cumplir la propiedad.

$$\leftarrow r = 6 \rightarrow$$

1	1	4	4	5	6	6	7	10	11
s				k					

Derivamos un algoritmo correcto para resolver el problema.

1. El invariante se obtiene generalizando la postcondición con una variable k que indica el avance de izquierda a derecha en el vector. Añadimos también la definición de la variable s que indica el comienzo del subvector que incluye el elemento $k - 1$ y que cumple la propiedad requerida: $\{ I \equiv (ord(v) \wedge 1 \leq k \leq n) \wedge (1 \leq p) \wedge (r = (\text{máx } i, j : 0 \leq i \leq j < k \wedge (v[j] - v[i] < p) : j - i + 1) \wedge (0 \leq s < k) \wedge (v[k - 1] - v[s] < p) \wedge (k - s \geq (\text{máx } i : 0 \leq i < k \wedge (v[k - 1] - v[i] < p) : (k - 1) - i + 1))) \}$. Mantenemos en el invariante la propiedad de que el vector está ordenado para poder derivar la instrucción que recupera el invariante utilizando únicamente la información relativa al último subvector.
2. La condición de parada es: $k == n$, por lo tanto la condición del bucle es: $k < n$.
3. Para fijar las instrucciones de inicialización observamos que el vector es no vacío, por lo tanto podemos seleccionar $k = 1$. El único valor de r que hace cierto el invariante para este valor de k es $r = 1$. Igualmente el único valor posible de s es $s = 0$.
4. El valor de k debe incrementarse para alcanzar la condición de parada, pero nunca debe sobrepasar el valor de n . Tenemos por lo tanto que la función cota es $n - k$ y la instrucción de avance $k = k + 1$.
5. Para mantener el invariante hacemos una distinción de casos como indicamos al principio de la explicación:
 - $v[k] - v[s] < p$. El elemento que estamos considerando incrementa la longitud del último subvector. Para recuperar el invariante la variable r debe tomar el valor máximo entre el que tiene y la longitud del último subvector $k - s + 1$. El valor de s no varía ya que si $k - s$ era la longitud del mayor subvector acabando en la componente $k - 1$ se cumple que $(k + 1) - s$ es la longitud del mayor subvector acabando en k .

- $v[k] - v[s] \geq p$ entonces para recuperar el invariante es necesario modificar la variable s , ya que no se cumple la parte del predicado I_k^{k+1} que indica $(v[k] - v[s] < p)$. Para que se cumpla avanzamos s hasta que se vuelva a cumplir la propiedad. Esto supone la realización de un bucle: `while (v[k] - v[s] >= p) ++s`. Cuando el bucle termina se cumple $(v[k] - v[s] < p)$. La variable r no es necesario modificarla ya que al incrementar s la longitud del intervalo nunca será mayor. La función cota es la longitud del intervalo $k - s + 1$ que decrece en cada vuelta de este bucle. La condición de parada se hace cierta en algún momento, porque cuando $s == k$ entonces $v[k] - v[s] == 0$ y el invariante garantiza $p \geq 1$
6. El algoritmo garantiza que la función cota del bucle principal $n - k$ decrece, ya que la única modificación de las variables n y k se produce en la función de avance $k = k + 1$. El algoritmo que obtenemos es:

```

1  int k = 1; int r = 1; int s = 0;
2  while (k < n) {
3      if (v[k] - v[s] < p)  r = std::max(r, k-s+1);
4      else while (v[k] - v[s] >= p) ++s;
5      ++k;
6  }
```

Observar que, aunque el algoritmo tiene un doble bucle anidado, el coste es lineal respecto al tamaño del vector. Esto se debe a que el bucle interno no se ejecuta en total mas de n veces ya que la variable s nunca disminuye su valor, solo se incrementa y el invariante garantiza $s < n$.

5.4. Algoritmo de partición

Dado un vector definido entre dos índices a y b , el problema pide separar las componentes del vector, dejando en la parte izquierda aquellos valores que sean menores o iguales que el valor de la componente $v[a]$ y en la parte derecha aquellos valores que sean mayores o iguales que el valor de dicha componente. El valor de $v[a]$ debe quedar separando ambas partes. Este algoritmo se utiliza en la implementación del algoritmo de ordenación rápida (*quicksort*).

Partimos de la especificación:

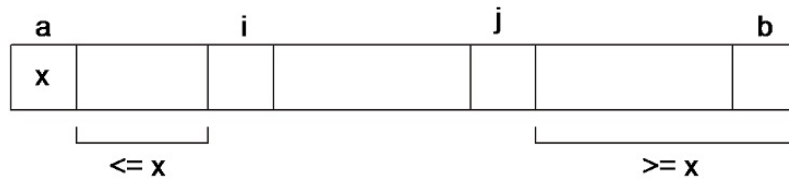
$$\{0 \leq a \leq b \leq \text{longitud}(v) - 1\}$$

```

proc particion(TElem v[], int a, int b, out p)
   $\{0 \leq a \leq p \leq b \leq \text{longitud}(v) - 1 \wedge (\forall x : a \leq x \leq p - 1 : v[x] \leq v[p])$ 
   $\wedge (\forall y : p + 1 \leq y \leq b : v[y] \geq v[p])\}$ 
```

La idea es obtener un bucle que mantenga invariante la situación de la Figura 1, de forma que i y j se vayan acercando hasta cruzarse, y finalmente intercambiamos $v[a]$ con $v[j]$.

- El invariante se obtiene generalizando la postcondición con la introducción de dos variables nuevas, i, j , que indican el avance por los dos extremos del sub-vector
- $$a + 1 \leq i \leq j + 1 \leq b + 1 \wedge$$
- $$(\forall x : a + 1 \leq x \leq i - 1 : v[x] \leq v[a]) \wedge (\forall y : j + 1 \leq y \leq b : v[y] \geq v[a])$$

Figura 1: Diseño de *particion*

- Condición de repetición

El bucle termina cuando se cruzan los índices i y j , es decir, cuando se cumple $i = j + 1$, y, por lo tanto, la condición de repetición es

$$i \leq j$$

A la salida del bucle, el vector estará particionado salvo por el pivote $v[a]$. Para terminar el proceso basta con intercambiar los elementos de las posiciones a y j , quedando la partición en la posición j .

```
p = j;
aux = v[a];
v[a] = v[p];
v[p] = aux;
```

- Expresión de acotación

$C : j - i + 1$

- Acción de inicialización

$i = a + 1;$
 $j = b;$

Esta acción hace trivialmente cierto el invariante porque $v[(a + 1)..(i - 1)]$ y $v[(j + 1)..b]$ se convierten en subvectores vacíos.

- Acción de avance

El objetivo del bucle es conseguir que i y j se vayan acercando, y además se mantenga el invariante en cada iteración. Para ello, se hace un análisis de casos comparando las componentes $v[i]$ y $v[j]$ con $v[a]$

- $v[i] \leq v[a] \rightarrow$ incrementamos i
- $v[j] \geq v[a] \rightarrow$ decrementamos j
- $v[i] > v[a] \wedge v[j] < v[a] \rightarrow$ intercambiamos $v[i]$ con $v[j]$, incrementamos i y decrementamos j

De esta forma el avance del bucle queda

```
if      ( v[i] <= v[a] ) i = i + 1;
else if ( v[j] >= v[a] ) j = j - 1;
else {  // (v[i] > v[a]) && (v[j] < v[a])
    aux = v[i];
    v[i] = v[j];
    v[j] = aux;
    i = i + 1;
    j = j - 1;
}
```

Nótese que las dos primeras condiciones no son excluyentes entre sí pero sí con la tercera y, por lo tanto, la distinción de casos se puede optimizar teniendo en cuenta esta circunstancia.

- Con todo esto el algoritmo queda:

```

void particion ( TElem v[], int a, int b, int & p) {
// Pre:  $0 \leq a \leq b \leq longitud(v) - 1$ 

    int i, j;
    TElem aux;

    i = a+1;
    j = b;

    while ( i <= j ) {
        if ( (v[i] > v[a]) && (v[j] < v[a]) ) {
            aux = v[i]; v[i] = v[j]; v[j] = aux;
            i = i + 1; j = j - 1;
        }
        else {
            if ( v[i] <= v[a] )
                i = i + 1;
            if ( v[j] >= v[a] )
                j = j - 1;
        }
    }

    p = j;
    aux = v[a]; v[a] = v[p]; v[p] = aux;

// Post:  $0 \leq a \leq p \leq b \leq longitud(v) - 1 \wedge$ 
//          $\forall x : a \leq x \leq p - 1 : v[x] \leq v[p] \wedge$ 
//          $\forall y : p + 1 \leq y \leq b : v[y] \geq v[p]$ 
}

```

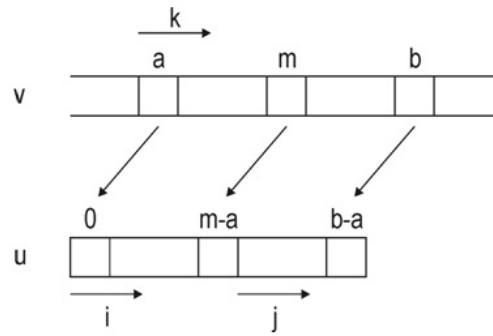
5.5. Mezcla de dos vectores ordenados

Dado un vector definido entre dos índices a y b y una posición intermedia entre ambos valores $a \leq m \leq b$ tal que se cumple que los subvectores $v[a..m]$ y $v[m+1..b]$ están ordenados, se pide ordenar los valores de forma que al finalizar el proceso el vector $v[a..b]$ este ordenado. Este algoritmo se utiliza en la implementación del algoritmo de ordenación por mezclas (*mergesort*).

Para conseguir una solución eficiente, $O(n)$, utilizaremos un vector auxiliar donde iremos realizando la mezcla, para luego copiar el resultado al vector original.

La idea del algoritmo es colocarse al principio de cada subvector e ir tomando, de uno u otro, el menor elemento, y así ir avanzando. Uno de los subvectores se acabará primero y habrá entonces que copiar el resto del otro subvector. En el array auxiliar tendremos los índices desplazados pues mientras el subvector a mezclar es $v[a..b]$, en el array auxiliar tendremos los elementos almacenados en $v[0..b-a]$, y habrá que ser cuidadoso con los índices que recorren ambos arrays.

Con todo esto, el procedimiento de mezcla queda:

Figura 2: Diseño de *mezcla*

```

void mezcla( TElem v[], int a, int m, int b ) {
// Pre:  $0 \leq a \leq m < b \leq \text{longitud}(v) - 1 \wedge \text{ord}(v, a, m) \wedge \text{ord}(v, m + 1, b)$ 

    TElem *u = new TElem[b-a+1];
    int i, j, k;

    for ( k = a; k <= b; k++ )
        u[k-a] = v[k];
    i = 0;
    j = m-a+1;
    k = a;
    while ( ( i <= m-a ) && ( j <= b-a ) ) {
        if ( u[i] <= u[j] ) {
            v[k] = u[i];
            i = i + 1;
        } else {
            v[k] = u[j];
            j = j + 1;
        }
        k = k + 1;
    }
    while ( i <= m-a ) {
        v[k] = u[i];
        i = i+1;
        k = k+1;
    }
    while ( j <= b-a ) {
        v[k] = u[j];
        j = j+1;
        k = k+1;
    }
    delete[] u;
// Post:  $\text{ord}(v, a, b)$ 
}

```

5.6. Algoritmos de matrices

Veamos como ejemplo un algoritmo que suma los elementos de una matriz. La especificación del algoritmo es:

$P \equiv \{ 0 \leq N \wedge 0 \leq M \}$
fun sumaMatriz(**int** v[N][M]) **return** **int** s
 $Q \equiv \{ s = (\sum i : 0 \leq i < N : \sum j : 0 \leq j < M : v[i][j]) \}$

- a) Derivamos en primer lugar un algoritmo que resuelva el sumatorio de cada fila de la matriz, es decir $\{ Q_2 \equiv s_2 = (\sum j : 0 \leq j < M : v[n][j]) \}$. Obtenemos el invariante generalizando la postcondición: $\{ I_2 \equiv s_2 = (\sum j : 0 \leq j < m : v[n][j]) \}$. La condición de parada de este bucle es $m == M$, por lo tanto la condición del bucle es: $m < M$.

Las instrucciones de inicialización buscan hacer el rango del sumatorio vacío, por lo tanto pondremos $m = 0$ y $s_2 = 0$.

La instrucción de avance es $m = m + 1$, con lo que se alcanza la condición de parada. El predicado intermedio queda: $\{ s_2 = (\sum j : 0 \leq j < m + 1 : v[n][j]) \}$. Para recuperar el invariante necesitamos una instrucción que incremente el valor de s_2 .

El algoritmo que resuelve el problema intermedio es:

```

1      int m = 0; s2 = 0;
2      while (m < M) {
3          s2 = s2 + v[n][m];
4          m = m + 1;
5      }
```

- b) Derivamos ahora el algoritmo completo utilizando el algoritmo que acabamos de obtener. Comenzamos generalizando la postcondición para obtener el invariante: $\{ I_1 \equiv s_1 = (\sum i : 0 \leq i < n : \sum j : 0 \leq j < M : v[i][j]) \}$. La condición de parada del bucle es $n == N$ y la condición del bucle $n < N$. La instrucción de avance $n = n + 1$. El predicado intermedio antes de ejecutar la instrucción de avance es: $\{ s_1 = (\sum i : 0 \leq i < n + 1 : \sum j : 0 \leq j < M : v[i][j]) \}$.

Para obtener la instrucción que recupera el invariante, descomponemos el predicado anterior: $s_1 = (\sum i : 0 \leq i < n + 1 : \sum j : 0 \leq j < M : v[i][j]) \equiv s_1 = (\sum i : 0 \leq i < n : \sum j : 0 \leq j < M : v[i][j]) + \sum j : 0 \leq j < M : v[n][j]$. Añadimos la instrucción $s_1 = s_1 + s_2$ donde s_2 se calcula utilizando el algoritmo derivado en el primer paso.

El algoritmo final que hemos derivado es:

```

1      int n = 0; int s1 = 0;
2      while (n < N) {
3          int m = 0; int s2 = 0;
4          while (m < M) {
5              s2 = s2 + v[n][m];
6              m = m + 1;
7          }
8          s1 = s1 + s2;
9          n = n + 1;
10     }
```

El coste del algoritmo es $\mathcal{O}(N * M)$, siendo N y M las dimensiones de la matriz.

5.7. Algoritmos numéricos

Las ideas vistas en este tema se pueden aplicar a la derivación de cualquier algoritmo iterativo, no solamente a algoritmos que utilicen vectores. A continuación vemos como obtener algoritmos que resuelven problemas que manejan números en lugar de vectores.

Deriva un algoritmo que verifique la siguiente especificación:

```
{a ≥ 0 ∧ b > 0}
proc divide(int a,b; out int c,r)
{a = b * c + r ∧ 0 ≤ r < b}
```

Solución:

a) La postcondición tiene forma conjuntiva:

- $I \equiv a = b * c + r \wedge 0 \leq r$
- Condición del bucle: $r \geq b$

b) Tenemos que asignar valores a c y r para que el invariante se cumpla:

$$\{P\}c = 0; r = a\{I\}$$

$$\Leftrightarrow ((a = b * c + r \wedge 0 \leq r)_c^a)^0 \Leftrightarrow a = b * 0 + a \wedge 0 \leq a \Leftarrow a \geq 0$$

c) Probemos $cota=r$: $I \wedge B \Rightarrow r \geq 0$ (ya que $cota=c$ sería creciente pues c vale inicialmente 0).

d) Consideremos restar b a r : $R \equiv I_r^{r-b} \Leftrightarrow$

$$\Leftrightarrow a = b * c + (r - b) \wedge 0 \leq (r - b)$$

$$\Leftrightarrow a = b * (c - 1) + r \wedge 0 \leq (r - b)$$

$$\Leftarrow^? I \wedge B \text{ Falso, el invariante no se mantiene!!!}$$

e) Consideremos, además, incrementar c en 1:

$$\{I \wedge B\}c = c + 1\{R\} : R_c^{c+1} \Leftrightarrow a = b * c + r \wedge 0 \leq r - b \Leftarrow I \wedge B$$

f) Veamos que la cota, r , decrece en cada vuelta del bucle:

$$\{I \wedge B \wedge r = T\}c = c + 1; r = r - b\{r < T\}$$

$((r < T)_r^{r-b})_c^{c+1} \Leftrightarrow r - b < T \Leftarrow I \wedge B \wedge r = T$, que es cierto si usamos la propiedad $b > 0$ de la precondition. Dicha propiedad puede añadirse al invariante, puesto que se cumple inicialmente y la variable b no se modifica a lo largo del algoritmo.

El algoritmo resultante:

```
1  proc divide(int a,b; out int c, r){
2    c = 0;
3    r = a;
4    while (r >= b)
5    {
6        c = c+1;
7        r = r-b;
8    }
9 }
```

Veamos otro ejemplo.

Deriva un algoritmo que verifique la siguiente especificación:


```

{n ≥ 0}
fun raiz-entera(int n) return int r
{n ≥ 0 ∧ r2 ≤ n < (r + 1)2}

```

Solución A:

a) La postcondición tiene forma conjuntiva:

- $I \equiv r \geq 0 \wedge r^2 \leq n$
- Condición del bucle: $n \geq (r + 1)^2$

b) Para hacer cierto el invariante r puede valer 0:

$$n \geq 0 \Rightarrow I_r^0$$

c) Consideremos la instrucción avanzar: $r = r + 1$:

$$I_r^{r+1} \Leftrightarrow r + 1 \geq 0 \wedge (r + 1)^2 \leq n \Leftarrow I \wedge B = r \geq 0 \wedge r^2 \leq n \wedge n \geq (r + 1)^2$$

d) Algo que decrezca y sea fácil de calcular puede ser: $n - r$ Luego, no es preciso que haya más instrucciones en el cuerpo del bucle.

e) El algoritmo resultante:

```

1  int raiz-entera(int n) {
2    r = 0;
3    while (n >= (r+1)*(r+1))
4    {
5        r = r+1;
6    }
7    return r;
8  }

```

f) Número de iteraciones del bucle: $\sqrt{n} \Rightarrow \Theta(\sqrt{n})$.

Solución B:

a) La postcondición tiene forma conjuntiva:

- $I \equiv r \geq 0 \wedge n < (r + 1)^2$
- Condición del bucle: $r^2 > n$

b) Para hacer cierto el invariante r puede valer n.

$$n \geq 0 \Rightarrow I_r^n$$

c) Consideremos la instrucción avanzar: $r = r - 1$:

$$I_r^{r-1} \Leftrightarrow r - 1 \geq 0 \wedge n < ((r - 1) + 1)^2 \Leftarrow I \wedge r^2 > n = r \geq 0 \wedge n < (r + 1)^2 \wedge r^2 > n$$

$$\underbrace{r \geq 0}_{\text{Invariante}} \wedge \underbrace{n \geq 0}_{\text{Precondición}} \wedge \underbrace{r^2 > n}_{\text{CondBucle}} \Rightarrow r > 0 \Rightarrow r - 1 \geq 0$$

d) Cota=r

e) El algoritmo resultante:

```

1  int raiz-entera(int n) {
2    r = n;
3    while (r*r > n)
4    {
5        r = r-1;
6    }
7    return r;
8  }

```

f) Número de iteraciones del bucle: $n - \sqrt{n} \Rightarrow \Theta(n)$.

Notas bibliográficas

Se recomienda ampliar el contenido de estas notas estudiando los capítulos 1 y 4 de (Peña, 2005).

Para coger agilidad en la aplicación de estas técnicas se recomiendan los capítulos 2, 3 y 4 de Martí Oliet et al. (2012) en los que se pueden encontrar numerosos ejemplos resueltos.

Ejercicios

1. Dado el siguiente algoritmo, determina su complejidad.

```
1  int valor(int n){
2  int i = 1, j, aux, r = 3;
3  while (i <= n){
4      j = 1;
5      while (j <= 20){
6          aux = j * 2;
7          aux = aux*i+r;
8          j++;
9      }
10     i++;
11 }
12 return aux;
13 }
```

2. Dado el siguiente algoritmo, determina su complejidad.

```
1  int valor(int n){
2  int i = 1, j, aux;
3  while (i <= n){
4      j = 1;
5      while (j <= n){
6          if (i < n){
7              aux = i + j;
8          }
9          else
10         {
11             aux = i - j;
12         }
13         j++;
14     }
15     i++;
16 }
17 return aux;
18 }
```

3. Dado el siguiente algoritmo, determina su complejidad.

```
1  int valor(int n){
2  int i = 1, j, aux;
3  while (i <= n){
```

```

4      j = 1;
5      while (j <= i) {
6          aux = i + j;
7          if (i + 2 < j) {
8              aux = aux * 2;
9          }
10         j++;
11     }
12     i++;
13 }
14 return aux;
15 }

```

4. Suponiendo que *int* x, y, n y *bool* b , determina el predicado más débil, P , que satisfaga la especificación dada

- a) $\{P\}x = 3 * x \{x \leq 20\}$
- b) $\{P\}x = 3 * x \{\forall i : 1 \leq i \leq n : x \neq 6 * i\}$
- c) $\{P\}x = x + 1 \{x = x + 1\}$

5. Suponiendo x, y enteros, calcula en cada caso el predicado P más débil que satisfaga la especificación dada:

- a) $\{P\}x = x + y; y = 2 * y - x \{y > 0\}$
- b) $\{P\}y = 4 * x; x = x * x - y; x = x + 4 \{x > 0\}$
- c) $\{P\}x = y; y = x \{x = A \wedge y = B\}$

6. Suponiendo x un número entero, determina el predicado más débil, P , que satisfaga la especificación dada:

```

{P}
if (x % 2 == 0) x = x / 2;
else x = x / 2 + 1;
{x=X}

```

donde X es un valor "constante".

7. Verifica el siguiente algoritmo:

$\{N \geq 0\}$

```

1  int suma(int V[], int N) {
2  int m, s;
3  s = 0;
4  m = 0;
5  while (m < N)
6  {
7      s = s + V[m];
8      m = m + 1;
9  }
10 return s;
11 }

```

$\{s = (\sum i : 0 \leq i < N : V[i])\}$

8. Suponiendo todas las variables enteras, demostrar la corrección de los tres programas siguientes para calcular potencias, con:

- Precondición: $P \equiv x = X \wedge y = Y \wedge Y \geq 0$.
- Postcondición $Q \equiv p = X^Y$.

a) _____

```

1      p = 1;
2      while (y != 0)
3      {
4          p = p*x;
5          y = y-1;
6      }
```

b) _____

```

1      p = 1;
2      while (y > 0)
3      {
4          if (par(y))
5          {
6              y = y / 2;
7              x = x*x;
8          }
9          else
10         {
11             y = y-1;
12             p = p*x;
13         }
14     }
```

c) _____

```

1      p = 1;
2      while (y > 0)
3      {
4          if (impar(y))
5          {
6              p = p*x;
7          }
8          y = y / 2;
9          x = x*x;
10     }
```

9. Demostrar la corrección de la siguiente especificación suponiendo x, y enteros.
- $\{x = X \wedge y = Y \wedge x > y > 0\}$

```

1      while (y != 0)
2      {
3          z = x;
4          x = y;
5          y = z % y;
6      }
```

$\{x = mcd(X, Y)\}$

10. Demuestra la corrección del siguiente programa:

```

1      r = 0;
2      while ( (r+1) * (r+1) <= x )
3      {
4          r = r+1;
5      }

```

$\{r \geq 0 \wedge r^2 \leq x < (r+1)^2\}$

11. Deriva un algoritmo que satisfaga la siguiente especificación:

$\{b > 1 \wedge n > 0\}$
fun log(**int** b, **int** n) **return int** r
 $\{b^r \leq n < b^{r+1}\}$

12. Especifica y deriva una función iterativa de coste lineal que reciba como argumento un vector de números enteros y calcule el producto de todos ellos.
13. Especifica y deriva una función que reciba como argumento un vector de números y calcule el máximo de todos ellos.
14. Deriva un algoritmo que satisfaga la siguiente especificación:

$\{N > 0\}$
fun pares(**int** V[N]) **return int** x
 $\{x = (\#i : 0 \leq i < N : V[i] \bmod 2 = 0)\}$

15. Deriva un algoritmo que satisfaga la siguiente especificación:

$\{N > 0\}$
fun suma-buenos(**int** X[N]) **return int** s
 $\{s = (\sum i : 0 \leq i < N \wedge \text{bueno}(i, X) : X[i])\}$

donde $\text{bueno}(i, X) \equiv (X[i] = 2^i)$. No se pueden emplear operaciones que calculen potencias.

16. **Definición:** Dado V[N] de enteros, el índice i es un pico si V[i] es el mayor elemento de V[0..i].
 Especifica y deriva un algoritmo de coste lineal que reciba un vector y calcule la suma de todos los valores almacenados en los picos de V.
17. Dado un vector de enteros X[N], el índice se llama “heroico” si X[i] es estrictamente mayor que el i-ésimo número de la sucesión de Fibonacci. Especifica y deriva un algoritmo de coste lineal que reciba como argumento un vector y determine cuántos índices heroicos tiene.
18. (ACR171) **Definición:** Dado X[N] de enteros, el índice i es un mirador si X[i] es el mayor elemento de X[i+1..n).
 Especifica y deriva un algoritmo de coste lineal que reciba un vector y calcule el número de miradores que hay en X.

19. **Definición:** Un vector $A[N]$ de enteros es *gaspariforme* si todas sus sumas parciales ($A[0] + \dots + A[i]$, $0 \leq i < n$) son no negativas y además la suma total es cero. Especifica y deriva una función de coste lineal que decida si $A[N]$ es gaspariforme.
20. (Martí Oliet et al. (2012)) Derivar un algoritmo de coste lineal (con respecto a la longitud del vector) que satisfaga la siguiente especificación:

$$\{ N \geq 2 \}$$

```

fun máx-resta(int A[N]) return int r
{  $r = (\text{máx } p, q : 0 \leq p < q < N : A[p] - A[q])$  }

```

21. (Martí Oliet et al. (2012)) Derivar un algoritmo de coste lineal (con respecto a la longitud del vector) que satisfaga la siguiente especificación:

$$\{ N \geq 0 \}$$

```

fun crédito-seg-máx(int A[N]) return int r
{  $r = (\text{máx } p, q : 0 \leq p \leq q \leq N : \text{crédito}(p, q))$  }

```

donde $\text{crédito}(p, q) = (\#i : p \leq i < q : A[i] > 0) - (\#i : p \leq i < q : A[i] < 0)$.

22. (Martí Oliet et al. (2012)) Derivar algoritmos de coste lineal (con respecto a la longitud del vector) para resolver los siguientes problemas de segmento de longitud máxima:

$$\{ N \geq 0 \}$$

```

fun seg-long-máx(int X[N]) return int r
{  $r = (\text{máx } p, q : 0 \leq p \leq q \leq N \wedge \mathcal{A}(p, q) : q - p)$  }

```

donde

- a) $\mathcal{A}(p, q) = (\forall i : p \leq i < q : X[i] = 0)$.
b) $\mathcal{A}(p, q) = (\forall i, j : p \leq i \leq j < q : X[i] = X[j])$.

23. (Martí Oliet et al. (2012)) Diseñar un algoritmo iterativo de coste lineal que satisfaga la siguiente especificación:

$$\{ N \geq 0 \}$$

```

fun separación(bool A[N]) return bool r
{  $r = \exists p : 0 \leq p \leq N : (\forall i : 0 \leq i < p : A[i]) \wedge (\forall j : p \leq j < N : \neg A[j])$  }

```

24. (Martí Oliet et al. (2012)) Diseñar un algoritmo iterativo de coste lineal que satisfaga la siguiente especificación:

$$\{ N \geq 1 \}$$

```

fun contar-ordenados(int A[N]) return int r
{  $r = \#i : 0 \leq i < N : (\forall p : i \leq p < N : A[i] \geq A[p])$  }

```

25. (ACR152) Dado un conjunto de valores, la *moda* es el valor (o valores) que más se repite en dicho conjunto. Diseñar un algoritmo iterativo de coste lineal que dado un vector de enteros positivos menores que 1000 devuelva una moda del vector.
26. (Martí Oliet et al. (2012)) Diseñar un algoritmo iterativo de coste lineal que satisfaga la siguiente especificación:

```

{N ≥ 0 ∧ h = H}
proc positivos-negativos(int h[N])
{perm(h, H) ∧ (∃p : 0 ≤ p ≤ N : (∀i : 0 ≤ i < p : h[i] ≤ 0) ∧ (∀j : p ≤ j <
N : h[j] < 0))}

```

27. (Martí Oliet et al. (2012)) Diseñar un algoritmo iterativo de coste lineal que satisfaga la siguiente especificación:

```

{N > 0}
fun máximos(int X[N]) return int h[N]
{∀i : 0 ≤ i < N : h[i] = (máx j : 0 ≤ j ≤ i : X[i])}

```

28. (Martí Oliet et al. (2012)) Diseñar un algoritmo iterativo de coste lineal llamado *fundir* que dados dos vectores de enteros a y b y un entero $n \geq 0$, obtenga un tercer vector c cuyas posiciones impares contengan los n primeros elementos de a en el mismo orden que allí figuraban y las pares contengan los n primeros elementos de b en el mismo orden que allí figuraban.
29. (La bandera holandesa, E. W. Dijkstra) Dado un vector que contiene bolas de colores rojo, azul y blanco, diseñar un algoritmo de coste lineal que coloque las bolas según la bandera holandesa: azul a la izquierda, blanca en el centro y roja a la derecha.
30. (La bandera aragonesa, (Peña (2005))) Dado un vector de bolas amarillas y rojas, diseñar algoritmos de coste lineal que resuelvan los siguientes problemas:
- Separar las bolas de forma que primero coloque las rojas y luego las amarillas.
 - Colocar las bolas para formar la bandera aragonesa: en las posiciones impares una roja y en las pares una amarilla. Las bolas restantes al final.
31. Diseñar un algoritmo que determine si una matriz M de dimensión $n \times n$ es o no simétrica con respecto a la diagonal principal.
32. Diseñar un algoritmo que dada una matriz M de dimensión $n \times n$ calcule su traspuesta M^t .
33. (ACR160) Diseñar un algoritmo que dada una matriz M de dimensión $n \times n$ determine si es una matriz triangular inferior, es decir, si todos los elementos bajo la diagonal principal son nulos.
34. Dada una matriz M de dimensión $m \times n$ de números enteros o reales, diseñar algoritmos que resuelvan los siguientes problemas:
- Encontrar el menor elemento de la matriz.
 - Devolver en un vector de tamaño m los mínimos de cada fila de la matriz, es decir en la posición i del vector ($0 \leq i < m$) debe guardarse el menor elemento de la fila i de la matriz.
 - Devolver un vector de tamaño $m + 1$ que en la posición i ($0 \leq i \leq m$) guarde la suma de los mínimos de las filas i hasta $m - 1$ (incluidas) de la matriz
35. Diseñar un algoritmo que dada una matriz M de dimensión $m \times n$ de números enteros o reales, sume los elementos de las posiciones pares, siendo estas aquellas en las que la suma de la fila y la columna sea un número par.

36. Otros problemas sobre vectores y matrices:

a) Búsqueda de un valor en un vector:

<https://www.aceptaelreto.com/problem/statement.php?id=168>.

b) Matriz identidad:

<https://www.aceptaelreto.com/problem/statement.php?id=151>.

c) Análisis de un tablero de sudoku:

<https://www.aceptaelreto.com/problem/statement.php?id=196>.

d) Cuadrados mágicos:

<https://www.aceptaelreto.com/problem/statement.php?id=101>.

Diseño de algoritmos recursivos¹

*Para entender la recursividad
primero hay que entender la recursividad*

Anónimo

RESUMEN: En este tema se presenta el mecanismo de la recursividad como una manera de diseñar algoritmos fiables y fáciles de entender. Se introducen distintas técnicas para diseñar algoritmos recursivos, analizar su complejidad, modificarlos para mejorar esta complejidad y verificar su corrección.

1. Introducción

- ★ La recursión aparece de forma natural en programación, tanto en la definición de tipos de datos (como veremos en temas posteriores) como en el diseño de algoritmos.
- ★ Hay lenguajes (funcionales y lógicos) en los que no hay iteración (no hay bucles), sólo hay recursión. En C++ tenemos la opción de elegir entre iteración y recursión.
- ★ Optamos por una solución recursiva cuando sabemos cómo resolver de manera directa un problema para un cierto conjunto de datos y para el resto de los datos somos capaces de resolverlo utilizando la solución al mismo problema con unos datos “más simples”.
- ★ Cualquier solución recursiva se basa en un análisis (clasificación) de los datos, \vec{x} , para distinguir los casos de solución directa y los casos de solución recursiva:
 - caso(s) directo(s): \vec{x} es tal que el resultado \vec{y} puede calcularse directamente de forma sencilla.
 - caso(s) recursivo(s): sabemos cómo calcular a partir de \vec{x} otros datos más pequeños \vec{x}' , y sabemos además cómo calcular el resultado \vec{y} para \vec{x} suponiendo conocido el resultado \vec{y}' para \vec{x}' .

¹Gonzalo Méndez y Clara Segura son los autores principales de este tema, que incluye material elaborado por Jaime Sánchez.

- ★ Para implementar soluciones recursivas en un lenguaje de programación tenemos que utilizar una acción que se invoque a sí misma con datos cada vez “más simples”: funciones o procedimientos.
- ★ Intuitivamente, el subprograma se invoca a sí mismo con datos cada vez más simples hasta que son tan simples que la solución es inmediata. Posteriormente, la solución se puede ir elaborando hasta obtener la solución para los datos iniciales.
- ★ Tres elementos básicos en la recursión:
 - Autoinvocación: el proceso se llama a sí mismo.
 - Caso directo: el proceso de autoinvocación eventualmente alcanza una solución directa (sin invocarse a sí mismo) al problema.
 - Combinación: los resultados de la llamada precedente son utilizados para obtener una solución, posiblemente combinados con otros datos.
- ★ Para entender la recursividad, a veces resulta útil considerar cómo se ejecutan las acciones recursivas en una computadora, como si se crearan múltiples copias del mismo código, operando sobre datos diferentes (en realidad sólo se copian las variables locales y los parámetros por valor). El ejemplo clásico del factorial:

```
{n ≥ 0}
fun factorial (int n) return int r
{r = n!}
```

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

¿Cómo se ejecuta la llamada *factorial(3)*?

¿Y qué ocurre en memoria?

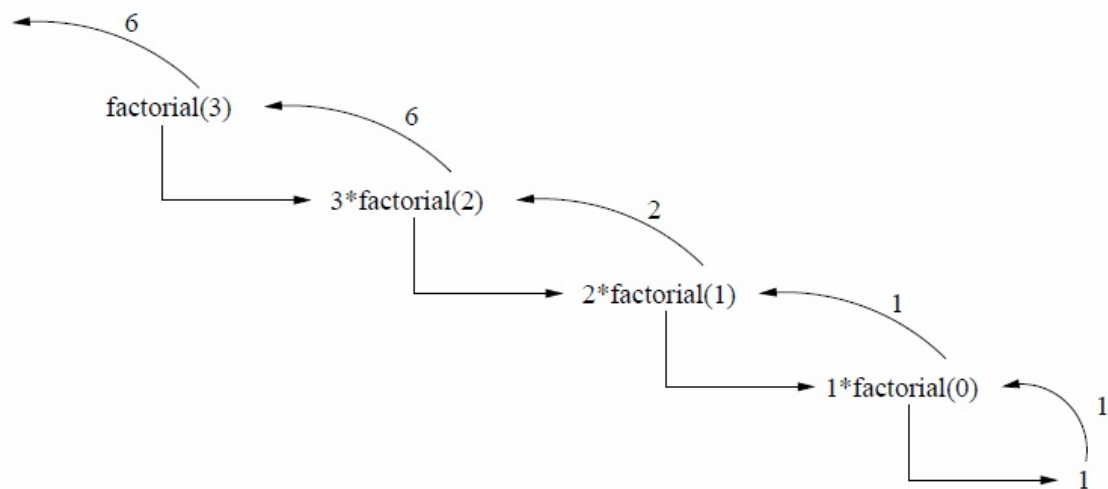
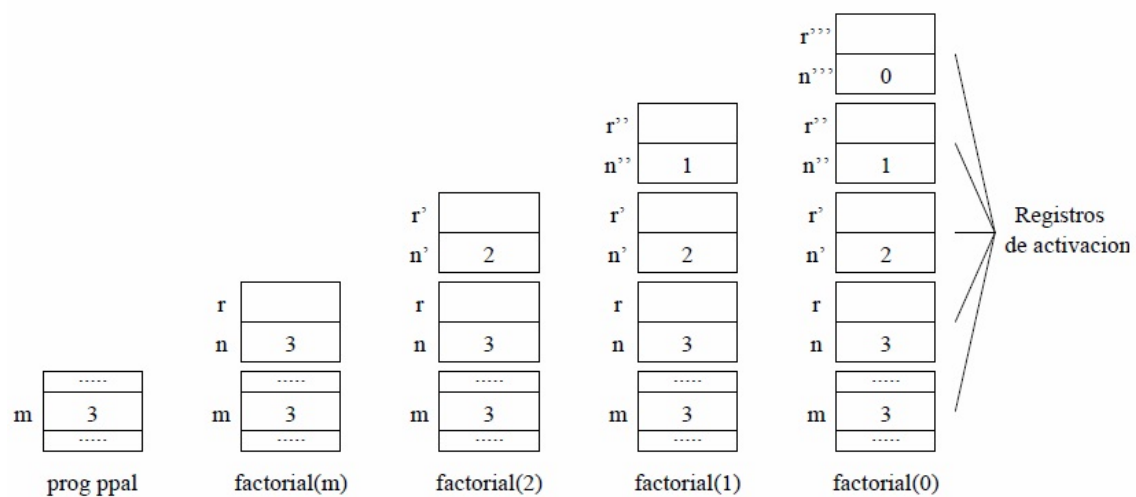
- ★ ¿La recursión es importante?
 - Un método muy potente de diseño y razonamiento formal.
 - Tiene una relación natural con la inducción y, por ello, facilita conceptualmente la resolución de problemas y el diseño de algoritmos.

1.1. Recursión simple

- ★ Una acción recursiva tiene recursión simple (o lineal) si cada caso recursivo realiza exactamente una llamada recursiva. Puede describirse mediante el esquema general:

```
void nombreProc (  $\tau_1 x_1$  , ... ,  $\tau_n x_n$  ,  $\delta_1$  &  $y_1$  , ... ,  $\delta_m$  &  $y_m$  ) {
// P: Precondición
// declaración de constantes

     $\tau_1 x'_1$  ; ... ;  $\tau_n x'_n$  ; //  $\vec{x}'$ , parámetros de la llamada recursiva
     $\delta_1 y'_1$  ; ... ;  $\delta_m y'_m$  ; //  $\vec{y}'$ , resultados de la llamada recursiva
```

Figura 1: Ejecución de *factorial(3)*Figura 2: Llamadas recursivas de *factorial(3)*

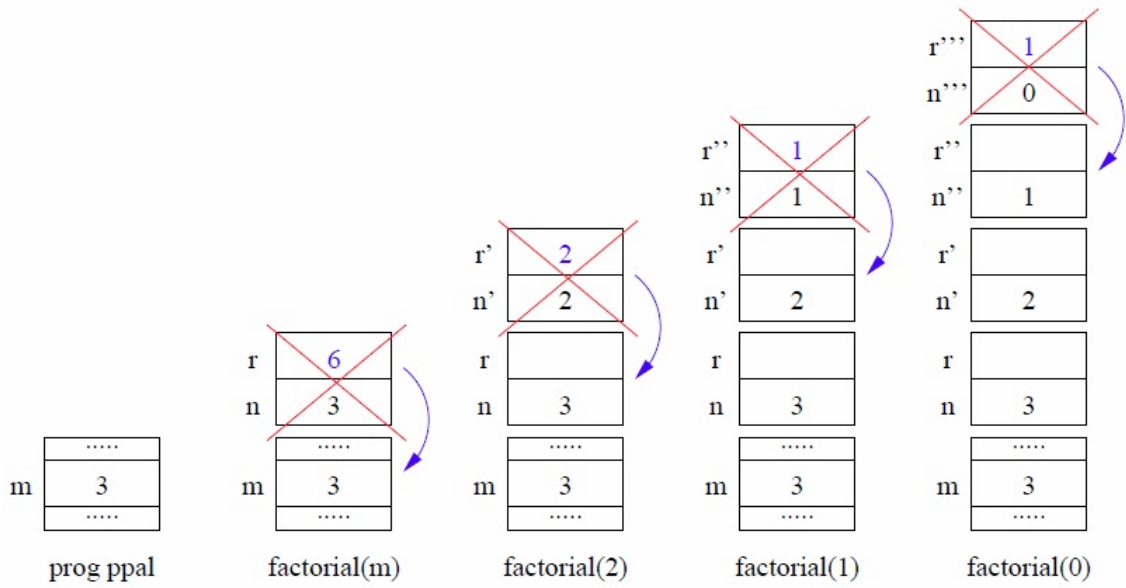
```

if (  $d(\vec{x})$  ) // caso directo
   $\vec{y} = g(\vec{x})$ ; // solución al caso directo
else if (  $r(\vec{x})$  ) { // caso no directo
   $\vec{x}' = s(\vec{x})$ ; // función sucesor: descomposición de datos
  nombreProc( $\vec{x}', \vec{y}'$ ); // llamada recursiva
   $\vec{y} = c(\vec{x}, \vec{y}')$ ; // función de combinación: composición de solución
}

// Q: Postcondición
}

```

donde:

Figura 3: Vuelta de las llamadas recursivas de *factorial(3)*

- \vec{x} representa a los parámetros de entrada x_1, \dots, x_n , \vec{x}' a los parámetros de la llamada recursiva x'_1, \dots, x'_n , \vec{y}' a los resultados de la llamada recursiva y'_1, \dots, y'_m , e \vec{y} a los parámetros de salida y_1, \dots, y_m
- $d(\vec{x})$ es la condición que determina el caso directo
- $r(\vec{x})$ es la condición que determina el caso recursivo
- g calcula el resultado en el caso directo
- s , la *función sucesor*, calcula los argumentos para la siguiente llamada recursiva
- c , la *función de combinación*, obtiene la combinación de los resultados de la llamada recursiva \vec{y}' junto con los datos de entrada \vec{x} , proporcionando así el resultado \vec{y} .

★ Veamos cómo la función factorial se ajusta a este esquema de declaración:

```

{ $n \geq 0$ }
fun factorial (int n) return int r
{ $r = n!$ }

int factorial ( int n ) {
    int r;

    if ( n == 0 ) r = 1;
    else // n > 0
        r = n * factorial(n-1);
    return r;
}

```

```

d(n) = (n == 0)
g(n) = 1
r(n) = (n > 0)

```

$$s(n) = n - 1$$

$$c(n, fact(s(n))) = n * fact(s(n))$$

- ★ El esquema de recursión simple puede ampliarse considerando varios casos directos y también varias descomposiciones para el caso recursivo. $d(\vec{x})$ y $r(\vec{x})$ pueden desdoblarse en una alternativa con varios casos. Lo importante es que las alternativas sean **exhaustivas** y **excluyentes**, y que en cada caso sólo se ejecute una llamada recursiva.
- ★ Ejemplo: multiplicación de dos naturales por el método del *campesino egipcio*.

```

{(a ≥ 0) ∧ (b ≥ 0)}
fun prod (int a, int b) return int r
{r = a * b}

int prod ( int a, int b ) {
    int r;

    if ( b == 0 ) {
        r = 0;
    } else if ( b == 1 ) {
        r = a;
    } else if (b % 2 == 0) {           // b > 1
        r = prod(2*a, b/2);
    } else {                          // b > 1 && (b % 2 == 1)
        r = prod(2*a, b/2) + a;
    }
    return r;
}

```

$$\begin{array}{ll}
 d_1(a, b) = (b == 0) & d_2(a, b) = (b == 1) \\
 g_1(a, b) = 0 & g_2(a, b) = 1 \\
 r_1(a, b) = ((b > 1) \ \&\& \ par(b)) & r_2(a, b) = ((b > 1) \ \&\& \ impar(b)) \\
 s_1(a, b) = (2 * a, b/2) & s_2(a, b) = (2 * a, b/2) \\
 c_1(a, b, prod(s_1(a, b))) = prod(s_1(a, b)) & c_2(a, b, prod(s_2(a, b))) = prod(s_2(a, b)) + a
 \end{array}$$

1.2. Recursión final

- ★ La recursión final o de cola (*tail recursion*) es un caso particular de recursión simple donde la función de combinación se limita a transmitir el resultado de la llamada recursiva. Se llama final porque lo último que se hace en cada pasada es la llamada recursiva.
- ★ El resultado será siempre el obtenido en uno de los casos base.
- ★ Los algoritmos recursivos finales tienen la interesante propiedad de que pueden ser traducidos de manera directa a soluciones iterativas, más eficientes.

```

void nombreProcItr (  $\tau_1 x_1$  , ... ,  $\tau_n x_n$  ,  $\delta_1$  &  $y_1$  , ... ,  $\delta_m$  &  $y_m$  ) {
// Precondición
// declaración de constantes
     $\tau_1 x'_1$  ; ... ;  $\tau_n x'_n$  ; //  $\vec{x}'$ 

     $\vec{x}' = \vec{x}$ ;
    while ( r( $\vec{x}'$ ) )
         $\vec{x}' = s(\vec{x}')$ ;
         $\vec{y} = g(\vec{x}')$ ;
    }
// Postcondición
}

```

- ★ Como ejemplo de función recursiva final veamos el algoritmo de cálculo del máximo común divisor por el algoritmo de Euclides.

```

{( $a > 0$ ) & ( $b > 0$ )}
fun mcd (int a, int b) return int r
{ $r = mcd(a, b)$ }

int mcd( int a, int b ) {
    int m;

    if      ( a == b ) m = a;
    else if ( a >  b ) m = mcd(a-b, b);
    else    // a <  b
        m = mcd(a, b-a);
    return m;
}

```

que se ajusta al esquema de recursión simple:

$$\begin{aligned}
 d(a, b) &= (a == b) & g(a, b) &= a \\
 r_1(a, b) &= (a > b) & r_2(a, b) &= (a < b) \\
 s_1(a, b) &= (a - b, a) & s_2(a, b) &= (a, b - a)
 \end{aligned}$$

y donde las funciones de combinación se limitan a devolver el resultado de la llamada recursiva

$$\begin{aligned}
 c_1(a, b, mcd(s_1(a, b))) &= mcd(s_1(a, b)) \\
 c_2(a, b, mcd(s_2(a, b))) &= mcd(s_2(a, b))
 \end{aligned}$$

Si traducimos esta versión recursiva a una iterativa del algoritmo obtenemos:

```

int itmcd( int a, int b ) {
    int auxa = a; int auxb = b;
    while (auxa != auxb)
        if (auxa > auxb)
            auxa = auxa - auxb;
        else
            auxb = auxb - auxa;
    return auxa;
}

```

1.3. Recursión múltiple

- ★ Este tipo de recursión se caracteriza por que, al menos en un caso recursivo, se realizan varias llamadas recursivas. El esquema correspondiente es el siguiente:

```

void nombreProc (  $\tau_1 x_1$  , ... ,  $\tau_n x_n$  ,  $\delta_1$  &  $y_1$  , ... ,  $\delta_m$  &  $y_m$  ) {
// Precondición
// declaración de constantes
 $\tau_1 x_{11}$  ; ... ;  $\tau_n x_{1n}$  ; ... ;  $\tau_1 x_{k1}$  ; ... ;  $\tau_n x_{kn}$  ;
 $\delta_1 y_{11}$  ; ... ;  $\delta_m y_{1m}$  ; ... ;  $\delta_1 y_{k1}$  ; ... ;  $\delta_m y_{km}$  ;

if (  $d(\vec{x})$  )
     $\vec{y} = g(\vec{x})$ ;
else if (  $r(\vec{x})$  ) {
     $\vec{x}_1 = s_1(\vec{x})$ ;
    nombreProc (  $\vec{x}_1$ ,  $\vec{y}_1$  );
    ...
     $\vec{x}_k = s_k(\vec{x})$ ;
    nombreProc (  $\vec{x}_k$ ,  $\vec{y}_k$  );
     $\vec{y} = c(\vec{x}, \vec{y}_1, \dots, \vec{y}_k)$ ;
}
// Postcondición
}

```

donde

- $k > 1$, indica el número de llamadas recursivas
- \vec{x} representa los parámetros de entrada x_1, \dots, x_n , \vec{x}_i a los parámetros de la i -ésima llamada recursiva x_{i1}, \dots, x_{in} , \vec{y}_i a los resultados de la i -ésima llamada recursiva y_{i1}, \dots, y_{im} , para $i = 1, \dots, k$, e \vec{y} a los parámetros de salida y_1, \dots, y_m
- $d(\vec{x})$ es la condición que determina el caso directo
- $r(\vec{x})$ es la condición que determina el caso recursivo
- g calcula el resultado en el caso directo
- s_i , las *funciones sucesor*, calculan la descomposición de los datos de entrada para realizar la i -ésima llamada recursiva, para $i = 1, \dots, k$
- c , la *función de combinación*, obtiene la combinación de los resultados \vec{y}_i de las llamadas recursivas, para $i = 1, \dots, k$, junto con los datos de entrada \vec{x} , proporcionando así el resultado \vec{y} .

- ★ Otro ejemplo clásico, los números de Fibonacci:

```

{n ≥ 0}
fun fib ( int n ) return int r
{r = fib(n)}

int fib( int n ) {
    int r;

    if      ( n == 0 ) r = 0;
    else if ( n == 1 ) r = 1;
    else    // n > 1

```

```

    r = fib(n-1) + fib(n-2);

    return r;
}

```

que se ajusta al esquema de recursión múltiple ($k = 2$)

$$\begin{aligned}
 d_1(n) &= (n == 0) & d_2(n) &= (n == 1) \\
 g(0) &== 0 & g(1) &== 1 \\
 r(n) &= (n > 1) \\
 s_1(n) &= n - 1 & s_2(n) &= n - 2 \\
 c(n, fib(s_1(n)), fib(s_2(n))) &= fib(s_1(n)) + fib(s_2(n))
 \end{aligned}$$

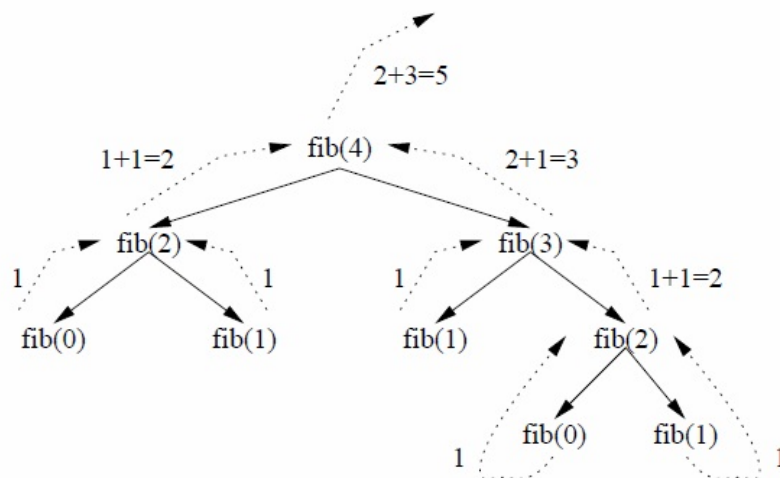


Figura 4: Ejecución de $fib(4)$

- ★ En la recursión múltiple, el número de llamadas puede crecer muy rápidamente, como se puede ver en el cómputo de $fib(4)$ que se muestra en la Figura 4. Nótese que algunos valores se computan más de una vez (p.e. $fib(2)$ se evalúa 2 veces).

1.4. Resumen de los distintos tipos de recursión

- ★ Para terminar con la introducción, recopilamos los distintos tipos de funciones recursivas que hemos presentado:
 - Simple. Una llamada recursiva en cada caso recursivo:
 - No final. Requiere combinación de resultados
 - Final. No requiere combinación de resultados
 - Múltiple. Más de una llamada recursiva en algún caso recursivo.

2. Diseño de algoritmos recursivos

- ★ Dada la especificación $\{P_0\}A\{Q_0\}$, hemos de obtener una acción A que la satisfaga.

- ★ Nos planteamos implementar A como una función o un procedimiento recursivo cuando podemos obtener -fácilmente- una definición recursiva de la postcondición.
- ★ Se propone un método que descompone la obtención del algoritmo recursivo en varios pasos.

(R.1) **Planteamiento recursivo.** Se ha de encontrar una estrategia recursiva para alcanzar la postcondición, es decir, la solución. A veces, la forma de la postcondición, o de las operaciones que en ella aparecen, nos sugerirá directamente una estrategia recursiva.

Se trata, por tanto, de describir de manera poco formal la estrategia a seguir para resolver el problema planteado.

(R.2) **Análisis de casos.** Se trata de identificar y obtener las condiciones que permiten discriminar los casos directos de los recursivos. Deben tratarse de forma exhaustiva y mutuamente excluyente todos los casos contemplados en la precondition:

$$P_0 \Rightarrow d(\vec{x}) \vee r(\vec{x})$$

(R.3) **Caso directo.** Hemos de encontrar la acción A_1 que resuelve el caso directo:

$$\{P_0 \wedge d(\vec{x})\} A_1 \{Q_0\}$$

Si hubiese más de un caso directo, repetiríamos este paso para cada uno de ellos.

(R.4) **Descomposición recursiva.** Se trata de obtener la función sucesor $s(\vec{x})$ que nos proporciona los datos que empleamos para realizar la llamada recursiva.

Si hay más de un caso recursivo, obtenemos la función sucesor para cada uno de ellos.

(R.5) **Función de acotación y terminación.** Determinamos si la función sucesor escogida garantiza la terminación de las llamadas, obteniendo una función que estime el número de llamadas restantes hasta alcanzar un caso base -la *función de acotación*- y justificando que se decrementa en cada llamada.

Si hay más de un caso recursivo, se ha de garantizar la terminación para cada uno de ellos.

(R.6) **Llamada recursiva.** Pasamos a ocuparnos entonces del caso recursivo. Cada una de las descomposiciones recursivas ha de permitir realizar la(s) llamada(s) recursiva(s), es decir, la función sucesor debe proporcionar unos datos que cumplan la precondition de la función recursiva para estar seguros de que se realiza la llamada recursiva con datos válidos.

$$P_0 \wedge r(\vec{x}) \Rightarrow P_0[\vec{x}/s(\vec{x})]$$

(R.7) **Función de combinación.** Lo único que nos resta por obtener del caso recursivo es la función de combinación, que, en el caso de la recursión simple, será de la forma $\vec{y} = c(\vec{x}, \vec{y}')$.

Si hubiese más de un caso recursivo, habría que encontrar una función de combinación para cada uno de ellos.

(R.8) **Escritura del caso recursivo.** Lo último que nos queda por decidir es si necesitamos utilizar en el caso recursivo todas las variables auxiliares que han ido apareciendo. Partiendo del esquema más general posible

```

{  $P_0 \wedge r(\vec{x})$  }
   $\vec{x}' = s(\vec{x})$ ;
  nombreProc( $\vec{x}'$ ,  $\vec{y}'$ );
   $\vec{y} = c(\vec{x}, \vec{y}')$ 
{  $Q_0$  }

```

llegamos a aquel donde el caso recursivo se reduce a una única sentencia

```

{  $P_0 \wedge r(\vec{x})$  }
   $\vec{y} = c(\vec{x}, \text{nombreFunc}(s(\vec{x})))$ 
{  $Q_0$  }

```

Repetimos este proceso para cada caso recursivo, si es que tenemos más de uno, y lo generalizamos de la forma obvia cuando tenemos recursión múltiple.

Veamos un 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. Vamos a diseñar una función que nos diga si un vector de naturales es o no pareado.

Especificación

```

{longitud(v) ≥ num ∧ ∀k : 0 ≤ k < num : v[k] ≥ 0}
fun pareado (int v[], int num) return bool b
{b = esPareado(v, 0, num)}

```

donde el predicado *esPareado* se define según el enunciado de la siguiente manera para $0 \leq i \leq j \leq \text{num}$:

$$\begin{aligned}
\text{esPareado}(v, i, j) \equiv & \\
& (j \leq i + 1) \vee \\
& (j > i + 1 \wedge |\text{numPares}(v, i, (i + j)/2) - \text{numPares}(v, (i + j)/2, j)| \leq 1 \\
& \wedge \text{esPareado}(v, i, (i + j)/2) \wedge \text{esPareado}(v, (i + j)/2, j))
\end{aligned}$$

siendo

$$\text{numPares}(v, c, f) \equiv \#z : c \leq z < f : v[z] \bmod 2 = 0$$

(R.1) Planteamiento recursivo

(R.2) y análisis de casos. El planteamiento recursivo parece directo, ya que para saber si un vector es pareado hemos de comprobar, entre otras cosas, que sus dos mitades cumplen a su vez la misma propiedad. La cuestión es cómo hacer referencia en la llamada recursiva a una mitad del vector. No basta con escribir *pareado*(*v*, *num*/2), ya que hay dos mitades, la de la izquierda y la de la derecha. Adicionalmente, cada una de ellas tiene a su vez dos mitades que habrá que comprobar que cumplen la propiedad y así sucesivamente, lo cual quiere decir que necesitamos saber cuándo un segmento válido cualquiera del vector es pareado.

Para ello vamos a utilizar una función auxiliar que nos permita hacer el planteamiento recursivo descrito anteriormente:

```

{ $P_0 \equiv 0 \leq i \leq j \leq \text{longitud}(v) \wedge \forall k : 0 \leq i \leq k < j : v[k] \geq 0$ }
fun pareado (int v[], int i, int j) return bool b
{b = esPareado(v, i, j)}

```

Se dice que esta función es una *generalización* de la función a desarrollar porque tiene más parámetros que la función que se desea definir, y además la función original se puede calcular como un caso particular de la auxiliar:

$$\text{pareado}(v, \text{num}) = \text{pareado}(v, 0, \text{num})$$

Para calcular lo que se desea por tanto basta con una llamada a $\text{pareado}(v, 0, \text{num})$. Esta llamada recibe el nombre de *llamada inicial*.

La generalización es una técnica muy utilizada en el diseño de algoritmos recursivos que consiste en añadir parámetros adicionales y/o resultados adicionales que ayudan a diseñar la función o a hacerla más eficiente.

Nótese que no es necesario inventarse otro nombre para la función auxiliar porque C++ permite la sobrecarga de funciones: definir varias funciones con el mismo nombre que se distinguen por los parámetros.

Vamos por tanto a diseñar recursivamente la función auxiliar. El problema se resuelve trivialmente cuando el segmento es vacío o unitario, es decir, cuando $j \leq i + 1$, ya que en tal caso el segmento se considera pareado.

$$\begin{aligned} d(v, i, j) &: j \leq i + 1 \\ r(v, i, j) &: j > i + 1 \end{aligned}$$

Claramente, estos dos casos cubren los admitidos por la precondition.

El planteamiento recursivo consiste en comprobar que las dos mitades son pareadas: $\text{pareado}(v, i, (i+j)/2)$ y $\text{pareado}(v, (i+j)/2, j)$, y hacer una comprobación adicional que consiste en contar el número de pares de cada una de esas mitades y comprobar que la diferencia en valor absoluto no supera a 1.

(R.3) Solución en el caso directo.

$$\begin{aligned} &\{ P_0 \wedge j \leq i + 1 \} \\ &\quad A_1 \\ &\{ \text{esPareado}(v, i, j) \} \end{aligned}$$

Claramente, por definición de *esPareado*: $A_1 \equiv b = \text{true}$;

(R.4) Descomposición recursiva.

$$\begin{aligned} s_1(v, i, j) &= (v, i, (i+j)/2) \\ s_2(v, i, j) &= (v, (i+j)/2, j) \end{aligned}$$

(R.5) Función de acotación y terminación. El valor que disminuye hasta llegar al caso base es

$$t(i, j) = j - i$$

Efectivamente, se decrementa en cada una de las dos llamadas recursivas:

$$\begin{aligned} t(s_1(i, j)) &= (i+j)/2 - i \\ t(s_2(i, j)) &= j - (i+j)/2 \end{aligned}$$

ya que si $j > i + 1$ entonces $i < (i+j)/2 < j$ y por tanto $(i+j)/2 - i < j - i$ y también $j - (i+j)/2 < j - i$.

- (R.6) Es posible hacer las dos llamadas recursivas, es decir, los argumentos de las llamadas recursivas cumplen la precondition

$$\begin{aligned} P_0(v, i, j) \wedge r(i, j) &\Rightarrow P_0(v, i, (i+j)/2) \\ P_0(v, i, j) \wedge r(i, j) &\Rightarrow P_0(v, (i+j)/2, j) \end{aligned}$$

Puesto que el vector no se modifica, la parte que corresponde al hecho de que el vector es de naturales se cumple trivialmente, por lo que nos centramos en los índices:

$$\begin{aligned} 0 \leq i \leq j \leq \text{longitud}(v) &\Rightarrow 0 \leq i \leq (i+j)/2 \leq \text{longitud}(v) \\ 0 \leq i \leq j \leq \text{longitud}(v) &\Rightarrow 0 \leq (i+j)/2 \leq j \leq \text{longitud}(v) \end{aligned}$$

- (R.7) Función de combinación y escritura del caso recursivo. Como hemos mencionado previamente hemos de comprobar que ambas mitades son pareadas, para lo cual haremos dos llamadas recursivas. Llamemos m a la posición central $(i+j)/2$. Además de las llamadas recursivas hemos de contar el número de pares en cada una de las dos mitades, para lo que vamos a utilizar una función iterativa, que diseñaremos después y cuya especificación es:

```
{0 ≤ i ≤ j ≤ longitud(v) ∧ ∀k : 0 ≤ i ≤ k < j : v[k] ≥ 0}
fun contarPares (int v[], int i, int j) return int r
{r = #l : i ≤ l < j : v[l] mod 2 = 0}
```

Utilizando dicha función, el caso recursivo será:

```
int m = (i+j)/2;
b = (abs(contarPares(v,i,m)-contarPares(v,m,j)) <= 1)
    && pareado(v,i,m) && pareado(v,m,j);
```

Finalmente, el algoritmo queda:

```
bool pareado (int v[], int i, int j) {
// Pre: 0 ≤ i ≤ j ≤ longitud(v) ∧ ∀k : 0 ≤ i ≤ k < j : v[k] ≥ 0
  bool b;
  if (j <= i+1) b = true;
  else
    b = (abs(contarPares(v,i,m)-contarPares(v,m,j)) <= 1)
        && pareado(v,i,m) && pareado(v,m,j);
  return b;
// Post: b = esPareado(v,i,j)
}
```

siendo la llamada inicial

```
bool pareado (int v[], int num) {
// Pre: longitud(v) ≥ num ∧ ∀k : 0 ≤ k < num : v[k] ≥ 0
  return pareado(v, 0, num);
// Post: b = esPareado(v, 0, num)
}
```

En secciones posteriores de este tema veremos que el coste de este algoritmo está en $O(\text{num} \log(\text{num}))$ y cómo modificar el diseño de la función para obtener una versión más eficiente con coste en $O(\text{num})$.

2.1. 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. Es decir, estamos buscando el punto del vector donde las componentes pasan de ser $\leq x$ a ser $> x$.
- ★ La idea es que en cada pasada por el bucle se reduce a la mitad el tamaño del subvector donde puede estar el elemento buscado.

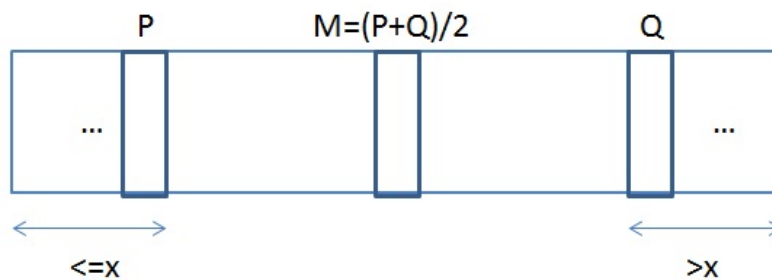


Figura 5: Cálculo del punto medio

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

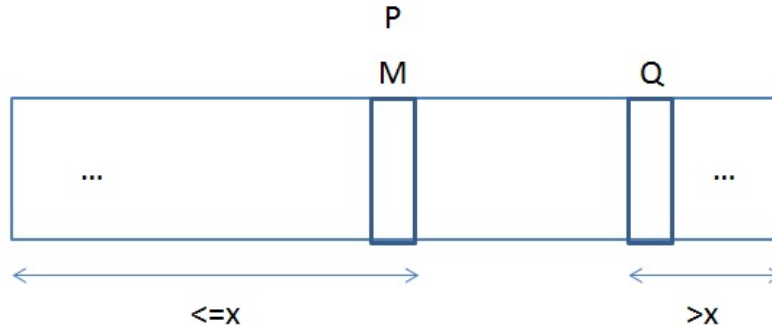


Figura 6: Búsqueda en la mitad derecha

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

- ★ Como el tamaño de los datos se reduce a la mitad en cada pasada tenemos claramente una complejidad logarítmica en el caso peor (en realidad en todos los casos, pues aunque encontremos x en el vector hemos de seguir buscando ya que puede que no sea la aparición más a la derecha).
- ★ Hay que ser cuidadoso con los índices, sobre todo si:
 - x no está en el vector, o si, en particular,
 - x es mayor o menor que todos los elementos del vector; además, es necesario pensar con cuidado cuál es el caso base.

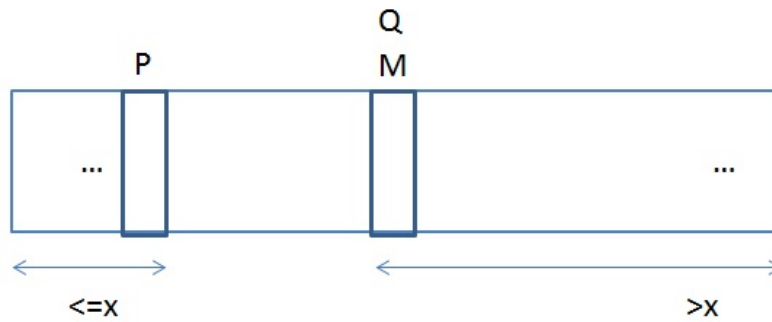


Figura 7: Búsqueda en la mitad izquierda

$$\{0 \leq \text{num} \leq \text{longitud}(v) \wedge \text{ord}(v, \text{num})\}$$

```

fun buscaBin (TElem v[], int num, TElem x) return int pos
{
   $(\exists u : 0 \leq u < \text{num} : v[u] \leq x \wedge \text{pos} = \text{máx } k : (0 \leq k \leq \text{num} - 1) \wedge v[k] \leq x : k)$ 
   $\vee (\forall z : 0 \leq z < \text{num} : x < v[z] \wedge \text{pos} = -1)$ 
}

```

```
typedef int TElem;
```

```

int buscaBin( TElem v[], int num, TElem x ) {
  int pos;

  // cuerpo de la función

  return pos;
}

```

Comentarios:

- Utilizamos el tipo TElem para resaltar la idea de que la búsqueda binaria es aplicable sobre cualquier tipo que tenga definido un orden, es decir, los operadores $=$ y \leq .
 - Si x no está en v devolvemos la posición anterior al lugar donde debería estar. En particular, si x es menor que todos los elementos de v , el lugar a insertarlo será la posición 0 y, por lo tanto, devolvemos -1.
- ★ El planteamiento recursivo parece claro: para buscar x en un vector de n elementos tenemos que comparar x con el elemento central y
- si x es mayor o igual que el elemento central, seguimos buscando recursivamente en la mitad derecha,
 - si x es menor que el elemento central, seguimos buscando recursivamente en la mitad izquierda.
- ★ De forma similar al ejemplo anterior, para comprobar si un vector es pareado, utilizaremos una función -o un procedimiento- auxiliar que nos permita implementar el planteamiento recursivo.

En el caso de la búsqueda binaria, se trata de una función que en lugar de recibir el número de elementos del vector, reciba dos índices, a y b , que señalen dónde empieza y dónde acaba el fragmento de vector a considerar (ambos incluidos en este caso).

```
int buscaBin( TElem v[], TElem x, int a, int b)
```

De esta forma, la función que realmente nos interesa se obtiene como

$$\text{buscaBin}(v, x) = \text{buscaBin}(v, x, 0, \text{longitud}(v) - 1)$$

- ★ La función recursiva es, por tanto, la función auxiliar:

```
int buscaBin( TElem v[], TElem x, int a, int b) {  
  
    // cuerpo de la función  
  
}
```

y para buscar un elemento en el vector podemos llamar a `buscaBin(v, x, 0, l-1)`, siendo l la longitud del vector v .

- ★ Diseño del algoritmo:

(R.1) Planteamiento recursivo

(R.2) y análisis de casos.

Aunque el planteamiento recursivo está claro: dados a y b , obtenemos el punto medio m y

- si $v[m] \leq x$ seguimos buscando en $m+1..b$
- si $v[m] > x$ seguimos buscando en $a..m-1$,

Es necesario ser cuidadoso con los índices. La idea consiste en garantizar que en todo momento se cumple que:

- todos los elementos a la izquierda de a -sin incluir $v[a]$ - son menores o iguales que x , y
- todos los elementos a la derecha de b -sin incluir $v[b]$ - son estrictamente mayores que x .

Una primera idea puede ser considerar como caso base $a = b$. Si lo hiciésemos así, la solución en el caso base quedaría:

```
if ( a == b )  
    if      ( v[a] == x ) p = a;  
    else if ( v[a] < x ) p = a;    // x no está en v  
    else p = a-1;  // x no está en v y v[a] > x
```

Sin embargo, también es necesario considerar el caso base $a = b+1$ pues puede ocurrir que en ninguna llamada recursiva se cumpla $a = b$. Por ejemplo, en un situación como esta

$$x = 8 \quad a = 0 \quad b = 1 \quad v[0] = 10 \quad v[1] = 15$$

el punto medio $m = (a+b)/2$ es 0, para el cual se cumple $v[m] > x$ y por lo tanto la siguiente llamada recursiva se hace con

$$a = 0 \quad b = -1$$

que es un caso base donde debemos devolver -1 y donde para alcanzarlo no hemos pasado por $a = b$.

Como veremos a continuación, el caso $a = b$ se puede incluir dentro del caso recursivo si consideramos como caso base el que cumple $a = b+1$, que además tiene una solución más sencilla y que siempre se alcanza.

Con todo lo anterior, la especificación de la función recursiva auxiliar queda:

```
{ord(v, longitud(v))
 $\wedge (0 \leq a \leq longitud(v)) \wedge (-1 \leq b \leq longitud(v) - 1) \wedge (a \leq b + 1)$ 
 $\wedge (\forall i : 0 \leq i < a : v[i] \leq x) \wedge (\forall j : b < j < longitud(v) : v[j] > x)$ }
fun buscaBin (TElem v[], TElem x, int a, int b) return int pos
{ $(\exists u : 0 \leq u < longitud(v) : v[u] \leq x \wedge pos = \text{máx } k : (0 \leq k \leq longitud(v) - 1) \wedge v[k] \leq x : k)$ 
 $\vee (\forall z : 0 \leq z < longitud(v) : x < v[z] \wedge pos = -1)$ }
```

```
typedef int TElem;
```

```
int buscaBin( TElem v[], TElem x, int a, int b ) {
    int p;

    // cuerpo de la función

    return p;
}
```

Donde se tiene la siguiente distinción de casos

$$d(v, x, a, b) : a = b + 1$$

$$r(v, x, a, b) : a \leq b$$

para la que efectivamente se cumple

$$P_0 \Rightarrow a \leq b + 1 \Rightarrow a = b + 1 \vee a \leq b$$

(R.3) Solución en el caso directo.

```
{  $P_0 \wedge a = b + 1$  }
 $A_1$ 
{  $Q_0$  }
```

Si

- todos los elementos a la izquierda de a son $\leq x$,
- todos los elementos a la derecha de b son $> x$, y
- $a = b + 1$, es decir, a y b se han cruzado,

entonces el último elemento que cumple que es $\leq x$ es $v[a - 1]$, y por lo tanto,

$$A_1 \equiv p = a - 1;$$

(R.4) Descomposición recursiva.

Los parámetros de la llamada recursiva dependerán del resultado de comparar x con la componente central del fragmento de vector que va desde a hasta b . Por lo tanto, obtenemos el punto medio

$$m = (a + b)/2;$$

de forma que

- si $x < v[m]$ la descomposición es

$$s_1(v, x, a, b) = (v, x, a, m - 1)$$

- si $x \geq v[m]$ la descomposición es

$$s_2(v, x, a, b) = (v, x, m + 1, b)$$

(R.5) Función de acotación y terminación.

Lo que va a ir disminuyendo, según avanza la recursión, es la longitud del subvector a considerar, por lo que tomamos como función de acotación:

$$t(v, x, a, b) = b - a + 1$$

y comprobamos

$$a \leq b \wedge a \leq m \leq b \Rightarrow t(s_1(\vec{x})) < t(\vec{x}) \wedge t(s_2(\vec{x})) < t(\vec{x})$$

que efectivamente se cumple, ya que

$$a \leq b \Rightarrow a \leq (a + b)/2 \leq b$$

$$(m - 1) - a + 1 < b - a + 1 \Leftarrow m - 1 < b \Leftarrow m \leq b$$

$$b - (m + 1) + 1 < b - a + 1 \Leftarrow b - m - 1 < b - a \Leftarrow b - m \leq b - a \Leftarrow m \geq a$$

(R.6) Llamada recursiva.

La solución que hemos obtenido sólo funciona si en cada llamada se cumple la precondition. Por lo tanto, debemos demostrar que de una llamada a la siguiente efectivamente se sigue cumpliendo la precondition.

Tratamos por separado cada caso recursivo

- $x < v[m]$
 $P_0 \wedge a \leq b \wedge a \leq m \leq b \wedge x < v[m] \Rightarrow P_0[b/m - 1]$

Que es cierto porque:

$$v \text{ ordenado entre } 0..longitud(v) - 1 \Rightarrow v \text{ ordenado entre } 0..longitud(v) - 1$$

$$0 \leq a \leq longitud(v) \Rightarrow 0 \leq a \leq longitud(v)$$

$$-1 \leq b \leq longitud(v) - 1 \wedge a \leq m \leq b \wedge 0 \leq a \leq longitud(v) \Rightarrow -1 \leq m - 1 \leq longitud(v) - 1$$

$$a \leq m \Rightarrow a \leq m - 1 + 1$$

todos los elementos a la izquierda de a son $\leq x \Rightarrow$ todos los elementos a la izquierda de a son $\leq x$

v está ordenado entre $0..longitud(v) - 1 \wedge$ todos los elementos a la derecha de b son $> x \wedge m \leq b \wedge x < v[m] \Rightarrow$ todos los elementos a la derecha de $m - 1$ son $> x$

- $x \geq v[m]$
 v está ordenado entre $0..longitud(v) - 1$
 $(0 \leq a \leq longitud(v)) \wedge (-1 \leq b \leq longitud(v) - 1) \wedge (a \leq b + 1)$

todos los elementos a la izquierda de a son $\leq x$
 todos los elementos a la derecha de b son $> x$

$\Rightarrow v$ está ordenado entre $0..longitud(v) - 1$
 $(0 \leq m + 1 \leq longitud(v)) \wedge (-1 \leq b \leq longitud(v) - 1) \wedge (m + 1 \leq b + 1)$

todos los elementos a la izquierda de $m + 1$ son $\leq x$
 todos los elementos a la derecha de b son $> x$

Se razona de forma similar al anterior.

Debemos razonar también que la llamada inicial a la función auxiliar cumple la precondición

v está ordenado entre $0..longitud(v) - 1 \wedge a = 0 \wedge b = longitud(v) - 1 \Rightarrow v$ está ordenado entre $0..longitud(v) - 1$

$(0 \leq a \leq longitud(v)) \wedge (-1 \leq b \leq longitud(v) - 1) \wedge (a \leq b + 1)$

todos los elementos a la izquierda de a son $\leq x$
 todos los elementos a la derecha de b son $> x$

Que es cierto porque $longitud(v) \geq 0$.

(R.7) Función de combinación.

En los dos casos recursivos nos limitamos a propagar el resultado de la llamada recursiva:

$$p = p'$$

(R.8) Escritura de la llamada recursiva.

Cada una de las dos llamadas recursivas se puede escribir como una sola asignación:

```
p = buscaBin( v, x, m+1, b )

y

p = buscaBin( v, x, a, m-1 )
```

★ Con lo que finalmente la función queda:

```
int buscaBin( TElem v[], TElem x, int a, int b) {
// Pre: v está ordenado entre 0 .. longitud(v)-1
// ( 0 ≤ a ≤ longitud(v) ) && ( -1 ≤ b ≤ longitud(v)-1 ) && ( a ≤ b+1 )
// todos los elementos a la izquierda de 'a' son ≤ x
// todos los elementos a la derecha de 'b' son > x
    int p, m;

    if ( a == b+1 )
        p = a - 1;
    else {          // a ≤ b
        m = (a+b) / 2;
        if ( v[m] ≤ x )
            p = buscaBin( v, x, m+1, b );
```

```

    else
        p = buscaBin( v, x, a, m-1 );
    }
    return p;
// Post: devuelve el mayor i ( $0 \leq i \leq longitud(v) - 1$ ) que cumple  $v[i] \leq x$ 
//      si x es menor que todos los elementos de v, devuelve -1
}

```

También nos puede interesar definir una versión de la función en la que se recibe el vector junto con su tamaño n :

```

int buscaBin( TElem v[], int n, TElem x ) {
//  $0 \leq n \leq longitud(v)$  y todos los elementos a la derecha de  $n-1$  son  $> x$ 
    return buscaBin( v, x, 0, n-1 );
}

```

En este caso debemos llamar a `buscaBin(v, x, l)`, siendo l la longitud del vector v .

Nótese que es necesario escribir primero la función auxiliar para que sea visible desde la otra función.

2.2. Algoritmos avanzados de ordenación

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

2.2.1. Ordenación rápida (*quicksort*)

★ Especificación:

$\{0 \leq n \leq longitud(v)\}$

proc quickSort (TElem v[], int n)

{ord(v,0,n-1)}

void quickSort (TElem v[], int n) {

 quickSort(v, 0, n-1);

}

$\{0 \leq a \leq longitud(v) \wedge -1 \leq b \leq longitud(v) - 1 \wedge a \leq b + 1\}$

proc quickSort(TElem v[], int a, int b)

{ord(v,a,b)}

void quickSort(TElem v[], int a, int 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.

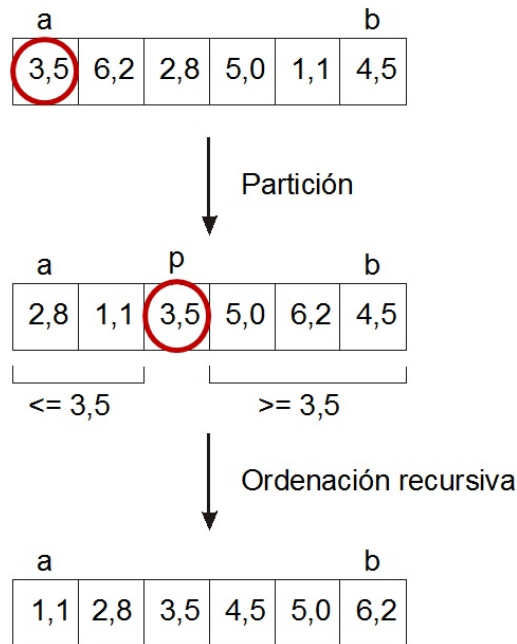


Figura 8: Planteamiento del algoritmo *quicksort*

★ El planteamiento recursivo consiste en:

- Elegir un pivote: un elemento cualquiera del subvector $v[a..b]$. Normalmente se elige $v[a]$ como pivote.
- Particionar 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 el centro, separando los menores de los mayores.

- Ordenar recursivamente los dos fragmentos que han quedado a la izquierda y a la derecha del pivote.

★ **Análisis de casos**

- Caso directo: $a = b + 1$ El subvector está vacío y, por lo tanto, ordenado.
- Caso recursivo: $a \leq b$ Se trata de un segmento no vacío y aplicamos el planteamiento recursivo:
 - considerar $x = v[a]$ como elemento pivote
 - reordenar parcialmente el subvector $v[a..b]$ para conseguir que x quede en la posición p que ocupará cuando $v[a..b]$ esté ordenado.
 - ordenar recursivamente $v[a..(p-1)]$ y $v[(p+1)..b]$.

★ **Función de acotación:**

$$t(v, a, b) = b - a + 1$$

★ **Utilizando el algoritmo de *partition* derivado en el tema 3, el algoritmo nos queda:**

```
void quickSort ( TElem v[], int a, int b) {
// Pre:  $0 \leq a \leq \text{longitud}(v)$  &&  $-1 \leq b \leq \text{longitud}(v)-1$  &&  $a \leq b+1$ 

    int p;

    if ( a <= b ) {
        particion(v, a, b, p);
        quickSort(v, a, p-1);
        quickSort(v, p+1, b);
    }

// Post: v está ordenado entre a y b
}
```

de forma que para ordenar todo el vector la llamada inicial es `quickSort(v, 0, l-1)` siendo l la longitud del vector v .

Otra versión que nos puede interesar es:

```
void quickSort ( TElem v[], int n) {
// Pre:  $0 \leq n \leq \text{longitud}(v)$ 

    quickSort(v, 0, n-1);

// Post: se ha ordenado v entre 0 y n-1
}
```

En este caso debemos llamar a `quickSort(v, l)`, siendo l la longitud del vector v .

2.2.2. Ordenación por mezcla (*mergesort*)

- ★ Partimos de una especificación similar a la del *quickSort*.

```

{0 ≤ n ≤ longitud(v)}
proc mergeSort ( TElem v[], int n)
{ord(v,0,n-1)}

void mergeSort ( TElem v[], int n) {
    mergeSort(v, 0, n-1);
}

{0 ≤ a ≤ longitud(v) ∧ -1 ≤ b ≤ longitud(v) - 1 ∧ a ≤ b + 1}
proc mergeSort( TElem v[], int a, int b)
{ord(v,a,b)}

void mergeSort( TElem v[], int a, int b) { }
```

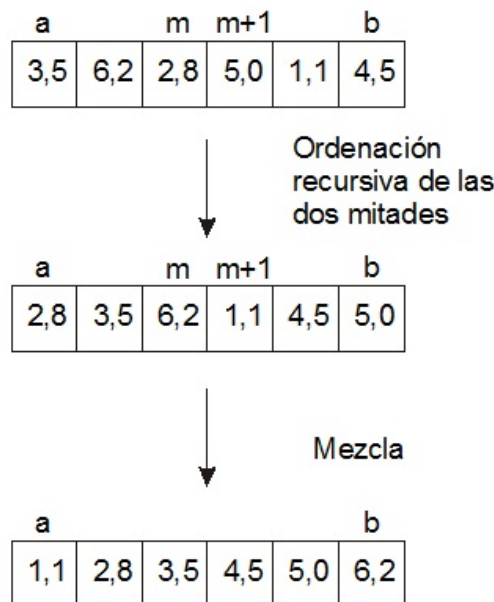


Figura 9: Planteamiento del algoritmo *mergesort*

- ★ 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 $v[(m+1)..b]$.
 - Mezclamos ordenadamente los subvectores $v[a..m]$ y $v[(m+1)..b]$ ya ordenados.
- ★ Análisis de casos
- Caso directo: $a \geq b$
 El subvector está vacío o tiene longitud 1 y, por lo tanto, está ordenado.
 $P_0 \wedge a \geq b \Rightarrow a = b \vee a = b + 1$
 $Q_0 \wedge (a = b \vee a = b + 1) \Rightarrow v = V$

■ Caso recursivo: $a < b$

Tenemos un subvector de longitud mayor o igual que 2, y aplicamos el planteamiento recursivo:

- Dividir $v[a..b]$ en dos mitades. Al ser la longitud ≥ 2 es posible hacer la división de forma que cada una de las mitades tendrá una longitud estrictamente menor que el segmento original (por eso hemos considerado como caso directo el subvector de longitud 1).
- Tomando $m = (a + b)/2$ ordenamos recursivamente $v[a..m]$ y $v[(m + 1)..b]$.
- Usamos un procedimiento auxiliar para mezclar las dos mitades, quedando ordenado todo $v[a..b]$.

★ Función de acotación. $t(v, a, b) = b - a + 1$

★ Utilizando la implementación para mezcla derivada en el tema 3, el procedimiento de ordenación queda:

```
void mergeSort( TElem v[], int a, int b) {
// Pre:  $0 \leq a \leq longitud(v) \ \&\& \ -1 \leq b \leq longitud(v)-1 \ \&\& \ a \leq b+1$ 

    int m;

    if ( a < b ) {
        m = (a+b) / 2;
        mergeSort( v, a, m);
        mergeSort( v, m+1, b);
        mezcla( v, a, m, b);
    }

// Post: v está ordenado entre a y b
}
```

de forma que para ordenar todo el vector la llamada inicial es `mergeSort (v, 0, l-1)` siendo l la longitud del vector v .

Otra versión que nos puede interesar es:

```
void mergeSort ( TElem v[], int n) {
// Pre:  $0 \leq n \leq longitud(v)$ 

    mergeSort(v, 0, n-1);

// Post: se ha ordenado v entre 0 y n-1
}
```

En este caso debemos llamar a `mergeSort (v, l)`, siendo l la longitud del vector v .

3. Análisis de la complejidad de algoritmos recursivos

3.1. Ecuaciones de recurrencias

- ★ La recursión no introduce nuevas instrucciones en el lenguaje. Sin embargo, cuando intentamos analizar la complejidad de una función o un procedimiento recursivo nos encontramos con que debemos conocer la complejidad de las llamadas recursivas.

La *definición natural* de la función de complejidad de un algoritmo recursivo también es recursiva, y viene dada por una o más *ecuaciones de recurrencia*.

3.1.1. Ejemplos

- ★ Cálculo del factorial.

Tamaño de los datos: n

Caso directo, $n = 0$: $T(n) = 2$

Caso recursivo:

- 1 de evaluar la condición +
- 1 de evaluar la descomposición $n - 1$ +
- 1 del producto $n * fact(n - 1)$ +
- 1 de la asignación de $n * fact(n - 1)$ +
- $T(n - 1)$ de la llamada recursiva.

Ecuaciones de recurrencia:
$$T(n) = \begin{cases} 2 & \text{si } n = 0 \\ 4 + T(n - 1) & \text{si } n > 0 \end{cases}$$

- ★ Multiplicación por el método del campesino egipcio.

Tamaño de los datos: $n = b$

Caso directo, $n = 0, 1$: $T(n) = 3$

En ambos casos recursivos:

- 4 de evaluar todas las condiciones en el caso peor +
- 1 de la asignación +
- 2 de evaluar la descomposición $2 * a$ y $b/2$ +
- 1 de la suma $prod(2 * a, b/2) + a$ en una de las ramas +
- $T(n/2)$ de la llamada recursiva.

Ecuaciones de recurrencia:
$$T(n) = \begin{cases} 3 & \text{si } n = 0, 1 \\ 8 + T(n/2) & \text{si } n > 1 \end{cases}$$

- ★ Para calcular el orden de complejidad no nos interesa el valor exacto de las constantes, ni nos preocupa que sean distintas (en los casos directos, o cuando se suma algo constante en los casos recursivos): ¡estudio asintótico!

3.1.2. Ejemplos

- ★ Números de Fibonacci.

Tamaño de los datos: n

Ecuaciones de recurrencia:
$$T(n) = \begin{cases} c_0 & \text{si } n = 0, 1 \\ T(n - 1) + T(n - 2) + c, & \text{si } n > 1 \end{cases}$$

★ Ordenación rápida (quicksort)

Tamaño de los datos: $n = num$

En el caso directo tenemos complejidad constante c_0 .

En el caso recursivo:

- El coste de la partición: $c * n +$
- El coste de las dos llamadas recursivas. El problema es que la disminución en el tamaño de los datos depende de los datos y de la elección del pivote.
 - El caso peor se da cuando el pivote no separa nada (es el máximo o el mínimo del subvector): $c_0 + T(n - 1)$
 - El caso mejor se da cuando el pivote divide por la mitad: $2 * T(n/2)$

Ecuaciones de recurrencia en el caso peor:

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

Ecuaciones de recurrencia en el caso mejor:

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

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.

3.2. Despliegue de recurrencias

- ★ Hasta ahora, lo único que hemos logrado es expresar la función de complejidad mediante ecuaciones recursivas. Pero es necesario encontrar una *fórmula explícita* que nos permita obtener el orden de complejidad buscado.
- ★ El objetivo de este método es conseguir una fórmula explícita de la función de complejidad, a partir de las ecuaciones de recurrencias. El proceso se compone de tres pasos:
 1. **Despliegue.** Sustituimos las apariciones de T en la recurrencia tantas veces como sea necesario hasta encontrar una fórmula que dependa del número de llamadas recursivas k .
 2. **Postulado.** A partir de la fórmula paramétrica resultado del paso anterior obtenemos una fórmula explícita. Para ello, se obtiene el valor de k que nos permite alcanzar un caso directo y, en la fórmula paramétrica, se sustituye k por ese valor y la referencia recursiva T por la complejidad del caso directo.
 3. **Demostración.** La fórmula explícita así obtenida sólo es correcta si la recurrencia para el caso recursivo también es válida para el caso directo. Podemos comprobarlo demostrando por inducción que la fórmula obtenida cumple las ecuaciones de recurrencia.

3.2.1. Ejemplos

★ Factorial

■ Ecuaciones

$$T(n) = \begin{cases} 2 & \text{si } n = 0 \\ 4 + T(n-1) & \text{si } n > 0 \end{cases}$$

■ Despliegue

$$\begin{aligned} T(n) &= 4 + T(n-1) \\ &= 4 + 4 + T(n-2) \\ &= 4 + 4 + 4 + T(n-3) \\ &\dots \\ &= 4 * k + T(n-k) \end{aligned}$$

■ Postulado

El caso directo se tiene para $n = 0$

$$n - k = 0 \Leftrightarrow k = n$$

$$T(n) = 4n + T(n-n) = 4n + T(0) = 4n + 2$$

Por lo tanto $T(n) \in O(n)$

★ Multiplicación por el método del campesino egipcio

■ Ecuaciones

$$T(n) = \begin{cases} 3 & \text{si } n = 0, 1 \\ 8 + T(n/2) & \text{si } n > 1 \end{cases}$$

■ Despliegue

$$\begin{aligned} T(n) &= 8 + T(n/2) \\ &= 8 + (8 + T(n/2/2)) \\ &= 8 + 8 + 8 + T(n/2/2/2) \\ &\dots \\ &= 8 * k + T(n/2^k) \end{aligned}$$

■ Postulado

Las llamadas recursivas terminan cuando se alcanza 1

$$n/2^k = 1 \Leftrightarrow k = \log n$$

$$T(n) = 8 \log n + T(n/2^{\log n}) = 8 \log n + T(1) = 8 \log n + 3$$

Por lo tanto $T(n) \in O(\log n)$

Si k representa el número de llamadas recursivas ¿qué ocurre cuando $k = \log n$ no tiene solución entera?

La complejidad $T(n)$ del algoritmo es una función monótona no decreciente, y, por lo tanto, nos basta con estudiar su comportamiento sólo en algunos puntos: los valores de n que son una potencia de 2. Esta simplificación no causa problemas en el cálculo asintótico.

★ Números de Fibonacci

■ Ecuaciones

$$T(n) = \begin{cases} c_0 & \text{si } n = 0, 1 \\ T(n-1) + T(n-2) + c, & \text{si } n > 1 \end{cases}$$

Podemos simplificar la resolución de la recurrencia, considerando que lo que nos interesa es una cota superior:

$$T(n) \leq 2 * T(n-1) + c_1 \quad \text{si } n > 1$$

■ Despliegue

$$\begin{aligned} T(n) &\leq c_1 + 2 * T(n-1) \\ &\leq c_1 + 2 * (c_1 + 2 * T(n-2)) \\ &\leq c_1 + 2 * (c_1 + 2 * (c_1 + 2 * T(n-3))) \\ &\leq c_1 + 2 * c_1 + 2^2 * c_1 + 2^3 * T(n-3) \\ &\dots \\ &\leq c_1 * \sum_{i=0}^{k-1} 2^i + 2^k * T(n-k) \end{aligned}$$

■ Postulado

Las llamadas recursivas terminan cuando se alcanzan 0 y 1. Como buscamos una cota superior, consideramos 0.

$$n - k = 0 \Leftrightarrow k = n$$

$$\begin{aligned} T(n) &\leq c_1 * \sum_{i=0}^{n-1} 2^i + 2^n T(n-n) \\ &= c_1 * \sum_{i=0}^{n-1} 2^i + 2^n T(0) \\ &= c_1 * \sum_{i=0}^{n-1} 2^i + 2^n T(0) \\ &= c_1 * (2^n - 1) + c_0 * 2^n \\ &= (c_0 + c_1) * 2^n - c_1 \end{aligned}$$

donde hemos utilizado la fórmula para la suma de progresiones geométricas:

$$\sum_{i=0}^{n-1} r^i = \frac{r^n - 1}{r - 1} \quad r \neq 1$$

Por lo tanto $T(n) \in O(2^n)$

Las funciones recursivas múltiples donde el tamaño del problema disminuye por sustracción tienen costes prohibitivos, como en este caso donde el coste es exponencial.

3.3. Resolución general de recurrencias

- ★ Utilizando la técnica de despliegue de recurrencias y algunos resultados sobre convergencia de series, se pueden obtener unos resultados teóricos para la obtención de fórmulas explícitas, aplicables a un gran número de ecuaciones de recurrencias.

3.3.1. Disminución del tamaño del problema por sustracción

- ★ Cuando: (1) la descomposición recursiva se obtiene restando una cierta cantidad constante, (2) el caso directo tiene coste constante, y (3) la preparación de las llamadas

y de combinación de los resultados tiene coste polinómico, entonces la ecuación de recurrencias será de la forma:

$$T(n) = \begin{cases} c_0 & \text{si } 0 \leq n < n_0 \\ a * T(n - b) + c * n^k & \text{si } n \geq n_0 \end{cases}$$

donde:

- c_0 es el coste en el caso directo,
- $a \geq 1$ es el número de llamadas recursivas,
- $b \geq 1$ es la disminución del tamaño de los datos, y
- $c * n^k$ es el coste de preparación de las llamadas y de combinación de los resultados.

Se puede demostrar:

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

Vemos que, cuando el tamaño del problema disminuye por sustracción,

- En recursión simple ($a = 1$) el coste es polinómico y viene dado por el producto del coste de cada llamada ($c * n^k$) y el coste lineal de la recursión (n).
- En recursión múltiple ($a > 1$), por muy grande que sea b , el coste siempre es exponencial.

3.3.2. Disminución del tamaño del problema por división

- ★ Cuando: (1) la descomposición recursiva se obtiene dividiendo por una cierta cantidad constante, (2) el caso directo tiene coste constante, y (3) la preparación de las llamadas y de combinación de los resultados tiene coste polinómico, entonces la ecuación de recurrencias será de la forma:

$$T(n) = \begin{cases} c_1 & \text{si } 0 \leq n < n_0 \\ a * T(n/b) + c * n^k & \text{si } n \geq n_0 \end{cases}$$

donde:

- c_1 es el coste en el caso directo,
- $a \geq 1$ es el número de llamadas recursivas,
- $b \geq 2$ es el factor de disminución del tamaño de los datos, y
- $c * n^k$ es el coste de preparación de las llamadas y de combinación de los resultados.

Se puede demostrar:

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

Si $a \leq b^k$ la complejidad depende de n^k que es el término que proviene de $c * n^k$ en la ecuación de recurrencias, y, por lo tanto, la complejidad de un algoritmo de este tipo se puede mejorar disminuyendo la complejidad de la preparación de las llamadas y la combinación de los resultados.

Si $a > b^k$ las mejoras en la eficiencia se pueden conseguir

- disminuyendo el número de llamadas recursivas a o aumentando el factor de disminución del tamaño de los datos b , o bien
- optimizando la preparación de las llamadas y combinación de los resultados, pues, si esto hace disminuir k suficientemente, podemos pasar a uno de los otros casos: $a = b^k$ o incluso $a < b^k$.

3.3.3. Ejemplos

- ★ Búsqueda binaria.

Tamaño de los datos: $n = num$

Recurrencias:

$$T(n) = \begin{cases} c_1 & \text{si } n = 0 \\ T(n/2) + c & \text{si } n > 0 \end{cases}$$

Se ajusta al esquema teórico de disminución del tamaño del problema por división, con los parámetros:

$$a = 1, b = 2, k = 0$$

Estamos en el caso $a = b^k$ y la complejidad resulta ser:

$$O(n^k \log n) = O(n^0 \log n) = O(\log n)$$

- ★ Ordenación por mezcla (mergesort).

Tamaño de los datos: $n = num$

Recurrencias:

$$T(n) = \begin{cases} c_1 & \text{si } n \leq 1 \\ 2 * T(n/2) + c * n & \text{si } n \geq 2 \end{cases}$$

donde $c * n$ es el coste del procedimiento mezcla.

Se ajusta al esquema teórico de disminución del tamaño del problema por división, con los parámetros:

$$a = 2, b = 2, k = 1$$

Estamos en el caso $a = b^k$ y la complejidad resulta ser:

$$O(n^k \log n) = O(n \log n)$$

Este es también el coste del algoritmo *pareado* que vimos en la sección anterior, ya que la recurrencia es la misma. En la siguiente sección vamos a ver cómo generalizar aun más esta función para hacerla más eficiente.

4. Técnicas de generalización de algoritmos recursivos

- ★ En este tema ya hemos utilizado varias veces este tipo de técnicas (también conocidas como *técnicas de inmersión*), con el objetivo de conseguir planteamientos recursivos. La ordenación rápida

```
void quickSort( TElem v[], int a, int b) {
// Pre: 0 ≤ a ≤ longitud(v) && -1 ≤ b ≤ longitud(v)-1 && a ≤ b+1

// Post: v está ordenado entre a y b
}
```

```

void quickSort ( TElem v[], int n) {
// Pre:  $0 \leq n \leq longitud(v)$ 

// Post: se ha ordenado v
}

```

- ★ Además de para conseguir realizar un planteamiento recursivo, las generalizaciones también se utilizan para

- transformar algoritmos recursivos ya implementados en algoritmos recursivos finales, que se pueden transformar fácilmente en algoritmos iterativos.
- mejorar la eficiencia de los algoritmos recursivos añadiendo parámetros y/o resultados acumuladores.

La versión recursiva final del factorial

```

int acuFact( int a, int n ) {
// Pre:  $a \geq 0 \ \&\& \ n \geq 0$ 

// Post: devuelve  $a * n!$ 
}

int fact( int n ) {
// Pre:  $n \geq 0$ 

// Post: devuelve  $n!$ 
}

```

- ★ Decimos que una acción parametrizada (procedimiento o función) F es una generalización de otra acción f cuando:

- F tiene más parámetros de entrada y/o devuelve más resultados que f .
- Particularizando los parámetros de entrada adicionales de F a valores adecuados y/o ignorando los resultados adicionales de F se obtiene el comportamiento de f .

En el ejemplo de la ordenación rápida:

- f : **void** quickSort (TElem v[], **int** n)
- F : **void** quickSort (TElem v[], **int** a, **int** b)
- En F se añaden los parámetros a y b . Mientras f siempre se aplica al intervalo $0..longitud(v) - 1$, F se puede aplicar a cualquier subintervalo del array determinado por los índices $a..b$.
- Particularizando los parámetros adicionales de F como $a = 0$, $b = longitud(v) - 1$, se obtiene el comportamiento de f . Razonando sobre las postcondiciones:

$$v \text{ está ordenado entre } a \text{ y } b \wedge a = 0 \wedge b = longitud(v) - 1 \Rightarrow \text{se ha ordenado } v$$

En el ejemplo del factorial:

- f : **int** fact(**int** n)
 - F : **int** acuFact(**int** a, **int** n)
-

- En F se ha añadido el nuevo parámetro a donde se va acumulando el resultado a medida que se construye.
- Particularizando el parámetro adicional de F como $a = 1$, se obtiene el comportamiento de f . Razonando sobre las postcondiciones:

devuelve $a * n! \wedge a = 1 \Rightarrow$ devuelve $n!$

4.1. Planteamientos recursivos finales

- ★ Dada una especificación E_f pretendemos encontrar una especificación E_F más general que admita una solución recursiva final.
- ★ El resultado se ha de obtener en un caso directo y, para conseguirlo, lo que podemos hacer es añadir nuevos parámetros que vayan acumulando el resultado obtenido hasta el momento, de forma que al llegar al caso directo de F el valor de los parámetros acumuladores sea el resultado de f .
- ★ Para obtener la especificación E_F a partir de E_f
 - Fortalecemos la precondition de E_f para exigir que alguno de los parámetros de entrada ya traiga calculado una parte del resultado.
 - Mantenemos la misma postcondición.
- ★ Ejemplo: cálculo recursivo de una potencia. Tenemos la siguiente función recursiva no final para calcular la potencia natural de un entero a :

```
// Pre:  $n \geq 0$ 
int potencia (int a, int n) {
    int p;

    if (n == 0)
        p = 1;
    else //  $n > 0$ 
        p = potencia(a, n-1) * a;
    return p;
}
//Post:  $p = a^n$ 
```

Una forma de obtener una versión recursiva final consiste en añadir un parámetro ac , que lleve calculado parte del producto. Para poder diseñar la función nos hace falta saber qué potencia lleva acumulada, así que introducimos otro parámetro i y aseguramos en la precondition que $ac = a^i$. Por tanto, el caso base, cuando $i == n$, ac ya lleva acumulado lo que queremos y ese es el resultado de la función. En caso contrario, acumulamos un factor a más a ac e incrementamos la i en 1 para que se cumpla la precondition en la llamada recursiva. El algoritmo es el siguiente:

De esa forma

```
int potencia(int a, int n, int ac, int i) {
//Pre :  $ac = a^i \wedge 0 \leq i \leq n$ 
    int p;
    if (i == n)
        p = ac;
    else
        p = potencia(a, n, ac*a, i+1);
}
```

```

    return p;
// Post:  $p = a^n$ 
}

```

donde la llamada inicial se hace con $i = 0$, ya que aun no hemos acumulado nada y consecuentemente $ac = 1$:

```

int potencia(int a, int n) {
// Pre:  $n \geq 0$ 
    return potencia(a, n, 1, 0);
// Post:  $p = a^n$ 
}

```

Recuérdese que esto no hace que la función sea asintóticamente más eficiente.

4.2. Generalización por razones de eficiencia

- ★ Suponemos ahora que partimos de un algoritmo recursivo ya implementado y que nos proponemos mejorar su eficiencia introduciendo parámetros y/o resultados adicionales.
- ★ Se trata de simplificar algún cálculo auxiliar, sacando provecho del resultado obtenido para ese cálculo en otra llamada recursiva. Introducimos parámetros adicionales, o resultados adicionales, según si queremos aprovechar los cálculos realizados en llamadas anteriores, o posteriores, respectivamente. En ocasiones, puede interesar añadir tanto parámetros como resultados adicionales.

4.2.1. Generalización con resultados acumuladores

- ★ Se aplica esta técnica cuando en un algoritmo recursivo f se detecta una expresión $e(\vec{x}', \vec{y}')$, que puede depender de los parámetros de entrada y los resultados de la llamada recursiva y cuyo cálculo se puede simplificar utilizando el valor de esa expresión en posteriores llamadas recursivas. Obviamente, si la expresión depende de los resultados de la llamada recursiva, debe aparecer después de dicha llamada.
- ★ Se construye una generalización F que posee resultados adicionales \vec{b} , cuya función es transmitir el valor de $e(\vec{x}, \vec{y})$. La precondition de F se mantiene constante

$$P'(\vec{x}) \Leftrightarrow P(\vec{x})$$

mientras que la postcondición de F se plantea como un fortalecimiento de la postcondición de f

$$Q'(\vec{x}, \vec{b}, \vec{y}) \Leftrightarrow Q(\vec{x}, \vec{y}) \wedge \vec{b} = e(\vec{x}, \vec{y})$$

- ★ El algoritmo F se obtiene a partir de f del siguiente modo:
 - Se reutiliza el texto de f , reemplazando $e(\vec{x}', \vec{y}')$ por \vec{b}'
 - Se añade el cálculo de \vec{b} , de manera que la parte $\vec{b} = e(\vec{x}, \vec{y})$ de la postcondición $Q'(\vec{x}, \vec{b}, \vec{y})$ quede garantizada, tanto en los casos directos como en los recursivos.

La técnica resultará rentable siempre que F sea más eficiente que f .

- ★ Ejemplo: recordemos el algoritmo *pareado* visto en la Sección 2. Podemos obtener una versión más eficiente del algoritmo añadiendo un resultado adicional que determina el número de pares que hay en el segmento considerado. De esa manera, las propias llamadas recursivas nos devolverán el número de pares de cada mitad y la comprobación adicional que necesitamos se hará en tiempo constante. Como ahora tenemos dos resultados, utilizamos un procedimiento con dos parámetros de salida: el booleano b y el entero p que representa el número de números pares:

```
{0 ≤ i ≤ j ≤ longitud(v) ∧ ∀k : 0 ≤ i ≤ k < j : v[k] >= 0}
proc pareado (int v[], int i, int j, out bool b, out int p)
{b = esPareado(v, i, j) ∧ p = #l : i ≤ l < j : v[l] mod 2 = 0}
```

El caso base sigue siendo el mismo, $j \leq i + 1$, el cual engloba segmentos vacíos y unitarios. En ambos casos habría que devolver $b = \text{true}$. Respecto a p , para segmentos vacíos se devolvería 0, y para segmentos unitarios se devolvería 0, o 1 si el número del segmento es par.

En el caso recursivo, cada una de las dos llamadas, devuelve un booleano ($b1$ y $b2$ respectivamente) y un entero ($p1$ y $p2$ respectivamente): *pareado*($v, i, m, b1, p1$) y *pareado*($v, m, j, b2, p2$).

Con lo cual el caso recursivo sería ser de la siguiente forma:

```
int m = (i+j)/2;
bool b1, b2;
int p1, p2;
pareado(v, i, m, b1, p1);
pareado(v, m, j, b2, p2);
b = b1 && b2 && (abs(p1-p2) <= 1); //comprobacion de coste constante

p = p1+p2;
```

Por tanto el algoritmo completo es el siguiente:

```
// Pre: 0 ≤ i ≤ j ≤ longitud(v) ∧ ∀k : 0 ≤ i ≤ k < j : v[k] >= 0
void pareado (int v[], int i, int j, bool &b, int &p) {
    if (i == j) {b = true; p = 0;} // Segmento vacio
    else if (j == i+1) { // Segmento unitario
        b = true;
        if (v[i] % 2 == 0) p=1; else p = 0;
    } else { // Segmento de longitud >= 2
        int m = (i+j)/2;
        bool b1, b2;
        int p1, p2;
        pareado(v, i, m, b1, p1);
        pareado(v, m, j, b2, p2);
        b = b1 && b2 && (abs(p1-p2) <= 1); //comprobacion de coste constante
        p = p1+p2;
    }
}
// Post: b = esPareado(v, i, j) ∧ p = #l : i ≤ l < j : v[l] mod 2 = 0}
```

siendo la llamada inicial

```
// Pre: longitud(v) ≥ num ∧ ∀k : 0 ≤ k < num : v[k] >= 0
bool pareado (int v[], int num) {
    bool b; int p;
    pareado(v, 0, num, b, p);
}
```

```

return b;
}
// Post: b = esPareado(v, 0, num)

```

En este caso tenemos la siguiente recurrencia, siendo n el tamaño del segmento ($j - i$)

$$T(n) = \begin{cases} c_0 & \text{si } n = 0 \\ 2 * T(n/2) + c_1 & \text{si } n > 0 \end{cases}$$

Estamos en el caso en que $a = 2$, $b = 2$ y $k = 0$, por lo que $T(n) \in O(n^{\log_2 2})$, es decir, $T(n) \in O(n)$.

4.2.2. Generalización con parámetros acumuladores

- ★ Se aplica esta técnica cuando en un algoritmo recursivo f se detecta una expresión $e(\vec{x})$, que sólo depende de los parámetros de entrada, cuyo cálculo se puede simplificar utilizando el valor de esa expresión en anteriores llamadas recursivas.
- ★ Se construye una generalización F que posee parámetros de entrada adicionales \vec{a} , cuya función es transmitir el valor de $e(\vec{x})$. La precondition de F se plantea como un fortalecimiento de la precondition de f .

$$P'(\vec{a}, \vec{x}) \Leftrightarrow P(\vec{x}) \wedge \vec{a} = e(\vec{x})$$

mientras que la postcondición se mantiene constante

$$Q'(\vec{a}, \vec{x}, \vec{y}) \Leftrightarrow Q(\vec{x}, \vec{y})$$

- ★ El algoritmo F se obtiene a partir de f del siguiente modo:
 - Se reutiliza el texto de f , reemplazando $e(\vec{x})$ por \vec{a}
 - Se diseña una nueva función sucesor $s'(\vec{a}, \vec{x})$, a partir de la original $s(\vec{x})$, de modo que se cumpla:

$$\{\vec{a} = e(\vec{x}) \wedge r(\vec{x})\}$$

$$(\vec{a}', \vec{x}') = s'(\vec{a}, \vec{x})$$

$$\{\vec{x}' = s(\vec{x}) \wedge \vec{a}' = e(\vec{x}')\}$$

La técnica resultará rentable cuando en el cálculo de \vec{a}' nos podamos aprovechar de los valores de \vec{a} y \vec{x} para realizar un cálculo más eficiente.

- ★ Ejemplo: supongamos que deseamos rellenar un vector de tamaño $2^i - 1$ para un cierto i con naturales de la siguiente manera:
 - El elemento del centro es 1.

			1			
--	--	--	---	--	--	--

- El punto medio de la mitad izquierda se obtiene multiplicando por 2 el elemento del centro y el de la mitad derecha multiplicando por 3 el elemento del centro.

	2		1		3	
--	---	--	---	--	---	--

- Para rellenar el resto de posiciones se procede de la misma manera: multiplicando por 2 cuando vamos a la izquierda y por 3 si vamos a la derecha.

4	2	6	1	6	3	9
---	---	---	---	---	---	---

Cada posición del vector contendrá por tanto $2^i * 3^j$ para ciertos valores de i y j , dependiendo de la posición. Para no tener que calcular dichas potencias en el punto de asignación al vector, llevaremos un parámetro adicional que acumula el valor de dichas potencias:

```
// Pre: 0<=i<=j+1<=n
void rellenar(int v[],int i,int j, int ac)
{ if (i<=j)
  { int m= (i+j)/2; v[m]=ac;
    rellenar(v,i,m-1,ac*2);
    rellenar(v,m+1,j,ac*3);
  }
}
// Post: rellenado(v,i,j,ac)
```

siendo la propiedad *rellenado* fácilmente definible de manera recursiva:

$$\begin{aligned} \text{rellenado}(v,i,j,ac) \equiv (i \leq j) \Rightarrow & v[(i+j)/2] == ac \\ & \wedge \text{rellenado}(v,i,(i+j)/2-1,ac*2) \\ & \wedge \text{rellenado}(v,(i+j)/2+1,j,ac*3) \end{aligned}$$

La llamada inicial sería $\text{rellenar}(v,0,n-1,1)$ siendo n la longitud del vector; y por tanto se cumplirá $\text{rellenado}(v,0,n-1,1)$.

5. Verificación de algoritmos recursivos

- ★ Supondremos que la llamada recursiva devuelve los valores deseados para demostrar que la etapa de combinación cumple con la especificación.
- ★ Debemos probar que:

1. Se cubren todos los casos:

$$P(\vec{x}) \Rightarrow d(\vec{x}) \vee r(\vec{x})$$

2. El caso base es correcto:

$$P(\vec{x}) \wedge d(\vec{x}) \Rightarrow Q(\vec{x}, \vec{y})$$

3. Los argumentos de la llamada recursiva deben satisfacer su precondition:

$$P(\vec{x}) \wedge r(\vec{x}) \Rightarrow P(s(\vec{x}))$$

4. El paso de inducción es correcto (\Leftrightarrow la etapa de combinación es correcta):

$$P(\vec{x}) \wedge r(\vec{x}) \wedge Q(s(\vec{x}), \vec{y}') \Rightarrow Q(\vec{x}, c(\vec{x}, \vec{y}'))$$

Para garantizar que termine la secuencia de llamadas a la función debemos exigir además:

1. Existe una función de cota $t(\vec{x})$ tal que:

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

2. La función de cota debe decrecer en cada iteración:

$$P(\vec{x}) \wedge r(\vec{x}) \Rightarrow t(s(\vec{x})) < t(\vec{x})$$

★ **Ejemplo.** Verifica el siguiente algoritmo:

```
// fun potencia (int a, int n): return (int p)
// P:  $n \geq 0$ 
int potencia (int a, int n)
{
    int p;

    if (n == 0)
        p = 1;
    else //  $n > 0$ 
        p = potencia(a, n-1) * a;
    return p;
}
// Q:  $p = a^n$ 
```

Solución:

- Como $n \geq 0$: $n = 0 \vee n > 0$ es cierto.
- El caso base verifica la postcondición:

$$\{n = 0\}p = 1\{p = a^n\}$$

ya que $1 = a^n \Leftarrow n = 0$.

- En cada iteración la llamada recursiva verifica la precondition. El único problema consistiría en que n dejase de ser ≥ 0 . Como en el caso recursivo $n > 0$, $n - 1$ sigue siendo ≥ 0 .
- El paso de inducción es correcto:

$$n \geq 0 \wedge n > 0 \wedge p = a^{n-1} \Rightarrow (p = a^n)_p^{p*a}$$

- Consideremos la siguiente cota:

$$t = n(\geq 0)$$

- Decrece:

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

Notas bibliográficas

El contenido de este capítulo se basa en su mayor parte en el capítulo correspondiente de (Rodríguez Artalejo et al., 2011). El apartado final de verificación de algoritmos recursivos se puede encontrar en (Peña, 2005). Finalmente, pueden encontrarse ejercicios resueltos relacionados con este tema en (Martí Oliet et al., 2012).

Ejercicios

Diseño de algoritmos recursivos

1. Dado un vector de enteros de longitud n , diseñar un algoritmo de complejidad $O(n \log n)$ que encuentre el par de enteros más cercanos (en valor). ¿Se puede hacer con complejidad $O(n)$?
2. Dado un vector de enteros de longitud n , diseñar un algoritmo de complejidad $O(n \log n)$ que encuentre el par de enteros más lejanos (en valor). ¿Se puede hacer con complejidad $O(n)$?
3. Diseñar un algoritmo recursivo que realice el cambio de base de un número binario dado en su correspondiente decimal.
4. Implementa una función recursiva simple *cuadrado* que calcule el cuadrado de un número natural n , basándote en el siguiente análisis de casos:
 - Caso directo: Si $n = 0$, entonces $n^2 = 0$
 - Caso recursivo: Si $n > 0$, entonces $n^2 = (n - 1)^2 + 2 * (n - 1) + 1$
5. Implementa una función recursiva *log* que calcule la parte entera de $\log_b n$, siendo los datos b y n enteros tales que $b \geq 2 \wedge n \geq 1$. El algoritmo obtenido deberá usar solamente las operaciones de suma y división entera.
6. Implementa una función recursiva *bin* tal que, dado un número natural n , $bin(n)$ sea otro número natural cuya representación decimal tenga los mismos dígitos que la representación binaria de n . Es decir, debe tenerse: $bin(0) = 0$; $bin(1) = 1$; $bin(2) = 10$; $bin(3) = 11$; $bin(4) = 100$; etc.
7. Implementa un procedimiento recursivo simple *dosFib* que satisfaga la siguiente especificación:

```
void dosFib( int n, int& r, int& s ) {
// Pre:  n ≥ 0
// Post: r = fib(n) ∧ s = fib(n + 1)
}
```

En la postcondición, $fib(n)$ y $fib(n+1)$ representan los números que ocupan los lugares n y $n+1$ en la sucesión de Fibonacci, para la cual suponemos la definición recursiva habitual.

8. Implementa una función recursiva que calcule el número combinatorio $\binom{n}{m}$ a partir de los datos m, n enteros tales que $n \geq 0 \wedge m \geq 0$. Usa la recurrencia siguiente:

$$\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m}$$

siendo $0 < m < n$

9. El problema de las torres de Hanoi consiste en trasladar una torre de n discos de diferentes tamaños desde la varilla *ini* a la varilla *fin*, con ayuda de la varilla *aux*. Inicialmente los n discos están apilados de mayor a menor, con el más grande en la base. En ningún momento se permite que un disco repose sobre otro menor que él.

Los movimientos permitidos consisten en desplazar el disco situado en la cima de una de las varillas a la cima de otra, respetando la condición anterior. Construye un procedimiento recursivo *hanoi* tal que la llamada *hanoi*(*n*, *ini*, *fin*, *aux*) produzca el efecto de escribir una serie de movimientos que represente una solución del problema de Hanoi. Supón disponible un procedimiento *movimiento*(*i*, *j*), cuyo efecto es escribir "Movimiento de la varilla i a la varilla j".

Análisis de algoritmos recursivos

10. En cada uno de los casos que siguen, plantea una ley de recurrencia para la función $T(n)$ que mide el tiempo de ejecución del algoritmo en el caso peor, y usa el método de desplegado para resolver la recurrencia.
 - a) La función *cuadrado* (ejercicio 4).
 - b) Función *log* (ejercicio 5).
 - c) Función *bin* (ejercicio 6).
 - d) Procedimiento *dosFib* (ejercicio 7).
 - e) Función del ejercicio 8.
 - f) Procedimiento *hanoi* (ejercicio 9).
11. Aplica las reglas de análisis para dos tipos comunes de recurrencia a los algoritmos recursivos del ejercicio anterior. En cada caso, deberás determinar si el tamaño de los datos del problema decrece por sustracción o por división, así como los parámetros relevantes para el análisis.
12. En cada caso, calcula a partir de las recurrencias el orden de magnitud de $T(n)$. Hazlo aplicando las reglas de análisis para dos tipos comunes de recurrencia.
 - a) $T(1) = c_1; T(n) = 4 * T(n/2) + n$, si $n > 1$
 - b) $T(1) = c_1; T(n) = 4 * T(n/2) + n^2$, si $n > 1$
 - c) $T(1) = c_1; T(n) = 4 * T(n/2) + n^3$, si $n > 1$
13. Usa el método de desplegado para estimar el orden de magnitud de $T(n)$, suponiendo que T obedezca la siguiente recurrencia:

$$T(1) = 1; T(n) = 2 * T(n/2) + n \log n, \text{ si } n > 1$$

¿Pueden aplicarse en este caso las reglas de análisis para dos tipos comunes de recurrencia? ¿Por qué?

Eliminación de la recursión final

14. A continuación se presentan dos implementaciones iterativas del algoritmo de la búsqueda binaria. La primera versión del algoritmo es la misma que aparecía en el tema anterior, mientras que la segunda versión es el resultado de transformar a iterativo la versión recursiva de este algoritmo que se ha visto en este tema.
 ¿En qué se diferencian estos dos algoritmos iterativos?
 Escribe un algoritmo recursivo final con la misma idea de la primera versión iterativa.

```

int buscaBin( TElem v[], int num, TElem x )
{
    int izq, der, centro;

    izq = -1;
    der = num;
    while ( der != izq+1 ) {
        centro = (izq+der) / 2;
        if ( v[centro] <= x )
            izq = centro;
        else
            der = centro;
    }
    return izq;
}

```

```

int buscaBin( TElem v[], int num, TElem x )
{
    int a, b, p, m;

    a = 0;
    b = num-1;
    while ( a <= b ) {
        m = (a+b) / 2;
        if ( v[m] <= x )
            a = m+1;
        else
            b = m-1;
    }
    p = a - 1;
    return p;
}

```

Técnicas de generalización

15. (🐘ACR227) Comprueba que el procedimiento *combiGen* especificado como sigue es una generalización de la función del ejercicio 11, y que *combiGen* admite un algoritmo recursivo simple, más eficiente que la recursión doble anterior, implementándolo.

```

void combiGen ( int a, int b, int v[]) {
    // Pre:  0 <= b <= a && b < longitud(v)
    // Post: para cada i entre 0 y b se tiene  $v[i] = \binom{a}{i}$  }

```

16. Especifica e implementa un algoritmo recursivo que dado un vector v de enteros, que viene dado en orden estrictamente creciente, determine si el vector contiene alguna posición i que cumpla $v[i] = i$. Utilizando el método de despliegue de recurrencias, analiza la complejidad temporal en el caso peor del algoritmo obtenido.
17. (Martí Oliet et al. (2012)) Un hostel ofrece alojamiento a turistas todos los días comprendidos en un intervalo $[0..N)$ cuya longitud $N \geq 0$ se sabe que es par. El precio de una estancia diaria varía cada día, siendo:

- 1 euro para los días 0 y $N - 1$ (comienzo y fin de temporada),
- 2 euros para los días 1 y $N - 2$,
- 2^2 euros para los días 2 y $N - 3$,

y así sucesivamente; es decir, el precio se va multiplicando por dos a medida que nos acercamos al centro del intervalo (temporada alta). Especificar e implementar una función recursiva que calcula los ingresos obtenidos por el hostel durante una temporada a partir de un vector que almacena en cada posición k el número de huéspedes del día k .

18. (Martí Oliet et al. (2012)) Dos cifras decimales (comprendidas entre 0 y 9) son *pareja* si suman 9. Dado un número natural n , llamaremos *complementario* de n al número obtenido a partir de la representación decimal de n , cambiando cada cifra por su pareja. Por ejemplo, el complementario de 146720 es 853279. Diseñar un algoritmo que, dado un número natural n , calcule su complementario.

Verificación de algoritmos recursivos

19. Verifica los algoritmos recursivos de los ejercicios anteriores
20. Verifica el siguiente algoritmo:

```

{ $n \geq 0$ }
int potenciaLog(int a, int n)
{
    int p;
    if (n == 0)
        p = 1;
    else
    {
        p = potenciaLog(a, n/2);
        p = p * p;
        if (n%2 == 1)
            p = p*a
    }
    return p;
}
{ $p = a^n$ }

```

21. Verifica el siguiente algoritmo:

```

{ $n \geq 0$ }
int productoEscalar(int v[], int w[], int n)
{
    int r;

    if (n == 0)
        r = 0;
    else
        r = v[n-1] * w[n-1] + productoEscalar(v, w, n-1);
    return r;
}
{ $r = (\sum i : 0 \leq i < n : v[i] * w[i])$ }

```

22. Verifica el siguiente algoritmo:

```
{ $n \geq 0 \wedge a \geq 0 \wedge a^2 \leq n$ }  
int raiz (int n, int a)  
{  
    int r;  
  
    if ((a+1)*(a+1) > n)  
        r = a;  
    else  
        r = raiz (n, a+1);  
    return r;  
}  
{ $r^2 \leq n < (r+1)^2$ }
```

El esquema “Divide y vencerás”¹

Divide et impera

Julio César (100 a.C.-44 a.C)

RESUMEN: En este tema se presenta el esquema algorítmico *Divide y vencerás*, que es un caso particular del diseño recursivo, y se ilustra con ejemplos significativos en los que la estrategia reporta beneficios claros. Se espera del alumno que incorpore este método a sus estrategias de resolución de problemas.

1. Introducción

- ★ En este capítulo iniciamos la presentación de un conjunto de *esquemas algorítmicos* que pueden emplearse como estrategias de resolución de problemas. Un esquema puede verse como un algoritmo *genérico* que puede resolver distintos problemas. Si se concretan los tipos de datos y las operaciones del esquema genérico con los tipos y operaciones específicos de un problema concreto, tendremos un algoritmo para resolver dicho problema.
- ★ Además de *divide y vencerás*, este curso veremos el esquema de *vuelta atrás*. En cursos posteriores aparecerán otros esquemas con nombres propios tales como el *método voraz*, el de *programación dinámica* y el de *ramificación y poda*. Cada uno de ellos resuelve una familia de problemas de características parecidas.
- ★ Los esquemas o métodos algorítmicos deben verse como un conjunto de algoritmos *prefabricados* que el diseñador puede ensayar ante un problema nuevo. No hay garantía de éxito pero, si se alcanza la solución, el esfuerzo invertido habrá sido menor que si el diseño se hubiese abordado desde cero.
- ★ El esquema *divide y vencerás* (DV) consiste en **descomponer** el problema dado en uno o varios subproblemas del mismo tipo, el tamaño de cuyos datos es **una fracción** del tamaño original. Una vez resueltos los subproblemas por medio de la aplicación recursiva del algoritmo, se **combinan** sus resultados para construir la solución del problema original. Existirán uno o más **casos base** en los que el problema no se

¹Ricardo Peña es el autor principal de este tema. Modificado por Clara Segura en el curso 2013-14.

subdivide más y se resuelve, o bien directamente si es sencillo, o bien utilizando un algoritmo distinto.

- ★ Aparentemente estas son las características generales de todo diseño recursivo y de hecho el esquema DV es un caso particular del mismo. Para distinguirlo de otros diseños recursivos que no responden a DV se han de cumplir las siguientes condiciones:
 - Los subproblemas han de tener un tamaño que sea una *fracción* del tamaño original (un medio, un tercio, etc ...). No basta simplemente con que sean más pequeños.
 - Los subproblemas se generan *exclusivamente* a partir del problema original. En algunos diseños recursivos, los parámetros de una llamada pueden depender de los resultados de otra previa. En el esquema DV, no.
 - La solución del problema original se obtiene *combinando los resultados* de los subproblemas entre sí, y posiblemente con parte de los datos originales. Otras posibles combinaciones no encajan en el esquema.
 - El (los) caso(s) base no son necesariamente los casos triviales. Como veremos más adelante podría utilizarse como caso base (incluso debería utilizarse en ocasiones) un algoritmo distinto al algoritmo recursivo DV.
- ★ Puesto en forma de código, el esquema DV tiene el siguiente aspecto:

```

template <class Problema, class Solución>
Solución divide-y-vencerás (Problema x) {
    Problema x_1,...,x_k;
    Solución y_1,...,y_k;

    if (base(x))
        return método-directo(x);
    else {
        (x_1,..., x_k) = descomponer(x);
        for (i=1; i<=k; i++)
            y_i = divide-y-vencerás(x_i);
        return combinar(x, y_1,..., y_k);
    }
}

```

- ★ Los tipos Problema, Solución, y los métodos base, método-directo, descomponer y combinar, son específicos de cada problema resuelto por el esquema.
- ★ Para saber si la aplicación del esquema DV a un problema dado resultará en una solución eficiente o no, se deberá utilizar la recurrencia vista en el Capítulo 4 en la que el tamaño del problema disminuía *mediante división*:

$$T(n) = \begin{cases} c_1 & \text{si } 0 \leq n < n_0 \\ a * T(n/b) + c * n^k & \text{si } n \geq n_0 \end{cases}$$

- ★ Recordemos que la solución de la misma era:

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

- ★ Para obtener una solución eficiente, hay que conseguir a la vez:
 - que el tamaño de cada subproblema sea lo más pequeño posible, es decir, **maximizar** b .
 - que el número de subproblemas generados sea lo más pequeño posible, es decir, **minimizar** a .
 - que el coste de la parte no recursiva sea lo más pequeño posible, es decir **minimizar** k .
- ★ La recurrencia puede utilizarse para **anticipar** el coste que resultará de la solución DV, sin tener por qué completar todos los detalles. Si el coste sale igual o peor que el de un algoritmo ya existente, entonces no merecerá la pena aplicar DV.

2. Ejemplos de aplicación del esquema con éxito

- ★ Algunos de los algoritmos recursivos vistos hasta ahora encajan perfectamente en el esquema DV.
- ★ La **búsqueda binaria** en un vector ordenado vista en el Cap. 4 es un primer ejemplo. En este caso, la operación *descomponer* selecciona una de las dos mitades del vector y la operación *combinar* es vacía. Obteníamos los siguientes parámetros de coste:

$b = 2$ Tamaño mitad del subvector a investigar en cada llamada recursiva.

$a = 1$ Un subproblema a lo sumo.

$k = 0$ Coste constante de la parte no recursiva.

dando un coste total $O(\log n)$.

- ★ La **ordenación mediante mezcla** o *mergesort* también responde al esquema: la operación *descomponer* divide el vector en dos mitades y la operación *combinar* mezcla las dos mitades ordenadas en un vector final. Los parámetros del coste son:

$b = 2$ Tamaño mitad de cada subvector.

$a = 2$ Siempre se generan dos subproblemas.

$k = 1$ Coste lineal de la parte no recursiva (la mezcla).

dando un coste total $O(n \log n)$.

- ★ La **ordenación rápida** o *quicksort*, considerando solo el caso mejor, también responde al esquema. La operación *descomponer* elige el pivote, particiona el vector con respecto a él y lo divide en dos mitades. La operación *combinar* en este caso es vacía. Los parámetros del coste son:

$b = 2$ Tamaño mitad de cada subvector.

$a = 2$ Siempre se generan dos subproblemas.

$k = 1$ Coste lineal de la parte no recursiva (la partición).

dando un coste total $O(n \log n)$.

- ★ La comprobación en un vector v estrictamente ordenado de si **existe un índice i tal que $v[i] = i$** (ver la sección de problemas del Cap. 4) sigue un esquema similar al de la búsqueda binaria:

$b = 2$ Tamaño mitad del subvector a investigar en cada llamada recursiva.

$a = 1$ Un subproblema a lo sumo.

$k = 0$ Coste constante de la parte no recursiva.

dando un coste total $O(\log n)$.

- ★ Un problema históricamente famoso es el de la solución DV a la transformada discreta de Fourier (DFT), dando lugar al algoritmo conocido como **transformada rápida de Fourier**, o FFT (J.W. Cooley y J.W. Tukey, 1965). La transformada discreta convierte un conjunto de muestras de amplitud de una señal, en el conjunto de frecuencias que resultan del análisis de Fourier de la misma. Esta transformación y su inversa (que se realiza utilizando el mismo algoritmo DFT) tienen gran interés práctico pues permiten filtrar frecuencias indeseadas (p.e. ruido) y mejorar la calidad de las señales de audio o de vídeo.

La transformada en esencia multiplica una matriz $n \times n$ de números complejos por un vector de longitud n de coeficientes reales, y produce otro vector de la misma longitud. El algoritmo clásico realiza esta tarea del modo obvio y tiene un coste $O(n^2)$. La FFT descompone de un cierto modo el vector original en dos vectores de tamaño $n/2$, realiza la FFT de cada uno, y luego combina los resultados de tamaño $n/2$ para producir un vector de tamaño n . Las dos partes no recursivas tienen coste lineal, dando lugar a un algoritmo FFT de coste $O(n \log n)$. El algoritmo se utilizó por primera vez para analizar un temblor de tierra que tuvo lugar en Alaska en 1964. El algoritmo clásico empleó 26 minutos en analizar la muestra, mientras que la FFT de Cooley y Tukey lo hizo en 6 segundos.

3. Problema de selección

- ★ Dado un vector v de n elementos que se pueden ordenar y un entero $1 \leq k \leq n$, el *problema de selección* consiste en encontrar el k -ésimo menor elemento.
- ★ El problema de encontrar la mediana de un vector es un caso particular de este problema en el que se busca el elemento $\lceil n/2 \rceil$ -ésimo del vector en el caso de estar ordenado (en los vectores de C++ corresponde a la posición $(n - 1) \div 2$).
- ★ Una primera idea para resolver el problema consiste en ordenar el vector y tomar el elemento $v[k]$, lo cual tiene la complejidad del algoritmo de ordenación utilizado. Nos preguntamos si podemos hacerlo más eficientemente.
- ★ Otra posibilidad es utilizar el algoritmo *partición* que vimos en el Capítulo 4 con algún elemento del vector:
 - Si la posición p donde se coloca el pivote es igual a k , entonces $v[p]$ es el elemento que estamos buscando.
 - Si $k < p$ entonces podemos pasar a buscar el k -ésimo elemento en las posiciones anteriores a p , ya que en ellas se encuentran los elementos menores o iguales que $v[p]$ y $v[p]$ es el p -ésimo elemento del vector.
 - Si $k > p$ entonces podemos pasar a buscar el k -ésimo elemento en las posiciones posteriores a p , ya que en ellas se encuentran los elementos mayores o iguales que $v[p]$ y $v[p]$ es el p -ésimo elemento del vector.

- ★ Al igual que hemos hecho en anteriores ocasiones generalizamos el problema añadiendo dos parámetros adicionales a y b tales que $0 \leq a \leq b \leq \text{long}(v) - 1$, que nos indican la parte del vector que nos interesa en cada momento. La llamada inicial que deseamos es `seleccion(v, 0, long(v) - 1, k)`.
- ★ En esta versión del algoritmo la posición k es una posición absoluta dentro del vector. Se puede escribir una versión alternativa en la que k hace referencia a la posición relativa dentro del subvector que se está tratando.

```
TElem seleccion1(TElem v[], int a, int b, int k) {
//Pre: 0<=a<=b<=long(v)-1 && a<=k<=b
    int p;
    if (a == b) return v[a];
    else {
        particion(v, a, b, p);
        if (k == p) return v[p];
        else if (k < p) return seleccion1(v, a, p-1, k);
        else return seleccion1(v, p+1, b, k);
    }
}
```

- ★ El caso peor de este algoritmo se da cuando el pivote queda siempre en un extremo del subvector correspondiente y la llamada recursiva se hace con todos los elementos menos el pivote: por ejemplo si el vector está ordenado y le pido el último elemento. En dicho caso el coste está en $O(n^2)$ siendo $n = b - a + 1$ el tamaño del vector. La situación es similar a la del algoritmo de ordenación rápida.
- ★ Nos preguntamos qué podemos hacer para asegurarnos de que el tamaño del problema se divida aproximadamente por la mitad. Si en lugar de usar el primer elemento del vector como pivote usásemos la mediana del vector, entonces solamente tendríamos una llamada recursiva de la mitad de tamaño, lo que nos da un coste en $O(n)$ siendo n el tamaño del vector.
- ★ Pero resulta que el problema de la mediana es un caso particular del problema que estamos intentando resolver y del mismo tamaño. Por ello, nos vamos a conformar con una aproximación suficientemente buena de la mediana, conocida como *mediana de las medianas*, con la esperanza de que sea suficiente para tener un coste mejor.
- ★ Para calcularla se divide el vector en trozos consecutivos de 5 elementos, y se calcula directamente la mediana para cada uno de ellos. Después, se calcula recursivamente la mediana de esas $n \text{ div } 5$ medianas mediante el algoritmo de selección. Con este pivote se puede demostrar que el caso peor anterior ya no puede darse.
- ★ Para implementar este algoritmo vamos a definir una versión mejorada de partición:
 - Recibe como argumento el pivote con respecto al cual queremos hacer la partición, con lo que es más general. Así podemos calcular primero la mediana de las medianas y dársela después a partición.
 - En lugar de devolver solamente una posición p vamos a devolver dos posiciones p y q que delimitan la parte de los elementos que son estrictamente menores que el pivote (desde a hasta $p - 1$), las que son iguales al pivote (desde p hasta q) y las que son estrictamente mayores (desde $q + 1$ hasta b). De esta forma, si el pivote se repite muchas veces el tamaño se reduce más.

- Este algoritmo también se puede usar en el de la ordenación rápida. Así, por ejemplo, si todos los elementos del vector son iguales, con el anterior algoritmo de partición se tiene coste $O(n \log n)$ mientras con el nuevo, descartando la parte de los que son iguales al pivote, tenemos coste en $O(n)$.
- Para implementarlo usamos tres índices p, k, q : los dos primeros se mueven hacia la derecha y el tercero hacia la izquierda. El invariante nos indica que los elementos en $v[a..p-1]$ son estrictamente menores que el pivote, los de $v[p..k-1]$ son iguales al pivote y los de $v[q+1..b]$ son estrictamente mayores. La parte $v[k..q]$ está sin explorar hasta que $k = q + 1$ en cuyo caso el bucle termina y tenemos lo que queremos.
- En cada paso se compara $v[k]$ con el pivote: si es igual, está bien colocado y se incrementa k ; si es menor se intercambia con $v[p]$ para ponerlo junto a los menores y se incrementan los índices p y k , ya que $v[i]$ es igual al pivote; si es mayor se intercambia con $v[q]$ para ponerlo junto a los mayores y se decrementa la q .

```

void particion2(TElem v[], int a, int b, TElem pivote, int& p, int& q) {
//PRE: 0<=a<=b<=long(v)-1
    int k;
    TElem aux;
    p = a; k = a; q = b;

    //INV: a<=p<=k<=q+1<=b+1<=long(v)
    //      los elementos desde a hasta p-1 son < pivote
    //      los elementos desde p hasta k-1 son = pivote
    //      los elementos desde q+1 hasta b son > pivote
    while (k <= q) {
        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;
        }
    }
}
//POST: los elementos desde a hasta p-1 son < pivote
//      los elementos desde p hasta q son = pivote
//      los elementos desde q+1 hasta b son > pivote
}

```

★ Por tanto, los pasos del nuevo algoritmo, *seleccion2*, son:

1. calcular la mediana de cada grupo de 5 elementos. En total $n \text{ div } 5$ medianas, y cada una se puede calcular en tiempo constante: ordenar los 5 elementos y quedarnos con el tercero. Para no usar espacio adicional, dichas medianas se trasladan al principio del vector.
2. calcular la mediana de las medianas, mm , con una llamada recursiva a *seleccion2* con $n \text{ div } 5$ elementos.
3. llamar a *particion2* (v, a, b, mm, p, q), utilizando como pivote mm .

4. hacer una distinción de casos similar a la de `seleccion1`:

```

if ((k >= p) && (k <= q))
    return mm;
else if (k < p)
    return seleccion2(v,a,p-1,k);
else
    return seleccion2(v,q+1,b,k);

```

- ★ Es necesario elegir adecuadamente los casos base, ya que si hay 12 elementos o menos, es decir $b - a + 1 \leq 12$, es más costoso seguir el proceso recursivo que ordenar directamente y tomar el elemento k .
- ★ Con estas ideas el algoritmo queda:

```

TElem seleccion2(TElem v[], int a, int b, int k){
//0<=a<=b<=long(v)-1 && a<=k<=b
    int l, p, q, s, pm, t;
    TElem aux,mm;

    t = b-a+1;
    if (t <= 12){ // Umbral base = 12
        ordenarInsercion(v,a,b);
        return v[k];
    } else {
        s = t/5;
        for (l = 1; l <= s; l++){
            ordenarInsercion(v,a+5*(l-1),a+5*l-1);
            pm = a+5*(l-1)+(5/2);
            aux = v[a+l-1];
            v[a+l-1] = v[pm];
            v[pm] = aux;
        }
        mm = seleccion2(v,a,a+s-1,a+(s-1)/2);
        particion2(v,a,b,mm,p,q);
        if ((k >= p) && (k <= q)) return mm;
        else if (k<p) return seleccion2(v,a,p-1,k);
        else return seleccion2(v,q+1,b,k);
    }
}
//POST: v[k] es mayor o igual que v[0..k-1] y
// menor o igual que v[k+1..long(v)-1]

```

donde *ordenarInsercion* es una versión del algoritmo de ordenación por inserción más general que el que vimos en el Capítulo 3, ya que podemos indicar el trozo del vector que deseamos ordenar.

- ★ La llamada inicial es `seleccion2(v, 0, long(v)-1, k)`.
- ★ Se puede demostrar, por inducción constructiva, que el tiempo requerido por *seleccion2* en el caso peor es lineal en $n = b - a + 1$ Brassard y Bratley (1997).

4. Organización de un campeonato

- ★ Se tienen n participantes para un torneo de ajedrez y hay que organizar un calendario para que todos jueguen contra todos de forma que:
 1. Cada participante juegue exactamente una partida con cada uno de los $n - 1$ restantes.
 2. Cada participante juegue a lo sumo una partida diaria.
 3. El torneo se complete en el menor número posible de días.

En este tema veremos una solución para el caso, más sencillo, en que n es potencia de 2.

- ★ Es fácil ver que el número de parejas distintas posibles es $\frac{1}{2}n(n - 1)$. Como n es par, cada día pueden jugar una partida los n participantes formando con ellos $\frac{n}{2}$ parejas. Por tanto se necesita un mínimo de $n - 1$ días para que jueguen todas las parejas.
- ★ Una posible forma de representar la solución al problema es en forma de matriz de n por n , donde se busca rellenar, en cada celda a_{ij} , el día en que se enfrentarán entre sí los contrincantes i y j , con $j < i$. Es decir, tratamos de rellenar, con fechas de encuentros, el área bajo la diagonal de esta matriz; sin que en ninguna fila o columna haya días repetidos.
- ★ Se ha de planificar las parejas de cada día, de tal modo que al final todos jueguen contra todos sin repetir ninguna partida, ni descansar innecesariamente.
- ★ Podemos ensayar una solución DV según las siguientes ideas (ver Figura 1):
 - Si n es suficientemente grande, dividimos a los participantes en dos grupos disjuntos A y B , cada uno con la mitad de ellos.
 - Se resuelven recursivamente dos torneos más pequeños: el del conjunto A jugando sólo entre ellos, y el del conjunto B también jugando sólo entre ellos. En estos sub-torneos las condiciones son idénticas a las del torneo inicial por ser n una potencia de 2; con la salvedad de que se pueden jugar ambos en paralelo.
 - Después se planifican partidas en las que un participante pertenece a A y el otro a B . En estas partidas, que no se pueden solapar con los sub-torneos, hay que rellenar todas las celdas de la matriz correspondiente.
- ★ Esta última parte se puede resolver fácilmente fila por fila, rotando, en cada nueva fila, el orden de las fechas disponibles. Como hay que rellenar $\frac{n}{2} \cdot \frac{n}{2}$ celdas, el coste de esta fase está en $\Theta(n^2)$.
- ★ Los casos base, $n = 2$ o $n = 1$, se resuelven trivialmente en tiempo constante.
- ★ Esta solución nos da pues los parámetros de coste $a = 2$, $b = 2$ y $k = 2$, que conducen a un coste esperado de $\Theta(n^2)$. No puede ser menor puesto que la propia planificación consiste en rellenar $\Theta(n^2)$ celdas. Pasamos entonces a precisar los detalles.

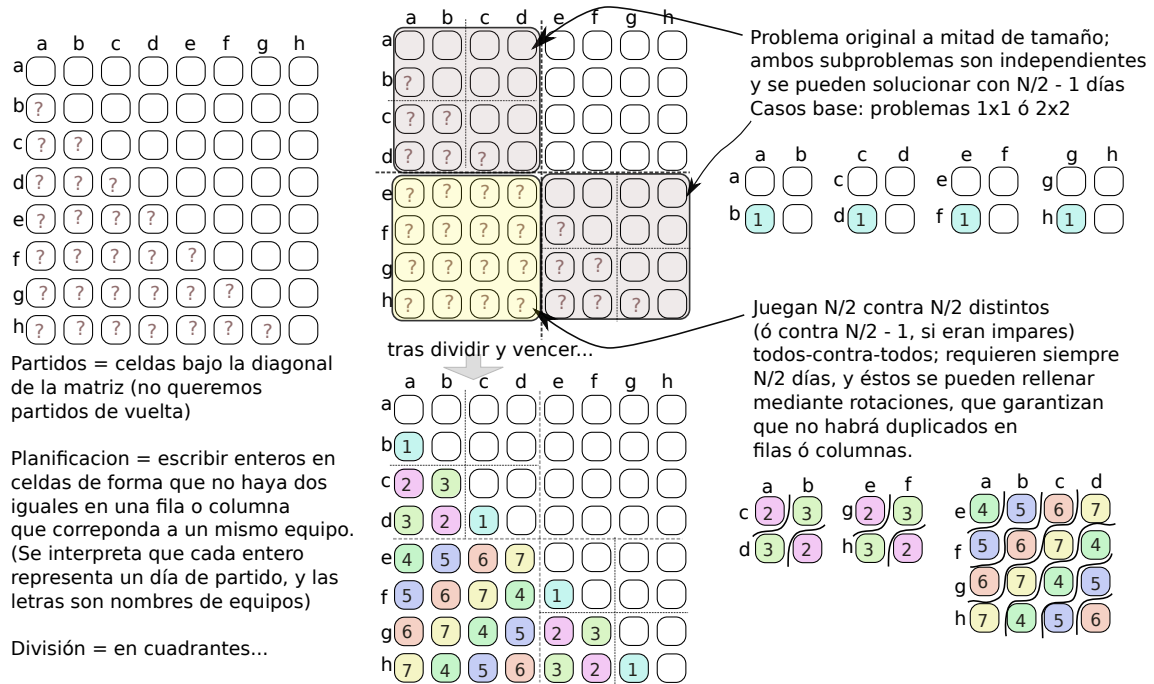


Figura 1: Solución gráfica del problema del torneo

4.1. Implementación

- ★ Usaremos una matriz cuadrada declarada como `int a[MAX][MAX]` (donde `MAX` es una constante entera) para almacenar la solución, inicializada con ceros. La primera fecha disponible será el día 1.
- ★ De forma similar a ejemplos anteriores la función recursiva, llamada *rellena*, recibe dos parámetros adicionales, *c* y *f* que delimitan el trozo de la matriz que estamos rellenando, y que por tanto se inicializan respectivamente con 0 y $num - 1$, siendo num el número de participantes. Se asume que $dim = f - c + 1$ es potencia de 2.
- ★ En el caso base en que solo haya un equipo ($dim = 1$) no se hace nada. Si hay dos equipos ($dim = 2$), juegan el día 1.
- ★ El cuadrante inferior izquierdo representa los partidos entre los equipos de los dos grupos. El primer día disponible es $mitad = dim/2$ y hacen falta $mitad$ días para que todos jueguen contra todos. Las rotaciones se consiguen con la fórmula $mitad + (i + j) \% mitad$.

```
void rellena(int a[MAX][MAX], int c, int f) {
//PRE:  $\exists k : f - c + 1 = 2^k \wedge 0 \leq c \leq f < MAX \wedge \forall i, j : c \leq i, j \leq f : a[i][j] = 0$ 
    int dim = f - c + 1;
    int mitad = dim / 2;
    // si dim = 1 nada, la diagonal principal no se rellena
    if (dim == 2) a[c + 1][c] = 1;
    else if (dim > 2) {
        rellena(a, c, c + mitad - 1); //cuadrante superior izquierdo
        rellena(a, c + mitad, f); //cuadrante inferior derecho
        for (int i = c + mitad; i <= f; i++) //cuadrante inferior izquierdo
            for (int j = c; j <= c + mitad - 1; j++)
                a[i][j] = mitad + (i + j) % mitad;
    }
}
```

```

    }

//POST:  $\forall i, j : c \leq j < i \leq f : 1 \leq a[i][j] \leq f - c \wedge$ 
//       $\forall i : c < i \leq f : (\forall j, k : c \leq j < k < i : a[i][j] \neq a[i][k]) \wedge$ 
//       $\forall j : c \leq j < f : (\forall i, k : j < i < k \leq f : a[i][j] \neq a[k][j]) \wedge$ 
//       $\forall i : c \leq i \leq f : (\forall j : c \leq j < i : (\forall k : i < k \leq f : a[i][j] \neq a[k][i]))$ 
}

```

5. El problema del par más cercano

- ★ Dada una nube de n puntos en el plano, $n \geq 2$, se trata de encontrar el par de puntos cuya distancia euclídea es menor (si hubiera más de un par con esa distancia mínima, basta con devolver uno de ellos). El problema tiene interés práctico. Por ejemplo, en un sistema de control del tráfico aéreo, el par más cercano nos informa del mayor riesgo de colisión entre dos aviones.
- ★ Dados dos puntos, $p_1 = (x_1, y_1)$ y $p_2 = (x_2, y_2)$, su distancia euclídea viene dada por $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. El algoritmo de “fuerza bruta” calcularía la distancia entre todo posible par de puntos, y devolvería el mínimo de todas ellas. Como hay $\frac{1}{2}n(n-1)$ pares posibles, el coste resultante sería **cuadrático**.
- ★ El enfoque DV trataría de encontrar el par más cercano a partir de los pares más cercanos de conjuntos de puntos que sean una fracción del original. Una posible estrategia es:

Dividir Crear dos nubes de puntos de tamaño mitad. Podríamos ordenar los puntos por la coordenada x y tomar la primera mitad como nube izquierda I , y la segunda como nube derecha D . Determinamos una línea vertical imaginaria l tal que todos los puntos de I están sobre l , o a su izquierda, y todos los de D están sobre l , o a su derecha.

Conquistar Resolver recursivamente los problemas I y D . Sean δ_I y δ_D las respectivas distancias mínimas encontradas y sea $\delta = \min(\delta_I, \delta_D)$.

Combinar El par más cercano de la nube original, o bien es el par con distancia δ , o bien es un par compuesto por un punto de la nube I y otro punto de la nube D . En ese caso, ambos puntos se hallan a lo sumo a una distancia δ de l . La operación *combinar* debe investigar los puntos de dicha banda vertical.

- ★ Antes de seguir con los detalles, debemos investigar el coste esperado de esta estrategia. Como el algoritmo fuerza-bruta tiene coste $\Theta(n^2)$, trataremos de conseguir un coste $\Theta(n \log n)$ en el caso peor. Sabemos por la experiencia de algoritmos como *mergesort* que ello exige unos parámetros $a = 2$, $b = 2$, $k = 1$, en decir tan solo podemos consumir un coste lineal en las operaciones de dividir y combinar.
- ★ La ordenación de los puntos por la coordenada de x se puede realizar una sola vez al principio (es decir, fuera del algoritmo recursivo DV) con un coste $\Theta(n \log n)$ en el caso peor, lo que es admisible para mantener nuestro coste total. Una vez ordenada la nube, la división en dos puede conseguirse con coste constante o lineal, dependiendo de si se utilizan vectores o listas como estructuras de datos de la implementación.

- ★ Una vez resueltos los dos subproblemas, se pueden filtrar los puntos de I y D para conservar sólo los que estén en la banda vertical de anchura 2δ y centrada en l . El filtrado puede hacerse con coste lineal tanto en tiempo como en espacio adicional. Llamemos B_I y B_D a los puntos de dicha banda respectivamente a la izquierda y a la derecha de l .
- ★ Para investigar si en la banda hay dos puntos a distancia menor que δ , aparentemente debemos calcular la distancia de cada punto de B_I a cada punto de B_D . Es fácil construir nubes de puntos en las que todos ellos caigan en la banda tras el filtrado, de forma que en el caso peor podríamos tener $|B_I| = |B_D| = \frac{n}{2}$. En ese caso, el cálculo de la distancia mínima entre los puntos de la banda sería cuadrático, y el coste total del algoritmo DV también.
- ★ Demostraremos que basta ordenar por la coordenada y el conjunto de puntos $B_I \cup B_D$ y después recorrer la lista ordenada comparando cada punto **con los 7 que le siguen**. Si de este modo no se encuentra una distancia menor que δ , concluimos que todos los puntos de la banda distan más entre sí. Este recorrido es claramente de coste lineal.
- ★ Suponiendo que esta estrategia fuera correcta, todavía quedaría por resolver la ordenación por la coordenada y . Si ordenáramos $B_I \cup B_D$ en cada llamada recursiva, gastaríamos un coste $\Theta(n \log n)$ en cada una, lo que conduciría un coste total en $\Theta(n \log^2 n)$.
- ★ Recordando la técnica de los **resultados acumuladores** explicada en la Sección 4.2 de estos apuntes, podemos exigir que cada llamada recursiva devuelva un resultado extra: la lista de sus puntos ordenada por la coordenada y . Este resultado puede propagarse hacia arriba del árbol de llamadas con un coste lineal, porque basta aplicar el algoritmo de mezcla de dos listas ordenadas. La secuencia de acciones de la operación *combinar* es entonces la siguiente:
 1. Realizar la mezcla ordenada de las dos listas de puntos devueltas por las llamadas recursivas. Esta lista se devolverá al llamante.
 2. Filtrar la lista resultante, conservando los puntos a una distancia de la línea divisoria l menor o igual que δ . Llamemos B a la lista filtrada.
 3. Recorrer B calculando la distancia de cada punto a los 7 que le siguen, comprobando si aparece una distancia menor que δ .
 4. Devolver los dos puntos a distancia mínima, considerando los tres cálculos realizados: parte izquierda, parte derecha y lista B .

5.1. Corrección

- ★ Consideremos un rectángulo cualquiera de anchura 2δ y altura δ centrado en la línea divisoria l (ver Figura 2). Afirmamos que, contenidos en él, puede haber a lo sumo 8 puntos de la nube original. En la mitad izquierda puede haber a lo sumo 4, y en caso de haber 4, situados necesariamente en sus esquinas, y en la mitad derecha otros 4, también en sus esquinas. Ello es así porque, por hipótesis de inducción, los puntos de la nube izquierda están separados entre sí por una distancia de al menos δ , e igualmente los puntos de la nube derecha entre sí. En la línea divisoria podrían coexistir hasta dos puntos de la banda izquierda con dos puntos de la banda derecha.

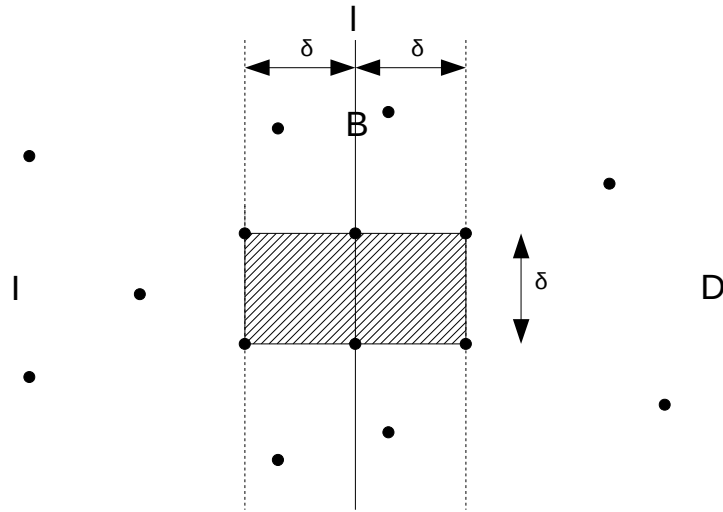


Figura 2: Razonamiento de corrección del problema del par más cercano

- ★ Si colocamos dicho rectángulo con su base sobre el punto p_1 de menor coordenada y de B , estamos seguros de que a los sumo los 7 siguientes puntos de B estarán en dicho rectángulo. A partir del octavo, él y todos los demás distarán más que δ de p_1 . Desplazando ahora el rectángulo de punto a punto, podemos repetir el mismo razonamiento. No es necesario investigar los puntos con menor coordenada y que el punto en curso, porque esa comprobación ya se hizo cuando se procesaron dichos puntos.
- ★ Elegimos como caso base de la inducción $n < 4$. De este modo, al subdividir una nube con $n \geq 4$ puntos, nunca generaremos problemas con un solo punto.

5.2. Implementación

- ★ Definimos un punto como una pareja de números reales.

```
struct Punto
{ double x;
  double y; };

```

La entrada al algoritmo será un vector p de puntos y los límites c y f de la nube que estamos mirando. El vector de puntos no se modificará en ningún momento y se supone que está ordenado respecto a la coordenada x .

- ★ La solución

```
void parMasCercano(Punto p[], int c, int f, int indY[], int& ini,
                  double& d, int& p1, int& p2)

```

constará de:

- Los límites c y f . Las llamadas correctas cumplen $0 \leq c \wedge f \leq \text{long}(v) - 1 \wedge f \geq c + 1$.

- La distancia d entre los puntos más cercanos.
- Los puntos p_1 y p_2 más cercanos.
- Un vector de posiciones $indY$ y un índice inicial ini que representa cómo se ordenan los elementos de p con respecto a la coordenada y . El índice inicial ini nos indica que el punto $p[ini]$ es el que tiene la menor coordenada y . El vector $indY$ contiene en cada posición la posición del siguiente punto en la ordenación, siendo -1 el valor utilizado para indicar que no hay siguiente. Por ejemplo, si el vector de puntos es:

$$\{\{-0.5, 0.5\}, \{0, 3\}, \{0, 0\}, \{0, 0.25\}, \{1, 1\}, \{1.25, 1.25\}, \{2, 2\}\}$$

la variable ini valdrá 2 y el vector $indY$ será:

$$\{4, -1, 3, 0, 5, 6, 1\}$$

Es decir:

- el elemento con menor y es el punto $p[2]$;
- como $indY[2] = 3$ el siguiente es $p[3]$;
- como $indY[3] = 0$, el siguiente es $p[0]$;
- como $indY[0] = 4$, el siguiente es $p[4]$;
- como $indY[4] = 5$, el siguiente es $p[5]$;
- como $indY[5] = 6$, el siguiente es $p[6]$;
- como $indY[6] = 1$, el siguiente es $p[1]$;
- como $indY[1] = -1$, ya no hay más puntos

- ★ Usaremos las siguientes funciones auxiliares:

```
double absolute(double x) {
    if (x >= 0) return x;
    else return -x;
}

double distancia(Punto p1, Punto p2) {
    return (sqrt((p1.x-p2.x)*(p1.x-p2.x) + (p1.y-p2.y)*(p1.y-p2.y)));
}

double minimo(double x, double y) {
    double z;
    if (x <= y) z = x; else z = y;
    return z;
}
```

- ★ Los casos base, cuando hay 2 o 3 puntos los resuelve la función

```
void solucionDirecta(Punto p[], int c, int f, int indY[], int& ini,
                    double& d, int& p1, int& p2) {
    double d1, d2, d3;
    if (f == c+1) {
        d = distancia(p[c], p[f]);
        if ((p[c].y) <= (p[f].y)) {
            ini = c; indY[c] = f; indY[f] = -1; p1 = c; p2 = f;
        } else {
```

```

        ini = f; indY[f] = c; indY[c] = -1; p1 = f; p2 = c;
    }
    else if (f == c+2){
        // Menor distancia y puntos que la producen
        d1 = distancia(p[c],p[c+1]);
        d2 = distancia(p[c],p[c+2]);
        d3 = distancia(p[c+1],p[c+2]);
        d = minimo(minimo(d1,d2),d3);
        if (d == d1) {p1 = c; p2 = c+1;}
        else if (d == d2) {p1 = c; p2 = c+2;}
        else {p1 = c+1; p2 = c+2;}

        //Ordenar
        if (p[c].y <= p[c+1].y){
            if (p[c+1].y <= p[c+2].y){
                ini = c; indY[c] = c+1; indY[c+1] = c+2; indY[c+2] = -1;
            } else if (p[c].y <= p[c+2].y){
                ini = c; indY[c] = c+2; indY[c+2] = c+1; indY[c+1] = -1;
            } else{
                ini = c+2; indY[c+2] = c; indY[c] = c+1; indY[c+1] = -1;
            }
        } else{
            if (p[c+1].y > p[c+2].y){
                ini = c+2; indY[c+2] = c+1; indY[c+1] = c; indY[c] = -1;
            } else if (p[c].y > p[c+2].y){
                ini = c+1; indY[c+1] = c+2; indY[c+2] = c; indY[c] = -1;
            } else{
                ini = c+1; indY[c+1] = c; indY[c] = c+2; indY[c+2] = -1;
            }
        }
    }
}

```

- ★ El método *mezclaOrdenada* recibe en *indY* dos listas de enlaces que comienzan en *ini1* y *ini2* que representan respectivamente la ordenación con respecto a *y* de los puntos de la nube izquierda y derecha, y las mezcla para obtener una única lista de enlaces con todos los puntos de la nube.

```

void mezclaOrdenada(Punto p[],int ini1, int ini2, int indY[], int& ini){
    int i = ini1;
    int j = ini2;
    int k;
    if (p[i].y <= p[j].y){
        ini = ini1; k = ini1; i = indY[i];
    } else{
        k = ini2; ini = ini2; j = indY[j];
    }
    while ((i != -1) && (j != -1)){
        if (p[i].y <= p[j].y){
            indY[k] = i; k = i; i = indY[i];
        } else{
            indY[k] = j; k = j; j = indY[j];
        }
    }
    if (i == -1) indY[k] = j;
    else indY[k] = i;
}

```

 }

En el ejemplo de antes, después de las dos llamadas recursivas tendríamos que $ini1 = 2$, $ini2 = 4$ y el vector $indY$ es:

$$\{1, -1, 3, 0, 5, 6, -1\}$$

representando dos listas de puntos $p[2]$, $p[3]$, $p[0]$, $p[1]$ y por otro lado $p[4]$, $p[5]$, $p[6]$. Después de ejecutar *mezclaOrdenada* obtenemos el resultado de arriba. Este método se puede utilizar igualmente en una versión del algoritmo *mergesort* que devuelva un vector de enlaces.

★ Así el algoritmo queda:

```

void parMasCercano(Punto p[], int c, int f, int indY[], int& ini,
                  double& d, int& p1, int& p2){
    int m; int i, j, ini1, ini2, p11, p12, p21, p22;
    double d1, d2;

    if (f-c+1 < 4)
        solucionDirecta(p, c, f, indY, ini, d, p1, p2);
    else{
        m = (c+f)/2;
        parMasCercano(p, c, m, indY, ini1, d1, p11, p12);
        parMasCercano(p, m+1, f, indY, ini2, d2, p21, p22);

        if (d1 <= d2){
            d = d1; p1 = p11; p2 = p12;
        } else{
            d = d2; p1 = p21; p2 = p22;
        }

        //Mezcla ordenada por la y
        mezclaOrdenada(p, ini1, ini2, indY, ini);

        //Filtrar la lista
        i = ini;
        while (absolute(p[m].x-p[i].x)>d) i = indY[i];

        int iniA = i;
        int indF[f-c+1];
        for (int l = 0; l <= f-c+1; l++) indF[l] = -1;
        int k = iniA - c;
        while (i != -1){
            if (absolute(p[m].x-p[i].x) <= d)    {indF[k] = i-c; k = i-c;}
            i = indY[i];
        }

        //Calcular las distancias
        i = iniA-c;
        while (i != -1){
            int count = 0; j = indF[i];
            while (j != -1 && count < 7){
                double daux = distancia(p[i+c], p[j+c]);
                if (daux < d) {d = daux; p1 = i+c; p2 = j+c;}
                j = indF[j];
            }
        }
    }
}

```

```

        count = count+1;
    }
    i = indF[i];
}
}
}

```

- ★ La llamada inicial es:

```
parMasCercano(p, 0, long(v)-1, indY, ini, d, p1, p2).
```

6. La determinación del umbral

- ★ Dado un algoritmo DV, casi siempre existe una versión asintóticamente menos eficiente pero de constantes multiplicativas más pequeñas. Le llamaremos el *algoritmo sencillo*. Eso hace que para valores pequeños de n , sea más eficiente el algoritmo sencillo que el algoritmo DV.
- ★ Se puede conseguir un algoritmo óptimo combinando ambos algoritmos de modo inteligente. El aspecto que tendría el algoritmo compuesto es:

```

Solucion divideYvenceras (Problema x, int n){
    if (n <= n_0)
        return algoritmoSencillo(x)
    else /* n > n_0 */ {
        descomponer x
        llamadas recursivas a divideYvenceras
        y = combinar resultados
        return y;
    }
}

```

- ★ Es decir, se trata de convertir en casos base del algoritmo recursivo los problemas que son *suficientemente* pequeños. Nos planteamos cómo determinar **el umbral** n_0 a partir del cual compensa utilizar el algoritmo sencillo con respecto a continuar subdividiendo el problema.
- ★ La determinación del umbral es un tema fundamentalmente **experimental**, depende del computador y lenguaje utilizados, e incluso puede no existir un óptimo único sino varios en función del tamaño del problema.
- ★ A pesar de eso, se puede hacer un estudio teórico del problema para encontrar un umbral aproximado. Para fijar ideas, centrémosnos en el problema de encontrar el par más cercano y escribamos su recurrencia con constantes multiplicativas (suponemos n potencia de 2):

$$T_1(n) = \begin{cases} c_0 & \text{si } 0 \leq n \leq 3 \\ 2T_1(n/2) + c_1n & \text{si } n \geq 4 \end{cases}$$

Si desplegamos esta recurrencia y la resolvemos exactamente, la expresión de coste resulta ser:

$$T_1(n) = c_1n \log n + \left(\frac{1}{2}c_0 - c_1\right)n$$

- ★ Por otra parte, el algoritmo sencillo tendrá un coste $T_2(n) = c_2 n^2$. Las constantes c_0 , c_1 y c_2 dependen del lenguaje y de la máquina subyacentes, y han de ser determinadas experimentalmente para cada instalación.
- ★ Aparentemente, para encontrar el umbral hay que resolver la ecuación $T_1(n) = T_2(n)$, es decir encontrar un n_0 que satisfaga:

$$c_1 n \log n + \left(\frac{1}{2}c_0 - c_1\right)n = c_2 n^2$$

Sin embargo, este planteamiento es **incorrecto** porque el coste del algoritmo DV está calculado subdividiendo n hasta los casos base. Es decir, estamos comparando el algoritmo DV puro con el algoritmo sencillo puro y lo que queremos saber es cuándo subdividir es más costoso que no subdividir.

- ★ La ecuación que necesitamos es la siguiente:

$$2T_2(n/2) + c_1 n = c_2 n^2 = T_2(n)$$

que expresa que en una llamada recursiva al algoritmo DV decidimos subdividir **por última vez** porque es tan costoso subdividir como no hacerlo. Nótese que el coste de las dos llamadas internas está calculado con el algoritmo sencillo, lo que confirma que esta subdivisión es la última que se hace.

- ★ Resolviendo esta ecuación obtenemos:

$$2c_2 \left(\frac{n}{2}\right)^2 + c_1 n = c_2 n^2 \Rightarrow n_0 = \frac{2c_1}{c_2}$$

Para $n > n_0$, la expresión de la izquierda crece más despacio que la de la derecha y merece la pena subdividir. Para valores menores que n_0 , la expresión de la derecha es menos costosa.

- ★ Como sabemos, c_1 mide el número de operaciones elementales que hay que hacer con cada punto de la nube de puntos en la parte no recursiva del algoritmo DV. Es decir la suma por punto de dividir la lista en dos, mezclar las dos mitades ordenadas, filtrar los puntos de la banda y recorrer la misma, comparando cada punto con otros siete. Por su parte, c_2 mide el coste elemental de cada una de las n^2 operaciones del algoritmo sencillo. Este coste consiste en esencia en la mitad de calcular la distancia entre dos puntos y comparar con el mínimo en curso.
Supongamos que, una vez medidas experimentalmente, obtenemos $c_1 = 32c_2$. Ello nos daría un umbral $n_0 = 64$.
- ★ Es interesante escribir y resolver la recurrencia del algoritmo híbrido así conseguido y comparar el coste con el del algoritmo DV original:

$$T_3(n) = \begin{cases} c_2 n^2 & \text{si } n \leq 64 \\ 2T_3(n/2) + c_1 n & \text{si } n > 64 \end{cases}$$

Si desplegamos i veces, obtenemos:

$$T_3(n) = 2^i T_3\left(\frac{n}{2^i}\right) + i c_1 n$$

que alcanza el caso base cuando $\frac{n}{2^i} = 2^6 \Rightarrow i = \log n - 6$. Entonces sustituimos i :

$$\begin{aligned} T_3(n) &= \frac{n}{2^6} T_3(2^6) + c_1(\log n - 6)n \\ &= c_1 n \log n + c_2 \frac{n}{2^6} 2^{12} - 6c_1 n \\ &= c_1 n \log n - 4c_1 n \end{aligned}$$

- ★ Comparando el coste $T_3(n)$ del algoritmo híbrido con el coste $T_1(n)$ del algoritmo DV puro, se aprecia una diferencia importante en la constante multiplicativa del término de segundo orden.

Notas bibliográficas

En (Martí Oliet et al., 2013, Cap. 11) se repasan los fundamentos de la técnica DV y hay numerosos ejercicios resueltos. El capítulo (Brassard y Bratley, 1997, Cap. 7) también está dedicado a la técnica DV y uno de los ejemplos es el algoritmo de Karatsuba y Ofman. El ejemplo del par más cercano está tomado de (Cormen et al., 2001, Cap. 33).

Ejercicios

1. Dos amigos matan el tiempo de espera en la cola del cine jugando a un juego muy sencillo: uno de ellos piensa un número natural positivo y el otro debe adivinarlo preguntando solamente si es menor o igual que otros números. Diseñar un algoritmo eficiente para adivinar el número.
2. Desarrollar un algoritmo DV para multiplicar n número complejos usando tan solo $3(n-1)$ multiplicaciones.
3. Dados un vector $v[0..n-1]$ y un valor k , $1 \leq k \leq n$, diseñar un algoritmo DV de coste constante en espacio que trasponga las k primeras posiciones del vector con las $n-k$ siguientes. Es decir, los elementos $v[0] \dots v[k-1]$ han de copiarse a las posiciones $n-k \dots n-1$, y los elementos $v[k] \dots v[n-1]$ han de copiarse a las posiciones $0 \dots n-k-1$.
4. Dado un vector $T[1..n]$ de n elementos (que tal vez no se puedan ordenar), se dice que un elemento x es *mayoritario en T* cuando el número de veces que x aparece en T es estrictamente mayor que $N/2$.
 - a) Escribir un algoritmo DV que en tiempo $O(n \log n)$ decida si un vector $T[0..n-1]$ contiene un elemento mayoritario y devuelva tal elemento cuando exista.
 - b) Suponiendo que los elementos del vector $T[0..n-1]$ se puedan ordenar, y que podemos calcular la mediana de un vector en tiempo lineal, escribir un algoritmo DV que en tiempo lineal decida si $T[0..n-1]$ contiene un elemento mayoritario y devuelva tal elemento cuando exista. La mediana se define como el valor que ocuparía la posición $\frac{n-1}{2}$ si el vector estuviese ordenado.
 - c) Idear un algoritmo que no sea DV, tenga coste lineal, y no suponga que los elementos se pueden ordenar.
5. a) (🐞ACR295) Dado un valor x fijo, escribir un algoritmo para calcular x^n con un coste $O(\log n)$ en términos del número de multiplicaciones.

- b) Sea F la matriz $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Calcular el producto del vector $(i \ j)$ y la matriz F .
¿Qué ocurre cuando i y j son números consecutivos de la sucesión de Fibonacci?
- c) (ACR306) Utilizando las ideas de los dos apartados anteriores, desarrollar un algoritmo para calcular el n -ésimo número de Fibonacci $f(n)$ con un coste $O(\log n)$ en términos del número de operaciones aritméticas elementales.
6. En un habitación oscura se tienen dos cajones, en uno de los cuales hay n tornillos de varios tamaños, y en el otro las correspondientes n tuercas. Es necesario emparejar cada tornillo con su tuerca correspondiente, pero debido a la oscuridad no se pueden comparar tornillos con tornillos ni tuercas con tuercas, y la única comparación posible es la de intentar enroscar una tuerca en un tornillo para comprobar si es demasiado grande, demasiado pequeña, o se ajusta perfectamente al tornillo. Desarrollar un algoritmo para emparejar los tornillos con las tuercas que use $O(n \log n)$ comparaciones en promedio.
7. Dado un vector $C[0..n-1]$ de números enteros distintos, y un número entero S , se pide:
- Diseñar un algoritmo de complejidad $\Theta(n \log n)$ que determine si existen o no dos elementos de C tales que su suma sea exactamente S .
 - Suponiendo ahora ordenado el vector C , diseñar un algoritmo que resuelva el mismo problema en tiempo $\Theta(n)$.
8. Mr. Scrooge ha cobrado una antigua deuda, recibiendo una bolsa con n monedas de oro. Su olfato de usurero le asegura que una de ellas es falsa, pero lo único que la distingue de las demás es su peso, aunque no sabe si este es mayor o menor que el de las otras. Para descubrir cuál es la falsa, Mr. Scrooge solo dispone de una balanza con dos platillos para comparar el peso de dos conjuntos de monedas. En cada pesada lo único que puede observar es si la balanza queda equilibrada, si pesan más los objetos del platillo de la derecha o si pesan más los de la izquierda. Suponiendo $n \geq 3$, diseñar un algoritmo DV para encontrar la moneda falsa y decidir si pesa más o menos que las auténticas.
9. Se dice que un punto del plano $A = (a_1, a_2)$ domina a otro $B = (b_1, b_2)$ si $a_1 > b_1$ y $a_2 > b_2$. Dado un conjunto S de puntos en el plano, el rango de un punto $A \in S$ es el número de puntos que domina. Escribir un algoritmo de coste $O(n \log^2 n)$ que dado un conjunto de n puntos calcule el rango de cada uno; demostrar que su coste efectivamente es el requerido.
10. **La línea del cielo de Manhattan.** Dado un conjunto de n rectángulos (los edificios), cuyas bases descansan todas sobre el eje de abscisas, hay que determinar mediante DV la cobertura superior de la colección (la línea del cielo).
- La línea del cielo puede verse como una lista ordenada de segmentos horizontales, cada uno comenzando en la abscisa donde el segmento precedente terminó. De esta forma, puede representarse mediante una secuencia alternante de abscisas y ordenadas, donde cada ordenada indica la altura de su correspondiente segmento:

$$(x_1, y_1, x_2, y_2, \dots, x_{m-1}, y_{m-1}, x_m)$$

con $x_1 < x_2 < \dots < x_m$. Por convenio, la línea del cielo está a altura 0 hasta alcanzar x_1 , y vuelve a 0 después de x_m . Se usa la misma representación para cada rectángulo,

es decir (x_1, y_1, x_2) corresponde al rectángulo con vértices $(x_1, 0)$, (x_1, y_1) , (x_2, y_1) y $(x_2, 0)$.

11. Sea un vector $V[0..n-1]$ que contiene valores enteros positivos que se ajustan al perfil de una curva cóncava; es decir, para una cierta posición k , tal que $0 \leq k < n$, se cumple que $\forall j \in \{0..k-1\}. V[j] > V[j+1]$ y $\forall j \in \{k+1..n-1\}. V[j-1] < V[j]$ (por ejemplo el vector $V = [9, 8, 7, 3, 2, 4, 6]$). Se pide diseñar un algoritmo que encuentre la posición del mínimo en el vector (la posición 4 en el ejemplo), teniendo en cuenta que el algoritmo propuesto debe de ser más eficiente que el conocido de búsqueda secuencial del mínimo en un vector cualquiera.
12. La *envolvente convexa* de una nube de puntos en el plano es el menor polígono convexo que incluye a todos los puntos. Si los puntos se representaran mediante clavos en un tablero y extendiéramos una goma elástica alrededor de todos ellos, la forma que adoptaría la goma al soltarla sería la envolvente convexa. Dada una lista l de n puntos, se pide un algoritmo de coste $\Theta(n \log n)$ que calcule su envolvente convexa.
13. Las matrices de Hadamard H_0, H_1, H_2, \dots , se definen del siguiente modo:
 - $H_0 = (1)$ es una matriz 1×1 .
 - Para $k > 0$, H_k es la matriz $2^k \times 2^k$

$$H_k = \left(\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array} \right)$$

Si v es un vector columna de longitud $n = 2^k$, escribir un algoritmo que calcule el producto $H_k \cdot v$ en tiempo $O(n \log n)$ suponiendo que las operaciones aritméticas básicas tienen un coste constante. Justificar el coste.

14. Diseñar un algoritmo de coste en $O(n)$ que dado un conjunto S con n números, y un entero positivo $k \leq n$, determine los k números de S más cercanos a la mediana de S .
15. La p -mediana generalizada de una colección de n valores se define como la *media* de los p valores ($p+1$ si p y n tuvieran distinta paridad) que ocuparían las posiciones centrales si se ordenara la colección. Diseñar un algoritmo que resuelva el problema en un tiempo a lo sumo $O(n \log(n))$, asumiendo p constante distinto de n .
16. Se tiene un vector V de números enteros distintos, con pesos asociados p_1, \dots, p_n . Los pesos son valores no negativos y verifican que $\sum_{i=1}^n p_i = 1$. Se define la *mediana ponderada* del vector V como el valor $V[m]$, $1 \leq m \leq n$, tal que

$$\left(\sum_{V[i] < V[m]} p_i \right) < \frac{1}{2} \quad \text{y} \quad \left(\sum_{V[i] \leq V[m]} p_i \right) \geq \frac{1}{2}.$$

Por ejemplo, para $n = 5$, $V = [4, 2, 9, 3, 7]$ y $P = [0.15, 0.2, 0.3, 0.1, 0.25]$, la media ponderada es $V[5] = 7$ porque

$$\sum_{V[i] < 7} p_i = p_1 + p_2 + p_4 = 0.15 + 0.2 + 0.1 = 0.45 < \frac{1}{2}, \text{ y}$$

$$\sum_{V[i] \leq 7} p_i = p_1 + p_2 + p_4 + p_5 = 0.15 + 0.2 + 0.1 + 0.25 = 0.7 \geq \frac{1}{2}.$$

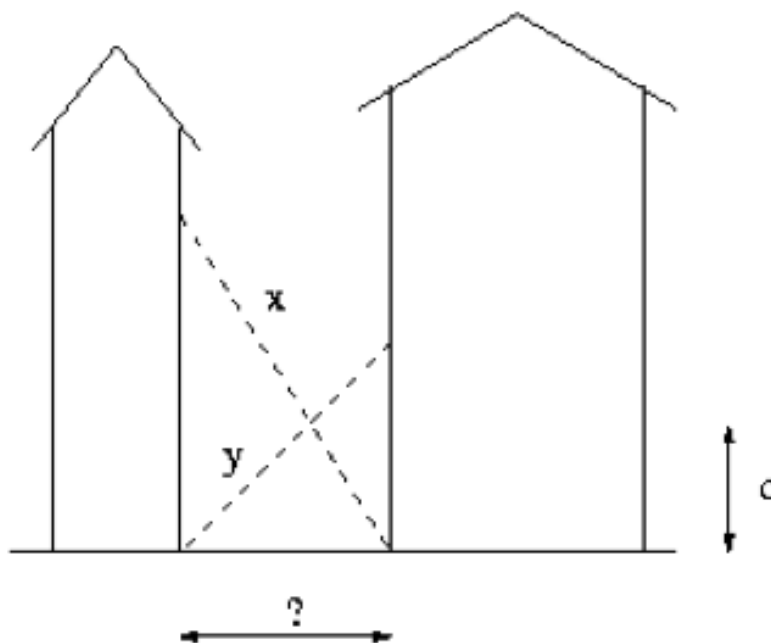
Diseñar un algoritmo de tipo *divide y vencerás* que encuentre la mediana ponderada en un tiempo lineal en el caso peor. (Obsérvese que V puede no estar ordenado.)

17. Se tienen n bolas de igual tamaño, todas ellas de igual peso salvo dos más pesadas, que a su vez pesan lo mismo. Como único medio para dar con dichas bolas se dispone de una balanza romana clásica. Diseñar un algoritmo que permita determinar cuáles son dichas bolas, con el mínimo posible de pesadas (que es logarítmico).

Indicación: Considerar también el caso en el que solo hay una bola diferente.

18. (ACR230)(Examen Junio 2013) Dado un vector de enteros, ¿cuántos intercambios haría el algoritmo de ordenación *burbuja* para ordenarlo?

19. **Escaleras cruzadas** Una calle estrecha está flanqueada por dos edificios muy altos. Se colocan dos escaleras en dicha calle como muestra el dibujo: una de ellas, de longitud x metros, colocada en la base del edificio que está en el lado derecho de la calle y apoyada sobre la fachada del edificio situado en el lado izquierdo de la calle; la otra, de longitud y metros, colocada en la base del edificio que está en el lado izquierdo de la calle y apoyada sobre la fachada del edificio situado en el lado derecho de la calle. El punto donde se cruzan ambas escaleras está a altura exactamente c metros del suelo. Se pide diseñar un algoritmo que calcule la anchura de la calle con tres decimales de precisión.



(Enunciado original en <http://uva.onlinejudge.org/external/105/10566.pdf>)

20. **Resolución de una ecuación** Dados números reales p, q, r, s, t, u tales que $0 \leq p, r \leq 20$ y $-20 \leq q, s, t \leq 0$, se desea resolver la siguiente ecuación en el intervalo $[0, 1]$ con cuatro decimales de precisión:

$$p * e^{-x} + q * \sin(x) + r * \cos(x) + s * \tan(x) + t * x^2 + u = 0$$

(Enunciado original en <http://uva.onlinejudge.org/external/103/10341.pdf>)

Capítulo 6

Vuelta Atrás¹

Más vale una retirada a tiempo que una batalla perdida.

Napoleón Bonaparte.

RESUMEN: En este tema se introduce el esquema algorítmico de *Vuelta atrás*, o *Backtracking*, que es una técnica de carácter general utilizada para resolver una gran clase de problemas, en especial de problemas que exigen un recorrido exhaustivo del universo de soluciones.

1. Motivación

Dado un mapa M y un número $n > 0$ se pide encontrar las formas de colorear los países de M utilizando un máximo de n colores, de tal manera que ningún par de países fronterizos tenga el mismo color.

Una forma de resolver el problema es generar todas las posibles maneras de colorear el mapa y después desechar aquellas en que dos países fronterizos tienen el mismo color. Esta forma de resolver el problema sólo resulta aceptable si el conjunto de países, m , y el conjunto de colores, n , son pequeños, ya que el número de posibles formas de colorear el mapa viene dado por las variaciones con repetición de n elementos tomados de m en m : $VR_n^m = n^m$. En la siguiente tabla se muestra el número de posibilidades que habría que generar para algunos valores de n y m .

n : colores	m : países	nº de posibilidades
3	17	129.140.163
5	17	762.939.453.125
3	50	717.897.987.691.852.588.770.249

Se observa, que si tomamos las 17 comunidades autónomas de España, el número de posibilidades a generar con 5 colores sería superior a los 700 mil millones. Si consideramos las 50 provincias, con sólo 3 colores, el número de posibilidades supera el millón de billones.

¹Miguel Valero Espada es el autor principal de este tema.
Modificado por Isabel Pita en el curso 2012/13.

En este capítulo se explica como abordar aquellos problemas cuya única forma conocida de resolver es la generación de todas sus posibles soluciones, para obtener de entre ellas la solución o soluciones reales.

2. Introducción

Existen problemas, como el que se presenta en el apartado anterior, para los que no parece existir una forma o regla fija que nos lleve a obtener una solución de una manera eficiente y precisa. La manera de resolverlos consiste en realizar una búsqueda exhaustiva entre todas las soluciones potenciales hasta encontrar una solución válida, o el conjunto de todas las soluciones válidas. A veces se requiere encontrar *la mejor* (en un sentido preciso) de todas las soluciones válidas.

La búsqueda exhaustiva en un espacio finito dado se conoce como *fuerza bruta* y consiste en ir probando sistemáticamente todas las soluciones potenciales hasta encontrar una solución satisfactoria, o bien agotar el universo de posibilidades. En general, la *fuerza bruta* es impracticable para espacios de soluciones potenciales grandes, lo cual ocurre muy a menudo, ya que el número de soluciones potenciales tiende a crecer de forma exponencial con respecto al tamaño de la entrada en la mayoría de los problemas que trataremos. Obsérvense los valores que se proporcionan en el ejemplo de la sección anterior.

El esquema algorítmico de *vuelta atrás* es una mejora a la estrategia de *fuerza bruta*, ya que la búsqueda se realiza de manera estructurada, descartando grandes bloques de soluciones para reducir el espacio de búsqueda. La diferencia principal entre ambos esquemas es que en el primero las soluciones se forman de manera progresiva, generando soluciones parciales, comprobando en cada paso si la solución que se está construyendo puede conducir a una solución satisfactoria. Mientras que en la *fuerza bruta* la estrategia consiste en probar una solución potencial completa tras otra sin ningún criterio.

En el esquema de *vuelta atrás*, si una solución parcial no puede llevar a una solución completa satisfactoria, la búsqueda se aborta y se vuelve a una solución parcial viable, deshaciendo decisiones previas. Este esquema debe su nombre a este salto hacia atrás.

Tomemos como ejemplo el siguiente problema: dadas n letras diferentes, diseñar un algoritmo que calcule las palabras con m letras ($m \leq n$) diferentes escogidas entre las dadas. El orden de las letras es importante: no será la misma solución *abc* que *bac*.

El número de soluciones potenciales o *espacio de búsqueda* es de n^m , que representan las variaciones con repetición de n letras tomadas de m en m . Realizar una búsqueda exhaustiva en este espacio es impracticable.

Antes de ponernos a probar sin criterio alguno entre todas las posibles combinaciones de letras, dediquemos un segundo a evaluar la estructura de la solución. Es evidente que no podemos poner dos letras iguales en la misma palabra, así que vamos a replantear el problema. Trataremos de colocar las letras de una en una, de forma que no se repitan. De esta manera, toda solución del problema se puede representar como una tupla (x_1, \dots, x_m) en la que x_i representa la letra que se coloca en el lugar i -ésimo de la palabra.

La solución del problema se construye de manera incremental, colocando una letra detrás de otra. En cada paso se comprueba que la última letra no esté repetida con las anteriores. Si la última letra colocada no está repetida, la solución parcial se dice *prometedora* y la búsqueda de la solución continúa a partir de ella. Si no es prometedora, se abortan todas las búsquedas que partan de esa tupla parcial.

De manera general, en los algoritmos de *vuelta atrás*, se consideran problemas cuyas soluciones se puedan construir por etapas. Una solución se expresa como n -tupla (x_1, \dots, x_n)

donde cada $x_i \in S_i$ representa la decisión tomada en la i -ésima etapa de entre un conjunto finito de alternativas.

Una solución tendrá que minimizar, maximizar, o simplemente satisfacer cierta *función criterio*. Se establecen dos categorías de restricciones para los posibles valores de una tupla:

- **Restricciones explícitas**, que indican los conjuntos S_i . Es decir, el conjunto finito de alternativas entre las cuales pueden tomar valor cada una de las componentes de la tupla solución.
- **Restricciones implícitas**, que son las relaciones que se han de establecer entre las componentes de la tupla solución para satisfacer la función criterio.

Volviendo al ejemplo de las palabras con letras diferentes, hemos visto que la solución será una m -tupla y que cada valor de la tupla es una letra en la palabra. Las restricciones serán:

- **Restricciones explícitas para el problema de las palabras:**
 - $S_i = \{1, \dots, n\}, 1 \leq i \leq m$. Es decir, cada letra tiene que pertenecer al alfabeto.
- **Restricciones implícitas para el problema de las palabras:**
 - No puede haber dos letras iguales en la misma palabra.
 - Las soluciones son, por tanto, variaciones sin repetición de n elementos tomados de m en m , es decir $\frac{n!}{(n-m)!}$. Si consideramos palabras de 5 letras sobre un alfabeto de 27 letras distintas, el número de posibilidades se reduce de $27^5 = 14.348.907$ a $\frac{27!}{22!} = 9.687.600$.

El espacio de soluciones potenciales a explorar estará formado por el conjunto de tuplas que satisfacen las restricciones explícitas. Este espacio, se puede estructurar como un *árbol de exploración*, donde en cada nivel se toma la decisión sobre la etapa correspondiente.

Se denomina *nodo de estado* a cualquier nodo del árbol de exploración que satisfaga **las restricciones explícitas**, y corresponde a una tupla parcial o una tupla completa. Los *nodos solución* serán los correspondientes a las tuplas completas que además satisfagan **las restricciones implícitas**.

Un elemento adicional imprescindible es la *función de poda* o *test de factibilidad*, que permite determinar cuándo una solución parcial puede conducir a una solución satisfactoria. De tal manera que si un *nodo* no satisface la función de poda es inútil continuar la búsqueda por esa rama del árbol. La *función de poda* permite pues reducir la búsqueda en el árbol de exploración.

Una vez definido el árbol de exploración, el algoritmo realizará un recorrido del árbol en cierto orden, hasta encontrar la primera solución. El mismo algoritmo, con ligeras modificaciones, se podrá utilizar para encontrar todas las soluciones, o una solución óptima.

El árbol de exploración no se construye de manera explícita, es decir no se almacena en memoria, sino que se va construyendo de manera implícita conforme avanza la búsqueda por medio de llamadas recursivas. Durante el proceso, para cada nodo se generarán los nodos sucesores (estados alcanzables tomando una determinada decisión correspondiente a la siguiente etapa). En el proceso de generación habrá distintos tipos de nodos:

Nodos vivos Aquellos para los cuales aún no se han generado todos sus hijos. Todavía pueden expandirse.

Nodo en expansión Aquel para el cual se están generando sus hijos.

Nodos muertos Aquellos que no hay que seguir explorando porque, o bien no han superado el test de factibilidad, o bien se han explorado insatisfactoriamente todos sus hijos.

El recorrido del árbol de exploración se realiza en profundidad. Cuando se llega a un nodo muerto, hay que deshacer la última decisión tomada y optar por otra alternativa (*vuelta atrás*). La forma más sencilla de expresar este retroceso es mediante un algoritmo recursivo, ya que la vuelta atrás se consigue automáticamente haciendo terminar la llamada recursiva y volviendo a aquella que la invocó.

El coste de los algoritmos de *vuelta atrás* en el caso peor es del orden del tamaño del árbol de exploración, ya que en el peor de los casos nos veremos obligados a recorrer exhaustivamente todas las posibilidades. El espacio de soluciones potenciales suele ser, como mínimo, exponencial en el tamaño de la entrada. La efectividad de la *vuelta atrás* va a depender decisivamente de las funciones de poda que se utilicen, ya que si son adecuadas permitirán reducir considerablemente el número de nodos explorados.

Se podrían realizar búsquedas más inteligentes haciendo que en cada momento se explore el nodo más prometedor, utilizando para ello algún tipo de heurística que permita ordenar los nodos en un tipo de datos denominado *cola de prioridad*. Esta estrategia da lugar al esquema conocido como de **ramificación y poda**. Las colas de prioridad y el esquema de ramificación y poda se estudiará el próximo curso.

2.1. Esquema básico de la *vuelta atrás*

El esquema de *Vuelta atrás* en pseudocódigo es el siguiente:

```

vueltaAtras (Tupla & sol, int k) {
    prepararRecorridoNivel(k);
    while (!ultimoHijoNivel(k)) {
        sol[k] = siguienteHijoNivel(k);
        if (esValida(sol, k)) {
            if (esSolucion(sol, k))
                tratarSolucion(sol);
            else
                vueltaAtras(sol, k + 1);
        }
    }
}

```

El tipo de la solución *sol* es una tupla de cierto tipo específico para cada problema. En ella se va acumulando la solución. La variable *k* es la que determina en qué nivel del árbol de exploración estamos.

El método `prepararRecorridoNivel` genera los candidatos para ampliar la solución en la siguiente etapa y depende del problema en concreto. En el cuerpo de la función, iteramos a lo largo de todas las posibles soluciones candidatas, dadas por la función: `siguienteHijoNivel` hasta la última candidata, dada por la función: `ultimoHijoNivel`. Para cada solución candidata, ampliamos la solución con el nuevo valor y comprobamos si satisface las restricciones implícitas/explicitas con la función booleana `esValida`. Esta función implementa la *función de factibilidad* que presentábamos más arriba.

En el caso de que la solución parcial sea válida tenemos dos posibilidades: o bien hemos alcanzado el final de la búsqueda, por lo que ya podemos mostrar la solución final, o bien continuamos nuestra búsqueda mediante la llamada recursiva.

Este esquema encontrará todas las soluciones del problema. Si quisiéramos que sólo encontrara una solución bastaría con añadir una variable booleana *éxito* que haga finalizar los bucles cuando se encuentra la primera solución.

2.2. Resolución del problema de las palabras

Veamos cómo se aplica el esquema al problema de las palabras que hemos descrito más arriba. El esquema presentado tendrá pequeñas variaciones en cada problema, hay que tomarlo como una referencia y no como un patrón estricto. En el caso de las palabras la función recursiva en C++ podría ser como sigue.

```
void variaciones(int solucion[], int k, int n, int m){
    for(int letra = 0; letra < n; letra++){
        solucion[k] = letra;
        if(esValida(solucion, k)){
            if(esSolucion(k, m))
                tratarSolucion(solucion,m);
            else
                variaciones(solucion, k + 1, n, m);
        }
    }
}
```

La función `prepararRecorridoNivel` no existe explícitamente ya que en este caso la iteración es muy simple y se aplica sobre valores numéricos del 0 al $n - 1$.

La función `esValida` es la encargada de comprobar que la nueva letra que hemos incorporado a la solución no está repetida con las anteriores. La podríamos escribir como sigue:

```
bool esValida(int solucion[], int k) {
    int i == 0;
    while(i < k && solucion[i] != solucion[k]) i++;
    return i == k;
}
```

La función `esSolucion` simplemente comprueba que hemos colocado todas las letras.

```
bool esSolucion(int k, int m){
    return k == (m - 1);
}
```

Cuando encontramos una solución, podemos simplemente escribirla por la salida estándar:

```
void tratarSolucion(int solucion[], int m){
    cout << "Solucion: ";
    for(int i = 0; i < m; i++)
        cout << solucion[i] << " ";
    cout << endl;
}
```

Por último, la llamada inicial será de la siguiente manera:

```
void variaciones(int n, int m){
    int solucion[m];
    variaciones(solucion, 0, n, m);
}
```

```

}

int main()
{
    variaciones(27, 5);
    return 0;
}

```

La función *esValida* recorre la lista de letras y comprueba si la última letra insertada en la solución coincide con alguna de las anteriores. Esta operación tiene un coste lineal en función de la entrada. La función se ejecuta muchas veces por lo que el coste de la función repercute negativamente en la ejecución del programa. Podríamos ahorrarnos este coste utilizando lo que se conoce como la técnica de *marcaje*.

3. Vuelta atrás con marcaje

La técnica de *marcaje* consiste en guardar cierta información que ayuda a decidir si una solución parcial es válida o no. La información del *marcaje* se pasa en cada llamada recursiva. Por lo tanto, reduce el coste computacional a cambio de utilizar más memoria. El esquema de *vuelta atrás* con marcaje es el siguiente:

```

vueltaAtrasConMarcaje (Tupla & sol, int k, Marca & marcas) {
    prepararRecorridoNivel(k);
    while(!ultimoHijoNivel(k)){
        sol[k] = siguienteHijoNivel(k);
        if (esValida(sol, k, marcas)){
            if (esSolucion(sol, k))
                tratarSolucion(sol);
            else{
                marcar(marcas, sol, k);
                vueltaAtrasConMarcaje(sol, k + 1, marcas);
                desmarcar(marcas, sol, k);
            }
        }
    }
}

```

El tipo *Marca* depende de cada problema concreto.

Normalmente, *desmarcaremos* después de la llamada recursiva para devolver las marcas a su estado anterior a la llamada. En algunos casos no es necesario *desmarcar*, como ocurre en el ejemplo 6.

En el ejemplo de las palabras, podemos utilizar marcaje para evitar el bucle de comprobación cada vez que aumentamos la solución. Para ello utilizamos un vector de booleanos de tamaño el número de letras del alfabeto considerado. Cada posición del vector indica si la letra correspondiente ha sido ya utilizada. De esta forma, las operaciones de *marcar* una letra como ya utilizada y consultar si una letra ya está utilizada tienen ambas coste constante. La solución con marcaje quedaría como sigue.

```

void variaciones(int solucion[], int k, int n, int m, bool marcas[]){
    for(int letra = 0; letra < n; letra++){
        if(!marcas[letra]){
            solucion[k] = letra;

```

```

        if(k == m - 1){
            tratarSolucion(solucion,m);
        }
        else{
            marcas[letra] = true; //marcar
            variaciones(solucion, k + 1, n, m, marcas);
            marcas[letra] = false; //desmarcar
        }
    }
}

```

Observar que al finalizar la llamada recursiva se procede a desmarcar la letra correspondiente a esa llamada recursiva. Los parámetros de entrada `solucion` y `marcas`, modifican su valor de una llamada recursiva a otra. No se utiliza en este caso el paso de parámetros por referencia de forma explícita, debido al tratamiento dado por C++ a los vectores, como punteros a la primera posición de memoria. Esto hace que las modificaciones realizadas en un vector queden reflejas en las sucesivas llamadas recursivas.

4. Ejemplo: problema de las n -reinas

El problema de las 8 reinas consiste en colocar 8 reinas en un tablero de ajedrez sin que se amenacen. Dos reinas se amenazan si comparten la misma fila, columna o diagonal.

El número de soluciones potenciales o espacio de búsqueda teórico es de $\binom{64}{8} = 4.426.165.368$, que representan todas las combinaciones en las que podemos poner 8 reinas en un tablero de 64 casillas. Realizar una búsqueda exhaustiva en este espacio es impracticable, siendo todavía más problemático si ampliamos el tamaño de la entrada, por ejemplo tratando de colocar 11 reinas en un tablero de 11×11 (743.595.781.824 soluciones potenciales).

Si evaluamos la estructura de la solución vemos que no podemos poner dos reinas en la misma fila. Trataremos de colocar una reina **en cada fila del tablero**, de forma que no se amenacen. De esta manera, toda solución del problema se puede representar como una 8-tupla (x_1, \dots, x_8) en la que x_i representa la columna en la que se coloca la reina que está en la fila i -ésima del tablero.

La tupla $(4, 7, 3, 8, 2, 5, 1, 6)$ representa el siguiente tablero.

			X				
						X	
		X					
							X
	X						
				X			
X							
					X		

Las restricciones serán:

- **Restricciones explícitas para el problema de las reinas:**

- $S_i = \{1, \dots, 8\}, 1 \leq i \leq 8$. Es decir, cada columna tiene que estar dentro del tablero.

- Esta representación hace que el espacio de soluciones potenciales se reduzca a 8^8 posibilidades (16.777.216 valores).

■ **Restricciones implícitas para el problema de las reinas:**

- No puede haber dos reinas en la misma columna, ni en la misma diagonal.
- Al no poder haber dos reinas en la misma columna, se deduce que todas las soluciones son permutaciones de la 8-tupla (1, 2, 3, 4, 5, 6, 7, 8). Por lo tanto el espacio de soluciones potenciales se reduce a $8!$ (40.320 valores diferentes).

Una función recursiva en C++ que resuelve el problema para n reinas es.

```
void nReinas(int solucion[], int k, int n){
    for(int i = 0; i < n; i++){
        solucion[k] = i;
        if (esValida(solucion, k)){
            if(k == n - 1){
                tratarSolucion(k, n);
            }
            else{
                nReinas(solucion, k + 1, n);
            }
        }
    }
}
```

La función `esValida` es la encargada de comprobar que la nueva reina que hemos incorporado a la solución no amenaza a las anteriores. La podríamos escribir como sigue:

```
bool esValida(int solucion[], int k) {
    bool correcto = true;
    int i = 0;
    while (i < k && correcto){
        if ((solucion[i] == solucion[k])
            || abs(solucion[k] - solucion[i]) == k - i)
            correcto = false;
        else
            i++;
    }
    return correcto;
}
```

Comprobamos que la nueva reina no está en la misma columna que las anteriores, `solucion[i] == solucion[k]` y que no comparten diagonal, `abs(solucion[k] - solucion[i]) == k - i`. Obviamente nunca puede estar en la misma fila por la manera en que construimos la solución.

Cuando encontramos una solución, podemos simplemente escribirla por la salida estándar:

```
void tratarSolucion(int solucion[], int n){
    cout << "Solucion: ";
    for(int i = 0; i < n; i++)
        cout << solucion[i] << " ";
    cout << endl;
}
```

Por último, la llamada inicial será de la siguiente manera:

```
void nReinas(int n) {
    int solucion[n];
    nReinas(solucion, 0, n);
}

int main()
{
    nReinas(8);
    return 0;
}
```

Podríamos reducir el espacio de búsqueda teniendo en cuenta que las soluciones son simétricas. Si hay una solución colocando la primera reina en la casilla 2 también la habrá colocando la reina inicial en la casilla $n - 2$. Así pues, podríamos lanzar el método recursivo para el primer nivel sólo para las casillas menores de $n/2$, reduciendo el espacio de búsqueda a la mitad.

La función `esValida` recorre la lista de reinas y comprueba si la última reina insertada en la solución amenaza a las anteriores. Esta operación tiene un coste lineal en función de la entrada. Se puede reducir este coste con la técnica de marcaje.

Podemos utilizar como *marca* una estructura de datos *tablero*. Cada vez que insertamos una reina en la solución *se marcan* las casillas amenazadas por la nueva reina. Para comprobar si una nueva reina está amenazada basta con comprobar si esta marcada la casilla correspondiente del tablero. El problema de esta solución es que marcar en el tablero las casilla que amenaza la nueva reina supone un coste lineal, lo cual sigue resultando poco eficiente.

La solución está en no utilizar un tablero completo para realizar el marcaje, sino dos vectores: uno con las columnas amenazadas y otro con las diagonales amenazadas. El vector de columnas tendrá tamaño n , sin embargo, existen muchas más diagonales. Para poder resolver el problema se deben de numerar las diagonales y considerar cada posición del vector como una de ellas. La modificación y acceso a ambos vectores tendrá coste constante. El problema está resuelto en detalle en el capítulo 14 del libro (Martí Oliet et al., 2004).

5. Ejemplo de búsqueda de una sola solución: Dominó

Se trata de encontrar una cadena circular de fichas de dominó. Teniendo en cuenta:

- Cada cadena tiene que utilizar las 28 fichas diferentes que contiene el juego de dominó.
- No se puede repetir ninguna ficha.
- Las cadenas tienen que ser correctas, es decir, cada ficha tiene que ser compatible con la siguiente y la cadena tiene que cerrar (el valor de un extremo de la última ficha tiene que coincidir con el otro extremo de la primera). Por ejemplo: $6|3 \rightarrow 3|4 \rightarrow 4|1 \rightarrow 1|0 \rightarrow \dots \rightarrow 5|6$ es una cadena correcta.

La solución va a ser una tupla de 29 valores (x_0, \dots, x_{28}) cada x_i es un número del 0 al 6. Es decir, en la solución no guardaremos las fichas, sino los valores de uno de los extremos. En el ejemplo de más arriba la solución tendría la siguiente forma: $(6, 3, 4, 1, 0, \dots, 5)$. No necesitamos guardar explícitamente los dos extremos de las fichas, ya que cada ficha tiene

que coincidir con la siguiente. Se declara una posición más en la tupla para poder realizar la comprobación de que la cadena es cerrada.

Para evitar fichas repetidas utilizaremos una matriz (7×7) donde marcaremos las fichas usadas. Hay que tener en cuenta que si marcamos la casilla (i, j) , habrá que marcar la simétrica (j, i) , ya que se trata de la misma ficha.

El problema pide que se encuentre una sola solución, no todas las que existan, así que vamos a tener que abortar la búsqueda en el momento que aparezca la primera. Para ello utilizaremos una variable de control `exito`.

La función principal será:

```
void domino(int sol[], int k, int n, bool marcas[NUM_VAL][NUM_VAL], bool &exito) {
    int i = 0;
    int m = (n * n + n) / 2;
    while (i < n && !exito) {
        if (!marcas[sol[k-1]][i]) {
            sol[k] = i;
            if (k == m) {
                if (sol[0] == sol[k]) {
                    tratarSolucion(sol, m);
                    exito = true;
                }
            }
            else {
                marcas[sol[k-1]][i] = true;
                marcas[i][sol[k-1]] = true;
                domino(sol, k + 1, n, marcas, exito);
                marcas[sol[k-1]][i] = false;
                marcas[i][sol[k-1]] = false;
            }
        }
        i++;
    }
}
```

donde n es el número de valores posibles de las fichas, en nuestro caso $n = 7$ ya que las fichas toman valores del 0 al 6, la matriz de marcas se declara de dimensión $n \times n$ y el vector solución es de tamaño $(n \times n + n)/2 + 1$.

Por tradición en el juego del dominó, siempre se empieza por el doble 6, así que pondremos los dos primeros valores como 6. Podríamos haber utilizado cualquier par de valores.

La llamada principal del programa será de la siguiente manera:

```
int main()
{
    const int NUM_VAL = 7;
    const int TAM_SOL = (NUM_VAL*NUM_VAL+NUM_VAL)/2+1;
    int sol[TAM_SOL];
    bool marcas[NUM_VAL][NUM_VAL];
    for(int i = 0; i < NUM_VAL; i++)
        for(int j = 0; j < NUM_VAL; j++)
            marcas[i][j] = false;

    sol[0] = NUM_VAL-1;
    sol[1] = NUM_VAL-1;
    marcas[NUM_VAL-1][NUM_VAL-1] = true;
    bool exito = false;
```

```

    domino(sol, 2, NUM_VAL, marcas, exito);
    return 0;
}

```

6. Ejemplo que no necesita desmarcar: El laberinto

Podemos representar un laberinto como una matriz booleana L de $n \times n$ de tal manera que se puede pasar por las casillas con *true*. Las casillas con *false* representan los muros infranqueables. Solo nos podemos desplazar a las cuatro casillas adyacentes: arriba, abajo, izquierda y derecha. Se pide escribir un algoritmo que encuentre la salida, asumiendo que la entrada al laberinto está en la casilla $(0, 0)$ y la salida en la $(n - 1, n - 1)$.

Las posibles soluciones son todas las listas de posiciones del laberinto, de longitud n^2 como máximo, dado que en el peor caso visitaremos todas y cada una de las casillas. Representamos la solución como un vector `solucion[n2]` de casillas, de tal manera que cada una de las casillas es transitable y cada casilla es adyacente a su siguiente.

Cada nodo del árbol de búsqueda tendrá 4 hijos correspondientes a cada una de las posibles continuaciones (arriba, abajo, izquierda y derecha).

Para controlar que no pasamos dos veces por la misma casilla mantendremos un marcador en forma de matriz `marcas[n][n]`, marcando aquellas casillas por las que hemos pasado. Veamos como queda el algoritmo principal:

```

void laberinto(bool lab[N][N], casilla solucion[], int k, int n,
               bool marcas[N][N], bool &exito){
    int dir=0;
    while ((dir < 4)&&!exito){
        solucion[k] = sigDireccion(dir, solucion[k-1]);
        if(esValida(lab, solucion[k], n, marcas)){
            if(esSolucion(solucion[k],n)){
                tratarSolucion(solucion, k);
                exito=true;
            }
            else{
                // marcar
                marcas[solucion[k].fila][solucion[k].columna] = true;
                laberinto(lab, solucion, k + 1, n, marcas,exito);
                //desmarcar
                //marcas[solucion[k].fila][solucion[k].columna] = false;
            }
        }
        dir++;
    }
}

```

En este caso, el algoritmo no *desmarca* las posiciones utilizadas ya que al volver de la llamada recursiva o hemos encontrado la solución o un callejón sin salida, por lo que no nos interesa volver a considerarla. Nótese que si se pidiesen todas las soluciones posibles, sería necesario desmarcar ya que un camino que haya sido utilizado, si ha conducido a una solución puede ser parte de otra solución.

Para saber si una casilla es válida basta con preguntar que esté dentro de los límites del tablero, que no sea un *muro* y que no esté marcada.

```

bool esValida(bool lab[N][N], casilla c, int n, const bool marcas[N][N]) {
    return c.fila >= 0 && c.columna >= 0 && c.fila < n && c.columna < n
    && lab[c.fila][c.columna] && !marcas[c.fila][c.columna];
}

```

Para enumerar las cuatro direcciones posibles de movimiento echamos mano de la siguiente función auxiliar:

```

casilla sigDireccion(int dir, casilla pos){
    switch (dir) {
        case 0:
            ++ pos.columna;
            break;
        case 1:
            ++ pos.fila;
            break;
        case 2:
            -- pos.columna;
            break;
        case 3:
            -- pos.fila;
            break;
        default:
            break;
    }
    return pos;
}

```

El problema asume que la salida está en la posición $(n-1, n-1)$, así que la comprobación de la solución será sencilla:

```

bool esSolucion(casilla pos, int n){
    return pos.fila == n - 1 && pos.columna == n - 1;
}

```

Por último, la llamada inicial será:

```

int main()
{
    const int N = ...;
    bool Laberinto[N][N];
    InicializarLab(Laberinto, N);
    bool marcas[N][N];
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            marcas[i][j] = false;

    casilla sol[N*N];
    sol[0].fila = 0;
    sol[0].columna = 0;
    marcas[0][0] = true;
    bool exito = false;
    laberinto(Laberinto, sol, 1, N, marcas, exito);
    return 0;
}

```

}

Vemos que fijamos el primer valor de la solución como (0,0), ya que esa es la posición de la salida. La función `Inicializarlab` inicializa el laberinto poniendo las paredes.

Este algoritmo no tiene porque encontrar el camino óptimo, encuentra simplemente una solución; Sin embargo, *vuelta atrás* se puede utilizar para problemas de optimización como se explica más adelante.

7. Optimización

En muchos casos necesitamos obtener la mejor solución entre todas las soluciones posibles.

Para tratar este tipo de problemas de optimización tenemos que modificar el esquema anterior de forma que se almacene la mejor solución hasta el momento. Así, a la hora de tratar una nueva solución se comparará con la que tenemos almacenada. En general, guardaremos la mejor solución junto con su valor.

7.1. Ejemplo: Problema del viajante

El problema del viajante, en inglés *Travelling Salesman Problem* es uno de los problemas de optimización más estudiados a lo largo de la historia de la computación. A priori parece tener una solución sencilla pero en la práctica encontrar soluciones óptimas es muy complejo computacionalmente.

El problema se puede enunciar de la siguiente manera. Sean N ciudades de un territorio. El objetivo es encontrar una ruta que, comenzando y terminando en una ciudad concreta, pase una sola vez por cada una de las ciudades intermedias y minimice la distancia recorrida por el viajante.

Básicamente hay que encontrar una permutación del conjunto de ciudades $P = \{c_0, \dots, c_N\}$ tal que la suma de las distancias entre una ciudad y la siguiente sea mínimo, es decir $\sum_{i:0..N-1} d[c_i, c_{(i+1) \% N}]$ sea mínimo. La distancia d entre dos ciudades viene dada en una matriz. Para indicar la ausencia de conexión entre dos ciudades se puede usar un valor especial en la matriz distancias. En el código utilizaremos una función booleana *hayArista* que nos indica si hay conexión entre dos ciudades dadas.

El tamaño del árbol de soluciones es $(N - 1)!$, ya que corresponde a todas las posibles permutaciones, teniendo en cuenta que el principio y el final es el mismo.

El problema tiene considerables aplicaciones prácticas, aparte de las más evidentes en áreas de logística de transporte. Por ejemplo, en robótica, permite resolver problemas de fabricación para minimizar el número de desplazamientos al realizar una serie de perforaciones en una plancha o en un circuito impreso. También puede ser utilizado en control y operativa optimizada de semáforos, etc.

Veamos cómo es la solución para este problema.

```
void viajante(int distancias[N][N], int solucion[], int &coste, int k,
             int n, int solucionMejor[], int &costeMejor, bool usadas[]){
    //para evitar soluciones repetidas fijamos como ciudad de comienzo la 0,
    //por lo que no se va a volver a considerar
    for(int i = 1; i < n; i++){
        solucion[k] = i;
        if (esValida(distancias, solucion, k, usadas)){
            coste += distancias[solucion[k-1]][solucion[k]];
            usadas[i]=true;
        }
    }
}
```

```

        if(k==n-1){
            if hayArista(distancias,solucion[k],solucion[0])
            {
                if(coste +distancias[k][0]< costeMejor){
                    costeMejor = coste+distancias[k][0];
                    copiarSolucion(solucion, solucionMejor);
                }
            }
        }
        else viajante(distancias, solucion, coste, k+1, n,
                      solucionMejor, costeMejor);
        usadas[i]=false;
        coste -= distancias[solucion[k-1]][solucion[k]];
    }
}

```

Para mejorar el coste de la función `esValida` se utiliza la técnica de marcaje. En este caso, se declara un vector `usadas` de n componentes, donde el valor de cada componente indica si la ciudad correspondiente ha sido visitada.

```

bool esValida(int distancias[N][N], int solucion[],int k,bool usadas[]){

    return (hayArista(distancias,solucion[k-1],solucion[k])
            && !usadas[solucion[k]]);

}

```

Llevamos dos parámetros en los que guardamos la mejor solución hasta el momento y el coste de la misma (*solucionMejor* y *costeMejor*). Cuando encontremos una solución comprobaremos si es mejor que la que tenemos almacenada, en caso positivo la consideraremos como la nueva mejor solución. El algoritmo de *vuelta atrás* garantiza que al final de la búsqueda la solución encontrada tiene el coste óptimo. El parámetro *coste* acumula el coste de la solución parcial, que se va calculando de forma incremental en cada llamada recursiva, evitando realizar el cálculo en cada llamada. Al finalizar la llamada recursiva debe actualizar su valor, igual que se hace en la técnica de marcaje.

- Podríamos mejorar la búsqueda del camino óptimo utilizando una estimación optimista para realizar podas tempranas. La idea es prever cuál es el mínimo coste de lo que falta por recorrer. Si ese coste, sumado al que llevamos acumulado, supera la mejor solución encontrada hasta el momento, entonces podemos abandonar la búsqueda porque estamos seguros de que ningún camino va a mejorar la solución. Utilizamos una estimación optimista, cuanto menos optimista mejor, para saber si es posible mejorar el resultado.
- Para calcular una estimación optimista es suficiente con encontrar la mínima distancia entre cualquier par de ciudades, y considerar que todos los desplazamientos van a tener esa distancia. Se pueden realizar cálculos más ajustados del coste del camino que queda por recorrer, pero hay que tener en cuenta que el cálculo debe ser sencillo para no aumentar el coste del algoritmo.

Así tendremos que añadir antes de realizar la llamada recursiva, el cálculo del coste estimado. Se calcula considerando que el resto de desplazamientos tienen un coste mínimo. El `costeMinimo` se puede calcular muy fácilmente recorriendo la matriz de distancias;

consideramos que se ha procesado al principio de la ejecución y que lo tenemos almacenado en un parámetro. Solo realizaremos la recursión si la solución se puede mejorar.

```
int costeEstimado = coste + (n - k + 1) * costeMinimo;
if(costeEstimado < costeMejor)
    viajante(distancias, solucion, coste, k+1, n, solucionMejor, costeMejor,
            usadas, costeMinimo);
```

En la llamada inicial se debe fijar una ciudad de comienzo para evitar soluciones repetidas:

```
costeMinimo=calcularMinimo(distancias);
solucion[0]=0;usadas[0]=true;
for (int i=1;i<n;i++) {usadas[i]=false;};
coste=0;
costeMejor = ...
//una cota superior, como por ejemplo la suma de todas las aristas del grafo
viajante(distancias,solucion,coste,1,n,solucionMejor,costeMejor,usadas,costeMinimo);
```

7.2. Ejemplo: Problema de la mochila

Otro problema clásico de optimización es el problema de la mochila. La idea es que tenemos n objetos con valor (v_0, \dots, v_{n-1}) y peso (p_0, \dots, p_{n-1}) , y tenemos que determinar qué objetos transportar en la mochila sin superar su capacidad m (en peso) para maximizar el valor del contenido de la mochila.

Para resolver este problema en cada nivel del árbol de búsqueda vamos a decidir si cogemos o no el i -ésimo elemento. Así pues la solución será una tupla (b_0, \dots, b_{n-1}) de booleanos.

Se consideran las siguientes restricciones:

- Deberemos maximizar el valor de lo que llevamos $\sum_{i:0..n-1} b_i v_i$.
- El peso no debe exceder el máximo permitido $\sum_{i:0..n-1} b_i p_i \leq m$.

Una posible solución es:

```
void mochila(float P[], float V[], bool solucion[], int k, int n, int m,
            float &peso, float &beneficio, int solucionMejor[], int &valorMejor){
    // hijo izquierdo [cogemos el objeto]
    solucion[k] = true;
    peso = peso + P[k];
    beneficio = beneficio + V[k];
    if(peso <= m){
        if(k == n-1){
            if(valorMejor < beneficio){
                valorMejor = beneficio;
                copiarSolucion(solucion, solucionMejor);
            }
        }
        else{
            mochila(P,V,solucion, k+1,n, m, peso, beneficio,
                    solucionMejor, valorMejor);
        }
    }
}
```

```

    peso = peso - P[k];           //desmarcamos peso y beneficio
    beneficio = beneficio - V[k];
    // hijo derecho [no cogemos el objeto]
    solucion[k] = false;
    if(k == n-1){
        if(valorMejor < beneficio){
            valorMejor = beneficio;
            copiarSolucion(solucion, solucionMejor);
        }
    }
    else{
        mochila(P,V,solucion, k+1,n, m,peso, beneficio,
            solucionMejor, valorMejor);
    }
}

```

En el hijo de la derecha no tendremos que comprobar si excedemos el peso total, ya que al descartar el objeto no aumentamos el peso acumulado.

Este problema da pie a optimización, ya que podemos calcular una cota superior (una evaluación optimista) del beneficio que podemos obtener con lo que nos resta para rellenar la mochila. Para calcularla, organizamos inicialmente los objetos en los vectores P y V de manera que estén ordenados por “densidad de valor” decreciente. Llamamos densidad de valor al cociente v_i/p_i . De esta forma, cogeremos primero los objetos que tienen más valor por unidad de peso. Si en un cierto nodo, la tupla parcial es (b_0, \dots, b_k) y hemos alcanzado un beneficio `beneficio` y un peso `peso`, estimamos el beneficio optimista como la suma de `beneficio` más el beneficio conseguido cogiendo los objetos que quepan en el orden indicado desde el $k + 1$ al $n - 1$. Si se llega a un objeto j que ya no cabe, se fracciona y se suma el valor de la fracción que quepa. Esta forma de proceder se llama solución *voraz* al problema de la mochila con posible fraccionamiento de objetos y produce siempre una cota superior a cualquier solución donde no se permita fraccionamiento.

La poda se produce si el beneficio optimista es **menor** que el beneficio de la mejor solución alcanzada hasta el momento.

8. Para terminar...

Terminamos el tema con la solución al problema del coloreado de mapas planteado en la primera sección. Como en los casos anteriores, lo primero que nos debemos preguntar es sobre la forma de la solución y del árbol de búsqueda.

En este caso:

- Si el mapa M tiene m países, numerados 0 a $m - 1$, entonces la solución va a ser una tupla (x_0, \dots, x_{m-1}) donde x_i es el color asignado al i -ésimo país.
- Cada elemento x_i de la tupla pertenecerá al conjunto $\{0, \dots, n - 1\}$ de colores válidos.

Cada vez que vayamos a pintar un país de un color tendremos que comprobar que ninguno de los adyacentes está pintado con el mismo color. En este caso es más sencillo hacer la comprobación cada vez que coloreamos un vértice en lugar de utilizar *marcaje*.

Así pues la función principal será:

```

void colorear(int solucion[], int k, int n, int m){
    for(int c = 0; c < n; c++){
        solucion[k] = c;
        if(esValida(solucion, k)){
            if(esSolucion(k,m)){
                tratarSolucion(solucion,m);
            }
            else{
                colorear(solucion, k + 1, n,m);
            }
        }
    }
}

```

La función `esValida` será la encargada de comprobar si la solución parcial no vulnera la restricciones de que dos países limítrofes compartan color; para implementarla asumiremos que tenemos acceso a cierto objeto `M` donde se guarda el mapa, y que tiene un método que dice si dos países son fronterizos.

```

bool esValida(int solucion[], int k) {
    int i = 0; bool valida = true;
    while (i < k && valida) {
        if (M.hayFrontera(i, k) && solucion[k] == solucion[i])
            valida = false;
        i++;
    }
    return valida;
}

```

A la hora de hacer la llamada inicial podemos asignar al primer país del mapa un color arbitrario.

Notas bibliográficas

Gran parte del contenido de este capítulo está basado en el capítulo correspondiente de (Martí Oliet et al., 2013).

Ejercicios

1. Vamos a realizar una modificación en el problema de las n -reinas. Asumimos que cada casilla del tablero de ajedrez tiene un número asignado que representa el valor de esa casilla.

Se pide hacer un algoritmo que resuelva el problema de las n -reinas, de tal manera que la suma de los números de las posiciones donde se colocan las reinas sea máxima.

2. Dado un número entero M y un vector de n números naturales $V[0..n-1]$, determinar si existe una forma de insertar entre los n números del vector (tal como están colocados en el vector) operaciones de suma y resta de forma que se obtenga el número M como resultado final.

El programa debe determinar si es posible o no realizar la expresión, y devolver la expresión oportuna en el caso afirmativo.

3. A partir de un tablero de ajedrez de $n \times n$ posiciones y de un caballo situado en una posición arbitraria (i, j) se pide diseñar un algoritmo que determine un secuencia de movimientos que visite todas las casillas del tablero una sola vez. El último movimiento debe devolver el caballo a su posición inicial.
4. Optimizar el problema de la mochila descrito más arriba. Para ello podemos utilizar la siguiente idea para realizar una estimación optimista:
 - Consideramos que tenemos los objetos ordenados, los que tienen mayor valor por unidad de peso son los primeros.
 - Asumimos que los objetos son fraccionables; que podemos coger una determinada parte de cada uno.

Con estas dos consideraciones podemos realizar un algoritmo de estimación optimista muy sencillo, que siempre escoja una solución óptima, que nunca podrá ser superada por objetos no fraccionables. Este algoritmo se puede utilizar como cota superior.

5. Encontrar n puntos del eje real a partir de las $n(n-1)/2$ distancias (no necesariamente diferentes) entre cada par de puntos, sabiendo que el menor de dichos puntos es el origen y que el multiconjunto de las distancias viene dado en orden creciente.

Ejemplo: si el multiconjunto de distancias es $\{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10\}$, una solución válida es $\{0, 3, 5, 6, 8, 10\}$.

6. Deseamos decorar una pared de L metros de ancho. Hemos tenido la innovadora idea de hacerlo colgando una hilera de cuadros pegados lado con lado. Nos disponemos a comprar los cuadros en la feria de arte moderno, donde tenemos la posibilidad de elegir entre n cuadros. Cada cuadro tiene un prestigio p_i , y unas dimensiones de a_i metros de alto por b_i metros de ancho, $1 \leq i \leq n$. Dado lo peculiar de los cuadros, podemos elegir colgar cada cuadro tanto en horizontal como en vertical sin que por ello se vea afectado su prestigio. Lo que no podemos hacer es trocear un cuadro. Diseñar un algoritmo que determine qué cuadros comprar de forma que la longitud de la hilera de cuadros sume exactamente L metros y se maximice el prestigio acumulado en la pared.
7. [Examen Junio, 2012] Implementar una función que encuentre la forma *más rápida* de viajar desde una casilla de salida hasta una casilla de llegada de una rejilla. Cada casilla de la rejilla está etiquetada con una letra, de forma que en el camino desde la salida hacia la llegada se debe ir formando (de forma cíclica) una palabra dada. Desde una celda se puede ir a cualquiera de las cuatro celdas adyacentes.

Como ejemplo, a continuación aparece la forma más corta de salir de una rejilla de 5×8 en la que el punto de salida está situado en la posición $(0, 4)$ y hay que llegar a la posición $(7, 0)$ y la palabra que hay que ir formando por el camino es EDA.

0	M	D	A	A	E	E	D	A
1	A	E	E	D	D	A	N	D
2	D	B	D	X	E	D	A	E
3	E	A	E	D	A	R	T	D
4	E	D	M	P	L	E	D	A
	0	1	2	3	4	5	6	7

El tablero tendrá un tamaño $N \times M$ con $N > 0$ y $M > 0$.

Las función recibirá el punto origen, el punto destino y la palabra a formar y deberá determinar la forma más rápida de viajar de uno a otro (si es que esto es posible), dando las direcciones que hay que ir cogiendo (en el caso del ejemplo será E, N, E, E, E, etc.). Si necesitas parámetros adicionales, añádelos indicando sus valores iniciales.

8. [Examen Septiembre, 2012] Los ferrys son barcos que sirven para trasladar coches de una orilla a otra de un río. Los coches esperan en fila al ferry y cuando éste llega un operario les va dejando entrar.

En nuestro caso el ferry tiene espacio para dos filas de coches (la fila de babor y la fila de estribor) cada una de N metros. El operario conoce de antemano la longitud de cada uno de los coches que están esperando a entrar en él y debe, según van llegando, mandarles a la fila de babor o a la fila de estribor, de forma que el ferry quede lo más cargado posible. Ten en cuenta que los coches deben entrar en el ferry **en el orden en que aparecen** en la fila de espera, por lo que en el momento en que un coche ya no entra en el ferry porque no hay hueco suficiente para él, no puede entrar ningún otro coche que esté detrás aunque sea más pequeño y sí hubiera hueco para él.

Implementa una función que reciba la capacidad del ferry en metros y la colección con las longitudes de los coches que están esperando (puedes utilizar el TAD que mejor te venga) y devuelva la colocación óptima de los coches, entendiendo ésta como la secuencia de filas en las que se van metiendo los coches. Supón la existencia de los símbolos BABOR y ESTRIBOR.

Ejemplo: Si el ferry tiene 5 metros de longitud y en la fila tenemos coches con longitudes (del primero al último) 2.5, 3, 1, 1, 1.5, 0.7 y 0.8, una solución óptima es [BABOR, ESTRIBOR, ESTRIBOR, ESTRIBOR, BABOR, BABOR].

9. [Examen Febrero, 2014] Deseamos organizar un festival de rock al aire libre para lo cual vamos a contratar exactamente a N artistas de entre M disponibles ($N < M$). No todos los artistas aceptan tocar juntos en el festival. Los “vetos” entre artistas son conocidos de antemano. Para cada artista $i \in \{1..M\}$ conocemos el beneficio o pérdida generado por dicho artista $B[i]$, es decir si $B[i] > 0$, dicho artista genera beneficio mientras que si $B[i] < 0$ genera pérdida. Diseñar un algoritmo de vuelta atrás que resuelva el problema de planificar el festival que maximice la suma de los beneficios/pérdidas de los artistas participantes.
10. [Examen Junio, 2014] Alonso Rodríguez tiene que hacer la compra de la semana. Ha hecho una lista de n productos que quiere comprar. En su barrio hay m supermercados en cada uno de los cuales se dispone de todos esos productos. Pero como es un

comprador compulsivo no quiere comprar más de tres productos en cada uno de los supermercados ya que así pasa más tiempo comprando (se puede suponer que $n \leq 3m$). Dispone de una lista de precios de los productos en cada uno de los supermercados. Se pide diseñar un algoritmo de vuelta atrás que permita a Alonso decidir cómo hacer la compra de forma que compre todo lo que necesita y que el coste total sea mínimo.

Bibliografía

*Y así, del mucho leer y del poco dormir, se le
secó el cerebro de manera que vino a perder el
juicio.*

Miguel de Cervantes Saavedra

- BRASSARD, G. y BRATLEY, P. *Fundamentos de Algoritmia*. Prentice Hall, 1997.
- CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L. y STEIN, C. *Introduction to Algorithms*. MIT Press, 2nd edición, 2001.
- MARTÍ OLIET, N., SEGURA DÍAZ, C. M. y VERDEJO LÓPEZ, J. A. *Algoritmos correctos y eficientes: Diseño razonado ilustrado con ejercicios*. Ibergarceta Publicaciones, 2012.
- MARTÍ OLIET, N., ORTEGA MALLÉN, Y. y VERDEJO LÓPEZ, J. A. *Estructuras y datos y métodos algorítmicos: 213 Ejercicios resueltos*. Ibergarceta Publicaciones, 2013.
- PEÑA, R. *Diseño de Programas: Formalismo y Abstracción*. Tercera edición. Pearson Prentice-Hall, 2005.
- RODRÍGUEZ ARTALEJO, M., GONZÁLEZ CALERO, P. A. y GÓMEZ MARTÍN, M. A. *Estructuras de datos: un enfoque moderno*. Editorial Complutense, 2011.