Algorítmica

Prof. Dr. Ángel Carmona Poyato

Departamento de Informática y Análisis Numérico
Escuela Politécnica Superior de Córdoba
Universidad de Córdoba

Tema 1. Introducción a la algorítmica

Tema 1. Introducción a la algorítmica

Competencias

- CB5. Que los estudiantes hayan desarrollado las habilidades de aprendizaje necesarias para emprender estudios posteriores con un alto grado de autonomía.
- CTEC1. Capacidad para tener un conocimiento profundo de los principios fundamentales y modelos de la computación y saberlos aplicar para interpretar, seleccionar, valorar, modelar, y crear nuevos conceptos, teorías, usos y desarrollos tecnológicos relacionados con la informática.
- CTEC3. Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de

Tema 1. Introducción a la algorítmica

Objetivos del tema

- Relacionar la asignatura con otras asignaturas.
- Plantear los objetivos de la asignatura.
- Definir algoritmo y algorítmica.
- Definir eficiencia y complejidad de un algoritmo.
- Estudio de los factores que influyen en la complejidad temporal de un algoritmo.
- Analizar los distintos enfoques para evaluar la eficiencia de un algoritmo.

Tema 1. Introducción a la algorítmica

Introducción

- Conocimientos adquiridos en las asignaturas de programación
  - Codificar algoritmos partiendo de su implementación.
  - Seleccionar estructuras de datos eficientes.
  - Diseñar e implementar algoritmos.
- Mecanismo usado hasta ahora para resolver problemas.
- Objetivos de la algorítmica.

Tema 1. Introducción a la algorítmica

Definiciones y conceptos

- Definición de algoritmo.
  - Precisión en las reglas.
  - Precisión en la respuesta.
  - Algoritmos aproximados y Algoritmos heurísticos.
- Definición de algorítmica.
  - Factores en la selección del algoritmo adecuado.
    - Los límites de memoria.
    - La velocidad del equipo disponible.
    - El tiempo empleado por el algoritmo.
    - Facilidad de implementación del algoritmo.
    - Tamaño del ejemplar del problema que queramos resolver.
- Ejemplares de un problema.
- Dominio de definición de un problema.

Tema 1. Introducción a la algorítmica

Eficiencia o complejidad de un algoritmo

- Cálculo de la eficiencia o complejidad.
  - Recursos a tener en cuenta.
    - Espacio de memoria (Complejidad espacial)
    - Tiempo de ejecución (Complejidad temporal).
  - Tiempo y espacio: objetivos contrapuestos.
  - Complejidad temporal para evaluar la eficiencia.

Tema 1. Introducción a la algorítmica

Factores que influyen en la complejidad temporal

- Tamaño de los datos de entrada.
- Contenido de los datos de entrada.
  - Caso medio.
  - Caso peor.
- Computador y código generado por el compilador.
  - Estudio independiente de máquina y lenguaje.
  - Principio de invariancia.

Tema 1. Introducción a la algorítmica

Enfoques en la evaluación de la eficiencia algorítmica

- Empírico o a posteriori:
  - Se hacen pruebas y se compara después de la implementación.
- Teórico o a priori:
  - Se determina matemáticamente en función del tamaño de los datos.
- Híbrido
  - Se determina teóricamente la función y después se ajustan parámetros para un programa y computador concretos.

Tema 1. Introducción a la algorítmica

Operaciones elementales

- Definición.
- Dichas operaciones y su coste definen un modelo de computación.
- Importancia del número de ellas y no del tiempo de cada una.
- Se determina teóricamente la función y después se ajustan parámetros para un programa y computador concretos.
- Tiempo unitario.
- Ejemplo:

$$t \leq s * t_s + m * t_m + a * t_a \leq max(t_s, t_m, t_a) * (s + m + a) \quad (1)$$

Tema 2. Notación asintótica

Tema 2. Notación asintótica

Competencias

- CTEC1. Capacidad para tener un conocimiento profundo de los principios fundamentales y modelos de la computación y saberlos aplicar para interpretar, seleccionar, valorar, modelar, y crear nuevos conceptos, teorías, usos y desarrollos tecnológicos relacionados con la informática.
- CTEC3. Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

Tema 2. Notación asintótica

Objetivos del tema

- Definición de análisis asintótico.
- Notación orden de $f(n)$ ($O()$).
- Notación $\Omega()$.
- Notación $\Theta()$

Tema 2. Notación asintótica

Introducción

- El cálculo se realizará en función del tamaño del problema.
- Si f es la función que indica cuanto tarda en ejecutarse un algoritmo, entonces $f(n)$ es el tiempo que requiere cuando el tamaño del problema es n.
- La eficiencia de los algoritmos se estudia por medio de $f(n)$, usando valores de n suficientemente grandes.
- Notaciones:
  - Notación orden de $f(n)$ ($O()$): da una cota superior.
  - Notación $\Omega()$: da una cota inferior.
  - Notación $\Theta()$: establece una cota inferior y superior.

Tema 2. Notación asintótica

Notación orden de $f(n)$ ($O()$)

- Sea $f : N \to R^+ \cup \{0\}$. El conjunto de las funciones del orden de $f(n)$, denotado por $O(f(n))$, se define como sigue:

$$O(f(n)) = \{g : N \to R^+ \cup \{0\} \mid \exists c \in R^+, n_0 \in N, \forall n \geq n_0 \; g(n) \leq cf(n)\}$$

  Se dice que g es del orden de $f(n)$ cuando $g \in O(f(n))$

- g es del orden de $f(n)$ si $g(n)$ está acotada superiormente por un múltiplo real positivo de $f(n)$ para todo n suficientemente grande.
- La definición de $O(f(n))$ garantiza el principio de invariancia.
- $O(f(n))$ define un orden de complejidad cuyo representante es la función $f(n)$ más sencilla posible dentro del mismo.

Tema 2. Notación asintótica

Notación $\Omega()$

- la notación $O(f(n))$ solo proporciona cotas superiores
- Sea $f : N \rightarrow R^+ \cup \{0\}$. El conjunto de las funciones $\Omega(f(n))$, leído *omega de* $f(n)$, se define como:

$$\Omega(f(n)) = \{g : N \rightarrow R^+ \cup \{0\} \mid \exists c \in R^+, n_0 \in N, \forall n \geq n_0 \; g(n) \geq cf(n)\}$$

- La notación $\Omega(f(n))$ da una cota inferior respecto del tiempo de ejecución de un algoritmo.

Tema 2. Notación asintótica

Notación $\Omega()$ (II)

- g está en Omega de $f(n)$ si $g(n)$ está acotada inferiormente por un múltiplo real positivo de $f(n)$ para todo n suficientemente grande.
- Asimetría entre las medidas $O(f(n))$ y $\Omega(f(n))$
- Regla de la dualidad: $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$

Tema 2. Notación asintótica

Notación $\Theta()$.

- Acota el tiempo de ejecución de un algoritmo tanto por encima como por debajo.
- Sea $f : N \rightarrow R^+ \cup \{0\}$. El conjunto de las funciones $\Theta(f(n))$, leído *del orden exacto de $f(n)$*, se define como:

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

- Esta definición es equivalente a

$$\Theta(f(n)) = \{g : N \rightarrow R^+ \cup \{0\} | \exists c, d \in R^+, n_0 \in N, \forall n \geq n_0 \\ df(n) \leq g(n) \leq cf(n)\}$$

Tema 2. Notación asintótica

Notación $\Theta()$ (II)

- La notación $\Theta()$ es más precisa que las notaciones ya que $g(n)$ *está en el orden exacto de* $f(n)$ ($g(n) \in \Theta(f(n))$) si y sólo si $f(n)$ es a la vez una cota inferior y superior de $g(n)$.

Tema 2. Notación asintótica

Notación asintótica con varios parámetros

- El tiempo de ejecución de un algoritmo puede depender de más de un parámetro del caso en cuestión.
- Sea $f : NxN \rightarrow R^+ \cup \{0\}$ una función $f(m, n)$ de parejas de números naturales en los reales no negativos.
- Se dice que $g(m, n)$ es del orden de $f(m, n)$ si está acotada superiormente por un múltiplo positivo de $f(m, n)$

$$O(f(m,n)) = \{g : NxN \rightarrow R^+ \cup \{0\} \mid \exists c \in R^+, m_0 \in N, \forall m \geq m_0, \forall n \geq m_0 \; g(m,n) \leq cf(m,n)\}$$

Tema 2. Notación asintótica

Propiedades de la notación $O()$.

1.- Propiedad reflexiva: $\forall f : N \rightarrow R^{+} \cup \{0\}$, $f(n) \in O(f(n))$

2.- Propiedad transitiva: $\forall f, g, h : N \rightarrow R^{+} \cup \{0\}$, si $f(n) \in O(g(n))$ y $g(n) \in O(h(n))$, entonces $f(n) \in O(h(n))$

3.- Si $g(n) \in O(f(n))$ y $f(n) \in O(g(n))$, entonces $O(f(n)) = O(g(n))$.

4.- $\forall c \in R^{+}$ $O(c\ f(n)) = O(f(n))$

Tema 2. Notación asintótica

Propiedades de la notación $O()$ (II).

5.- Si $a, b > 1$, entonces $O(log_a n) = O(log_b n)$

6.- La regla del máximo (o de la suma):
$O(f(n) + g(n)) = O(máximo(f(n), g(n)))$

Tema 2. Notación asintótica

Propiedades de la notación $O()$ (III).

7.- La regla del límite: $\forall f, g : N \rightarrow R^{+} \cup \{0\}$

1. si $\lim_{n\rightarrow\infty} \frac{f(n)}{g(n)} \in R^{+}$ entonces $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$
2. si $\lim_{n\rightarrow\infty} \frac{f(n)}{g(n)} = 0$ entonces $f(n) \in O(g(n))$ y $g(n) \notin O(f(n))$
3. si $\lim_{n\rightarrow\infty} \frac{f(n)}{g(n)} = +\infty$ entonces $f(n) \notin O(g(n))$ y $g(n) \in O(f(n))$

Tema 2. Notación asintótica

Jerarquía de los órdenes de complejidad.

- $O(1)$: complejidad constante.
- $O(logn)$: complejidad logarítmica.
- $O(n)$: complejidad lineal.
- $O(nlogn)$: complejidad n-logarítmica.
- $O(n^2)$: complejidad cuadrática.

Tema 2. Notación asintótica

Jerarquía de los órdenes de complejidad. (II)

- $O(n^3)$: complejidad cúbica.
- $O(n^k)$: complejidad polinómica.
- $O(2^n)$: complejidad exponencial.
- $O(n!)$: complejidad factorial.

Tema 2. Notación asintótica

Consideraciones importantes.

- El hecho de que un algoritmo tenga una eficiencia que sea buena en términos generales no implica que lo sea en términos particulares.
- Si se comparan las eficiencias de dos algoritmos, se ha de tener en cuenta que el cálculo del orden de complejidad es una medida asintótica.

Tema 3. Análisis de algoritmos

Tema 3. Análisis de algoritmos

Competencias

- CTEC3. Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

Tema 3. Análisis de algoritmos

Objetivos del tema

- Se pretende seleccionar el algoritmo más adecuado.
- Estudio de las técnicas básicas:
  - Estructuras de control.
  - Ecuaciones de recurrencia.

Tema 3. Análisis de algoritmos

Introducción

- El análisis de algoritmos permite determinar la eficiencia de un algoritmo, para seleccionar el más eficiente.
- No hay un método general.
- Estudio de las técnicas básicas:
  - Estructuras de control.
  - Ecuaciones de recurrencia.
- Se hace de lo particular a lo general.

Tema 3. Análisis de algoritmos

Algoritmos no recursivos

- Asignaciones y expresiones simples. El tiempo requerido para evaluar una constante, una expresión formada por términos simples o una asignación simple es $t(I) = c$, de donde $O(t(I)) = O(c) = O(1)$.
- Secuencia de instrucciones. El tiempo de ejecución de una secuencia de instrucciones es igual a la suma de sus tiempos de ejecución respectivos.

$$t(I_1; I_2) = t(I_1) + t(I_2)$$

Aplicando la regla de la suma, su orden es

$$O(t(I_1; I_2)) = O(t(I_1) + t(I_2)) = O(max(t(I_1), t(I_2)))$$

Tema 3. Análisis de algoritmos

Algoritmos no recursivos (II)

- Instrucciones condicionales.

$$t(si\ \ entonces\ \ finsi) = t(condición) + t(consecuente)$$

$$O(t(si\ \ entonces\ \ finsi)) = O(t(cond.) + t(cons.)) =$$
$$O(max(t(cond.), t(cons.)))$$

$$t(si\ \ entonces\ \ sino\ \ finsi) = t(cond.) + max(t(cons.), t(alt.))$$

$$O(t(si\ \ entonces\ \ sino\ \ finsi)) =$$
$$O(t(cond.) + max(t(cons.), t(alt.)) =$$
$$O(max(t(cond.), max(t(cons.), t(alt.))))$$

Tema 3. Análisis de algoritmos

Algoritmos no recursivos (III)

- Instrucciones de iteración
  - para finpara es el producto del número de iteraciones por la complejidad de instrucciones del cuerpo del bucle.
  - mientras finmientras y repetir hasta que se tiene en cuenta el caso más desfavorable.
- Llamadas subprogramas

$$t(SP(f_1, ..., f_n)) = \sum_{i=1}^{n} t(f_i) + t(cuerpo)$$

$$O(t(SP(f_1, ..., f_n))) = O(\sum_{i=1}^{n} t(f_i) + t(cuerpo)) =$$

$$O(max(\sum_{i=1}^{n} t(f_i), t(cuerpo)))$$

Tema 3. Análisis de algoritmos

Algoritmos no recursivos (IV)

- Ejemplos. (Ver apuntes de teoría)
  - Ejemplos simples.
  - Ordenación por inserción.
  - Ordenación burbuja.

Tema 3. Análisis de algoritmos

Algoritmos recursivos. Ejemplos. (Ver apuntes de teoría)

- Métodos
  - Método de expansión de recurrencias: sustituir las recurrencias por su igualdad hasta llegar a cierta $t(n_0)$ conocida.
  - Método de acotación: elegir dos funciones $f(n)$ y $g(n)$ que sean, respectivamente, *unas cotas superior e inferior* del mismo orden y usar la ecuación de recurrencia para probar que
  $$\forall n \in N \; g(n) \leq \; t(n) \leq \; f(n).$$
  - Método de la ecuación característica.
  - Conjeturar una solución $f(n)$ y usar la recurrencia para demostrar que
  $$t(n) \leq f(n).$$

Tema 3. Análisis de algoritmos

Series recurrentes

- **Progresiones aritméticas**

  Si $a_n = a_{n-1} + c; \ \sum_{i=1}^{n} a_i = \frac{n(a_1+a_n)}{2}$

- **Progresiones geométricas**

  Si $a_n = ra_{n-1}; \ \sum_{i=1}^{n} a_i = \frac{a_1+a_n r^n}{1-r} \quad (r \neq 1)$

  $\sum_{i=1}^{\infty} a_i = \frac{a_1}{1-r} \quad (0 < r < 1)$

- **Suma de cuadrados**

  $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

Tema 4. Recursividad

Tema 4. Recursividad

Competencias

- CTEC3. Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

Tema 4. Recursividad

Objetivos del tema

- Estudiar la recursividad de una forma más rigurosa y exhaustiva.
- Ventajas e inconvenientes de la recursividad.
- Problemas que surgen de los algoritmos recursivos.
- Forma de solucionar las llamadas repetidas.
- Ejemplos ilustrativos.

Tema 4. Recursividad

Introducción

- La recursividad se basa en la repetición anidada de ciertos procedimientos.
- Es aplicable a problemas que puedan ser reducidos a problemas similares pero para un conjunto más reducido de datos.
- Programando dicha reducción se resolverá el problema.
  - Caso particular: caso elemental cuya solución se conoce. $factorial(1) = 1$.
  - Caso general: es el caso de reducción general. $factorial(n) = n * factorial(n-1)$
- La descomposición origina subproblemas tan simples cuya solución es obvia.

Tema 4. Recursividad

Ventajas e inconvenientes de la recursividad

- Las llamadas recursivas se realizan secuencialmente haciendo uso de la pila.
- Traducción automática de recursivo a iterativo.
- Versión iterativa más rápida, pero:
  - A veces la recursiva es más simple de obtener.
  - Tiempo en el proceso de traducción recursiva a iterativa.
- Principal inconveniente: repetición de llamadas.
- Almacenar las llamadas y el valor devuelto en una estructura de datos para evitar su repetición.
- Ejemplo de la sucesión de Fibonacci.

Tema 4. Recursividad

Ventajas e inconvenientes de la recursividad (II)

- Conclusiones:
  - Más tiempo y más uso de la memoria.
  - Casos en que la solución recursiva es más “natural” (más de una llamada recursiva).
  - La traducción a iterativo consume tiempo y además no siempre es fácil.
  - Traducir sólo cuando el iterativo sea más eficiente.

.

Tema 4. Recursividad

Ejemplo (Juego de la rayuela)

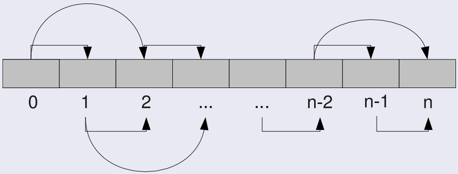


Figura : Juego de la rayuela

- Caso elemental: $Caminos(n) = n \; si \; n <= 2$
- Caso general: $Caminos(n) = Caminos(n-1) + Caminos(n-2)$

Tema 4. Recursividad

```
Función caminos(n; - ; -)
```

inicio

  si $(n \leq 2)$ **entonces**

    devolver 2

  sino

    devolver $caminos(n-1) + caminos(n-2)$

  finsi

fin

Tema 4. Recursividad

Ejemplo (Plano de la ciudad)

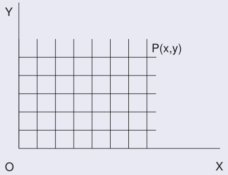


Figura : Plano de la ciudad

- Caso elemental: $Plano(x, y) = 1 \ si \ x = 0 \ o \ y = 0$
- Caso general: $Plano(x, y) = Plano(x, y - 1) + Plano(x - 1, y)$

Tema 4. Recursividad

```
Función Plano(x, y; - ; -)
inicio
    si (x = 0) entonces
        devolver 1
    sino
        si (y = 0) entonces
            devolver 1
        sino
            devolver Plano(x − 1, y) + Plano(x, y − 1)
        finsi
    finsi
fin
```

Tema 4. Recursividad

```
Algoritmo Hanoi(m, i, j; - ; -)
```

inicio

si $(m > 0)$ **entonces**

$Hanoi(m-1, i, 6-i-j)$

escribir $i \rightarrow j$

$Hanoi(m-1, 6-i-j, j)$

finsi

fin

Tema 4. Recursividad

Ejemplo (Torres de Hanoi)

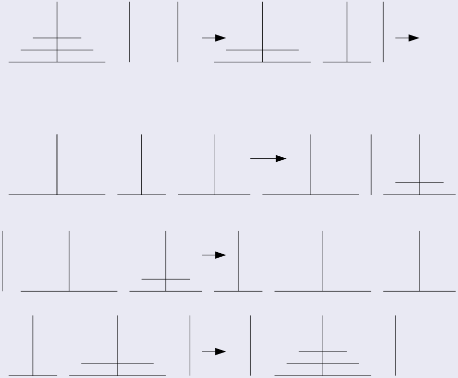


Figura : Solución de torres de Hanoi con tres discos

Tema 5. Divide y vencerás

Tema 5. Divide y vencerás

Competencias

- CTEC3. Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

Tema 5. Divide y vencerás

Objetivos del tema

- Estudiar la técnica “Divide y Vencerás”.
- Si el enemigo es demasiado fuerte, divídelo.
- Planteamiento general.
- Algoritmo recursivo genérico como solución.
- Ejemplos ilustrativos.

Tema 5. Divide y vencerás

Introducción

- Si el enemigo es demasiado fuerte para tí, divídelo.
- Descompone el problema en subproblemas que son a su vez subcasos de dicho problema.
- Resolviendo todos los subproblemas, y combinando las soluciones de los subproblemas se obtiene la solución del problema original.
- Si los subproblemas obtenidos no son lo suficientemente pequeños para resolverlos, se vuelven a dividir.
- Los subproblemas resultantes suelen ser del mismo tipo que el original, lo cual favorece un diseño recursivo para la solución del problema.

Tema 5. Divide y vencerás

El método general

- Se basa en la división de los n datos de entrada en k subconjuntos distintos, donde $1 < k \leq n$, obteniéndose k subproblemas, del mismo carácter que el original.

`Algoritmo DyV(A,p,q)`

inicio

  si $Pequeno(A, p, q) = cierto$ **entonces**

    devolver $Solucion(A, p, q)$

  sino

    $m \leftarrow dividir(A, p, q)$

    devolver $COMBINA(DyV(A, p, m), DyV(A, m + 1, q))$

  finsi

fin

Tema 5. Divide y vencerás

El método general (II)

- Para que esta técnica merezca la pena aplicarla es necesario que se cumplan tres condiciones:
  - Se ha de seleccionar de una forma conveniente cuando usar el algoritmo básico en vez de seguir descomponiendo el problema.
  - Ha de ser posible la descomposición del problema en subproblemas y recomponer las soluciones a dichos problemas de una manera eficiente.
  - Los subproblemas han de ser aproximadamente del mismo tamaño.

Tema 5. Divide y vencerás

Búsqueda binaria

- Se trata de buscar un elemento de valor x, en un vector n elementos en orden creciente $v(n)$.
- Problema original $P(n, v(1), v(2)..., v(n), x)$ se divide en tres subproblemas:
  - $SP_1(k-1, v(1), ..., v(k-1), x)$
  - $SP_2(1, v(k), x)$
  - $SP_3(n-k, v(k+1), ..., v(n), x)$
- La solución de 2 de los 3 subproblema es inmediata.

Tema 5. Divide y vencerás

```
Algoritmo BusquedaBinaria(v, n, x; ; )
inicio
    izq ← 1
    der ← n
    mientras izq ≤ der hacer
        k ← (izq+der)/2
        si x < v(k) entonces
            der ← k − 1
        sino
            si x > v(k) entonces
                iz ← k + 1
            sino
                j ← k
                devolver j
            finsi
        finsi
    finmientras
    j ← 0
    devolver j

fin
```

Tema 5. Divide y vencerás

Máximo y mínimo elemento de un vector

- El algoritmo hace $2(n-1)$ comparaciones.

Algoritmo $MaximoMinimo1(v, n; ; maximo, minimo)$
inicio
    $maximo \leftarrow v(1)$
    $minimo \leftarrow v(1)$
    para i **de** 2 **a** n **hacer**
        si $v(i) > maximo$ **entonces**
            $maximo \leftarrow v(i)$
        finsi
        si $v(i) < minimo$ **entonces**
            $minimo \leftarrow v(i)$
        finsi
    finpara
fin

Tema 5. Divide y vencerás

Máximo y mínimo elemento de un vector(II)

- El algoritmo hace $3n/2 - 3/2)$ comparaciones por término medio.

Algoritmo *MaximoMinimo*2(v, n; ; *maximo*, *minimo*)
inicio
    maximo $\leftarrow v(1)$
    minimo $\leftarrow v(1)$
    para i **de** 2 **a** n **hacer**
        si $v(i) >$ *maximo* **entonces**
            maximo $\leftarrow v(i)$
        sino
            si $v(i) <$ *minimo* **entonces**
                minimo $\leftarrow v(i)$
            finsi
        finsi
    finpara
fin

Tema 5. Divide y vencerás

Máximo y mínimo elemento de un vector(III)

- Problema inicial $P(n, v(1), v(2), ..., v(n))$ y se divide en dos:
  - $SP_1(n/2, v(1), v(2), ..., v(n/2))$
  - $SP_2(n - n/2, v(n/2+1), v(n/2+2), ..., v(n))$
- $maximo(P) = maximo(maximo(SP_1), maximo(SP2))$
- $minimo(P) = minimo(minimo(SP_1), minimo(SP2))$

Tema 5. Divide y vencerás

`Siempre` $2n{-}2$ `comparaciones.`

Algoritmo *MaximoMinimo*$(v, n, i, j; ; maximo, minimo)$
inicio
  si $i = j$ **entonces**
    $maximo \leftarrow v(i); minimo \leftarrow v(i)$
  sino
    si $i = j - 1$ **entonces**
      si $v(i) < v(j)$ **entonces**
        $maximo \leftarrow v(j); minimo \leftarrow v(i)$
      sino
        $maximo \leftarrow v(i); minimo \leftarrow v(j)$
      finsi
    sino
      $mitad \leftarrow \frac{i+j}{2}$
      $MaximoMinimo(v, n, i, mitad; ; maximo1, minimo1)$
      $MaximoMinimo(v, n, mitad + 1, j; ; maximo2, minimo2)$
      $maximo = Maximo(maximo1, maximo2), minimo = Minimo(minimo1, minimo2)$
    finsi
  finsi

fin

Tema 5. Divide y vencerás

Algoritmo $QuickSort(iz, de, n; v;)$

inicio

$i \leftarrow iz, j \leftarrow de, x \leftarrow v(E(\frac{iz+de}{2}))$

repetir

mientras $v(i) < x$ **hacer** $i \leftarrow i + 1$ **finmientras**

mientras $v(j) > x$ **hacer** $j \leftarrow j - 1$ **finmientras**

si $i \leq j$ **entonces**

$aux \leftarrow v(i), v(i) \leftarrow v(j), v(j) \leftarrow aux, i \leftarrow i + 1, j \leftarrow j - 1$

finsi

hasta que $i > j$

si $iz < j$ **entonces** *El izquierdo tiene mas de un elemento*

$QuickSort(iz, j, n, v)$

finsi

si $i < de$ **entonces** *El derecho tiene mas de un elemento*

$QuickSort(i, de, n, v)$

finsi

fin

Tema 5. Divide y vencerás

```
Algoritmo KMenores(iz, de, n, k; v; )
inicio
    Particion(iz, de, n; ; i)
     si k = i − 1 entonces problema resuelto
        escribir los k primeros
      sino
        si k < i − 1 entonces
            KMenores(iz, j, n, k, v)
        sino
            KMenores(i, de, n, k, v)
        finsi
    finsi
fin
```

Tema 5. Divide y vencerás

Ejemplo (Aritmética de los enteros grandes)

Figura : Descomposición de los operandos

- $u = 10^s w + x$
- $v = 10^s y + z$
- $uv = 10^{2s} wy + 10^s (wz + xy) + xz$

Tema 5. Divide y vencerás

Algoritmo $MultEG1(u, v)$

inicio

$n \leftarrow MayorMagnitud(u, v)$

si $pequeno(n) = cierto$ **entonces**

devolver $u * v$ $(producto\ de\ u\ y\ v\ por\ el\ algoritmo\ clasico)$

sino

$s \leftarrow \frac{n}{2}$

$w \leftarrow \frac{u}{10^s}$

$x \leftarrow u\ mod\ 10^s$

$y \leftarrow \frac{v}{10^s}$

$z \leftarrow v\ mod\ 10^s$

devolver $MultEG1(w, y) * 10^{2s} + (MultEG1(w, z) + MultEG1(x, y)) * 10^s + MultEG1(x, z)$

finsi

fin

- `Es de orden cuadrático.`

Tema 5. Divide y vencerás

Algoritmo $MultEG2(u, v)$

inicio

$n \leftarrow MayorMagnitud(u, v)$

si $pequeno(n) = cierto$ **entonces**

devolver $u * v$ $(producto\ de\ u\ y\ v\ por\ el\ algoritmo\ clasico)$

sino

$s \leftarrow \frac{n}{2}, w \leftarrow \frac{u}{10^s}$

$x \leftarrow u\ mod\ 10^s, y \leftarrow \frac{v}{10^s}$

$z \leftarrow v\ mod\ 10^s, r \leftarrow MultEG2(w + x, y + z)$

$p \leftarrow MultEG2(w, y), q \leftarrow MultiplicarEG2(x, z)$

devolver $10^{2s} * p + 10^s * (r - p - q) + q$

finsi

fin

- Es de orden $n^{log 3}$.

Tema 5. Divide y vencerás

Ordenación por fusión

- Se basa en la fusión de dos subvectores ordenados.

Algoritmo *fusionar*$(u(m+1), v(n+1), t(m+n))$
inicio
    $i \leftarrow 1, j \leftarrow 1, u(m+1) \leftarrow \infty, v(n+1) \leftarrow \infty$
    para k **de** 1 **a** $m+n$ **hacer**
        si $u(i) < v(j)$ **entonces**
            $t(k) \leftarrow u(i), i \leftarrow i+1$
        sino
            $t(k) \leftarrow v(j), j \leftarrow j+1$
        finsi
    finpara
fin

Tema 5. Divide y vencerás

Algoritmo *OrdenarFusion*$(t(n))$
inicio
  si *pequeno*$(n) =$ *cierto* **entonces** `/*Ordena usando un metodo no sofisticado */`
    ordenar(t)
  sino
    Crear vector u con $\frac{n}{2}+1$ *elementos*
    Crear vector v con $n-\frac{n}{2}+1$ *elementos*
    para i **de** 1 **a** $\frac{n}{2}$ **hacer**
      $u(i) \leftarrow t(i)$
    finpara
    para i **de** $1+\frac{n}{2}$ **a** n **hacer**
      $v(i-\frac{n}{2}) \leftarrow t(i)$
    finpara
    OrdenarFusion(u)
    OrdenarFusion(v)
    fusionar(u, v, t)
  finsi
fin

- Es de orden $nlog(n)$.

Tema 5. Divide y vencerás

Exponenciación

Algoritmo *Exponenciacion*(a, n; ; *resultado*)

inicio

resultado $\leftarrow a$

para i **de** 1 **a** $n-1$ **hacer**

resultado $\leftarrow a *$ *resultado*

finpara

fin

- Es de orden n.
- Para usar divide y vencerás podemos usar las siguientes consideraciones:
  - Si n es par $a^n = (a^{n/2})^2$
  - Si n es impar $a^n = a(a^{(n-1)/2})^2$

Tema 5. Divide y vencerás

Algoritmo *ExponenciacionDyV*$(a, n; ;)$

inicio

    si $n = 1$ **entonces**

        devolver a

    sino

        si $n \ mod \ 2 = 0$ **entonces** $\mathit{n\ es\ par}$

            $aux \leftarrow ExponenciacionDyV(a, n/2)$

            devolver $aux * aux$

        sino

            devolver $a * ExponenciacionDyV(a, n-1)$

        finsi

    finsi

fin

- El tiempo de ejecución del algoritmo sería $O(logn)$.

Tema 5. Divide y vencerás

- Se puede plantear una versión iterativa de este algoritmo, descomponiendo a^n en productos de potencias de a cuyos exponentes sean potencias de 2. $a^{61} = a^{32}a^{16}a^8a^4a^1$.

Algoritmo *ExponenciacionDyVIterativo*$(a, n; ; r)$

inicio

$i \leftarrow n, x \leftarrow a, r \leftarrow 1$

mientras $i > 0$ **hacer**

si $i \ mod \ 2 = 1$ **entonces** $i \ es \ impar$

$r \leftarrow rx$

finsi

$x \leftarrow x^2$

$i \leftarrow \frac{i}{2}$

finmientras

fin

Tema 6. Algoritmos voraces

Tema 6. Algoritmos voraces

Competencias

- CTEC3. Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

Tema 6. Algoritmos voraces

Objetivos del tema

- Estudiar el método de los Algoritmos Voraces (método greedy o método devorador).
- Aplicabilidad del método en problemas de optimización cuando la solución se puede obtener a trozos.
- Hay que demostrar cuando la solución obtenida es óptima (no siempre lo es).
- No hay vuelta atrás cuando se tiene un trozo de solución.
- A veces se usan para obtener soluciones subóptimas.
- Ejemplos ilustrativos.

Tema 6. Algoritmos voraces

Introducción

- Mantiene la idea de la división pero ahora se divide la solución.
- Se aplica en problemas cuya solución se puede obtener a trozos.
- Cada trozo se obtiene buscando el óptimo entre los trozos aún no seleccionados.
- Se suele aplicar en problemas de optimización.
- Cada vez que se selecciona un trozo, éste ya es definitivo sin verse afectado por lo que ocurra después.
- Ejemplo del problema del cambio, resaltando que no es óptimo.

Tema 6. Algoritmos voraces

Algoritmo *cambio*(n)

inicio

$C = \{100, 50, 20, 10, 5, 2, 1\}$

$S \leftarrow \phi$ *Solucion*

$s \leftarrow 0$ *suma parcial*

mientras $s \neq n$ **hacer**

$x \leftarrow maximo(C)$ *tal que* $s + x \leq n$

si *existe*(x) **entonces**

$S \leftarrow S \cup \{x\}$

$s \leftarrow s + x$

sino

devolver *no encuentro solucion*

finsi

finmientras

fin

Tema 6. Algoritmos voraces

El método general

- Problema de optimización en el que la solución se construye partiendo de un conjunto de candidatos (monedas).
- Se usan dos conjuntos:
  - El primero contiene candidatos evaluados y seleccionados.
  - El segundo contiene los evaluados y rechazados.
- Una función comprueba si los candidatos seleccionados hasta el momento constituyen una solución del problema. (Las monedas seleccionadas ya son iguales a la cantidad que se quiere conseguir).
- Otra función comprueba si hay candidatos puede mejorar la solución haciendo crecer el conjunto. (Ver monedas que se pueden seleccionar).

Tema 6. Algoritmos voraces

El método general(II)

- Función de selección del mejor de los candidatos que pueden mejorar la solución (seleccionar la moneda de más valor dentro de las seleccionables).
- Función objetivo a optimizar que proporciona el valor de la solución encontrada (número de monedas empleadas en el cambio).
- Tiempo de ejecución $O(n^2)$ u $O(n^3)$:
  - n candidatos (se reducen en 1 en cada iteración).
  - Función de selección y objetivo constante o lineal.
  - Número de candidatos que forman la solución.

Tema 6. Algoritmos voraces

Algoritmo *voraz*(C)

inicio

$S \leftarrow \phi$ *Solucion*

mientras $C \neq \phi$ **y no** *solucion*(S) **hacer**

$x \leftarrow$ *seleccionar*(C)

$C \leftarrow C - \{x\}$

si *viable*$(S \cup \{x\})$ **entonces**

$S \leftarrow S \cup \{x\}$

finsi

finmientras

si *solucion*(S) **entonces**

devolver S

sino

devolver *No hay solucion*

finsi

fin

Tema 6. Algoritmos voraces

Elementos de un algoritmo voraz en el caso del problema del cambio

- Los candidatos son el conjunto de monedas disponibles, suponiendo que no hay límite para ninguna de ellas.
- La función de solución comprueba si el valor de las monedas seleccionadas es igual al valor que hay que conseguir.
- Un conjunto de monedas es viable si no sobrepasa la cantidad buscada.
- La función de selección elige la moneda de más valor que quede en el conjunto de candidatos.
- La función objetivo contabiliza el número de monedas usadas en la selección.

Tema 6. Algoritmos voraces

El problema de la mochila

- Problema:
  - Mochila de volumen V.
  - Materiales divisibles m_i de volumen v_i y de precio unitario p_i
  - Llenar la mochila con máximo coste.
- Solución:
  - Llenar la mochila comenzando con el material de mayor precio y cuando se agota éste, si queda volumen disponible, seleccionar el de siguiente mayor precio, y así hasta que se llene la mochila.
  - El máximo se puede seleccionar en cada iteración $O(kn)$, o se pueden ordenar los materiales según el precio $O(nlogn)$.

Tema 6. Algoritmos voraces

Algoritmo *Mochila*$(n, V; D;)$
inicio
  $resto \leftarrow V$
  para i **de** 1 **a** n **hacer**
    $D(i).usado \leftarrow$ "*nada*"
  finpara
  repetir
    $precioMaximo \leftarrow 0$, $materialMaximo \leftarrow 0$, $materialDisponible \leftarrow falso$
    para i **de** 1 **a** n **hacer**
      si $D(i).usado =$ "*nada*" **entonces**
        $materialDisponible \leftarrow cierto$
        si $D(i).precio > precioMaximo$ **entonces**
          $precioMaximo \leftarrow D(i).precio$
          $materialMaximo \leftarrow i$
        finsi
      finsi

    finpara

Tema 6. Algoritmos voraces

Comprobamos si el material de maximo coste cabe en la mochila

si $materialDisponible = cierto$ **entonces**

  si $resto \geq D(materialMaximo).volumen$ **entonces**

    $D(materialMaximo).usado \leftarrow$ "*total*"

    $resto \leftarrow resto - D(materialMaximo).volumen$

  sino

    $D(materialMaximo).usado \leftarrow$ "*parcial*"

    $resto \leftarrow 0$

  finsi

finsi

hasta que $resto = 0$ **o** $materialDisponible = falso$

fin

Tema 6. Algoritmos voraces

Minimización del tiempo de espera

- Problema:
  - En un determinado servicio se han de atender a n clientes, y de antemano se conoce el tiempo t_i que se atiende a cada uno. ¿En qué orden deben ser atendidos los clientes para que la suma de los tiempos de todos los clientes que están en el servicio (tiempo de espera y tiempo de atención) sea mínima?.
- Solución:
  - La estrategia voraz a seguir consiste en atender en cada paso al cliente no atendido con menor tiempo de atención.

Tema 6. Algoritmos voraces

Minimización del tiempo de espera (II)

- Demostración:
  - Supongamos que un algoritmo va construyendo la secuencia óptima paso a paso.
  - Después de haber calculado la secuencia óptima $(i_1, i_2, ..., i_m)$ para los m primeros clientes, supongamos que se añade a la misma el cliente j, con tiempo de atención t_j. El crecimiento del tiempo total en el servicio T será:
    $t_{i1} + t_{i2} + t_{i3} + \cdots + t_{im} + t_j$
  - Para minimizar este crecimiento, dado que un algoritmo voraz no reconsidera sus decisiones y los tiempos previos seleccionados ya no se pueden cambiar, lo único factible es el minimizar t_j
- Ejemplo de la cinta de casette.

Minimización del tiempo de espera (III)

- Ejemplo: Supongamos que se tienen tres clientes, y los tiempos de atención son $t_1 = 5, t_2 = 10, t_3 = 3$.
- Las posibilidades de atención son:
  - 1, 2, 3. El tiempo total en servicio sería: $5 + (5 + 10) + (5 + 10 + 3) = 38$.
  - 1, 3, 2. El tiempo total en servicio sería: $5 + (5 + 3) + (5 + 3 + 10) = 31$.
  - 2, 1, 3. El tiempo total en servicio sería: $10 + (10 + 5) + (10 + 5 + 3) = 43$.
  - 2, 3, 1. El tiempo total en servicio sería: $10 + (10 + 3) + (10 + 3 + 5) = 41$.
  - 3, 1, 2. El tiempo total en servicio sería: $3 + (3 + 5) + (3 + 5 + 10) = 29$.
  - 3, 2, 1. El tiempo total en servicio sería: $3 + (3 + 10) + (3 + 10 + 5) = 34$.
- La secuencia óptima es la $3, 1, 2$ cuyo tiempo total en el servicio es 29.

Tema 6. Algoritmos voraces

Planificación de tareas a plazo fijo

- Problema:
  - Se tienen n tareas (t_i) y cada una se realiza en una unidad de tiempo y solo genera beneficio (b_i) si se ejecuta antes de un plazo (p_i).
  - Calcular secuencia de tareas de más beneficio.

Tema 6. Algoritmos voraces

Planificación de tareas a plazo fijo(II)

- Un conjunto de tareas es factible si al menos una secuencia del conjunto se puede realizar en plazos.
- Se puede obtener una solución seleccionando en cada paso la tarea aún no seleccionada de mayor beneficio, siempre que la secuencia resultante sea factible.
- Un conjunto T de tareas es factible si y solo si la permutación de ese conjunto de tareas en orden creciente de plazos de ejecución también lo es.
- Se demuestra que la solución que obtiene una permutación ordenada en orden creciente de plazos es la solución óptima.

Tema 6. Algoritmos voraces

Algoritmo *secuencia*$(p, n; ; k, s)$

inicio

$p(0) \leftarrow 0$

$s(0) \leftarrow 0$

$k \leftarrow 1$

$s(1) \leftarrow 1$ `La tarea 1 se selecciona siempre`

para i **de** 2 **a** n **hacer** `en orden decreciente de los beneficios`

`busca tarea y prueba insertarla sin que las ya seleccionadas`

`queden fuera de plazo y salgan de la solucion`

$r \leftarrow k$ `almacena la última tarea seleccionada`

mientras $p(s(r)) > p(i)$ **y** $p(s(r)) \neq r$ **hacer**

`la primera parte del predicado busca la posición de inserción`

`la segunda comprueba si la tarea r, ya colocada, puede ser desplazada`

`sin violar plazos`

$r \leftarrow r - 1$

finmientras

Tema 6. Algoritmos voraces

```
Encuentra la posición de inserción comparando con las
ya seleccionadas e inserta ó inserta porque no se cumplirían plazos
```

si $p(s(r)) \leq p(i)$ **y** $p(i) > r$ **entonces**

```
la primera parte del predicado comprueba que ha encontrado
la posición de inserción
la segunda comprueba que la tarea se inserta sin violar su plazo
se inserta i en la posicion r+1
```

para j **de** k **a** $r+1$ **inc** -1 **hacer**

```
Desplaza una posición las tareas desplazables
```

$s(j+1) \leftarrow s(j)$

finpara

```
Inserta la tarea nueva en la posición r+1
```

$s(r+1) \leftarrow i$

```
Pasa a evaluar la siguiente tarea
```

$k \leftarrow k+1$

finsi

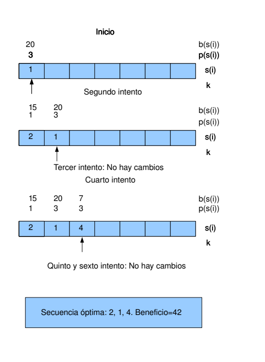
finpara

fin

Tema 6. Algoritmos voraces

Ejemplo (Planificación de tareas a plazo fijo)

- $n = 4$ *beneficio* $b_i = \{50, 10, 15, 30\}$ *plazos* $p_i = \{2, 1, 2, 1\}$



Tema 6. Algoritmos voraces

Algoritmo de Kruskal.

- Problema: Obtener el árbol abarcador de coste mínimo en un grafo conexo y no dirigido de n nodos.
- Solución:
  - Ordenar los lados de menor a mayor y seleccionar $n - 1$ lados en orden creciente siempre y cuando enlacen dos componentes conexas distintas.
  - Al finalizar hay una componente conexa que enlaza todos los nodos.

Tema 6. Algoritmos voraces

Algoritmo *Kruskal*(*GRAFOG*; ; *GRAFOL*)

inicio

ordenar(*CL*) `ordena crecientemente el conjunto de lados`

$L \leftarrow \phi$ `Inicialmente ningún lado forma parte de la solucion`

inicializar n conjuntos `Inicialmente hay tantos conjuntos como nodos`

repetir

$(u, v) \leftarrow$ `Lado mas corto no considerado`

$uconjunto \leftarrow buscar(u)$ `Conjunto al que pertenece nodo u`

$vconjunto \leftarrow buscar(v)$ `Conjunto al que pertenece nodo v`

si $uconjunto \neq vconjunto$ **entonces**

$fusionar(uconjunto, vconjunto)$ `Se fusionan los conjuntos de u y v`

$L \leftarrow L + (u, v)$ `El lado (u,v) se añade al grafo solucion`

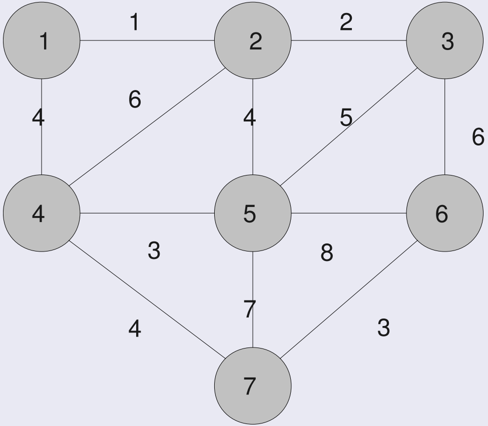
finsi

hasta que L *tenga* $n - 1$ *lados*.

fin

Tema 6. Algoritmos voraces

Ejemplo (Ejemplo algoritmo de Kruskal)



Tema 6. Algoritmos voraces

Viajante de comercio.

- Problema: Consiste en recorrer todos los nodos de un grafo conexo no dirigido, volviendo al nodo de partida y sin pasar dos veces por el mismo nodo, a un coste mínimo.
- Se puede usar un algoritmo voraz para obtener una solución aproximada.
- Se seleccionan *n* lados de forma que:
    - En orden creciente.
    - Sin formar ciclos (ciclo sólo al seleccionar el último).
    - Sin que más de dos lados confluyan en un nodo.
- Ver ejemplo de los apuntes.

Tema 7. Programación dinámica

Tema 7. Programación dinámica

Competencias

- CTEC3. Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

Tema 7. Programación dinámica

Objetivos del tema

- Estudiar el método de la Programación Dinámica.
- Dos enfoques:
  - No repetir dos veces lo mismo (Divide y vencerás).
  - Principio de optimalidad de Bellman.
- Produce soluciones óptimas (mejora de algunos algoritmos voraces).
- Algoritmos con alto orden de complejidad temporal.
- Ejemplos ilustrativos.

Tema 7. Programación dinámica

Introducción

- Dos enfoques:
  - No repetir dos veces lo mismo. (Ejemplo de Fibonacci).
  - Principio de optimalidad de Bellman. Toda subsolución de una solución óptima también es óptima
    - No siempre es aplicable. Ejemplos.
    - Dificultad en fijar subproblemas que cumplan el principio de optimalidad en algunos ejemplos.

Tema 7. Programación dinámica

El método general

- Se ha de cumplir que:
  - La solución del problema se pueda dividir en etapas.
  - La sucesión de subsoluciones óptimas es la solución óptima del problema.
- Solución hacia delante o hacia atrás.
  - $Dinamico(A, B) = Optimo((A, C_i) + Dinamico(C_i, B))$
  - $Dinamico(C_i, B) = Optimo((C_i, Dj) + Dinamico(D_j, B))$
- Planteamiento original produce llamadas recursivas repetidas.
- Se resuelven los problemas de $n + 1$ etapas basándonos en subproblemas resueltos de n etapas.

Tema 7. Programación dinámica

Competición internacional

- Dos equipos A y B se enfrentan. Gana el que consiga n victorias.
- Máximo de $2n - 1$ partidos.
- p es la probabilidad de que A gane un partido.
- $1 - p$ es la probabilidad de que gane B un partido.
- Calcular la probabilidad de que A gane la competición.
- Caso general : $P(i, j) = pP(i - 1, j) + (1 - p)P(i, j - 1)$
- Caso particular 1: $P(0, j) = 1$
- Caso particular 2: $P(i, 0) = 0$

Tema 7. Programación dinámica

Algoritmo $P(i, j, p)$
inicio
    si $i = 0$ **entonces**
        devolver 1
    sino
        si $j = 0$ **entonces**
            devolver 0
        sino
            devolver $pP(i-1, j, p) + (1-p)P(i, j-1, p)$
        finsi
    finsi
fin

- Orden $O(4^n)$.

Tema 7. Programación dinámica

Algoritmo *Competicion*$(p, n; ; P)$
inicio
    para s **de** `1` **a** `n` **hacer**
        $P(0, s) \leftarrow 1$
        $P(s, 0) \leftarrow 0$
    finpara
    para i **de** `1` **a** `n` **hacer**
        para j **de** `1` **a** `n` **hacer**
            $P(i, j) = pP(i-1, j) + (1-p)P(i, j-1)$
        finpara
    finpara
fin

- Orden $O(n^2)$.

Tema 7. Programación dinámica

Caminos mínimos entre todos los pares de nodos

- El principio se cumple, ya que si k es intermedio en el camino mínimo entre i y j, la parte del camino que va desde i a k y la que va de k a j también han de ser óptimas.
- Inicialmente se calculan las distancias directas (sin usar ningún nodo intermedio), entre todos los pares de nodos.
- En la iteración k, obtendremos las distancias más cortas usando única y exclusivamente los nodos $1, 2, 3, \ldots, k$.
  $D_k(i,j) = min\{D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j)\}$
- Al cabo de n iteraciones, en D contiene las distancias mínimas teniendo en cuenta todos nodos como intermedios.

Tema 7. Programación dinámica

Problema del cambio (versión 2)

- Tenemos n tipos de monedas y una cantidad N.
- Se genera una tabla c de $n(N+1)$ donde se reflejan los resultados de los subproblemas intermedios.
- El elemento $c(i,j)$ indica el número de monedas mínimo usando monedas que van de la 1 a la i para obtener la cantidad j.
- $c(i,j) = min\{c(i-1,j), 1 + c(i,j-v(i))\}$.
  - si $c(i,j) = c(i-1,j)$ no se selecciona la moneda i.
  - si $c(i,j) = 1 + c(i,j-v(i))$ se selecciona la moneda i.
- Aplica el principio de optimalidad ya que cuando se calcula $c(i,j)$ ya que $c(i-1,j)$ y $c(i,j-v(i))$ también son soluciones óptimas de los problemas que representan.

Tema 7. Programación dinámica

Ejemplo (Problema del cambio)

- *Cambio $N = 4$ monedas de valor $v_i = \{1, 4, 5\}$.*

Cantidad:	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
$v_1 = 1$	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
$v_2 = 4$	*0*	*1*	*2*	*3*	*1*	*2*	*3*	*4*	*2*
$v_3 = 5$	*0*	*1*	*2*	*3*	*1*	*1*	*2*	*3*	*2*

Cuadro : Tabla de secuencias y beneficios para el ejemplo.

Tema 7. Programación dinámica

Problema del cambio (III)

- Cualquier elemento de fuera de la tabla vale infinito.
- Obtención de los elementos de la tabla usando la fórmula.
- Para ver las monedas de cada tipo:
  - La solución son 2 monedas ya que $c(3,8) = 2$.
  - Como $c(3,8) = c(2,8)$ no se usan monedas del tipo 3.
  - Como $c(2,8) = 1 + c(2,4)$ se contabiliza una moneda del tipo 2.
  - Como $c(2,4) = 1 + c(2,0)$ se contabiliza otra moneda del tipo 2. Ya hemos llegado a un cambio de 0.
  - La solución serán dos monedas del tipo 2 (valor 4)

Tema 7. Programación dinámica

```
Algoritmo cambio2(v(n), N; ; C(n, N))
inicio
    para i de 1 a n hacer
        c(i, 0) ← 0
    finpara
    para i de 1 a n hacer
        para j de 1 a N hacer
            si i = 1 y j < v(i) entonces c(i, j) ← ∞
            sino
                si i = 1 entonces c(i, j) ← 1 + c(i, j − v(1))
                sino
                    si j < v(i) entonces c(i, j) ← c(i − 1, j)
                    sino c(i, j) ← mín(c(i − 1, j), 1 + c(i, j − v(i)))
                    finsi
                finsi
            finsi
        finpara
    finpara

fin
```

Tema 7. Programación dinámica

Problema de la mochila (versión 2)

- Similar al que se vió en el tema anterior, pero los materiales no son divisibles. Ver contraejemplo.
- La solución será una tupla de reales $(x_1, x_2, \ldots, x_n)$ donde $x_i = \{0, v_i\}$. Un material se selecciona completo o no se selecciona.
- Se crea una tabla C de $n(V+1)$ donde $n =$ número de materiales y $V =$ número de unidades de volumen de la mochila.
- $C(i,j)$ indica el coste máximo obtenido empleando los i primeros materiales para conseguir un volumen j.
- $C(i,j) = max\{C(i-1,j), p_i * v_i + C(i-1, j-v(i))\}$.
  - si $C(i,j) = C(i-1,j)$ no se selecciona el material i.
  - si $C(i,j) = p_i * v_i + C(i-1, j-v(i))$ se selecciona el material i.
- Un material solo se puede seleccionar una vez (diferencia con el del cambio).
- Cumple el principio de optimalidad.

Tema 7. Programación dinámica

Ejemplo (Problema de la mochila)

- *5 materiales de volúmenes* $v_i = \{1, 2, 5, 6, 7\}$ *y de precios* $p_i = \{1, 3, 3'60, 3'67, 4\}$ *y que el volumen máximo es de* $V = 11$

v_i	p_i	0	1	2	3	4	5	6	7	8	9	10	11
1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	3	0	1	6	7	7	7	7	7	7	7	7	7
5	3.6	0	1	6	7	7	18	19	24	25	25	25	25
6	3.67	0	1	6	7	7	18	22	24	28	29	29	40
7	4	0	1	6	7	7	18	22	28	29	34	35	40

Cuadro : Tabla ejemplo de problema de la mochila versión 2.

Problema de la mochila (III)

- Los elementos de la fila y columna 0 son 0. Si la columna es negativa, el elemento es $-\infty$.
- Obtención de los elementos de la tabla usando la fórmula.
- La solución es $C(5, 11) = 40$.
  - $C(5, 11) = C(4, 11)$ el material 5 no entra en la mochila.
  - $C(4, 11) = p_4 *_v 4 + C(3, 11 - 6) = 6 * 3'67 + 18$. Por tanto el material 4 entra en la mochila.
  - $C(3, 5) = p_3 * v_3 + C(2, 5 - 5) = 5 * 3, 6 + 0$. Por lo tanto el material 3 entra en la mochila.
  - Como $c(2, 4) = 1 + c(2, 0)$ se contabiliza otra moneda del tipo 2. Ya hemos llegado a un cambio de 0.
  - Ya hemos llegado a volumen 0. La solución la forman los materiales 3 y 4.

Tema 7. Programación dinámica

Algoritmo *mochila*2($(n, V; D; C)$

inicio

  para j **de** 1 **a** V **hacer**

    $C(0, j) \leftarrow 0$

  finpara

  para j **de** 1 **a** n **hacer**

    $C(j, 0) \leftarrow 0$

  finpara

  para i **de** 1 **a** n **hacer**

    para j **de** 1 **a** V **hacer**

      si $j < D(i).volumen$ **entonces** $C(i, j) \leftarrow C(i-1, j)$

      sino $C(i, j) \leftarrow \max(C(i-1, j), D(i).precio * D(i).volumen + C(i-1, j - D(i).volumen)$

      finsi

    finpara

  finpara

fin

Tema 7. Programación dinámica

Camino más corto en grafo polietápico.

- Grafo dirigido con N nodos que se agrupan en k etapas ordenadas.
- La primera y la última etapa tienen un nodo.
- Cualquier camino que vaya de la primera a última etapa tiene k-1 lados.
- Hay que calcular el camino mínimo que va de la primera a la última etapa.
- En la solución aparece un nodo de cada etapa.
- Cumple el principio de optimalidad de Bellman.

Tema 7. Programación dinámica

Camino más corto en grafo polietápico (II).

- El camino mínimo para llegar a cualquier nodo de una etapa $i+1$ se obtiene a partir del camino mínimo para llegar a cualquier nodo de la etapa i.
- Sea X_k cualquier nodo de la etapa E_i, e Y un nodo de la etapa E_{i+1} , el coste c de ir de A a Y y el camino C serán:
  - $c(A,Y) = min\{c(A,X_k) + d(X_k,Y)\}\ \forall X_k \epsilon E_i$
  - $C(A,Y) = C(A,X) \cup (X,Y)$ siendo X el nodo seleccionado de la etapa i.

Tema 7. Programación dinámica

Ejemplo (Planificación de tareas a plazo fijo)

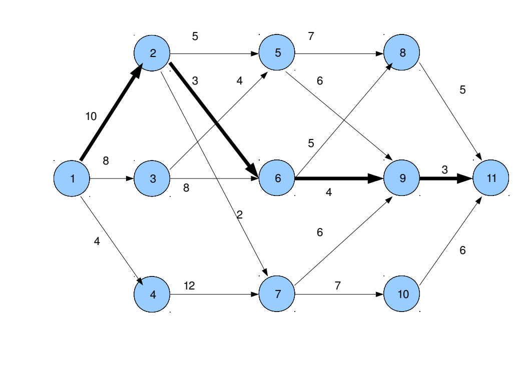


Figura : Ejemplo de camino mínimo en grafo polietápico

Camino mínimo en grafo polietápico (III)

- En la primera etapa tenemos:
  - $c(1,2) = 10 \;\; C(1,2) = \{1,2\}$
  - $c(1,3) = 8 \;\; C(1,3) = \{1,3\}$
  - $c(1,4) = 4 \;\; C(1,4) = \{1,4\}$
- En la segunda etapa tenemos:
  - $c(1,5) = \min\{c(1,2) + d(2,5), c(1,3) + d(3,5)\} = 12$ $C(1,5) = \{1,3,5\}$
  - $c(1,6) = \min\{c(1,2) + d(2,6), c(1,3) + d(3,6)\} = 13$ $C(1,6) = \{1,2,6\}$
  - $c(1,7) = \min\{c(1,2) + d(2,7), c(1,4) + d(4,7)\} = 12$ $C(1,7) = \{1,2,7\}$

Tema 7. Programación dinámica

Camino mínimo en grafo polietápico (IV)

- En la tercera etapa tenemos:
  - $c(1,8) = \min\{c(1,5) + d(5,8), c(1,6) + d(6,8)\} = \min\{19, 18\} = 18$, $C(1,8) = \{1,2,6,8\}$
  - $c(1,9) = \min\{c(1,5) + d(5,9), c(1,6) + d(6,9), c(1,7) + d(7,9)\} = \min\{18, 17, 18\} = 17$, $C(1,9) = \{1,2,6,9\}$
  - $c(1,10) = \min\{c(1,7) + d(7,10)\} = 19$, $C(1,10) = \{1,2,7,10\}$
- En la última etapa tenemos:
  - $c(1,11) = \min\{c(1,8) + d(8,11), c(1,9) + d(9,11), c(1,10) + d(10,11)\} = \min\{23, 20, 25\} = 20$, $C(1,11) = \{1,2,6,9,11\}$

Tema 7. Programación dinámica

Algoritmo *caminoMinimoGrafo*(*Etapa*, *n*; ; *coste*, *camino*)

inicio

  coste(*Etapa*(0).*nodo*(1).*etiqueta*) $\leftarrow$ 0

  camino(*Etapa*(0).*nodo*(1).*etiqueta*) $\leftarrow$ *Etapa*(0).*nodo*(1).*etiqueta*

  origen $\leftarrow$ *Etapa*(0).*nodo*(1).*etiqueta*

  para *i* **de** 1 **a** *Etapa*(1).*numeroNodosEtapa* **hacer**

    destino $\leftarrow$ *Etapa*(1).*nodo*(*i*).*etiqueta*

    coste(*destino*) $\leftarrow$ *d*(*origen*, *destino*)

    camino(*destino*) $\leftarrow$ *origen*

  finpara

  para *i* **de** 2 **a** *n* **hacer**

    para *j* **de** 1 **a** *Etapa*(*i*).*numeroNodosEtapa* **hacer**

      destino $\leftarrow$ *Etapa*(*i*).*nodo*(*j*).*etiqueta*

      minimoCoste $\leftarrow$ ∞

      indiceMinimo $\leftarrow$ 0

Tema 7. Programación dinámica

```
            para k de 1 a Etapa(i − 1).numeroNodosEtapa hacer
                origen ← Etapa(i − 1).nodo(k).etiqueta
                si d(origen, destino) < ∞ entonces
                    si coste(origen) + d(origen, destino) < minimoCoste entonces
                        minimoCoste ← coste(origen) + d(origen, destino)
                        indiceMinimo ← Etapa(i − 1).nodo(k).etiqueta
                    finsi
                finsi
            finpara
            coste(destino) ← minimoCoste
            camino(destino) ← camino(indiceMinimo) ∪ indiceMinimo
        finpara
    finpara

fin
```

Tema 7. Programación dinámica

Problema del viajante de comercio

- La solución del problema se puede plantear como una sucesión de decisiones que verifique el principio de optimalidad de Bellman.
- Cada recorrido está formado por un lado $L(n_1, n_k)$ para algún nodo n_k perteneciente a $N - \{n_1\}$ y un camino de n_k al nodo n_1.
- Cada trozo ha de cumplir el principio de optimalidad.
- $d(n_1, N - \{n_1\}) = \text{mín}_{2 \leq k \leq n}\{L_{n_1,n_k} + d(n_k, N - \{n_1, n_k\}\}$

Tema 7. Programación dinámica

Problema del viajante de comercio (II)

$$\begin{pmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{pmatrix}$$

- En la primera etapa.
  - $d(2, \phi) = L_{2,1} = 5$
  - $d(3, \phi) = L_{3,1} = 6$
  - $d(4, \phi) = L_{4,1} = 8$

Tema 7. Programación dinámica

Problema del viajante de comercio (III)

- En la segunda etapa se obtienen los caminos óptimos que llegan al nodo 1, con dos nodos previos:
  - $d(2, \{3\}) = L_{2,3} + d(3, \phi) = 9 + 6 = 15$
  - $d(2, \{4\}) = L_{2,4} + d(4, \phi) = 10 + 8 = 18$
  - $d(3, \{2\}) = L_{3,2} + d(2, \phi) = 13 + 5 = 18$
  - $d(3, \{4\}) = L_{3,4} + d(4, \phi) = 12 + 8 = 20$
  - $d(4, \{2\}) = L_{4,2} + d(2, \phi) = 8 + 5 = 13$
  - $d(4, \{3\}) = L_{4,3} + d(3, \phi) = 9 + 6 = 15$
- En la tercera etapa se obtienen los caminos óptimos que llegan al nodo 1, con tres nodos previos.
  - $d(2, \{3, 4\}) = \text{mín}\{L_{2,3} + d(3, \{4\}), L_{2,4} + d(4, \{3\})\} = \text{mín}\{9 + 20, 10 + 15\} = 25$
  - $d(3, \{2, 4\}) = \text{mín}\{L_{3,2} + d(2, \{4\}), L_{3,4} + d(4, \{2\})\} = \text{mín}\{13 + 18, 12 + 13\} = 25$
  - $d(4, \{2, 3\}) = \text{mín}\{L_{4,2} + d(2, \{3\}), L_{4,3} + d(3, \{2\})\} = \text{mín}\{8 + 15, 9 + 18\} = 23$
- Por último, se obtiene el camino óptimo que llega al 1, con cuatro nodos previos, comenzando en 1:
  - $d(1, \{2, 3, 4\}) = \text{mín}\{L_{1,2} + d(2, \{3, 4\}), L_{1,3} + d(3, \{2, 4\}), L_{1,4} + d(4, \{2, 3\})\} = \text{mín}\{10 + 25, 15 + 25, 20 + 23\} = 35$

Problema del viajante de comercio (IV)

- Para obtener el camino:
- En la segunda etapa
  - $l(2, \{3\}) = 3$, $l(2, \{4\}) = 4$
  - $l(3, \{2\}) = 2$, $l(3, \{4\}) = 4$
  - $l(4, \{2\}) = 2$, $l(4, \{3\}) = 3$
- En la tercera etapa
  - $l(2, \{3, 4\}) = 4$
  - $l(3, \{2, 4\}) = 4$
  - $l(4, \{2, 3\}) = 2$
- Finalmente:
  - $l(1, \{2, 3, 4\}) = 2$
- La ruta óptima es:

  $1 \rightarrow l(1, \{2, 3, 4\}) = 2 \rightarrow l(2, \{3, 4\}) = 4 \rightarrow l(4, \{3\}) = 3 \rightarrow 1$

Tema 7. Programación dinámica

```
Algoritmo d(i, S)
inicio
    si vacia(S) = cierto entonces
        devolver L(i, 1)
    sino
        masCorto ← ∞
        para j ∈ S hacer
            distancia ← L(i, j) + d(j, S − j)
            si distancia < masCorto entonces
                masCorto ← distancia
            finsi
        finpara
        devolver masCorto
    finsi

fin
```

Tema 8. Backtracking

Tema 8. Backtracking

Competencias

- CTEC3. Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

Tema 8. Backtracking

Objetivos del tema

- Estudiar el método del Backtracking.
- Problemas que no se han podido resolver con los métodos de los temas precedentes.
- Partir de un conjunto de posibles soluciones.
- Dentro de este conjunto:
  - Buscar todas las soluciones.
  - Buscar la solución óptima.
- Exploración del conjunto de posibles soluciones.
- Ejemplos ilustrativos.

Tema 8. Backtracking

Introducción

- Usa un conjunto a priori de posibles soluciones, que contiene a todas, y encuentra las que realmente lo son.
- Busca una solución o todas o la óptima.
- Exploración las posibles soluciones de un problema, y éstas se pueden dividir en etapas.
- La solución es una n-tupla $(x_1, x_2, \ldots, x_n)$. Donde los x_i pertenecen a un conjunto S de candidatos.
- En optimización la tupla optimiza una función criterio $P(x_1, x_2, \ldots, x_n)$.

Introducción (II)

- La exploración del conjunto de soluciones se representa en forma de árbol:
  - El árbol se crea de forma similar al recorrido en profundidad de un grafo.
  - Cada nodo representa un trozo de solución.
  - El camino desde la raíz a un nodo de nivel k, representa la solución de las k primeras etapas.
  - Los nodos hijos serán posibles prolongaciones al añadir una nueva etapa.

Tema 8. Backtracking

El método general

- Primero se fija la descomposición en etapas de la solución, definiendo la estructura del árbol a analizar.
- Las opciones de cada etapa dependen de:
  - La etapa en la que nos encontremos.
  - Del trozo de solución construido hasta ese momento.
- En el árbol hay que identificar que nodos que son posibles soluciones, y cuales son sólo etapas previas.
- Nodos fracaso: a partir de ellos no se obtiene solución.
  - Si se detecta a tiempo: se busca camino alternativo.
  - Si se detecta a través de sus descendientes (todos son fracaso) se retrocede y se busca camino alternativo. De ahí el nombre de backtracking o vuelta atrás.

Tema 8. Backtracking

Ejemplo (Backtracking aplicado a 4 reinas)

x			
		x	

Las tuplas que comienzan por {1,3} no dan lugar a solución

x			
			x
	x		

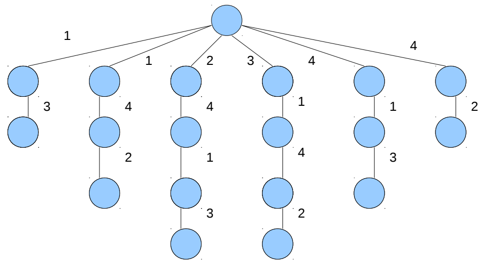
Las tuplas que comienzan por {1,4,2} no dan lugar a solución

	x		
			x
x			
		x	

Se ha retrocedido (backtracking) hasta la primera fila y se ha encontrado la primera solución. {2,4,1,3}

Tema 8. Backtracking

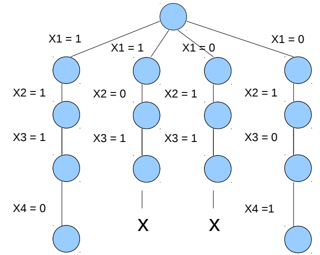
Ejemplo (Árbol obtenido con el backtracking aplicado a 4 reinas)



Tema 8. Backtracking

Ejemplo (Árbol obtenido con el backtracking aplicado a la suma de subconjuntos)

- *Suma subconjuntos.* $S = \{5, 10, 15, 20\}$, $M = 30$.



Tema 8. Backtracking

El método general (II)

- Proceso para encontrar todas las soluciones:
  - Sea $(x_1, x_2, \ldots, x_i)$ el camino desde la raíz a un nodo cualquiera (hasta la etapa i).
  - Sea $C(x_1, x_2, \ldots, x_i)$ el conjunto de posibles candidatos para la etapa $i + 1$.
  - Sea $P_{i+1}(x_1, x_2, \ldots, x_i, x_{i+1})$ el conjunto de predicados que son ciertos si x_{i+1} forma parte de la solución.
  - En la etapa $i + 1$ los candidatos son aquellos elementos de C que satisfacen P_{i+1}.

Tema 8. Backtracking

Algoritmo *Backtracking*$(n; ; X)$

inicio

$k \leftarrow 1$

mientras $k > 0$ **hacer**

si $\exists X(k) \in C(X(1), X(2), \ldots, X(k-1))$ **y** $P_k(X(1), \ldots, X(k)) = true$ **entonces**

$X(k) \leftarrow C(X(1), X(2), \ldots, X(k-1))$

si $X(1), \ldots, X(k)$ *es solucion* **entonces**

escribir $X(1), \ldots, X(k)$

sino

$k \leftarrow k + 1$ *pasa al siguiente nodo*

finsi

sino

$k \leftarrow k - 1$ *se retrocede al nodo anterior*

finsi

finmientras

fin

Tema 8. Backtracking

Algoritmo *BacktrackingRecursivo*$(n, k;\,; X)$

inicio

  mientras $\exists X(k) \in C(X(1), X(2), \ldots, X(k-1))$ **y** $P_k(X(1), \ldots, X(k)) = true$ **hacer**

    $X(k) \leftarrow C(X(1), X(2), \ldots, X(k-1))$

    si $X(1), \ldots, X(k)$ *es solucion* **entonces**

      escribir $X(1), \ldots, X(k)$

    finsi

    BacktrackingRecursivo$(n, k+1;\,; X)$

  finmientras

fin

Tema 8. Backtracking

El método general (III)

- Factores que determinan la eficiencia del método:
  - El conjunto de partida o de búsqueda ha de ser lo más pequeño posible (conteniendo todas las soluciones).
  - La complejidad de las pruebas que detectan nodos fracaso.
    - Árboles en general: pruebas sencillas de poco coste.
    - Árboles gigantescos: Pruebas costosas para reducir la búsqueda.

Tema 8. Backtracking

Problema de las n-reinas.

- Colocar *n*-reinas en un tablero de *nxn* sin que se apunten.
- La solución es una n-tupla $(x_1, x_2, \ldots, x_n)$. Donde los i indican la fila y los x_i indican columna.
- Los x_i han de ser distintos así se garantiza que no se apunten ni en horizontal ni en vertical.
- $|x(i) - x(k)| \neq |i - k|$ para que no se apunten en diagonal.

Tema 8. Backtracking

Algoritmo $Lugar(k, x; ;)$

inicio

    para i **de** 1 **a** $k-1$ **hacer**

        si $(x(i) = x(k)$ **o** $|x(i) - x(k)| = |i - k|$ **entonces**

            devolver *falso*

        finsi

    finpara

    devolver *cierto*

fin

Tema 8. Backtracking

```
Algoritmo n − reinas(n; ; )
inicio
    colocar primera reina en columna 0 (fila 1)
    mientras no se tengan todas las soluciones hacer
        desplazar reina a la siguiente columna
        mientras no salga del tablero y sea amenazada por una anterior hacer
            desplazar reina a la siguiente columna
        finmientras
        si una posicion correcta ha sido encontrada entonces
            si es la ultima reina, se tiene una solucion entonces
                escribir la solucion
            sino No es la ultima reina y hay que probar la siguiente
                pasar a probar la siguiente reina ubicandola en columna 0
            finsi
        sino la reina no se puede ubicar
            volvemos a reina anterior
        finsi
    finmientras

fin
```

Tema 8. Backtracking

```
Algoritmo n − reinas(n)
inicio
    x(1) ← 0
    k ← 1
    mientras k > 0 hacer x(k) ← x(k) + 1
        mientras x(k) ≤ n y Lugar(k, x) = falso hacer
            x(k) ← x(k) + 1
        finmientras
        si x(k) ≤ n entonces
            si k = n entonces
                escribir x(1), x(2), . . . , x(k)
            sino
                k ← k + 1
                x(k) ← 0
            finsi
        sino
            k ← k − 1
        finsi
    finmientras

fin
```

Tema 8. Backtracking

Problema de la suma de subconjuntos

- Se tiene un conjunto de n enteros positivos, de valores $V(i)$, ver que subconjuntos suman M.
- La solución es una tupla de n elementos que son 1 ó 0.
- El árbol que genera las soluciones es binario.
  - El hijo izquierdo de nivel i, indica que $v(i)$ entra en la solución.
  - El hijo derecho de nivel i, indica que $v(i)$ no entra.
- El predicado para comprobar candidatos consiste en verificar que la suma de los enteros ya introducidos más los candidatos que quedan por introducir ha de ser como mínimo M.
- Si se ordenan en orden creciente los candidatos, los k primeros candidatos no pueden formar solución si al sumarle el $k+1$ se supera M.

Tema 8. Backtracking

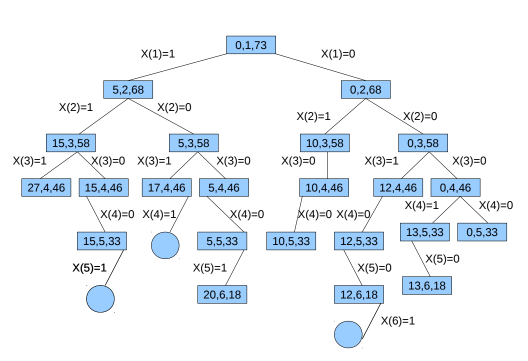
Algoritmo *SumaSubconjuntos*$(s, k, r, n, M, V; ; X)$
inicio
  Genera hijo izquierdo. $s + V(k) \leq M$ *ya que* $P_{k-1}(X(1), \ldots, X(k-1)) = true$
  $X(k) \leftarrow 1$
  si $s + V(k) = M$ **entonces** *Subconjunto encontrado* **escribir** $(X(1), \ldots, X(k)$
  sino
    Comprueba si el $k+1$ *es viable, así* k *podría formar parte de la solución*
    si $s + V(k) + V(k+1) \leq M$ **entonces** $P_k(X(1), \ldots, X(k)) = true$
      SumaSubconjuntos$(s + V(k), k+1, r - V(k), n, M, V; ; X)$
    finsi
  finsi
  Se genera el hijo derecho y se evalua P_k
  si $s + r - V(K) \geq M$ **y** $s + V(k+1) \leq M$ **entonces** $P_k(X(1), \ldots, X(k)) = true$
    La primera parte comprueba si al eliminar $V(k)$ *del resto se sobrepasa o iguala* M
    La segunda parte comprueba que al añadir $V(k+1)$ *no se sobrepasa* M
    $X(k) \leftarrow 0$
    Se sigue buscando solucion sin incluir al $V(k)$
    SumaSubconjuntos$(s, k+1, r - V(k), n, M, V; ; X)$
  finsi

fin

Tema 8. Backtracking

Ejemplo (Suma de subconjuntos)

- *Suma subconjuntos.* $S = \{5, 10, 12, 13, 15, 18\}, M = 30.$



Tema 8. Backtracking

Ciclos hamiltonianos

- Es un camino que recorre todos los nodos pasando por todos una sola vez y vuelve al nodo de origen.
- El vector solución $(X(1), \ldots, X(n))$, se define de forma tal que el i-ésimo elemento nodo visitado queda representado por $X(i)$.
- La solución se generará de la siguiente forma:
  - $X(1) = 1$ para no repetir ciclos.
  - Si $1 < k < n$ y se tienen $k - 1$ nodos ya seleccionados, el candidato $X(k)$ será un nodo no seleccionado y que esté unido al $X(k-1)$.
  - $X(n)$ solo puede ser el único nodo restante y ha de estar unido a $X(n-1)$ y al $X(1)$.

Tema 8. Backtracking

```
Algoritmo SiguienteNodo(k, n, C; X; )
inicio
    iterar
        X(k) ← (X(k) + 1)mod(n + 1) recorre de forma cíclica los nodos del grafo
        salir si X(k) = 0 Ha recorrido todos los nodos sin encontrar ningún candidato
        si C(X(k − 1), X(k)) = cierto entonces lado que conecta anterior y evaluado
            j ← 1
            mientras X(j) ≠ X(k) hacer
            Comprobamos si coincide con alguno anterior
                j ← j + 1
            finmientras
            si j = k entonces No coincide con anteriores. Puede ser candidato
                Se comprueba también si es el último, para comprobar el cierre
                salir si k < n o (k = n y C(X(n), 1) = cierto)
            finsi
        finsi
    finiterar

fin
```

Tema 8. Backtracking

Algoritmo *CicloHamiltoniano*$(k, n, C; X;)$

inicio

  iterar

    SiguienteNodo$(k, n, C; X;)$

    salir si $X(k) = 0$ *No ha encontrado candidatos*

    si $k = n$ **entonces** *ha encontrado un ciclo y lo escribe*

      escribir $X(1), X(2), \ldots, X(n), X(1)$

    sino *Se busca el siguiente nodo del ciclo*

      CicloHamiltoniano$(k+1, n, C; X;)$

    finsi

  finiterar

fin

- Los ciclos comenzarán por el nodo 1, el vector X está inicializado a 0, excepto $X(1) = 1$. En La primera llamada $k = 2$.
- El algoritmo se podría adaptar para resolver el problema del viajante de comercio guardando el ciclo de coste mínimo de entre todos los ciclos calculados.

Tema 8. Backtracking

Problema de la mochila (versión 3)

- La solución se almacena en una tupla de n elementos de 0 o 1.
- El árbol sería binario similar al de la suma de subconjuntos
- Es necesaria una condición que elimine nodos fracaso con bastante antelación dado que el árbol puede ser gigantesco.
- Condición eficiente:
  - Establecer un límite superior para la solución que se obtiene al expandir el nodo.
  - Si éste límite no supera al óptimo encontrado, no se expande el nodo.
  - La condición será usar el algoritmo voraz (particionando materiales) para expandir el nodo evaluado. Dicha solución será un límite superior a la solución obtenida al expandir el nodo.

Tema 8. Backtracking

Algoritmo $Limite(n, pActual, vActual, k, p, v, V;;)$

inicio

  $valor \leftarrow pActual$

  $volumen \leftarrow vActual$

  para i **de** $k+1$ **a** n **hacer**

    Se introducen los restantes más caros hasta llegar al volumen de la mochila

    $volumen \leftarrow volumen + v(i)$

    si $volumen < V$ **entonces** *Se puede introducir el material i completo*

      $valor \leftarrow valor + p(i) * v(i)$

    sino *Se puede introducir el material i parcialmente*

      devolver $valor + (V - (volumen - v(i))) * p(i)$

    finsi

  finpara

  devolver $valor$

fin

- Se deduce que el límite para un posible hijo izquierdo de un nodo es el mismo que para ese nodo. Debido a ello no se usa siempre que se desciende de un hijo izquierdo a otro.
- La función límite se usa sólo después de una serie de movimientos con éxito hacia la izquierda (movimientos factibles a hijo izquierdo).

Tema 8. Backtracking

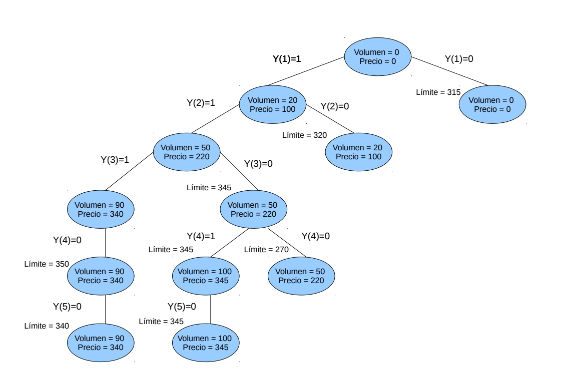
```
    Algoritmo MochilaBacktracking(n, p(n), v(n), V;; X(n), precioFinal, volumenFinal)
inicio precio ← 0, volumen ← 0, precioFinal ← −1, k ← 1
    iterar
        mientras k ≤ n y volumen + v(k) ≤ V hacer  El material k entra en la mochila
            volumen ← volumen + v(k), precio ← precio + p(k) ∗ v(k), Y(k) ← 1, k ← k + 1
        finmientras
        si k > n entonces Actualiza solución
            precioFinal ← precio, volumenFinal ← volumen, k ← n, X ← Y
        sino El volumen se supera con el objeto k. Se quita el objeto k. Y(k) ← 0
        finsi despues de que precioFinal se calcule arriba, Limite=precioFinal
        mientras Limite(n, precio, volumen, k, p(n), v(n), V) ≤ precioFinal hacer
            mientras k ≠ 0 y Y(k) ≠ 1 hacer
                k ← k − 1 Busca el último material de la mochila
            finmientras
            salir si k = 0 aqui termina el algoritmo. Borra el k-ésimo elemento
            Y(k) ← 0, volumen ← volumen − v(k), precio ← precio − p(k) ∗ v(k)
        finmientras
        k ← k + 1 Pasa al siguiente material
    finiterar

fin
```

Tema 8. Backtracking

Ejemplo (Problema de la mochila)

- $V = 100$, $v(i) = \{20, 30, 40, 50, 60\}$, $p(i) = \{5, 4, 3, 2{,}5, 1\}$.



Tema 9. Algoritmos probabilistas

Tema 9. Algoritmos probabilistas

Competencias

- CTEC3. Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

Tema 9. Algoritmos probabilistas

Objetivos del tema

- Definir algoritmo probabilista.
- ¿Cuándo es aplicable?.
- Ventajas.
- Tipos de algoritmos probabilistas.
- Ejemplos.

Tema 9. Algoritmos probabilistas

Introducción

- Selecciona aleatoriamente una alternativa entre las distintas posibilidades en una etapa del algoritmo.
- Se utilizan cuando el tiempo que se necesitaría para determinar la mejor alternativa es prohibitivo.
- Ventajas:
  - Se pueden comportar de forma distinta (tiempo de ejecución y resultados) cuando se aplica dos veces a un mismo problema.
  - Puede obtener todas las soluciones ejecutándolo varias veces. En un determinista habría que implementarlo en cada una de las variantes posibles.
  - Puede dar resultados erróneos que se subsanan ejecutándolo otra vez (erroneos con baja probabilidad).

Tema 9. Algoritmos probabilistas

Introducción(II)

- Algunos proporcionan una respuesta probabilista cuyo error se puede acotar lo que se quiera, incluso por debajo del error "físico" en uno determinista.
- Existen problemas en los que no hay algoritmo deterministas o probabilistas con respuesta fiable en un tiempo razonable y sin embargo existen algoritmos probabilistas que pueden resolverlos en tiempo razonable si se admite una pequeña probabilidad de error.

Tema 9. Algoritmos probabilistas

Introducción(III)

- Algoritmos probabilistas que no garantizan corrección del resultado:
  - Algoritmos numéricos: producen un intervalo de confianza con un nivel de probabilidad.
  - Algoritmos Monte Carlo: dan la respuesta exacta con una alta probabilidad, aunque pueden proporcionar una respuesta incorrecta
- Los algoritmos probabilistas que nunca dan una respuesta incorrecta se denominan de Las Vegas.
- El tiempo esperado de un algoritmo probabilista se define para cada caso individual y sería el tiempo promedio para ese caso si se resolviese varias veces.
- El tiempo esperado en el caso peor se refiere al tiempo que requiere el peor caso posible de un tamaño dado.

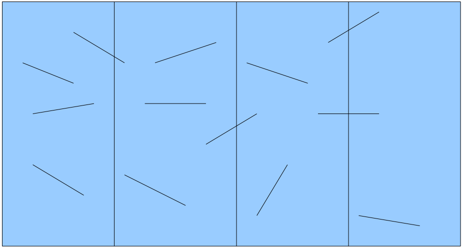
Tema 9. Algoritmos probabilistas

Algoritmos probabilistas numéricos

- Una de las aplicaciones más extendidas es la simulación.
- Mediante la simulación se obtienen soluciones aproximadas a problemas en los que es complicado obtener soluciones por métodos deterministas.
- La respuesta es aproximada, pero esta precisión aumenta si aumenta el tiempo de ejecución del algoritmo.
- El error es inversamente proporcional a la raíz cuadrada de la cantidad de tiempo que se haya empleado, así que se necesita aumentar cien veces el tiempo empleado para aumentar la precisión un dígito.

Tema 9. Algoritmos probabilistas

Ejemplo (La aguja de Buffon (I))



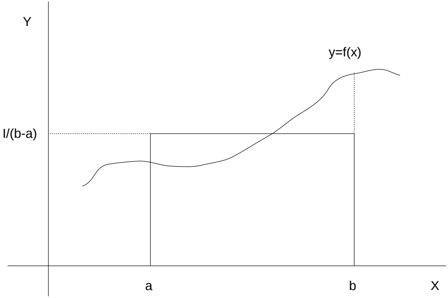
Tema 9. Algoritmos probabilistas

Aguja de Buffon(II)

- Proceso para encontrar todas las soluciones:
    - Aguja de longitud L.
    - Ancho de las planchas de longitud $2L$.
    - Probabilidad para 1 aguja toque línea $= 1/\pi$
    - Promedio esperado de agujas $n = N/\pi$
    - Por tanto $\pi = N/n$

Tema 9. Algoritmos probabilistas

Ejemplo (Integración numérica (I))



Tema 9. Algoritmos probabilistas

Algoritmo $IntegracionNumerica(N, f, a, b; ; superficie)$
inicio
 $suma \leftarrow 0$
 para i **de** 1 **a** N **hacer**
  $x \leftarrow uniforme(a, b)$
  $suma \leftarrow suma + f(x)$
 finpara
 $superficie \leftarrow (b - a)\frac{suma}{N}$
fin

- Necesita multiplicar por 100 las simulaciones para aumentar un dígito de precisión.
- Ideal, frente a métodos tradicionales, en funciones de varias variables.

Tema 9. Algoritmos probabilistas

Algoritmos de Monte Carlo

- Existen problemas para los que no hay algoritmo eficiente (probabilista o determinista) que proporcione una solución correcta en todas las ocasiones.
- Los algoritmos de Monte Carlo pueden cometer un error, pero encuentran la solución correcta con una probabilidad alta, independientemente del caso estudiado.
- Pueden devolver resultados erróneos sin avisar.

Tema 9. Algoritmos probabilistas

Algoritmos de Monte Carlo (II)

- Esto no implica que falle de vez en cuando en casos especiales, implica que no hay ningún caso en el cual la probabilidad de error es elevada, que es lo deseable.
- Un algoritmo de Monte Carlo es p-correcto si devuelve una respuesta correcta con una probabilidad no menor que p, sea cual sea el caso considerado.
- Amplificación de la ventaja estocástica: se puede reducir la probabilidad de error aumentando el tiempo de cálculo (repitiendo las pruebas).

Tema 9. Algoritmos probabilistas

Verificación del producto de matrices.

- Sean A, B, C matrices cuadradas de orden n.
- Se quiere comprobar que $C = AB$.
- El algoritmo tradicional del producto es de $O(n^3)$ aunque se puede mejorar hasta $O(n^{2,37})$.
- ¿se puede mejorar dicho tiempo admitiendo una pequeña probabilidad de error?.
- Se va demostrar que para cualquier $\epsilon > 0$, prefijado, se necesita un tiempo $O(n^2)$ para obtener un error menor que ϵ.

Tema 9. Algoritmos probabilistas

Verificación del producto de matrices(II).

- Suponemos que $AB \neq C$, entonces $D = AB - C \neq 0$.
- Sea i un entero que representa una fila de D que contenga algún elemento $\neq 0$.
- Si se seleccionan aleatoriamente una serie de filas de D. Sea S el conjunto que almacena los numeros de filas seleccionadas.
- Sea $\sum_S D$ un vector que guarda la suma de los elementos de cada fila de D almacenada en el conjunto S.
- $P(\sum_S D \neq 0) \geq 1/2$ ya que $P(i \epsilon S) = 1/2$

Verificación del producto de matrices(III).

- $\sum_S D = 0$ siempre que $C = AB$.
- Por tanto se selecciona un conjunto S aleatoriamente y verificando si $\sum_S D = 0$, se comprueba si $C = AB$.
- El problema es que para calcular D necesitamos tener AB ya que $D = AB - C$.
- Empleando el siguiente recurso se puede calcular D en $O(n^2)$:
  - Sea $X(j)$ un vector de 0 y 1 donde $X(j) = 1$ si $j \epsilon S$ y 0 en caso contrario.
  - Por tanto $\sum_S D = XD = X(AB - C)$
  - Ahora se comprueba si $X(AB - C) = 0$ o $XAB = XC$.
  - Se puede demostrar que si XAB se calcula como $(XA)B$ es de $O(n^2)$ ya que X es una matriz de una fila y XA también.

Tema 9. Algoritmos probabilistas

Algoritmo *VerificacionMultiplicacionMatrices*$(A, B, C, n;;)$

inicio

  para i **de** 1 **a** n **hacer**

    $x(i) \leftarrow uniforme\{0, 1\}$

  finpara

  si $(XA)B = XC$ **entonces**

    devolver *cierto*

  sino

    devolver *falso*

  finsi

fin

Tema 9. Algoritmos probabilistas

Verificación del producto de matrices(IV).

- El algoritmo devuelve una respuesta correcta con una probabilidad del 50 % en todos los casos
- Este algoritmo es 1/2-correcto por definición, ya que si hubiese algún error, D tendría como mínimo una fila errónea
- La única ventaja es que si devuelve falso la respuesta es correcta, en caso contrario no sabemos si la respuesta es correcta (50 % de probabilidades).
- Una mejora de la fiabilidad consistiría en aplicar el algoritmo varias veces (k veces) sobre el mismo caso. $P(Error) = (1/2)^k$ y sería $1 - (1/2)^k$-correcto.

Tema 9. Algoritmos probabilistas

Algoritmos de Las Vegas

- Nunca proporcionan una respuesta incorrecta. Hay dos tipos:
  - Los que usan la aleatoriedad para buscar una solución correcta, aunque sean poco eficientes (a costa de tiempo encuentran la solución). Un ejemplo es el quicksort en la elección del pivote.
  - Los que pueden tomar caminos equivocados, que conducen a caminos sin salida, haciendo imposible la obtención de una solución en esta ejecución del algoritmo. Un ejemplo es la versión probabilista de las 8-reinas.
- En tiempo promedio no mejoran los algoritmos deterministas.

Tema 9. Algoritmos probabilistas

Algoritmos de Las Vegas (II)

- Aceleran los peores casos pero no mejoran los mejores casos ya que los enlentecen hasta un valor medio.
- El algoritmo reconoce su error cuando llega a un callejón sin salida. En este caso se aplica nuevamente.
- Un algoritmo será de Las Vegas si su probabilidad de éxito para cualquier caso es mayor de 0. Esto asegura que encontrará una solución siempre si se sigue repitiendo el algoritmo.

Tema 9. Algoritmos probabilistas

El problema de las 8-reinas (versión 2).

- Usando backtracking:
  - Árboles muy grandes.
  - Posiciones casi aleatorias de las reinas en las soluciones.
- Solución: colocar reinas aleatoriamente en las filas hasta que:
  - No se pueda colocar ninguna más. Se empieza de nuevo.
  - Colocar la última reina. Ya hay solución.
- Se puede demostrar que por término medio se necesitan 8 pruebas para obtener una solución correcta.

Tema 9. Algoritmos probabilistas

Algoritmo $n - reinasLasVegas(n; ; X, exito)$
inicio
    para i **de** 1 **a** n **hacer**
        $x(i) \leftarrow 0$
    finpara
    para k **de** 1 **a** n **hacer**
        $contador \leftarrow 0$
        para j **de** 1 **a** n **hacer**
            $X(k) \leftarrow j$
            si $Lugar(k, x) = cierto$ **entonces**
                $contador \leftarrow contador + 1, \; ok(contador) \leftarrow j$
            finsi
        finpara
        salir si $contador = 0$
        $columna \leftarrow ok(uniforme(1, contador)), \; X(k) \leftarrow columna$
    finpara
    si $contador = 0$ **entonces** $exito \leftarrow falso$
    sino $exito \leftarrow cierto$ `Hay solución`
    finsi

fin

Algorítmica

Prof. Dr. Ángel Carmona Poyato

Departamento de Informática y Análisis Numérico
Escuela Politécnica Superior de Córdoba
Universidad de Córdoba