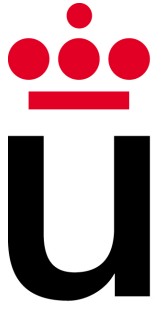
Divide y vencerás

Diseño y Análisis de Algoritmos



Universidad
Rey Juan Carlos

Contenidos

Introducción

Divide y vencerás

- Uno de los más importantes paradigmas de diseño algorítmico
- Se basa en la resolución de un problema dividiéndolo en dos o más subproblemas de igual tipo o similar
- El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente
- Al final, las soluciones a cada uno de los subproblemas se combinan para dar una solución al problema original
- Muchos algoritmos basados en esta estrategia son recursivos
  - También hay algoritmos iterativos de divide y vencerás

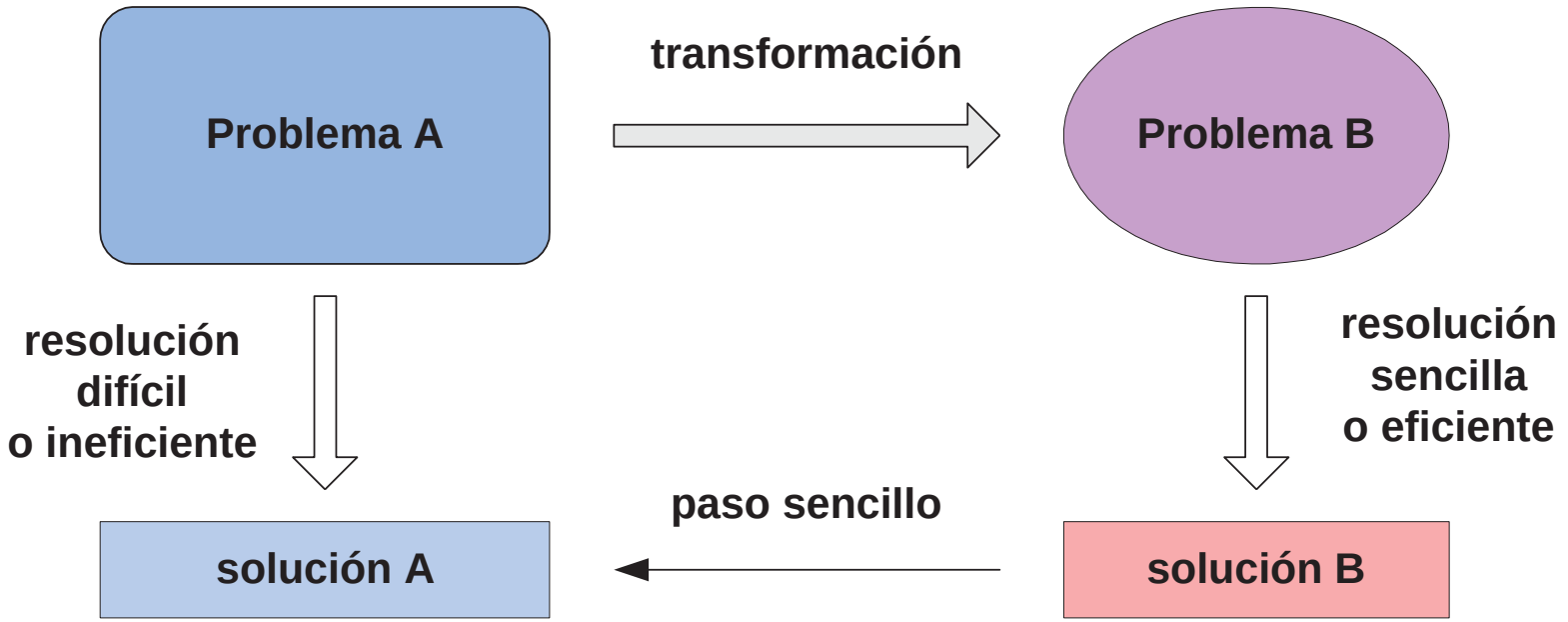
Simplificación y descomposición de problemas

- Las palabras clave en este tema van a ser la **simplificación** y la **descomposición** de problemas
- Veremos tres estrategias para simplificar y descomponer problemas:
  - **Transforma y vencerás**
  - **Decrementa y vencerás**
  - **Divide y vencerás**

Transforma y vencerás

Transforma y vencerás

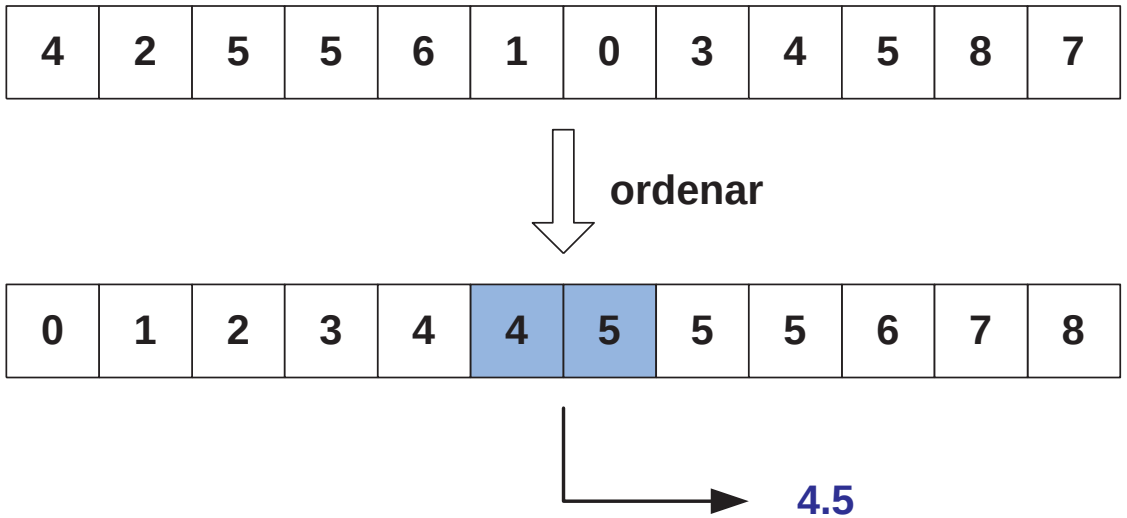
- Esta estrategia también se denomina **reducción**
- Esta técnica procura resolver un problema transformándolo en otro más simple, conocido, o para el que existan algoritmos eficientes



Encontrar la mediana de un array desordenado

- Sea un array v de n números

1. Ordenar el array $v \longrightarrow \Theta(n \log n)$
2. Devolver
   - $v[(n+1)/2]$ si n es impar
   - $\dfrac{v[n/2] + v[1 + n/2]}{2}$ si n es par



Estimación del valor de π

- Planteamos un problema alternativo
- Buscamos una **aproximación del perímetro p de un círculo** de radio R

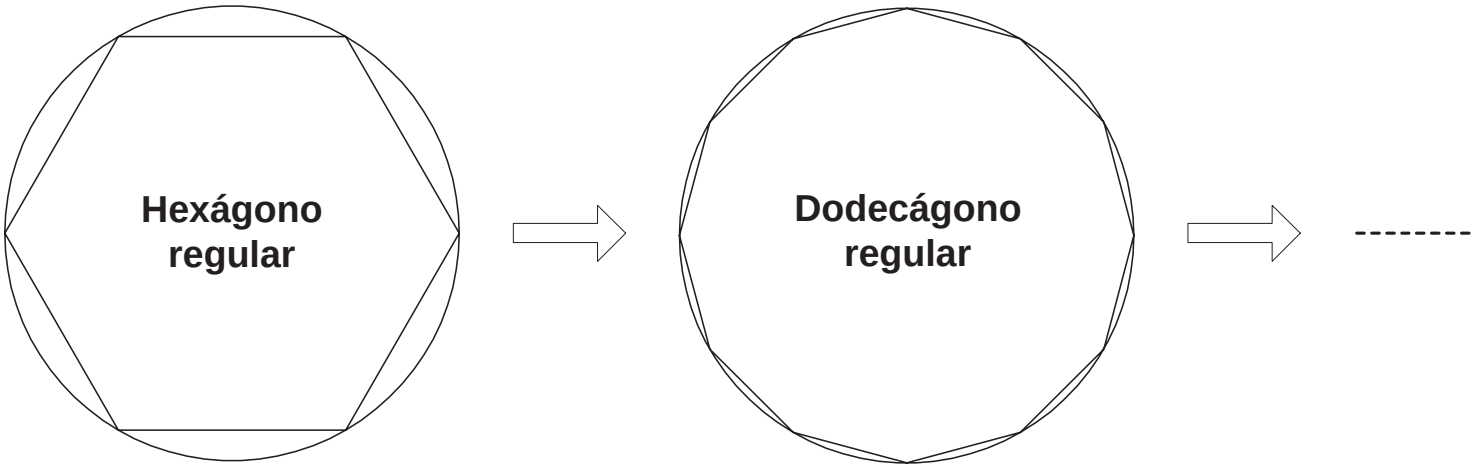
$$p = 2\pi R \quad \Rightarrow \quad \pi = \frac{p}{2R}$$

- Aproximamos un círculo con un polígono regular inscrito o circunscrito al círculo
- El perímetro del polígono inscrito p_i nos dará una cota inferior de p
- El perímetro del polígono circunscrito p_c nos dará una cota superior de p

$$\frac{p_i}{2R} \leq \pi \leq \frac{p_c}{2R}$$

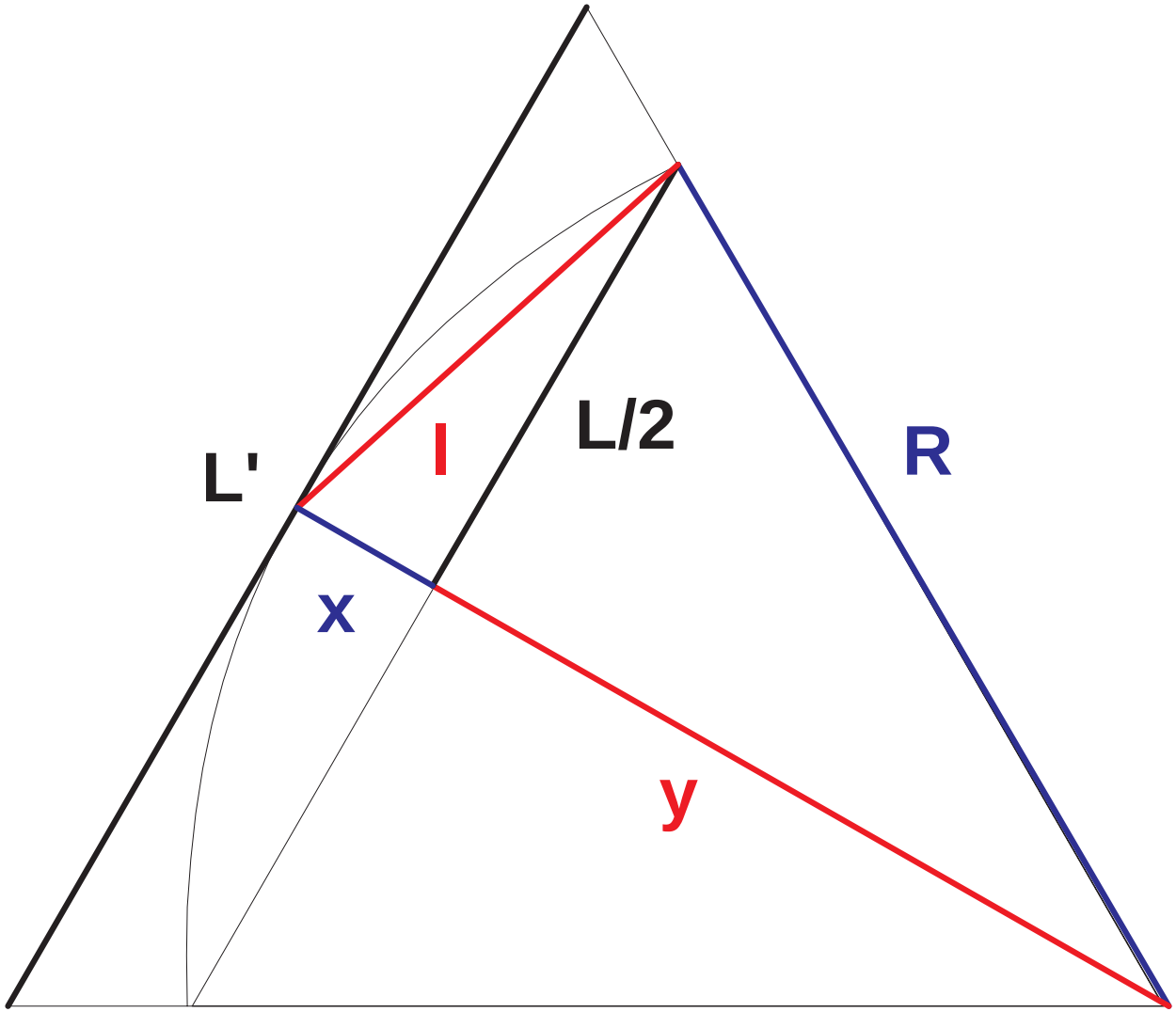
Estimación del valor de π

- Comenzamos por un polígono regular inscrito (por ejemplo, un hexágono), cuyo perímetro será $6R$
- A continuación, doblamos el número de lados, obteniendo una mejor aproximación



- El perímetro de un polígono con el doble de lados se puede obtener fácilmente a partir del primero
- Los perímetros de los polígonos circunscritos se pueden obtener de los perímetros de los polígonos inscritos

Estimación del valor de π



Estimación del valor de π

- Si el lado de un polígono inscrito mide L, se puede hallar la longitud un lado del polígono l con el doble de lados

$$R^2 = y^2 + \left(\frac{L}{2}\right)^2 \quad \Rightarrow \quad y = \sqrt{R^2 - \left(\frac{L}{2}\right)^2} \qquad x = R - y = R - \sqrt{R^2 - \left(\frac{L}{2}\right)^2}$$

$$l = \sqrt{\left(\frac{L}{2}\right)^2 + R^2 + R^2 - \left(\frac{L}{2}\right)^2 - 2R\sqrt{R^2 - \left(\frac{L}{2}\right)^2}} = R\sqrt{2 - \sqrt{4 - \left(\frac{L}{R}\right)^2}}$$

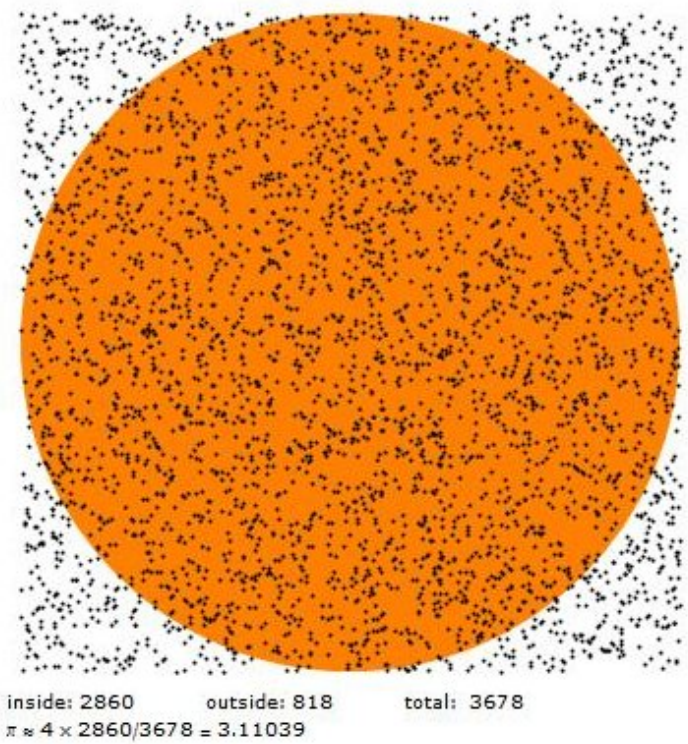
- El lado L' de un polígono circunscrito se puede hallar a partir del lado L de un polígono inscrito

$$\frac{L'}{R} = \frac{L}{y} \quad \Rightarrow \quad L' = \frac{LR}{y}$$

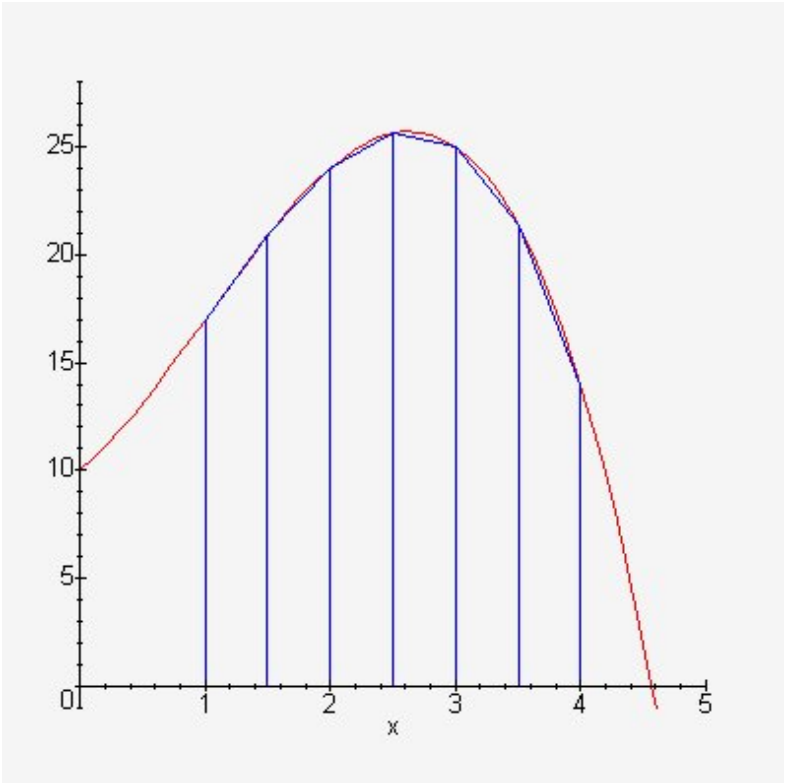
- la estimación de π se obtiene a partir de los perímetros hallados:

$$\frac{L \cdot \text{número de lados}}{2R} \leq \pi \leq \frac{L' \cdot \text{número de lados}}{2R}$$

Otros métodos para estimar del valor de π



Simulación Monte Carlo



Integración numérica

- Series

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots \qquad \text{para } x = 1 \quad \Rightarrow \quad \frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$$

Raíz cuadrada

- Deseamos hallar $\sqrt{a}$
- Transformamos el problema en el de **hallar un cero de una función**

$$x = \sqrt{a} \quad \rightarrow \quad x^2 = a \quad \rightarrow \quad x^2 - a = 0$$

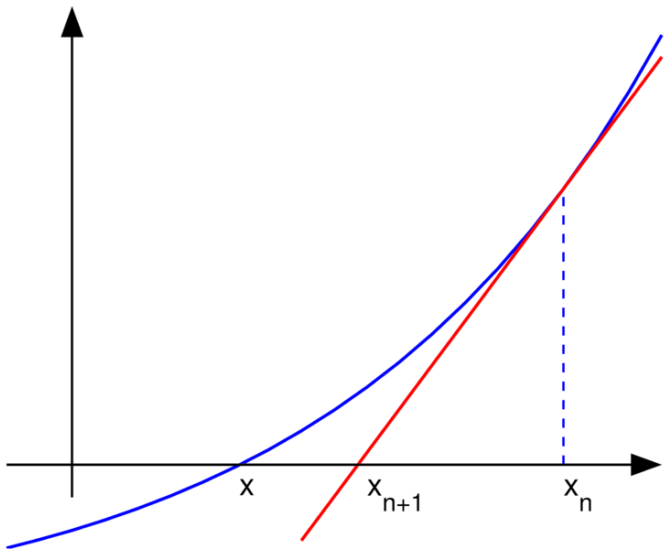
- Por tanto, querremos hallar un cero de la función $y = x^2 - a$
- Como x será generalmente un número real, nos podemos conformar con una aproximación
- Hay varios métodos para resolver este nuevo problema, por ejemplo:
  - Método de Newton-Raphson (que veremos a continuación)
  - Método de bipartición (de "decrementa y vencerás")

Método de Newton-Raphson

- Se trata de un método iterativo, que parte de un valor inicial x_0, y aplica la siguiente regla para hallar un cero de la función $f(x)$:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

- En cada iteración se busca la recta tangente a $f(x)$ que pasa por el punto $(x_n, f(x_n))$
- El corte de la recta con el eje de abscisas constituye el nuevo punto x_{n+1}



Método de Newton-Raphson

- El método surge del desarrollo de $f(x)$ en serie de Taylor, para un entorno del punto x_n:

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + (x - x_n)^2 \frac{f''(x_n)}{2!} + \ldots$$

- Si se trunca el desarrollo a partir del término de grado 2, se obtiene la recta tangente (que pasa por el punto $(x_n, f(x_n))$)
- Posteriormente evaluamos en x_{n+1}:

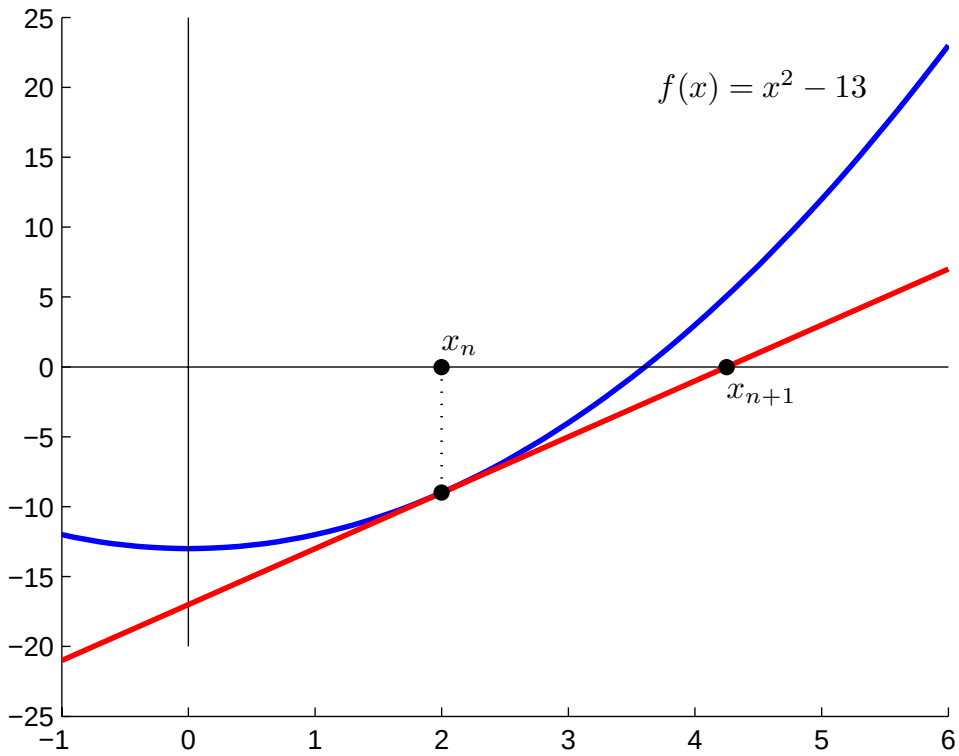
$$f(x_{n+1}) = f(x_n) + f'(x_n)(x_{n+1} - x_n)$$

- Si asumimos que x_{n+1} tiende hacia el cero de la función, podemos sustituir $f(x_{n+1}) = 0$, obteniéndose el algoritmo

Raíz cuadrada - método de Newton-Raphson

- Debemos aplicar la regla para $f(x) = x^2 - a$, con $f'(x) = 2x$

$$x_{n+1} = x_n - \frac{x_n^2 - a}{2x_n} = \frac{2x_n^2}{2x_n} - \frac{x_n^2 - a}{2x_n} = \frac{x_n^2 + a}{2x_n}$$



Factorización LU

- Supongamos que queremos resolver un sistema de ecuaciones $Ax = b$ repetidas veces, para muchos valores diferentes de b, pero para la misma matriz A
- Para acelerar el cálculo se puede realizar una factorización LU
- Sea A una matriz cuadrada en invertible, buscamos una factorización $A = LU$, donde L es triangular inferior, y U triangular superior (del mismo tamaño). Para una matrix de 3×3 tendríamos:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

- Para resolver el sistema $Ax = b$ ahora se van a resolver dos, pero más sencillos:

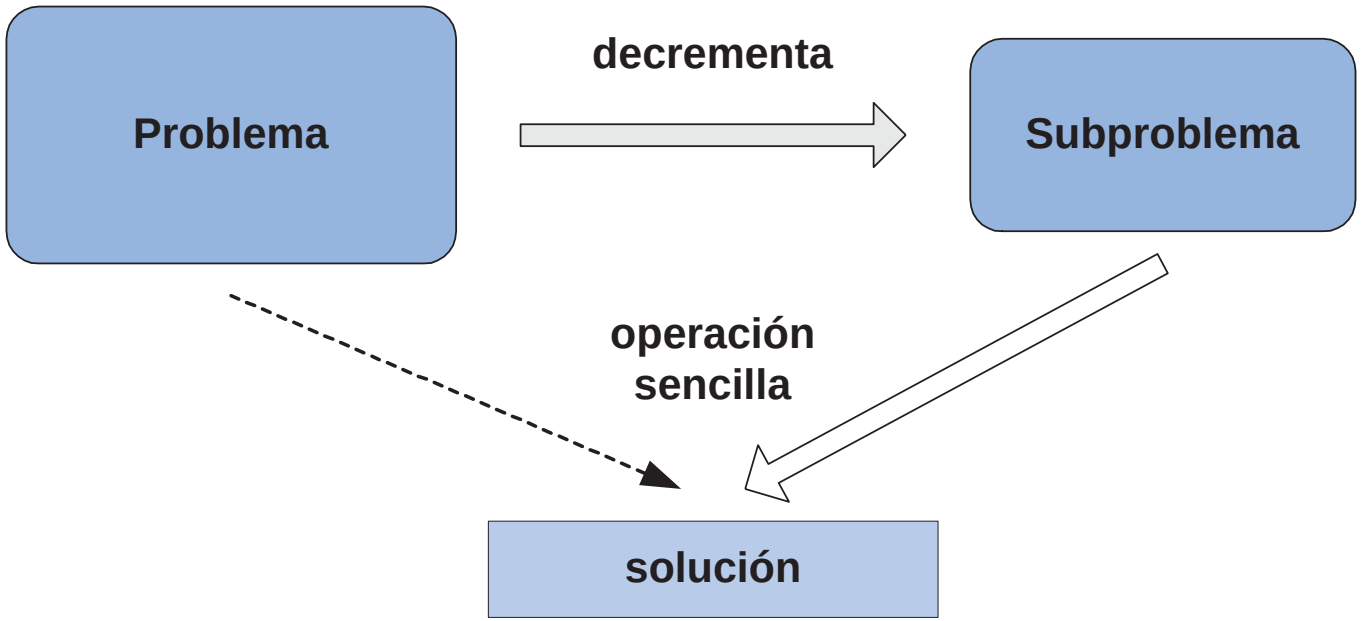
$$Ax = LUx = b \quad \Rightarrow \quad Ly = b, \quad Ux = y$$

- Una descomposición es útil cuando hay que hacer repetidas operaciones con la matriz A

Decrementa y vencerás

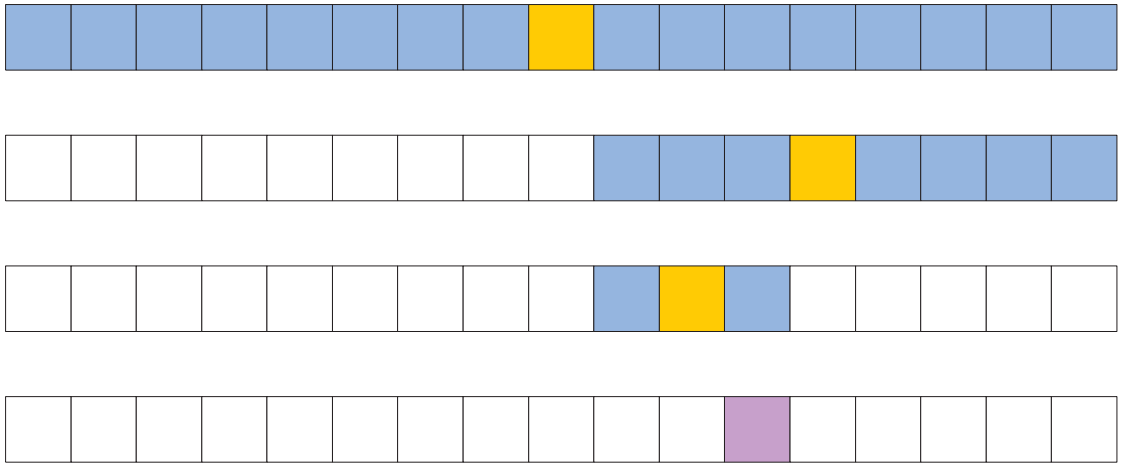
Decrementa y vencerás

- La descomposición de un problema genera un solo subproblema
- La solución al subproblema puede ser la solución al problema original
- Para otros problemas se requiere realizar alguna operación adicional con la solución al subproblema



Búsqueda binaria en un array ordenado

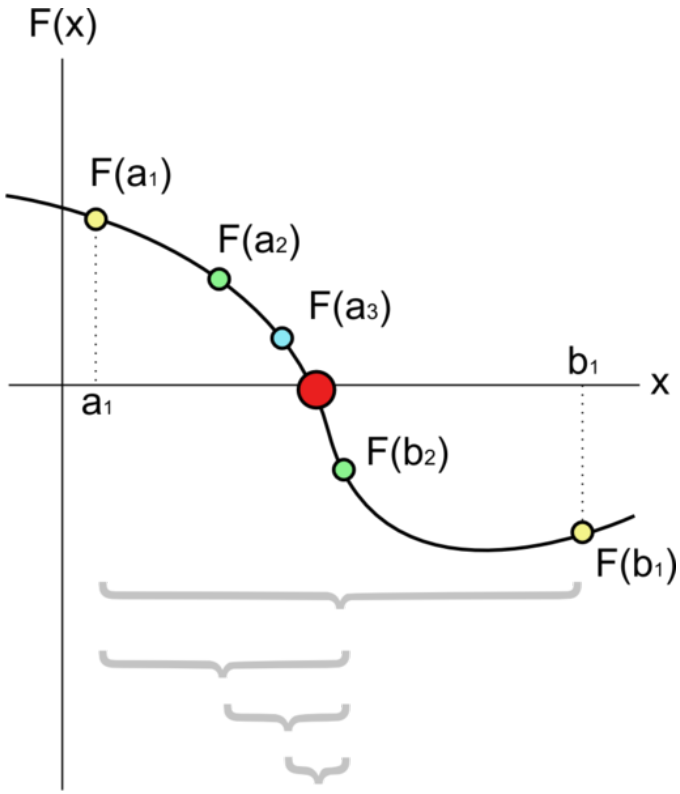
- Se compara el elemento central con el elemento a buscar
- Si no se encuentra se busca en alguna de las dos mitades



$$T(n) = \begin{cases} a & \text{si } n = 1 \\ b + T(n/2) & \text{si } n \geq 2 \end{cases} \quad \in \quad \Theta(\log n)$$

Método de bipartición

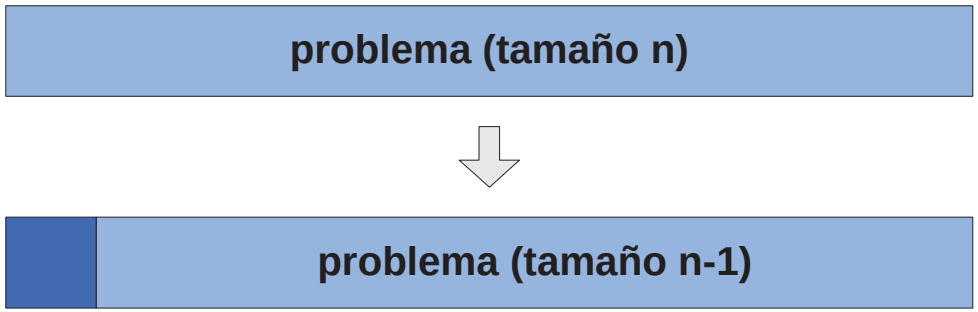
- Método para hallar un cero de una función f continua en un intervalo $[a, b]$
- Los signos de $f(a)$ y $f(b)$ son distintos



- Se busca un cero en la mitad del intervalo $c = (a + b)/2$
- Si no se encuentra se busca en $[a, c]$ o $[c, b]$
- Se repite el proceso hasta una determinada precisión ε (hasta que el ancho del intervalo sea menor que 2ε)

Ordenación por selección

1. Encontrar el menor elemento de una lista
2. Intercambiarlo con el primero de la lista
3. Aplicar el algoritmo de nuevo al resto de la lista (toda la lista salvo el primer elemento)



- Observad que se ha omitido a propósito mostrar más niveles para ilustrar 2, 3, 4... elementos ordenados
- Sólo se muestra la descomposición necesaria para realizar el diseño

Algoritmo de Euclides

- La reducción del tamaño del problema no siempre es tan obvia
- El algoritmo de Euclides encuentra el máximo común divisor de dos números naturales
- Hay dos versiones

$$mcd1(m,n) = \begin{cases} n & \text{si } m = 0 \\ mcd1(n,m) & \text{si } m > n \\ mcd1(m,n-m) & \text{si } (m \leq n) \text{ y } (m \neq 0) \end{cases}$$

$$mcd2(m,n) = \begin{cases} n & \text{si } m = 0 \\ mcd2(n\,\%\,m,m) & \text{si } m \neq 0 \end{cases}$$

- En cada paso nos vamos acercando al caso base

Algoritmo de Euclides - Demostración $mcd1$

$$mcd1(m,n) = \begin{cases} n & \text{si } m = 0 \\ mcd1(n,m) & \text{si } m > n \\ mcd1(m,n-m) & \text{si } (m \leq n) \text{ y } (m \neq 0) \end{cases}$$

- Supongamos que $m \leq n$ (en caso contrario el propio algoritmo intercambia los parámetros)
- $m = az$, $n = bz$, con $a \leq b$
  - z son los factores primos (máximo común múltiplo)
  - a y b no comparten factores primos
- Hacemos $b = a + c$
- $n = bz = (a + c)z = (a_1 \cdots a_k + c_1 \cdots c_l)z$
  - No se puede sacar factor común. Si se pudiera sería otro factor de z.
  - a, c no comparten primos
- Por tanto, dado que $z = mcd1(az, bz) = mcd1(az, cz)$, y $c = b - a$ tenemos:

$$mcd1(m,n) = mcd1(az,bz) = mcd1(az,cz) = mcd1(m,n-m)$$

Algoritmo de Euclides - Demostración *mcd2*

$$mcd2(m,n) = \begin{cases} n & \text{si } m = 0 \\ mcd2(n\,\%m, m) & \text{si } m \neq 0 \end{cases}$$

- Supongamos que $m \leq n$ (en caso contrario el propio algoritmo intercambia los parámetros)
- $m = az$, $n = bz$, con $a \leq b$
  - z son los factores primos
  - a y b no comparten factores primos
- Hacemos $b = ac + d$, donde c y d son el cociente y el resto de b/a
- a y d no pueden compartir factores, de lo contrario serían también factores de b
- Por tanto:

$$z = mcd2(az, bz) = mcd2(dz, az) \Rightarrow mcd2(m,n) = mcd2(n\,\%m, m)$$

Potencia en tiempo $\Theta(\log n)$

$$5^{13} = 5^{1101_2} = 1 \cdot 5^8 + 1 \cdot 5^4 + 0 \cdot 5^2 + 1 \cdot 5^1$$

- Algoritmo iterativo:

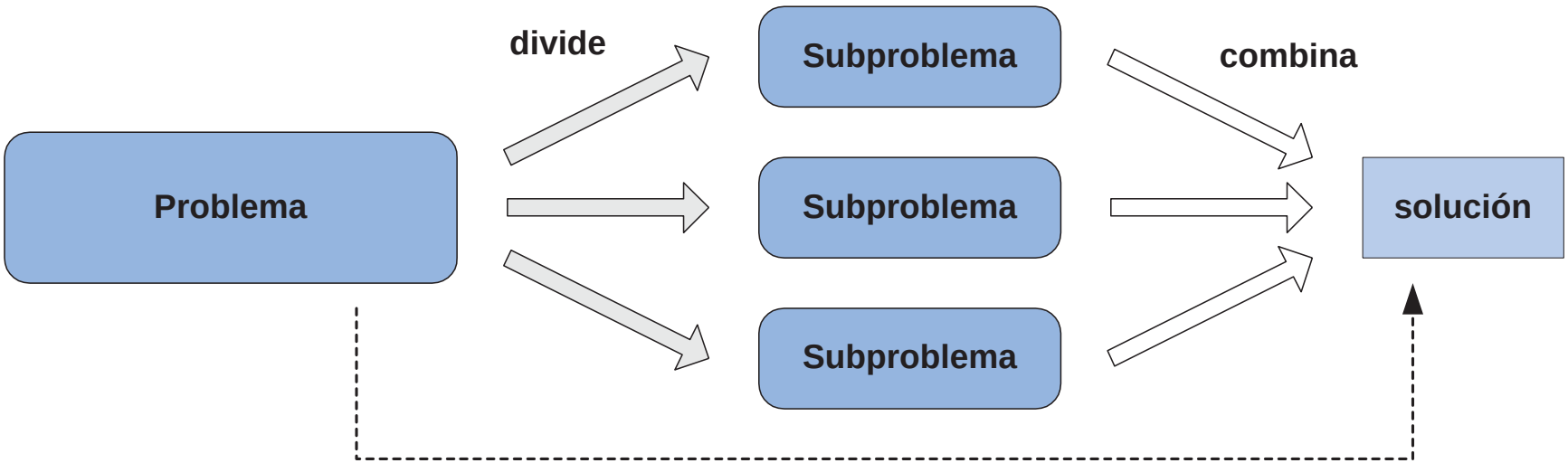
aux	e	e%2	suma
5^1	13	1	5^1
5^2	6	0	$0 + 5^1$
5^4	3	1	$5^4 + 0 + 5^1$
5^8	1	1	$5^8 + 5^4 + 0 + 5^1$

- Algoritmo recursivo (se calculan potencias de la mitad de tamaño, y luego se combina su resultado):
  - $b^0 = 1$
  - $b^e = b^{e/2} \cdot b^{e/2}$ para e par
  - $b^e = b \cdot b^{(e-1)/2} \cdot b^{(e-1)/2}$ para e impar
- Sin embargo, como los subproblemas son idénticos, sólo se calculan una vez, obteniéndose un algoritmo eficiente

Divide y vencerás

Divide y vencerás

- La descomposición de un problema genera varios subproblemas
- Las soluciones a los subproblemas se combinan para generar la solución al problema original, la cual puede requerir alguna operación adicional



Máximo de un vector de números

- Caso base
  - Si el vector tiene tamaño 1 se devuelve el elemento
- Descomposición
  - $m_1 =$ máximo de la primera mitad del vector
  - $m_2 =$ máximo de la segunda mitad del vector
- Combinación
  - máximo $= \text{máx}\{m_1, m_2\}$

$$max(v) = \begin{cases} v[1] & \text{si } n = 1 \\ \text{máx}(max(v[1 : n/2]), max(v[n/2 + 1 : n])) & \text{si } n > 1 \end{cases}$$

$n = v.length$

Máximo de un vector de números

- ¿Conseguimos un algoritmo más eficiente que una búsqueda lineal?
- No. Sea cual sea nuestra estrategia tenemos que mirar todos los elementos del vector
  - Este problema tiene una cota inferior $\Omega(n)$

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

- A pesar de dividir el problema por dos, pero tenemos que resolver dos problemas
- El árbol de recursión tiene del orden de 2^h nodos, donde h es la altura del árbol
- Pero en este problema $h = \log_2 n$, y el número de nodos es n
- $T(n) \in \Theta(n)$

Máximo de un vector de números

```
1   public static int max2 (int[] v) {
2      return maxDyV (v, 0, v.length-1);
3   }
4   private static int max (int m, int n) {
5      return (m>n?m:n);
6   }
7   public static int maxDyV (int[] v, int inf, int sup) {
8      if (inf==sup)
9         return v[inf];
10     else {
11        int medio = (inf+sup)/2;
12        return max (maxDyV (v, inf, medio),
13                    maxDyV (v, medio+1, sup));
14     }
15  }
```

Merge-sort

- Sea un vector de tamaño n
- Algoritmo de ordenación en tiempo $\Theta(n \log n)$
  - En todos los casos: mejor, peor, y medio
- Memoria auxiliar
  - $\Theta(n)$ en todos los casos: mejor, peor, y medio
  - Se denominan: "out-of-place": se requiere un vector auxiliar de tamaño n. Es decir, la ordenación no se realiza en el propio vector

Merge-sort

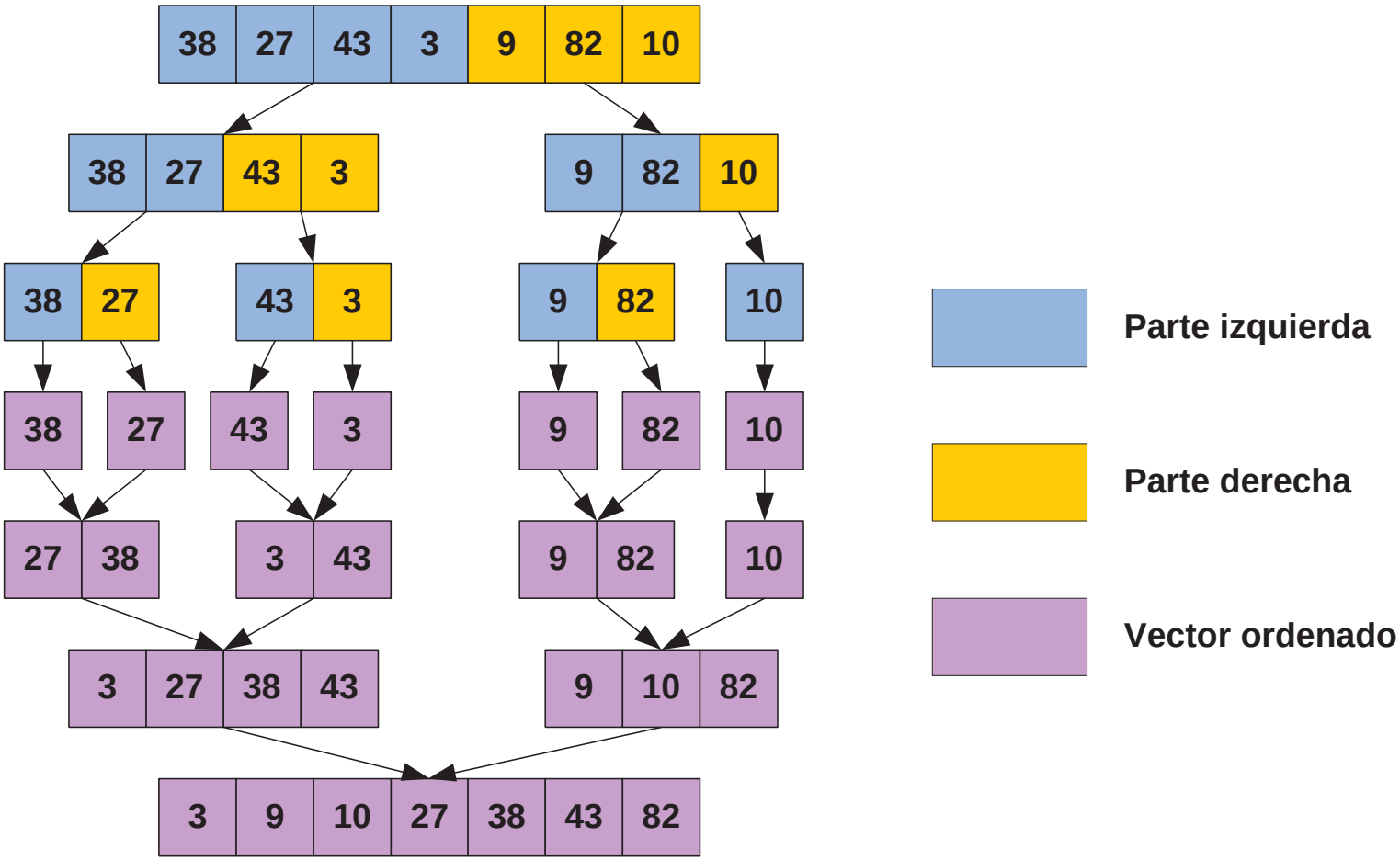
- Caso base
  - Si el vector tiene tamaño 1 se devuelve el elemento
- Descomposición
  - Se divide el vector en dos, generalmente por la mitad, y se ordenan las dos partes por separado
- Combinación
  - Se mezclan las dos mitades ordenadas

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

- ¿Cuál es el tiempo si no se divide el vector por la mitad, sino según un 10 % y 90 %?
  - También $\Theta(n \log n)$

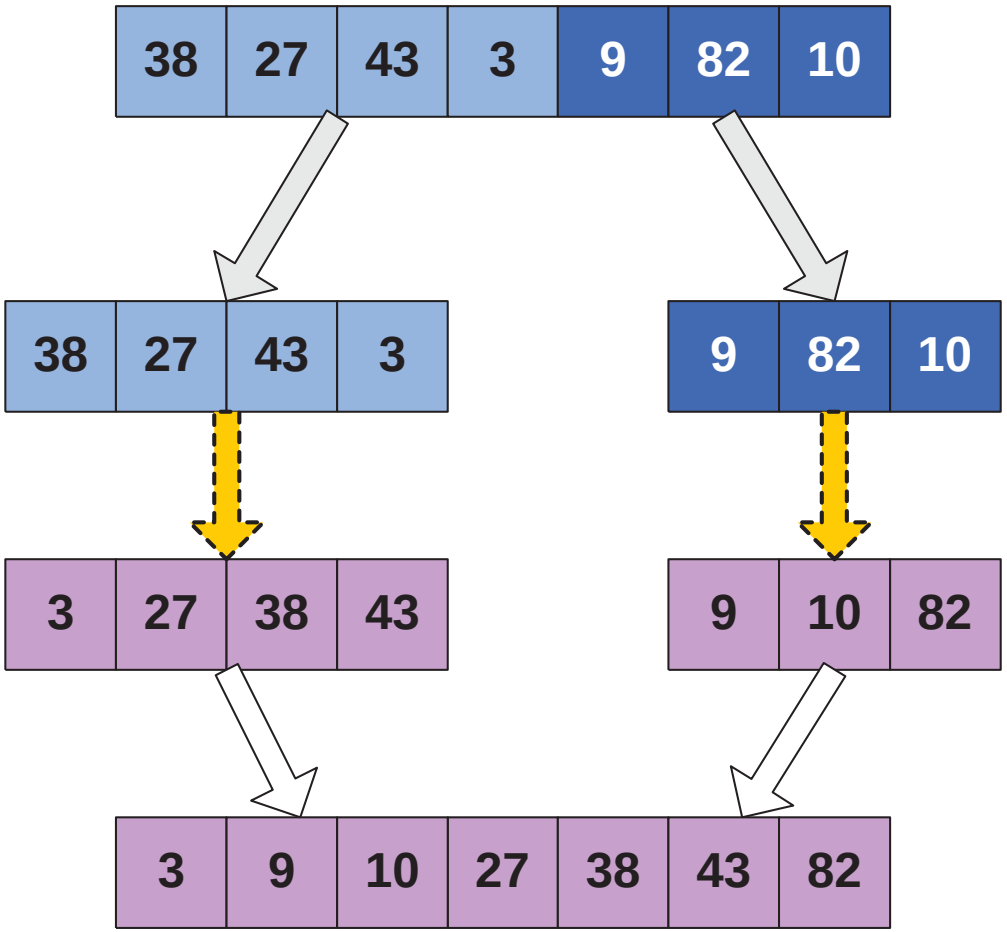
Merge-sort

- Paso a paso



Merge-sort

- Para diseñar el código, es mejor pensar en este esquema



Descomponer el problema

Ordenar subproblemas

Combiar soluciones

Implementación del Merge-sort I

```
1  void mergeSort1 (int[] v) {
2     ordenarPorMezcla1 (v, 0, v.length-1);
3  }
4
5  void ordenarPorMezcla1 (int[] v, int inf, int sup) {
6     if (inf<sup) {
7        int medio = (inf+sup)/2;
8        ordenarPorMezcla1 (v, inf, medio);
9        ordenarPorMezcla1 (v, medio+1, sup);
10       mezclar1 (v, inf, medio, sup);
11    }
12 }
```

Implementación del Merge-sort I

- Mezcla ineficiente (puede ocasionar errores en tiempo de ejecución)

```
1 void mezclar1 (int[] v, int inf, int medio, int sup) {
2    //vector auxiliar de longitud n (lento)
3    int[] vAux = new int[v.length];
4 ...
```

- Mezcla eficiente

```
1 void mezclar1_mejor (int[] v, int inf, int medio, int sup) {
2    //vector auxiliar de longitud exacta
3    int[] vAux = new int[sup-inf+1];
4 ...
```

Implementación alternativa del Merge-sort II

- El parámetro auxiliar debe pasarse por referencia

```
1  void mergeSort2 (int[] v) {
2     //vector auxiliar de longitud n, se propaga
3     int[] vAux = new int[v.length];
4
5     ordenarPorMezcla2 (v, 0, v.length-1, vAux);
6  }
7
8  void ordenarPorMezcla2 (int[] v, int inf, int sup, int[] vAux) {
9     if (inf<sup) {
10       int medio = (inf+sup)/2;
11       ordenarPorMezcla2 (v, inf, medio, vAux);
12       ordenarPorMezcla2 (v, medio+1, sup, vAux);
13       mezclar2 (v, inf, medio, sup, vAux);
14    }
15 }
```

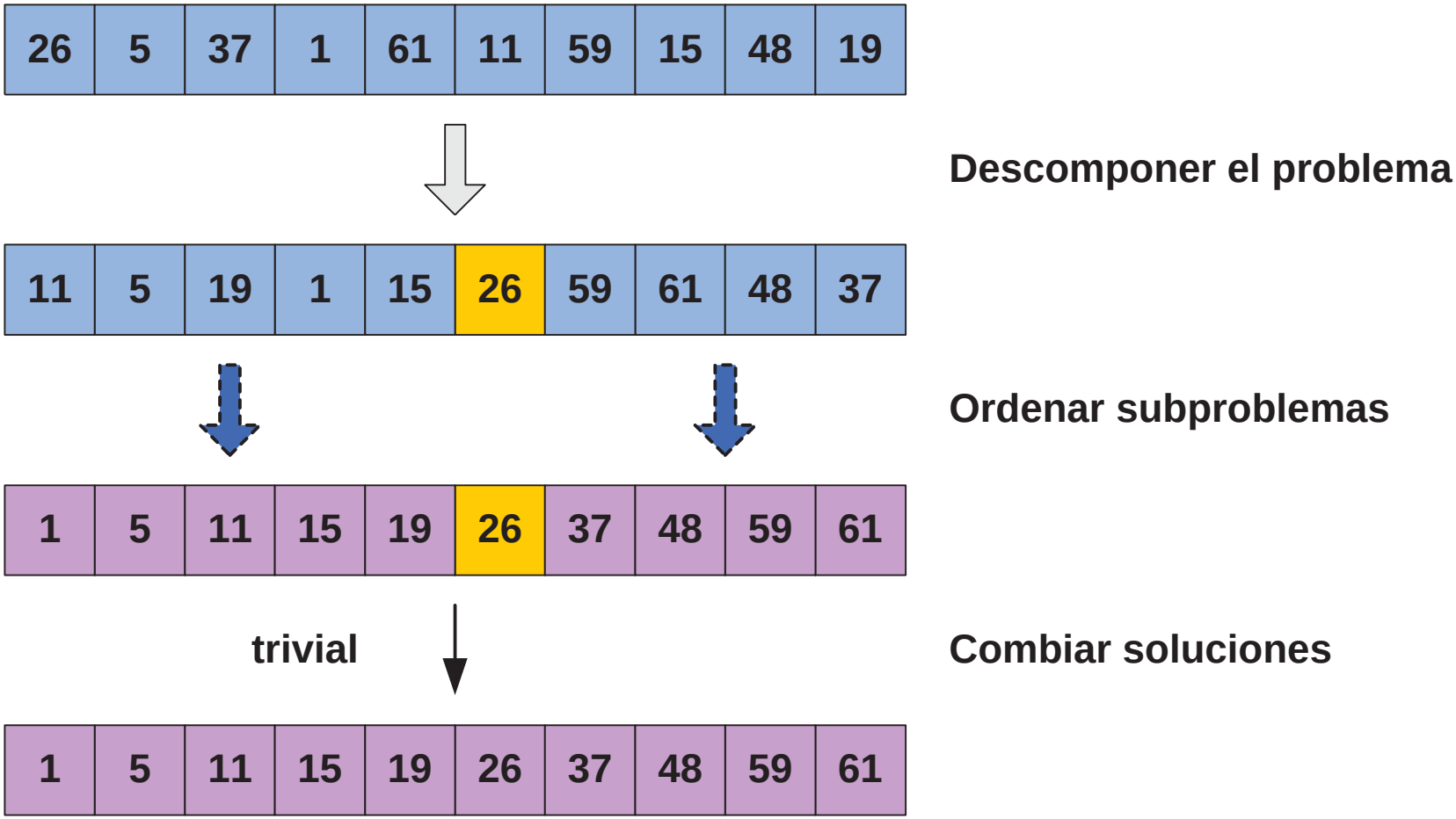
```
1  void mezclar2 (int[] v, int inf, int medio, int sup, int [] vAux) {
2  ...
```

Quick-sort

- Caso base
  - Si el vector tiene tamaño 1 se devuelve el elemento
- Descomposición
  - Se escoge un elemento "pivote"
  - Se divide el vector en tres partes:
    - Parte izquierda: elementos menores o iguales que el pivote
    - El propio pivote
    - Parte derecha: elementos mayores que el pivote
  - Posteriormente se ordenan las partes izquierda y derecha
- Combinación
  - Trivial: el vector ya está ordenado, no hay que hacer nada más

Quick-sort

- Esquema:



Quick-sort

- Caso mejor (el elemento pivote está en la mitad)

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases} \quad \Rightarrow \quad \Theta(n \log n)$$

- Caso peor (un subvector queda siempre vacío)

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

Quick-sort

- Algoritmo de ordenación rápido en la práctica
  - Caso peor: $\Theta(n^2)$
  - Caso mejor: $\Theta(n \log n)$
  - Caso medio: $\Theta(n \log n)$
- Suele superar a otros algoritmos $\Theta(n \log n)$ al utilizar mejor las jerarquías de memoria
- Memoria auxiliar
  - $\mathcal{O}(n)$
  - Puede ser "out-of-place" o "in-place" (la ordenación no se realiza en el propio vector)

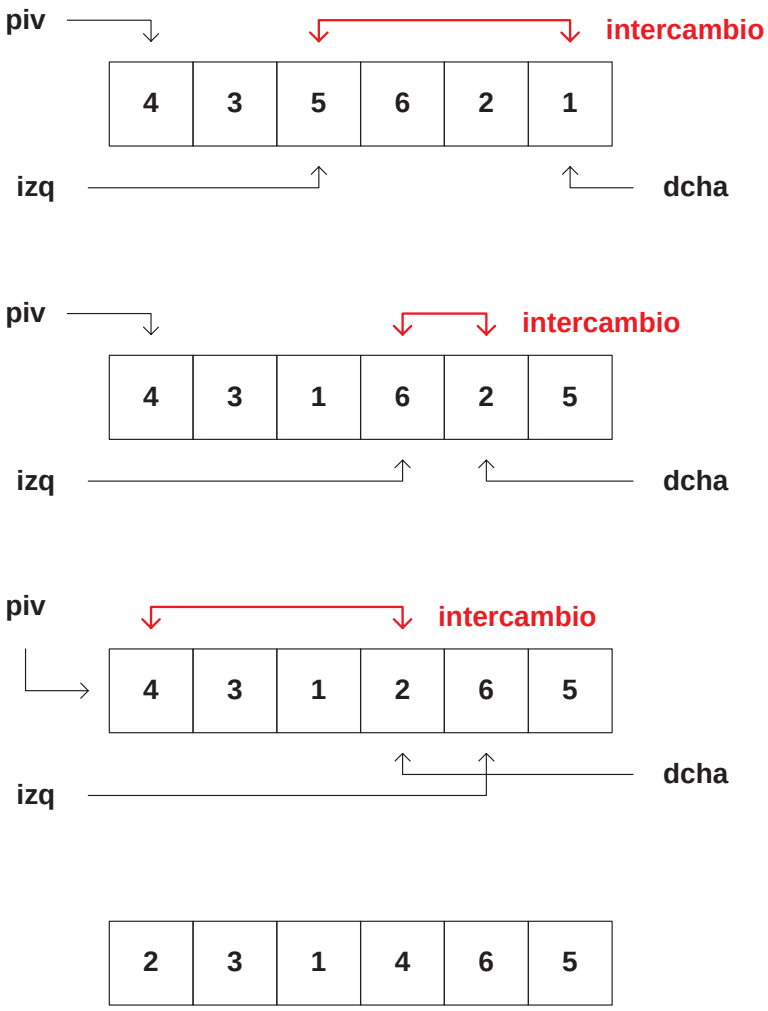
Implementación del Quick-sort

```
1  void quickSort (int[] v) {
2     ordenarRapido (v, 0, v.length-1);
3  }
4
5  void ordenarRapido (int[] v, int inf, int sup) {
6     if (inf<sup) {
7        int med = partir (v, inf, sup);
8        if (inf<med) ordenarRapido (v, inf, med-1);
9        if (med<sup) ordenarRapido (v, med+1, sup);
10    }
11 }
```

Implementación del Quick-sort

```
 1  int partir (int[] v, int inf, int sup) {
 2     int piv  = v[inf];
 3     int izq  = inf+1;
 4     int dcha = sup;
 5     int temp;
 6     do {
 7        for (; v[izq]<=piv  && izq<sup   ; izq++);
 8        for (; v[dcha]>piv /*&&dcha>inf*/; dcha--);
 9        if (izq<dcha) {
10           temp = v[izq];
11           v[izq] = v[dcha];
12           v[dcha] = temp;
13        }
14     } while (izq<dcha);
15     /* v[inf] <-> v[dcha] */
16     temp = v[inf];
17     v[inf] = v[dcha];
18     v[dcha] = temp;
19     return dcha;
20  }
```

Implementación del Quick-sort, método `partir`



- Hay otras implementaciones que no escogen el primer elemento como pivote

Traspuesta de una matriz

- Sea una matriz A de dimensiones $n \times n$, donde n es una potencia de 2
- Se puede definir su traspuesta A^T dividiendo la matriz en bloques:

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \Rightarrow A^T = \left(\begin{array}{c|c} A_{11}^T & A_{21}^T \\ \hline A_{12}^T & A_{22}^T \end{array}\right)$$

Implementación de la traspuesta de una matriz

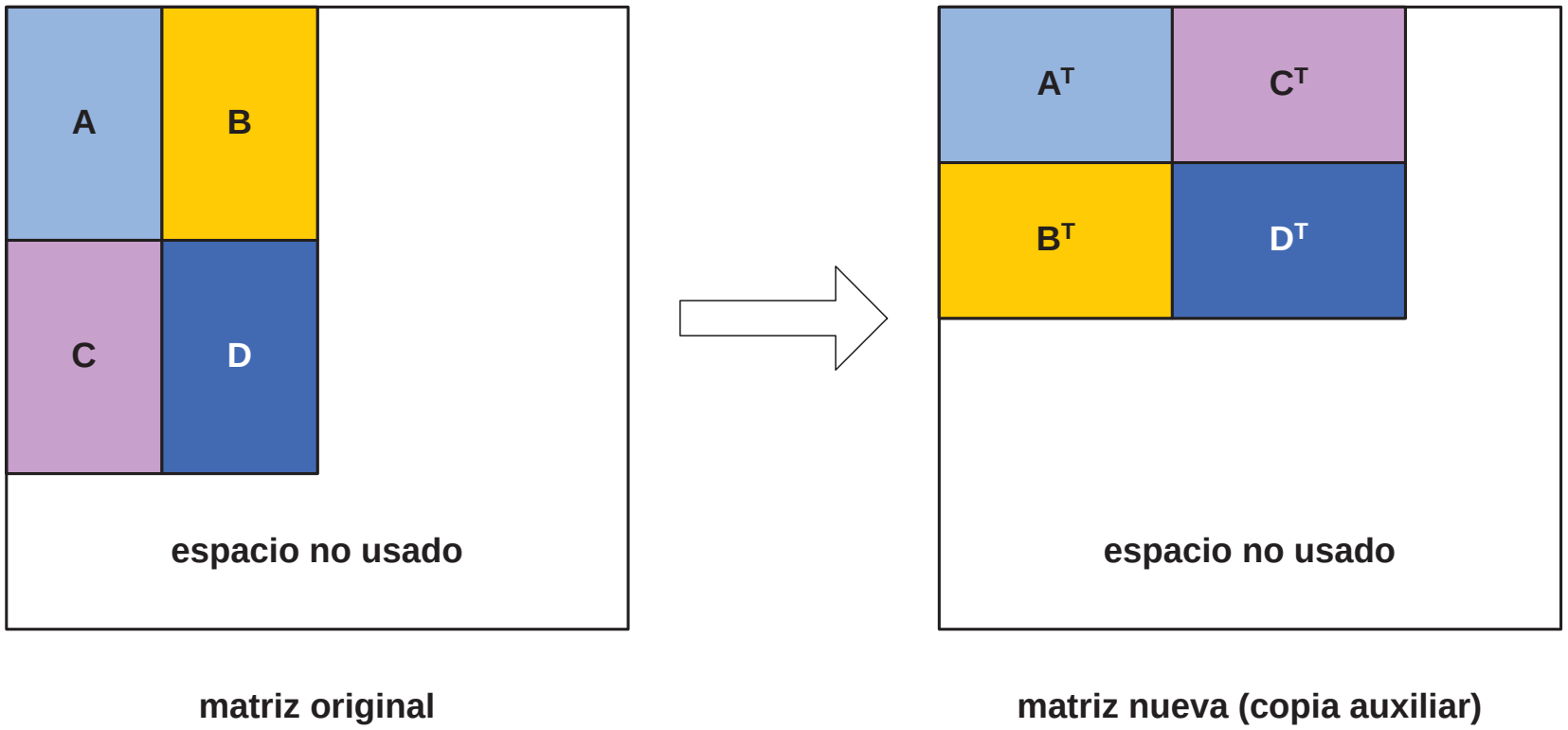
```
1  void trasponer (int[][] m) {
2     trasponerDyV1 (m, 0, m.length-1, 0, m.length-1);
3  }
4
5  void trasponerDyV1 (int[][] m,
6                      int fInicio, int fFin, int cInicio, int cFin) {
7     // caso básico de 1x1
8     if (fInicio<fFin) {
9     int fMedio = (fInicio+fFin)/2;
10    int cMedio = (cInicio+cFin)/2;
11    trasponerDyV1 (m, fInicio, fMedio, cInicio, cMedio);
12    trasponerDyV1 (m, fInicio, fMedio, cMedio+1, cFin);
13    trasponerDyV1 (m, fMedio+1, fFin, cInicio, cMedio);
14    trasponerDyV1 (m, fMedio+1, fFin, cMedio+1, cFin);
15    intercambiar (m, fMedio+1, cInicio, fInicio, cMedio+1, fFin-fMedio);
16    }
17 }
```

Implementación de la traspuesta de una matriz

```
1 void intercambiar (int[][] m,int fIniA, int cIniA,
2                    int fIniB, int cIniB, int dimen) {
3    for (int i=0; i<=dimen-1; i++)
4       for (int j=0; j<=dimen-1; j++) {
5          int aux = m[fIniA+i][cIniA+j];
6          m[fIniA+i][cIniA+j] = m[fIniB+i][cIniB+j];
7          m[fIniB+i][cIniB+j] = aux;
8       }
9 }
```

Traspuesta de una matriz no cuadrada

- También puede implementarse cuando n no es una potencia de dos, o incluso cuando la matriz no es cuadrada
- Se pueden usar matrices de tamaño fijo:



- Por supuesto, también se puede usar memoria dinámica

Multiplicación de matrices

- La división de una matriz en bloques tiene varias ventajas:
  - Simplifica operaciones algebraicas
  - Se puede aprovechar para construir algoritmos eficientes que hagan buen uso de la memoria caché
- Para realizar una multiplicación hay varias formas de dividir la matriz

$$A \cdot B = \left(\begin{array}{c} A_1 \\ \hline A_2 \end{array}\right) \cdot \left(\begin{array}{c|c} B_1 & B_2 \end{array}\right) = \begin{pmatrix} A_1 B_1 & A_1 B_2 \\ A_2 B_1 & A_2 B_2 \end{pmatrix}$$

$$A \cdot B = \left(\begin{array}{c|c} A_1 & A_2 \end{array}\right) \cdot \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right) = \begin{pmatrix} A_1 B_1 + A_2 B_2 \end{pmatrix}$$

$$A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11} B_{11} + A_{12} B_{21} & A_{11} B_{12} + A_{12} B_{22} \\ A_{21} B_{11} + A_{22} B_{21} & A_{21} B_{12} + A_{22} B_{22} \end{pmatrix}$$

Multiplicación de matrices - complejidad

- Sea A una matriz de $p \times q$, y B una matriz de $q \times r$
- El producto $C = AB$ requiere $\mathcal{O}(pqr)$ operaciones
- Considérese que las matrices son cuadradas de dimensión $n \times n$, y la siguiente descomposición:

$$A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 8\,T(n/2) + 4\Theta(n^2) & \text{si } n > 1 \end{cases}$$

  - 8 multiplicaciones (de matrices de $n/2 \times n/2$)
  - 4 sumas (de matrices de $n/2 \times n/2$)
  - Por el teorema maestro: $T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$

Multiplicación de matrices - algoritmo de Strassen

- La operación básica más costosa es la multiplicación
- El algoritmo de Strassen consigue reducir el número de multiplicaciones
- Considérese la siguiente descomposición:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

Multiplicación de matrices - algoritmo de Strassen

- Con esa factorización el algoritmo requiere
  - 7 multiplicaciones
  - 18 sumas

$$T(n) = \begin{cases} 1 & \text{si } n = 1 \\ 7T(n/2) + 18\Theta(n^2) & \text{si } n > 1 \end{cases}$$

  - Por el teorema maestro: $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2,807})$

- Solo es más eficiente que una multiplicación ordinaria si el tamaño de las matrices es muy elevado
- Actualmente hay un algoritmos en $\mathcal{O}(n^{2,376})$ (Coppersmith - Winograd)
- En la práctica la mejor estrategia para acelerar el producto de dos matrices consiste en aprovechar la estructura jerárquica de la memoria