

Se desea encontrar el camino más corto entre dos ciudades. Para ello se dispone de una tabla con la distancia entre los pares de ciudades en los que hay carreteras o un valor centinela (por ejemplo, -1) si no hay, por lo que para ir de la ciudad inicial a la final es posible que haya que pasar por varias ciudades. También se conocen las coordenadas geográficas de cada ciudad y por tanto la distancia geográfica (en línea recta) entre cada par de ciudades. Se pretende acelerar la búsqueda de un algoritmo de ramificación y poda priorizando los nodos vivos (ciudades) que estén a menor distancia geográfica de la ciudad objetivo.

Seleccione una:

-   El nuevo algoritmo siempre sea más rápido
- Esta estrategia no asegura que se obtenga el camino más corto
-   El nuevo algoritmo solo será más rápido para algunas instancias del problema

Tratándose de un problema de optimización, en la lista de nodos vivos de ramificación y poda...

- sólo se introducen nodos prometedores, es decir, nodos que pueden mejorar la mejor solución que se tiene en ese momento
- puede haber nodos que no son prometedores
- las otras dos opciones son ciertas

Se desea obtener todas las permutaciones de una lista compuesta por  $n$  elementos, ¿Qué esquema es el más adecuado?

- Divide y vencerás, puesto que la división en sublistas se podría hacer en tiempo constante
- Ramificación y poda, puesto que con buenas funciones de cota es más eficiente para este problema que vuelta atrás
- Vuelta atrás, para este problema no hay un esquema más eficiente

Al resolver el problema del viajante de comercio mediante vuelta atrás y asumiendo un grafo de  $n$  vértices totalmente conexo, ¿cuál de estas es una buena cota pesimista al iniciar la búsqueda?

- Se ordenan las aristas restantes de menor a mayor distancia y se calcula la suma de las  $n$  aristas más cortas
- Se resuelve el problema usando un algoritmo voraz que añade cada vez al camino el vértice más cercano al último añadido
- Se multiplica  $n$  por la distancia de la arista más corta que nos queda por considerar

Si para resolver un mismo problema usamos un algoritmo de ramificación y poda y lo modificamos mínimamente para convertirlo en un algoritmo de vuelta atrás, ¿qué cambiamos realmente?

- provocamos que las cotas optimistas pierdan eficacia
- cambiamos la función que damos a la cota pesimista
- sería necesario comprobar si las soluciones son factibles o no puesto que ramificación y poda solo genera nodos factibles

¿Cuál es la diferencia principal entre una solución de vuelta atrás y una solución de ramificación y poda para el problema de la mochila?

- El orden de exploración de las soluciones
- El hecho que la solución de ramificación y poda puede empezar con una solución subóptima voraz y la de vuelta atrás no
- El coste asintótico en el caso peor

Se desea encontrar el camino más corto entre dos ciudades.

Para ello se dispone de una tabla con la distancia entre los pares de ciudades en los que hay carreteras o un valor centinela (por ejemplo, -1) si no hay, por lo que para ir de la ciudad inicial a la final es posible que haya que pasar por varias ciudades. Como también se conocen las coordenadas geográficas de cada ciudad se quiere usar la distancia geográfica (en línea recta) entre cada par de ciudades para como cota para limitar la búsqueda en un algoritmo de vuelta atrás.

¿Qué tipo de cota sería?

  Una cota pesimista

  Una cota óptimista

No se trataría de ninguna poda puesto que es posible que esa heurística no encuentre una solución factible

La ventaja de la estrategia ramificación y poda frente a vuelta atrás es que la primera genera las soluciones posibles al problema mediante...

- las otras dos opciones son verdaderas
  - un recorrido guiado por una cola de prioridad de donde se extraen
  - primero los nodos que representan los subárboles más prometedores del espacio de soluciones
  - un recorrido guiado por estimaciones de las mejores ramas del árbol que representa el espacio de soluciones

La estrategia de vuelta atrás es aplicable a problemas de selección y optimización en los que:

- El espacio de soluciones es un conjunto infinito
- El espacio de soluciones es un conjunto finito
- El espacio de soluciones puede ser tanto finito como infinito pero en este último caso debe ser al menos numerable

Se desea encontrar el camino más corto entre dos ciudades. Para ello se dispone de una tabla con la distancia entre los pares de ciudades en los que hay carreteras o un valor centinela (por ejemplo, -1) si no hay, por lo que para ir de la ciudad inicial a la final es posible que haya que pasar por varias ciudades. También se conocen las coordenadas geográficas de cada ciudad y por tanto la distancia geográfica (en línea recta) entre cada par de ciudades. Para limitar la búsqueda en un algoritmo de vuelta atrás, se utiliza la solución de un algoritmo voraz basado en moverse en cada paso a la ciudad, de entre las posibles según el mapa de carreteras, que esté más cercana al destino según su distancia geográfica.

Este algoritmo voraz, ¿serviría como cota pesimista?

Seleccione una:

- No, ya que no asegura que se encuentre una solución factible
- No, ya que en algunos casos puede dar distancias menores que la óptima
- Sí, puesto que la distancia geográfica asegura que otra solución mejor no es posible

Cuando resolvemos un problema mediante un esquema de ramificación y poda...

-   las decisiones sólo pueden ser binarias
-  los valores entre los cuales se elige en cada una de las decisiones pueden formar un conjunto infinito
-   los valores entre los cuales se elige en cada una de las decisiones tienen que formar un conjunto finito

Cuando se resuelve un algoritmo de vuelta atrás un problema de  $n$  decisiones, en el que siempre hay como mínimo dos opciones para cada decisión, ¿cuál de las siguientes complejidades en el caso peor es la mejor que nos podemos encontrar?

- $O(2^n)$
- $O(n^2)$
- $O(n!)$

La complejidad en el peor de los casos de un algoritmo de ramificación y poda...

- Puede ser exponencial con el número de alternativas por cada decisión
- Es exponencial con el número de decisiones a tomar
- Puede ser polinómica con el número de decisiones a tomar

En ausencia de cotas optimista y pesimistas, la estrategia de vuelta atrás...

-   no se puede usar para resolver problemas de optimización
-   no recorre todo el árbol si hay manera de descartar subárboles que representan conjuntos de soluciones no factibles
- debe recorrer siempre todo el árbol

Se desea encontrar el camino más corto entre dos ciudades. Para ello se dispone de una tabla con la distancia entre los pares de ciudades en los que hay carreteras o un valor centinela (por ejemplo, -1) si no hay, por lo que para ir de la ciudad inicial a la final es posible que haya que pasar por varias ciudades. Para limitar la búsqueda en un algoritmo de vuelta atrás, se utiliza la solución de un algoritmo voraz basado en moverse en cada paso a la ciudad, de entre las posibles según el mapa de carreteras, que esté más cercana al destino en línea recta.

¿Qué tipo de cota sería?

- Sería una cota pesimista siempre que se tenga la certeza de que esa aproximación encuentra la solución factible.
- Sería una cota optimista siempre que se tenga la certeza de que esa aproximación encuentra una solución factible
- Ninguna de las otras dos opciones

Decid cuál de estas tres no sirve como cota optimista para obtener el valor óptimo de la mochila discreta:

- El valor de la mochila discreta que se obtiene usando un algoritmo voraz basado en el valor específico de los objetos
- El valor de la mochila continua correspondiente
- El valor de una mochila que contiene todos los objetos aunque se pase del peso máximo permitido

La complejidad en el menor de los casos de un algoritmo de ramificación y poda...

- suele ser polinómica con el número de alternativas por cada decisión
- es siempre exponencial con el número de decisiones a tomar
- puede ser polinómica con el número de decisiones a tomar

## ¿Para qué sirven las cotas pesimistas en ramificación y poda?

-   Para descartar nodos basándose en la preferencia por algún otro nodo ya completado
-   Para tener la certeza de que la cota optimista está bien calculada
-   Para descartar nodos basándose en el beneficio esperado

En los algoritmos de ramificación y poda, ¿el valor de una cota pesimista es menor que el valor de una cota optimista?

- X ● Sí, siempre es así
- ✓ ○ En general sí, si se trata de un problema de maximización, aunque en ocasiones ambos valores pueden coincidir
- En general sí, si se trata de un problema de minimización, aunque en ocasiones ambos valores pueden coincidir

La estrategia de ramificación y poda necesita cotas pesimistas...

-   para decidir el orden de visita de los nodos del árbol de soluciones
-   para determinar si una solución es factible
-   sólo si se usa para resolver problemas de optimización

En el esquema de vuelta atrás, los mecanismos de poda basados en la mejor solución hasta el momento...

-   Las dos anteriores son verdaderas
  - garantizan que no se va a explorar nunca todo el espacio de soluciones posibles
  - pueden eliminar soluciones parciales que son factibles

## El uso de funciones de cota en ramificación y poda...

- garantiza que el algoritmo va a ser más eficiente ante cualquier instancia del problema
- puede reducir el número de instancias del problema que pertenecen al caso peor
- transforma en polinómicas complejidades que antes eran exponenciales

Decid cuál de estas tres es la cota pesimista más ajustada al valor óptimo de la mochila discreta:

-   El valor de la mochila continua correspondiente
-   El valor de una mochila que contiene todos los objetos aunque se pase del peso máximo permitido
-   El valor de la mochila discreta que se obtiene usando un algoritmo voraz basado en el valor específico de los objetos

## En la estrategia ramificación y poda...

- cada nodo tiene su propia cota optimista, la cota pesimista sin embargo, es común para todos los nodos
- cada nodo tiene su propia cota pesimista y también su propia cota optimista
- cada nodo tiene su propia cota pesimista, la cota optimista sin embargo, es común para todos los nodos

## En los algoritmos de ramificación y poda...

- Una cota optimista es necesariamente un valor insuperable, de no ser así se podría podar el nodo que conduce a la solución óptima
- Una cota optimista es necesariamente un valor alcanzable, de no ser así no está garantizado que se encuentre la solución óptima
- El uso de cotas pesimistas sólo resulta eficaz cuando se dispone de una posible solución de partida

Decid cuál de estas tres es la cota optimista que poda más eficientemente cuando se usa la estrategia de vuelta atrás para resolver el problema de la mochila:

-   El valor de una mochila
-   El valor óptimo de la mochila continua correspondiente
- El valor de la mochila discreta que se obtiene usando un algoritmo voraz basado en el valor específico del objeto

En los algoritmos de ramificación y poda, ¿el valor de una cota pesimista es mayor que el valor de una cota optimista? (entendiendo que ambas cotas se aplican sobre el mismo nodo)

-   No, nunca es así
-   En general sí, si se trata de un problema de maximización, aunque en ocasiones ambos valores pueden coincidir
-   En general sí, si se trata de un problema de minimización, aunque en ocasiones ambos valores pueden coincidir

Dado un problema de optimización cualquiera, ¿la estrategia de vuelta atrás garantiza la solución óptima?

- Es condición necesaria que el dominio de las decisiones sea discreto o discretizable y que el número de decisiones a tomar esté acotado
- Sí, siempre que el dominio de las decisiones sea discreto o discretizable y además se empleen mecanismos de poda basados en la mejor solución hasta el momento
- Sí, puesto que ese método analiza todas las posibilidades

El problema de la asignación de turnos tiene solución...

### **EL PROBLEMA DE LA ASIGNACION DE TURNOS.**

Estamos al comienzo del curso y los alumnos quieren buscar compañero para formar un grupo de prácticas. Para solucionar este problema se propone que elijan a varias personas y les asignen una prioridad. El número de alumnos es  $N$ . Se dispone una matriz cuadrada  $M$  con  $N$  filas en la que cada alumno escribió, en su fila correspondiente, un número entero (entre  $-1$  y  $N-1$ ) indicando dicha prioridad (un valor  $-1$  indica que no quiere o no puede estar con la persona de la columna correspondiente,  $0$  indica indiferencia y, cuanto más alto es, mayor es la preferencia por esa persona) . Ningún alumno puede formar grupo consigo mismo.

Se pretende encontrar una solución para satisfacer el número máximo de alumnos según su orden de preferencia. Suponiendo que la matriz  $M$  ya existe, diseñar un algoritmo que resuelva el problema de forma óptima

- Optima mediante bactracking
- Aproximada (sub-óptima) mediante voraz
- Ambas.

El tiempo de ejecución de un algoritmo de ramificación y poda depende de:

- La instancia del problema
- La función de selección de nodos para su expansión
- De ambos

Es un problema de optimización, si el dominio de las decisiones es un conjunto infinito

-   Podemos aplicar el esquema vuelta atrás siempre que se trate de un conjunto infinito numerable
-   Una estrategia voraz puede ser la única alternativa
- Es probable que a través de programación dinámica se obtenga un algoritmo eficaz que lo soluciones

Backtracking es aplicable a problemas de selección y optimización en los que:

- El espacio de soluciones es un conjunto finito.
- En cualquiera de los casos
- El espacio de soluciones es un conjunto infinito.

Bactracking es una técnicas de resolución general de problemas basada en:

- La búsqueda sistemática de soluciones.
- La construcción directa de la solución.
- Ninguna de las anteriores

Voraz siempre da solución óptima:



A los dos

Al problema de la mochila sin fraccionamiento.



Al problema de la mochila con fraccionamiento.

El problema de la mochila, ¿encuentra su solución óptima empleando la estrategia voraz?:

- Sólo para el caso de la mochila sin fraccionamiento
- En cualquiera de los casos anteriores
- Sólo para el caso de la mochila con fraccionamiento

El problema del viajante de comercio puede resolverse correctamente empleando estos esquemas de programación:

- Sólo programación dinámica
- Empleando cualquiera de estos: Voraz y backtracking.
- Solo backtracking

En el método Voraz, aunque las decisiones son irreversibles, podemos asegurar que:

- Siempre obtendremos una solución factible.
- Siempre obtendremos la solución óptima.
- Sólo obtendremos la solución óptima para algunos problemas.

Al aplicar backtracking obtenemos la solución óptima a un problema:

- ⚗ Siempre
- ⚡ Sólo cuando el problema cumple el principio de Optimidad.
- ⚡ En algunos casos

Dado un problema resuelto mediante backtracking y mediante ramificación y poda, el coste computacional de la solución por ramificación y poda, en comparación con la de backtracking es:

- Menor
- Igual
- Mayor

El problema de la asignación de turnos tiene solución óptima empleando:

### **EL PROBLEMA DE LA ASIGNACION DE TURNOS.**

Estamos al comienzo del curso y los alumnos quieren buscar compañero para formar un grupo de prácticas. Para solucionar este problema se propone que elijan a varias personas y les asignen una prioridad. El número de alumnos es  $N$ . Se dispone una matriz cuadrada  $M$  con  $N$  filas en la que cada alumno escribió, en su fila correspondiente, un número entero (entre  $-1$  y  $N-1$ ) indicando dicha prioridad (un valor  $-1$  indica que no quiere o no puede estar con la persona de la columna correspondiente,  $0$  indica indiferencia y, cuanto más alto es, mayor es la preferencia por esa persona) . Ningún alumno puede formar grupo consigo mismo.

Se pretende encontrar una solución para satisfacer el número máximo de alumnos según su orden de preferencia. Suponiendo que la matriz  $M$  ya existe, diseñar un algoritmo que resuelva el problema de forma óptima

- Bactracking
- Voraz
- Ambos

Al aplicar vuelta atrás a la solución de problemas, obtenemos algoritmos con costes computacionales:

- Exponenciales.
- Polinómicos.
- Los dos son correctos. Depende del problema

Vuelta atrás se emplea en la resolución de problemas de optimización en los que se pretende encontrar:

- La dos respuestas anteriores son correctas.
- Una solución que satisfaga unas restricciones y optimice una cierta función objetivo.
- Todas las soluciones que satisfagan unas restricciones.

El problema de la asignación de turnos tiene solución óptima voraz aplicando la siguiente estrategia:

### **EL PROBLEMA DE LA ASIGNACION DE TURNOS.**

Estamos al comienzo del curso y los alumnos quieren buscar compañero para formar un grupo de prácticas. Para solucionar este problema se propone que elijan a varias personas y les asignen una prioridad. El número de alumnos es  $N$ . Se dispone una matriz cuadrada  $M$  con  $N$  filas en la que cada alumno escribió, en su fila correspondiente, un número entero (entre  $-1$  y  $N-1$ ) indicando dicha prioridad (un valor  $-1$  indica que no quiere o no puede estar con la persona de la columna correspondiente,  $0$  indica indiferencia y, cuanto más alto es, mayor es la preferencia por esa persona) . Ningún alumno puede formar grupo consigo mismo.

Se pretende encontrar una solución para satisfacer el número máximo de alumnos según su orden de preferencia. Suponiendo que la matriz  $M$  ya existe, diseñar un algoritmo que resuelva el problema de forma óptima

- El problema no tiene solución óptima voraz
- Seleccionamos los alumnos en orden descendente de preferencia respetando las restricciones de cabida de cada turno.
- Seleccionamos los alumnos en orden ascendente de preferencia respetando las restricciones de cabida de cada turno

El problema de la mochila, ¿puede solucionarse empleando vuelta atrás?:

-   Sólo para el caso de la mochila con fraccionamiento
-   Sólo para el caso de la mochila sin fraccionamiento
- Se puede aplicar para ambos casos.

Dado un problema de optimización y un algoritmo Voraz que lo soluciona, ¿cuándo podemos estar seguros de que la solución obtenida será óptima?:

- Voraz siempre encuentra solución óptima.
- En ambos casos. Las dos son correctas
- Cuando demostremos formalmente que el criterio conduce a una solución óptima para cualquier instancia del problema.

El problema de la asignación de turnos resuelto mediante backtracking tiene una complejidad:

### **EL PROBLEMA DE LA ASIGNACION DE TURNOS.**

Estamos al comienzo del curso y los alumnos quieren buscar compañero para formar un grupo de prácticas. Para solucionar este problema se propone que elijan a varias personas y les asignen una prioridad. El número de alumnos es  $N$ . Se dispone una matriz cuadrada  $M$  con  $N$  filas en la que cada alumno escribió, en su fila correspondiente, un número entero (entre  $-1$  y  $N-1$ ) indicando dicha prioridad (un valor  $-1$  indica que no quiere o no puede estar con la persona de la columna correspondiente,  $0$  indica indiferencia y, cuanto más alto es, mayor es la preferencia por esa persona). Ningún alumno puede formar grupo consigo mismo.

Se pretende encontrar una solución para satisfacer el número máximo de alumnos según su orden de preferencia. Suponiendo que la matriz  $M$  ya existe, diseñar un algoritmo que resuelva el problema de forma óptima

- Exponencial
- Polinómica
- Ninguna de la dos

Bactracking procede a obtener la solución a un problema de optimización empleando la siguiente estrategia:

- Genera todas las combinaciones de la solución y selecciona la que optimiza la función objetivo.
- Genera todas las soluciones factibles y selecciona la que optimiza la función objetivo.
- Genera una solución factible empleando un criterio óptimo

Dado un grafo  $G$  que representa las poblaciones de la provincia de Alicante de más de 20.000 habitantes junto con todas las carreteras de conexión entre ellas. Queremos obtener el recorrido que nos permita pasar por todas estas ciudades una única vez y volver al punto de origen recorriendo el mínimo número de kilómetros. Si aplicamos una estrategia voraz sobre este grafo obtendremos...

- Puede que no encuentre ninguna solución aunque ésta exista
- Una solución factible
- La solución óptima

Si aplicamos un algoritmo voraz que no nos garantiza la solución óptima sobre un problema entonces...

- Si el problema tiene solución óptima, el esquema voraz nos garantiza que la encuentra
- Obtenremos una solución factible.
- Puede que no encuentre ninguna solución aunque ésta exista.

El método voraz se emplea en la resolución de problemas de selección y optimización en los que se pretende encontrar:

- La dos respuestas anteriores son correctas.
- Todas las soluciones que satisfagan unas restricciones.
- Una solución que satisfaga unas restricciones y optimice una cierta función objetivo.

Si aplicamos un esquema backtracking que no nos garantiza la solución óptima sobre un problema entonces

-   Obtenremos una solución factible.
- Puede que no encuentre ninguna solución aunque ésta exista.
-   Ninguna de las anteriores.

La estrategia de ramificación y poda genera las soluciones posibles al problema mediante...

- un recorrido en profundidad del árbol que representa el espacio de soluciones
- un recorrido en anchura que representa el espacio de soluciones
- un recorrido guiado por estimaciones de las mejores ramas del árbol que representa el espacio de soluciones

Dada la solución recursiva mediante vuelta atrás al problema de la asignación de turnos ¿cuántas nuevas llamadas recursivas genera cada llamada recursiva?

### EL PROBLEMA DE LA ASIGNACION DE TURNOS.

Estamos al comienzo del curso y los alumnos quieren buscar compañero para formar un grupo de prácticas. Para solucionar este problema se propone que elijan a varias personas y les asignen una prioridad. El número de alumnos es  $N$ . Se dispone una matriz cuadrada  $M$  con  $N$  filas en la que cada alumno escribió, en su fila correspondiente, un número entero (entre  $-1$  y  $N-1$ ) indicando dicha prioridad (un valor  $-1$  indica que no quiere o no puede estar con la persona de la columna correspondiente,  $0$  indica indiferencia y, cuanto más alto es, mayor es la preferencia por esa persona) . Ningún alumno puede formar grupo consigo mismo.

Se pretende encontrar una solución para satisfacer el número máximo de alumnos según su orden de preferencia. Suponiendo que la matriz  $M$  ya existe, diseñar un algoritmo que resuelva el problema de forma óptima

- una o dos
- una o ninguna
- ✓ ninguna de las anteriores

¿Cuál de estas afirmaciones es falsa?

- La complejidad en el peor caso de las soluciones Backtracking y poda a un mismo problema es la misma.
- Para un mismo problema, ramificación y poda explora siempre un número menor o igual que backtracking
- Backtracking inspecciona todo el espacio de soluciones de un problema mientras que Ramificación y poda no.

Si para resolver un mismo problema usamos un algoritmo de vuelta atrás y lo modificamos mínimamente para convertirlo en un algoritmo de ramificación y poda, ¿qué cambiamos realmente?

- Cambiamos la función que damos a la cota pesimista
- La comprobación de las soluciones factibles: en ramificación y poda no es necesario puesto que sólo genera nodos factibles
- Aprovechamos mejor las cotas optimistas

Bactracking genera las soluciones posibles al problema:

- Ⓛ Mediante el recorrido en profundidad del árbol que representa el espacio de soluciones.
- Ⓜ Mediante el recorrido en anchura del árbol que representa el espacio de soluciones
- Ⓝ Ninguna de las anteriores

En un problema resuelto por backtracking, el conjunto de valores que pueden tomar las componentes de la tupla solución, ha de ser:

- Infinito.
- continuo
- finito

Cuando la descomposición recursiva de un problema da lugar a subproblemas de tamaño similar, ¿qué esquema promete ser más apropiado?

- Programación dinámica.
- Divide y vencerás, siempre que se garantice que los subproblemas no son del mismo tamaño.
- El método voraz

## El uso de funciones de cota en ramificación y poda...



- puede reducir el número de instancias del problema que pertenecen al caso peor.
- garantiza que el algoritmo va a ser más eficiente ante cualquier instancia del problema.
- transforma en polinómicas complejidades que antes eran exponenciales.

Sí un problema de optimización lo es para una función que toma valores continuos ...

- La programación dinámica iterativa siempre es mucho más eficiente que la programación dinámica iterativa en cuanto al uso de memoria.
- El uso de memoria de la programación dinámica iterativa y de la programación dinámica recursiva es el mismo independientemente de si el dominio es discreto o continuo.
- La programación dinámica recursiva puede resultar mucho más eficiente que la programación dinámica iterativa en cuanto al uso de memoria.

Si  $f(n) \in O(n^3)$ , ¿puede pasar que  $f(n) \in O(n^2)$ ?

- No, porque  $n^3$  no  $\in O(n^2)$
- Es perfectamente posible, ya que  $O(n^2) \subset O(n^3)$
- Sólo para valores bajos de  $n$

Sea la siguiente relación de recurrencia

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$$

Si  $T(n) \in O(n^2)$ , ¿en cuál de estos tres casos nos podemos encontrar?

- $\bullet$   $g(n) = 1$
- $\circ$   $g(n) = n^2$
- $\circ$   $g(n) = n$

La solución recursiva ingenua (pero correcta) a un problema de optimización llama más de una vez a la función con los mismos parámetros. Una de las siguientes tres afirmaciones es falsa.

-  Se puede mejorar la eficiencia del algoritmo guardando en una tabla el valor devuelto para cada conjunto de parámetros de cada llamada cuando ésta se produce por primera vez.
-  Se puede mejorar la eficiencia del algoritmo convirtiendo el algoritmo recursivo directamente en iterativo sin cambiar su funcionamiento básico.
-  Se puede mejorar la eficiencia del algoritmo definiendo de antemano el orden en el que se deben calcular las soluciones a los subproblemas y llenando una tabla en ese orden.

Un Problema de tamaño  $n$  puede transformarse en tiempo  $O(n^2)$  en otro de tamaño  $n - 1$ . Por otro lado, la solución al problema cuando la talla es 1 requiere un tiempo constante. ¿cuál de estas clases de coste temporal asintótico es la más ajustada?

- $O(n^3)$
- $O(2^n)$
- $O(n^2)$

Cuando se usa un algoritmo voraz para abordar la resolución de un problema de optimización por selección discreta (es decir, un problema para el cual la solución consiste en encontrar un subconjunto del conjunto de elementos que optimiza una determinada función), ¿cuál de estas tres cosas es imposible que ocurra?



Que la solución no sea la óptima.

Que el algoritmo no encuentre ninguna solución.

Que se reconsidere la decisión ya tomada anteriormente respecto a la



selección de un elemento anterior respecto a la selección de un elemento a la vista de la decisión que se debe tomar en el instante actual.

Uno de estos tres problemas no tiene una solución eficiente que siga el esquema de programación dinámica

- El Problema de la mochila discreta.
- El Problema de las torres de Hanoi
- El problema de cortar un tubo de longitud  $n$  en segmentos de longitud entera entre 1 y  $n$  de manera que se maximice el precio de acuerdo con una tabla que da el precio para cada longitud.

¿Cuál de estos tres problemas de optimización no tiene, o no se le conoce, una solución voraz (greedy) que es óptima?

- El problema de la mochila discreta.
- El árbol de cobertura de coste mínimo de un grafo conexo.
- El problema de la mochila continua o con fraccionamiento.

Cuál de los siguientes algoritmos proveería una cota pesimista para el problema de encontrar el camino mas corto entre dos ciudades (se supone que el grafo es conexo).

- Calcular la distancia geométrica (en línea recta) entre la ciudad origen y destino.
- Para todas las ciudades que son alcanzables en un Paso desde la ciudad inicial, sumar la distancia a dicha ciudad y la distancia geometrica hasta la ciudad destino.
- Calcular la distancia recorrida moviéndose al azar por el grafo hasta llegar(por azar) a la ciudad destino.

Dado un problema de optimización cualquiera, ¿la estrategia de vuelta atrás garantiza la solución óptima?

- Sí, siempre que el dominio de las decisiones sea discreto o discretizable y además se empleen mecanismos de poda basados en la mejor solución hasta el momento.
- Sí, puesto que ese método analiza todas las posibilidades.
- Es condición necesaria que el dominio de las decisiones sea discreto o discretizable y que el número de decisiones a tomar esté acotado.

¿Para cuál de estos problemas de optimización existe una solución voraz?

- El árbol de recubrimiento mínimo para un grafo no dirigido con pesos.
- El problema de la mochila discreta.
- El problema de la asignación de coste mínimo de  $n$  tareas a  $n$  trabajadores cuando el coste de asignar la tarea  $i$  al trabajador  $j$ ,  $C_{ij}$  está tabulado en una matriz.

La complejidad temporal en el mejor de los casos...

- es el tiempo que tarda el algoritmo en resolver el problema de tamaño o talla más pequeña que se le puede presentar.
- es una función del tamaño o talla del problema que tiene que estar definida para todos los posibles valores de ésta.
- las otras dos opciones son ciertas.

Garantiza el uso de una estrategia "divide y vencerás" la existencia de una solución de complejidad temporal polinómica a cualquier problema?

- Sí, pero siempre que la complejidad temporal conjunta de las operaciones de descomposición del problema y la combinación de las soluciones sea polinómica.
- No
- Sí, en cualquier caso.

Decid cuál de estas tres es la cota pesimista más ajustada al valor óptimo de la mochila discreta:

- El valor de la mochila discreta que se obtiene usando un algoritmo voraz basado en el valor específico de los objetos
- El valor de una mochila que contiene todos los objetos aunque se pase del peso máximo permitido.
- El valor de la mochila continua correspondiente

Al resolver el problema del viajante de comercio mediante vuelta atrás, ¿cuál de estas cotas optimistas se espera que padezca mejor el árbol de búsqueda?

- Se resuelve el resto del problema usando un algoritmo voraz que añade cada vez al camino el vértice más cercano al último añadido.
- Se ordenan las aristas restantes de menor a mayor distancia y se calcula la suma de las  $k$  aristas más cortas, donde  $k$  es el número de saltos que nos quedan por dar.
- Se multiplica  $k$  por la distancia de la arista más corta que nos queda por considerar, donde  $k$  es el número de saltos que nos quedan por dar.

Se desea encontrar el camino mas corto entre dos ciudades. Para ello se dispone de una tabla con la distancia entre los pares de ciudades en los que hay carreteras o un valor centinela [por ejemplo, -1] si no hay, por lo que para ir de la ciudad inicial a la final es Possible que haya que pasar por varias ciudades. También se conocen las coordenadas geográficas de cada ciudad y por tanto la distancia geométrica (en línea recta) entre cada par de ciudades. Se pretende acelerar la búsqueda de un algoritmo de ramificación y poda priorizando los nodos vivos (ciudades) que estén a menor distancia geográfica de la ciudad objetivo.

- El nuevo algoritmo no garantiza que vaya a ser más rápido para todas las instancias del problema posibles.
- Esta estrategia no asegura que se obtenga el camino mas corto.
- El nuevo algoritmo siempre sera más rápido.

## En los algoritmos de ramificación y poda

- Una cota optimista es necesariamente un valor insuperable, de no ser así se podría podar el nodo que conduce a la solución óptima.
- Una cota optimista es necesariamente un valor alcanzable, de no ser así no está garantizado que se encuentre la solución óptima.
- Una cota pesimista es el valor que a lo sumo alcanza cualquier nodo factible que no es el óptimo.

## La complejidad en el mejor de los casos de un algoritmo de ramificación y poda

...

- es siempre exponencial con el número de decisiones a tomar.
- suele ser polinómica con el número de alternativas por cada decisión.
- puede ser polinómica con el número de decisiones a tomar.

El valor que se obtiene con el método voraz para el problema de la mochila discreta es ...

- ... una cota superior para el valor óptimo.
- ... una cota inferior Para el valor óptimo, pero que nunca coincide con este.
- ... una cota inferior Para el valor óptimo que a veces puede ser igual a este.

En el esquema de vuelta atrás el orden en el que se van asignando los distintos valores a las componentes del vector que contendrá la solución...

- Las otras dos opciones son ciertas.
- puede ser relevante si se utilizan mecanismos de poda basados en estimaciones optimistas.
- es irrelevante si no se utilizan mecanismos de poda basados en la mejor solución hasta el momento.

La mejor solución que se conoce para el problema de la mochila continua sigue el esquema

- ...divide y vencerás.
- ...voraz.
- ...ramificación y poda.

Sea A una matriz cuadrada  $n \times n$ . Se trata de buscar una permutación de las columnas tal que la suma de los elementos de la diagonal de la matriz resultante sea mínima. Indicad cuál de las siguientes afirmaciones es falsa.

-  Si se construye una solución al problema basada en el esquema de ramificación y poda, una buena elección de cotas optimistas y pesimistas podría evitar la exploración de todas las permutaciones posibles.
-  La complejidad temporal de la mejor solución posible al problema es  $O(n^2)$ .
-  La complejidad temporal de la mejor solución posible al problema es  $O(n!)$ .

En una cuadrícula se quiere dibujar el contorno de un cuadrado de  $n$  casillas de lado. ¿cuál será la complejidad temporal del mejor algoritmo que pueda existir?

- $O(\sqrt{n})$
- $O(n^2)$
- $O(n)$

Dado un problema de optimización, el método voraz...

-   siempre obtiene la solución óptima.
- siempre obtiene una solución factible.
-   garantiza la solución óptima sólo para determinados problemas.

Di cuál de estos tres algoritmos no es un algoritmo de "divide y vencerás"

  Quicksort

Mergesort

  El algoritmo de Prim

Cuando se resuelve el problema de La mochila discreta usando la estrategia de vuelta atrás, ¿puede ocurrir que se tarde menos en encontrar la solución óptima si se prueba primero a meter cada objeto antes de no meterlo?

- No, ya que en cualquier caso se deben explorar todas las soluciones factibles.
- Sí, pero sólo si se usan cotas optimistas para podar el árbol de búsqueda.
- Sí, tanto si se usan cotas optimistas para podar el árbol de búsqueda como si no.

Dadas las siguientes funciones: (imagen)

Se quiere reducir la complejidad temporal de la función g usando programación dinámica iterativa. ¿cuál sería la complejidad espacial?

```
// Precondición: { 0 <= i < v.size(); i < j <= v.size() }
unsigned f( const vector<unsigned>&v, unsigned i, unsigned j ) {
    if( i == j+1 )
        return v[i];
    unsigned sum = 0;
    for( unsigned k = 0; k < j - i; k++ )
        sum += f( v, i, i+k+1) + f( v, i+k+1, j );
    return sum;
}

unsigned g( const vector<unsigned>&v ) {
    return f( v, v.begin(), v.end() );
}
```



cuadrática

exponencial

cúbica

En un problema de optimización, si el dominio de las decisiones es un conjunto infinito,

- es probable que a través de programación dinámica se obtenga un algoritmo eficaz que lo solucione.
- una estrategia voraz puede ser la única alternativa.
- podremos aplicar el esquema vuelta atrás siempre que se trate de un conjunto infinito numerable.

Si para resolver un mismo problema usamos un algoritmo de vuelta atrás y lo modificamos mínimamente para convertirlo en un algoritmo de ramificación y poda, ¿qué cambiamos realmente?

- El algoritmo puede aprovechar mejor las cotas optimistas.
- Cambiamos la función que damos a la cota pesimista.
- La comprobación de las soluciones factibles: en ramificación y poda no es necesario puesto que sólo genera nodos factibles.

¿Cuál de estos problemas tiene una solución eficiente utilizando programación dinámica?

-   La mochila discreta sin restricciones adicionales.
-   El problema del cambio.
- El problema de la asignación de tareas.

El siguiente programa resuelve el problema de cortar un tubo de longitud  $n$  en segmentos de longitud entera entre 1 y  $n$  de manera que se maximice el precio de acuerdo con una tabla que da el precio para cada longitud, pero falta un trozo. ¿Qué debería ir en lugar de XXXXXXXX?

- p,r,n-r[n]
- p,r-1,n
- p,r,n-i

```
void fill(price r[]) {  
    for (index i=0;i<=n;i++) r[i]=-1;  
}  
  
price cutrod(price p[], price r[], length n) {  
    price q;  
    if (r[n]>=0) return r[n];  
    if (n==0) q=0;  
    else {  
        q=-1;  
        for (index i=1;i<=n;i++)  
            q=max (q,p[i]+cutrod(r,p,i-1));  
    }  
    r[n]=q;  
    return q;  
}
```

En los algoritmos de ramificación y poda, ¿el valor de una cota pesimista es mayor que el valor de una cota optimista? (se entiende que ambas cotas se aplican sobre el mismo nodo)



No, nunca es así.

En general si, si se trata de un problema de maximización, aunque en ocasiones ambos valores pueden coincidir.



En general si, si se trata de un problema de minimización, aunque en ocasiones ambos valores pueden coincidir.

Una de estas tres situaciones no es posible:

- a  $f(n) \in O(n)$  y  $f(n) \in \Omega(1)$
- b  $f(n) \in \Omega(n^2)$  y  $f(n) \in O(n)$
- c  $f(n) \in O(n)$  y  $f(n) \in O(n^2)$

  a

  b

c

La mejora que en general aporta la programación dinámica frente a la solución ingenua se consigue gracias al hecho de que...

- en la solución ingenua se resuelve pocas veces un número relativamente grande de subproblemas distintos.
- El número de veces que se resuelven los subproblemas no tiene nada que ver con la eficiencia de los problemas resueltos mediante programación dinámica.
- en la solución ingenua se resuelve muchas veces un número relativamente pequeño de subproblemas distintos.

La versión de Quicksort que utiliza como pivote el elemento del vector que ocupa la primera posición ...

- se comporta mejor cuando el vector ya está ordenado.
- no presenta caso mejor y peor para instancias del mismo tamaño.
- se comporta peor cuando el vector ya está ordenado.

En el esquema de vuelta atrás, los mecanismos de poda basados en la mejor solución hasta el momento...

- Las otras dos opciones son ciertas.
- garantizan que no se va a explorar nunca todo el espacio de soluciones posibles.
- pueden eliminar soluciones parciales que son factibles.

Un algoritmo recursivo basado en el esquema divide y vencerás ...

-   ...nunca tendrá una complejidad exponencial.
- Las dos anteriores son ciertas.
-   ...será más eficiente cuanto más equitativa sea la división en subproblemas.

Se quieren ordenar  $d$  números distintos comprendidos entre 1 y  $n$ . Para ello se usa un array de  $n$  booleanos que se inicializan primero a false. A continuación se recorren los  $d$  números cambiando los valores del elemento del vector de booleanos correspondiente a su número a true. Por último se recorre el vector de booleanos escribiendo los índices de los elementos del vector de booleanos que son true. ¿Es este algoritmo más rápido (asintóticamente) que el mergesort?

- Sí, ya que el mergesort es  $O(n \log n)$  y este es  $O(n)$
- No, ya que este algoritmo ha de recorrer varias veces el vector de booleanos.
- Sólo si  $d \log d > kn$  (donde  $k$  es una constante que depende de la implementación)

¿En ramificación y poda, tiene sentido utilizar la cota optimista de los nodos como criterio para ordenar la lista de nodos vivos?

- No, la cota optimista sólo se utiliza para determinar si una n-tupla es prometedora.
- Sí, aunque no es una garantía de que sea una buena estrategia de búsqueda.
- Sí, en el caso de que se ordene la lista de nodos vivos, siempre debe hacerse según el criterio de la cota optimista.

En el esquema de vuelta atrás, los mecanismos de poda basados en la mejor solución hasta el momento...

- Pueden eliminar vectores que representan posibles soluciones factibles.
- Garantizan que no se va a explorar todo el espacio de soluciones posibles.
- Las otras dos opciones son ambas verdaderas...

Se desea obtener todas las permutaciones de una lista compuesta por  $n$  elementos. ¿Qué esquema es el más adecuado?

- Divide y vencerás, puesto que la división en sublistas se podría hacer en tiempo constante.
- Vuelta atrás, es el esquema más eficiente para este problema.
- Ramificación y poda, puesto que con buenas funciones de cota es más eficiente que vuelta atrás.

En ausencia de cotas optimistas y pesimistas, la estrategia de vuelta atrás...

- ... no recorre todo el árbol si hay manera de descartar subárboles que representan conjuntos de soluciones no factibles.
- ... debe recorrer siempre todo el árbol.
- ... no se puede usar para resolver problemas de optimización.

## El esquema voraz...

-   Garantiza encontrar una solución a cualquier problema, aunque puede que no sea óptima.
- Puede que no encuentre una solución pero si lo hace se garantiza que es óptima.
-   Las otras dos opciones son ambas falsas.

Un algoritmo recursivo basado en el esquema divide y vencerás...

- Las otras dos opciones son ambas verdaderas.
- ... alcanza su máxima eficiencia cuando el problema de tamaño  $n$  se divide en  $a$  problemas de tamaño  $n/a$ .
- ... nunca tendrá un coste temporal asintótico (o complejidad temporal) exponencial.

Cuando la descomposición de un problema de lugar a subproblemas de tamaño similar al original, muchos de los cuales se repiten, ¿qué esquema es a priori más apropiado?

- Programación dinámica
- Divide y vencerás.
- Ramificación y poda.

Dado el problema del laberinto con tres movimientos, ¿se puede aplicar un esquema de programación dinámica para obtener un camino de salida?

- No, para garantizar que se encuentra un camino de salida hay que aplicar métodos de búsqueda exhaustiva como vuelta atrás o ramificación y poda.
- Si, en caso de existir con este esquema siempre se puede encontrar un camino de salida
- No, con este esquema se puede conocer el número total de caminos distintos que conducen a la salida pero no se puede saber la composición de ninguno de ellos.

Se desea resolver el problema de la potencia enésima ( $x^n$ ), asumiendo que  $n$  es par y que se utilizará la siguiente recurrencia:

$\text{pot.}(x, n) = \text{pot.}(x, n/2) * \text{pot.}(x, n/2)$ ; ¿Qué esquema resulta ser más eficiente en cuanto al coste temporal?

- En este caso tanto programación dinámica como divide y vencerás resultan ser equivalentes en cuanto a la complejidad temporal.
- Programación dinámica.
- Divide y vencerás

Decid cuál de estas tres es la cota optimista más ajustada al valor óptimo de la mochila discreta:

- El valor de una mochila que contiene todas los objetos aunque se pase del peso máximo permitido.
- El valor de la mochila continua correspondiente.
- El valor de la mochila discreta que se obtiene usando un algoritmo voraz basado en el valor específico de los objetos.

Una de las prácticas de laboratorio consistió en el cálculo empírico de la complejidad temporal promedio del algoritmo de ordenación de vectores Quicksort tomando como centinela el elemento del vector que ocupa la posición central. ¿Cuál es el orden de complejidad que se obtuvo?

   $n^2$

   $n \log n$

$n \log^2 n$

¿Qué se deduce de  $f(n)$  y  $g(n)$  si se cumple  $\lim_{n \rightarrow \infty} (f(n)/g(n)) = k$ , con  $k \neq 0$ ?

- $\bullet$   $g(n) \in O(f(n))$  pero  $f(n) \notin O(g(n))$
- $\circ$   $f(n) \in O(g(n))$  y  $g(n) \in O(f(n))$
- $\circ$   $f(n) \in O(g(n))$  pero  $g(n) \notin O(f(n))$

Dado el problema del laberinto con tres movimientos, se desea saber el número de caminos distintos desde la casilla inicial  $(1, 1)$  hasta la casilla  $(n, m)$  y para ello se aplica un esquema de divide y vencerás. ¿Cuál sería la recurrencia apropiada para el caso general?

- $nc(n, m) = nc(n - 1, m) + nc(n, m - 1) + nc(n - 1, m - 1)$
- $nc(n, m) = nc(n - 1, m) * nc(n, m - 1) * nc(n - 1, m - 1)$
- Ninguna de las otras dos recurrencias se corresponde con un esquema de divide y vencerás.

Sea la siguiente relación de recurrencia: (imagen)

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$$

Si  $T(n) \in O(n)$ , ¿en cuál de estos tres casos nos podemos encontrar?

- $g(n) = \sqrt{n}$
- $g(n) = \log n$
- Las otras dos opciones son ambas ciertas

## El esquema de vuelta atrás...



- Garantiza que encuentra la solución Óptima & cualquier problema de selección discreta.
- Las otras dos opciones son ambas verdaderas.
- Se puede aplicar a cualquier tipo de problema aunque el coste temporal es elevado.

¿Qué ocurre si la cota pesimista de un nodo se corresponde con una solución que no es factible?

- Que el algoritmo sería más lento pues se explorarían más nodos de los necesarios.
- Nada especial, las cotas pesimistas no tienen por qué corresponderse con soluciones factibles.
- Que el algoritmo sería incorrecto pues podría descartarse un nodo que conduce a la solución óptima.

Dado el problema del laberinto con tres movimientos, se desea saber el número de caminos distintos desde la casilla inicial  $(1, 1)$  hasta la casilla  $(n, m)$  y para ello se aplica un esquema de programación dinámica. En cuanto a la complejidad temporal, ¿cuál es la mejora de la versión recursiva con memoización frente a la recursiva ingenua que se obtiene a partir del esquema divide y vencerás?

-  La mejora no está garantizada puesto que la versión recursiva con memoización podría ser peor que la obtenida a partir del esquema divide y vencerás.
-  De una complejidad cuadrática que se obtendría con la ingenua se reduciría a lineal con la de memoización.
-  De una complejidad exponencial que se obtendría con la ingenua se reduciría a polinómica con la de memoización.

Dado el problema del laberinto con tres movimientos, se pretende conocer la longitud del camino de salida más corto. Para ello se aplica el esquema voraz con un criterio de selección que consiste en elegir primero el movimiento "Este" siempre que la casilla sea accesible. Si no lo es se descarta ese movimiento y se prueba con "Sureste" y por último, si este tampoco es posible, se escoge el movimiento "Sur". ¿Qué se puede decir del algoritmo obtenido?

- Que en realidad no es un algoritmo voraz pues el criterio de selección no lo es.
- Que es un algoritmo voraz pero sin garantía de solucionar el problema...
- Que en realidad no es un algoritmo voraz pues las decisiones que se toman no deberían reconsiderarse.

(imagen)

Dada la siguiente función:

```
int exa (vector <int>& v){  
    int j, i=1, n=v.size();  
  
    if (n>1) do{  
        int x = v[i];  
        for (j=i; j >0 and v[j-1] >x; j--)  
            v[j]=v[j-1];  
        v[j]=x;  
        i++;  
    } while (i<n);  
    return 0;  
}
```

- La complejidad temporal en el mejor de los casos es  $\Omega(n)$
- La complejidad temporal en el mejor de los casos es  $\Omega(1)$
- La complejidad temporal exacta es  $\Theta(n^2)$

¿Qué estrategia de búsqueda es a priori más apropiada en un esquema de vuelta atrás?

- Explorar primero los nodos con mejor cota optimista.
- Explorar primera los nodos que están más completados.
- En el esquema de vuelta atrás no se pueden definir estrategias de búsqueda.

Dado el problema de las torres de Hanoi resuelto mediante divide y vencerás, ¿cuál de las siguientes relaciones de recurrencia expresa mejor su complejidad temporal para el caso general, siendo  $n$  el número de discos?

- $\bullet \quad T(n) = T(n - 1) + n$
- $\circ \quad T(n) = 2T(n - 1) + 1$
- $\circ \quad T(n) = 2T(n - 1) + n$

Si el coste temporal de un algoritmo es  $T(n)$ , ¿cuál de las siguientes situaciones es imposible?

- $\bullet$   $T(n) \in O(n)$  y  $T(n) \in \Theta(n)$
- $\bullet$   $T(n) \in \Theta(n)$  y  $T(n) \in \Omega(n^2)$
- $\bullet$   $T(n) \in \Omega(n)$  y  $T(n) \in \Theta(n^2)$

Un tubo de  $n$  centímetros de largo se puede cortar en segmentos de 1 centímetro, 2 centímetros, etc... Existe una lista de los precios a los que se venden los segmentos de cada longitud. Una de las maneras de cortar el tubo es la que más ingresos nos producirá. Se quiere resolver el problema mediante vuelta atrás. ¿Cuál sería la forma más adecuada de representar las posibles soluciones?

- Un vector de booleanos.
- Una tabla que indique, para cada posición donde se va a cortar, cada uno de los posibles valores acumulados.
- Un par de enteros que indiquen los cortes realizados y el valor acumulada.

¿Cuál sería la complejidad temporal de la siguiente función tras aplicar programación dinámica?

```
double f(int n, int m) {  
    if(n == 0) return 1;  
    return m * f(n-1,m) * f(n-2,m);  
}
```

- $\Theta(n^2)$
- $\Theta(n \times m)$
- $\Theta(n)$

Dada la siguiente función:

```
int exa (vector <int>& v) {  
    int i,sum=0, n=v.size();  
  
    if (n>0){  
        int j=n;  
        while (sum<100){  
            j=j/2;  
            sum=0;  
            for (i=j;i<n;i++)  
                sum+=v[i];  
            if (j==0) sum=100;  
        }  
        return j;  
    }  
    else return -1;  
}
```

Marcad la opción correcta.

(imagen)

- La complejidad temporal en el mejor de los casos es  $\Omega(n)$
- La complejidad temporal en el mejor de los casos es  $\Omega(1)$
- La complejidad temporal exacta es  $\Theta(n \log n)$

El coste temporal asintótico de insertar un elemento en un vector ordenado de forma que continue ordenado es:

- $O(\log n)$
- $\Omega(n^2)$
- $O(n)$

Dado un problema de maximización resuelto mediante un esquema de ramificación y poda, ¿qué ocurre si la cota optimista resulta ser un valor excesivamente elevado?

- Que se podría explorar menos nodos de los necesarios.
- Que se podría podar el nodo que conduce a la solución óptima.
- Que se podría explorar más nodos de los necesarios.

Si  $f \in \Omega(g_1)$  y  $f \in \Omega(g_2)$  entonces:

-    $f \in \Omega(g_1 * g_2)$
- $f \in \Omega(\min(g_1, g_2))$
-    $f \in \Omega(g_1 + g_2)$

¿Qué complejidad se obtiene a partir de la relación de recurrencia  $T(n) = 8T(n/2) + n^3$  con  $T(1) = O(1)$ ?

- $O(n \log n)$
- $O(n^3)$
- $O(n^3 \log n)$

En el problema del viajante de comercio (travelling salesman problem) queremos listar todas las soluciones factibles.

Lo más importante es conseguir una cota pesimista adecuada.

- Las diferencias entre ramificación y poda y vuelta atrás son irrelevantes en este caso.

El orden en el que se exploran las soluciones parciales no es relevante;

- por ello, la técnica ramificación y poda no aporta nada con respecto a vuelta atrás.

La más adecuado sería usar una técnica de ramificación y poda ya que

- es muy importante el orden en el que se exploran las soluciones parciales.

(imagen)

Dada la siguiente función (donde  $\max(a, b) \in \Theta(1)$ ):

```
float exa(vector<float>&v, vector<int>&p, int P, int i)
{
    float a, b;
    if (i >= 0){
        if (p[i] <= P)
            a = v[i] + exa(v, p, P - p[i], i - 1);
        else a = 0;
        b = exa(v, p, P, i - 1);
        return max(a, b);
    }
    return 0;
}
```

- X  La complejidad temporal en el mejor de los casos es  $\Omega(n^2)$
- La complejidad temporal en el peor de los casos es  $O(n^2)$
- ✓  La complejidad temporal en el peor de los casos es  $O(2^n)$

En el esquema de ramificación y poda, ¿qué estructura es la más adecuada si queremos realizar una exploración por niveles?

- Cola de prioridad
- Cola
- Pila

(imagen)



O(n)

O(log n)

O( $n^2$ )

Dada la siguiente función:

```
int exa (string & cad, int pri, int ult){  
    if (pri>=ult)  
        return 1;  
    else  
        if (cad[pri]==cad[ult])  
            return exa(cad, pri+1, ult-1);  
        else  
            return 0;  
}
```

¿Cuál es su complejidad temporal asintótica?

Dado el problema del laberinto con tres movimientos, ¿cuál de las estrategias siguientes proveería de una cota optimista para ramificación y poda?

- Suponer que en adelante todas las casillas del laberinto son accesibles
- Suponer que ya no se van a realizar más movimientos.
- Las otras dos estrategias son ambas válidas

De las siguientes expresiones, o bien dos son verdaderas y una es falsa, o bien dos son falsas y una es verdadera. Marca la que (en este sentido) es distinta a las otras dos.

- ⚡  $n + n \log n \in \Omega(n)$
- ⚡  $O(2^{(\log n)}) \subset O(n^2)$
- ⚡  $\Theta(n) \subset \Theta(n^2)$

¿Qué nos proporciona la media entre el coste temporal asintótico (o complejidad temporal) en el peor caso y el coste temporal asintótico en el mejor caso?

- El coste temporal promedio
- El coste temporal asintótico en el caso medio
- Nada de interés

Se desea ordenar una lista enlazada de  $n$  elementos haciendo uso del algoritmo Mergesort. En este caso, al tratarse de una lista, la complejidad temporal asintótica de realizar la división en subproblemas resulta ser lineal con el tamaño de esa lista. ¿Cuál sería entonces el coste temporal de realizar dicha ordenación?

- $\Theta(n^2)$
- $\Theta(n \log n)$
- Ninguna de las otras dos opciones es cierta

Dado el problema del laberinto con tres movimientos, se desea saber el número de caminos distintos desde la casilla inicial  $(1, 1)$  hasta la casilla  $(n, m)$  y para ello se aplica el esquema programación dinámica para obtener un algoritmo lo más eficiente posible en cuanto a complejidad temporal y espacial. ¿Cuáles serían ambas complejidades?

- Temporal  $\Theta(n \times m)$  y espacial  $\Theta(n \times m)$
- Temporal  $\Theta(\max\{n, m\})$  y espacial  $\Theta(\max\{n, m\})$
- Temporal  $\Theta(n \times m)$  y espacial  $\Theta(\min\{n, m\})$

De las siguientes afirmaciones marca la que es verdadera.

- Las cotas pesimistas no son compatibles con un esquema de vuelta atrás.
- En un esquema de vuelta atrás, las cotas pesimistas no tienen sentido si lo que se pretende es obtener todas las soluciones factibles.
- El esquema de vuelta atrás no es compatible con el uso conjunto de cotas pesimistas y optimistas.

Dado un problema de minimización resuelto mediante un esquema de ramificación y poda, ¿qué propiedad cumple una cota optimista?

- Siempre es mayor o igual que la mejor solución pasible alcanzada.
- Asegura un ahorro en la comprobación de todas las soluciones factibles.
- Las otras dos opciones son ambas falsas.

Dada la siguiente función (imagen)  
marca la respuesta correcta:

```
int exa (vector <int>& v) {  
    int i, sum=0, n=v.size();  
  
    if (n>0){  
        int j=n;  
        while (sum<100){  
            j=j/2;  
            sum=0;  
            for (i=j;i<n;i++)  
                sum+=v[i];  
            if (j==0) sum=100;  
        }  
        return j;  
    }  
    else return -1;  
}
```

- La complejidad temporal exacta es  $\Theta(n \log n)$
- La complejidad temporal en el mejor de los casos es  $\Omega(n)$
- La complejidad temporal en el mejor de los casos es  $\Omega(1)$

¿Se puede reducir el coste temporal de un algoritmo recursivo almacenando los resultados devueltos por las llamadas recursivas?

- No, sólo se puede reducir el coste convirtiendo el algoritmo recursivo en iterativo
- Sí, si se repiten llamadas a la función con los mismos argumentos
- No, ello no reduce el coste temporal ya que las llamadas recursivas se deben realizar de cualquier manera

De las siguientes expresiones, o bien dos son verdaderas y una es falsa, o bien dos son falsas y una es verdadera. Marca la que (en este sentido) es distinta a las otras dos.

- a  $\Theta(f) = O(f) \cap \Omega(f)$
- b  $\Omega(f) = \Theta(f) \cap O(f)$
- c  $O(f) = \Omega(f) \cap \Theta(f)$



a

b

c

El algoritmo Quicksort divide el problema en dos subproblemas. ¿Cuál es la complejidad temporal asintótica de realizar esa división?

- $\Omega(n)$  y  $O(n^2)$
- $O(n)$
- $O(n \log n)$

La complejidad temporal de la solución de vuelta atrás al problema de la mochila discreta es...

- exponencial en el caso peor
- cuadrática en el caso peor
- exponencial en cualquier caso

En el problema del coloreado de grafos (mínimo número de colores necesarios para colorear todos los vértices de un grafo de manera que no queden dos adyacentes con el mismo color) resuelto mediante ramificación y poda, una cota optimista es el resultado de asumir que...

- se van a utilizar tantos colores distintos a los ya utilizados como vértices quedan por colorear
- solo va a ser necesario un color más
- no se van a utilizar colores distintos a los ya utilizados

La versión Quicksort que utiliza como pivote el elemento del vector que ocupa la posición central...

- no presenta caso mejor y peor para instancias del mismo tamaño
- se comporta peor cuando el vector ya está ordenado
- se comporta mejor cuando el vector ya está ordenado

Si  $\lim_{n \rightarrow \infty} (f(n)/n^2) = k$ , y  $k \neq 0$ , ¿cuál de estas tres afirmaciones es falsa?

- f(n) ∈ O(n<sup>3</sup>)
- f(n) ∈ Θ(n<sup>2</sup>)
- f(n) ∈ Θ(n<sup>3</sup>)

Tenemos un conjunto de  $n$  enteros positivos y queremos encontrar el subconjunto de tamaño  $m$  de suma mínima...

- ✓  Una técnica voraz daría una solución óptima.

Lo más adecuado sería usar una técnica de ramificación y poda,  
 aunque en el peor caso el coste temporal asintótico (o complejidad temporal) sería exponencial.

Para encontrar la solución habría que probar con todas las  
 combinaciones posibles de  $m$  enteros, con lo que la técnica de ramificación y poda no aporta nada con respecto a vuelta atrás.

¿Qué algoritmo es asintóticamente más rápido, Quicksort o Mergesort?

- Son los dos igual de rápidos, ya que el coste temporal asintótico de ambos es  $O(n \log n)$
- Como su nombre indica, el Quicksort
- El Mergesort es siempre más rápido o igual (salvo una constante) que el Quicksort

Sea  $f(n)$  la solución de la relación de recurrencia  $f(n) = 2f(n/2) + n$ ;  $f(1) = 1$ . Indicad cuál de estas tres expresiones es cierta:

- f(n) ∈ Θ(n<sup>2</sup>)
- f(n) ∈ Θ(n log n)
- f(n) ∈ Θ(n)

¿Cuál de estas afirmaciones es cierta?



- La memoización evita que un algoritmo recursivo ingenuo resuelva repetidamente el mismo problema
- La ventaja de la solución de programación dinámica iterativa al problema de la mochila discreta es que nunca se realizan cálculos innecesarios
- Los algoritmos iterativos de programación dinámica utilizan memoización para evitar resolver de nuevo los mismos subproblemas que se vuelven a presentar

Sea  $g(n) = \sum_{i=0}^K a_i n^i$ . Di cuál de las siguientes afirmaciones es falsa:

- $\bullet$   $g(n) \in \Omega(n^k)$
- $\circ$  Las otras dos afirmaciones son falsas.
- $\circ$   $g(n) \in \Theta(n^k)$

Indicad cuál de estas tres expresiones es falsa:

- $\Theta(n) \subset \Theta(n^2)$
- $\Theta(n) \subset O(n)$
- $\Theta(n/2) = \Theta(n)$

Indicad cuál es el coste temporal asintótico (o complejidad temporal), en función de  $n$ , del programa siguiente:

```
s=0; for(i=0; i<n; i++) for(j=i; j<n; j++) s+=n*i*j;
```

- Es  $\Theta(n^2)$
- Es  $\Theta(n)$
- Es  $O(n^2)$  pero no  $\Omega(n^2)$

¿Pertenece  $3n^2 + 3$  a  $O(n^3)$ ?

  No

  Sí

Sólo para  $c = 1$  y  $n_0 = 5$

Los algoritmos de vuelta atrás que hacen uso de cotas optimistas generan soluciones posibles al problema mediante...

- un recorrido guiado por estimaciones de las mejores ramas del árbol que representan el espacio de soluciones
- un recorrido guiado por una cola de prioridad de donde se extraen primero los nodos que representan los subárboles más prometedores del espacio de soluciones
- un recorrido en profundidad del árbol que representa el espacio de soluciones

Si  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$  entonces ...

- f(n) ∈ Θ(g(n))
- g(n) ∈ O(f(n))
- f(n) ∈ O(g(n))

Si  $f \in \Theta(g_1)$  y  $f \in \Theta(g_2)$  entonces:

- $f \in \Theta(g_1 * g_2)$
- $f \in \Theta(g_1 + g_2)$
- $f \notin \Theta(\max(g_1, g_2))$

¿Cuál es el coste espacial asintótico del siguiente algoritmo?

```
int f( int n ) {  
    int a = 1, r = 0;  
    for( int i = 0; i < n; i++ ) {  
        r = a + r;  
        a = 2 * r;  
    }  
    return r;  
}
```

   $O(n)$

   $O(1)$

$O(\log n)$

Se quiere reducir la complejidad temporal de la siguiente función haciendo uso de programación dinámica. ¿Cuál sería la complejidad temporal resultante?

```
unsigned g( unsigned n, unsigned r) {  
    if (r==0 || r==n)  
        return 1;  
    return g(n-1, r-1) + g(n-1, r);  
}
```

- Se puede reducir hasta lineal
- Cuadrática
- La función no cumple con los requisitos necesarios para poder aplicar programación dinámica

¿Cuál de estas afirmaciones es falsa?

- Hay problemas de optimización en los cuales el método voraz sólo obtiene la solución óptima para algunas instancias y un subóptimo para muchas otras instancias
- Todos los problemas de optimización tienen una solución voraz que es óptima sea cual sea la instancia a resolver
- Hay problemas de optimización para los cuales se puede obtener siempre la solución óptima utilizando una estrategia voraz

Dada la relación de recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ pT\left(\frac{n}{a}\right) + g(n) & \text{en otro caso} \end{cases}$$

(donde  $p$  y  $a$  son enteros mayores que 1 y  $g(n) = n^k$ ), ¿qué tiene que ocurrir para que se cumpla  $T(n) \in \Theta(n^k)$ ?

- p > a<sup>k</sup>
- p < a<sup>k</sup>
- p = a<sup>k</sup>

¿Cuál es la diferencia principal entre una solución de vuelta atrás y una solución de ramificación y poda para el problema de la mochila?

- El orden de exploración de las soluciones
- El coste asintótico en el caso peor
- El hecho que la solución de ramificación y poda puede empezar con una solución subóptima voraz y la de vuelta atrás no.

La complejidad temporal (o coste temporal asintótico) en el mejor de los casos...

- es una función de la talla, o tamaño del problema, que tiene que estar definida para todos los posibles valores de ésta.
- es el tiempo que tarda el algoritmo en resolver la talla más pequeña que se le puede presentar
- las dos anteriores son verdaderas.

Dado un problema de optimización, ¿cuándo se puede aplicar el método de vuelta atrás?



- Es condición necesaria (aunque no suficiente) que el dominio de las decisiones sea discreto o discretizable.
- Es condición necesaria y suficiente que el dominio de las decisiones sea discreto o discretizable.
- No sólo es condición necesaria que el dominio de las decisiones sea discreto o discretizable; además, debe cumplirse que se puedan emplear mecanismos de poda basados en la mejor solución hasta el momento.

La siguiente relación de recurrencia expresa la complejidad de un algoritmo recursivo, donde  $g(n)$  es una función polinómica:  
(imagen)

Di cuál de las siguientes afirmaciones es cierta:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$$

- Si  $g(n) \in \Theta(n)$  la relación de recurrencia representa la complejidad temporal en el caso mejor del algoritmo de ordenación Quicksort
- Si  $g(n) \in \Theta(n)$  la relación de recurrencia representa la complejidad temporal en el caso peor del algoritmo de ordenación Quicksort
- Si  $g(n) \in \Theta(1)$  la relación de recurrencia representa la complejidad temporal en el caso mejor del algoritmo de ordenación Quicksort

Un algoritmo recursivo basado en el esquema divide y vencerás...

- ⚡ Nunca tendrá una complejidad exponencial
- ✓ Será más eficiente cuanto más equitativa sea la división en subproblemas
- Las demás opciones son verdaderas

El coste temporal de un algoritmo se ajusta a la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} 1 & n = 0 \\ n + \sum_{j=0}^{n-1} T(j) & n > 1 \end{cases}$$

¿qué coste temporal asintótico (o complejidad temporal) tendrá el algoritmo?

- O( $n \log n$ )
- O( $2^n$ )
- O( $n^2$ )

. Estudiad la relación de recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ pT\left(\frac{n}{q}\right) + g(n) & \text{en otro caso} \end{cases}$$

(donde  $p$  y  $q$  son enteros mayores que 1). Di cuál de los siguientes esquemas algorítmicos produce de manera natural relaciones de recurrencia así.

- Programación dinámica
- Divide y vencerás
- Ramificación y poda

Cogemos el algoritmo de Mergesort y en lugar de dividirlo el vector en dos partes, lo dividimos en tres. Posteriormente combinamos las soluciones parciales. ¿Cuál sería la complejidad temporal asintótica de la combinación de las soluciones parciales?

- $\Theta(\log n)$
- $\Theta(n)$
- Ninguna de las otras dos opciones es cierta

Uno de estos tres problemas no tiene una solución trivial y eficiente que siga el esquema voraz

- El problema de la mochila continua
- El problema de la mochila discreta sin limitación en la carga máxima de la mochila
- El problema del cambio

¿Qué se entiende por "tamaño del problema"?

- El valor máximo que puede tomar una instancia cualquiera de ese problema
- El número de parámetros que componen el problema
- La cantidad de espacio en memoria que se necesita para codificar una instancia de ese problema

Si  $\lim_{n \rightarrow \infty} (f(n)/n^2) = k$ , y  $k \neq 0$ , ¿cuál de estas tres afirmaciones es cierta?

- f(n) ∈ Θ(n<sup>2</sup>)
- f(n) ∈ Ω(n<sup>3</sup>)
- f(n) ∈ Θ(n<sup>3</sup>)

## Las relaciones de recurrencia...

- sirven para reducir el coste temporal de una solución cuando es prohibitivo
- expresan recursivamente el coste temporal de un algoritmo
- aparecen sólo cuando la solución sea del tipo divide y vencerás

Los algoritmos de ordenación Quícksort y Mergesort tienen en común...

- ... que se ejecutan en tiempo  $O(n)$ .
- ... que aplican la estrategia de divide y vencerás.
- ... que ordenan el vector sin usar espacio adicional.

¿Cuál de estas tres expresiones es cierta?

- $O(2^{\log n}) \subset O(n^2) \subset O(2^n)$
- $O(n^2) \subset O(2^{\log n}) \subseteq O(2^n)$
- $O(n^2) \subset O(2^{\log n}) \subset O(2^n)$

El coste temporal del algoritmo de ordenación por inserción es...

- $O(n \log n)$
- $O(n^2)$
- $O(n)$

Tenemos  $n$  substancias diferentes en polvo y queremos generar todas las distintas formas de mezclarlas de forma que el peso total no supere un gramo. Como la balanza que tenemos solo tiene una precisión de 0.1 gramos, no se considerarán pesos que no sean múltiplos de esta cantidad. Queremos hacer un programa que genere todas las combinaciones posibles.

- X  No se puede usar vuelta atrás porque las decisiones no son valores discretos.
- ✓  No hay ningún problema en usar una técnica de vuelta atrás.
- No se puede usar vuelta atrás porque el número de combinaciones es infinito.

¿Cuál es la complejidad temporal en el mejor de los casos de la siguiente función?

```
void examen (vector <int>& v) {
    int i=0, j, x, n=v.size();
    bool permuta=1;
    while (n>0 && permuta) {
        i=i+1;
        permuta=0;
        for (j=n-1; j>=i; j--)
            if (v[j] < v[j-1]) {
                x=v[j];
                permuta=1;
                v[j]=v[j-1];
                v[j-1]=x;
            }
    }
}
```

- Esta función no tiene caso mejor
- $\Omega(n)$
- $\Omega(1)$

¿Cuál de estas afirmaciones es falsa?

- Los algoritmos iterativos de programación dinámica utilizan memoización para evitar resolver de nuevo los mismos subproblemas que se vuelven a presentar
- La solución de programación dinámica iterativa al problema de la mochila discreta realiza cálculos innecesarios
- La memoización evita que un algoritmo recursivo ingenuo resuelva repetidamente el mismo problema

Sea la siguiente relación de recurrencia

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + g(n) & \text{en otro caso} \end{cases}$$

Si  $T(n) \in O(n)$ , ¿en cuál de estos tres casos nos podemos encontrar?

- $\bullet$   $g(n) = n^2$
- $\circ$   $g(n) = 1$
- $\circ$   $g(n) = n$

De las siguientes expresiones, o bien dos son verdaderas y una es falsa, o bien dos son falsas y una es verdadera. Marca la que (en este sentido) es distinta a las otras dos.

(imagen)

$$O(n^2) \subset O(2^{\log_2(n)}) \subset O(2^n)$$

$$(4^{\log_2(n)}) \subseteq O(n^2) \subset O(2^n)$$

$$O(2^{\log_2(n)}) \subseteq O(n^2) \subset O(n!)$$

- a
- b
- c

Si  $f \in \Theta(g_1)$  y  $f \in \Theta(g_2)$  entonces

- a  $f^2 \in \Theta(g_1 \cdot g_2)$
- b Las otras dos opciones son ambas ciertas.
- c  $f \in \Theta(\max(g_1, g_2))$

X

a

✓

b

c

La complejidad en el mejor de los casos de un algoritmo de ramificación y poda...

- es siempre exponencial con el número de decisiones a tomar
- puede ser polinómica con el número de decisiones a tomar
- suele ser polinómica con el número de alternativas por cada decisión

El problema de la función compuesta mínima consiste en encontrar, a partir de un conjunto de funciones dadas, la secuencia mínima de composiciones de éstas que permita transformar un número  $n$  en otro  $m$ . Se quiere resolver mediante ramificación y poda. ¿Cuál sería la forma más adecuada de representar las posibles soluciones?



Mediante un vector de booleanos

Mediante un vector de reales

Este problema no se puede resolver usando ramificación y poda si



no se fija una cota superior al número total de aplicaciones de funciones

La función  $\gamma$  de un número semientero positivo (un número es semientero si al restarle 0.5 es entero) se define como:

```
double gamma( double n ) { // Se asume n>=0.5 y n-0.5 entero
    if( n == 0.5 )
        return sqrt(PI);
    return n * gamma( n - 1 );
}
```

¿Se puede calcular usando programación dinámica iterativa?

- X  No, ya que el índice del almacén sería un número real y no entero
- ✓  Sí, pero la complejidad temporal no mejora
- No, ya que no podríamos almacenar los resultados intermedios en el almacén

¿Cuál de estas expresiones es falsa?

- n + n log n  $\in \Omega(n)$
- $2n^2 + 3n + 1 \in O(n^3)$
- n + n log n  $\in \Theta(n)$

Dada la relación de recurrencia:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ pT\left(\frac{n}{a}\right) + g(n) & \text{en otro caso} \end{cases}$$

(donde  $p$  y  $a$  son enteros mayores que 1 y  $g(n) = n^k$ ), ¿qué tiene que ocurrir para que se cumpla  $T(n) \in \Theta(n^k \log_a(n))$

- p > a<sup>k</sup>
- p = a<sup>k</sup>
- p < a<sup>k</sup>

Ante un problema de optimización resuelto mediante backtracking, ¿Puede ocurrir que el uso de las cotas pesimistas y optimistas sea inútil, incluso perjudicial?

- Según el tipo de cota, las pesimistas puede que no descarten ningún nodo pero el uso de cotas optimistas garantiza la reducción del espacio de búsqueda
- No, las cotas tanto optimistas como pesimistas garantizan la reducción del espacio de soluciones y por tanto la eficiencia del algoritmo
- Sí, puesto que es posible que a pesar de utilizar dichas cotas no se descarte ningún nodo

Di cuál de estos resultados de coste temporal asintótico es falsa:

- La búsqueda binaria en un vector ordenado requiere en el peor caso un tiempo en  $O(\log n)$ .
- La ordenación de un vector usando el algoritmo Quicksort requiere en el peor caso un tiempo en  $\Omega(n^2)$
- La ordenación de un vector usando el algoritmo Mergesort requiere en el peor caso un tiempo en  $\Omega(n^2)$

Un problema se puede resolver por Divide y Vencerás siempre que:

- Cumpla el principio de optimidad
- Cumpla el teorema de reducción
- Ninguna de las anteriores

¿Con qué esquema de programación obtenemos algoritmos que calculan la distancia de edición entre dos cadenas?

- Programación Dinámica
- Divide y Vencerás
- Ambos

¿Qué esquema de programación es el adecuado para resolver el problema de la búsqueda binaria?

- Divide y Vencerás
- Programación Dinámica
- Ninguno de los dos

## ¿Cuándo utilizaremos Programación Dinámica en lugar de Divide y Vencerás?

- Cuando se reduce el coste espacial
- Cuando se incrementa la eficiencia
- Cuando se incrementa la eficacia

Si  $n$  es el número de elementos del vector, el coste del algoritmo Mergesort es:

- $\Theta(n^2)$  y  $\Omega(n \log(n))$
- $\Theta(n \log(n))$
- $\Theta(n^2)$

Dado el algoritmo de búsqueda binaria, supongamos que, en vez de dividir la lista de elementos en dos mitades del mismo tamaño, la dividamos en dos partes de tamaños  $1/3$  y  $2/3$ . El coste de este algoritmo:

- Es menor que el del original
- Es mayor que el del original
- Es el mismo que el original

Para que un problema de optimización se pueda resolver mediante programación dinámica es necesario que:

- Cumpla el principio de optimalidad
- Cumpla el teorema de reducción
- Cumpla las dos anteriores

La serie de números de Fibonacci se define de la siguiente forma: (imagen)

¿Qué implementación de entre las siguientes supone el menor coste?

$$fib(n) = \begin{cases} 1 & n \leq 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

- Divide y vencerás
- Programación dinámica
- Ambas tienen el mismo coste asintótico

En programación dinámica, dónde almacenaremos los valores de los problemas resueltos?

- En un vector bidimensional
- Depende del problema
- En un vector unidimensional

¿Qué esquema de programación es el adecuado para resolver el problema k-ésimo mínimo en un vector?

-   Programación Dinámica
-   Divide y Vencerás
- Ninguno de los dos

¿Qué esquema algorítmico utiliza el algoritmo de ordenación Quicksort?

- Programación Dinámica
- Divide y Vencerás
- Backtracking

Dada la solución recursiva al problema de encontrar el  $k$ -ésimo mínimo de un vector. Cada llamada recursiva, ¿cuántas nuevas llamadas recursivas genera?

- ⚡ dos o ninguna
- ⚡ una o dos
- ⚡ una o ninguna

Si aplicamos programación dinámica a un problema que también tiene solución por divide y vencerás podemos asegurar que...

-   El coste temporal se reduce y el espacio aumenta con respecto a la solución por DyV
- El coste temporal aumenta y el espacio se reduce con respecto a la solución por DyV
-   Ninguna de las anteriores

Ante un problema que presenta una solución recursiva siempre podemos aplicar:

- ⚡ Divide y vencerás
- ⚡ Programación Dinámica
- ⚡ Cualquiera de las dos anteriores

La serie de números de Fibonacci se define de la siguiente forma: (imagen)

Para implementar esta función podemos emplear...

$$fib(n) = \begin{cases} 1 & n \leq 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

- ⚡ Divide y vencerás
- Programación dinámica
- Cualquiera de las dos anteriores

La solución al problema de encontrar el  $k$ -ésimo mínimo de un vector pone en práctica la siguiente estrategia:

-   No ordena ningún elemento del vector
-   Ordena parcialmente el vector
- Ordenar totalmente el vector

En cual de los siguientes casos no se puede aplicar el esquema Divide y Vencerás:

-   Cuando los subproblemas son de tamaños muy diferentes
- Cuando el problema no cumple el principio de optimalidad
-   Se puede aplicar en ambos casos

La programación dinámica, para resolver un problema, aplica la estrategia...

- Se resuelven los problemas más pequeños y, combinando las soluciones, se obtienen las soluciones de problemas sucesivamente más grandes hasta llegar al problema original
- Se descompone el problema a resolver en subproblemas más pequeños, que se resuelven independientemente para finalmente combinar las soluciones de los subproblemas para obtener la solución del problema original
- Ninguna de las anteriores

Supongamos el problema de la mochila resuelto mediante Programación Dinámica y particularizado para  $n$  elementos y un peso máximo trasportable de  $P$ . ¿Es necesario calcular valores para toda la matriz auxiliar para obtener el resultado?

  Si

  No

Depende de los valores de  $n$  y  $P$

Disponemos de dos cadenas de longitudes m y n. Si resolvemos el problema de la distancia de edición mediante programación dinámica, ¿De qué tamaño debemos definir la matriz que necesitaremos?

- (m-1) x (n-1)
- m x n
- (m+1) x (n+1)

Si  $n$  es el número de elementos de un vector. La solución de menor coste al problema de encontrar su  $k$ -ésimo mínimo tiene la siguiente complejidad:

- $\Omega(n)$  y  $O(n \log(n))$
- $\Omega(n)$  y  $O(n^2)$
- Ninguna de las dos

Si  $n$  es el número de elementos de un vector. Podemos encontrar una solución al problema de encontrar su  $k$ -ésimo que esté acotada superiormente por:

- O(  $n^3$  )
- O( $n$ )
- Ninguna de las anteriores

Dada una solución recursiva a un problema, ¿Cómo podemos evitar la resolución de los mismos subproblemas muchas veces?

- Resolver los subproblemas de mayor a menor y guardar su resultado en una tabla, inicializándola con los problemas pequeños
- Resolver los subproblemas de mayor a menor y guardar su resultado en una tabla, inicializándola con los problemas más grandes
- Resolver los subproblemas de menor a mayor y guardar su resultado en una tabla, inicializándola con los problemas pequeños

Un problema de optimización cuya solución se puede expresar mediante una secuencia de decisiones cumple el principio de optimalidad si, dada una secuencia óptima:

- Existe al menos una subsecuencia de esa solución que corresponde a la solución óptima de su subproblema asociado
- Existe una subsecuencia de esa solución que corresponde a la solución óptima de su subproblema asociado
- Cualquier subsecuencia de esa solución corresponde a la solución óptima de su subproblema asociado

Si  $n$  es el número de elementos de un vector. La solución de menor coste al problema de la búsqueda binaria tiene la siguiente complejidad:

- $\Omega(\log(n))$  y  $O(n \log(n))$
- $\Omega(1)$  y  $O(\log(n))$
- $\Theta(n \log(n))$

¿Qué esquema de programación es el adecuado para resolver el problema k-ésimo mínimo en un vector?

- Programación Dinámica
- Divide y Vencerás
- Ninguno de los dos

El problema de la mochila, ¿Puede solucionarse de forma óptima empleando la estrategia de divide y vencerás?

- Sólo para el caso de la mochila con fraccionamiento
- Sólo para el caso de la mochila sin fraccionamiento
- Sí, se puede aplicar para ambos casos

Un vector de enteros de tamaño  $n$  tiene sus elementos estructurados en forma de montículo (heap). ¿Cuál es la complejidad temporal en el mejor de los casos de borrar el primer elemento del vector y reorganizarlo posteriormente para que siga manteniendo la estructura de montículo?

- Ninguna de las otras dos opciones es correcta
- Constante con el tamaño del vector
- $O(n)$

Di cuál de estos resultados de coste temporal asintótico es falsa:

- La ordenación de un vector usando el algoritmo Quicksort requiere en el peor caso un tiempo en  $\Omega(n^2)$
- La ordenación de un vector usando el algoritmo Mergesort requiere en el peor caso un tiempo en  $\Omega(n^2)$
- La búsqueda binaria en un vector ordenado requiere en el peor caso un tiempo en  $O(\log(n))$

La versión Quicksort que utiliza como pivote el elemento del vector que ocupa la primer posición...

- ... se comporta peor cuando el vector ya está ordenado
- ... se comporta mejor cuando el vector ya está ya ordenado
- ... El hecho de que el vector estuviera previamente ordenado o no, no influye en la complejidad temporal de este algoritmo

De las siguientes expresiones, o bien dos son verdaderas y una falsa, o bien dos son falsas y una verdadera.  
Marca la que (en este sentido) es distinta a las otras dos.

   $\Theta(n^2) \subset \Theta(n^3)$

$\Omega(n^2) \subset \Omega(n^3)$

   $O(n^2) \subset O(n^3)$

Sea  $f(n)$  la solución de la relación de recurrencia  $f(n) = 2*f(n/2) + n$ ;  $f(1) = 1$ . Indicad cuál de estas tres expresiones es cierta:

- $f(n) \in \Theta(n)$
- $f(n) \in \Theta(n \log(n))$
- $f(n) \in \Theta(n^2)$

Se dispone de un vector de tamaño  $n$  cuyos elementos están de antemano organizados formando un montículo (heap). ¿Cuál sería la complejidad temporal de reorganizar el vector para que quede ordenado?

- ⊗  $O(\log(n))$
- $O(n)$
- $O(n \log(n))$

Indica cuál es la complejidad, en función de  $n$ , del fragmento siguiente:

```
int a = 0;
for( int i = 0; i < n; i++ )
    for( int j = i; j > 0; j /=2 )
        a += A[i][j];
```

- O( $n \log(n)$ )
- O( $n$ )
- O( $n^2$ )

Sea  $f(n)$  la solución de la relación de recurrencia... (imagen)

Indicad cuál de estas tres expresiones es cierta:

$$f(n) = 2f(n - 1) + 1; f(1) = 1.$$

- $\bullet$   $f(n) \in \Theta(n^2)$
- $\circ$   $f(n) \in \Theta(2^n)$
- $\circ$   $f(n) \in \Theta(n)$

¿Cuál es la complejidad temporal de la siguiente función recursiva?

- $\Theta(n^2 \log(n))$
- $\Theta(n^2)$
- $\Theta(2^n)$

```
unsigned desperdicio
(unsigned n){
    if (n<=1)
        return 0;
    unsigned sum = desperdicio
        (n/2) + desperdicio (n/2);
    for (unsigned i=1; i<n-1;
        i++)
        for (unsigned j=1;
            j<=i; j++)
            sum+=i*j;
    return sum;
}
```

Un programa con dos bucles anidados uno dentro del otro. El primero hace  $n$  iteraciones aproximadamente y el segundo la mitad, tarda un tiempo...

- $O( n^2 )$
- $O( n\sqrt{n} )$
- $O( n \log(n) )$

¿Cuál es la complejidad temporal de la siguiente función recursiva?

```
unsigned desperdicio (unsigned n) {
    if (n<=1)
        return 0;
    unsigned sum = desperdicio (n/2) + desperdicio (n/2);
    for (unsigned i=1; i<n-1; i++)
        for (unsigned j=1; j<=i; j++)
            for (unsigned k=1; k<=j; k++)
                sum+=i*j*k;
    return sum;
}
```

✗   $\Theta(2^n)$

$\Theta(n^3 \log(n))$

✓   $\Theta(n^3)$

Dado el siguiente fragmento de código... (imagen)

Indica cuál de las siguientes expresiones es la que mejor refleja su complejidad temporal.

```
for(i=0;i<n;i++) for(j=1;j< i;j+=2) s+=i*j;
```

- $\Sigma \log(j)$
- Ninguna de las otras dos opciones son correctas
- $\Sigma \log(i)$

Un algoritmo recursivo basado en el esquema divide y vencerás...

- Las demás opciones son verdaderas
- ... nunca tendrá una complejidad exponencial.
- ... será más eficiente cuanto más equitativa sea la división en subproblemas.

Sea  $f(n)$  la solución de la relación de recurrencia  $f(n) = f(n/2) + 1$ ;  $f(1) = 1$ . Indicad cuál de estas tres expresiones es cierta:

- $f(n) \in \Theta(n)$
- $f(n) \in \Theta(n \log(n))$
- $f(n) \in \Theta(\log(n))$

Indica cuál es la complejidad en función de n del fragmento siguiente:

```
k=n/4;  
for( int i = k; i < n - k; i++) {  
    A[i] = 0;  
    for( int j = i - k; j < i + k; j++ )  
        A[i] += B[j];  
}
```

- O(  $n^2$  )
- O( n )
- O(  $n \log(n)$  )

Un problema de tamaño  $n$  puede transformarse en tiempo  $O(n^3)$  en ocho de tamaño  $n/2$ ; por otro lado, la solución al problema cuando la talla es 1 requiere un tiempo constante.

¿Cuál de estas clases de coste temporal asintótico es la más ajustada?

- O(  $n^3$  )
- O(  $n^2 \log(n)$  )
- O(  $n^3 \log(n)$  )

Para la complejidad de un algoritmo presente caso mejor y peor distintos...

- ... es condición necesaria que existan instancias distintas del problema con el mismo tamaño.
- ... es condición suficiente que existan instancias distintas del problema con el mismo tamaño.
- ... es condición necesaria y suficiente que existan instancias distintas del problema con el mismo tamaño.

De las siguientes expresiones, o bien dos son verdaderas y una falsa, o bien dos son falsas y una verdadera. Marca la que (en este sentido) es distinta a las otras dos.

- $O( n^2 ) \subset O( n^3 )$
- $\Omega( n^3 ) \subset \Omega( n^2 )$
- $\Theta( n^2 ) \subset \Theta( n^3 )$

De las siguientes expresiones, o bien dos son verdaderas y una falsa, o bien dos son falsas y una verdadera. Marca la que (en este sentido) es distinta a las otras dos.

- $O( 2^{(\log_2(n))} ) \subset O( n^2 ) \subset O( 2^n )$
- $O( n^2 ) \subset O( 2^{(\log_2(n))} ) \subseteq O( 2^n )$
- $O( n^2 ) \subset O( 2^{(\log_2(n))} ) \subset O( 2^n )$

De las siguientes expresiones, o bien dos son verdaderas y una falsa, o bien dos son falsas y una verdadera. Marca la que (en este sentido) es distinta a las otras dos.

- $\Theta(\log^2(n)) = \Theta(\log(n^2))$
- $\Theta(\log^2(n)) = \Theta(\log(n))$
- $\Theta(\log(n)) = \Theta(\log(n^2))$

Dada la siguiente relación de recurrencia, ¿Qué cota es verdadera?

$$f(n) = \begin{cases} 1 & n = 1 \\ n^2 + 3f(n/3) & n > 1 \end{cases}$$

- f(n) ∈ Θ( n<sup>2</sup> log(n) )
- f(n) ∈ Θ( n<sup>2</sup> )
- f(n) ∈ Θ( n )

Dada la siguiente relación de recurrencia, ¿Qué cota es verdadera?

$$f(n) = \begin{cases} 1 & n = 1 \\ n + 3f(n/3) & n > 1 \end{cases}$$

- f(n) ∈ Θ( n log(n) )
- f(n) ∈ Θ( n^3 )
- f(n) ∈ Θ( n )

Un problema de tamaño  $n$  puede transformarse en tiempo  $O(n)$  en siete de tamaño  $n/7$ ; por otro lado, la solución al problema cuando la talla es 1 requiere un tiempo constante.

¿Cuál de estas clases de coste temporal asintótico es la más ajustada?

   $O( n )$

   $O( n \log(n) )$

$O( n^2 )$

Se pretende implementar mediante programación dinámica iterativa la función recursiva: (imagen)

¿Cuál es la mejor complejidad espacial que se puede conseguir?

```
unsigned f( unsigned y, unsigned x){ // suponemos y >= x
    if (x==0 || y==x) return 1;
    return f(y-1, x-1) + f(y-1, x);
}
```

- O( $y^2$ )
- O( $y$ )
- O(1)

La mejora que en general aporta la programación dinámica frente a la solución ingenua se consigue gracias al hecho de que...

- ... en la solución ingenua se resuelve pocas veces un número relativamente grande de subproblemas distintos.
- ... en la solución ingenua se resuelve muchas veces un número relativamente pequeño de subproblemas distintos.
- El número de veces que se resuelven los subproblemas no tiene nada que ver con la eficiencia de los problemas resueltos mediante programación dinámica.

La mejora que en general aporta la programación dinámica frente a la solución ingenua se consigue gracias al hecho de que...

- el número de veces que se resuelven los subproblemas no tiene nada que ver con la eficiencia de los problemas resueltos mediante programación dinámica
- en la solución ingenua se resuelve muchas veces un número relativamente pequeño de subproblemas distintos
- en la solución ingenua se resuelve pocas veces un número relativamente grande de subproblemas distintos

Supongamos que una solución recursiva a un problema de optimización muestra estas dos características: por un lado, se basa en obtener soluciones óptimas a problemas parciales más pequeños, y por otro, estos subproblemas se resuelven más de una vez durante un proceso recursivo. Este problema es candidato a tener una solución alternativa basada en...

- Un algoritmo de programación dinámica
- Un algoritmo de estilo Divide y Vencerás
- Un algoritmo voraz

## La programación dinámica...



- Las otras dos opciones son ciertas

- En algunos casos se puede utilizar para resolver problemas de optimización con dominios continuos pero probablemente pierda su eficacia ya que puede disminuir drásticamente el número de subproblemas repetidos
- Normalmente se usa para resolver problemas de optimización con dominios discretizables puesto que las tablas se han de indexar con este tipo de valores

La solución de programación dinámica iterativa del problema de la mochila discreta...

- ... tiene la restricción de que los valores tienen que ser enteros positivos
- ... tiene un coste temporal asintótico exponencial con respecto al número de objetos
- ... calcula menos veces al valor de la mochila que la correspondiente solución de programación dinámica recursiva

El valor que se obtiene con el método voraz para el problema de la mochila discreta es...

- una cota inferior para el valor óptimo que a veces puede ser igual a este
- una cota superior para el valor óptimo
- una cota inferior para el valor óptimo, pero que nunca coincide con este

Se pretende implementar mediante programación dinámica iterativa la función recursiva: (imagen)

¿Cuál es la mejor estructura para el almacén?

```
unsigned f( unsigned x, unsigned v[] ) {  
    if (x==0)  
        return 0;  
    unsigned m = 0;  
    for ( unsigned k = 0; k < x; k++ )  
        m = max( m, v[k] + f( x-k, v ) );  
    return m;  
}
```

  int A

int A [] []

  int A[]

Se pretende implementar mediante programación dinámica iterativa la función recursiva: (imagen)

¿Cuál es la mejor estructura para el almacén?

```
unsigned f( unsigned y, unsigned x){ // suponemos y >= x
    if (x==0 || y==x) return 1;
    return f(y-1, x-1) + f(y-1, x);
}
```

  int A

  int A[] []

int A[]

De los problemas siguientes, indicad cuál no se puede tratar eficientemente como los otros dos:

- El problema de cortar un tubo de forma que se obtenga el máximo beneficio posible
- El problema del cambio, o sea, el de encontrar la manera de entregar una cantidad de dinero usando el mínimo de monedas posibles
- El problema de la mochila sin fraccionamiento y sin restricciones en cuanto al dominio de los pesos de los objetos y de sus valores

¿Cuál de estos tres problemas de optimización no tiene, o no se le conoce, una solución voraz óptima?

- El árbol de cobertura de coste mínimo de un grafo conexo
- El problema de la mochila continua o con fraccionamiento
- El problema de la mochila discreta o sin fraccionamiento

Se pretende implementar mediante programación dinámica iterativa la función recursiva: (imagen)

¿Cuál es la mejor complejidad espacial que se puede conseguir?

```
int f( int x, int y ) {  
    if( x <= y ) return 1;  
    return x + f(x-1,y);  
}
```

- O( $x$ )
- O(1)
- O( $x^2$ )

El problema de encontrar un árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado...

- no se puede resolver en general con una estrategia voraz
- sólo se puede resolver con una estrategia voraz si existe una arista para cualquier par de vértices del grafo
- se puede resolver siempre con una estrategia voraz

## En el método voraz...

- ... es habitual preparar los datos para disminuir el coste temporal de la función que determina cuál es la siguiente decisión a tomar
- ... siempre se encuentra solución pero puede que no sea óptima
- ... el dominio de las decisiones sólo pueden ser conjuntos discretos o discretizables

¿Cuál de estas tres estrategias voraces obtiene un mejor valor para la mochila discreta?

- Meter primero los elementos de mayor valor
- Meter primero los elementos de menor peso
- Meter primero los elementos de mayor valor específico o valor por unidad de peso

Si ante un problema de decisión existe un criterio de selección voraz entonces...

-   al menos una solución factible está garantizada
- la solución óptima está garantizada
-   ninguna de las otras dos opciones es cierta

De los problemas siguientes, indicad cuál no se puede tratar eficientemente como los otros dos:

-   El problema de cortar un tubo de forma que se obtenga el máximo beneficio posible
-   El problema de la mochila sin fraccionamiento y sin restricciones en cuanto al dominio de los pesos de los objetos y de sus valores
- El problema del cambio, o sea, el de encontrar la manera de entregar una cantidad de dinero usando el mínimo de monedas posibles

Un tubo de  $n$  centímetros de largo se puede cortar en segmentos de 1 centímetro, 2 centímetros, etc... Existe una lista de los precios a los que se venden los segmentos de cada longitud. Una de las maneras de cortar el tubo es la que más ingresos nos producirá.

Di cuál de estas tres afirmaciones es FALSA.

-   Hace una evaluación exhaustiva "de fuerza bruta" de todas las posibles maneras de cortar el tubo consume un tiempo  $\Theta(2^n)$
- Es posible evitar hacer la evaluación exhaustiva "de fuerza bruta"
- guardando, para cada posible longitud  $j < n$  el precio más elevado posible que se puede obtener dividiendo el tubo correspondiente
-   Hacer una evaluación exhaustiva "de fuerza bruta" de todas las posibles maneras de cortar el tubo consume un tiempo  $\Theta(n!)$

¿Cómo se vería afectada la solución voraz al problema de la asignación de tareas en el caso de que se incorporaran restricciones que contemplen que ciertas tareas no pueden ser adjudicadas a ciertos trabajadores?

- Ya no se garantizaría la solución óptima pero sí una factible
  - Habría que replantearse el criterio de selección para comenzar por aquellos trabajadores con más restricciones en cuanto a las tareas que no pueden realizar para asegurar, al menos, una solución factible
  - La solución factible ya no estaría garantizada, es decir, pudiera ser que el algoritmo no llegue a solución alguna

Cuando la descomposición recursiva de un problema da lugar a subproblemas de tamaño similar, ¿qué esquema promete ser más apropiado?

- El método voraz
- Programación dinámica
- Divide y vencerás, siempre que se garantice que los subproblemas no son del mismo tamaño

En la solución al problema de la mochila continua ¿por qué es conveniente la ordenación previa de los objetos?

- Para reducir la complejidad temporal en la toma de cada decisión: de  $O(n)$  a  $O(1)$ , donde n es el número de objetos a considerar.
- Porque si no se hace no es posible garantizar que la toma de decisiones siga un criterio voraz.
- Para reducir la complejidad temporal en la toma de cada decisión: de  $O(n^2)$  a  $O(n \log(n))$ , donde n es el número de objetos a considerar

Supongamos que un informático quiere subir a una montaña y como no es una persona normal decide que tras cada paso, el siguiente debe tomarlo en la dirección de máxima pendiente hacia arriba. Además, entenderá que ha alcanzado la cima cuando llegue a un punto en el que no haya ninguna dirección que sea cuesta arriba.

¿Qué tipo de algoritmo está usando nuestro informático?

- Un algoritmo divide y vencerás
- Un algoritmo de programación dinámica
- Un algoritmo voraz

Los algoritmos de programación dinámica hacen uso...

- ... de que la solución óptima se puede construir añadiendo a la solución el elemento óptimo de los elementos restantes, uno a uno.
- ... de que se puede ahorrar cálculos guardando resultados anteriores en un almacén.
- ... de una estrategia trivial consistente en examinar todas las soluciones posibles.

La eficiencia de los algoritmos voraces se basa en el hecho de que...

- antes de tomar una decisión se comprueba si satisface las restricciones del problema
- las decisiones tomadas nunca se reconsideran
- con antelación, las posibles decisiones se ordenan de mejor a peor

Se pretende implementar mediante programación dinámica iterativa la función recursiva: (imagen)

¿Cuál es la mejor complejidad espacial que se puede conseguir?

```
float f(unsigned x, int y){  
    if( y < 0 ) return 0;  
    float A = 0.0;  
    if ( v1[y] <= x )  
        A = v2[y] + f( x-v1[y], y-1 );  
    float B = f( x, y-1 );  
    return min(A,2+B);  
}
```

- O( $y^2$ )
- O(y)
- O(1)

Dada la suma de la recurrencia: (imagen)

¿cuál de las siguientes afirmaciones es cierta?

$$T(n) = \begin{cases} 1 & n=0 \\ \sum_{k=0}^{n-1} T(k) & n>0 \end{cases}$$

- T(n) ∈ Θ(2^n)
- T(n) ∈ Θ(n!)
- T(n) ∈ Θ(n^2)

Dado un problema de optimización, el método voraz...

-   siempre obtiene una solución factible
-   siempre obtiene la solución óptima
-   garantiza la solución óptima sólo para determinados problemas

¿Cuál de estas estrategias para calcular el  $n$ -ésimo elemento de la serie de Fibonacci (imagen) es más eficiente?

$$(f(n) = f(n-1) + f(n-2), f(1) = f(2) = 1)$$

- La estrategia voraz
- Las dos estrategias citadas serían similares en cuanto a eficiencia
- Programación Dinámica

El problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado...

- no se puede resolver en general con una estrategia voraz
- sólo se puede resolver con una estrategia voraz si existe una arista para cualquier par de vértices del grafo
- se puede resolver siempre con una estrategia voraz

Se pretende implementar mediante programación dinámica iterativa la función recursiva: (imagen)

¿Cuál es la mejor complejidad espacial que se puede conseguir?

```
unsigned f( unsigned x, unsigned v[] ) {  
    if (x==0)  
        return 0;  
  
    unsigned m = 0;  
  
    for ( unsigned k = 1; k < x; k++ )  
        m = max( m, v[k] + f( x-k, v ) );  
  
    return m;  
}
```

- O( $x^2$ )
- O( $x$ )
- O(1)

¿Cuál de estas estrategias voraces obtiene siempre un mejor valor para la mochila discreta?

- ninguna de las otras dos opciones es cierta
- meter primero los elementos de mayor valor específico o valor por unidad de peso
- meter primero los elementos de mayor valor

Los algoritmos de programación dinámica hacen uso ...

- ... de una estrategia trivial consistente en examinar todas las soluciones posibles.
- ... de que se puede ahorrar cálculos guardando resultados anteriores en un almacén.
- ... de que la solución óptima se puede construir añadiendo a la solución el elemento óptimo de los elementos restantes, uno a uno.

En la solución al problema de la mochila continua ¿por qué es conveniente la ordenación previa de los objetos?

- Para reducir la complejidad temporal en la toma de cada decisión: de  $O(n)$  a  $O(1)$ , donde  $n$  es el número de objetos a considerar
- Porque si no se hace no es posible garantizar que la toma de decisiones siga un criterio voraz
- Para reducir la complejidad temporal en la toma de cada decisión: de  $O(n^2)$  a  $O( n * \log(n) )$ , donde  $n$  es el número de objetos a considerar

Se pretende implementar mediante programación dinámica iterativa la función recursiva: (imagen)

¿Cuál es la mejor complejidad espacial que se puede conseguir?

```
unsigned f( unsigned y, unsigned x){ // suponemos y >= x
    if (x==0 || y==x) return 1;
    return f(y-1, x-1) + f(y-1, x);
}
```

- O( $x^2$ )
- O(y)
- O( $y^2$ )

¿Cómo se vería afectada la solución voraz al problema de la asignación de tareas en el caso de que incorporaran restricciones que contemplen que ciertas tareas no pueden ser adjudicadas a ciertos trabajadores?

- Ya no se garantizaría la solución óptima pero sí una factible
- Habría que replantearse el criterio de selección para comenzar por aquellos trabajadores con más restricciones en cuanto a las tareas que no pueden realizar para asegurar, al menos, una solución factible
- La solución factible ya no estaría garantizada, es decir, pudiera ser que el algoritmo no llegue a solución alguna

En el método voraz...

- El dominio de las decisiones sólo pueden ser conjuntos discretos o discretizables.
- Es habitual preparar los datos para disminuir el coste temporal de la función que determina cuál es la siguiente decisión a tomar.
- Siempre se encuentra solución pero puede que no sea la óptima.

La mejora que en general aporta la programación dinámica frente a la solución ingenua se consigue gracias al hecho de que...

- En la solución ingenua se resuelve pocas veces un número relativamente grande de subproblemas distintos
- En la solución ingenua se resuelve muchas veces un número relativamente pequeño de subproblemas distintos
- El número de veces que se resuelven los subproblemas no tiene nada que ver con la eficiencia de los problemas resueltos mediante programación dinámica

Cuando se calculan los coeficientes binomiales usando la recursión  $\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$ , con  $\binom{n}{0} = \binom{n}{n} = 1$ , qué problema se da y cómo se puede resolver?

- Se repiten muchos cálculos y ello se puede evitar usando programación dinámica.
- Se repiten muchos cálculos y ello se puede evitar haciendo uso de una estrategia voraz.
- La recursión puede ser infinita y por tanto es necesario organizarla según el esquema iterativo de programación dinámica.

¿Por qué se utiliza el TAD "Union-find" en el algoritmo de Kruskal?

- Para comprobar si dos vértices son equivalentes
- Para comprobar si un arco forma ciclos
- Para comprobar si un vértice ya ha sido visitado

¿Cuál de estos tres problemas de optimización no tiene, o no se le conoce, una solución voraz óptima?

- El árbol de cobertura de coste mínimo de un grafo conexo
- El problema de la mochila discreta o sin fraccionamiento
- El problema de la mochila continua o con fraccionamiento

De los problemas siguientes, indicad cuál no se puede tratar eficientemente como los otros dos:

- El problema del cambio, o sea, el de encontrar la manera de entregar una cantidad de dinero usando el mínimo de monedas posibles
- El problema de cortar un tubo de forma que se obtenga el máximo beneficio posible
- El problema de la mochila sin fraccionamiento y sin restricciones en cuanto al dominio de los pesos de los objetos y de sus valores

Se pretende implementar mediante programación dinámica iterativa la función recursiva: (Imagen)

¿Cuál es la mejor complejidad temporal que se puede conseguir?

```
float f(unsigned x, int y){  
    if( y < 0 ) return 0;  
    float A = 0.0;  
    if ( v1[y] <= x )  
        A = v2[y] + f( x-v1[y], y-1 );  
    float B = f( x, y-1 );  
    return min(A,2+B);  
}
```

- O(x)
- O( $x^y$ )
- O(v)

La solución óptima al problema de encontrar el árbol de recubrimiento de coste mínimo para un grafo no dirigido, conexo y ponderado...

- Debe construirlo vértice a vértice: arista a arista no puede ser
- Puede construirlo tanto vértice a vértice como arista a arista
- Debe construirlo arista a arista: vértice a vértice no puede ser

¿Qué mecanismo se usa para acelerar el algoritmo de Prim?

- Mantener para cada vértice su "padre" más cercano
- Mantener una lista de los arcos ordenador según su peso
- El TAD "Union-find"

El valor que se obtiene con el método voraz para el problema de la mochila discreta es...

- Una cota superior para el valor óptimo.
- Una cota inferior para el valor óptimo, pero que nunca coincide con este.
- Una cota inferior para el valor óptimo que a veces puede ser igual a este.

La programación dinámica...

- Normalmente se usa para resolver problemas de optimización con dominios discretizables puesto que las tablas se han de indexar con este tipo de valores
- Las otras dos opciones son ciertas
- En algunos casos se puede utilizar para resolver problemas de optimización con dominios continuos pero probablemente pierda su eficacia ya que puede disminuir drásticamente el número de subproblemas repetidos

¿Cuál de estas estrategias para calcular el n-ésimo elemento de la serie Fibonacci ( $f(n) = f(n-1) + f(n-2)$ ,  $f(1) = f(2) = 1$ ) es más eficiente?

- La estrategia voraz
- Programación dinámica
- Las dos estrategias citadas serían similares en cuanto a eficiencia

Se pretende aplicar la técnica memoización a la siguiente función recursiva: (imagen)

En el caso más desfavorable, ¿qué complejidades temporal y espacial cabe esperar de la función resultante?

```
int f( int x, int y ) {  
    if( x > y ) return 1;  
    return x*(y-1) + f(x,y-2);  
}
```

- ⚪ Temporal  $O(x*y)$  y espacial  $O(x)$
- ⚫ Ninguna de las otras dos opciones es correcta
- ⚫  $O(y - x)$ , tanto temporal como espacial

La eficiencia de los algoritmos voraces se basa en el hecho de que...

- antes de tomar una decisión se comprueba si satisface las restricciones del problema
- con antelación, las posibles decisiones se ordenan de mejor a peor
- las decisiones tomadas nunca se reconsideran

Se pretende implementar mediante programación dinámica iterativa la función recursiva: (imagen)

¿Cuál es la mejor complejidad temporal que se puede conseguir?

```
unsigned f( unsigned y, unsigned x){ // suponemos y >= x
    if (x==0 || y==x) return 1;
    return f(y-1, x-1) + f(y-1, x);
}
```

- O( $x \cdot y$ )
- O( $y$ )
- O( $x$ )

¿Cuál de los siguientes pares de problemas son equivalentes en cuanto al tipo de solución (óptima, factible, etc.) aportada por el método voraz?

- El fontanero diligente y el problema del cambio
- El fontanero diligente y la mochila continua
- El fontanero diligente y la asignación de tareas

Se pretende implementar mediante programación dinámica iterativa la función recursiva: (imagen)

¿Cuál es la mejor complejidad espacial que se puede conseguir?

```
unsigned f( unsigned y, unsigned x) { // suponemos y >= x
    if (x==0 || y==x) return 1;
    return f(y-1, x-1) + f(y-1, x);
}
```

- O( $y^2$ )
- O( $x^2$ )
- O( $y$ )

Un tubo de  $n$  centímetros de largo se puede cortar en segmentos de 1 centímetro, 2 centímetros, etc. Existe una lista de los precios a los que se venden los segmentos de cada longitud. Una de las maneras de cortar el tubo es la que más ingresos nos producirá. Di cuál de estas tres afirmaciones es falsa.

- Hacer una evaluación exhaustiva "de fuerza bruta" de todas las posibles maneras de cortar el tubo consume un tiempo  $\Theta(n!)$
- Hacer una evaluación exhaustiva "de fuerza bruta" de todas las posibles maneras de cortar el tubo consume un tiempo  $\Theta(2^n)$
- Es posible evitar hacer la evaluación exhaustiva "de fuerza bruta" guardando, para cada posible longitud  $j < n$  el precio más elevado posible que se puede obtener dividiendo el tubo correspondiente.

Se pretende implementar mediante programación dinámica iterativa la función recursiva: (imagen)

¿Cuál es la mejor complejidad temporal que se puede conseguir?

```
unsigned f( unsigned x, unsigned v[] ) {  
    if (x==0)  
        return 0;  
  
    unsigned m = 0;  
  
    for ( unsigned y = 1; y < x; y++ )  
        m = max( m, v[y] + f( x-y, v ) );  
  
    return m;
```

- O(x)
- O( $x^2$ )
- O( $x \cdot v$ )

Los algoritmos directos de ordenación, respecto de los indirectos:

-   Presentan una menor complejidad temporal y sus tiempos de ejecución absolutos son menores.
-   Presentan una mayor complejidad temporal y sus tiempos de ejecución absolutos son mayores.
- Presentan una mayor complejidad temporal si bien sus tiempos de ejecución absolutos son menores.

Si  $f_1(n) \in O(g_1(n))$  y  $f_2(n) \in O(g_2(n))$  entonces:

- f<sub>1</sub>(n)+f<sub>2</sub>(n) ∈ O(maximo(g<sub>1</sub>(n),g<sub>2</sub>(n)))
- f<sub>1</sub>(n)+f<sub>2</sub>(n) ∈ O (g<sub>1</sub>(n)+g<sub>2</sub>(n))
- Ambas son correctas

¿Por que se emplean funciones de coste para expresar el coste de una algoritmo?

- Para poder expresar el coste de los algoritmos con mayor exactitud
- Para que la expresión del coste del algoritmo sea válida para cualquier entrada al mismo
- Para poder expresar el coste de un algoritmo mediante una expresión matemática

$f(n) = 5n + 3m \cdot n + 11$  entonces  $f(n)$  pertenece a:

- $O(n \cdot m)$ .
- $O(n^m)$ .
- Las dos son correctas

Si dos algoritmos tienen la misma complejidad asintótica:

- ⚡ No necesitan exactamente el mismo tiempo para su ejecución.
- ⚡ Necesitan exactamente el mismo tiempo para su ejecución.
- ⚡ Ninguna de las anteriores

$f(n) = 5n+5$  ¿  $f(n)$  pertenece a  $O(n)$ ?

- Sí. El valor de  $c$  es 6 y el valor mínimo de  $n_0$  es de 5
- Sí. El valor de  $c$  es 5 y el valor mínimo de  $n_0$  es de 3
- Sí. El valor de  $c$  es 9 y el valor mínimo de  $n_0$  es de 1

¿Cuál de los siguientes algoritmos de ordenación tiene menor complejidad?

- Mergesort
- Inserción directa
- Burbuja

Dado  $\epsilon$  polinomio  $f(n) = A(m) n^m + A(m-1) n^{(m-1)} + \dots + A_0$ , con  $A(m) \in \mathbb{R}^+$  entonces  $f$  pertenece al orden:

- $\mathcal{O}(n^m)$ .
- La dos respuestas anteriores son correctas.
- $\Omega(n^m)$ .

Cual de las siguientes definiciones es cierta:

- Las cotas de complejidad se emplean cuando para una misma talla se obtienen diferentes complejidades dependiendo de la entrada al problema.
- Las cotas de complejidad se emplean cuando para diferentes tallas se obtienen diferentes complejidades dependiendo de la entrada al problema.
- Ninguna de las anteriores

Si  $f(n) \in \Omega(g(n))$  entonces:

- $\exists c, n_0 \in \mathbb{R}^+: f(n) \leq c \cdot g(n) \forall n \geq n_0$
- $\exists c, n_0 \in \mathbb{R}^+: f(n) \geq c \cdot g(n) \forall n$
- $\exists c, n_0 \in \mathbb{R}^+: f(n) \geq c \cdot g(n) \forall n \geq n_0$

¿Cuál es el objetivo de la etapa de análisis en el Diseño y Análisis de un Algoritmo?

- Estimar los recursos que consumirá el algoritmo una vez implementado.
- Determinar el lenguaje y herramientas disponibles para su desarrollo.
- Estimar la potencia y características del equipo informático necesarios para el correcto funcionamiento del algoritmo.

En un algoritmo recursivo, la forma de dividir el problema en subproblemas:

- Influye en su complejidad temporal.
- No influye en ninguna de sus complejidades.
- Influye en la complejidad espacial del mismo.

Ordena de menor a mayor las siguientes complejidades:

1.  $O(1)$
2.  $O(n^2)$
3.  $O(n \lg n)$
4.  $O(n!)$



3,1,2 y 4



1,3,2 y 4

1,3,4 y 2

Cuando para distintas instancias de problema con el mismo tamaño no obtenemos el mismo resultado:

- Calculamos el máximo y mínimo coste que nos puede dar el algoritmo.
  - No es posible calcular la complejidad a priori y debemos ejecutar el programa varias veces con la misma talla y obtener el tiempo medio para hallar la complejidad media.
  - No se puede aplicar la técnica de paso de programa, ya que esta técnica es para calcular la complejidad a priori.

El coste asociado a la siguiente ecuación de recurrencia es:

$$f(n) = \begin{cases} 1 & n \leq 1 \\ n + f(n/2) + f(n/2) & n > 1 \end{cases}$$

- ⊗  $\Theta(n^2 \lg(n))$
- ⊗  $\Theta(n \lg(n^2))$
- ⊗  $\Theta(n \lg(n))$

El estudio de la complejidad resulta realmente interesante para tamaños grandes de problema por varios motivos:

- Ninguna de las anteriores.
- Las diferencias reales en tiempo de compilación de algoritmos con diferente coste para tamaños pequeños del problema no suelen ser muy significativas.
- Las diferencias reales en tiempo de ejecución de algoritmos con diferente coste para tamaños grandes del problema no suelen ser muy significativas.

$f(n) = 10n+7$  ¿  $f(n)$  pertenece a  $O(n^2)$ ?

- Sí Para cualquier valor de  $c$  positivo siempre existe un  $n_0$  a partir del que se cumple.
- Sí. Para  $c = 1$  y a partir de un valor de  $n_0 = 10$ .
- No.

La talla o tamaño de un problema depende de:

- Conjunto de valores asociados a la entrada del problema.
- Conjunto de valores asociados a la salida del problema.
- Conjunto de valores asociados a la entrada y salida del problema.

El sumatorio, desde  $i=1$  hasta  $n$ , de  $i^k$  pertenece a:

- $O(n^{k+1})$
- $O(n^k)$
- Ninguna de las anteriores

La complejidad de la función A2 es:

```
Funcion A2 (n, a: entero):entero;
Var r: entero; fvar
    si ( $a^2 > n$ ) devuelve 0
    sino
        r:= A2(n, 2a);
        opción
            n <  $a^2$ :     devuelve r;
            n  $\geq a^2$ :   devuelve r + a;
        fopción
    fsi
fin
```

- O(  $\sqrt{n} / a$  )
- O(  $n / \sqrt{a}$  )
- O(  $\sqrt{n} \cdot a$  )

Si  $f_1(n) \in O(g_1(n))$  y  $f_2(n) \in O(g_2(n))$  entonces:

- $f_1(n) \cdot f_2(n) \in O(\max(g_1(n), g_2(n)))$
- $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$
- Ambas son correctas

¿El tiempo de ejecución de un algoritmo depende de la talla del problema?

- Sí, siempre
- Nc, nunca
- Nc necesariamente

La complejidad de la función TB es:

```
función TB (A: vector[λ]; iz , de : N) : N
var n,i:N;
n=iz-de+1
opcion
  (n < 1) : devuelve ( 0 );
  (n = 1) : devuelve ( 1 );
  (n > 1) : si (A[iz] = A[de]) entonces
              devuelve (TB( A, iz + 1, de - 1 ) + 1);
            sino
              devuelve (TB( A, iz + 1, de - 1 ) );
            finsi ;
fopcion
fin
```

- $\Theta(n^2 \cdot \lg n)$
- $\Theta(n)$
- $\Theta(n \cdot \lg n)$

Un algoritmo cuya talla es  $n$  y que tarda  $40^n$  segundos en resolver cualquier instancia tiene una complejidad temporal:

- $\Theta(n^n)$
- $\Theta(4^n)$
- Ninguna de las anteriores

El caso base de una ecuación de recurrencia asociada a la complejidad temporal de un algoritmo expresa:

-   El coste de dicho algoritmo en el mejor de los casos.
-   El coste de dicho algoritmo en el peor de los casos.
-   Ninguna de las anteriores

¿Cuál de las siguientes jerarquías de complejidades es la correcta?

- ... <  $(2^n)$  <  $O(n!)$  <  $O(n^n)$
- ... <  $(n!)$  <  $O(2^n)$  <  $O(n^n)$
- $O(1)$  <  $O(\lg n)$  <  $O(\lg \lg n)$  < ...