# Diseño II: Backtracking Programación dinámica Algoritmos greedy

M. Andrea Rodríguez-Tastets
Ayudante: Erick Elejalde

Universidad de Concepción,Chile
www.inf.udec.cl\ ∼andrea
andrea@udec.cl

I Semestre - 2013

# Introducción

## Backtracking

- Backtracking es una forma sistemática de iterar sobre TODAS las posibles configuraciones de un espacio de búsqueda. Estas configuraciones pueden ser posibles permutaciones o subconjuntos del espacio.
- Estos problemas generan cada una de las posibles configuraciones UNA VEZ.
- Backtracking construye un árbol de soluciones parciales, donde cada vértice representa una solución parcial. Existe un arco desde $x$ a $y$ si es que $y$ fue creado por un avance o extensión de $x$.
- El proceso de construir una solución corresponde exactamente a un recorrido en profundidad (depth-first) del árbol.

## Backtracking (2)

Se puede modelar la búsqueda combinatorial del backtracking como sigue:

- Una solución de la búsqueda es un vector $a = (a_1, \ldots, a_n)$, donde cada elemento $a_i$ es seleccionado de un conjunto finito y ordenado $S_i$.
- A cada paso del backtracking, se trata de extender el vector $a = (a_1, \ldots, a_k)$ agregando un elemento al final. Si es una solución, entonces se debe reportar como tal.

## Backtracking (3)

Algoritmo

```
Backtracking-DFS(A,k)
begin
  if A = (a_1, ..., a_k) es una solución then
    return A
  else
    k = k + 1
    calcule (construya) S_k
    while S_k ≠ 0 do
      a_k = un elemento en S_k
      S_k = S_k − a_k
      Backtracking−DFS(A,k)
    endwhile
  endif
```

## Backtracking (4)

- Backtracking asegura correctitud al enumerar todas las posibilidades.
- Sin embargo, puede ser muy costoso construir todas las posibilidades (permutaciones) primero y luego analizarlas. Asuma que se parte por un nodo $n_1$ del árbol de soluciones y que no existe un arco entre nodo $n_1$ y $n_2$. Entonces las próximas $(n-2)!$ permutaciones que siguen de $(n_1, n_2)$ no son soluciones.
- Otra forma de evitar una búsqueda exhaustica es explotar simetría donde soluciones parciales son idénticas para distintos caminos y evitar la computación repetitiva de soluciones parciales (dynamic programming).
- Otra técnica es *pruning*, la cual que corta ciertos caminos de búsqueda en el árbol de solución (esto es usado en greedy).

# Programación dinámica

## Introducción

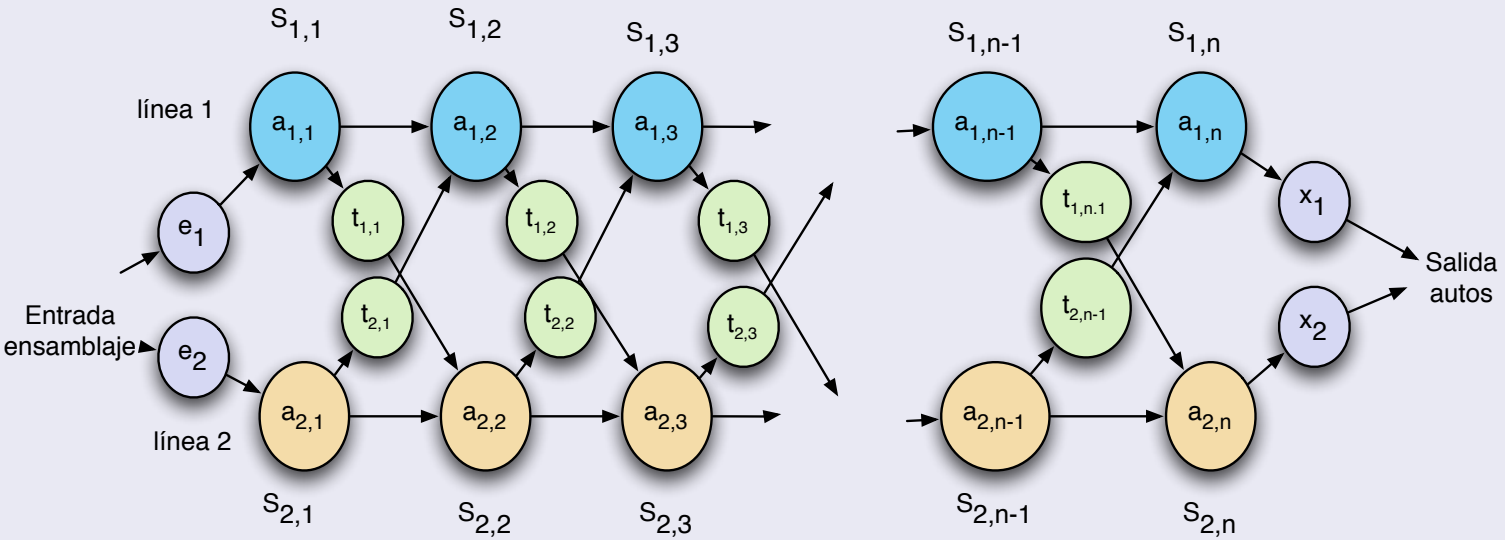
- La programación dinámica, al igual que la estrategia de *Dividir para Vencer*, resuelve problemas por la combinación de soluciones a subproblemas.
- Sin embargo, a diferencia de Dividir para Vencer, la programación dinámica es aplicable cuando los subproblemas comparten subsubproblemas.
- La programación dinámica resuelve cada subsubproblema solo una vez y salva la respuesta en una tabla, evitando el trabajo de recalcular la respuesta cada vez que el subsubproblema se encuentra.
- La programación dinámica es usualmente aplicada a problemas de optimización, donde se necesitan hacer ciertas selecciones para llegar a una solución óptima, y donde al hacer una selección, subproblemas de la misma forma deben resolverse.

## Introducción (cont.)

El desarrollo de un algoritmo que use la técnica de programación dinámica puede ser desglosado en los siguientes pasos

- Caracterización de la estructura de una solución óptima.
- Definir recursivamente el valor de una solución óptima.
- Calcular el valor de una solución óptima en un esquema bottom-up
- Construir una solución óptima a partir de la información calculada. Este último paso puede ser omitido si solo el valor de una solución óptima es requerido.

## Ejemplo 1: Línea de ensamblaje




Hay $2^n$ formas diferentes de escoger estaciones en una línea de ensamblaje, lo que es imposible para un número muy grande de $n$.

## Línea de ensamblaje: Estructura de una solución óptima

Para determinar la forma más rápida de que un chassis llegue desde la entrada y pase por la estación $S_{1,j}$.

- Si $j = 1$, solo existe una forma posible y el tiempo en conocido $(e_{1,1} + a_{1,1})$.
- Si $j > 1$, hay dos formas posibles:

  (i) Venir de la estación $S_{1,j-1}$, con un tiempo negligente de pasar de $S_{1,j-1}$ a $S_{1,j}$.

  (ii) Venir de estación $S_{2,j-1}$ y transferirse a la línea 1 con un tiempo de transferencia de $t_{2,j-1}$.

  En ambos casos, una solución óptima en $S_{1,j}$ pasa por una solución óptima del chassis desde la entrada a la estación $S_{1,j-1}$ o $S_{2,j-1}$. De lo contrario, se podría sustituir la forma más rápida de llegar a $S_{1,j}$, lo que es una contradicción. A esto se le llama **subestructura óptima**.

## Línea de ensamblaje: Estructura de una solución óptima (cont.)

Se usa la subestructura óptima para mostrar que uno puede reconstruir una solución óptima a un problema desde soluciones óptimas de los subproblemas.

En este caso, la forma más rápida de pasar por $S_{1,j}$ es una de las dos siguientes opciones:

- La forma más rápida de pasar por $S_{1,j-1}$ y entonces directamente por $S_{1,j}$.
- La forma más rápida de pasar por $S_{2,j-1}$ y entonces transferirse a línea 1 y pasar por $S_{1,j}$.

## Línea de ensamblaje: Solución recursiva

Sea $f_i[j]$ el menor tiempo posible de tomar un chassis desde la entrada y pasarlo por $S_{i,j}$. El objetivo final es obtener $f^*$ que representa el tiempo total de pasar por todas las estaciones. Para ello, el chassis debe pasar por todas las estaciones a través de la línea 1 o línea 2. En forma recursiva:

$$
\begin{aligned}
f^* &= min(f_1[n] + x_1, f_2[n] + x_2) \\
f_1[j] &= \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j} & \text{if } j \geq 2 \end{cases} \\
f_2[j] &= \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j} & \text{if } j \geq 2 \end{cases}
\end{aligned}
$$

## Línea de ensamblaje: Calculando el mejor tiempo

Uno podría ahora pensar en escribir un algoritmo recursivo, pero lamentablemente este algoritmo correría en tiempo exponencial en $n$, $\Theta(2^n)$.

Se puede hacer mucho mejor considerando que para $j \geq 2$, cada valor $f_i[j]$ depende solo de los valores de $f_1[j-1]$ y $f_2[j-1]$. Calculando $f_i[j]$ en orden creciente en el número de la estación (izquierda a derecha), se puede determinar la forma más rápida de pasar por la fábrica, y eso toma $\Theta(n)$.

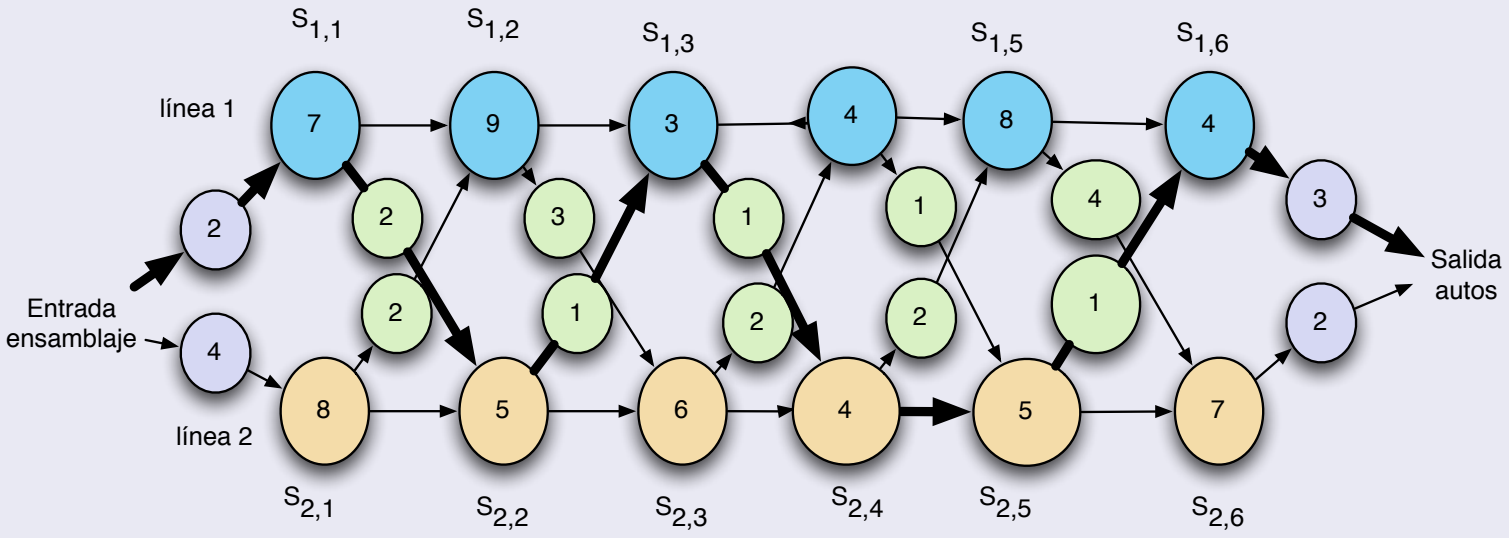
# Línea de ensamblaje: Algoritmo

En este algoritmo $a$, $t$, $e$, $x$ representan arreglos de valores de tiempo en una estación $a_{i,j}$, tiempo de transferencia $t_{i,j}$, tiempo de entrada $e_i$ y tiempo de salida $x_i$.

*Ensamblaje*$(a, t, e, x, n)$

**begin**
  $f_1[1] \leftarrow e_1 + a_{1,1}$
  $f_2[1] \leftarrow e_2 + a_{2,1}$
  **for** $j \leftarrow 2$ **to** $n$ **do**
    **if** $f_1[j-1] + a_{1,j} \leq f_2[j-1] + t_{2,j-1} + a_{1,j}$ **then**
      $f_1[j] \leftarrow f_1[j-1] + a_{1,j}$
      $l_1[j] \leftarrow 1$
    **else** $f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$
      $l_1[j] \leftarrow 2$
    **endif**
    **if** $f_2[j-1] + a_{2,j} \leq f_1[j-1] + t_{1,j-1} + a_{2,j}$ **then**
      $f_2[j] \leftarrow f_2[j-1] + a_{2,j}$
      $l_2[j] \leftarrow 2$
    **else** $f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$
      $l_2[j] \leftarrow 1$
    **endif**
  **endfor**
  **if** $f_1[n] + x_1 \leq f_2[n] + x_2$ **then**
    $f^* \leftarrow f_1[n] + x_1$
    $l^* \leftarrow 1$
  **else** $f^* \leftarrow f_2[n] + x_2$
    $l^* \leftarrow 2$
  **endif**
**end**

# Línea de ensamblaje: algoritmo (cont.)




| j | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $f_1[j]$ | 9 | 18 | 20 | 24 | 32 | 35 |
| $f_2[j]$ | 12 | 16 | 22 | 25 | 30 | 37 |

$f^* = 38$

| | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| $l_1[j]$ | 1 | 2 | 1 | 1 | 2 |
| $l_2[j]$ | 1 | 2 | 1 | 2 | 2 |

$l^* = 1$

## Línea de ensamblaje: Construcción de una solución óptima

Teniendo los valores de $f_i[j]$, $f^*$, $l_i[j]$ y $l^*$m se puede ahora reconstruir la secuencia de estaciones usando la trayectoria más rápida del ensamblaje.

*Imprime*

```
begin
  i ← l*
  printf "línea "i, "estación "n
  for j ← n downto 2 do
    i ← l_i[j]
    printf "línea "i, "estación "j − 1
  enddo
end
```

## Multiplicación de matrices

Asuma una secuencia (cadena) de $n$ matrices $(A_1, A_2, \ldots, A_n)$ para ser multiplicadas: $A_1 A_2 \ldots A_n$. Este problema se puede resolver usando las funciones de multiplicación entre dos matrices cuando las matrices han sido organizadas con paréntesis. La multiplicación en asociativa de manera que todas las formas de colocar paréntesis deberían llevar a un mismo producto.

La forma en que se colocan los paréntesis puede tener un fuerte impacto en el costo en que se evalúa el producto. Por ejemplo, asuma una cadena $(A_1, A_2, A_3)$ de tres matrices. Suponga además que las matrices son de dimensiones $10 \times 100, 100 \times 5, 5 \times 50$, respectivamente. Entonces, si agrupamos de la siguiente forma $((A_1 A_2)A_3)$ se realizan $10 \times 100 \times 5 = 5000$ multiplicaciones escalares para multiplicar $A_1 A_2$ que resulta en una matriz de $10 \times 5$, más $10 \times 5 \times 50 = 2500$ multiplicaciones para obtener la multiplicación de la matriz resultante con $A_3$, sumando un total de 7500 multiplicaciones. Si agrupamos $(A_1(A_2 A_3))$, sin embargo, se realizan 75000 multiplicaciones.

## Multiplicación de matrices (cont).

El problema de la *multiplicación de una cadena de matrices* se puede resumir como: *Dada una cadena de n matrices* $(A_1, A_2, \ldots, A_n)$*, para* $i = 1, 2, \ldots, n$ *y* $A_i$ *una matriz con dimensión* $p_{i-1} \times p_i$*, se quiere encontrar la forma de colocar los paréntesis de manera que el número de multiplicaciones escales para encontrar el producto total sea el mínimo.*

Uno puede obtener una recurrencia sobre el número de alternativas para colocar los paréntesis definida de la siguiente forma, donde $P(n)$ es el número de formas alternativas de colocar paréntesis en una cadena de $n$ matrices, $k$ identifica la $k-$ésima y $(k+1)-$ésima matrices entre las cuales se divide el problema es dos subcadenas con paréntesis:

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

Esta recurrencia es acotada por $\Omega(4^n/n^{3/2})$. Por lo tanto, una forma exhaustiva de buscar las formas de evaluar una cadena de matrices no es un eficiente algoritmo.

## Multiplicación de matrices: estructura de la solución óptima

Asuma que colocar paréntesis en forma óptima de $A_i A_{i+1} \ldots A_j$ divide el producto entre $A_k$ y $A_{k+1}$, entonces la subcadena prefijo $A_i A_{i+1} \ldots A_k$ en la cadena con paréntesis óptima $A_i A_{i+1} \ldots A_j$ debe ser una cadena con paréntesis óptima, ya que si no fuera así se podría sustituir $A_i A_{i+1} \ldots A_k$ por esta otra subcadena óptima y obtener así un costo menor para $A_i A_{i+1} \ldots A_j$. La misma observación se puede hacer para la subcadena $A_{k+1} A_{k+2} \ldots A_j$.

En consecuencia se puede construir una solución óptima para una instancia del problema de mutiplicación de una cadena de matrices al dividir el problema en dos subproblemas $A_i A_{i+1} \ldots A_k$ y $A_{k+1} A_{k+2} \ldots A_j$, para los cuales se pueden encontrar soluciones óptimas y finalmente combinar estas soluciones.

## Multiplicación de matrices: estructura de la solución óptima (cont.)

Cualquier solución al problema no trivial de mutiplicación de cadenas de matrices requiere dividir el producto, y cualquier solución óptima contiene en sí misma soluciones óptimas a subproblemas de multiplicación.

Debemos asegurarnos, entonces, que cuando se busca por el lugar correcto de división del producto, se consideren todos los posibles lugares de manera de estar seguros de haber examinado la solución óptima.

## Multiplicación de matrices: solución recursiva

Sea $m[i, j]$ el números mínimo de multiplicaciones escalares necesarias para calcular $A_{i\ldots j}$, con $A_{i\ldots j}$ el resultado de la multiplicación de $A_i A_{i+1} \ldots A_j$. Entonces $m[1, n]$ queda definida de la siguiente forma:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases}$$

Para mantener el seguimiento de $k$ en la construcción de la solución óptima, hagamos $s[i, j]$ el valor de $k$ para el cual $A_i A_{i+1} \ldots A_j$ se divide para obtener una solución óptima.

## Multiplicación de matrices: algoritmo

En vez de calcular $m[1, n]$ en forma recursiva se puede aplicar el paradigma de programación dinámica y calcular la solución óptima usando un enfoque bottom-up y tabular.

Considere que la matriz $A_i$ tiene dimensión $p_{i-1} \times p_i$, para $i = 1 \ldots n$. La entrada al algoritmo es una secuencia $(p_0, p_1 \ldots p_n)$ de largo $\mathit{length}[p] = n + 1$. El algoritmo usa una tabla $m[1 \ldots n, 1 \ldots n]$ para almacenar $m[i, j]$ y una tabla $s[1 \ldots n, 1 \ldots n]$ que almacena el índice $k$ que alcanza el valor óptimo $m[i, j]$.

Para un $k = i, i + 1, \ldots, j - 1$, la matriz $A_{i \ldots k}$ es un producto de $k - i + 1 < j - i + 1$ matrices y la matriz $A_{k+1 \ldots j}$ es un producto de $j - k < j - i + 1$ matrices. Entonces, el algoritmo debe llenar la tabla $m$ en una manera que corresponda resolver la colocación de paréntesis en una cadena de matrices de largo creciente.

## Multiplicación de matrices: algoritmo (cont.)

*CadenaMatrizOrden*($p$)

```
begin
  n ← length[p] − 1
  for i = 1 to n do
    m[i, i] ← 0
  endfor
  for l = 2 to n do  ** l el largo de la cadena
    for i = 1 to n − l + 1 do
      j ← i + l − 1
      m[i, j] ← ∞
      for k = i to j − 1 do
      q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k , p_j
      if q < m[i, j] then
        m[i, j] ← q
        s[i, j] ← k
      endif
      endfor
    endfor
  endfor
end
```

## Multiplicación de matrices: algoritmo (cont.)

Considere la ejecución del algoritmo para $n = 6$, con las matrices de las siguientes dimensiones:

| **Matriz** | **Dimension** |
|---|---|
| $A_1$ | $30 \times 35$ |
| $A_2$ | $35 \times 15$ |
| $A_3$ | $15 \times 5$ |
| $A_4$ | $5 \times 10$ |
| $A_5$ | $10 \times 20$ |
| $A_6$ | $20 \times 25$ |

Matriz $m$

| i:j | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 15750 | 7875 | 9375 | 11875 | 15125 |
| 2 | | 0 | 2625 | 4375 | 7125 | 10500 |
| 3 | | | 0 | 750 | 2500 | 5375 |
| 4 | | | | 0 | 1000 | 3500 |
| 5 | | | | | 0 | 5000 |
| 6 | | | | | | 0 |

Matriz $s$

| i:j | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 3 | 3 | 3 |
| 2 | | 2 | 3 | 3 | 3 |
| 3 | | | 3 | 3 | 3 |
| 4 | | | | 4 | 5 |
| 5 | | | | | 5 |

El orden del algoritmo da una complejidad de $O(n^3)$.

## Multiplicación de matrices: Construcción de la solución óptima

*ParentesisOptimos*$(s, i, j)$

```
begin
  if i = j then
    imprime " A"_i
  else
    printf "("
    ParentesisOptimos(s, i, s[i, j])
    ParentesisOptimos(s, s[i, j] + 1, j)
    printf ")"
end
```

## Ejercicio

Encuentre los paréntesis para solucionar la cadena de multiplicación de matrices cuya secuencia de dimensiones es $(5, 10, 3, 12, 5, 50, 6)$.

## Subestructuras óptimas

El primer paso de usar programación dinámica para resolver un problema es caracterizar la estructura de la solución óptima. Esto es, un problema tiene una subestructura óptima si es que la solución óptima del problema contiene en ella soluciones óptimas a los subproblemas. Consecuentemente, se debe asegurar que el rango de subproblemas incluye aquellos usados en la solución óptima.

## Subestructuras óptimas (cont.)

Para caracterizar el problema con una subestructura óptima se puede tomar en cuenta las siguientes indicaciones.

- Identificar que una solución a un problema consiste de tomar una elección. Tomar esta elección deja una o más subproblemas a solucionar.
- Se asume que la selección hecha te lleva a una solución óptima
- Dada una selección, se identifica la mejor forma de caracterizar los subproblemas
- Se muestra que las soluciones a los subproblemas usados en la solución óptima son en sí soluciones óptimas. Para ello, asuma que cada solución a los subproblemas no es óptima y derive una contradicción.

## Subestructuras óptimas (cont.)

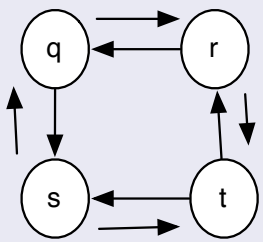
Las soluciones óptimas varían para diferentes dominios de problemas en dos formas

1. El número de subproblemas que son usados en un solución óptima al problema original
2. El número de selecciones que se hacen en determinar cuál de los subproblemas se usa en la solución óptima

En general, el tiempo de cómputo de un algoritmo de programación dinámica depende del producto de dos factores: el número de subproblemas y cuántas selecciones se miran por cada subproblema.

## Subestructuras óptimas: independencia

Uno debe tener cuidado en no caracterizar un problema con subestructura óptima cuando no lo es. Considere, por ejemplo, el caso del problema de encontrar un camino simple (sin ciclos) que contenga la mayor cantidad de nodos (unweighted longest simple path problem - WLSP).
Para este problema uno podría estar tentado en definir que una solución puede descomponer el problema de WLSP entre $u$ y $v$ en dos subproblemas, uno para encontrar el WLSP entre un nodo $u$ y $w$ y luego entre $w$ y $v$. La siguiente figura da un contra ejemplo de esta suposición.




En este caso, el camino $q \rightarrow r \rightarrow t$ es el camino más largo entre $q$ y $t$. Sin embargo el camino $q \rightarrow r$ no es el camino más largo entre $q$ y $r$, y similarmente, el camino $r \rightarrow t$ no es el camino más largo entre $r$ y $t$. El problema aquí es que los dos subproblemas no son independientes. Esto es, que la solución a un subproblema afecta la solución del otro subproblema.

## Subestructuras óptimas: sobreposición

La idea de superposición es que en la programación dinámica el espacio de subproblemas debe ser tan pequeño como sea posible, es decir, uno resuelve los mismos subproblemas una y otra vez. En general este espacio es polinomial en el tamaño de la entrada.

## Reconstrucción de una solución óptima

En programación dinámica uno almacena las elecciones que se hacen en cada subproblema en una tabla, de manera de poder reconstruir una solución óptima.

## Memoization

Un alternativa a la programación dinámica, que mantiene su rendimiento pero que usa una estrategia top-down, es la de memorizar los resultados de un algoritmo recursivo pero ineficiente. Un algoritmo recursivo en este enfoque mantiene una entrada en una tabla por cada solución a un subproblema. Cada entrada contiene inicialmente un valor que indica que la entrada debe aún ser llenada.

# Multiplicación de matrices: Memoization

*Memoized* $-$ *CadenaMatrices*$(p)$

**begin**
  $n \leftarrow$ *length*$[p] - 1$
  **for** $i \leftarrow 1$ **to** $n$ **do**
    **for** $j \leftarrow i$ **to** $n$ **do**
      $m[i, j] \leftarrow \infty$
    **endfor**
  **endfor**
  **return** *revisarCadena*$(p, 1, n)$
**end**

*revisarCadena*$(p, i, j)$

**begin**
  **if** $m[i, j] < \infty$ **then**
    **return** $m[i, j]$
  **endif**
  **if** $i = j$ **then**
    $m[i, j] \leftarrow 0$
  **else for** $k \leftarrow i$ **to** $j - 1$ **do**
    $q \leftarrow$ *revisarCadena*$(p, i, k)$ + *revisarCadena*$(p, k + 1, j) + p_{i-1} p_k p_j$
    **if** $q < m[i, j]$ **then**
      $m[i, j] \leftarrow q$
    **endif**
    **endfor**
  **endif**
**end**

El algoritmo corre en $O(n^3)$ al igual que la solución bottom-up.

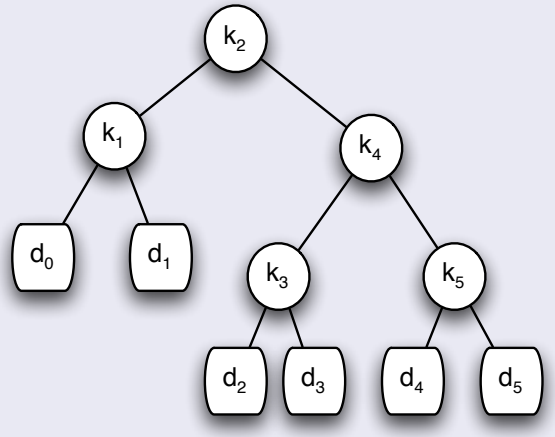
## Árboles binarios de búsqueda óptimos

Una forma de optimizar un árbol binario de búsqueda es lograr que palabras frecuentes se ubiquen cerca de la raíz y palabras menos frecuentes más alejados de la raíz.

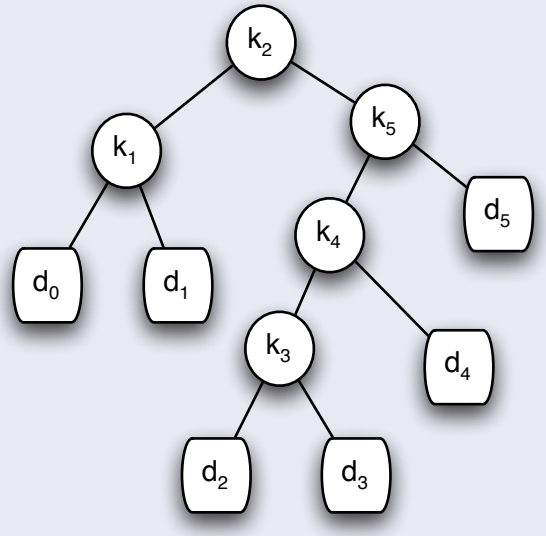
Formalmente, dada un sucuencia $K = (k_1, k_2, k_3, \ldots, k_n)$ de $n$ claves distintas ordenadas $k_1 < k_2 < k_3 \ldots k_n$, se desea construir un árbol binario de estas claves. Para clave $k_i$ se tiene una probabilidad $p_i$ de que se haga una búsqueda por esa clave $k_i$. Además, se tienen $n + 1$ claves ficticias $d_0, d_1, \ldots d_n$ que representan claves que no están en $K$. En particular $d_0$ representa claves menores a $k_1$, $d_n$ representa claves mayores que $k_n$, y para cada $i$ con $i = 2, \ldots n - 1$, $d_i$ representa los valores entre $k_i$ y $k_{i+1}$. Cada $d_i$ tiene una probabilidad $q_i$ de búsqueda.

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

# Árboles binarios de búsqueda óptimos (cont.)




Costo esparado de 2.8




Costo esperado 2.75

| i | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $p_i$ | | 0.15 | 0.1 | 0.05 | 0.1 | 0.2 |
| $q_i$ | 0.05 | 0.1 | 0.05 | 0.05 | 0.05 | 0.1 |

## Árboles binarios de búsqueda óptimos: costo esperado

Sea E[T] el costo esperado de buscar en un árbol $T$, que se determina como el número de nodos examinados, osea la profundidad + 1.

$$\begin{aligned} E[T] &= \sum_{i=1}^{n} (depth_T(k_i) + 1)p_i + \sum_{i=0}^{n} (depth_T(d_i) + 1)q_i \\ &= 1 + \sum_{i=1}^{n} depth_T(k_i)p_i + \sum_{i=0}^{n} depth_T(d_i)q_i \end{aligned}$$

El objetivo es construir un árbol con el costo mínimo.

## Árboles binarios de búsqueda óptimos: estructura óptima

Si un árbol de búsqueda binario óptimo $T$ tienes un subárbol $T'$ conteniendo claves $k_i, \ldots k_j$, entonces este subárbol $T'$ debe ser óptimo también para el subproblema con claves $k_i, \ldots k_j$ and claves ficticias $d_{i-1}, d_i, \ldots d_j$. Si hubiera un subárbol $T''$ cuyo costo esperado fuera menor que $T'$, entonces podríamos reemplazar $T'$ por $T''$ en $T$, resultando un árbol de menor costo que $T$, lo que contradice que $T$ sea óptimo.

Dada claves $k_i, \ldots k_j$, una de las claves, digamos $k_r$ ($i \leq r \leq j$), será la raíz del subárbol óptimo conteniendo estas claves. El subárbol izquierdo de $k_r$ deberá contener las claves $k_i, \ldots k_{r-1}$ (con claves ficticias $d_{i-1}, \ldots d_{r-1}$) y el subárbol derecho de $k_r$ deberá contener las claves $k_{r+1}, \ldots k_j$ (conclaves ficticias $d_r, \ldots d_j$). Si examinamos todos los posibles $k_r$ y determinamos los subárboles izquierdo y derecho, entonces podemos garantizar que el árbol binario de búsqueda es óptimo.

Notar que para subárboles sin claves, existe la clave ficticia.

## Áboles binarios de búsqueda óptimos: solución recursiva

Considere el subproblema de encontrar un subárbol binario óptimo conteniendo las claves $k_i \ldots k_j$, donde $i \geq 1, j \leq n$ y $j \geq i - 1$ cuando no hay una clave y solo se tiene una clave ficticia $d_{j-1}$.

Definamos $e[i, j]$, el costo esperado de buscar en árbol binario de búsqueda conteniendo claves $k_i, \ldots, k_j$. Se desea al final encontrar $e[1, n]$.

Cuando $j = i - 1$, entonces $e[i, i - 1] = q_{i-1}$.

Cuando $j \geq i$, se necesita seleccionar una raíz $k_r$ entre $k_i, \ldots, k_j$ y hacer el árbol binario de búsqueda óptimo con claves $k_i, \ldots, k_{r-1}$ como árbol izquierdo y claves $k_r, \ldots, k_j$ como subárbol derecho.

## Áboles binarios de búsqueda óptimos: solución recursiva (cont.)

Notar que la profundidad de cada nodo es un subárbol aumenta en 1 cuando el subárbol pasa a ser un hijo de un nodo. Entonces, el costo de búsqueda esperado de este subárbol aumenta en la suma de todas las probabilidades en el subárbol. Para un subárbol con claves $k_i, \ldots, k_j$, denotaremos la suma de las probabilidades:

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

Si $k_r$ es la raíz de un subárbol óptimo conteniendo las claves $k_i, \ldots, k_j$, entonces

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j))$$

Notar que

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j)$$

## Áboles binarios de búsqueda óptimos: solución recursiva (cont.)

Entonces el costo de búsqueda esperado menor $e[i, j]$ es :

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ min_{i \leq r \leq j}\{e[i, r-1] + e[r+1, j] + w(i,j)\} & \text{if } i \leq j \end{cases}$$

# Áboles binarios de búsqueda óptimos: algoritmo

Sea $p_1, \ldots, p_n$ y $q_0, \ldots, q_n$ las probabilidades de buscar claves $k_i$ $(1 \leq i \leq n)$ y claves ficticias $d_j$ $(0 \leq j \leq n)$, el siguiente algoritmo de orden $O(n^3)$ retorna una tabla $e[1 \ldots n+1, 0 \ldots n]$ y la tabla $root[i, j]$ para almacenar la raíz del subárbol conteniendo claves $k_i \ldots, k_j$. Además se usará una tabla $w[1 \ldots n+1, 0 \ldots n]$ para almacenar valores temporales que no se recalcularán cada vez que se necesitan.

*optimalBST*$(p, q, n)$

```
begin
  n ← length[p] − 1
  for i = 1 to n + 1 do
    e[i, i − 1] ← q_{i−1}
    w[i, i − 1] ← q_{i−1}
  endfor
  for l = 1 to n do
    for i = 1 to n − l + 1 do
      j ← i + l − 1
      e[i, j] ← ∞
      w[i, j] ← w[i, j − 1] + p_j + q_j
      for r = i to j do
      t ← e[i, r − 1] + e[r + 1, j] + w[i, j]
      if t < e[i, j] then
        e[i, j] ← t
      else
        root[i, j] ← r
      endif
      endfor
    endfor
  endfor
end
```

## Áboles binarios de búsqueda óptimos: ejemplo

Considere los siguientes valores de probabilidad para claves de búsquedas y claves ficticias, donde la clave $d_0$ representa las claves ficticias menores a $k_1$.

| node | probabilidad |
|---|---|
| $k_1$ | 0.15 |
| $k_2$ | 0.1 |
| $k_3$ | 0.05 |
| $k_4$ | 0.1 |
| $k_5$ | 0.2 |
| $d_0$ | 0.05 |
| $d_1$ | 0.1 |
| $d_2$ | 0.05 |
| $d_3$ | 0.05 |
| $d_4$ | 0.05 |
| $d_5$ | 0.1 |

## El Problema de partición (patición lineal)

- Problema: Partición de entero sin reordenamiento (redistribución)
- Input: Una secuencia $S$ de números positivos $\{s_1, \ldots, s_n\}$ y un entero k.
- Output: Partición de $S$ en $k$ o menos rangos que minimice la máxima suma sobre todos los rangos, sin reordenar los números.

## El Problema de partición: subestructura

- La k-ésimo partición comienza una vez que se han colocado los $k-1$ divisores. Esto se puede pensar tomando una posición de división que divide la secuencia en dos. El k-ésimo divisor se coloca entre el $i$-ésimo y el $(i+1)$-ésimo elemento, para algún $1 \leq i \leq n$. El costo será el costo mayor de las dos cantidades: (1) el costo de la última partición $\sum_{j=i+1}^{n} s_j$ y (2) el costo mayor de la partición formada a la izquierda de $i$.
- Asumamos que la división $k$ es a óptima, pero que las subdivisiones hasta $k-1$ no son lo son. Entonces siempre se podría subtituir las $k-1$ particiones alguna otra que nos daría un costo menor y entonces la división $k$ no podría ser la óptima, lo que es una contradicción.

## El Problema de partición: solución recursiva

Sea $M[n,k]$ el costo mínimo posible para particionar $\{s_1, \ldots s_n\}$ en $k$ rangos, donde el costo de una partición es el la suma de elementos mayor en una de sus partes. Así , la función a evaluar es la siguiente:

$$M[n,k] = \begin{cases} s_1 & \text{Si } n = 1, \text{ para todo } k > 0 \\ \sum_{i=1}^{n} s_i & \text{Si } k = 1 \\ min_{i=1}^{n} max(M[i,k-1], \sum_{j=i+1}^{n} s_j) & enotrocaso \end{cases}$$

# El Problema de partición: algoritmo

*Particion*($s[]$, $n$, $k$)

**begin**
  Sea $m$ una matriz de dimensiones $n+1 \times k+1$ de valores de costos
  Sea $d$ una matriz de dimensiones $n+1 \times k+1$ de divisores
  Sea $p$ un arreglo de dimensión $n+1$ de sumas de prefijos
  $p[0] \leftarrow 0$
  **for** $i = 1$ **to** $i \leq n$ **do**
    $p[i] \leftarrow p[i-1] + s[i]$
    $i \leftarrow i+1$
  **for** $i = 1$ **to** $i \leq n$ **do**
    $m[i][1] \leftarrow p[i]$
    $i \leftarrow i+1$
  **for** $i = 1$ **to** $i \leq k$ **do**
    $m[1][j] \leftarrow s[1]$
    $i \leftarrow i+1$
  **for** $i = 2$ **to** $i \leq n$ **do**
    **for** $j = 2$ **to** $j \leq k$ **do**
      $m[i][j] \leftarrow MAXINT$
      **for** $l = 1$ **to** $l \leq (i-1)$ **do**
        $costo = max(m[l][j-1], p[i] - p[l])$
        **if** $(m[i][j] > costo$ **then**
          $m[i][j] = costo$
          $d[i][j] = l$
      $l \leftarrow l+1$
    $j \leftarrow j+1$
  $j \leftarrow i+1$
**end**

El algoritmo corre en $O(n^2)$.

## Ejemplo

| m | k | | | | d | k | | |
|---|---|---|---|---|---|---|---|---|
| n | 1 | 2 | 3 | | n | 1 | 2 | 3 |
| 1 | 1 | 1 | 1 | | 1 | - | - | - |
| 2 | 3 | 2 | 2 | | 2 | - | 1 | 1 |
| 3 | 6 | 3 | 3 | | 3 | - | 2 | 2 |
| 4 | 10 | 6 | 4 | | 4 | - | 3 | 3 |
| 5 | 15 | 9 | 6 | | 5 | - | 3 | 4 |
| 6 | 21 | 11 | 9 | | 6 | - | 4 | 5 |
| 7 | 28 | 15 | 11 | | 7 | - | 5 | 6 |
| 8 | 36 | 21 | 15 | | 8 | - | 5 | 6 |
| 9 | 45 | 24 | 17 | | 9 | - | 6 | 7 |

## El Problema de partición: algoritmo (cont)

reconstruir($s[]$, $d[]$, $n$, $k$)

```
begin
  if k = 1 then
    imprimir_bloque(s,1,n)
  else
    reconstruir(s,d,d[n][k], k-1)
end
```

imprimir_bloque($s[]$, $start$, $end$)

```
begin
  for i = start to  i ≤ end do
    imprimir s[i]
    i ← i + 1
end
```

# Algoritmos Greedy

## Introducción

- Los algoritmos de optimización usualmente pasan por un secuencia de pasos, con un conjunto de elecciones a realizar en cada paso.
- Para algunos casos la programación dinámica en buscar una solución óptima puede ser muy costosa.
- Un algoritmo greedy siempre escoge una alternativa que parece mejor en el momento de la elección. Esto logra una elección óptima local.
- Algoritmo greedy no siempre llevan a una solución óptima. En este sesión veremos problemas para los cuales algoritmos greedy encuentran soluciones óptimas.

## Ejemplo 1: Selección de actividad

Asuma que se tiene un conjunto $S = \{a_1, \ldots, a_n\}$ de $n$ actividades propuestas que desean usar un recurso. Cada actividad $a_i$ tiene un tiempo de inicio $s_i$ y un tiempo final $f_i$, donde $0 \leq s_i \leq f_i \leq \infty$. Si es seleccionada, una actividad $a_i$ toma lugar durante el intervalo de tiempo semi abierto $[s_i, f_i)$. Actividades $a_i$ y $a_j$ son compatibles si $s_i \geq f_j$ o $s_j \geq f_i$. El problema de la selección de actividad (activity-selection problem - ASP) desea maximizar el tamaño del subconjunto de actividades mutuamente compatibles.

## Ejemplo 1: Selección de actividad (cont.)

Considere la siguiente tabla, donde asumimos que las actividades están ordenadas por tiempo final $f_i$.

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $s_i$ | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
| $f_i$ | 4 | 5 | 6 | 7 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

Para este ejemplo, el subconjunto $\{a_3, a_9, a_{11}\}$ contiene actividades compatibles, pero no es una solución óptima. Una solución óptima es $\{a_1, a_4, a_8, a_{11}\}$ o $\{a_2, a_4, a_9, a_{11}\}$

## Ejemplo 1: Pasos a seguir

- Se parte pensando en una solución de programación dinámica, en la cual se consideran alternativas para determinar cuáles subproblemas utilizar en una solución óptima.
- Se observa que sólo se necesita elegir una alternativa.
- Luego se diseña un algoritmo greedy recursivamente para resolver el problema.
- Se transforma el algoritmo recursivo en uno iterativo.

## Selección de actividades: Subestructura óptima

Denotemos $S_{ij}$ el conjunto de actividades que comienzan después de que la actividad $a_i$ finaliza y antes de que la actividad $a_j$ comienza. Para representar el problema completo, agregamos las actividades ficticias $a_0$ y $a_{n+1}$, para las cuales $f_0 = 0$ y $s_{n+1} = \infty$. Luego $S = S_{0,n+1}$ con los rangos de $i$ y $j$ dados por $0 \leq i, j \leq n + 1$.

Si se asume que los tiempos finales de las actividades están ordenadas en orden creciente, entonces, el problema se transforma en encontrar el conjunto máximo de actividades mutuamente compatibles de $S_{i,j}$, con $0 \leq i < j \leq n + 1$.

Asuma que tenemos un problema$S_{i,j}$ no vacío, entonces asuma que una solución a $S_{i,j}$ incluye una actividad $a_k$, tal que $f_i \leq s_k < f_k \leq s_j$. Esto genera dos subproblemas $S_{i,k}$ y $S_{k,j}$. Entonces la solución a $S_{i,j}$ es la union de las soluciones $S_{i,k}$ y $S_{k,j}$.

## Selección de actividades: Subestructura óptima (cont.)

La subestructura óptima es como sigue. Suponga ahora que una solución $A_{i,j}$ de $S_{i,j}$ incluye una actividad $a_k$. Entonces las soluciones $A_{i,k}$ a $S_{i,k}$ y $A_{k,j}$ a $S_{k,j}$ que se usan en la solución óptima de $S_{i,j}$ deben ser óptimas.

**Demo.** Se asume que $A_{i,j}$ es la solución a $A_{i,j}$. Si tenemos una solución $A'_{i,k}$ a $S_{i,k}$ con más actividades que $A_{i,k}$, entonces podríamos reemplazar $A_{i,k}$ por $A'_{i,k}$ en $A_{i,j}$, generando otra solución a $S_{i,j}$ con más actividades. Ya que asumimos que $A_{i,j}$ era una solución óptima, entonces tenemos una contradicción. Lo mismo se aplica para $S_{k,j}$.

## Selección de actividades: solución recursiva

Asuma que queremos encontrar el conjunto más grande de actividades mutuamente compatibles en $S_{ij}$ y asuma que ese conjunto es $A_{ij}$ que incluye una actividad $a_k$. Al incluir $a_k$, entonces tenemos dos subproblemas: encontrar $S_{ik}$ y $S_{kj}$. Sea $A_{ik} = A_{ij} \cap S_{ik}$ y $A_{kj} = A_{ij} \cap S_{kj}$, entonces $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ y $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$.

Si denotamos $c[i, j]$ el tamaño de la solución para el conjunto $S_{ij}$, este valor a optimizar se define como:

$$c[i,j] = \begin{cases} 0 & \text{If } S_{ij} = \emptyset \\ max_{a_k \in S_{i,j}}\{c[i,k] + c[k,j] + 1\} & \text{If } S_{i,j} \neq \emptyset \end{cases}$$

## Selección de actividades: la elección greedy

La idea detrás de la elección es poder consider una sola alternativa en vez de resolver todos los subproblemas del enfoque de programación dinámica.

Para este problema, la idea sería escoger una actividad que deje el recurso disponible para la mayor cantidad de otras actividades. Entonces, una forma de hacer eso es escoger una actividad $S$ que termine lo más temprano posible. En este caso considerar la actividad $a_1$.

Una vez elegida la actividad que termina primero, ahora se debe escoge la actividad que comienza después de $a_1$. Note que toda actividad compatible con $a_1$ debe comenzar después de que termina $a_1$.

## Selección de actividades: la elección greedy (cont.)

**Teorema:** Considere un subproblema $S_k$ no vacío, y sea $a_m$ una actividad en $S_k$ que termina primero. Entonces $a_m$ está dentro de un subconjunto de tamaño máximo de actividades mutuamente compatibles de $S_k$.

**Demo:** Sea $A_k$ el subconjunto de mayor tamaño de actividades mutuamente compatibles en $S_k$, y sea $a_j$ la actividad que termina primero en $A_k$. Si $a_j = a_m$ entonces se cumple el teorema. Si $a_j \neq a_m$, sea el conjunto $A'_k = \{A_k - \{a_j\}\} \cup \{a_m\}$ $A_k$ al cual se le ha sustituído $a_j$ por $a_m$. Las actividades en $A'_k$ están separadas, ya que las actividades en $A_k$ son separadas, $a_j$ es la primera actividad en $A_k$ en terminar y $f_m \leq f_j$. Ya que $|A'_k| = |A_k|$, se concluye que $A'_k$ es un subconjunto de mayor tamaño de actividades mutuamente excluyente de $S_k$ y que incluye a $a_m$.

Esto indica que aunque podemos resolver este problema por programación dinámica, no es necesario.

## Selección de actividades: algoritmo recursivo

En este algoritmo, $s$ y $f$ representan arreglos con el tiempo de comienzo y fin de un actividad, $k$ define el subproblem $S_k$ a resolver y $n$ el tamaño del problema original. Este algoritmo asumo que $f[0] = 0$ y que se llama inicialmente a la rutina como $ASPRecursivo(s, f, 0, n)$

$ASPRecursivo(s, f, k, n)$

```
begin
  m ← k + 1
  while m ≤ n and s[m] < f[k] do
   m ← m + 1
  endwhile
  if m ≤ n then
   return {a_m} ∪ ASPRecursivo(s, f, m, n)
  else
   return ∅
  endif
end
```

Este algoritmo corre el $\Theta(n)$

## Selección de actividades: algoritmo iterativo

En este algoritmo, $s$ y $f$ representan arreglos con el tiempo de comienzo y fin de un actividad, $k$ define el subproblem $S_k$ a resolver y $n$ el tamaño del problema original.

*ASPIterartivo*$(s, f)$

```
begin
  n ← s.length
  A ← {a_1}
  k ← 1
  for m = 2 to n do
   if s[m] ≥ f[k] then
    A ← A ∪ {a_m}
   k ← m
  endif
  return A
end
```

Este algoritmo corre el $\Theta(n)$

## Estrategia Greedy: Elementos

- Determinar la subestructura óptima del problema
- Desarrollar un algoritmo recursivo (No se usa la recurrencia para el algoritmo, se basa en una elección)
- Mostrar que si se hace la elección solo un subproblema queda.
- Mostrar que es siempre seguro hacer una elección greedy. Esto es, probar que hay siempre una solución óptima al problema original que hace una elección greedy, tal que la elección es siempre segura. Habiendo hecho una elección, lo que queda es un subproblema con la propiedad que si combinamos una solución óptima al subproblema que queda con la elección greedy hecha, se llega a un solución del problema original.
- Desarrollar un algoritmo recursivo que implemente la estrategia
- Convertir el algoritmo recursivo a uno iterativo

## Propiedad de elección greedy

Esta propiedad indica que uno puede construir un solución óptima globalmente haciendo elecciones óptimas locales. A diferencia de una programación dinámica, la elección se basa en el estado actual del proceso, sin considerar los resultados a subproblemas. Este proceso de resolución de problema usa una estrategia top-down, donde la elección no depende de elecciones futuros o de la solución a subproblemas.

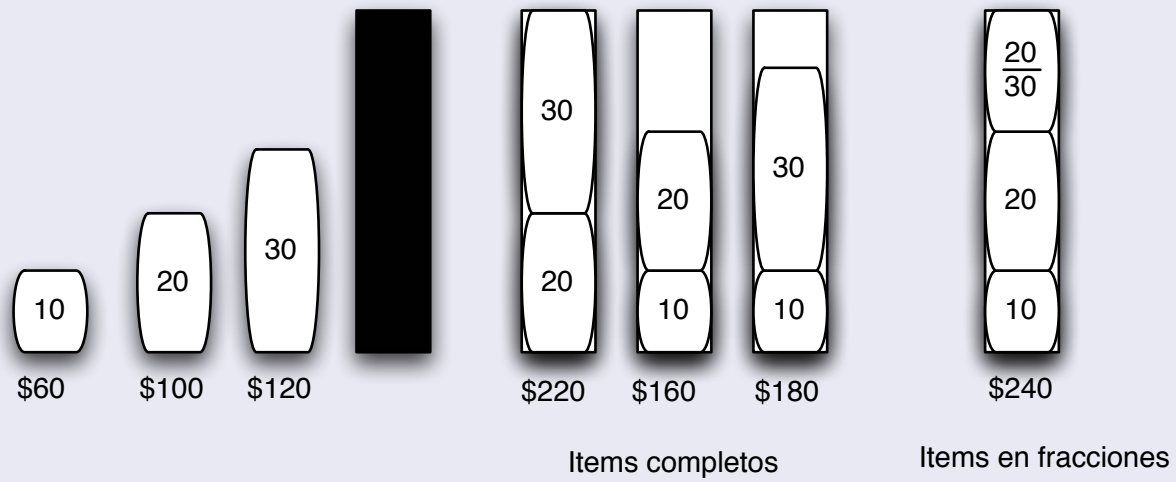
Uno debe demostrar que cada elección lleva a una solución óptima globalmente. Tipicamente, al igual que se hizo en el teorema asociado al ASP, la prueba examina la solución óptima globalmente a algún subproblema. Muestra cómo modificar la solución sustituyendo la elección greedy de alguna otra elección, resultando en uno problema similar, pero más pequeño.

## Subestructura óptima

El problema exhibe una subestructura óptima si una solución óptima para el problema contiene soluciones óptimas a subproblemas. Esto es clave tanto para programación dinámica como para algoritmos greedy.

En el caso de algoritmos greedy, lo que se necesita hacer es mostrar que una solución óptima a un subproblema, combinado con la elección hecha en ese paso, lleva a una solución óptima del problema original. Este esquema usa intuitivamente inducción en los subproblemas para probar que hacer la elección greedy en cada paso produce una solución óptima.

## Greedy versus Programación dinámica



## Código Huffman

- La codificación de Huffman es una codificación de prefijo de largo variable para la compresión de cadena de caracteres.
- A diferencia de una codificación de largo fijo (como el ASCII), largo variable permite optimizar y comprimir en más de un 20 % la cadena de caracteres
- Códigos son asignados a caracteres tal que el largo del código depende de la frecuencia relativa del caracter.

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| Frecuencia | 45 | 13 | 12 | 16 | 9 | 5 |
| Código de largo fijo | 000 | 001 | 010 | 011 | 100 | 101 |
| Código de largo variable | 0 | 101 | 100 | 111 | 1101 | 1100 |

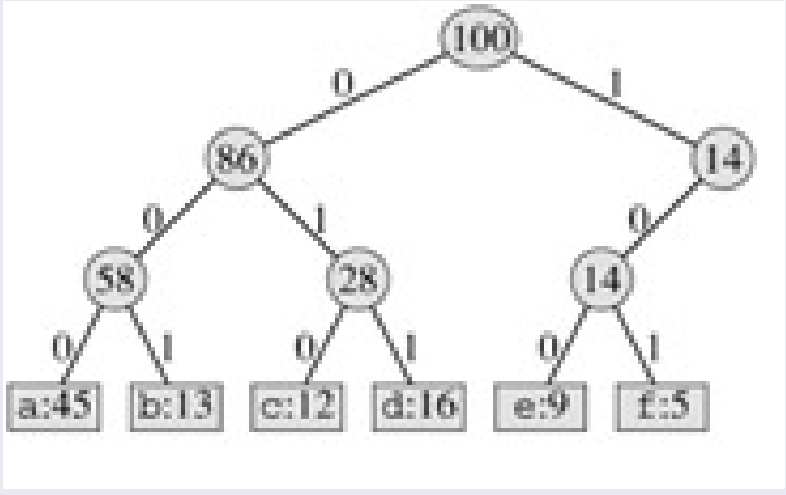
## Código libre de prefijo

- Códigos libres de prefijo (también llamados simplemente códigos de prefijo) son aquellos que no son prefijo de ningún otro código. Una compresión óptima se alcanza cuando se usan códigos libres de prefijos.
- Para código de caracteres binarios, la codificación es simple. Cada código que representa un caracter en el documento se concatena. Por ejemplo, siguiendo el ejemplo anterior la codificación para $abc = 0.101.100 = 010110$, donde“.”denota la concatenación.

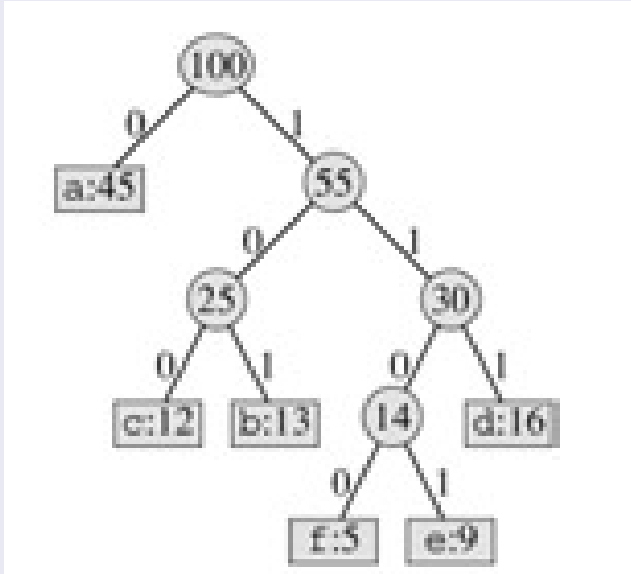
## Código libre de prefijo (cont.)

- Código de libre de prefijo son deseables porque simplifican la codificación ya que el comienzo del documento codificado no es ambigüo.
- La decodificación requiere una representación conveniente para que el primer código sea fácilmente encontrado. Esto se logra con una representación de los códigos en un árbol binario lleno, en el cual todo nodo no hoja tiene 2 hijos.

## Árboles de presentación




Árbol no óptimo




Árbol óptimo

## Árboles de presentación (cont.)

- Árboles binarios llenos cumplen la condición que dado un alfabeto con $C$ caracteres, el árbol óptimo tiene exactamente $|C|$ hojas, una por cada letra del alfabeto, y exactamente $|C| - 1$ nodos internos.
- Sea $f(c)$ la frecuencia de $c \in C$ en un archivo y sea $d_T(c)$ la profundidad de la hoja de $c$ en el árbol $T$ (lo que es también el largo del código asociado a $c$). Entonces, el número de bits requeridos para codificar el archivo es:

  $B(T) = \sum_{c \in C} f(c) d_T(c)$

## Árbol de Código Huffman

Un árbol de Código Huffman se construye en base a la frecuencia de los caracteres en un documento

- El documento es recorrido y la frecuencia de los caracteres es determinada
- Un árbol binario es construido en el cual nodos hojas almacenan la frecuencia de los caracteres observada en el documento. En la práctica se usa la frecuencia típica de caracteres en vez de determinar la frecuencia exacta de caracteres en un documento

## Construcción del Árbol de Codificación

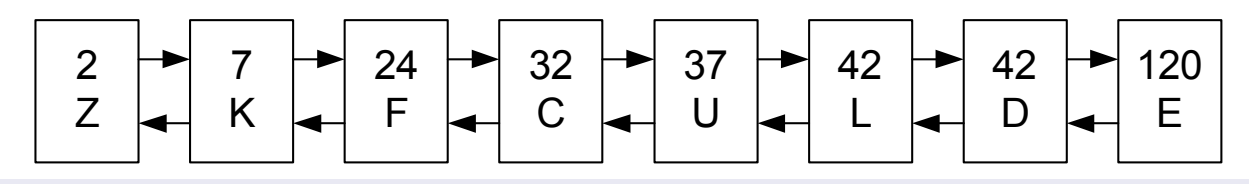
Considere la siguiente frecuencia de caracteres en una cadena que requiere ser codificada[a].

| caracter | c | d | e | f | k | l | u | z |
|---|---|---|---|---|---|---|---|---|
| frecuencia | 32 | 42 | 120 | 24 | 7 | 42 | 37 | 2 |

[a] Ejemplo extraído desde *Huffman Coding* por Lawrence Brown, 1999

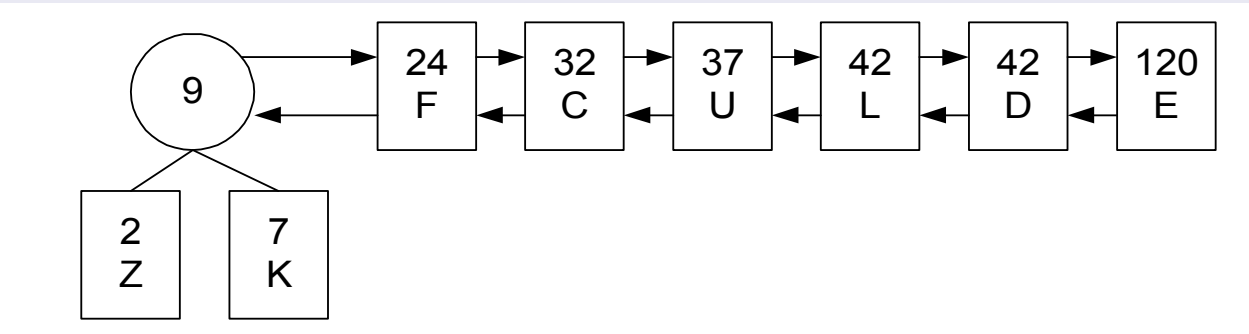
## Construcción: paso 1

El primer paso construye una cola de prioridad por frecuencia de caracteres



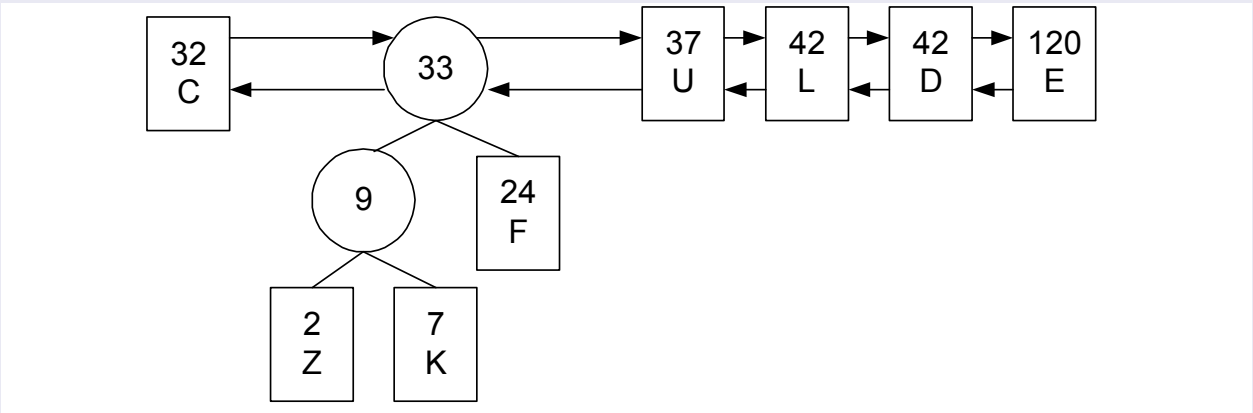
## Construcción: paso 2

En el segundo paso, los dos elementos con menor frecuencia son removidos de la cola de prioridad. Se crea un árbol binario con el elemento de menor frecuencia en la izquierda y el de mayor frecuencia en la derecha. El árbol se insertar al final de la cola

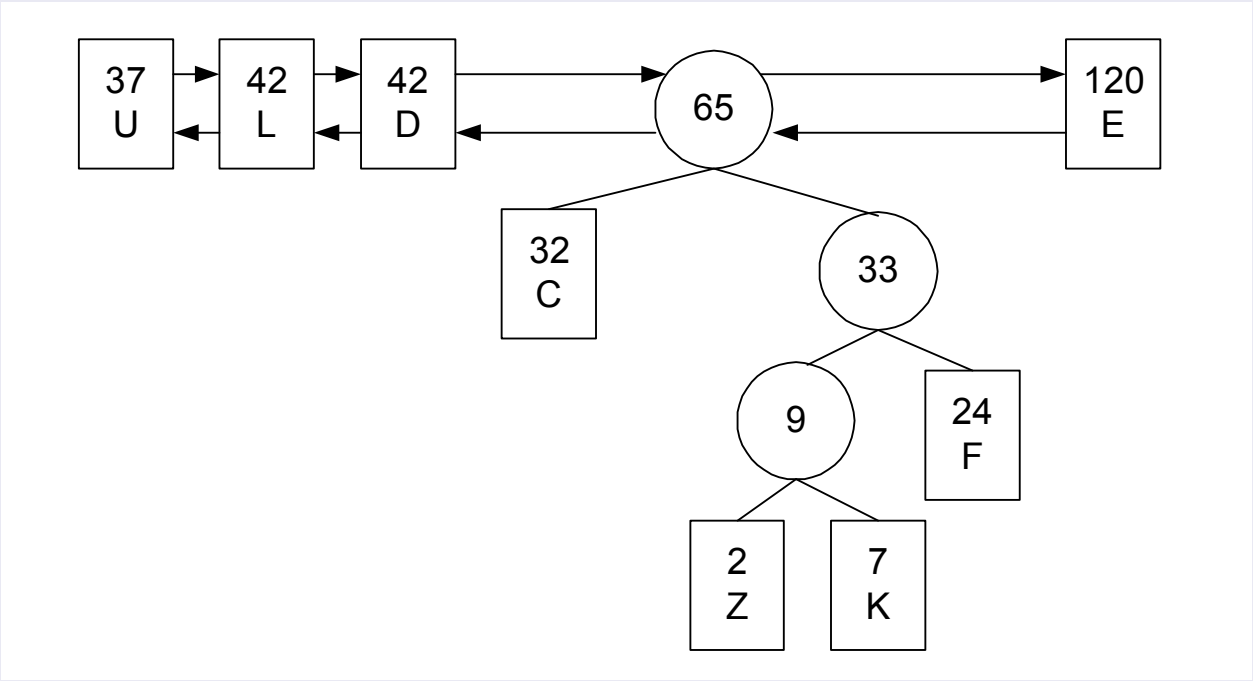


## Construcción: paso 3

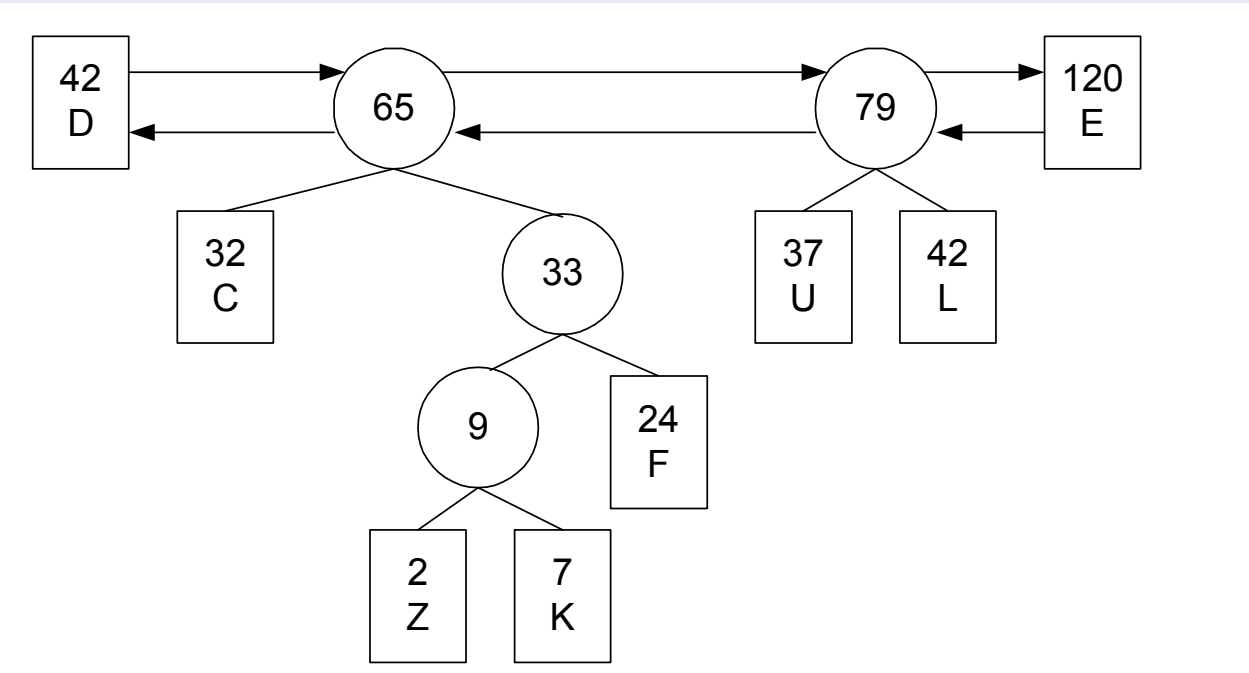
El proceso continua de la misma forma



## Construcción: paso 4

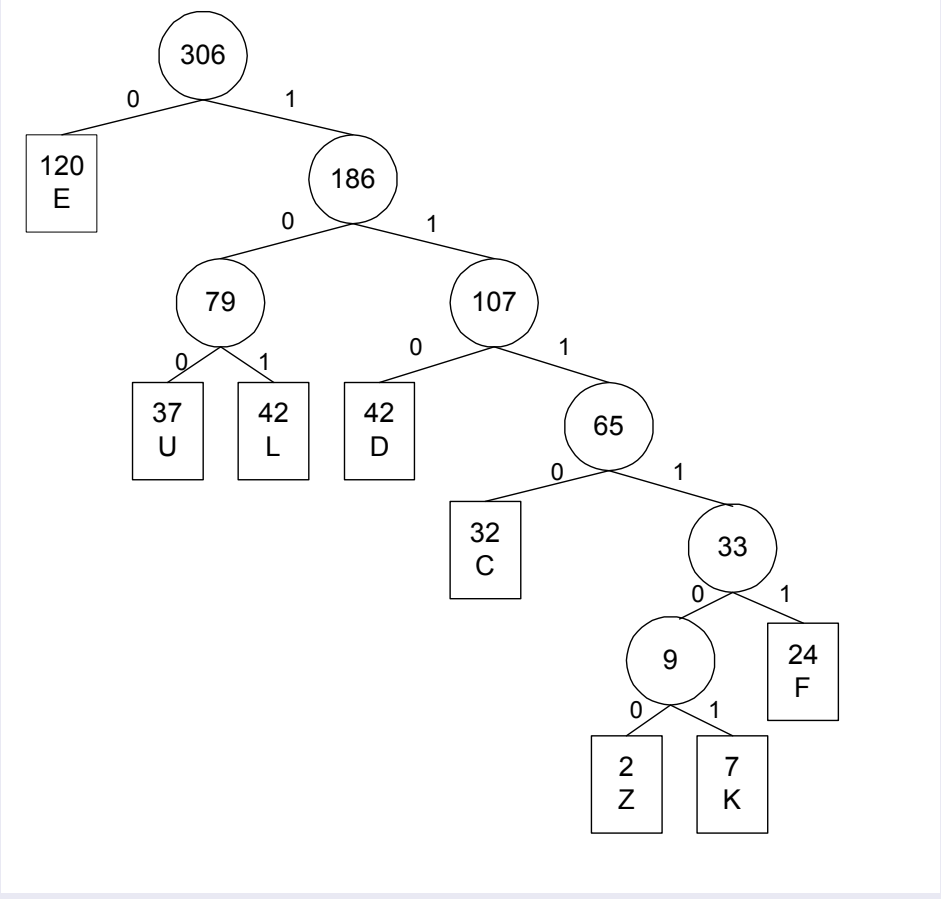


## Construcción: paso 5



## Construcción: paso final

Hasta llegar al final, lo que ocurre en este caso en el paso $n = 8$



## Algoritmo

Algorithm Huffmann( $X$ ):
Input: String $X$ of length $n$.
Ouput: Coding tree for $X$.

Compute frequency $f(c)$ of each character $c$ in $X$.
Initialize a priority queue $Q$.
**for each** character $c$ in $X$ **do**
Create a single-node tree $T$ storing $c$.
Insert $T$ into $Q$ with key $f(c)$.
**while** $Q$.size() > 1 **do**
$f_1 \leftarrow Q$.minKey()
$T_1 \leftarrow Q$.removeMinElement()
$f_2 \leftarrow Q$.minKey()
$T_2 \leftarrow Q$. removeMinElement()
Create a new tree $T$ with left subtree $T_1$ and right subtree $T_2$.
Insert $T$ into $Q$ with key $f_1 + f_2$.
return $Q$.removeMinElement() // return tree

En este algoritmo $Q$ es implementada como un $min - heap$. El tiempo tiempo entonces de este algoritmo es de $nlogn$ ya que se realizan $n - 1$ operaciones de extracción de valores de la fila (costo $O(logn)$).

## Algoritmo: propiedad de la elección greedy

**Lema:** Sea $C$ un alfabeto en el cual el caracter $c \in C$ tiene frecuencia $c.freq$. Sea $x$ e $y$ dos caracteres en $C$ teniendo la frecuencia menor. Entonces, existe un código de prefijo óptimo para $C$ en el cual los códigos para $x$ e $y$ tiene el mismo largo y solo difieren en el último bit.

**Demo:** La idea de la demostración es tomar un árbol $T$ representando un código de prefijo óptimo arbitrario y modificarlo para generar un árbol que represente otro código de prefijo óptimo tal que los caracteres $x$ e $y$ aparezcan como hojas hermanas de mayor profundidad en el árbol. Si construímos eso, entonces, los códigos para $x$ e $y$ tendrán el mismo largo y diferirán en un solo bit.

## Algoritmo: propiedad de la elección greedy (cont.)

**Demo:** Sean $a, b$ dos caracteres que son hojas hermanas de profundidad máxima en $T$. Asumamos que $a.freq \leq b.freq$ y que $x.freq \leq y.freq$, entonces $x.freq \leq a.freq$ y $y.freq \leq b.freq$. Además asumamos que $x \neq b$, para hacer la demostración no trivial. Intercambiando las posiciones de $a$ por $x$, se genera un árbol $T'$, con una diferencia en el costo de búsqueda dada para árbol optimizados de

$$\begin{aligned} B(T) - B(T') &= \\ &= \sum_{c \in C} c.freq \times d_T(c) - \sum_{c \in C} c.freq \times d_{T'}(c) \\ &= x.freq \times d_T(x) + a.freq \times d_T(a) - x.freq \times d_{T'}(x) - a.freq \times d_{T'}(a) \\ &= x.freq \times d_T(x) + a.freq \times d_T(a) - x.freq \times d_T(a) - a.freq \times d_T(x) \\ &= (a.freq - x.freq)(d_T(a) - d_T(x)) \end{aligned}$$

con $d_T(c)$ la profundidad de $c$ en árbol $T$. Tanto $a.freq - x.freq$ y $d_T(a) - d_T(x)$ son no negativos, teniendo entonces que $B(T') < B(T)$. Así podemos entonces hacer el intercambio de $b$ por $y$ logrando un árbol $T''$ tal que $B(T'') < B(T)$.
Con este lema entonces podemos decir que una solución óptima puede comenzar por escoger los dos frecuencias más bajas. Ahora se necesita ver que exista una subestructura óptima.

## Algoritmo: propiedad de la elección greedy (cont.)

**Lema:** Sea $T$ un árbol de codificación de prefijo óptimo sobre un alfabeto $C$ en el cual el caracter $c \in C$ tiene frecuencia $c.freq$. Sea $x$ e $y$ dos caracteres en $C$ teniendo la frecuencia menor que aparecen como hermanas hojas en $T$ y sea $z$ el padre de $x$ e $y$. Entonces, considerando $z$ como un caracter con frecuencia $z.freq = x.freq + y.freq$, el árbol $T' = T - \{x, y\}$ representa un código de prefijo óptimo para $C' = C - \{x, y\} \cup z$.

## Algoritmo: propiedad de la elección greedy (cont.)

**Demo:** Por cada caracter $c \in C - \{x, y\}$, tenemos que $d_T(c) = d_{T'}(c)$. Debido a que $d_T(x) = d_T(z) + 1$, entonces

$$\begin{aligned} x.freq \times d_T(x) + y.freq \times d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \times d_{T'}(z) + (x.freq + y.freq) \end{aligned}$$

Entonces $B(T) = B(T') + x.freq + y.freq$ o, equivalente
$B(T') = B(T) - x.freq - y.freq$

## Algoritmo: propiedad de la elección greedy (cont.)

Ahora por contradicción. Asuma que $T$ no representa una codificación de prefijo óptimo para $C$. Entonces existe un $T''$ tal que $B(T'') < B(T)$. Sin pérdida de generalidad, $T''$ tiene $x$ e $y$ como hermanos. Sea $T'''$ el árbol $T''$ con el padre común de $x$ e $y$ reemplazos por una hoja $z$ con frecuencia $z.freq = x.freq + y.freq$. Entonces

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') \end{aligned}$$

Lo que es una contradicción a que $T'$ represente un código de prefijo óptimo para $C'$. Entonces, $T$ debe representar el código óptimo para el alfabeto $C$.

## Árbol de cobertura mínimo

**Definición:** Un árbol de cobertura de un grafo conectado es su subgrafo conectado acíclico (osea su árbol) que contiene todos los vértices del grafo. Se dice que es un árbol de cobertura mínimo de un grafo conectado y con pesos, cuando la suma de los pesos de las aristas en la menor.

La subestructura óptima de este problema se da ya que si tenemos un árbol de cobertura mínima para $n$ vértices de un grafo conectado y con pesos, entonces, este árbol puede obtenerse al considerar el árbol de cobertura mínimo para $n - 1$ vértices agregando el vértice que conecta a $n$ con los demás sin formar ciclos y que minimiza los pesos totales.

## Árbol de cobertura mínimo: Algoritmo

*Prim*($G$)
//Entrada: un grafo conectado y con pesos $G = (V, G)$
//Salida: $E_T$ el conjunto de arcos componiendo un árbol de cobertura mínimo de $G$

**begin**
  $V_T \leftarrow \{v_0\}$
  $E_T \leftarrow \emptyset$
  **for** $i \leftarrow 1$ **to** $|V| - 1$ **do**
   encontrar el arcos de peso mínimo $e^* = (v^*, u^*)$ tal que $v$ es en $V_T$ y $u$ es en $V - V_T$
   $V_T \leftarrow V_T \cup \{u^*\}$
   $E_T \leftarrow E_T \cup \{e^*\}$
  **return** $E_T$ **end**

Si el grafo es representado como una matriz con pesos, y la fila es implementada como un arreglo sin orden, el tiempo es de $\Theta(|V|^2)$. Si embargo, si la fila es implementada como un min-heap, el insertar y eliminar en la fila tiene un costo de $O(logn)$. Si además se representa el grafo como una lista de adyacencia y se realiza $|V| - 1$ eliminaciones del elemento menor y hace $|E|$ verificaciones, entonces la complejidad está dada por $(|V| - 1) + |E|O(log|V|)$, lo que es $O(|E|log|V|)$.

## Árbol de cobertura mínimo: elección greedy

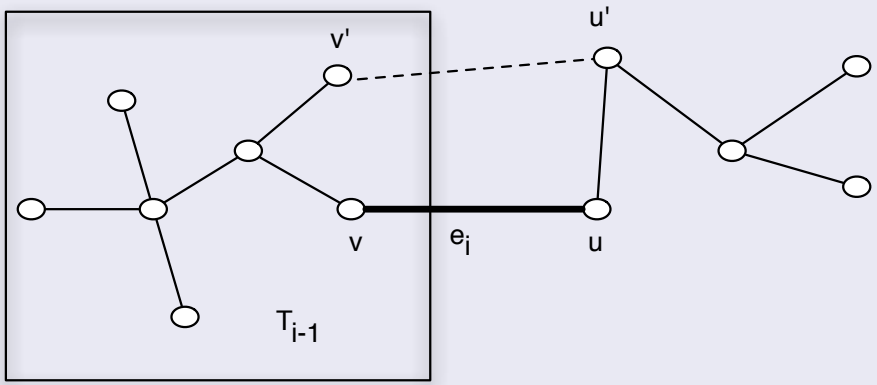
Queremos demostrar que el algoritmo anterior con la elección de ir escogiendo el arco con menor peso nos lleva a un árbol de cobertura mínima. Esto se puede demostrar por inducción de los subárboles $T_i$, con $i = 0, \ldots, n-1$ generados por el algoritmo.

Sea $T_0$ el árbol que consiste de un vértice único, la base de inducción, consecuentemente debe ser parte de cualquier árbol de cobertura mínimo y queda demostrado.

Para el paso de inducción, asumamos que $T_{i-1}$ es parte de un árbol de cobertura mínimo $T$. Necesitamos probar que $T_i$ generado a partir de $T_{i-1}$ por el algoritmo dado es también parte de un árbol de cobertura mínimo.

## Árbol de cobertura mínimo: elección greedy

Esto se puede probar por contradicción asumiendo que ningún árbol de cobertura mínimo puede contener $T_i$. Sea $e_i = (v, u)$ el arco de peso mínimo con $v \in T_{i-1}$ y $u \notin T_{i-1}$ usado por el algoritmo para pasar de $T_{i-1}$ a $T_i$. Por el supuesto, $e_i$ no puede pertenecer a ningún árbol de cobertura mínimo, incluyendo $T$. Por lo que si agregamos $e_i$ a $T$, un ciclo debe formarse. Además de $e_i$, este ciclo debe contener otro arco $(v', u')$, conectando un vértice $v' \in T_{i-1}$ a un vértice $u' \notin T_{i-1}$ (estos vértices pueden ser tal que $v = v'$ o $u = u'$ pero no ambos).




Si ahora borramos el otro arco $(v', u')$ del ciclo, se obtiene otro árbol de cobertura mínimo del grafo completo cuyo peso total es menor que el peso de $T$, ya que el peso de $e_i$ es menor o igual al de $(v', u')$. Hemos llegado a una contradicción.

## Matroids: Algoritmos Greedy

Teoría útil para determinar si un método greedy nos lleva a una solución óptima.
Aunque no cubre todos los casos, si cubre muchos casos de interés práctico.
A matroid es un par ordenado $M = (S, I)$, que satisface las siguientes condiciones:

- S es un conjunto finito no vacío.
- I es una familia no vacía de subconjuntos de $S$, llamados subconjuntos independientes de $S$, tales que si $B \in I$ y $A \subseteq B$, entonces $A \in I$. Si se cumple esta condición, se dice que $I$ es heriditario.
- Si $A \in I$ y $B \in I$ y $|A| < |B|$, entonces $\exists x \in B - A$ tal que $A \cup \{x\} \in I$. Se dice entonces que M cumple la Propiedad de Intercambio

## Ejercicio: Matroid.

Asuma que $S = \{a, b, c, d\}$. Construya el matroid más pequeño $(S, I)$ tal que $\{a, b\}$ y $\{c, d\}$ están en $I$.
$I = \{\emptyset, \{a\}, \{b\}, \{c\}, \{d\}, \{a, b\}, \{c, d\}$, por propiedad hereditaria
$\{a, c\}, \{b, c\}, \{a, d\}, \{b, d\}$, por propiedad de intercambio $\}$

## Matroids: ejemplo

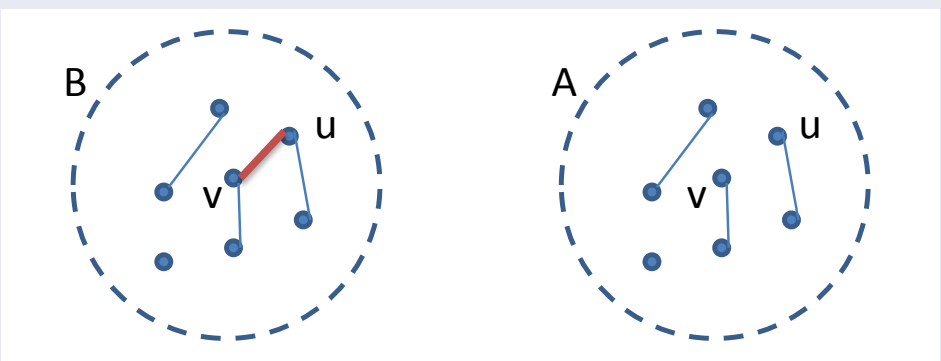
Un ejemplo de matroid es un graphic matroid $M_G = (S_G, I_G)$ definido en términos de un grado no dirigido dado $G = (V, E)$ y las siguientes condiciones:

- $S_G$ es el conjunto de aristas $E$.
- Si $A$ es un subconjunto de $E$, $A \in I_G$ si y solo si $A$ es acíclico. Esto es, un conjunto de arcos $A$ es independiente si y solo si el subgrafo $G_A = (V, A)$ forma un bosque.

## Matroids: ejemplo (cont.)

**Teorema:** Si $G = (V, E)$ es un grafo no dirigido, entonces $M_G = (S_G, I_G)$ es un matroide.
**Demo:** Es trivial que $S_G = E$ es finito. También $I_G$ es hereditario, ya que un subconjunto de un bosque es un bosque. Entonces, se necesita probar que $M_G$ satisface la propiedad de exchange.

Supongamos que $G_A = (V, A)$ y $G_B = (V, B)$ son bosques de $G$ y $|A| < |B|$. Sabemos que un bosque con $k$ aristas tiene $|V| - k$ árboles, por tanto $A$ tiene $(|V| - |A|)$ árboles y $B$ tiene $(|V| - |B|)$ árboles. Entonces, $B$ tiene al menos un árbol cuyos vértices están en 2 árboles distintos de $A$. Como este árbol esta conectado en $B$ tiene que tener una arista $(u, v)$ tal que $u$ y $v$ estén en árboles distintos de $A$. Entonces puedo adicionar esta arista $(u, v)$ a $A$ sin crear ningún ciclo. Entonces $M_G$ cumple la Propiedad de Intercambio



## Matroids: propiedades

Si tenemos un Matroid $M(S, I)$, decimos que $x \notin A$ es una extensión de $A \in I$ si puedo adicionar $x$ a $A$ y que éste siga siendo independiente.
Si $A$ es un conjunto independiente de un Matroid decimos que $A$ es maximal si no tiene extensiones.
**Teorema:** Todos los subconjuntos maximales en un matroid tienen el mismo tamaño
**Demo:** Supongamos que $A$ es un conjunto maximal de $M$ y que existe otro conjunto maximal de $M$ mayor que $A$, digamos $B$. Entonces la propiedad de intercambio implica que $\exists x \in B - A$ tal que $A \cup \{x\} \in I$ , es decir, podemos extender $A$ con $x$ y que siga siendo independiente. Esto contradice el hecho de que $A$ es maximal.

Notar que para un grafo no dirigido conexo, entonces a maximal independent subset es un árbol de cobertura mínima de $G$.

## Matroids: propiedades (cont.)

Se dice que $M = (S, I)$ es un Matroid con costo si existe una función de costo $w$ que le asigna un costo estrictamente positivo $w(x)$ a cada elemento de $x \in S$.
Esta función se extiende a los subconjuntos de $S$ por la expresión $w(A) = \Sigma_{x \in A} w(x)$ , para cualquier $A \subseteq S$.

Un subconjunto óptimo de un Matroid es un subconjunto independiente de costo máximo. Como los costos $w(x)$ siempre son positivos, un subconjunto óptimo es siempre un subconjunto independiente maximal.

Muchos problemas para los cuales el enfoque greedy entrega una solución óptima puede ser formulado como encontrar un subconjunto independiente de peso máximo en un matroid.

## Matroids: árbol de cobertura mínimo

Tenemos un grafo conexo no dirigido $G = (V, E)$ y una función de peso $w$ tal que $w(e)$ es el peso (positivo) de la arista $e$. Queremos encontrar un subconjunto de aristas que conecten todos los vértices y que tengan el peso mínimo total. Consideremos el Matroid con peso $M_G$ con función de peso $w'$, donde $w'(e) = w_0 - w(e)$ y $w_0$ es mayor que el máximo valor de los pesos de las aristas.

En este Matroid todos los pesos son positivos y un subconjunto independiente óptimo es un árbol de cobertura de peso total mínimo en el grafo original. En particular, $w'(A) = (|V| - 1)w_0 - w(A)$, por lo tanto si se maximiza $w'(A)$ se minimiza $w(A)$.

## Matroids: algoritmo greedy

*Greedy*$(M, w)$
$M$ es un matroid, con componentes $S[M]$ y $I[M]$,
y $w$ es función de peso.

---

**begin**
  $A \leftarrow \emptyset$
  ordene $S[M]$ en orden decreciente por peso $w$
  **for each** $x \in S[M]$ **do**
   **if** $A \cup \{x\} \in I[M]$ **then** $A \leftarrow A \cup \{x\}$
  **return** $A$
**end**

El tiempo de ejecución está dado por el ordenamiento $O(nlogn)$. Si chequear la condición $A \cup \{x\} \in I[M]$ toma $O(f(n))$, entonces el algoritmo toma $O(nlogn + nf(n))$.

## Matroids: subestructura óptima (cont.)

**Teorema:** Si $M = (S, I)$ es Matroide con funciones de peso $w$, entonces el método GREEDY(M,w) retorna una solución óptima.

## Problema de planificación de tareas como un matroid

La idea es planificar en forma óptima las tareas de una unidad de tiempo en un procesador único, donde cada tarea tiene un deadline, con una penalidad si la tarea no se termina en el deadline.

Las tareas de una unidad de tiempo, tal como las que corren un computador, requiere exactamente una unidad de tiempo completa. Dado un conjunto finito $S$ de tareas de una unidad de tiempo, una planifiación es una permutación de $S$ especificando el orden en que se realizan las tareas. La primera tarea comienza en tiempo 0 y finaliza en tiempo 1, la segunda en tiempo 1 y termina en tiempo 2.

## Problema de planificación de tareas como un matroid (cont.)

El problema tiene las siguientes entradas:

- un conjunto $S = \{a_1, a_2, \ldots, a_n\}$ de $n$ tareas de una unidad de tiempo.
- un conjunto de $n$ enteros que son los deadlines $d_1, \ldots, d_n$, tal que cada $d_i$ satisface $1 \leq d_1 \leq n$ y una tarea $a_i$ se supone que debe terminar en $d_i$, y
- un conjunto $n$ no negativo de pesos o penalidades $w_1, \ldots, w_n$, tal que incurrimos en una penalidad $w_i$ si la tarea $a_i$ no finaliza en $d_i$, de lo contrario, no se incurre en una penalidad.

Se desea entonces encontrar la planificación que minimiza las penalidad total incurrida.

## Problema de planificación de tareas como un matroid (cont.)

Sea una tarea tardía, aquella que termina después de su deadline. Si termina antes es una tarea temprana. Se puede entonces transformar una planificación en una forma tempranas-primero, lo que implica que las tareas que son tempranas se realizan primero.

Entonces, el problema se resuelve si se busca un conjunto $A$ de tareas que se asignan tempranamente en la planificación. Diremos que el conjunto $A$ es independiente si existe una planificación para estas tareas tal que ninguna es tardía.

## Problema de planificación de tareas como un matroid (cont.)

Queremos entonces ver si un conjunto dado $A$ es independiente.

**Lema:** Para cualquier conjunto de tareas $A$, los siguientes planteamientos son equivalentes.

1. El conjunto $A$ es independiente.
2. Para $t = 0, 1, 2, \ldots, n$ se tiene que $N_t(A) \leq t$, con $N_t(A)$ el número de tareas cuyo deadline es $t$ o antes. Notar que $N_0(A) = 0$, para cualquier conjunto $A$.
3. Si las tareas se ordenan en orden creciente de su deadline, entonces no hay tareas tardías.

## Problema de planificación de tareas como un matroid (cont.)

**Teorema:** Si $S$ es un set de tareas de una unidad de tiempo con deadlines, y $I$ es el set de todos los conjuntos de tareas independientes, entonces el sistemas $(S, I)$ es un matroid.

**Demo:** Cada subconjunto de un conjunto independiente de tareas es ciertamente independiente. Para demostrar la propiedad de intercambio, asuma que $A$ y $B$ son conjuntos independientes y que $|B| > |A|$. Sea $k$ el mayor $t$ tal que $N_t(B) \leq N_t(A)$. Ya que $N_n(B) = |B|$ y $N_n(A) = |A|$, pero $|B| > |A|$, debe haber un $k < n$ tal que que $N_j(B) > N_j(A)$ para todo $j$ en el rango $k + 1 \leq j \leq n$. Entonces, $B$ contiene más tareas con deadline $k + 1$ que $A$. Sea $a_i$ una tarea en $B - A$ con deadline $k + 1$ y sea $A' = A \cup \{a_i\}$.
Para $0 \leq t \leq k$, se tiene que $N_t(A') = N_t(A) \leq t$, ya que $A$ es independiente. Para $k < t \leq n$, se tiene que $N_t(A') \leq N_t(B) \leq t$, ya que $B$ es independiente. Entonces $A'$ es independiente.

Se puede entonces aplicar el algoritmo greedy con un costo de $O(n^2)$.

## Problema de planificación de tareas como un matroid: ejemplo

| | Tareas | | | | | | |
|---|---|---|---|---|---|---|---|
| $a_i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| $d_i$ | 4 | 2 | 4 | 3 | 1 | 4 | 6 |
| $w_i$ | 70 | 60 | 50 | 40 | 30 | 20 | 10 |

El algoritmo greedy ordena las tareas en orden decreciente de la penalidad y parte de tiempo 0. Luego selecciona tareas $a_1$, $a_2$, $a_3$, $a_4$, rechaza luego $a_5$ ya que $N_4(\{a_1, a_2, a_3, a_4, a_5\}) = 5$ y a $a_6$ ya que $N_4(\{a_1, a_2, a_3, a_4, a_6\}) = 5$, y finalmente acepta $a_7$.
La planificación óptima es entonces $a_2, a_4, a_1, a, a_7, a_5, a_6$, con penalidad $w_5 + w_6 = 50$.

## Problema de cambios de monedas

Asuma que se tiene $n$ tipos de monedas con valores $v[1] > \cdots > v[n] > 0$. Se le pide dar cambio de $M$ con el menor número de monedas.

*Coins*( , $CV$, $n$)
//Entrada: un arreglo $V$ de $n$ elementos ordenados definiendo los valores de las monedas
//Salida: un arreglo $M$ de $b$ elementos que indica la ocurrencia de cada moneda en el cambio.

---

**begin**
  Inicialice $M$ con valores 0.
  **for** $i \leftarrow 1$ **to** $n$ **do**
    **while** $C > V[i]$ **do**
      $C \leftarrow C - v[i]$; $m[i]$++;
    **endowhile**
  **endfor end**

## Problema de cambios de monedas (cont.)

*Coins*( , *CV* , *n*)
//Entrada: un arreglo *V* de *n* elementos ordenados definiendo los valores de las monedas
//Salida: un arreglo *M* de *b* elementos que indica la ocurrencia de cada moneda en el cambio.

---

**begin**
  Inicialice *M* con valores 0.
  **for** $i \leftarrow 1$ **to** *n* **do**
    **while** $C > V[i]$ **do**
      $C \leftarrow C - v[i]; m[i]$++;
    **endowhile**
  **endfor end**

¿Cuándo el algoritmo es correcto?
¿Entrega el mínimo número de monedas?

## Problema de cambios de monedas (cont.)

- Sea $v[1] = 5; v[2] = 2; v[3] = 1;$
- Si $C < 2$, entonces el algoritmo entrega una moneda única, la que es óptima.
- Si $C < 5$, entonces el algoritmo entrega a lo más dos moneda como se muestra en los siguientes casos que son óptimos.

  - $C = 4 = 2 * 2$ (2 coins)

  - $C = 3 = 2 + 1$ ( 2 coins)

  - $C = 2 = 2$ (1 coins)
- Si $C >= 5$, entonces el algoritmo usa la mayor cantidad de monedas de 5 y entonces un cambio óptimo para los valores que quedan.
- Ahora, considere $v[1] = 4; v[2] = 2; v[3] = 1$ y $C = 6$. ¿Entrega una solución óptima o no?

## Problema de cambios de monedas (cont.)

Cualquier solución óptima para $C = v[1] * m[1] + v[2] * m[2] + v[3] * m[3]$ con $v[1] = 4; v[2] = 3; v[3] = 1$ debe satisfacer:

- $m[3] <= 2$ (Si $m[3] > 2$, entonces reemplace los 1's por un 3
- No se puede tener que $m[3] > 0$ y $m[2] > 0$, de lo contrario se usarían 4 para reduce el número de monedas.
- $m[2] < 4$, de lo contrario use 4 para reducir el número de monedas.
- En general, Greedy falla en obtener la solución óptima, dependiendo de los tipos de monedas. Sin embargo, se pueden encontrar soluciones óptimas pero que no son únicas. $9 = 3 + 3 + 3 = 4 + 4 + 1$.

## Problema de asignación de tareas

Asuma un lista de trabajos, cada una teniendo tiempo de comienzo y fin. La tarea es asignar trabajadores a estos trabajos tal que el número de trabajores es mínimo.
Una elección greedy:

- Ordene los trabajos por el tiempo de comienzo.
- Para tarea siguiente, asigne el siguiente trabajor disponible.

**Prueba**: Asuma que este algoritmo usa $k$ trabajadores en total. El trabajador $k$ comienza a trabajar cuando todos los otros trabajadores $1, 2, \ldots, k-1$ han sido ocupados por otro trabaajo. Luego, $k$ trabajos se sobreponen al momento, probando lo que requiere.