# Análisis y Diseño de Algoritmos

## Diseño de Algoritmos

Arturo Díaz Pérez
**Sección de Computación**
**Departamento de Ingeniería Eléctrica**
**CINVESTAV-IPN**
**Av. Instituto Politécnico Nacional No. 2508**
**Col. San Pedro Zacatenco**
**México, D. F. CP 07300**

**Tel. (5)747 3800 Ext. 3755**
**e-mail: adiaz@cs.cinvestav.mx**



## Estrategias Generales

- Estrategias generales para el Diseño de Algoritmos
  - Divide y Vencerás
  - Programación Dinámica
  - Algoritmos Ávidos
  - Backtracking
  - Branch and Bound

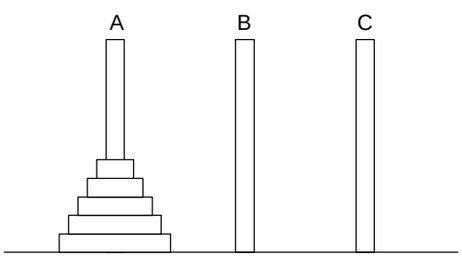
# Estrategias Generales

- Estrategias generales para resolver problemas de optimización sobre espacios de búsqueda exponenciales
  - Búsqueda Local
  - Recocido Simulado (Simulated Annealing)
  - Búsqueda Tabú
  - Algoritmos Genéticos
  - Redes Neuronales
  - Técnicas de Aleatorización



# Las Torres de Hanoi

- Se tienen 3 postes, A, B, y C. Inicialmente el poste A tiene apilados un número de discos, iniciado con el diámetro más grande en el fondo hasta el más pequeño en el tope.
- El problema es mover los discos de un poste a otro, uno a la vez, sin colocar nunca un disco de diámetro mayor sobre uno de diámetro menor




```
struct Poste P[3];

void Hanoi( int n, int A, int B, int C )
{
        if(n > 1){
                Hanoi(n-1, A, C, B);
                MueveDisco(A, B);
                Hanoi(n-1, C, B, A);
        } else
                MueveDisco(A,B);
}
```

¿Cuál es el tiempo de ejecución de la función Hanoi?

# Multiplicación de Enteros

☞ Considere el problema de multiplicar los enteros $x$ e $y$ de $n$ bits

x = | A | B | $x = A\,2^{n/2} + B$

y = | C | D | $y = C\,2^{n/2} + D$

$$xy = AC2^n + (AD + BC)2^{n/2} + BD$$

La estrategia anterior requiere de
- 4 multiplicaciones de $n/2$ bits: AC, AD, BC y BD
- 3 sumas de a lo más $2n$ bits
- 2 corrimientos



# Multiplicación de Enteros

← Si las sumas y corrimientos toman un tiempo $O(n)$, la siguiente recurrencia determina el número total de operaciones de bits que se requieren para multiplicar enteros de $n$ bits

$$T(1) = 1$$
$$T(n) = 4T\left(\tfrac{n}{2}\right) + cn$$

← La recurrencia anterior es un $O(n^2)$ operaciones (Tarea)

# Multiplicación de Enteros Reformulada

- Escribamos el producto de enteros de la siguiente forma

$$xy = AC2^n + [(A+B)(C+D) - AC - BD]2^{n/2} + BD$$

La estrategia requiere de
- 3 multiplicaciones de *n*/2 bits: AC, (A+B)(C+D) y BD
- 6 sumas (sustracciones) de *n*/2 bits
- 2 corrimientos

La siguiente recurrencia determina el número total de operaciones de bits

$$T(1) = 1$$
$$T(n) = 3T\left(\tfrac{n}{2}\right) + cn$$

La cual tiene una solución

$$O\left(n^{\log_2 3}\right) = O\left(n^{1.59}\right)$$

Tarea: resolver la recurrencia anterior



# Divide y Vencerás

- Divide y vencerás
  - Un problema se divide en problemas correspondientes a instancias más sencilla, éstos se resuelven y las soluciones se componen para obtener una solución del problema original
  - Ejemplo: MergeSort

```
void Sort( int A, int n )
{
  int    A1[], A2[];

  if( n > 1 ) {
  /* Divide a A en dos mitades, A1 y A2,
     cada una de longitud n/2   */
    Sort( A1, n/2 );
    Sort( A2, n/2 );
    Mezcla( A1, A2, A);
  }
}
```

## Divide y Vencerás

- MergeSort
  - Divídase un arreglo dado de *n* entradas en dos de *n*/2 entradas
  - Vénzase ordenando por este mismo algoritmo a los arreglos
  - Combínese los arreglos ya ordenados para obtener el arreglo original ya ordenado

- Fases de un algoritmo Divide y Vencerás
  - Divide
  - Vence
  - Combina

$$T(1) = 1$$
$$T(n) = aT\left(\tfrac{n}{b}\right) + d(n)$$



## Divide y Vencerás

- Divide y vencerás es un enfoque de arriba hacia abajo

- Con frecuencia no hay manera de dividir un problema en un número de subproblemas de tamaño menor cuyas soluciones se puedan combinar para resolver el problema original

# Programación Dinámica

- Divide un problema en tantos subproblemas como sea necesario cuyas soluciones son triviales.
- Es útil cuando solo hay un número polinomial de subproblemas diferentes a resolver
- Las soluciones de los subproblemas resueltos se colocan en una tabla y cada vez que se requieren se hace una consulta
- Es un enfoque de abajo hacia arriba
- El término proviene de la teoría del control
  - ← Programación se refiere al uso de tablas (arreglos)
  - ← Dinámica se refiere a la forma de llenar la tabla



# Pronósticos

- Supongamos que dos equipos, A y B, están jugando una serie para ver quien es el primero en ganar $n$ juegos, para algún $n$ particular
  - ← Sea $P(i, j)$ la probabilidad de que A gane la serie dado que A necesita $i$ juegos para ganar y B necesita $j$ juegos.
  - ← Bajo el supuesto que los dos equipos son igualmente competitivos, se puede expresar la siguiente fórmula.

$$P(i, j) = \begin{cases} 1 & si \quad i = 0, j > 0 \\ 0 & si \quad i > 0, j = 0 \\ \dfrac{P(i-1, j) + P(i, j-1)}{2} & si \quad i > 0 \wedge j > 0 \end{cases}$$

# Pronósticos

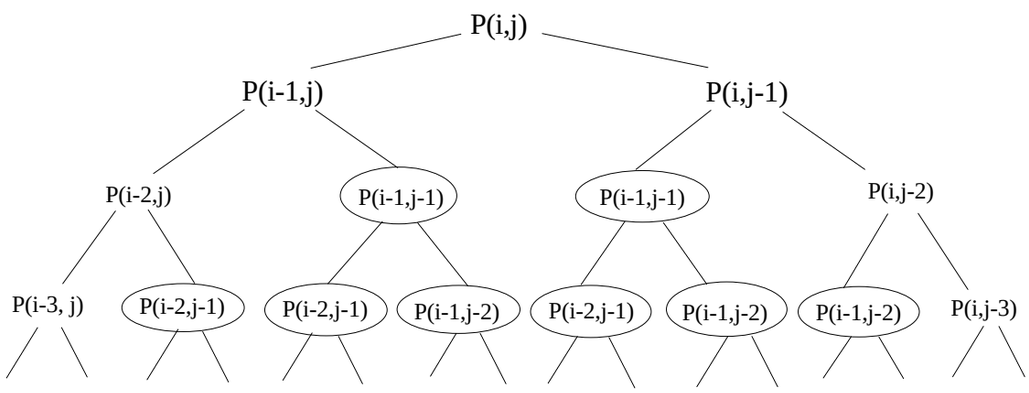
- Usando la recurrencia para calcular directamente el valor de P(i,j) se tiene que
  - ← Si $i+j = n$, $P(i,j)$ se puede calcular en un tiempo de ejecución

$$T(1) = c$$
$$T(n) = 2T(n-1) + d$$

- Para algunas constantes *c* y *d*. Esta tiene una solución $O(2^n)$ (Tarea)



# Pronósticos: Arbol de Llamados



# Pronósticos: Arbol de Llamados

- Es posible ver que el número total de llamados a *P* es el número de formas de elegir *i* posibilidades entre *i+j*.

$$\binom{i+j}{i}$$

- El valor anterior puede ser acotado por.

$$\Omega\left(2^n/\sqrt{n}\right)$$

cuando *n = i+j, i=j*



# Pronósticos: Tabla de Programación Dinámica

- La tabla siguiente constituye la tabla de programación dinámica para el cálculo de los pronósticos

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1/2 | 21/32 | 13/16 | 15/16 | 1 | 4 |
| | 11/32 | 1/2 | 11/16 | 7/8 | 1 | 3 |
| | 3/16 | 5/16 | 1/2 | 3/4 | 1 | 2 |
| | 1/16 | 1/8 | 1/4 | 1/2 | 1 | 1 |
| | 0 | 0 | 0 | 0 | | 0 |
| | 4 | 3 | 2 | 1 | 0 | |

j

i

- ¿Puede identificar las dependencias entre problemas?

# Pronósticos: Tabla de Programación Dinámica

```
float Pronóstico( int i, int j )
{
    int s,k;

    for( s=1; s<=i+j; s++ ) {
        P[0][s] = 1.0;
        P[s][0] = 0.0;
        for( k=1; k<=s-1; k++ )
            P[k][s-k] = (P[k-1][s-k] + P[k][s-k-1] )/2.0;
    }

    return P[i][j];
}
```

- El tiempo que toma Pronóstico es $O(ij)$. Si $i=j=n$, entonces, es $O(n^2)$



# Multiplicación de una Secuencia Matrices

- Motivación
  - Se quiere multiplicar una secuencia larga de matrices
    - *AxBxCxDxE*
  - La multiplicación de matrices no es *conmutativa*
  - La forma en que se asocian las matrices tiene un efecto importante en el número de multiplicaciones
  - Se quiere evitar crear matrices intermedias grandes
  - Ya que la multiplicación de matrices es asociativa, se pueden colocar paréntesis en cualquier forma

$$\begin{bmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix} = \begin{bmatrix} 13 & 18 & 23 \\ 18 & 25 & 32 \\ 23 & 32 & 41 \end{bmatrix}$$

## Multiplicación de una Secuencia Matrices

- Ejemplo
  - Considere AxBxCxD, donde
    - A es 30x1, B es 1x40, C es 40x10 y D ex 10x25
  - ((AB)C)D = 30x1x40 + 30x40x10 + 30x10x25 = **20,700**
  - (A(BC)D) = 1x40x10 + 30x1x10 + 30x10x25 = **8,200**
  - (AB)(CD) = 30x1x10 + 40x10x25 + 30x40x25 = **41,200**
  - A((BC)D) = 1x40x10 + 1x10x25 + 30x1x25 = **1,400**
  - A(B(CD)) = 40x10x25 + 1x40x25 + 30x1x25 = **11,750**

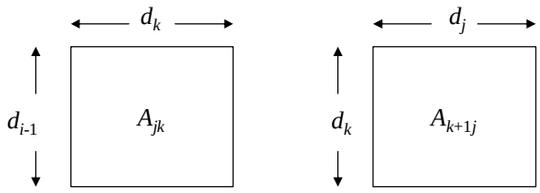


## Parentización Optima

- La forma en que se asocian las matrices tiene un gran impacto en la cantidad de operaciones que se realizan
- ¿Cómo se determina la mejor parentización?
- Sea $M(i, j)$ la menor cantidad de multiplicaciones necesarias para multiplicar $A_i \times A_{i+1} \times ... \times A_j$
- Los paréntesis más externos dividen la secuencia de matrices en algún $k$, $(A_i \times A_{i+1} \times ... \times A_k)$ $(A_{k+1} \times ... \times A_j)$
- La parentización óptima es óptima en ambos lados de $k$

# Parentización Optima

- $(A_i \times A_{i+1} \times \ldots \times A_k)\ (A_{k+1} \times \ldots \times A_j)$



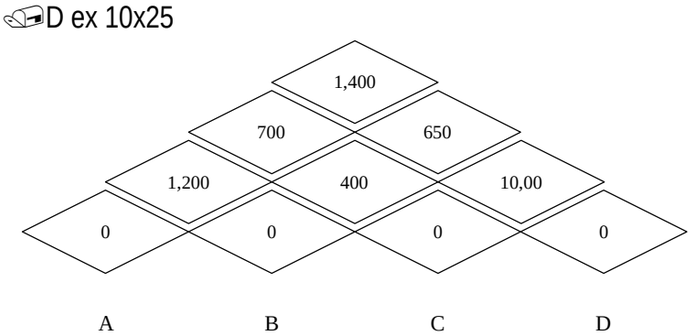

La recurrencia que modela esto es:

$$M(i,j) = \begin{cases} 0 & i = j \\ min_{i \le j < k} \{M(i,k) + M(k+1,j) + d_{i-1} d_k d_j\} & i < j \end{cases}$$



# Parentización Optima

- Ejemplo
  - ← Considere AxBxCxD, donde
    - A es 30x1,
    - B es 1x40,
    - C es 40x10 y
    - D ex 10x25



# Parentización Optima

```
for( i = 1; i <= n; i++ )
    m[i][i] = 0;
for( l = 2; l <= n; l++ ) {
    for( i = 1; i <= n-l+1; i++ ) {
        j = i + l - 1;
        m[i][j] = ∝
        for( k = i; k <= j-1; k++ ) {
            q = m[i][k] + m[k+1][j] + d_{i-1}d_k d_j
            if( q < m[i][j] )
                m[i][j] = q
        }
    }
}
```

- Se llena la matriz por diagonales (l)
- Cada diagonal tiene (n-l) elementos
- El tiempo de ejecución es $O(n^3)$



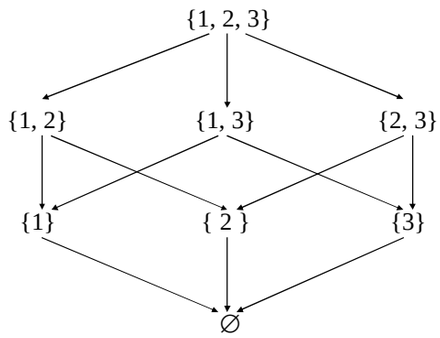
# Problema 0-1 Knapsack

- Llenar un saco de una resistencia fija con los objetos más valiosos. Se toma el objeto completo o se rechaza
  - ← Dado $n$ elementos. Sea
    - $S = \{ o_1, o_2, ..., o_n \}$
    - $w_i$ el peso del objeto $o_i$, $1 \leq i \leq n$
    - $p_i$ el valor del objeto $o_i$, $1 \leq i \leq n$
    - $W$ el peso máximo que el saco puede resistir, $W > 0$
    - maximizar

$$\sum_{o_j \in A} p_j, \quad \text{dado que,} \quad \sum_{o_j \in A} w_j \leq W$$

# Problema 0-1 Knapsack

- Fuerza bruta:
  - Generar todos los subconjuntos de objetos
  - Olvidarse de aquellos que exceden a *W*
  - Seleccionar el de mayor valor de los restantes
  - ¿Cuantos subconjuntos de conjunto de *n* elementos hay?



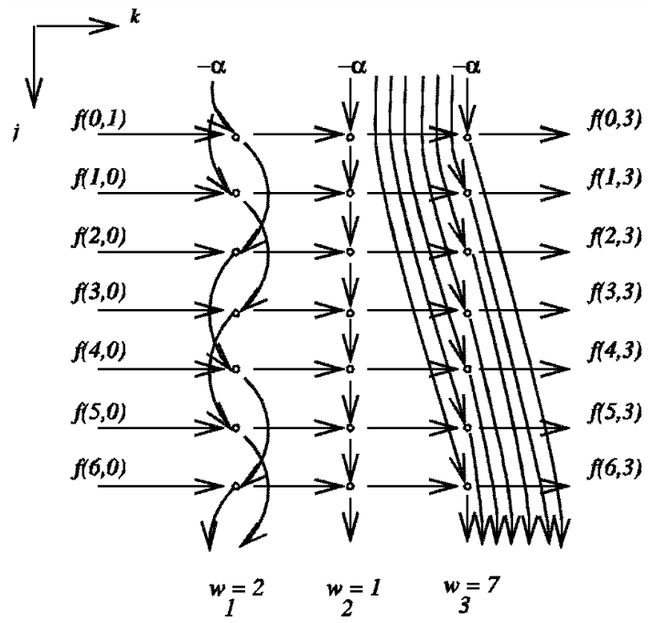



# 0-1 Knapsack: Programación Dinámica

- Sea *A* un conjunto óptimo de *n* elementos. Existen dos casos
  - Si *A* no contiene al objeto $o_n$, entonces, *A* es igual a un subconjunto óptimo de *n*-1 elementos
  - Si *A* contiene al objeto $o_n$, entonces, *A* es igual a $p_n$ más el valor de un subconjunto óptimo de *n*-1 elementos, bajo la restricción de que el peso no debe exceder a $W - w_n$.

$$P[i,w] = \begin{cases} max(P[i-1,w], P[i-1,w-w_i]) & \text{si } w_i \le w \\ P[i-1,w] & \text{si } w_i > w \end{cases}$$

# 0-1 Knapsack: Ejemplo






# 0-1 Knapsack: Programación Dinámica

- La forma de llenar la tabla $P$ es directa. ¿Cuál es el tamaño de la tabla?
- El tamaño del arreglo es el número de objetos x el peso máximo
- El algoritmo toma un tiempo $O(nW)$
- Se puede mostrar que para el problema 0-1 Knapsack el tiempo de ejecución es $O(\min(2^n, nW))$
- Nadie ha encontrado un algoritmo cuyo peor caso sea mejor que el exponencial y nadie ha probado que tal algoritmo no es posible

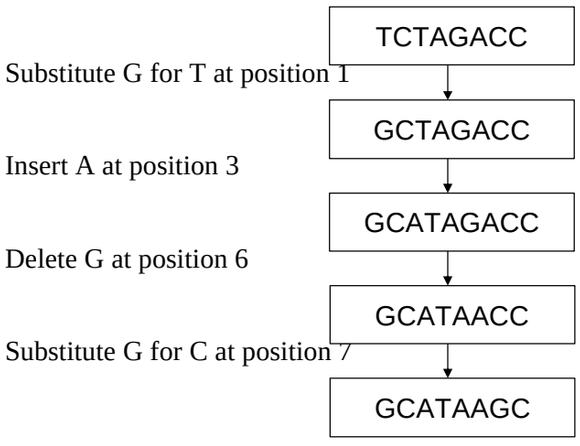
# DNA String Matching

- Genetic sequence data are being generated at an ever increasing rate
  - ← Human Genome Inititative
  - ← Genetic Sequencing Technology
- New sequences are obtained to be classified and compared to existing databases
  - ← Great databases require faster methods of comparing sequences



# Edit Distance

List of operations to transform TCTAGACC into GCATAAGC



# Dynamic Programming Algorithm

- Let $S = [s_1 s_2 ... s_m]$ be the source sequence and $T = [t_1 t_2 ... t_n]$ be the target sequence, and $d_{ij}$ the distance between the subsequences $[s_1 s_2 ... s_i]$ and $[t_1 t_2 ... t_j]$
- Para $1 \le i \le m$, $1 \le j \le n$,
  - ← $\Psi(s_i, \varnothing)$ is the cost of deleting $s_i$.
  - ← $\Psi(\varnothing, t_j)$ is the cost of inserting $t_j$.
  - ← $\Psi(s_i, t_j)$ is the cost of substituting $t_i$ by $s_i$.

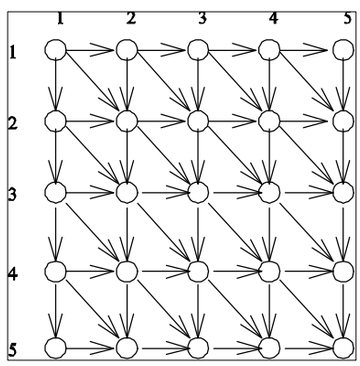
$$d_{00} = 0,$$
$$d_{i0} = d_{i-1\,j} + \Psi(s_i, \varnothing)$$
$$d_{0\,j} = d_{ij-1} + \Psi(\varnothing, t_j)$$

$$d_{ij} = min \begin{cases} d_{i-1\,j} + \Psi(s_i, \varnothing) \\ d_{ij-1} + \Psi(\varnothing, t_j) \\ d_{i-1\,j-1} + \Psi(s_i, t_j) \end{cases}$$



# Ejemplo

| | | G | C | A | T | A | A | G | C |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| T | 1 | 2 | 3 | 4 | 3 | 4 | 5 | 6 | 7 |
| C | 2 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 6 |
| T | 3 | 4 | 3 | 4 | 3 | 4 | 5 | 6 | 7 |
| A | 4 | 5 | 4 | 3 | 4 | 3 | 4 | 5 | 6 |
| G | 5 | 4 | 5 | 4 | 5 | 4 | 5 | 4 | 5 |
| A | 6 | 5 | 6 | 5 | 6 | 5 | 4 | 5 | 6 |
| C | 7 | 6 | 5 | 6 | 7 | 6 | 5 | 6 | 5 |
| C | 8 | 7 | 6 | 7 | 8 | 7 | 6 | 7 | 6 |



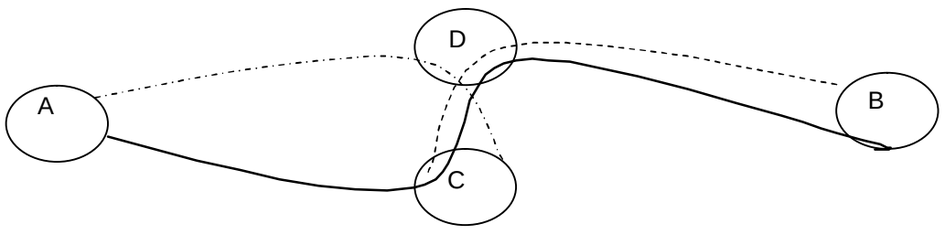
# Principio de Optimalidad

- Principio de Optimalidad
  - La estrategia de programación dinámica se aplica con frecuencia a problemas de optimización
  - Ejemplo: Encuentre el camino más corto entre un par de ciudades
  - *Dada una secuencia óptima de opciones, cada subsecuencia debe ser también óptima*
  - No todos los problemas de optimización siguen el principio de optimalidad



# Principio de Optimalidad

- ¿Cuál es el camino más corto entre la ciudades A y B?
  - Simple: nunca visite el mismo punto intermedio más de dos veces



# Cálculo del Coeficiente Binomial

- Coeficiente Binomial
  - Mediante la siguiente relación es posible calcular el coeficiente binomial

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \\ 0 & \text{en otro caso} \end{cases}$$

  - Escriba una función recursiva para el cálculo del coeficiente binomial
  - Determine una mejor manera de calcular el coeficiente
  - Calcule el tiempo de ejecución del algoritmo de programación dinámica



# Algoritmos Avidos

- Un ***algoritmo ávido*** inicia con una solución a un pequeño subproblema y construye una solución a un problema mayor fijándose únicamente en las ganancias locales

- Ejemplo: Suponga que se tienen monedas de 25, 10 y 5 y 1 pesos con las cuales se desea dar un cambio de 63 pesos, con el menor número de monedas

  - **63 = 2 x 25 + 1 x 10 + 3 x 1**

## Algoritmos Avidos

- **63 = 2 x 25 + 1 x 10 + 3 x 1**
  - Algoritmo
    - Hacer *c* = 63
    - Seleccionar la moneda con la denominación más grande, *m*, tal que, *m* no es mayor que *c*.
    - Hacer *c* = *c*-*m*
    - Si *c* > 0, repetir el paso 2
  - El algoritmo es ávido debido a que en el paso 2 se selecciona la moneda que es localmente óptima

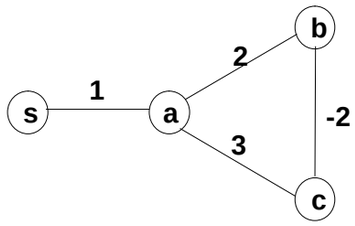


## Algoritmos Avidos

- Nuevo ejemplo: Suponga que se tienen monedas de 1, 5 y 11 pesos. Se desea dar un cambio de 15 pesos. El algoritmo anterior produciría la solución
  - 15 = 1 x 11 + 4 x 1
  - Sin embargo, la solución óptima es:
    - 15 = 3 x 5
  - Un algoritmo ávido no siempre produce la solución óptima

# Algoritmos Avidos

☞El algoritmo de Dijkstra, para encontrar el camino más corto desde un origen, es un algoritmo ávido. Pues, en cada paso se elige el vértice más cercano al origen entre aquellos cuyo camino más corto aún no se conoce




Algoritmo de Dijkstra produce:
$s \rightarrow a \rightarrow b$, de costo 2

Camino óptimo
$s \rightarrow a \rightarrow c \rightarrow b$, de costo 1



# Algoritmos Avidos: Estructura General

☞ Estrategia general

- ←El algoritmo ávido inicia con un conjunto vacío y agrega elementos en secuencia hasta que el conjunto representa una solución
- ←**Procedimiento de selección**: Elige el siguiente elemento a agregar al conjunto. La selección se basa en un criterio de avidez que satisface algún criterio localmente óptimo
- ←**Verificación de factibilidad**: Verifica si el nuevo conjunto puede conducir a una solución
- ←**Verificación de solución**: Verifica si el nueve conjunto es ciertamente una solución

## Algoritmos Avidos vs. Programación Dinámica

- Algoritmos ávidos vs. Programación Dinámica
  - Ambas son estrategias para resolver problemas de optimización
  - Se pueden aplicar ambas al mismo problema
    - El algoritmo ávido es, generalmente, más eficiente pero no garantiza optimalidad
    - Para algoritmos de programación solo se tiene que mostrar que obedece el principio de optimalidad



## 0-1 Knapsack: Algoritmo Avido

- Estrategia: Ordenar los objetos por valor por unidad de peso. Seleccionar los objetos en orden decreciente
  - S = { ($o_1$, 5 Kg, \$50), ($o_2$, 10 Kg, \$60), ($o_3$, 20 Kg, \$140) },
  - $W$ = 30 Kg
  - Solución ávida:
    - $A$ = {($o_1$, 5 Kg, \$50), ($o_3$, 20 Kg, \$140) }, Residuo = 5 Kg, Valor = \$190
  - Solución óptima:
    - $A$ = {($o_2$, 10 Kg, \$60), ($o_3$, 20 Kg, \$140), Residuo = 0 Kg, Valor = \$200

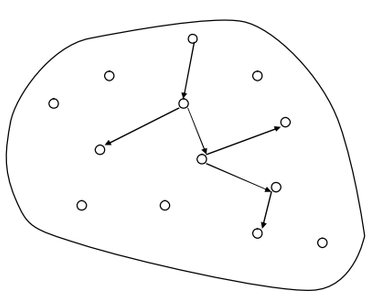
# Knapsack Fraccional

- Knapsack fraccional
  - Es posible seleccionar parte de un objeto
  - Problema anterior: solución ávida
    - $A$ = { ($o_1$, 5 Kg, \$50), ($o_3$, 20 Kg, \$140), ($o_2$, 5 Kg, \$30), }, Valor = \$230
  - Para mostrar que el algoritmo ávido siempre produce la solución óptima para el problema fraccional, se tendría que probar que:
    - si los objetos se seleccionan en el orden decreciente $p_i/w_i$, entonces, se encuentra la solución óptima
  - Puede haber más de una solución óptima



# Espacio de Búsqueda

- La solución de un problema se puede modelar como la búsqueda de un estado particular en un espacio de estados
- El espacio de estados se puede representar mediante una gráfica (dirigida o no dirigida) en donde los vértices son estados y los arcos representan formas de llegar a otro estado



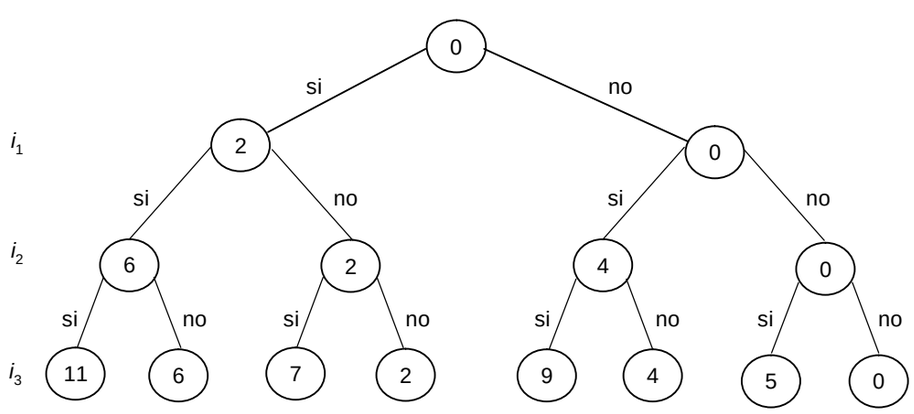
# Recorridos

- Las búsqueda de una solución se representa como el recorrido en una gráfica
  - Recorrido en profundidad (Depth-First Search)
  - Recorrido en amplitud (Breadth-First Search)
  - Recorrido sobre el mejor (Best-First Search)
- El recorrido de una gráfica da origen a un árbol,
  - el árbol del recorrido
- El árbol del recorrido se interpreta como el árbol de búsqueda

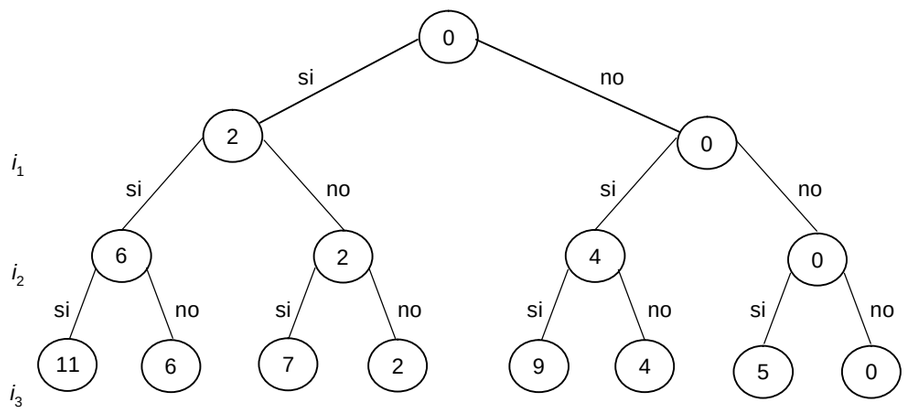


# Arbol de Búsqueda para 0-1 Knapsack

- Arbol de búsqueda para el espacio de estados del problema 0-1 Knapsack
  - Tres elementos: $i_1 = 2$, $i_2 = 4$, $i_3 = 5$



# Arbol de Búsqueda para 0-1 Knapsack




¿Cuál es la profundidad del árbol de búsqueda para el problema 0-1 Knapsack con $n$ nodos?



# Backtracking

- Se utiliza un árbol implícito para modelar todas las soluciones de un problema
- La solución del problema se construye mediante un recorrido de la raíz a una de sus hojas
- El visitar un nodo del árbol corresponde a agregar un objeto a una solución parcial
- Backtracking opera como una búsqueda en profundidad sobre una gráfica dirigida
- Backtracking difiere de la búsqueda en profundidad en que las ramas que no proporcionan solución alguna no son visitadas

## Backtracking

```
void checknode( node v )
{
   if( promisorio( v ) )
      if( existe una solución en v )
         escribe la solución;
      else
         for( cada hijo u de v )
            checknode( u );
}
```



## Suma de Subconjuntos

☞ Definición

← Dado un conjunto finito R $\subset$ **N** y un entero positivo *W*, el problema es encontrar el subconjunto S $\subset$ R cuyos elementos sumen *W*.

← Nota: si se obtiene un subconjunto, es muy fácil probar que es una solución. Sin embargo, existe $2^{|R|}$ posibles subconjuntos

Considere el siguiente ejemplo:

$i_1 = 3$, $i_2 = 4$, $i_3 = 5$, $i_4 = 6$; $W = 13$

# Suma de Subconjuntos

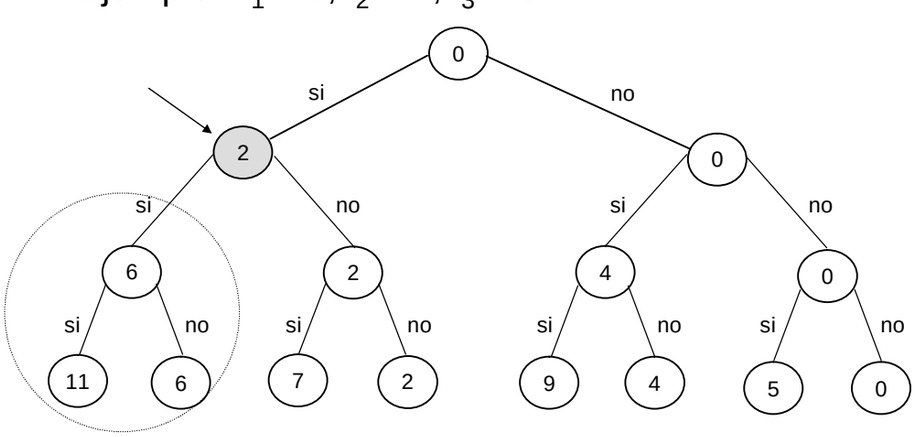
- ¿Cómo se puede probar que un nodo no es promisorio?
  - Sea $W_t$ la suma total de los pesos de los elementos de $R$
  - Sea $Peso_i$ la suma total de los pesos de los elementos $s_1, ..., s_i$
  - Sea *PesoActual* la suma total de los pesos de los elementos ***incluidos*** hasta el elemento $i$
  - Sea $PesoDisponible = W_t - Peso_i$, esto es, la suma de los pesos de los objetos que no han sido considerados

- Un nodo **no es promisorio** si
  - $PesoActual + w_{i+1} > W$, y
  - $PesoActual + PesoDisponible < W$



# Backtracking: Ejemplo

Un nodo no es promisorio si
$PesoActual + w_{i+1} > W$, y
$PesoActual + PesoDisponible < W$
Ejemplo. $i_1 = 2$, $i_2 = 4$, $i_3 = 5$. $W = 4$

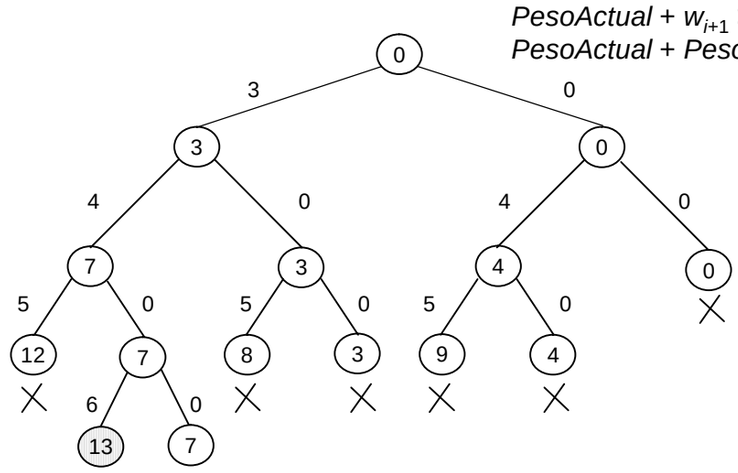


# Backtracking: Ejemplo

Considere el siguiente ejemplo:

$i_1 = 3, i_2 = 4, i_3 = 5, i_4 = 6;\ W = 13$

*PesoActual* + $w_{i+1}$ > *W*, o
*PesoActual* + *PesoDisponible* < *W*






# Suma de Conjuntos: Algoritmo

```
SumadeConjuntos( i, PesoActual, PesoDisponible )
{
1) if( promisorio( i ) )
2)  if( PesoActual == W )
3)    print include[ 1 ] to include[ i ];
4)  else {
5)    include[ i + 1 ] = ”si”:
6)    SumadeConjuntos( i + 1, PesoActual + w[i + 1],
                       PesoDisponible - w[i + 1] );
7)    include[ i + 1 ] = "no”;
8)    SumadeConjuntos( i + 1, PesoActual,
                              PesoDisponible - w[i + 1] );
    }
}
bool promisorio( i  )
{
1) return (PesoActual + PesoDisponible ≥ W)  &&
          (PesoActual == W  ||  PesoActual +  w[i+1] ≤ W);
}
```

# Backtracking: Estructura General

*best* es el mejor valor hasta el momento; es iniciado a un valor que es peor que cualquier solución posible

*value*(*v*) es el valor de la solución en el nodo

```
void checknode( node v )
{
   node u;
   if( value(v) es mejor que best )
      best = value(v);
   if( promisorio( v ) )
      for( cada hijo u de v )
         checknode( u );
}
```



# 0-1 Knapsack Problem

- *weight*: es la suma de los pesos de los objetos incluidos hasta algún nodo
  - ← El nodo no es promisorio (no se quiere expandir sus hijos) si
    - ☜ $weight \geq W$
- *profit*: es la suma de los valores de los objetos incluidos hasta algún nodo
  - ← Se ordenan los objetos en orden no creciente de acuerdo a los valores $p_i / w_i$ .

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

$$bound = \left( profit + \sum_{j=i+1}^{k-1} p_j \right) + (W - totweight) \times (p_k / w_k)$$

# Backtracking: 0-1 Knapsack

☞ Un nodo también no es promisorio si *bound* ≤ *maxprofit*

← *maxprofit* es el valor de *profit* de la mejor solución hasta el momento

☞ Supongamos que $n = 4$, $W = 16$, y se tiene lo siguiente:

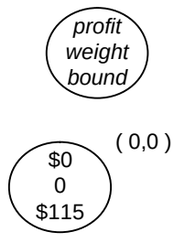
| $i$ | $p_i$ | $w_i$ | $p_i / w_i$ |
|---|---|---|---|
| 1 | $40 | 2 | $20 |
| 2 | $30 | 5 | $6 |
| 3 | $50 | 10 | $5 |
| 4 | $10 | 5 | $2 |



# Ejemplo

☞ $n = 4$, $W = 16$

| $i$ | $p_i$ | $w_i$ | $p_i / w_i$ |
|---|---|---|---|
| 1 | $40 | 2 | $20 |
| 2 | $30 | 5 | $6 |
| 3 | $50 | 10 | $5 |
| 4 | $10 | 5 | $2 |

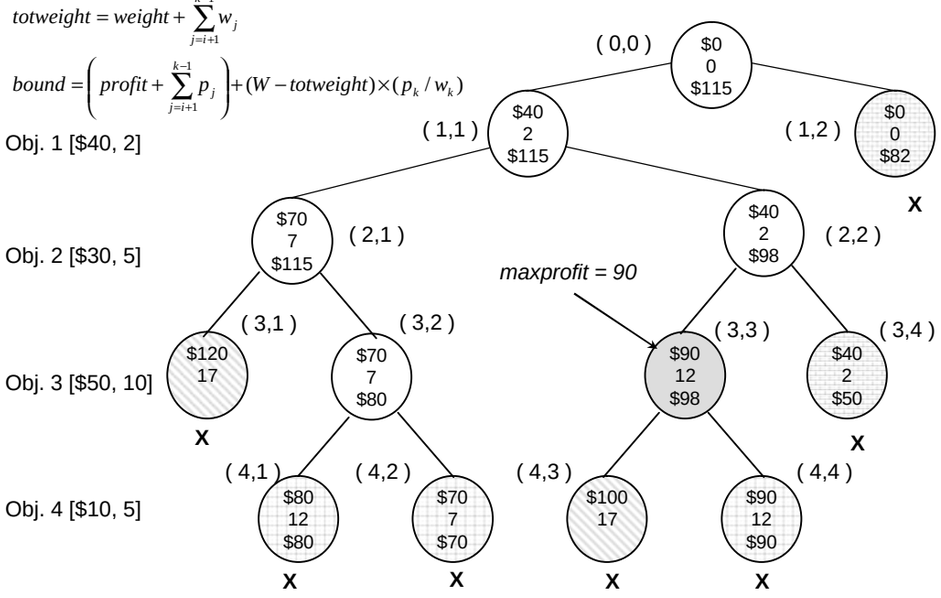



1) Defina *maxprofit* a $0 (Recuerde $n = 4$, $W = 16$ )
2) Visite el nodo (0,0)
   a) Calcule profit y weight: *profit* = $ 0, *weight* = 0
   b) Calcule su cota. Ya que $w_1 + w_2 + w_3 = 2 + 5 + 10 = 17 > 16$.
      Por lo tanto, $k = 3$ ( para mantener la suma de los pesos <= 16).
      Así que *totweight* = *weight* + $w_1 + w_2$ = 0 + 2 + 5 = 7
      *bound* = *profit* + $p_1 + p_2$ + (*W* - *totweight* ) X $p_3 / w_3$
      = $0 + $40 + $30 + (16 -7) X $50/10 =$115
   c) Es promisorio ya que su peso es igual a 0 < *W* = 16, y su cota es $115 > 0, el valor actual de *maxprofit.*

# Ejemplo: Arbol de Búsqueda

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

$$bound = \left( profit + \sum_{j=i+1}^{k-1} p_j \right) + (W - totweight) \times (p_k / w_k)$$

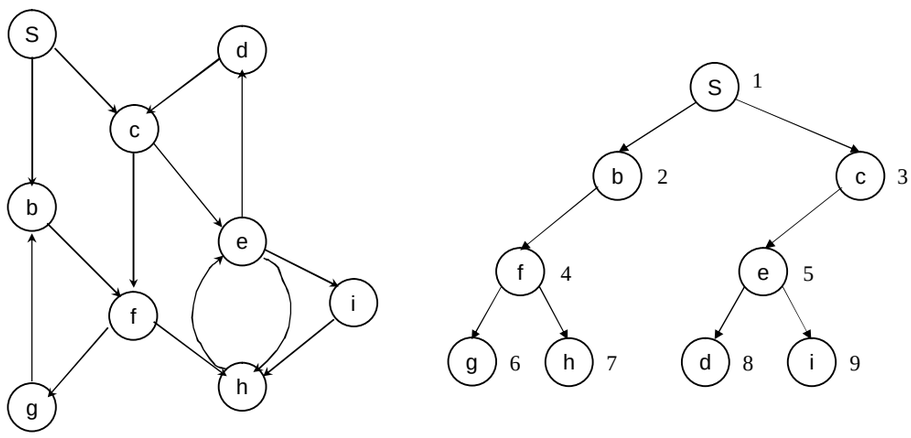





# Recorrido en Amplitud

```
Q es una cola
void BFS ( S )
{
1. for( cada u ∈ V - { S } )
2.     d[ u ] ← ▤ ;
3.  d[ S ] ← 0;
4.  Enqueue(Q, S);
5.  while(  Q ≠ ∅ ) {
6.     Dequeue( Q, u );
7.     for( cada v ∈ Adj [ u ] )
8.         if( d [ v ] = ▤) {
9.             d [v ] = d [u ] + 1;
10.            Enqueue ( Q, v );
           }
    }
}
```

# Recorrido en Amplitud: Ejemplo






# Branch and Bound

- Se utiliza un árbol implícito para modelar todas las soluciones posibles de un problema
- No está limitado a una forma particular de recorrer el árbol
- Se utiliza únicamente para problemas de optimización
- La solución del problema se construye mediante un recorrido de la raíz a una de sus hojas
- El visitar un nodo del árbol corresponde a agregar un objeto a una solución parcial
- Branch and bound opera como una recorrido en amplitud (breadth first search) sobre una gráfica dirigida
- Los nodos generados se visitan
  - de acuerdo al orden en que fueron creados
  - se ordenan y se visita primero el que contiene el mejor valor para el problema

## Branch and Bound con Breadth First Search

```
void BFSBranch&Bound (tree T, number &best)
{
    queueOfNodes Q;
    node u, v ;
    v = root of T ;
    enqueue( Q, v );
    best = value(v);

    while( Q ≠ ∅ ) {
       dequeue (Q ,u );
       for( each v ∈ child [ u ] ) {
           if( value( v ) is better than best )
               best = value (v);
           if( bound(v) is better than best )
               enqueue (Q ,v );
       }
    }
}
```



## Branch and Bound con Best First Search

```
void BFSBranch&Bound (tree T, number &best)
{
    priorityQueue PQ;
    node u, v ;
    v = root of T ;
    best = value(v);
    insert( PQ, v );

    while( PQ ≠ ∅ ) {
       u = ExtractMinimum( PQ );
       if( bound(u) is better than best )
           for( each v ∈ child [ u ] ) {
               if( value( v ) is better than best )
                   best = value (v);
               if( bound(u) is better than best )
                   insert( PQ ,v );
           }
    }
}
```

# Ejemplo: Arbol de Búsqueda

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j$$

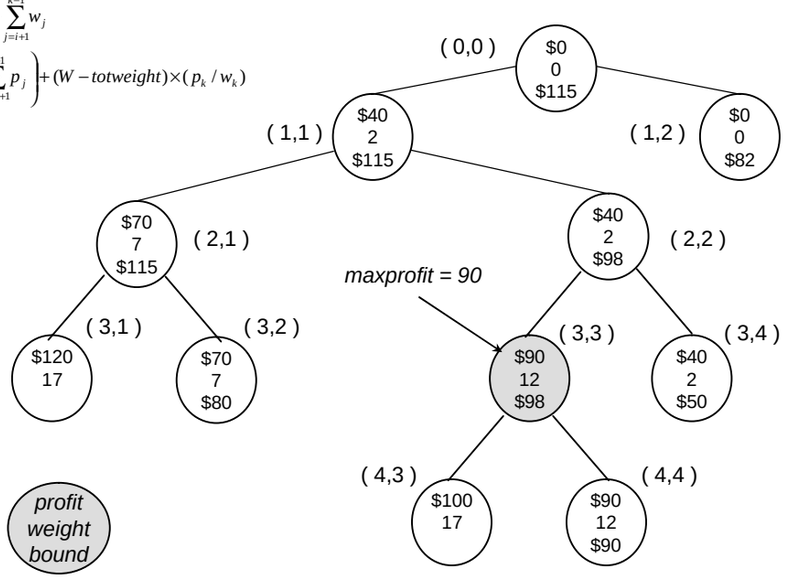
$$bound = \left( profit + \sum_{j=i+1}^{k-1} p_j \right) + (W - totweight) \times (p_k / w_k)$$

Obj. 1 [$40, 2]

Obj. 2 [$30, 5]

Obj. 3 [$50, 10]

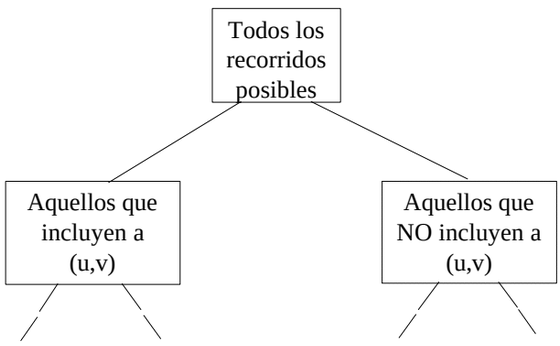
Obj. 4 [$10, 5]



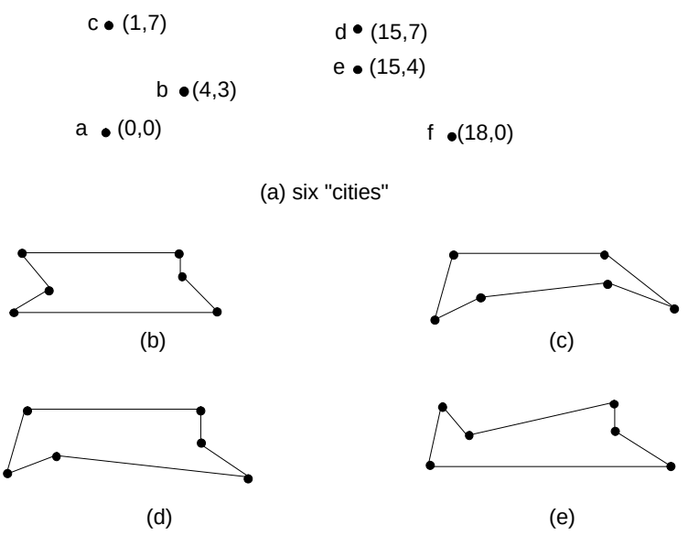



# El Problema del Agente Viajero

- Estrategia Branch and Bound
  - Iniciar construyendo un árbol cuya raíz representa a todos los recorridos posibles de las *n* ciudades
  - Cada nodo tiene dos hijos y los recorridos que representa cada nodo constituyen dos grupos: aquellos que tienen un arco particular y aquellos que no lo tienen

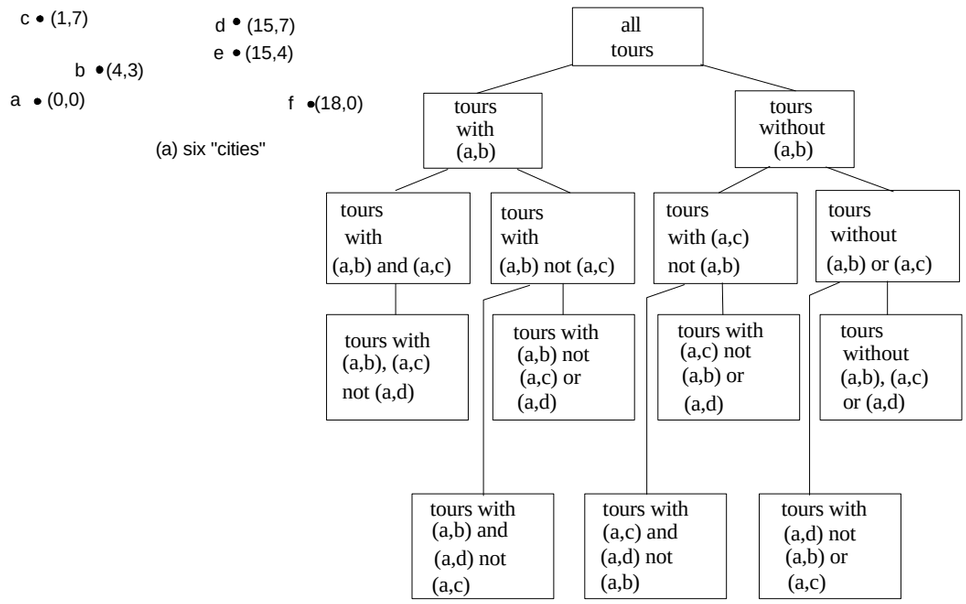


# TSP: Ejemplo 1

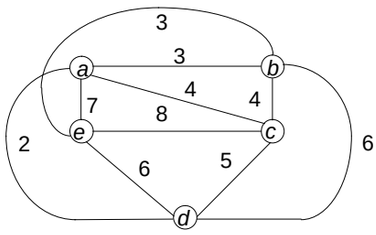





# TSP: Ejemplo 1



# TSP: Ejemplo 2



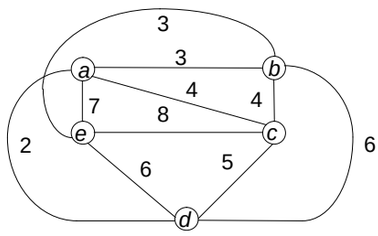

Los dos arcos de menor costo para cada vértice

| NODO | $a_1$ | $c_1$ | $a_2$ | $c_2$ | total |
|---|---|---|---|---|---|
| a | (a,d) | 2 | (a,b) | 3 | 5 |
| b | (a,b) | 3 | (b,e) | 3 | 6 |
| c | (a,c) | 4 | (b,c) | 4 | 8 |
| d | (a,d) | 2 | (c,d) | 5 | 7 |
| e | (b,e) | 3 | (d,e) | 6 | 9 |

Una cota inferior para un recorrido es: $\frac{5+6+8+7+9}{2} = 17.5$



# TSP: Ejemplo 2




Supongamos que el recorrido debe incluir al arco (*a*, *e*) y debe excluir al arco (*b*, *c*)

| NODO | $a_1$ | $c_1$ | $a_2$ | $c_2$ | total |
|---|---|---|---|---|---|
| a | (a,d) | 2 | (a,e) | 7 | 9 |
| b | (a,b) | 3 | (b,e) | 3 | 6 |
| c | (a,c) | 4 | (c,d) | 5 | 9 |
| d | (a,d) | 2 | (c,d) | 5 | 7 |
| e | (b,e) | 3 | (a,e) | 7 | 10 |

La cota inferior para un recorrido se actualiza a: $\frac{9+6+9+7+10}{2} = 20.5$

## Heurísticas para TSP

- Cuando se consideran los hijos de un nodo, se trata de hacer inferencias sobre cuales arcos incluir o excluir del recorrido representado por los nodos
  - Si se excluyera un arco $(x, y)$ se hace imposible que $x$ o $y$ tengan dos arcos incidentes en el recorrido, entonces, $(x, y)$ debe ser incluido
  - Si al incluir $(x, y)$ se provoca que $x$ o $y$ tengan más de dos arcos incidentes en el recorrido, entonces se debe excluir a $(x, y)$



## Heurísticas para TSP

- Después de hacer las inferencias se pueden calcular las cotas inferiores para cada hijo
  - Si la cota inferior para un hijo es mayor que el menor costo encontrado hasta el momento para un recorrido, entonces, se puede podar ese hijo y no se necesita considerar a sus descendientes
  - Si ningún hijo se puede podar, considerar primero el hijo con la menor de las cotas inferiores.
    - Después de considerar a un hijo, se debe ver si su hermano puede ser podado, ya que pudo haberse encontrado una solución mejor

# Recorrido Branch and Bound