Tema 2 - Diseño de Algoritmos

César González Ferreras

Departamento de Informática
Universidad de Valladolid

Curso 2020/21

Contenidos

Contenidos

Recursividad

- Un objeto es recursivo cuando forma parte de sí mismo o se define en función de sí mismo.
- La recursividad, como la iteración, permite describir cálculos que, con pequeñas variaciones, se repiten un número dado de veces.
- La recursividad genera programas más compactos que la iteración y facilita la verificación formal pero consume más recursos.
- Ejemplo: el factorial de un número:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{si } n \geq 1 \end{cases}$$

- En todo algoritmo recursivo debe haber:
  - Uno o más casos básicos.
  - El resto son casos recurrentes: una forma de definir un caso genérico a partir de otros más sencillos, es decir más próximos a los casos básicos.

Recursividad

- El ejemplo del factorial lo podemos escribir en pascal:

Factorial

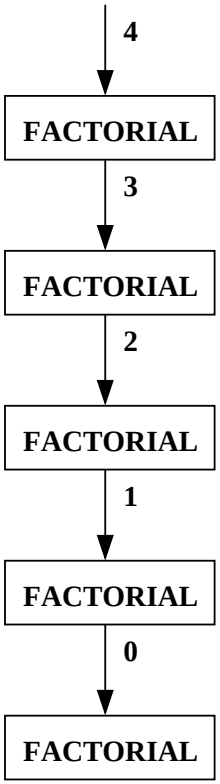
```
function factorial(n:integer):integer;
{calcula el factorial de n}
begin
  if n=0 then
    factorial := 1   {caso básico}
  else
    factorial := n * factorial(n-1); {caso genérico}
end;
```

Esquema general de recursividad

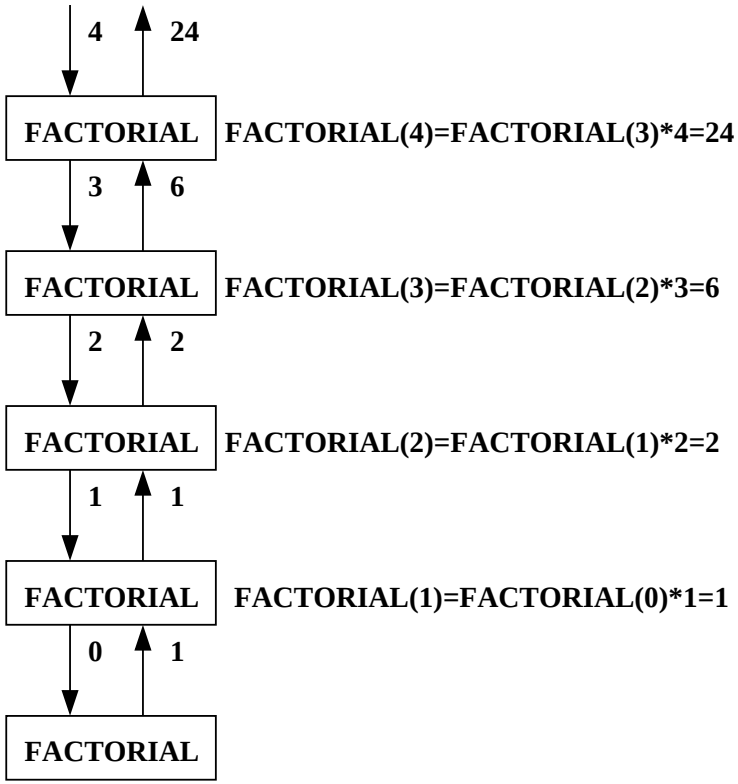
- Para plantear la solución recursiva debemos identificar:
  1. **Parámetros**: qué parámetros intervienen en la solución del problema, aquellos que nos permiten expresar el problema en subproblemas.
  2. **Casos básicos**: son aquellos que no provocan llamada recursiva.
  3. **Casos recurrentes**: son aquellos que sí producen una llamada recursiva. Estos casos plantean la solución en base a combinar la solución de subproblemas más sencillos.
- El proceso de ejecución de un subprograma recursivo consiste en una cadena de llamadas al subprograma, suspendiéndose los restantes cálculos.
- En cada llamada al subprograma recursivo:
  - Se reserva espacio en memoria necesario para almacenar los parámetros y los demás objetos locales del subprograma.
  - Se reciben los parámetros y se cede la ejecución de instrucciones al subprograma, que comienza a ejecutarse.
  - Al terminar éste, se libera el espacio reservado, los identificadores locales dejan de tener vigencia y pasa a ejecutarse la instrucción siguiente a la de la llamada.
- Una vez llegamos al caso base, se produce el retorno de cada una de las llamadas, realizándose cada uno de los cálculos.

Esquema general de recursividad

- Cadena de llamadas recursivas que se desencadenan al ejecutar 4!

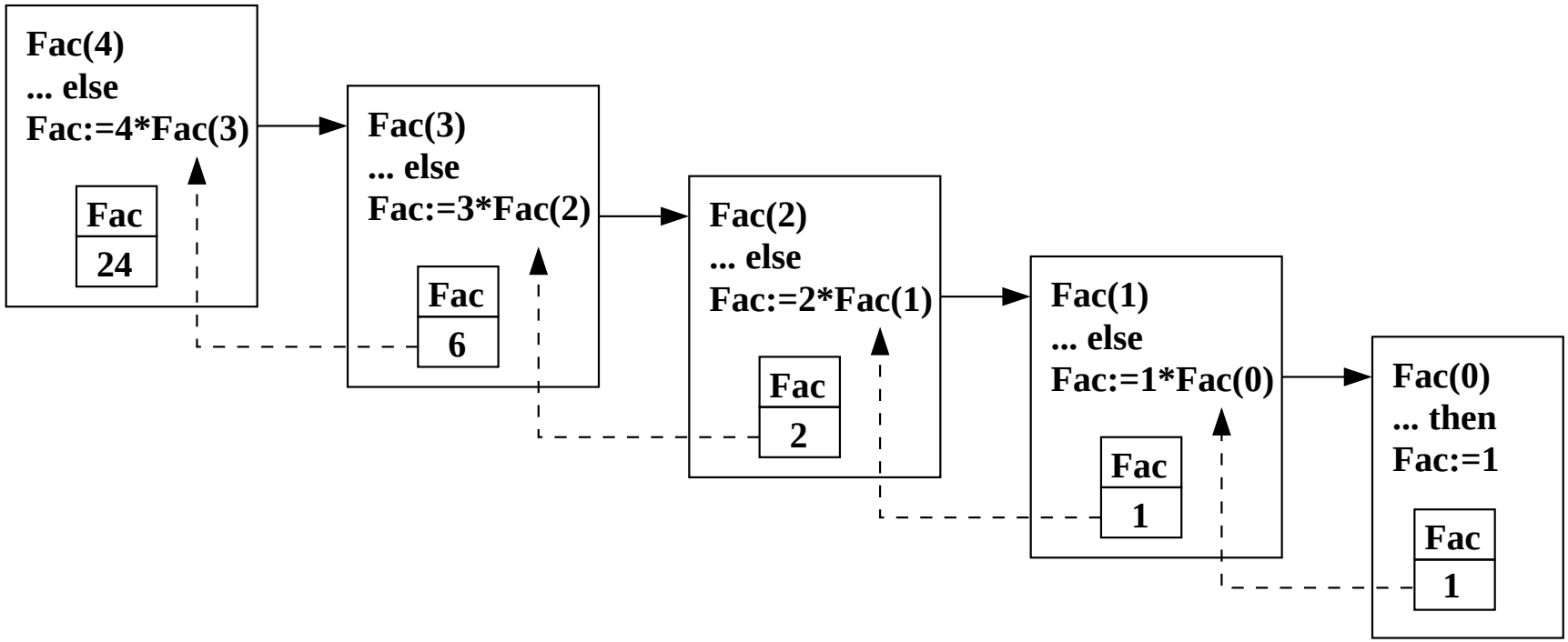


- Retorno de las llamadas recursivas al calcular 4!



Esquema general de recursividad

- Cada llamada genera un nuevo ejemplar del subprograma con sus correspondientes objetos locales.
- Esquema de las llamadas recursivas a la función factorial para el cálculo de 4!



Corrección de programas recursivos

- En primer lugar, debemos asegurarnos que el subprograma recursivo termina:
  - Debe existir un caso base.
  - Los casos recurrentes deben construir la solución combinando las soluciones de problemas más sencillos (más cercanos al caso base).
- A continuación hay que demostrar que devuelve un valor correcto. Para ello podemos utilizar el **principio de inducción**.
- El principio de inducción sirve para demostrar propiedades P(n) que dependen de una variable $n \in N$:
  - Si 0 cumple la propiedad P, y
  - Si para cualquier $x \in N$ se tiene que, si todo $y < x$ tiene la propiedad P (**hipótesis de inducción**), entonces x también tiene la propiedad P (**paso de inducción**).
  - Entonces todo $z \in N$ tiene la propiedad P.

Tipos de recursividad

- Podemos identificar distintos tipos de recursividad:
  - **Directa**: el subprograma se llama a sí mismo. **Indirecta o cruzada**: el subprograma se llama a sí mismo a través de otro.
  - **Lineal**: sólo una llamada recursiva. **Múltiple**: más de una llamada recursiva.
  - **Final**: la llamada recursiva es la última sentencia del algoritmo. En caso contrario la recursividad es **no final**.
- Un ejemplo de recursividad directa, lineal y final es el factorial que hemos visto en la sección anterior.
- Un ejemplo de recursividad no lineal es la que permite calcular la sucesión de números de Fibonacci.

Fibonacci recursivo

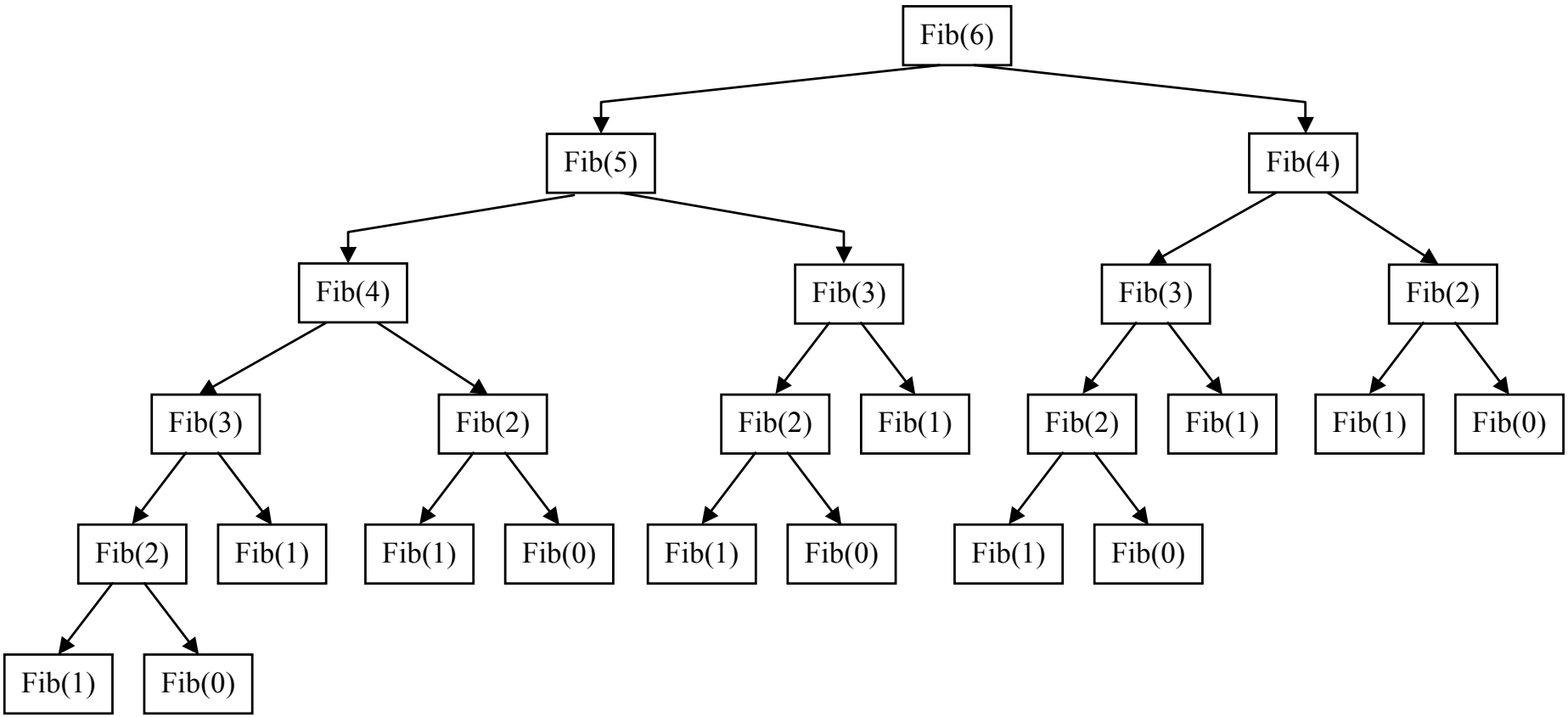
```
function fibo(n:integer):integer;
begin
  if (n=0) or (n=1) then
    fibo:= 1
  else
    fibo := fibo(n-1) + fibo(n-2);
end;
```

Recursión vs. iteración

- La recursión es una forma especial de iteración.
- En el caso recursivo empezamos a resolver el problema partiendo del problema completo y lo vamos descomponiendo en problemas más sencillos cada vez; a continuación componemos esas soluciones para resolver el problema completo.
- En el caso iterativo por contra, empezamos en el problema más sencillo y vamos construyendo la solución a problemas más grandes cada vez hasta construir la solución al problema completo.
- La solución recursiva a un problema produce un gasto adicional de memoria (para almacenar los datos de cada invocación a función) y más tiempo (en la invocación de las funciones).
- En general la solución iterativa es más eficiente frente a la recursiva.
- La solución recursiva es más sencilla que la iterativa, y puede ser más recomendable que la iterativa debido a que es más fácil de comprender y de demostrar su corrección.

Recursión vs. iteración

- Llamadas recursivas generadas al ejecutar *Fib*(6).



Recursión vs. iteración

Fibonacci iterativo

```
function fibo(n:integer): integer;
var i,x,anterior,actual:integer;
begin
  if n < 2 then
    fibo:= n
  else
  begin
    anterior :=0; actual:=1;
    for i:=2 to n do
    begin
      x := anterior;
      anterior := actual;
      actual := actual + x;
    end;
    fibo:= actual;
  end;
end;
```

Contenidos

Divide y vencerás

- Divide y vencerás es una técnica para diseñar algoritmos que consiste en:
  - Descomponer el caso que haya que resolver en un cierto número de subcasos más pequeños.
  - Resolver sucesiva e independientemente todos estos subcasos.
  - Combinar después las soluciones obtenidas de esta manera para obtener la solución del caso original.

Divide y vencerás: caso general

- Consideremos un problema arbitrario, y sea `ad_hoc` un algoritmo sencillo capaz de resolver el problema. Lo llamaremos **subalgorimo básico** y debe ser eficiente para casos pequeños.

Caso general

función DV(x)

if *x es suficientemente pequeño* **then**
  devolver ad_hoc(x)
else
  descomponer x en casos más pequeños $x_1, x_2, \ldots, x_k$;
  for $i \leftarrow 1$ **to** k **do**
    $y_i \leftarrow DV(x_i)$
  end
  recombinar los y_i para obtener una solución y de x;
  devolver y
end

Divide y vencerás: caso general

- Para que el enfoque divide y vencerás merezca la pena, es necesario que se cumplan tres condiciones:
  1. La decisión de utilizar el subalgoritmo básico debe tomarse cuidadosamente.
  2. Tiene que se posible descomponer el ejemplar en subejemplares y recomponer las soluciones parciales de forma bastante eficiente.
  3. Los subejemplares deben ser en lo posible del mismo tamaño.
- El análisis de tiempos de ejecución para estos algoritmos se hace mediante el teorema maestro.
- Si el tamaño de los k subejemplares es aproximadamente $\frac{n}{b}$ para alguna constante b, en donde n es el tamaño del caso original.

$$t(n) = k \cdot t\left(\frac{n}{b}\right) + g(n)$$

Ejemplo: búsqueda binaria

- Sea $T[1..n]$ un vector ordenado por orden no decreciente, esto es, $T[i] \leq T[j]$ siempre que sea $1 \leq i \leq j \leq n$.
- Sea x un elemento.
- El problema consiste en buscar x en el vector T, si es que está.
- Buscamos el índice i tal que $1 \leq i \leq n+1$ y $T[i-1] < x \leq T[i]$, con la convención de que $T[0] = -\infty$ y $T[n+1] = +\infty$.

Ejemplo: búsqueda binaria

- La solución más sencilla a este problema es la búsqueda secuencial:

Búsqueda secuencial

```
función secuencial(T[1..n], x)
for i ← 1 to n do
    if T[i] ≥ x then
        devolver i
    end
end
devolver n + 1
```

- Tanto en el caso promedio como en el peor caso, la búsqueda secuencial requiere un tiempo que está en $\Theta(n)$.

Ejemplo: búsqueda binaria

- Para acelerar la búsqueda, deberíamos buscar x o bien en la primera mitad del vector, o bien en la segunda.
- Para averiguar cuál de estas búsquedas es la correcta, comparamos x con un elemento del vector: $k = \lceil n/2 \rceil$.
- Si $x \leq T[k]$, entonces podemos restringir la búsqueda de x a $T[1..k]$.
- En caso contrario basta con buscar en $T[k+1..n]$.

Ejemplo: búsqueda binaria

Búsqueda binaria - recursiva

función busquedabin($T[1..n], x$)

if $n = 0$ ***o*** $x > T[n]$ **then**
| **devolver** $n + 1$
else
| **devolver** binrec($T[1..n], x$)
end

función binrec($T[i..j], x$)
{Búsqueda binaria de x en el subvector $T[i..j]$ con la seguridad de que $T[i-1] < x \leq T[j]$ }

if $i = j$ **then devolver** i;
$k \leftarrow (i + j)/2$;
if $x \leq T[k]$ **then**
| **devolver** binrec(T[i..k],x)
else
| **devolver** binrec(T[k+1..j],x)
end

Ejemplo: búsqueda binaria

Búsqueda binaria - iterativo

función biniter($T[1..n], x$)
{Búsqueda binaria iterativa de x en el vector T}

if $x > T[n]$ **then devolver** $n+1$;
$i \leftarrow 1; j \leftarrow n;$
while $i < j$ **do**
    $k \leftarrow (i+j)/2;$
    if $x \leq T[k]$ **then**
        $j \leftarrow k$
    else
        $i \leftarrow k+1$
    end
end
devolver i

Ejemplo: búsqueda binaria

- Búsqueda binaria de $x = 12$ en $T[1.,11]$.

1	2	3	4	5	6	7	8	9	10	11	
-5	-2	0	3	8	8	9	12	12	26	31	$x \leq T[k]$?
i					k					j	no
						i		k		j	sí
						i	k	j			sí
						ik	j				no
							ij				$i = j$: alto

Ejemplo: búsqueda binaria

- Coste en tiempo de la búsqueda binaria:
  - Sea $t(m)$ el tiempo requerido por una llamada a *binrec(T[i..j],x)*, donde $m = j - i + 1$ es el número de elementos sobre los que se busca.

$$t(m) = t\left(\frac{m}{2}\right) + g(m)$$

$$g(m) \in O(1)$$

  - Por tanto, la búsqueda binaria tiene un coste temporal:

$$t(m) \in \Theta(log(m))$$

Búsqueda binaria en Java

- java.util.Collections
  - https://docs.oracle.com/javase/10/docs/api/java/util/Collections.html
- java.util.Arrays
  - https://docs.oracle.com/javase/10/docs/api/java/util/Arrays.html

Ejemplo: ordenación

- Sea V[1..n] un vector de n elementos
- Queremos ordenar esos elementos en orden ascendente
- Hay dos algoritmos clásicos de ordenación que siguen el esquema de divide y vencerás:
  - Ordenación por fusión (*mergesort*)
  - Ordenación rápida (*quicksort*)

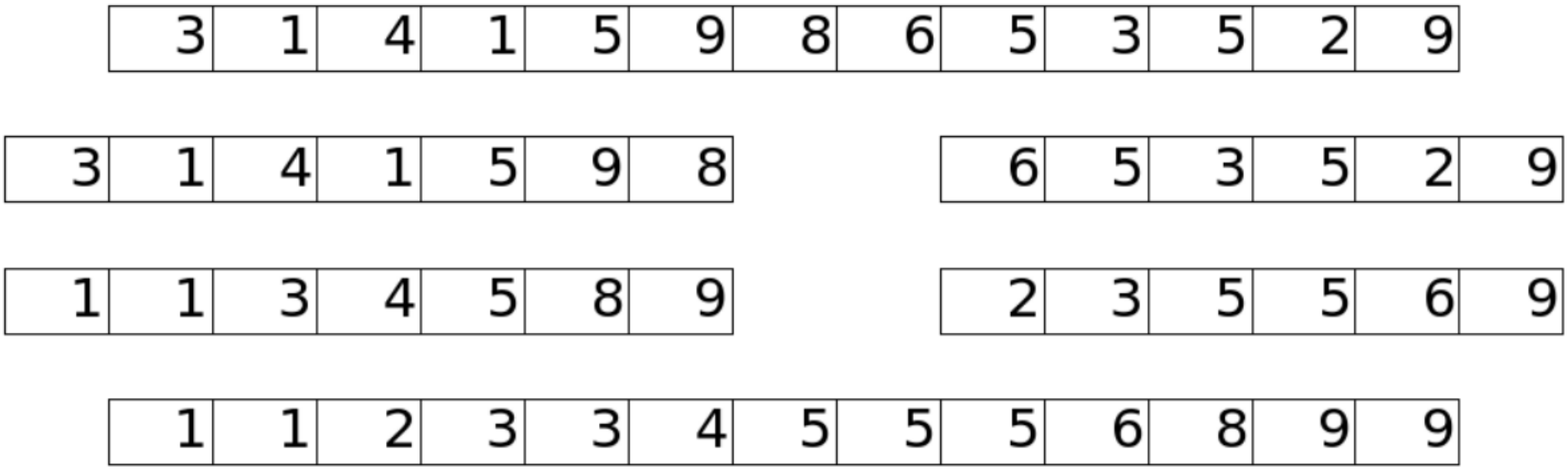
Ejemplo: ordenación por fusión

```
type
  TDato = record Clave: TClave; ... end;
  TVector = array of TDato;

procedure OrdFus(var V: TVector);  { Se supone que Low(V) = 0 }
var W : TVector; { Vector temporal }
begin
  SetLength(W,Length(V));
  OrdFusRec(V,W,0,High(V))
end;

procedure OrdFusRec(var V,W: TVector; Ini,Fin: integer);
{ Ordena V[Ini..Fin] }
var Med,I : integer;
begin
  if Ini < Fin then
  begin
    Med := (Ini+Fin) div 2;
    OrdFusRec(V,W,Ini,Med);     { Ordenación primera mitad }
    OrdFusRec(V,W,Med+1,Fin);   { Ordenación segunda mitad }
    Fusion(V,W,Ini,Med,Fin);    { Fusionar mitades en W }
    for I := Ini to Fin do V[I] := W[I] { Copiar a V }
  end { else caso_base }
end;
```

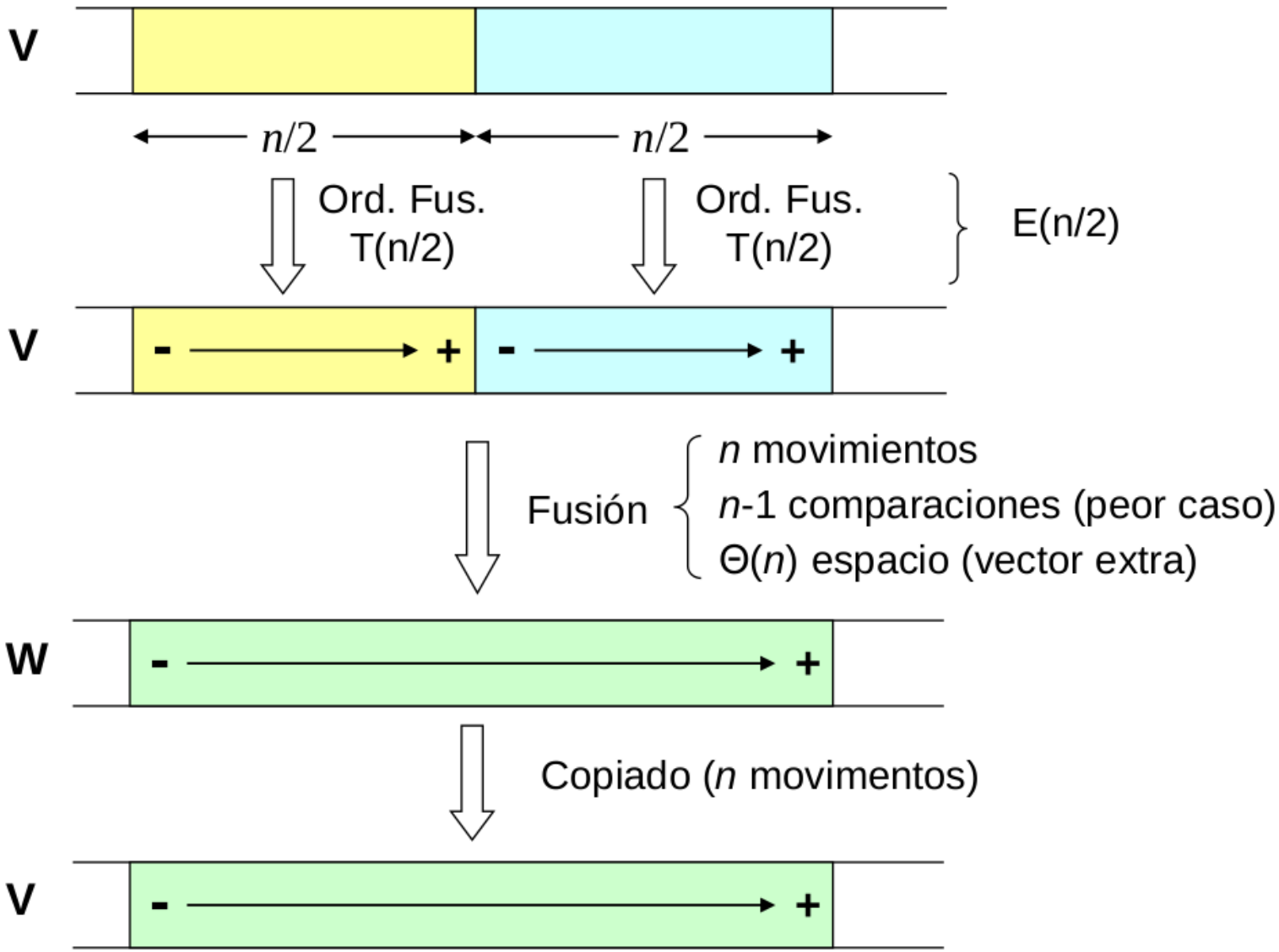
Ejemplo: ordenación por fusión

```
procedure Fusion(var V,W: TVector; Ini,Med,Fin: integer);
{ Fusiona V[Ini..Med] y V[Med+1..Fin] en W[Ini..Fin] }
var Ia,Ib,Ic : integer;
begin
  { "Extraer" mínimos y llevarlos a W }
  Ia := Ini; Ib := Med+1; Ic := Ini;
  while (Ia <= Med) and (Ib <= Fin) do
    if V[Ia].Clave < V[Ib].Clave then
    begin
      W[Ic] := V[Ia]; Inc(Ia); Inc(Ic)
    end else begin
      W[Ic] := V[Ib]; Inc(Ib); Inc(Ic)
    end;
  { Copiar zona no vacía a W }
  while Ia <= Med do
  begin
    W[Ic] := V[Ia]; Inc(Ia); Inc(Ic)
  end;
  while Ib <= Fin do
  begin
    W[Ic] := V[Ib]; Inc(Ib); Inc(Ic)
  end;
end;
```

Ejemplo: ordenación por fusión



Análisis ordenación por fusión



Propiedades ordenación por fusión

- Eficiencia $\Theta(n \log n)$
  - $2n \log_2 n$ movimientos
  - $n \log_2 n - n$ comparaciones (peor caso $\approx$ caso promedio)
  - n datos (vector extra) + $\Theta(\log n)$ espacio adicional
- Método universal
- Fácilmente adaptable a acceso secuencial (separando ambas mitades en estructuras distintas)
- Estable
- No adaptativo
- Sobre el propio vector (no se puede usar el vector extra para devolver el resultado)

Ordenación estable

- Se mantiene el orden relativo de los registros con igual clave.
- Ejemplo:
  - Vector a ordenar (la clave es el primer elemento):
    (**4**, 1) (**3**, 1) (**3**, 7) (**5**, 6)
  - Ordenación estable (se mantiene el orden relativo):
    (**3**, 1) (**3**, 7) (**4**, 1) (**5**, 6)
  - Ordenación no estable (no se mantiene el orden relativo):
    (**3**, 7) (**3**, 1) (**4**, 1) (**5**, 6)

Ejemplo: ordenación rápida

```
procedure OrdRap(var V: TVector);
var I,J : integer;
begin
  { Desordenar vector (Knuth shuffle algorithm) }
  for I := Low(V) to High(V)-1 do
    V[I] ⇔ V[Low(V)+Random(High(V)-I+1)]
  { Ordenación recursiva }
  OrdRapRec(V,Low(V),High(V))
end;

procedure OrdRapRec(var V: TVector; Ini,Fin: integer);
{ Ordena V[Ini..Fin] }
var Fin_Men,Ini_May : integer;
begin
  if Ini < Fin then
  begin
    Particion(V,Ini,Fin,Fin_Men,Ini_May);
    OrdRapRec(V,Ini,Fin_Men);
    OrdRapRec(V,Ini_May,Fin);
  end { else caso_base }
end;
```

Opcional

Ejemplo: ordenación rápida

```
procedure Partición(var V: TVector; Ini,Fin: integer;
                    var Fin_Men, Ini_May: integer);
{ Reorganiza V[Ini..Fin] de manera que termine organizado en tres zonas:
  · V[Ini..Fin_Men] contiene elementos menores que el pivote.
  · V[Fin_Men+1] contiene el pivote.
  · V[Ini_May..Fin] contiene elementos mayores o iguales al pivote.
  IniMay = Fin_Men+2 }
var
  I,Lim : integer;
begin
  { Se toma como pivote el 1er elemento, V[Ini] (hay otras alternativas) }
  Lim := Ini+1;
  for I := Ini+1 to Fin do
    { Invariante: V[Ini+1..Lim-1] < Pivote, V[Lim..I-1] >= Pivote }
    if V[I].Clave < V[Ini].Clave then
    begin
      V[I] ⇔ V[Lim];
      Inc(Lim)
    end; { if-for }
  V[Ini] ⇔ V[Lim-1]; { Pivote entre menores y mayores/iguales }
  Fin_Men := Lim-2; Ini_May := Lim
end;
```

Ejemplo: ordenación rápida

12	14	18	14	4	19	14	4	5	16	1	3	4	11	11

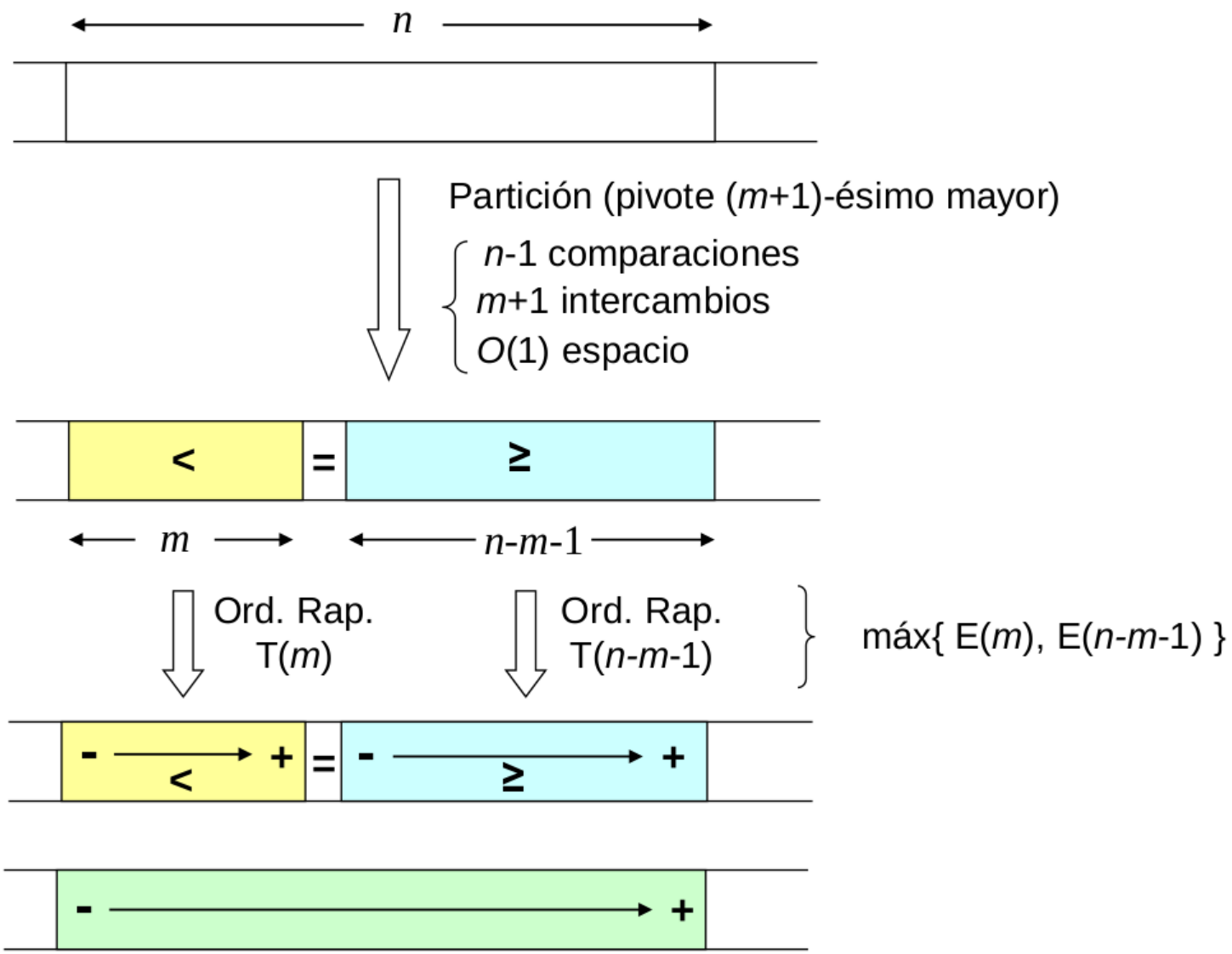
Pivote: 12

12	14	18	14	4	19	14	4	5	16	1	3	4	11	11
12	14	18	14	4	19	14	4	5	16	1	3	4	11	11
12	14	18	14	4	19	14	4	5	16	1	3	4	11	11
12	14	18	14	4	19	14	4	5	16	1	3	4	11	11
12	4	18	14	14	19	14	4	5	16	1	3	4	11	11
12	4	18	14	14	19	14	4	5	16	1	3	4	11	11
12	4	18	14	14	19	14	4	5	16	1	3	4	11	11
12	4	4	14	14	19	14	18	5	16	1	3	4	11	11
12	4	4	5	14	19	14	18	14	16	1	3	4	11	11
12	4	4	5	14	19	14	18	14	16	1	3	4	11	11
12	4	4	5	1	19	14	18	14	16	14	3	4	11	11
12	4	4	5	1	3	14	18	14	16	14	19	4	11	11
12	4	4	5	1	3	4	18	14	16	14	19	14	11	11
12	4	4	5	1	3	4	11	14	16	14	19	14	18	11
12	4	4	5	1	3	4	11	11	16	14	19	14	18	14

11	4	4	5	1	3	4	11	12	16	14	19	14	18	14

1	3	4	4	4	5	11	11	12	14	14	14	16	18	19

Análisis ordenación rápida



Mejor caso ordenación rápida

- Se da cuando las particiones son equilibradas ($m \approx n/2$)
- $T(n) = 2T(n/2) + \Theta(n)$, $E(n) = E(n/2) + O(1)$
- Comparaciones: $n \log_2 n + O(n)$ [igual a ord. fusión]
- Movimientos: $1.5\, n \log_2 n + O(n)$ [mejor que ord. fusión]
- Espacio: $\Theta(\log n)$ [mucho mejor que ord. fusión]

Peor caso ordenación rápida

- Se da cuando las particiones son desequilibradas: Una de las zonas está vacía ($m = 0$ ó $m = n\text{-}1$)
- $T(n) = T(n\text{-}1) + \Theta(n)$, $E(n) = E(n\text{-}1) + O(1)$
- Comparaciones: $n^2/2 + O(n)$
- Movimientos: $\Theta(n)$ si $m = 0$, $1.5\ n^2 + O(n)$ si $m = n\text{-}1$
- Espacio: $\Theta(n)$
- La ordenación rápida es un algoritmo $O(n^2)$
- Tipos de vectores que provocan el peor caso:
  - Pivote primero o último: Vector ordenado o en orden inverso
  - Pivote elem. medio: Vector con elementos en orden creciente hasta la mitad y decreciente a partir de entonces, o al revés.
  - Pivote al azar: La probabilidad de caer en el peor caso decrece exponencialmente.

Caso promedio ordenación rápida

- Relación de recurrencia general (2º método partición)

$$T(n,m) = T(m,\cdot) + T(n\text{-}m\text{-}1,\cdot) + f(n,m)$$

- Donde m es el número de elementos menores que el pivote
- $f(n,m)$ son las operaciones no recursivas (partición):
  - n-1 comparaciones
  - m+2 intercambios
- Si el pivote es un elemento cualquiera de la zona, entonces cualquier valor de $m \in [0..n\text{-}1]$ es equiprobable.
- Número de operaciones promedio:

$$\widehat{T}(n) = \frac{1}{n} \sum_{m=0}^{n-1} T(n,m)$$

Caso promedio ordenación rápida

- Aplicandolo a la relación de recurrencia (comparaciones):

$$\widehat{T}(n)=\frac{1}{n}\sum_{m=0}^{n-1}\left(\widehat{T}(m)+\widehat{T}(n-m-1)+n-1\right)$$

- Operando:

$$\widehat{T}(n)=\frac{2}{n}\sum_{m=0}^{n-1}\widehat{T}(m)+n-1$$

$$n\cdot\widehat{T}(n)=2\sum_{m=0}^{n-1}\widehat{T}(m)+n\cdot(n-1) \qquad \text{(eq. A)}$$

$$(n-1)\cdot\widehat{T}(n-1)=2\sum_{m=0}^{n-2}\widehat{T}(m)+(n-1)\cdot(n-2) \qquad \text{(eq. B)}$$

Caso promedio ordenación rápida

- Restando las ecuaciones A y B:

$$n \cdot \tilde{T}(n) - (n-1) \cdot \tilde{T}(n-1) = 2 \cdot \tilde{T}(n-1) + 2(n-1)$$

$$n \cdot \tilde{T}(n) = (n+1) \cdot \tilde{T}(n-1) + 2(n-1)$$

$$\frac{\tilde{T}(n)}{n+1} = \frac{\tilde{T}(n-1)}{n} + \frac{2(n-1)}{n \cdot (n+1)}$$

$$\frac{\tilde{T}(n)}{n+1} = \frac{\tilde{T}(n-1)}{n} + \frac{4}{n} - \frac{2}{n+1}$$

$$\tilde{T}(n) = n \cdot \ln n + O(n) = 1.44 \cdot n \cdot \log_2 n$$

Propiedades ordenación rápida

- Eficiencia $O(n^2)$
  - Peor caso: $\Theta(n^2)$ tiempo, $\Theta(n)$ espacio.
  - Mejor caso: $\Theta(n \log n)$ tiempo, $\Theta(\log n)$ espacio
  - Promedio: $\Theta(n \log n)$ tiempo, $\Theta(\log n)$ espacio
  - El tiempo promedio sólo es un 40% mayor que el mejor
- Método universal
- Acceso secuencial: No.
- No Estable
- No adaptativo $\rightarrow$ Antiadaptativo
- Sobre el propio vector

Ordenación en Java

- java.util.Arrays
  - https://docs.oracle.com/javase/10/docs/api/java/util/Arrays.html
  - Ordenación de tipos básicos
  - Implementación: quicksort (ordenación rápida)
  - **No necesita ser estable**:
    - Tipos básicos no tienen información asociada.
- java.util.Collections
  - https://docs.oracle.com/javase/10/docs/api/java/util/Collections.html
  - Ordenación de tipos complejos
  - Usa una variante de mergesort (ordenación por fusión)
    - TimSort (https://en.wikipedia.org/wiki/Timsort)
  - **Es estable**

Contenidos

Fuerza bruta: caso general

- Fuerza bruta consiste en enumerar de forma sistemática todos los posibles candidatos a solución y comprobar si cada uno de ellos es una solución válida.
- Sencillo de implementar.
- Siempre encuentra la solución cuando existe.
- Coste proporcional al número de candidatos: tiende a crecer muy rápido.
- Se usa en problemas de tamaño muy pequeño.
- Se usa en los casos en los que es más importante la sencillez que la velocidad.
- Siempre que se pueda buscar una solución mejor.

Esqueleto fuerza bruta

```
c ← primera_solución_candidata();
while c ≠ candidato_nulo do
    if valida(c) then
        imprimir(c)
    end
    c ← siguiente_solución_candidata(c);
end
```

Ejemplo: 8 reinas

- **Problema de las 8 reinas**: consiste en colocar 8 reinas en un tablero de ajedrez (8x8) de manera que ninguna de ellas amenace a otra.
- Solución por fuerza bruta:
  - 8 reinas: todas posibilidades 64^8 formas de colocar 8 reinas.
  - Si eliminamos las que la misma reina está en la misma casilla $\frac{64!}{56!}$
- Una forma de mejorar las soluciones de fuerza bruta es tratar de reducir el espacio de búsqueda, es decir, el número de soluciones candidatas a comprobar:
  - 1 reina por fila 8^8.
  - 8! si ninguna reina en misma columna.

Backtracking

- Backtracking es una técnica de resolución de problemas basado en la búsqueda sistemática y en que la solución se puede ir construyendo en varios pasos (solución parcial).
- Representamos la solución como un vector $a = (a_1, a_2, \ldots, a_n)$.
- En cada paso del algoritmo partimos de una solución parcial $a = (a_1, a_2, \ldots, a_k)$ y tratamos de extenderla añadiendo otro elemento al final $a = (a_1, a_2, \ldots, a_k, a_{k+1})$.
- Después de extender comprobamos si lo que tenemos es una solución parcial válida:
  - En caso afirmativo:
    - Si es solución completa: imprimir.
    - En otro caso: seguimos adelante.
  - En caso negativo:
    - Retroceso (backtrack): eliminamos el último elemento añadido y probamos otro.
- Es más eficiente que la búsqueda exhaustiva.

Backtracking: caso general

Esqueleto backtracking

```
procedimiento ensayar(entero : i, ...)
begin
    inicializa las posibilidades de extensión;
    while queden posibilidades do
        toma la siguiente posibilidad;
        if la posibilidad es aceptable then
            anota la posibilidad como parte de la solución;
            if se encontró solución general then
                presenta la solución obtenida;
            else
                ensayar(i+1,...);
            end
            desanota la posibilidad como parte de la solución;
        end
    end
end
```

Ejemplo: 8 reinas

8 reinas

```
{Backtracking - coloca la reina en la fila n
   n- fila en la que colocar reina
   psol- solución parcial}
procedure ensaya(n:integer;var psol:pos);
var i:integer;
begin
  for i:=1 to TAM do
  begin
    if aceptable(n,i,psol) then
    begin
      psol[n]:=i;
      if n = TAM then
        mostrar(psol)
      else
        ensaya(n+1,psol);
    end;
  end;
end;
```

Ejemplo: 8 reinas

8 reinas

```
{Comprueba si una solución parcial es aceptable
   f,c- fila y columna donde se coloca nueva reina
   psol- la solución parcial a comprobar}
function aceptable(f,c:integer;var psol:pos):boolean;
var i:integer;
begin
  aceptable:=true;
  for i:=1 to f-1 do
    if psol[i] = c then
      aceptable:=false;
  for i:=1 to f-1 do
    if psol[f-i] = c-i then
      aceptable:=false;
  for i:=1 to f-1 do
    if psol[f-i] = c+i then
      aceptable:=false;
end;
```

Ejemplo: 8 reinas

8 reinas

```
const TAM=8;
type pos=array[1..TAM] of integer;

var psol: pos; {solución parcial}
begin
  ensaya(1,psol);
end.
```

Ejemplo: 8 reinas

8 reinas

```
{Muestra una solución completa por pantalla
   psol- solución a mostrar}
procedure mostrar(var psol:pos);
var i,j:integer;
begin
  writeln('-----------------');
  for i:=1 to TAM do
  begin
    for j:=1 to TAM do
      if psol[i]=j then
        write('X|')
      else
        write(' |');
    writeln();
    writeln('-----------------');
    end;
end;
```

Contenidos

Algoritmos voraces

- Aplican un enfoque *miope* y toman decisiones basándose en la información que tiene disponible de modo inmediato, sin tener en cuenta los efectos que estas decisiones puedan tener en un futuro.
- Resultan fáciles de inventar y fáciles de implementar.
- Hay muchos problemas que no se pueden resolver correctamente con un enfoque tan grosero.
- Los algoritmos voraces se utilizan típicamente para resolver problemas de optimización.

Ejemplo: dar la vuelta

- **Dar la vuelta**: pagar una cierta cantidad a un cliente, utilizando el menor número de monedas.
- Monedas disponibles: 100 pesetas, 25 pesetas, 10 pesetas, 5 pesetas y 1 peseta.
- Todos nosotros solemos resolver este problema de forma voraz: empezamos por nada, y en cada fase vamos añadiendo a las monedas que ya estén seleccionadas una moneda de la mayor denominación posible, pero que no debe llevarnos más allá de la cantidad que haya que pagar.

Ejemplo: dar la vuelta

Dar la vuelta

función devolver_cambio(n):conjunto de monedas
{Da el cambio de n unidades utilizando el menor número posible de monedas. La constante C especifica las monedas disponibles}

begin
  const C={100,25,10,5,1};
  $S \leftarrow \emptyset$ {S es un conjunto que contendrá la solución };
  $s \leftarrow 0$ {s es la suma de los elementos de S};
  while $s \neq n$ **do**
    $x \leftarrow$ el mayor elemento de C tal que $s + x \leq n$;
    if *no existe ese elemento* **then**
      devolver “no encuentro la solución”;
    end
    $S \leftarrow S \cup$ {una moneda de valor x};
    $s \leftarrow s + x$
  end
  devolver S;
end

Ejemplo: dar la vuelta

- El algoritmo es voraz porque en cada paso selecciona la mayor de las monedas que pueda encontrar, sin preocuparse por lo correcto de esta decisión a la larga.
- El algoritmo nunca cambia de opinión.
- Con los valores dados para las monedas y disponiendo de un suministro adecuado de cada denominación, este algoritmo siempre produce una solución óptima para nuestro problema.
- Con una serie de valores diferente, o si el suministro de alguna de las monedas está limitado, el algoritmo voraz puede no funcionar.

Características generales de los algoritmos voraces

- Tenemos que resolver algún problema de forma óptima.
- Disponemos de un conjunto de candidatos.
- Según avanza el algoritmo tenemos 2 conjuntos:
  - Uno que contiene los candidatos que han sido considerados y seleccionados.
  - Uno que contiene los candidatos que han sido considerados y rechazados.
- Existe una función que comprueba si un conjunto de candidatos constituye una solución.
- Hay una función que comprueba si un cierto conjunto de candidatos es factible.
- Hay una función de selección que indica cuál es el más prometedor de los candidatos restantes.
- Existe una función objetivo que da el valor de la solución.

Características generales de los algoritmos voraces

Esqueleto algoritmo voraz

función voraz(C : *conjunto*) : *conjunto*
{C es el conjunto de candidatos}

```
begin
    S ← ∅ {Construimos la solución en el conjunto S};
    while C ≠ ∅ y no solución(S) do
        x ←seleccionar(C);
        C ← C − {x};
        if factible(S ∪ {x}) then
            S ← S ∪ {x};
        end
    end
    if solución(S) then
        devolver S
    else
        devolver "no hay soluciones"
    end
end
```

Contenidos

Programación dinámica

- La estrategia divide y vencerás puede provocar subejemplares solapados. Es decir, subejemplares idénticos que resolvemos más de una vez.
- El resultado es un algoritmo ineficiente.
- La programación dinámica trata de evitar calcular dos veces una misma cosa, normalmente manteniendo una tabla de resultados conocidos que se va llenando a medida que se resuelven los subcasos.
- La programación dinámica es una técnica ascendente:
  - Se empieza por los subcasos más pequeños.
  - Combinando sus soluciones obtenemos solución para subcasos de tamaños cada vez mayores.
  - Hasta que llegamos a la solución del caso original.

Ejemplo: coeficiente binomial

- Cálculo del coeficiente binomial:

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \\ 0 & \text{en caso contrario} \end{cases}$$

- Supongamos $0 \leq k \leq n$. Si calculamos $\binom{n}{k}$ mediante:

Coeficiente binomial recursivo

```
función C(n, k)
begin
    if k = 0 o k = n then
        devolver 1;
    else
        devolver C(n-1,k-1) + C(n-1,k);
    end
end
```

- Muchos de los valores $C(i, j)$, con $i < n$, $j < k$ se calculan una y otra vez.

Ejemplo: coeficiente binomial

- Podemos utilizar una tabla de resultados intermedios:

n/k	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

- En realidad basta con mantener un vector k, que representa la linea actual y actualizar el vector de derecha a izquierda.
- Para calcular $\binom{n}{k}$ el algoritmo requiere un tiempo que está en $\Theta(nk)$ y un espacio que está en $\Theta(k)$.

Ejemplo: coeficiente binomial

Coeficiente binomial - programación dinámica

```
function bino(n,k:integer):integer;
var i,j: integer;
    c:   array of integer;
begin
  setlength(c,k+1);
  for i:=0 to n do begin
    for j:=k downto 0 do begin
      if (j=0) or (j=i) then
        c[j]:=1
      else if (j<i) then
        c[j] := c[j-1] + c[j]
    end;
  end;
  bino:=c[k];
end;
```

Principio de optimalidad

- Principio de optimalidad: en una sucesión óptima de decisiones u opciones, toda subsecuencia debe ser también óptima.
- Aunque el único valor de la tabla que nos interesa sea el último ($C(n, k)$ en el ejemplo del coeficiente binomial) todas las demás entradas de la tabla deben representar también selecciones óptimas.
- Cuando el principio de optimalidad no es aplicable, es probable que no sea posible atacar el problema en cuestión empleando programación dinámica.
- Por ejemplo, si el **camino más corto** entre Montreal y Toronto pasa por Kingston, entonces la parte del camino que va desde Montreal hasta Kingston también debe de seguir el camino más corto posible, al igual que la parte del camino que une Kingston con Toronto. Por tanto, es aplicable el principio de optimalidad.

Ejemplo: problema de la mochila (sin fragmentar)

- Tenemos n objetos y una mochila.
- Los objetos **no se pueden fragmentar** en trozos más pequeños.
- Para $i = 1, 2, ..., n$:
  - El objeto i tiene un peso positivo w_i.
  - El objeto i tiene un valor positivo v_i.
- La mochila puede llevar un peso que no supere W.
- Nuestro objetivo es llenar la mochila de tal forma que se maximice el valor de los objetos incluidos, respetando la limitación de capacidad.
- Sea x_i igual a 0 si decidimos no tomar el objeto i, o bien 1 si incluimos el objeto i.
- El problema se puede expresar como:

$$\text{maximizar} \sum_{i=1}^{n} x_i \cdot v_i \qquad \text{con la restricción} \sum_{i=1}^{n} x_i \cdot w_i \leq W$$

donde $v_i > 0$, $w_i > 0$ y $x_i \in \{0, 1\}$ para $1 \leq i \leq n$.

Ejemplo: problema de la mochila (sin fragmentar)

- Para resolver el problema mediante programación dinámica, preparamos una tabla $V[1..n, 0..W]$ que tiene una fila para cada objeto disponible, y una columna para cada peso desde 0 hasta W.
- $V[i, j]$ será el valor máximo de los objetos que podemos transportar si el límite de peso es j, con $0 \leq j \leq W$, y si solamente incluimos los objetos numerados desde 1 hasta i, con $1 \leq i \leq n$.
- La solución se encontrará en $V[n, W]$.
- En la situación general, $V[i, j]$ será el máximo de:
  - $V[i-1, j]$ si el objeto i no se añade a la carga.
  - $V[i-1, j-w_i] + v_i$ si se selecciona el objeto i.

$$V[i, j] = \text{máx}(V[i-1, j], V[i-1, j-w_i] + v_i)$$

- $V[0, j]$ será 0 cuando $j \geq 0$.
- $V[i, j]$ será $-\infty$ para todo i cuando $j < 0$.

Ejemplo: problema de la mochila (sin fragmentar)

- Ejemplo de funcionamiento del algoritmo:

  peso máximo mochila: $W = 11$

	0	1	2	3	4	5	6	7	8	9	10	11
$w_1 = 1, v_1 = 1$	**0**	1	1	1	1	1	1	1	1	1	1	1
$w_2 = 2, v_2 = 6$	**0**	1	6	7	7	7	7	7	7	7	7	7
$w_3 = 5, v_3 = 18$	0	1	6	7	7	**18**	19	24	25	25	25	25
$w_4 = 6, v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	**40**
$w_5 = 7, v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	**40**

- La tabla V nos permite también recuperar la composición de la carga óptima. Para ello, partimos de $V[n, W]$ y vamos determinando la decisión que tomó el algoritmo en cada paso:
  - $V[5, 11] = V[4, 11]$: el objeto 5 no se incluye.
  - $V[4, 11] = V[3, 11 - w_4] + v_4$: el objeto 4 se incluye.
  - $V[3, 5] = V[2, 5 - w_3] + v_3$: el objeto 3 se incluye.
  - $V[2, 0] = V[1, 0]$: el objeto 2 no se incluye.
  - $V[1, 0] = V[0, 0]$: el objeto 1 no se incluye.
  - Por tanto, la carga óptima consta de los objetos 3 y 4.

Ejemplo: problema de la mochila (sin fragmentar)

Problema de la mochila - los objetos no se pueden fraccionar

```
función mochilaPD(w[1..N],v[1..N],W):vector[1..N]
inicio
  matriz T[1..N, 0..W]
  vector s[1..N]

  para i <- 1 hasta N hacer
    para j <- 0 hasta W hacer
      si i=1 y j<w[i] entonces T[i,j] <- 0
      si_no si i=1    entonces T[i,j] <- v[i]
      si_no si j<w[i] entonces T[i,j] <- T[i-1,j]
      si_no T[i,j] <- max(T[i-1,j],T[i-1,j-w[i]]+v[i])
    fin_para
  fin_para

...
```

Ejemplo: problema de la mochila (sin fragmentar)

Problema de la mochila - los objetos no se pueden fraccionar

```
...
  {Obtenemos una de las soluciones}
  para i <- 1 hasta N hacer s[i] <- 0;
  i <- N
  j <- W
  mientras (i>0) hacer
    si ((i>1) AND (T[i,j] = T[i-1,j])) OR
       ((i=1) AND (T[i,j] = 0)) entonces
      i = i-1;
    si_no
      s[i] = 1;
      j = j - w[i];
      i = i-1;
    fin_si
  fin_mientras

  devolver s
fin
```

Ejemplo: problema de la mochila (sin fragmentar)

- El tiempo que se tarda en construir la tabla V es $\Theta(nW)$.
- El tiempo en determinar la carga óptima a partir de la tabla es $\Theta(n)$.