

# Esquemas Algorítmicos

- Conocer un conjunto de técnicas de resolución de familias de problemas.
- Dado un problema concreto: caracterizar convenientemente el problema, valorar y elegir la técnica más adecuada.

# Esquemas Algorítmicos

## Contenidos

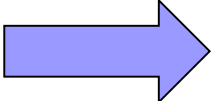
1. **Introducción**
2. **Algoritmos voraces.**
3. **Algoritmos de vuelta atrás.**
4. **Divide y vencerás.**

# Esquemas Algorítmicos

- **Algoritmia** = tratamiento sistemático de técnicas fundamentales para el diseño y análisis de algoritmos eficientes.



# Esquemas Algorítmicos

- Computadores cada vez más rápidos y a más bajo precio:
  - Se resuelven problemas de cálculo antes impensables.
  - Inconscientemente se tiende a restar importancia al concepto de **eficiencia**.
- Existen problemas que seguirán siendo intratables si se aplican ciertos algoritmos por mucho que se aceleren los computadores.   
Importancia de nuevas soluciones eficientes.

# Esquemas Algorítmicos

- Un curso de algoritmia (o “esquemas algorítmicos”)

## **NO ES**

- ni un curso de programación (ya debéis saber programar)
- Ni un curso de estructuras de datos (ya debéis conocer las fundamentales)

## **TAMPOCO ES**

- una colección de “recetas” o algoritmos listos para ser introducidos en el ordenador para resolver problemas específicos.

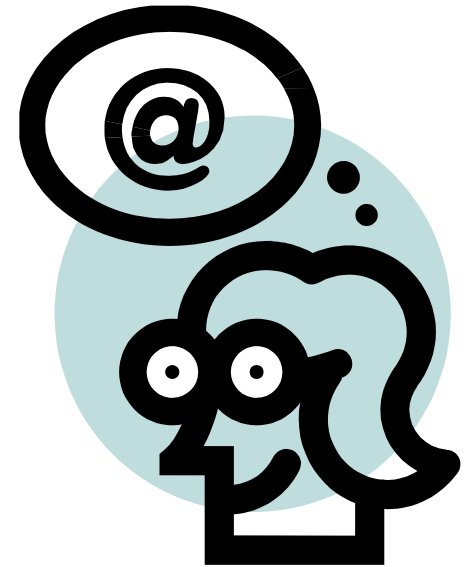
# Esquemas Algorítmicos

- Un curso de algoritmia tiene como objetivos principal:
    - Dar más herramientas fundamentales para facilitar el desarrollo de programas.
    - ¿Qué herramientas?
- Técnicas o “esquemas” de diseño de algoritmos eficientes.



# Esquemas Algorítmicos

- Un medio para alcanzar ese objetivo es:
  - Presentar cada esquema de forma genérica, incidiendo en los principios que conducen a él, e
  - ilustrar el esquema mediante ejemplos concretos de algoritmos tomados de varios dominios de aplicación



# Esquemas Algorítmicos

- Un ejemplo muy sencillo: Multiplicación de dos enteros positivos con lápiz y papel.

En España:

$$\begin{array}{r} 981 \\ \times 1234 \\ \hline 3924 \\ 2943 \\ 1962 \\ 981 \\ \hline 1210554 \end{array}$$



En Inglaterra:

$$\begin{array}{r} 981 \\ \times 1234 \\ \hline 981 \\ 1962 \\ 2943 \\ 3924 \\ \hline 1210554 \end{array}$$

- Ambos métodos son muy similares, los llamaremos algoritmo “clásico” de multiplicación.



# Esquemas Algorítmicos

- Algoritmo de multiplicación “a la rusa”:

☑ 981	1234	1234
490	2468	
☑ 245	4936	4936
122	9872	
☑ 61	19744	19744
30	39488	
☑ 15	78976	78976
☑ 7	157952	157952
☑ 3	315904	315904
☑ 1	631808	<u>631808</u>
		1210554

- Ventaja: no hay que almacenar los productos parciales.
- Sólo hay que saber sumar, multiplicar y dividir por 2.

# Esquemas Algorítmicos

## ■ Otro algoritmo:

- De momento, exigimos que ambos números tengan igual  $n^0$  de cifras y que éste sea potencia de 2.
- Por ejemplo: 0981 y 1234.
- En primer lugar, partimos ambos números por la mitad y hacemos cuatro productos:

	multiplicar		desplazar	resultado
1.	09	12	4	108....
2.	09	34	2	306..
3.	81	12	2	972..
4.	81	34	0	2754
				1210554

Doble del número de cifras      número de cifras

- Es decir, hemos reducido un producto de números de 4 cifras en cuatro productos de números de 2 cifras, varios desplazamientos y una suma.

# Esquemas Algorítmicos

- Los productos de números de 2 cifras pueden hacerse con la misma técnica.
- Por ejemplo 09 y 12

	multiplicar		desplazar	resultado
1.	0	1	2	0..
2.	0	2	1	0.
3.	9	1	1	9.
4.	9	2	0	18
				<hr/>
				108

- Es un ejemplo de algoritmo que utiliza la técnica de “divide y vencerás”.
- Tal y como se ha presentado....
  - NO mejora en eficiencia al algoritmo clásico.
- Pero puede mejorarse: Es posible reducir un producto de dos números de muchas cifras a 3 (en vez de 4) productos de números de la mitad de cifras, y éste SI que mejora al algoritmo clásico.

# Esquemas Algorítmicos

## ■ Ideas clave:

- Incluso para un problema tan básico pueden construirse **MUCHAS** soluciones.
  - El método clásico lo usamos con lápiz y papel porque nos resulta muy **familiar** (lo que se aprende en la infancia...).
  - El método “a la rusa” se implementa en hardware en los computadores por la naturaleza **elemental** de los cálculos intermedios.
  - El método de divide y vencerás es más **rápido** si se quiere multiplicar números grandes.
- 
- La **algoritmia** estudia las propiedades de los algoritmos y nos ayuda a **elegir** la solución más adecuada en cada situación.
  - En muchos casos, una buena elección **ahorra tiempo y dinero**.
  - En algunos casos, una buena elección marca la diferencia entre **poder** resolver un problema y **no poder** hacerlo.

# El esquema voraz: Introducción

- Es uno de los esquemas más simples y al mismo tiempo de los más utilizados.
- Típicamente se emplea para resolver **problemas de optimización**:
  - Existe una entrada de tamaño  $n$  que son los **candidatos** a formar parte de la solución.
  - Existe un subconjunto de esos  $n$  candidatos que satisface ciertas restricciones: se llama **solución factible**.
  - Hay que obtener la solución factible que maximice o minimice una cierta función objetivo: se llama **solución óptima**.

# Esquema voraz

El **esquema voraz** procede por pasos:

- inicialmente el conjunto de candidatos escogidos es vacío;
- en cada paso, se intenta añadir al conjunto de los escogidos “el mejor” de los no escogidos (sin pensar en el futuro), utilizando una **función de selección** basada en algún criterio de optimización (puede ser o no ser la función objetivo);
- tras cada paso, hay que ver si el conjunto seleccionado es **factible** (i.e., si añadiendo más candidatos se puede llegar a una solución);
  - si el conjunto no se puede completar, se rechaza el último candidato elegido **y no se vuelve a considerar en el futuro**;
  - si se completa, se incorpora al conjunto de escogidos y **permanece siempre en él, (función objetivo)**;
- tras cada incorporación se comprueba si el conjunto resultante es una solución (**función de solución**);
- el algoritmo termina cuando se obtiene una solución;
- el algoritmo es correcto, si se encuentra solución siempre es óptima;

# Esquema voraz

## ■ Esquema genérico:

Conjunto voraz( $C$ :conjunto)

{

$\{C$  es el conjunto de candidatos $\}$

$S \leftarrow \Phi$  {Construimos la solución en el conjunto  $S$ }

  mientras (  $C \neq \Phi$  y no solución(  $s$  ) )

$x \leftarrow$  seleccionar (  $C$  )

$C \leftarrow C \setminus \{x\}$

    Si factible (  $S \cup \{x\}$  )  $S \leftarrow S \cup \{x\}$

si solución( $S$ ) devolver  $S$

  sino devolver "no hay soluciones"

}

# Ejemplo de Algoritmo voraz: Número cromático

- El **algoritmo de número cromático** indica el número de colores mínimo necesario para colorear los vértices de un grafo. El *teorema de los cuatro colores* justifica que el número cromático de un grafo plano es menor o igual que cuatro.
- El objetivo de este ejercicio consiste en, dado un grafo plano que representa un mapa político, indicar cómo deben colorearse cada uno de los vértices de dicho grafo, empleando el menor número posible de colores (según el teorema citado anteriormente como máximo cuatro).



# Ejemplo de Algoritmo voraz: Número cromático

- Siguiendo el esquema voraz:
  1. Los candidatos son los vértices del grafo.
  2. La **función de solución** comprueba si ya se han visitado todos los vértices.
  3. Un vértice será **factible** si todavía no ha sido visitado.
  4. La **función de selección** toma un vértice cualquiera de los no visitados.
  5. La **función objetivo** elegir un color para el vértice seleccionado, de tal forma que no sea igual a ninguno del de sus adyacentes. Objetivo colorear todos los vértices del grafo utilizando el menor número posible de colores (como máximo serán 4) de tal forma que dos vértices adyacentes no tengan el mismo color.

# Ejemplo de Algoritmo voraz: Número cromático

**función** colorear(Grafo g, String [] c): colección de clave-valor  
(String)

```
{  
    porVisitar = V  
    mapaColores con todos los vértices sin color  
  
    mientras porVisitar  $\neq \emptyset$  hacer  
        v = vértice a colorear  
        color = seleccionar color para "v" de tal forma que sea  
        distinto del de sus adyacentes
```

mapaColores.añadir(v,color)

**fin** mientras

```
devolver mapaColores  
}
```

**Conjunto voraz**(C:conjunto)  
{

```
{C es el conjunto de candidatos}  
S  $\leftarrow \Phi$  {Construimos la solución en el  
conjunto S}  
mientras ( C  $\neq \Phi$  y no solución( s ) )  
    x  $\leftarrow$  seleccionar ( C )  
    C  $\leftarrow$  C  $\setminus$  {x}
```

Si factible ( S U {x} ) S  $\leftarrow$  S U {x}

```
si solución(S) devolver S  
sino devolver "no hay soluciones"  
}
```

# Ejemplo de Algoritmo voraz: Problema del viajante

- Consideremos un mapa de carreteras, con dos tipos de componentes: las ciudades (*nodos*) y las carreteras que las unen. Cada tramo de carreteras (*arco*) está señalado con su longitud. “Un viajante debe recorrer una serie de ciudades interconectadas entre sí, de manera que recorra todas ellas con el menor número de kilómetros posible”.



# Ejemplo de Algoritmo voraz: Algoritmo del viajante

- Siguiendo el esquema voraz:
  1. Los candidatos son los vértices del grafo.
  2. La **función de solución** comprueba si ya se han visitado todos los vértices.
  3. Un vértice será **factible** si todavía no ha sido visitado.
  4. La **función de selección** toma el arco cuya distancia desde el vértice actual es la menor de todos los vértices que quedan por visitar (del conjunto de candidatos).
  5. La **función objetivo** obtener la distancia mínima desde el vértice origen al resto de los vértices, quedando el conjunto de candidatos vacío.

# Ejemplo de Algoritmo voraz: Problema del viajante

**función** viajante (Grafo g, Vertice v): grafo unidireccional

```
{  
    porVisitar = V  
    gSolucion = grafo vacio  
    ciudadActual = v  
  
    mientras porVisitar  $\neq \emptyset$  hacer  
        u = Arco cuyo vertice Destino  $\in$  porVisitar y cuya  
        distancia a ciudad actual es la mínima  
  
        porVisitar.eliminar(u.getDestino);  
        gSolucion.añadirArco(u)  
        ciudadActual = u.getDestino;  
    fin mientras  
  
    devolver gSolucion  
}
```

**Conjunto voraz**(C:conjunto)

```
{  
    {C es el conjunto de candidatos}  
    S  $\leftarrow \Phi$  {Construimos la solución en el  
    conjunto S}  
  
    mientras ( C  $\neq \Phi$  y no solución( s ) )  
        x  $\leftarrow$  seleccionar ( C )  
  
        C  $\leftarrow$  C  $\setminus$  {x}  
        Si factible ( S  $\cup$  {x} ) S  $\leftarrow$  S  $\cup$  {x}  
  
    si solución(S) devolver S  
    sino devolver "no hay soluciones"  
}
```

# Ejemplo de Algoritmo voraz: Algoritmo de Prim

- Consideremos un mapa de carreteras, con dos tipos de componentes: las ciudades (*nodos*) y las carreteras que las unen. Cada tramo de carreteras (*arco*) está señalado con su longitud. Se desea implantar un tendido eléctrico siguiendo los trazos de las carreteras de manera que conecte todas las ciudades y que la longitud total sea mínima.

# Ejemplo de Algoritmo voraz: Algoritmo de Prim

- Siguiendo el esquema voraz:
  1. Los candidatos son los vértices del grafo.
  2. La **función de solución** comprueba si ya se han visitado todos los vértices.
  3. Un vértice será **factible** si todavía no ha sido visitado.
  4. La **función de selección** toma el arco cuya distancia entre uno de los vértices visitados y otro de los vértices que quedan por visitar (del conjunto de candidatos) es la menor.
  5. La **función objetivo** conectar todos los vértices entre sí con la menor distancia posible.

# Ejemplo de Algoritmo voraz: Algoritmo de Prim

**función** prim (Grafo g, Vertice v): grafo unidireccional

```
{  
  porVisitar = V  
  gSolucion = grafo vacio  
  visitados = v  
  porVisitar.eliminar(v)  
  
  mientras porVisitar  $\neq \emptyset$  hacer  
    u = Arco cuyo vértice destino  $\in$  porVisitar y  
    vértice origen  $\in$  visitados, cuya distancia es la  
    mínima  
    porVisitar.eliminar(u.getDestino)  
    visitados.añadir(u.getDestino)  
    gSolucion.añadirArco(u)  
  fin mientras  
  devolver gSolucion  
}
```

**Conjunto voraz**(C:conjunto)

```
{  
  {C es el conjunto de candidatos}  
  S  $\leftarrow \Phi$  {Construimos la solución en el  
  conjunto S}  
  
  mientras ( C  $\neq \Phi$  y no solución( s ) )  
    x  $\leftarrow$  seleccionar ( C )  
  
    C  $\leftarrow$  C  $\setminus$  {x}  
    Si factible ( S U {x} ) S  $\leftarrow$  S U {x}  
  
  si solución(S) devolver S  
  sino devolver "no hay soluciones"  
}
```



# Ejemplo de Algoritmo voraz: Algoritmo de Dijkstra

- **Algoritmo de Dijkstra**, también llamado **algoritmo de caminos mínimos**, es un algoritmo para la determinación del camino más corto dado un vértice origen al resto de vértices en un grafo dirigido y con pesos en cada arista.
- La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen, al resto de vértices que componen el grafo, el algoritmo se detiene.

# Ejemplo de Algoritmo voraz: Algoritmo de Dijkstra

- Siguiendo el esquema voraz:
  1. Los candidatos son los vértices del grafo.
  2. La **función de solución** comprueba si ya se han visitado todos los vértices.
  3. Un vértice será **factible** si todavía no ha sido visitado.
  4. La **función de selección** toma el vértice cuya distancia al vértice origen es la menor de todos los vértices que quedan por visitar (del conjunto de candidatos).
  5. La **función objetivo** modifica la distancia del vértice adyacente al visitado si la distancia del vértice visitado + la distancia del arco entre el vértice seleccionado y el adyacente, es menor que la distancia del vértice origen al vértice adyacente.  
Objetivo obtener la distancia mínima al vértice origen.

# Ejemplo de Algoritmo voraz: Algoritmo de Dijkstra

**función** dijkstra(Grafo g, Vertice v): colección de clave-valor  
(enteros)

```
{  
  porVisitar = V  
  distancia[v] = 0 y distancia[u] =  $\infty \forall u \neq v$   
  
  mientras porVisitar  $\neq \emptyset$  hacer  
    u = vertice de porVisitar cuya distancia a v es la mínima  
    porVisitar.eliminar(u);  
  
    para cada vértice w adyacente a u tal que w  $\in$  porVisitar  
      hacer  
        si distancia[u] + w((u, w)) < distancia[w] entonces  
          distancia[w] = distancia[u] + w((u, w))  
        fin si  
      fin para  
    fin mientras  
  devolver distancia  
}
```

Conjunto voraz(C:conjunto)  
{

```
  {C es el conjunto de candidatos}  
  S  $\leftarrow \Phi$  {Construimos la solución en el conjunto S}  
  mientras ( C  $\neq \Phi$  y no solución( s ) )  
    x  $\leftarrow$  seleccionar ( C )  
    C  $\leftarrow$  C  $\setminus$  {x}  
  
    Si factible ( S U {x} ) S  $\leftarrow$  S U {x}  
  
  si solución(S) devolver S  
  sino devolver "no hay soluciones"  
}
```

# El esquema de vuelta atrás:

## Introducción

- Los algoritmos de vuelta atrás (también conocidos como de *backtracking*) hacen una búsqueda sistemática de todas las posibilidades, sin dejar ninguna por considerar. Cuando intenta una solución que no lleva a ningún sitio, retrocede deshaciendo el último paso, e intentando una nueva variante desde esa posición (es normalmente de naturaleza recursiva).
- El proceso general de los algoritmos de *vuelta atrás* se contempla como un método de prueba y búsqueda, que gradualmente construye tareas básicas y las inspecciona para determinar si conducen a la solución del problema. Si una tarea no conduce a la solución, prueba con otra tarea básica. Es una prueba sistemática hasta llegar a la solución, o bien determinar que no hay solución por haberse agotado todas las opciones que probar.
- La característica principal de los algoritmos de vuelta atrás es intentar realizar pasos que se acercan cada vez más a la solución completa. Cada paso es anotado, borrándose tal anotación si se determina que no conduce a la solución, esta acción constituye la vuelta atrás. Cuando se produce una *vuelta atrás* se ensaya otro paso (otro movimiento). En definitiva, se prueba sistemáticamente con todas las opciones posibles hasta encontrar una solución, o bien agotar todas las posibilidades sin llegar a la solución.

# El esquema de vuelta atrás: Esquema general

*EnsayarSolucion*

{

*Inicializar cuenta de opciones de selección*

**mientras** *(no se hayan mirado todas las posibilidades y no se haya alcanzado el objetivo)*

{

*Seleccionar nuevo paso hacia la solución*

*Si se ha llegado al final, Anotar el paso en la solución  
objetivo alcanzado*

**si no**

{

*EnsayarSolucion a partir del nuevo paso*

**si (objetivo)**

*anotar el paso en la solución*

**si no** *deshacer el último paso realizado*

}

}

# El esquema de vuelta atrás: Problema de las 8 reinas

- El problema consiste en colocar ocho reinas en un tablero de ajedrez sin que se den jaque (dos reinas se dan jaque si comparten fila, columna o diagonal).
- Puesto que no puede haber más de una reina por fila, podemos replantear el problema como "colocar una reina en cada fila del tablero de forma que no se den jaque". En este caso, para ver si dos reinas se dan jaque basta con ver si comparten columna o diagonal. Por lo tanto, toda solución del problema se puede representar como una 8-tupla  $(X_0, \dots, X_7)$  en la que  $X_i$  es la columna en la que se coloca la reina que está en la fila  $i$  del tablero.
- Representación de la información:
  - Debe permitir interpretar fácilmente la solución:
  - $X$  vector  $[0..n-1]$  de enteros.  $X[i]$  almacena la columna de una reina en la fila  $i$ -ésima.
  - Evaluación:
    - Se utilizará un método buenSitio que devuelva el valor verdad si la  $k$ -ésima reina se puede colocar en el valor almacenado en  $X[k]$ , es decir, si está en distinta columna y diagonal que las  $k-1$  reinas anteriores.
    - Dos reinas están en la misma diagonal si tienen el mismo valor de "fila + columna", también están en la diagonal si tienen el mismo valor de "fila - columna".
      - $(f_1 - c_1 = f_2 - c_2) \vee (f_1 + c_1 = f_2 + c_2)$
      - $\Leftrightarrow (c_1 - c_2 = f_1 - f_2) \vee (c_1 - c_2 = f_2 - f_1)$
      - $\Leftrightarrow |c_1 - c_2| = |f_1 - f_2|$



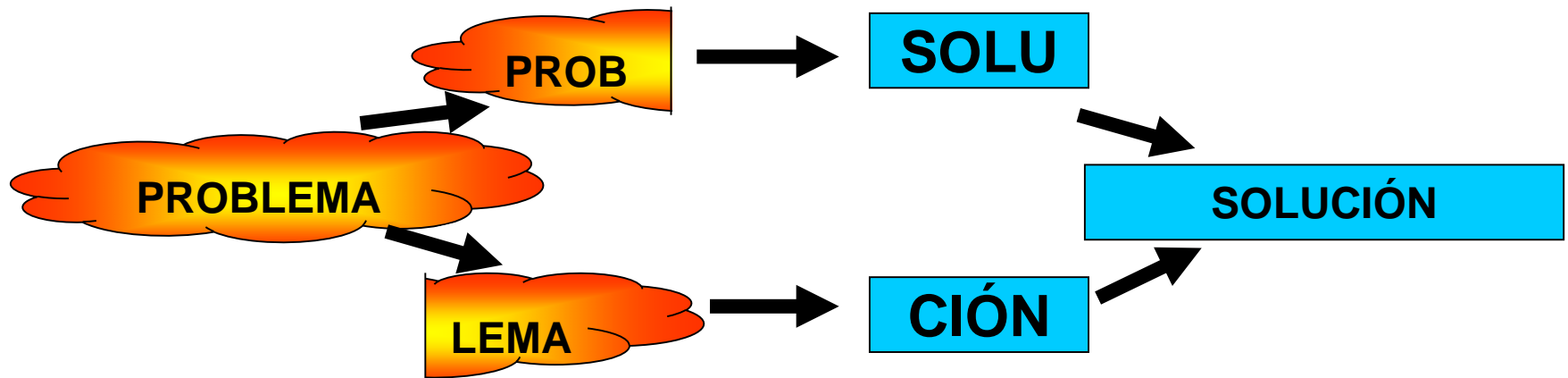
# El esquema de vuelta atrás: Problema del laberinto

- Nos encontramos en una entrada de un laberinto y debemos intentar atravesarlo. Dentro del algoritmo nos encontraremos con muros que no podremos atravesar, sino que habrá que rodear, lo que hará que nos encontremos a veces en un callejón sin salida. Es necesario tener en cuenta las siguientes condiciones:
- Representación: Array  $N \times N$ , de casillas marcadas como libre u ocupada por una pared.
- Es posible pasar de una casilla a otra moviéndose solamente en vertical u horizontal.
- El problema se soluciona cuando desde la casilla (0,0) se llega a las casilla (n-1, n-1).
- Para resolver este problema se diseñará un algoritmo de búsqueda con retroceso de forma que se marcará en la misma matriz del laberinto un camino solución si existe.
- Si por un camino recorrido se llega a una casilla desde la que es imposible encontrar una solución, hay que volver atrás y buscar otro camino.
- Además hay que marcar las casillas por donde ya se ha pasado para evitar meterse varias veces por el mismo callejón sin salida, dar vueltas alrededor de columnas....



# El esquema divide y vencerás: Introducción

- descomponer el ejemplar a resolver en un cierto número de subejemplares más pequeños del mismo problema;
  - resolver independientemente cada subejemplar;
  - combinar los resultados obtenidos para construir la solución del ejemplar original.
- aplicar esta técnica recursivamente





# El esquema divide y vencerás

- Como ejemplo, está que el problema de ordenar un vector admite dos soluciones siguiendo este esquema: los algoritmos *Merge Sort* y *Quick Sort*. Algoritmos que se trataron en la asignatura AED I en el tema de *algoritmos de ordenación y búsqueda*.

El primer nivel de diseño de *Merge Sort* puede expresarse así:

*si v es de tamaño 1 entonces v ya está ordenado*

*sino*

*Dividir v en dos subvectores A y B*

*Ordena A y B usando Merge Sort*

*Mezclar las ordenaciones de A y B para generar el vector ordenado.*

El siguiente algoritmo, *Quick Sort*, resuelve el mismo problema siguiendo también la estrategia de divide y vencerás:

*si v es de tamaño 1 entonces v ya está ordenado*

*sino*

*Dividir v en dos bloques A y B*

*Con todos los elementos de A menores que los de B*

*Ordenar A y B usando Quick sort*

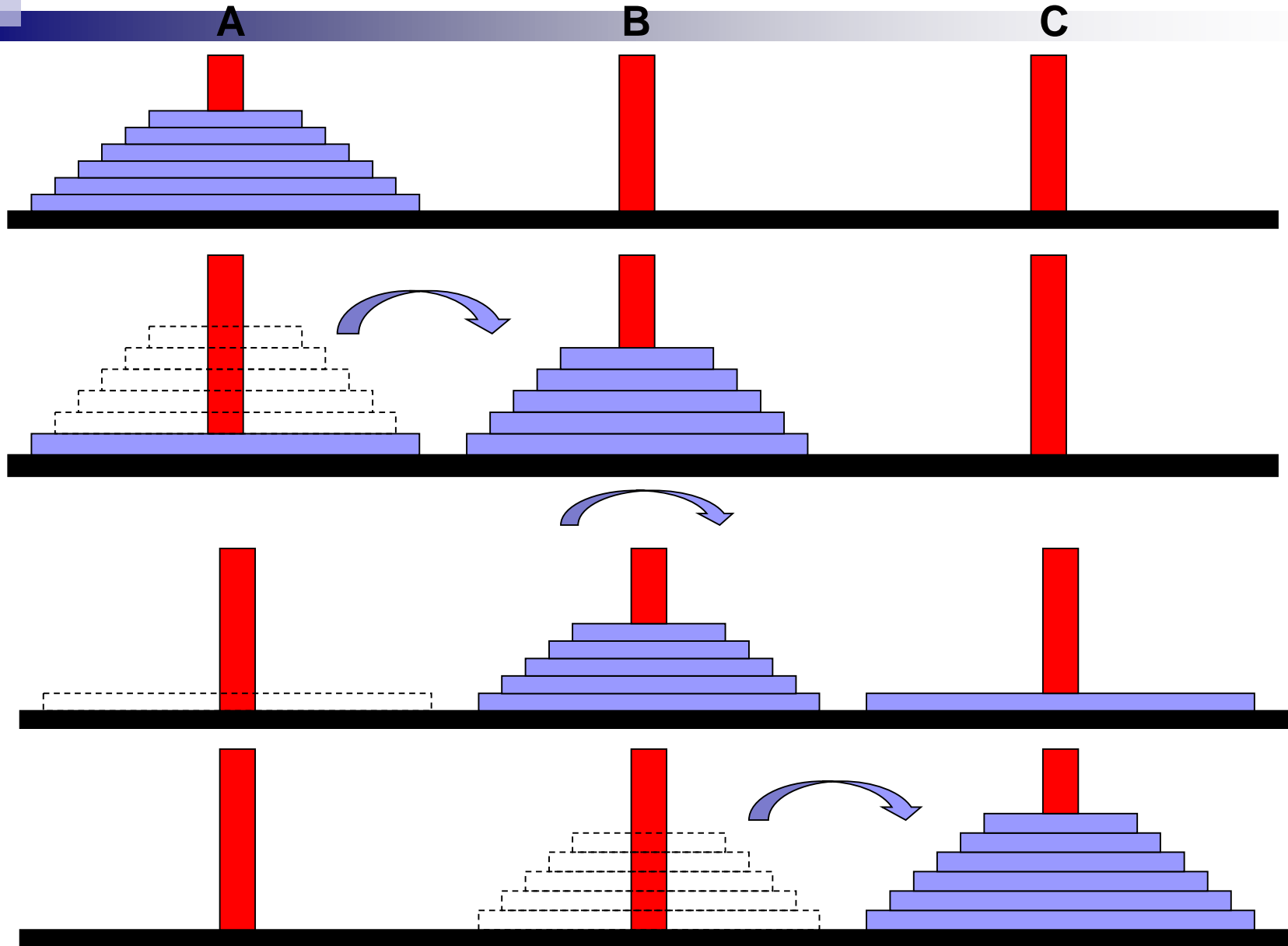
*Devolver v ya ordenado como concatenación de las ordenaciones de A y de B.*

Para que el esquema algorítmico divide y vencerás sea eficiente es necesario que el tamaño de los subproblemas obtenidos sea similar.

# El esquema divide y vencerás: Torres de Hanoi

- El de las **Torres de Hanoi** es un juego matemático consistente en mover unos discos de una torre a otra. La leyenda cuenta que existe un templo (llamado *Benares*), bajo la bóveda que marca el centro del mundo, donde hay tres varillas de diamante creadas por Dios al crear el mundo, colocando 64 discos de oro en la primera de ellas. Unos monjes mueven los discos a razón de uno por día, y, el día en que tengan todos los discos en la tercera varilla, el mundo terminará. Como se comprobará a continuación, en realidad 64 discos son suficientes para muchos años.
- En este juego, se trata de pasar un número de discos (típicamente, con tres existe una dificultad suficiente como para plantearlo como un pasatiempo), de un poste de origen (el primero, más a la izquierda) a un poste de destino (el tercero, a la derecha), utilizando como poste auxiliar el del medio. Sólo se puede mover un disco de cada vez, y nunca poner un disco sobre un segundo que sea de menor diámetro que el primero. Así, al comienzo del juego todos los discos están apilados en el primero (el de la izquierda), cada disco se asienta sobre otro de mayor diámetro, de manera que tomados desde la base hacia arriba, su tamaño es decreciente. El objetivo, como ya se ha dicho, es mover uno a uno los discos desde el poste A (origen) al poste C (destino) utilizando el poste B como auxiliar, para lo cuál se puede emplear una técnica *divide y vencerás*, como se explica a continuación.

# El esquema divide y vencerás: Torres de Hanoi



# El esquema divide y vencerás: Torres de Hanoi

Vamos a plantear la solución de tal forma que el problema vaya dividiendo en problemas más pequeños, y a cada uno de ellos aplicarles la misma solución. Se puede expresar así:

El problema de mover  $n$  discos de A a C consiste en:

*mover los  $n-1$  discos superiores de A a B*

*mover el disco  $n$  de A a C*

*mover los  $n-1$  discos de B a C*

Un problema de tamaño  $n$  ha sido transformado en un problema de tamaño  $n-1$ . A su vez cada problema de tamaño  $n-1$  se transforma en otro de tamaño  $n-2$  (empleando el poste libre como auxiliar).

El problema de mover los  $n-1$  discos de A a B consiste en:

*mover los  $n-2$  discos superiores de A a C*

*mover el disco  $n-1$  de A a B*

*mover los  $n-2$  discos de C a B*

De este modo se va progresando, reduciendo cada vez un nivel de dificultad del problema hasta que sólo haya que mover un único disco. La técnica consiste en ir intercambiando la finalidad de los postes, origen, destino y auxiliar. La condición de terminación es que el número de discos sea 1. Cada acción de mover un disco realiza los mismos pasos, por lo que puede ser expresada de manera recursiva.