Programación dinámica

ANÁLISIS Y DISEÑO DE ALGORITMOS
MTI. PEDRO O. PÉREZ M.
TECNOLÓGICO DE MONTERREY, CAMPUS QUERÉTARO

Contenido

- ¿En qué consiste la programación dinámica?
- Algoritmos que aplican este paradigma.
  - Programación de intervalos.
  - Secuencia de suma máxima.
  - Subsecuencia creciente más larga.
  - Mochila 0/1.
  - Cambio de monedas.
  - Subsecuencia común más larga.
  - Algoritmo de Warshall.

¿En qué consiste la Programación Dinámica?

- Esta técnica se baja en el principio de optimalidad de Bellman.
  - Cualquier subsecuencia de decisiones de una secuencia óptima de decisiones que resuelve un problema también debe ser óptima respecto al subproblema que resuelve.

¿En qué consiste la Programación Dinámica?

- Toma la estrategia básica de Divide y conquista, separando el problema en subproblemas más pequeños.
- Para cada subproblema se exploran todas las posibles soluciones, eligiendo la óptima (a diferencia de Algoritmos ávidos que elige la mejor que tiene en ese momento).
- Programación dinámica es usado, por lo mismo, en problemas de optimización (minimizar, maximizar) y conteo.

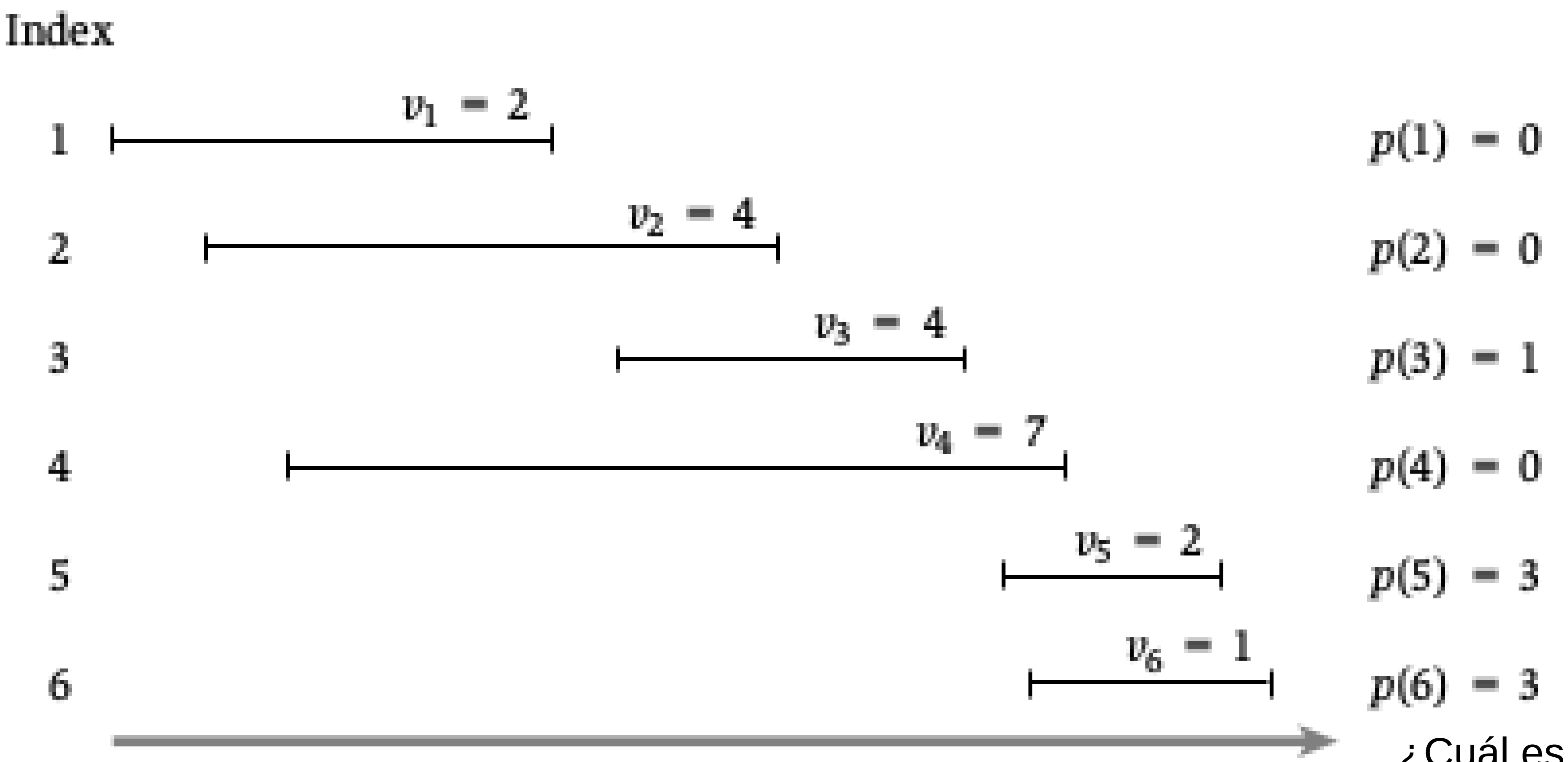
Programación de intervalos

- Recuerdas este problema: Considera una situación en la cual tenemos un sólo recurso y un conjunto de peticiones para usar el recurso por un cierto intervalo de tiempo. Cada petición *i* tiene un límite de tiempo d_i, y requiere de un intervalo de tiempo ininterrumpido para su terminación, t_i.
- Solo que ahora los intervalos le vamos a dar a cada intervalo un peso específico, v_i, y lo que queremos es aceptar el conjunto de peticiones compatibles con el máximo peso posible.

Programación de intervalos

- Primero, ¿cómo evaluamos que seleccionar un intervalo determinado es mejor que otro?
- Definimos $p(j)$ como la actividad i más cercana que termina antes de que j empiece. Decimos que $p(j) = 0$, si no existe ninguno intervalo con esas características.

Programación de intervalos



¿Cuál es la solución óptima que podemos generar para este problema?

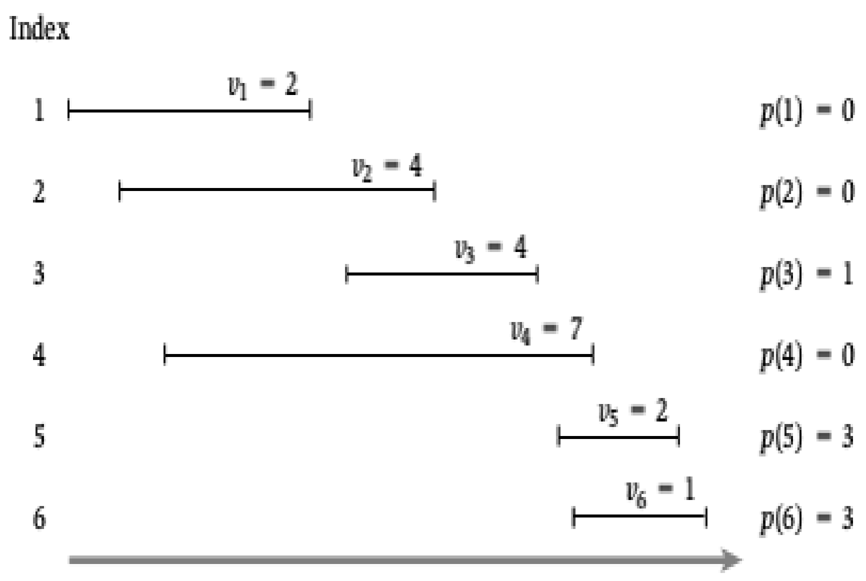
Programación de intervalos

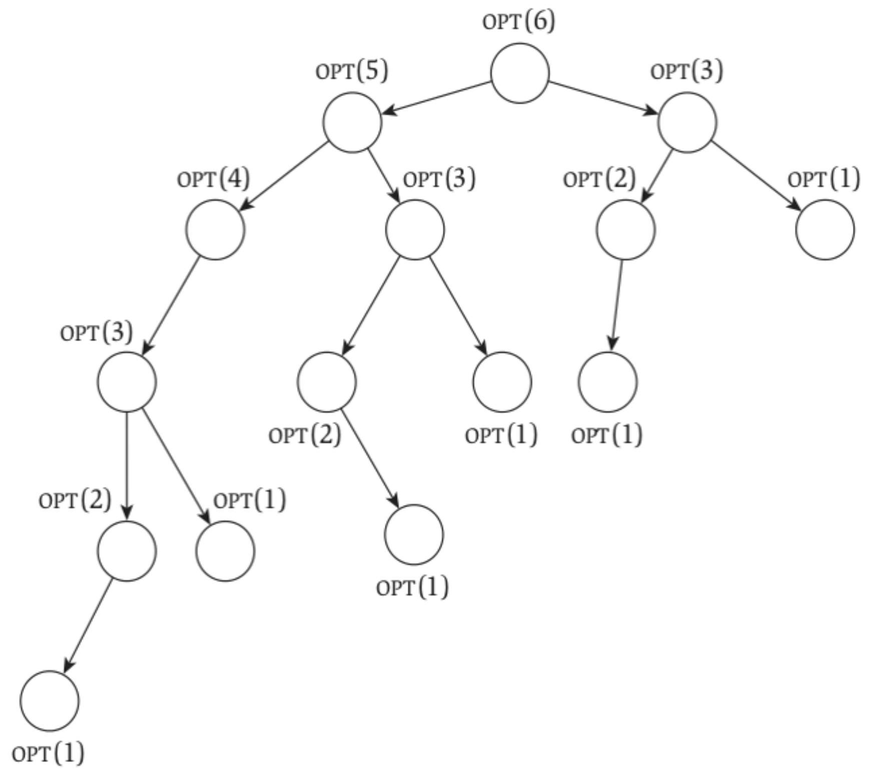
- Sea $P = \{p_1, p_2, \ldots, p_n\}$, el conjunto de peticiones. Cada petición i tienen un tiempo límite, d_i, un tiempo de ejecución, t_i, un peso determinado, vi, y

Programación de intervalos

```
procedure SCHEDULING_REC1(j) :
if j == 0 then
    return 0
else
    return MAX(v_j + SCHEDULING_REC1(opt(j)),
                   SCHEDULING_REC1(j - 1))
```

Programación de intervalos





Programación de intervalos

- La solución es precalcular todas las posibles llamadas. Esta técnica es usada como *memoization*.

Programación de intervalos

```
procedure SCHEDULING_REC2(M, j) :
if j == 0 then
    return 0
else if M[j] is not empty then
    return M[j]
else
begin
    M[j] = MAX(v_j + SCHEDULING_REC2(opt(j)),
                SCHEDULING_REC2(j - 1))
    return M[j]
end
```

Programación de intervalos

```
procedure SCHEDULING_FINAL(N) :
Array M[0..N]
M[0] = 0
for j ← 1 until N do
    M[j] = max(v_j + M[opt(j)], M[j − 1])
```

Secuencia de suma máxima

- Recuerdas este problema: Dado un arreglo de n números enteros positivos y negativos, encontrar los i elementos del arreglo cuya suma sea la máxima posible.

Secuencia de suma máxima

```
procedure MAX_SUM(A, N):
sum ← 0
ans ← 0
for i ← 1 until N do
    sum ← sum + A[i]
    ans ← MAX(ans, sum)
    if (sum < 0) then
        sum ← 0
return ans
```

Subsecuencia creciente más larga

- Considera una cadena A = {a, b, c, d}. Una subsecuencia (también llamada subcadena) se obtiene eliminando 0 o más símbolos (no necesariamente consecutivos) de A.
- Por ejemplo:
  - {a, b, c, d} , {a}, {b}, {c}, {d}, {a, d}, {a, c}, {b, d} son subsecuencias de A.
  - Pero {d, b}, {d, a}, {d, a, c} no son subsecuencias de A.

Subsecuencia creciente más larga

- Problema: dado un arreglo de n símbolo comparables, dar la subsecuencia creciente más larga.
- Por ejemplo, $A = \{-7, 10, 9, 2, 3, 8, 8, 1\}$, la subsecuencia creciente más larga es $\{-7, 2, 3, 8\}$.

Subsecuencia creciente más larga

```
procedure LIS(A):
n ← A.length
Array Q[1..n]

Q[1] ← 1
for i ← 2 until n do
    max ← 0
    for j ← 1 until (i - 1) do
        if A[j] < A[i] then
            if Q[j] > max then
                max ← Q[j]
    Q[i] ← max + 1
max ← 0
for i ← 1 until n do
    if Q[i] > max then
        max ← Q[i]
return max
```

Mochila 0/1

- Problema: sean n objetos no fraccionables de pesos w_i, y beneficios v_i, y sea C el peso máximo que puede llevar la mochila. El problema consiste en llenar la mochila con objetos, tal que se maximice el beneficio.

Por ejemplo, $W = \{10, 4, 6, 12\}$, $V = \{100, 70, 50, 10\}$, y $C = 12$. ¿Cuál es la beneficio máximo que podemos obtener?

Mochila 0/1

```
procedure KNAPSACK(W, V, C):
n ← W.length
Matrix M[0..n][0..C]

Initialize M with zeros
for i ← 1 until n do
    for j ← 1 until C do
        if j < W[i] then
            M[i][j] ← M[i - 1][j]
        else
            M[i][j] ← MAX(V[i] + M[i - 1][j - W[i]],
                              M[i - 1][j])
return M[n,C]
```

Cambio de monedas

- Recuerdas esta problema: dado un sistema monetario *S* con *N* monedas de diferentes denominaciones y una cantidad de cambio *C*, indique el menor número de monedas de *S* equivalente a *C*.
- Por ejemplo, si *S* = {1, 3, 4, 5} y el cambio a dar *C* = 7, ¿cuál debería ser el resultado?

Cambio de moneda

$$C[j] = \begin{cases} \infty & \text{if } j < 0, \\ 0 & \text{if } j = 0, \\ 1 + \min\limits_{1 \leq i \leq k} \{C[j - d_i]\} & \text{if } j \geq 1 \end{cases}$$

Cambio de monedas

```
procedure CHANGE(S, c)
Array T[0..c1]

T[0] ← 0
for i ← 1 until S.length do
    for j ← S[i] until c do
        T[j] ← MIN(1 + T[j - S[i]], T[j])
return T[c]
```

Subsecuencia común más larga

- Problema: dado dos cadenas, A y B, determinar la subsecuencia común más larga de ambas cadenas.
- Por ejemplo, si A = {`c, d, c, c, f, g, e`} y B = {`e, c, c, e, g, f, e`}, ¿cuál la subsecuencia común más larga?

Subsecuencia común más larga

- Ahora veamos como se plantea el problema: considere dos secuencias $A = a_1, a_2, \ldots, a_m$ y B = $b_1, b_2, \ldots, b_n$. Para resolver el problema nos fijaremos en los últimos dos símbolos: a_m y b_n. Como podemos ver hay dos posibilidades:
  - Caso 1: $a_m = b_n$. En este caso, la subsecuencia común más larga debe contener a_m. Simplemente basta encontrar la subsecuencia común más larga de $a1, a2, \ldots, a_{m-1}$ y $b_1, b_2, \ldots, b_{n-1}$.
  - Caso 2: $a_m \neq b_n$. En este caso, puede hacerse corresponder $a_1, a_2, \ldots, a_m$ con $b_1, b_2, \ldots, b_{n-1}$ y también $a1, a2, \ldots, a_{m-1}$ con $b_1, b_2, \ldots, b_{n,}$ y nos quedamos con el mayor de los dos resultados.

Subsecuencia común más larga

```
procedure LCS(A, B):
m ← A.length
n ← B.length
Matrix M[0..m][0..n]

Initialize M with zeros
for i ← 1 until n do
    for j ← 1 until m do
        if A[i] == B[i] then
            M[i][j] ← 1 + M[i - 1][j - 1]
        else
            M[i][j] ← MAX(M[i - 1][j], M[i][j - 1])
return M[m][n]
```

Algoritmo de Warshall

- Problema: Determinar la conectividad total de un grafo.

Algoritmo de Warshall

```
procedure WARSHALL(M):
n ← M.length
for k ← 1 until n do
    for i ← 1 until n do
        for j ← 1 until n do
            if M[i][k] == 1 and M[k][j] then
                M[i][j] ← 1
```