VC2 – Diseño de Algoritmos

03MAIR – Algoritmos de Optimización

viu Universidad Internacional de Valencia

Agenda

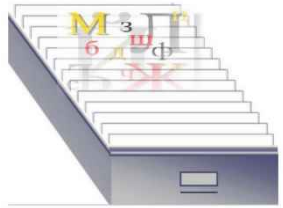
1. Algoritmos de ordenación
2. Algoritmos voraces. Grafos (Prim y Kruskal)
3. Técnica de Vuelta atrás
4. Técnica de Divide y Vencerás
5. Técnica de Programación Dinámica

Algoritmos de Ordenación

- **Definición**: Ordenar una lista de valores numéricos

- Algoritmos:
  - ✓ Por Inserción
  - ✓ Por Selección
  - ✓ Burbuja
  - ✓ Mezcla
  - ✓ Montículos
  - ✓ Ordenación rápida (Quick Sort)

- Utilizaremos algunas **operaciones no elementales**:
  - ✓ Obtener el máximo o el mínimo de una lista : $O(n)$
  - ✓ Intercambiar dos valores de una lista: $O(1)$



Algoritmos de Ordenación(I)

- ✓ **Inserción**
  - ✓ Se recorre la lista completa para en cada iteración construir una sub-lista ordenada.
  - ✓ Aconsejable para listas pequeñas y parcialmente ordenadas.
  - ✓ Es el método que usamos manualmente.
  - ✓ Complejidad (varia con la disposición inicial):
    - $O(n)$ para el caso más favorable
    - $O(n^2)$ para el caso más desfavorable
  - ✓ Aconsejable para **listas pequeñas y parcialmente ordenadas**

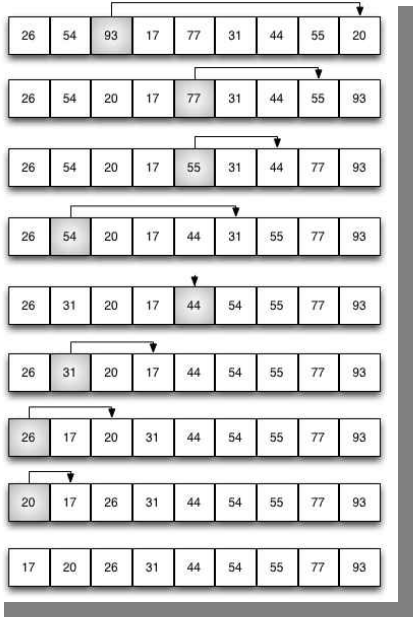
Algoritmos de Ordenación(II)

- ✓ **Burbuja**
  - ✓ Se recorre la lista para intercambiar cada elemento con el adyacente(si es preciso) en cada iteración, reduciendo progresivamente(de uno en uno) el tamaño de la lista.
  - ✓ Se puede mejorar deteniendo el proceso si no hay intercambios
  - ✓ Complejidad: siempre $\boldsymbol{O(n^2)}$ independientemente de la disposición inicial(excepto por mejoras)

54	26	93	17	77	31	44	55	20
26	54	93	17	77	31	44	55	20
26	54	93	17	77	31	44	55	20
26	54	17	93	77	31	44	55	20
26	54	17	77	93	31	44	55	20
26	54	17	77	31	93	44	55	20
26	54	17	77	31	44	93	55	20
26	54	17	77	31	44	55	93	20
26	54	17	77	31	44	55	20	93

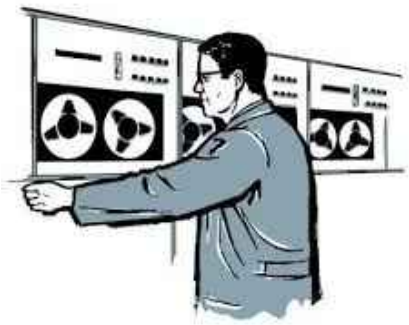
Algoritmos de Ordenación(III)

- ✓ **Selección**
  - ✓ Se recorre la lista para encontrar el máximo (o el mínimo) en cada iteración y situarlo en la posición que le corresponde, reduciendo progresivamente(de uno en uno) el tamaño de la lista.
  - ✓ También es un método usado manualmente.
  - ✓ Complejidad: siempre $O(n^2)$ independientemente de la disposició



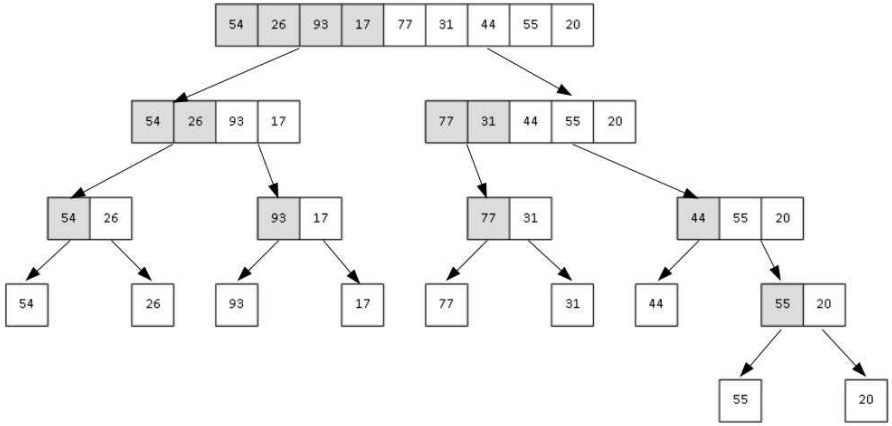
Algoritmos de Ordenación(IV)

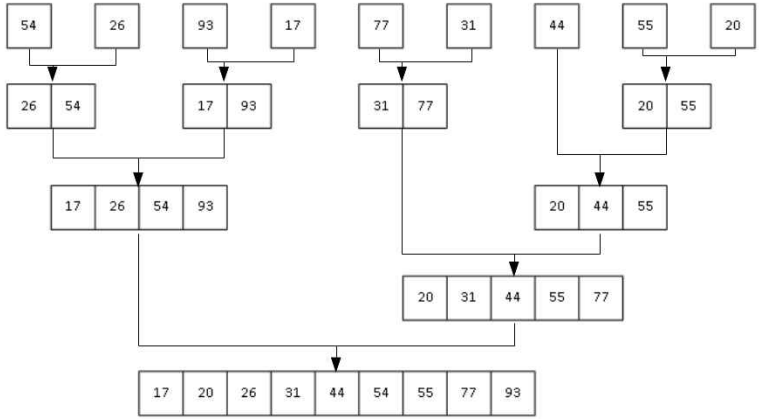
- ✓ **Mezcla(Merge Sort)**
  - ✓ Divide recursivamente la lista en dos mitades y combina los resultados
  - ✓ Basado en la técnica "**Divide y Vencerás**"
  - ✓ Muy usado en el pasado dado que se adapta bien al acceso secuencial.
  - ✓ Complejidad: siempre $\mathbf{O(n \cdot log(n))}$ en todos los casos
  - ✓ El cálculo se realiza por **ecuaciones en recurrencia**



Algoritmos de Ordenación(V)

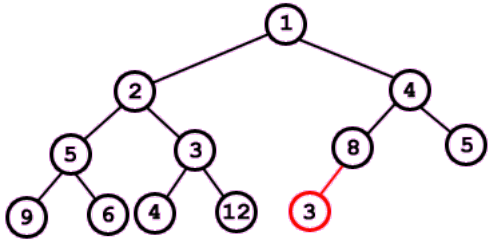
- ✓ Mezcla(Merge Sort)

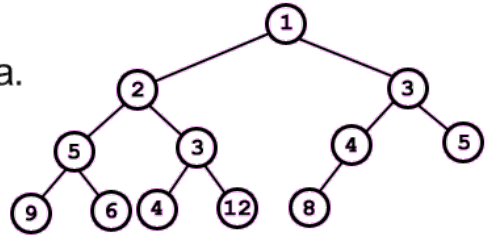




Algoritmos de Ordenación(VI)

- ✓ **Montículos(Heap Sort)**
  - ✓ Construye un árbol binario completo donde el valor del padre es menor que el de sus hijos.
  - ✓ En cada iteración se coloca un nuevo nodo reajustando recursivamente el resto si es necesario.
  - ✓ Complejidad: siempre $\mathbf{O(n \cdot log(n))}$ en el peor caso aunque mejora en casos favorables.
  - ✓ Se comporta bien en casos desfavorables y muy bien en favorables y sin coste de almacenamiento extra.
  - ✓ Algo complicado de implementar.





https://brilliant.org/wiki/heap-sort/

https://d18l82el6cdm1i.cloudfront.net/uploads/uv9rgMfetq-heapsort-example.gif

Algoritmos de Ordenación(VI)

- ✓ **Ordenación Rápida(Quick Sort)**
  - ✓ Proceso en dos partes:
    1) Divide recursivamente la lista inicial en dos de tal manera que todos los elementos de la primera lista son menores que los de la segunda
    2) Se combinan las dos listas
  - ✓ Basado en la técnica “Divide y Vencerás”
  - ✓ Complejidad: $O(n^2)$ en el peor caso y $\mathbf{O(n \cdot log(n))}$ en el mejor caso o en promedio.
  - ✓ Los casos favorables no dependen de los datos sino de la elección del pivote para dividir las listas.

Algoritmos de Ordenación(VII)

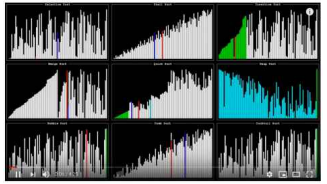
- **Resumen de complejidades**:

Algoritmo	Mejor	Peor	Medio	Espacio
Ordenacir mezcla	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(n)$
Ordenación por inserción	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Ordenación por burbuja	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Ordenación rápida	$O(n\log(n))$	$O(n^2)$	$O(n\log(n))$	$O(1)^a$
Ordenación por montículos	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$	$O(1)$

[a] En el mejor caso incluso es $\log(n)$.

Análisis comparativo de algoritmos de ordenamiento
https://pereiratechtalks.com/analisis-de-algoritmos-de-ordenamiento/

- **Otras ordenaciones**
  - ✓ Shell: mejora a inserción por comparación con varias posiciones.
  - ✓ Burbuja bidireccional (cocktail sort). Ordena por los dos extremos
  - ✓ Timsort, Tree Sort, Bucket Sort, Radix Sort, ...



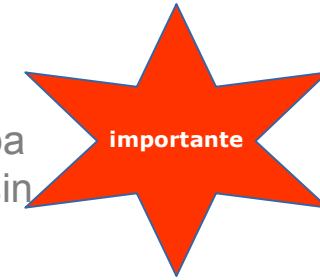
https://www.youtube.com/watch?v=ZZuD6iUe3Pc

Algoritmos de Ordenación(VIII)

- **Resumen de aplicaciones.¿Donde se utiliza?**:

  - ✓ Ordenar resultados
  - ✓ Realizar búsquedas
  - ✓ Encontrar la mediana o el par mas cercano
  - ✓ Identificar “outliers” / anomalías
  - ✓ Detectar duplicados
  - ✓ …
  - ✓ Compresión de datos
  - ✓ Planificación de tareas
  - ✓ Simulaciones
  - ✓ ….

Técnica Voraz. Algoritmos voraces(I) - (*Greedy algorithms)*



- **Definición:** Basada en la división del problema por etapas, eligiendo en cada etapa una decisión para construir la solución que resulte la más adecuada o ambiciosa sin considerar las consecuencias. Las decisiones descartadas será descartadas para siempre. **Resumen: elegir en cada etapa la decisión optima.**
- Características que permiten identificar problemas aplicables:
  - ✓ Conjunto de candidatos (elementos seleccionables por etapas)
  - ✓ Solución parcial
  - ✓ **Función de selección** para determinar el mejor candidato en cada etapa.
  - ✓ Función objetivo
  - ✓ Función de factibilidad que asegure que una selección parcial es “prometedora”
  - ✓ Criterio o función que compruebe que una solución parcial ya es una solución final.

Técnica Voraz. Algoritmos voraces(II)

- Resulta muy eficiente y aconsejable en muchos problemas donde por **características del problema**, tomar en cada etapa la mejor decisión conduce a la optima.
- Pero no en todos los casos se cumple! Deberemos demostrar formalmente que las mejores decisiones parciales llevan a la mejor solución final.
- Otra dificultad. Encontrar la **función de selección** que determine la mejor decisión parcial.
- Si se cumplen todas las condiciones favorablemente es la técnica mas recomendable por su eficiencia.
- Es apropiada para problemas de optimización pero también la podemos usar para encontrar soluciones factibles cuando el espacio de soluciones es grande.

Técnica Voraz. Algoritmos voraces(III)

- Esquema general o seudocódigo

```
S = ∅
while (S no sea una solución y C ≠ ∅) {
      x = selección(C)
      C = C - {x}
      if (S∪{x} es factible)
         S = S∪{x}
}

if (S es una solución)
   return S;
else
   return "No se encontró una solución";
```

Técnica Voraz. Algoritmos voraces(IV)

Problema: Devolver cambio de monedas

- Dado un sistema monetario (1,v1,v2,v3,… vn) descomponer cualquier cantidad C de manera que usemos el menor numero de monedas.
- Elementos:
  - ✓ Conjunto de candidatos: Todas las combinaciones posibles de obtener C
  - ✓ Función de selección en cada etapa, tomar tantas monedas como sea posible del mayor valor, sin pasarse (voracidad)
  - ✓ Función de factibilidad es el valor alcanzado con la suma de todas las monedas elegidas hasta el momento
  - ✓ Función objetivo es la cantidad total de monedas(función a minimizar)
  - ✓ Criterio de solución final es la similar(en este caso) que la función de factibilidad pero que debe coincidir con C.

Técnica Voraz. Algoritmos voraces(V)

Problema: Devolver cambio de monedas

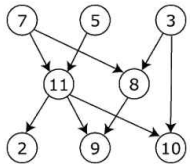
- La aplicación de la técnica voraz es eficiente bajo algunos **supuestos**:
  - ✓ Debe existir el valor unitario (1) en el conjunto de monedas pues en otro caso no será posible obtener todas las cantidades.
  - ✓ Disponemos una cantidad infinita de monedas
  - ✓ La “voracidad” no es eficiente en todos los sistemas monetarios.

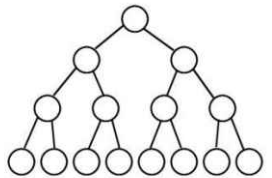
P.Ej: (1,5,11) para conseguir la cantidad C=15

Técnica Voraz. Algoritmos voraces(VI). Grafos

Uso de grafos para modelizar problemas

- **Definición de grafo:** Es una pareja de conjuntos V(**vértices**) y A(**Aristas**) donde cada arista se compone de un par de vértices.
- Si el par de vértice (u,v) se considera diferente de (v,u) decimos que el **grafo es dirigido** y se representan las aristas con flechas. En caso contrario decimos que es no dirigido y las aristas se representan con líneas.
- Decimos que es **conexo** si todos los vértices están conectados directamente o por nodos intermedios.
- Decimos que es **ponderado** si todas las aristas tienen asociado un valor numérico.
- Definimos un **árbol** como un subconjunto de aristas que conectan todos los nodos de forma única (no hay ciclos).

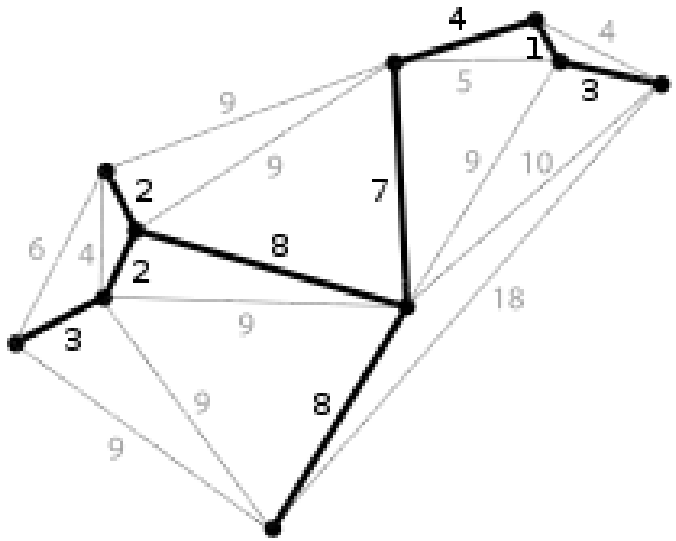




Técnica Voraz. Algoritmos voraces(VII). Grafos

- **Definición: Árbol de recubrimiento mínimo de un grafo**.

  Es un árbol del grafo ponderado que contiene todos los vértices y cuyo peso es mínimo. El peso del árbol es la suma del valor de las aristas.

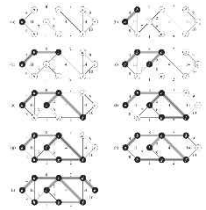


Técnica Voraz. Algoritmos voraces(VII). Grafos

- Existen dos algoritmos que proporcionan una solución para obtener el **árbol de recubrimiento mínimo**.

Algoritmo Prim

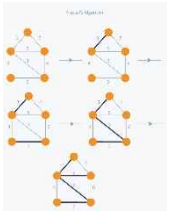
- Para cada vértice elegimos la arista de menor coste que une el subconjunto de vértices no seleccionados con el subconjunto de vértices seleccionados.
- El vértice del subconjunto de los no seleccionados pasa al conjunto de los seleccionados y se continua con uno nuevo.



https://i.stack.imgur.com/TTwpR.png

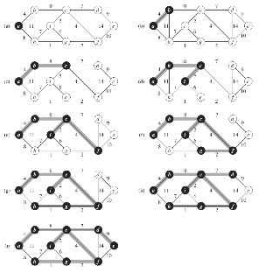
Algoritmo Kruskal

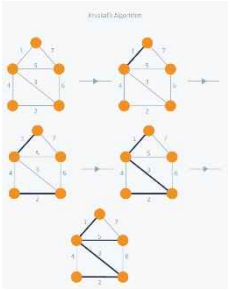
- Se ordenan las aristas por orden decreciente.
- Se van seleccionando aristas de tal manera que no se formen ciclos(las aristas que formen ciclos se eliminan para no ser considerados en las siguientes etapas)



https://he-s3.s3.amazonaws.com/media/uploads/6322896.jpg

Técnica Voraz. Algoritmos voraces(VIII). Grafos





- Complejidad

Algoritmo Prim

- Siempre $O(n^2)$.

Algoritmo Kruskal

- $O(\ a \cdot \log(n)\ )$. Donde a= nº de aristas y n=nº de vértices

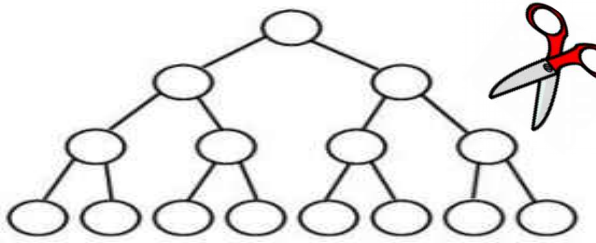
La elección de uno u otro dependerá del número total del aristas:

✓ Si hay muchas aristas (se aproxima a n·(n-1)) la complejidad de Kruskal se acerca a $O(\ n^2 \cdot \log(n))$ peor que Prim

✓ Si hay pocas aristas (se aproximan a n) la complejidad de Kruskal se acercará a $O(\ n \cdot \log(n)\ )$ mejorando a Prim

Vuelta Atrás. Backtracking (I)

- **Definición:** Construcción sistemática y por etapas de todas las soluciones posibles que pueden representarse mediante una tupla (x1, x2,… Xn) en la que cada componente xi puede explorarse en la etapa i-esima. A través de un **árbol de expansión** se modela todo el espacio de soluciones donde cada nodo es un valor diferente para cada elemento xi.
- Características que permiten identificar problemas aplicables:
  - ✓ Problemas discretos en los que las soluciones se componen de elementos que pueden ser relacionado para expresarlos en un árbol de expansión.
  - ✓ Es posible encontrar un criterio para descartar determinadas ramas(ramificación y poda[*]) y evitar un análisis exhaustivo (fuerza bruta)



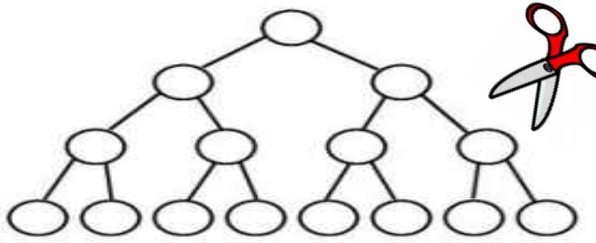
(*) La veremos más adelante asociada a la técnica general de búsqueda en arboles

Vuelta Atrás. Backtracking (II)

- La eficiencia en general no será buena (exponencial $O(2^n)$) a menos que podamos incorporar mejoras para evitar la exploración en toda la profundidad.

- La función de descarte debe ser tan sencilla en términos de complejidad como el coste de la exploración exhaustiva. En caso contrario no mejoraremos.

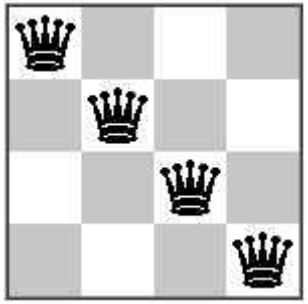
- El modelo debe tener en cuenta las restricciones explicitas del problema. Nuestro diseño será mejor en la medida que seamos capaces de identificar e implementar las restricciones implícitas para evitar la exploración completa.



Vuelta Atrás. Backtracking (III)

Problema: Problema de las 4 reinas en 4x4

- Colocar en un tablero 8 reinas sin que se amenacen. (simplificamos a 4 reinas)
- ¿Como lo modelamos? : Estructura de datos para establecer las soluciones y restricciones.



n	Size of solution space ($n!$)	Number of solutions
1	1	1
2	2	0
3	6	0
4	24	2
5	120	10
6	720	4
7	5040	40
8	40320	92
9	362880	352
10	3628800	724
11	39916800	2680
12	479001600	14200
13	6227020800	73712
14	87178291200	365596
15	1307674368000	2279184
16	20922789888000	14772512
17	355687428096000	95815104
18	6402373705728000	666090624
19	121645100408832000	4968057848
20	2432902008176640000	39029188884
21	51090942171709440000	314666222712
22	1124000727777607680000	2691008701644
23	25852016738884976640000	24233937684440
24	620448401733239439360000	227514171973736
25	15511210043330985984000000	2207893435808352
26	403291461126605635584000000	22317699616364044

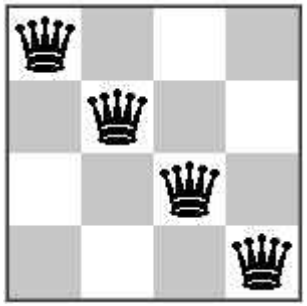
Referencia: Fast Search Algorithms for the N-Queens Problem(Rok Sosic and Jun Gu)
https://pdfs.semanticscholar.org/79d2/fa13d4a5cfc02ff6936b6083bb620e4e0ce1.pdf

Fuente: https://www.kaggle.com/mrknoot/genetic-algorithms-solving-the-n-queens-problem

Vuelta Atrás. Backtracking (III)

Problema: Problema de las 8 reinas en 8x8

- Colocar en un tablero 4 reinas sin que se amenacen. (simplificamos a 4x4 reinas)
- ¿Como lo modelamos? : Estructura de datos para establecer las soluciones y restricciones.



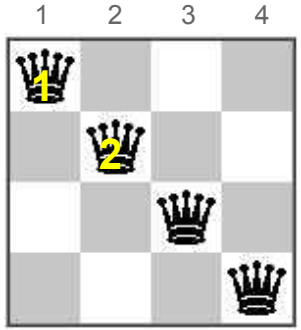
Vuelta Atrás. Backtracking (IV)

Problema: Las 4 reinas en 4x4

- Solución : 4-tuplas (x1,x2,x3,x4) donde el valor de cada elemento es la fila donde está posicionada la reina en la columna i-esima. P.ej la del dibujo es (1,2,3,4)
- El árbol de expansión recorrerá todas las posibilidades.
- Con este modelo, es posible determinar si una solución parcial (rama del árbol) es "prometedora"
- No puede haber dos reinas en la misma columna. Esta restricción se verifica por el modelo que hemos adoptado.
- Dos reinas estarán en la misma fila si hay dos valores iguales para una solución parcial.

P.Ej: (1,1,*,*) representa las dos primeras reinas en la 1ª fila.

- Dos reinas estará en la misma diagonal si |xi -xj| = |i-j|



Divide y vencerás (I)

- **Definición:** Es posible dividir el problema principal recursivamente en sub-problemas más pequeños similares al original o sencillos de resolver.
- Características que permiten identificar problemas aplicables:
  - ✓ El problema puede ser descompuesto en problemas mas pequeños pero de la misma naturaleza que el principal.
  - ✓ Es posible resolver estos sub-problemas de manera recursiva o de otra manera sencilla.
  - ✓ Es posible combinar las soluciones de los sub-problemas para componer la solución al problema principal.
  - ✓ Los sub-problemas son disjuntos entre si. No hay solapamiento entre los sub-problemas.
  - ✓ Debemos asegurar que el proceso de divisiones recursivas finaliza en algún momento.

Divide y vencerás (III)

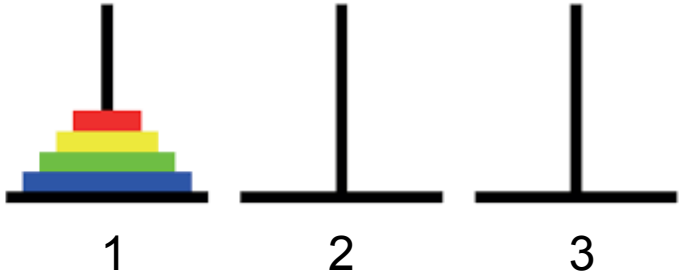
Aplicaciones:

- Algoritmos de ordenación mergesort y quicksort
- Multiplicación de matrices (A.de Strassen). (https://es.wikipedia.org/wiki/Algoritmo_de_Schonhage-Strassen)
- Sub-secuencia de suma máxima (https://es.wikipedia.org/wiki/Subvector_de_suma_m%C3%A1xima)
- Par de puntos mas cercanos (https://es.wikipedia.org/wiki/Problema_del_par_de_puntos_m%C3%A1s_cercanos)
- Eliminación de superficies ocultas en reproducciones 3D
- Calendario de una liga
- ….

Divide y vencerás (IV)

Problema: Torres de Hanoy

- Llevar las fichas de la torre 1 a la torre 3
- Restricciones:
  - ✓ Solo es posible mover las fichas de una en una.
  - ✓ No se puede colocar una ficha encima mayor encima de una ficha menor.
  - ✓ Es posible usar la torre 2 a modo de pivote.



Divide y vencerás (V)

Problema: Torres de Hanoy

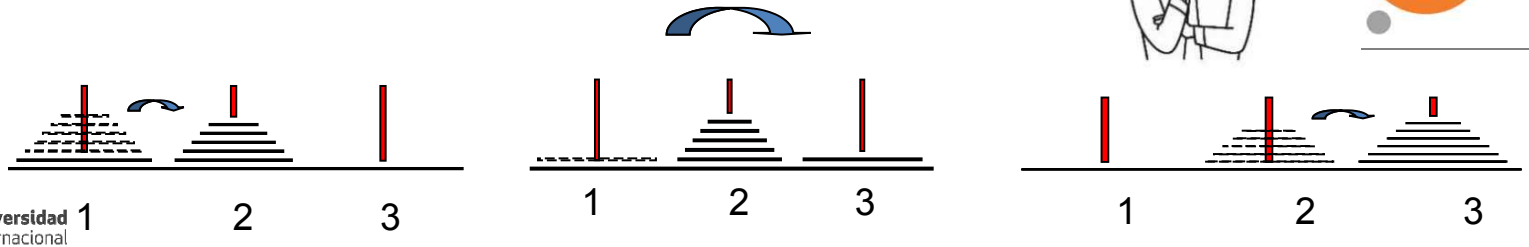
- La estrategia consiste en darse cuenta que se puede dividir el problema en otro problema similar con una ficha menos.
- En nuestro caso (4 fichas) :

Resolver(Total_fichas=4, Desde=1 , Hasta=3) es valido con:

✓ Resolver(Total_fichas=3, Desde=1, Hasta=2)

✓ Mover(Desde=1, Hasta=3)

✓ Resolver(Total_fichas=3, Desde=2, Hasta=3)



Divide y vencerás(VII). Recursividad

- La recursividad permite diseños sencillos y legibles pero debemos tener en cuenta evaluar la complejidad en cuanto a uso de memoria o espacio en disco pues **en cada iteración multiplicamos el uso de recursos**.
- Solución a ecuaciones en recurrencia:

Ecuaciones

$$T_n = \begin{cases} c \cdot n^k, 1 \leqslant n < b \\ a \cdot T\left(\frac{n}{b}\right) + c \cdot n^k, \wedge n \geqslant b \end{cases}$$

Soluciones

$$T(n) = O(n^k) \text{ si } a < b^k$$

$$T(n) = O(n^k \cdot \log(n)) \text{ si } a = b^k$$

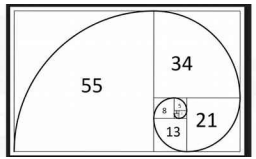
$$T(n) = O(n^{\log_b(a)}) \text{ si } a > b^k$$

Divide y vencerás (VIII)

Problema: Sucesión de Fibonacci

- Fn = Fn-1 + Fn-2 F1=1 , F2=1 (cada termino es la suma de los 2 anteriores)
  - ✓ A través de recurrencia la eficiencia se "resiente". La complejidad es exponencial! $O(2^n)$
  - ✓ La razón es que se repiten cálculos Para el termino F5 se deben calcular F4 y F3 pero para el termino F6 se vuelve a calcular F4
  - ✓ Es posible reducirla a lineal ($O(n)$) si se utiliza una técnica iterativa

Practica y comparte:
- Diseña un algoritmo con recursión
- Mejora el algoritmo con iteración
- ¿Ves algo raro usando recursión?



Programación dinámica (I)

- **Definición:** Es posible dividir el problema en subproblemas más pequeños, guardando las soluciones para ser utilizadas más adelante.
- Características que permiten identificar problemas aplicables:
  - ✓ Es posible almacenar soluciones de los subproblemas para ser utilizados más adelante
  - ✓ Debe verificar el **principio de optimalidad** de Bellman: "*en una secuencia optima de decisiones, toda sub-secuencia también es óptima*" (*)
  - ✓ La necesidad de guardar la información acerca de las soluciones parciales unido a la recursividad provoca la necesidad de preocuparnos por la complejidad espacial (cuantos recursos de espacio usaremos)

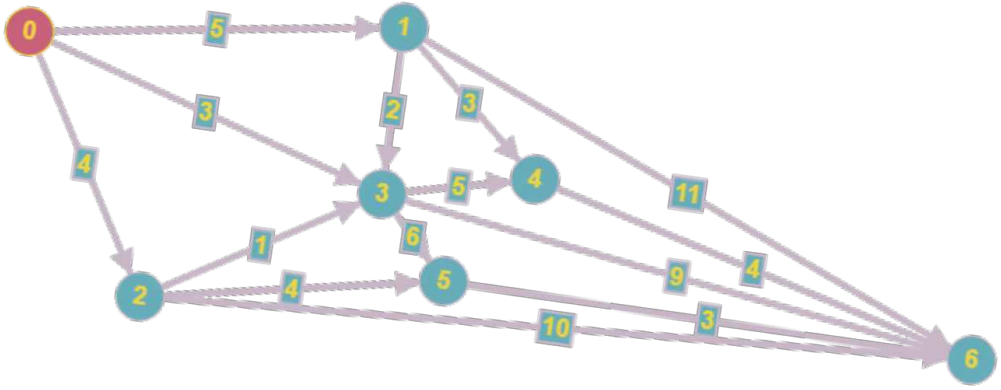
(*) Ojo! No significa que si tenemos soluciones optimas de los subproblemas las podamos combinar para obtener el optimo del problema principal.

Programación dinámica (II)

Problema: Viaje por el rio

- En un río hay n embarcaderos y debemos desplazarnos río abajo desde un embarcadero a otro. Cada embarcadero tiene precios diferentes para ir de un embarcadero a otro situado más abajo. Para ir del embarcadero i al j, puede ocurrir que sea más barato hacer un trasbordo por un embarcadero intermedio k. El problema consiste en determinar la combinación más barata.
- Restricciones:

✓ No hay posibilidad de remontar el rio. Siempre debemos ir rio abajo

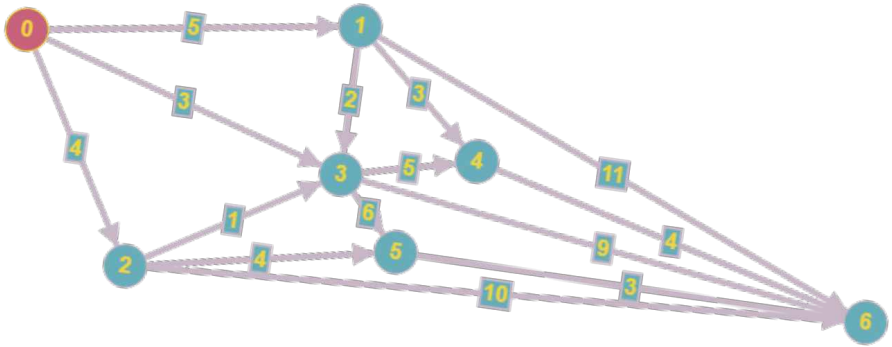


Programación dinámica (III)

Problema: Viaje por el rio

- Consideramos una tabla T(i,j) para almacenar todos los precios que nos ofrecen los embarcaderos
- Si no es posible ir desde i a j daremos un valor alto para garantizar que ese trayecto no se va a elegir en la ruta óptima(modelado habitual para restricciones)
- Establecer una tabla intermedia para de soluciones óptimas parciales para ir desde i a j.

$$C(i,j) = \min \{T(i,j) , T(i,k)+C(k,j) \text{ para todo } i<k<=j \}$$

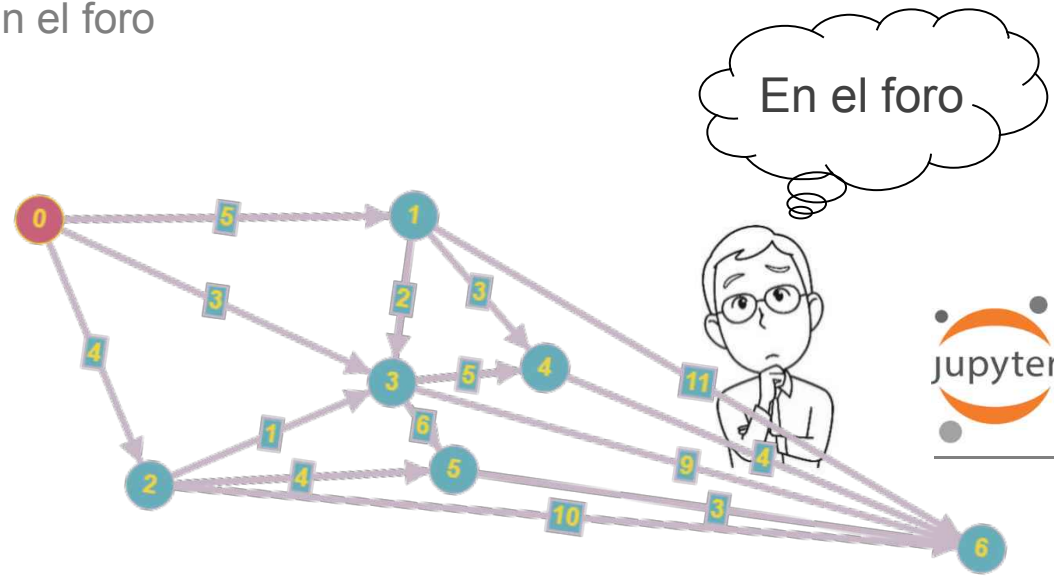


Programación dinámica (III)

Problema: Viaje por el rio

- Solución e implementación en Python para el próximo día

- Propuestas bienvenidas en el foro



Gracias

raul.reyero@campusviu.es

viu Universidad Internacional de Valencia