

# Semillero de Programación

## Programación Dinámica

Ana Echavarría    Juan Francisco Cardona

Universidad EAFIT

5 de abril de 2013

# Contenido

- 1 Problemas semana anterior
  - Problema A - The Farnsworth Parabox
  - Problema B - Flying to Fredericton
  - Problema C - Brick Wall Patterns
- 2 Motivación
- 3 Programación dinámica
- 4 Problema de la mochila
- 5 Longest Common Subsequence
- 6 Longest Increasing Subsequence
- 7 Tarea

# Contenido

- 1 Problemas semana anterior
  - Problema A - The Farnsworth Parabox
  - Problema B - Flying to Fredericton
  - Problema C - Brick Wall Patterns

# Problema A - The Farnsworth Parabox

Verificar si en un grafo no dirigido tiene un ciclo de peso negativo alcanzable desde el nodo 0.

# Implementación I

```
1  const int MAXN = 105;
2  const int INF = 1 << 30;
3  typedef pair <int, int> edge;
4  vector <edge> g[MAXN];
5  int d[MAXN];
6
7  bool bellman_ford(int s, int n){
8      for (int u = 0; u <= n; ++u) d[u] = INF;
9      d[s] = 0;
10
11     for (int i = 1; i <= n - 1; ++i){
12         for (int u = 0; u < n; ++u){
13             for (int k = 0; k < g[u].size(); ++k){
14                 int v = g[u][k].first;
15                 int w = g[u][k].second;
16                 d[v] = min(d[v], d[u] + w);
17             }
18         }
```

# Implementación II

```
19     }
20
21     for (int u = 0; u < n; ++u){
22         for (int k = 0; k < g[u].size(); ++k){
23             int v = g[u][k].first;
24             int w = g[u][k].second;
25             if (d[v] > d[u] + w){
26                 // Hay ciclo de peso negativo
27                 return true;
28             }
29         }
30     }
31     // No hubo ciclo de peso negativo
32     return false;
33 }
34
35 int main(){
36     int n, m;
37     while (cin >> n >> m){
```

# Implementación III

```
38     if (n == 0 and m == 0) break;
39
40     for (int i = 0; i <= n; ++i) g[i].clear();
41
42     for (int i = 0; i < m; ++i){
43         int u, v, t;
44         cin >> u >> v >> t;
45         u--; v--;
46         g[u].push_back(edge(v, t));
47         g[v].push_back(edge(u, -t));
48     }
49
50     if (bellman_ford(0, n)) puts("Y");
51     else puts("N");
52 }
53 return 0;
54 }
```

# Problema B - Flying to Fredericton

Hallar la mínima distancia entre un nodo  $s$  y un nodo  $t$  haciendo máximo  $i$  paradas.

Si se hacen  $i$  paradas es porque se han utilizado  $i + 1$  aristas.



# Implementación I

```
1  const int MAXN = 105;
2  const int INF = 1 << 30;
3  typedef pair <int, int> edge;
4  map <string, int> city2int;
5  vector <edge> g[MAXN];
6  int L[MAXN][MAXN];
7
8  bool bellman_ford(int s, int n){
9      for (int u = 0; u <= n; ++u) L[0][u] = INF;
10     L[0][s] = 0;
11
12     for (int i = 1; i <= n - 1; ++i){
13         for (int u = 0; u < n; ++u) L[i][u] = L[i-1][u];
14         for (int u = 0; u < n; ++u){
15             for (int k = 0; k < g[u].size(); ++k){
16                 int v = g[u][k].first;
17                 int w = g[u][k].second;
18                 L[i][v] = min(L[i][v], L[i-1][u] + w);
            }
```

# Implementación II

```
19     }
20 }
21 }
22 int main(){
23     int cases; cin >> cases;
24     for (int run = 1; run <= cases; ++run){
25         int n; cin >> n;
26
27         city2int.clear();
28         for (int i = 0; i < n; ++i){
29             g[i].clear();
30             string name; cin >> name;
31             city2int[name] = i;
32         }
33
34         int m; cin >> m;
35         for (int i = 0; i < m; ++i){
36             string s, t; int c;
37             cin >> s >> t >> c;
```

# Implementación III

```
38         int u = city2int[s];
39         int v = city2int[t];
40         g[u].push_back(edge(v, c));
41     }
42     printf("Scenario #%d\n", run);
43     bellman_ford(0, n);
44     int q; cin >> q;
45     while (q--){
46         int stops; cin >> stops;
47         int edges = min(stops + 1, n-1);
48         if (L[edges][n-1] < INF) printf("Total cost of
49             flight(s) is $%d\n", L[edges][n-1]);
50         else puts("No satisfactory flights");
51     }
52     if (run != cases) puts("");
53     return 0;
54 }
```

# Problema C - Brick Wall Patterns

Sea  $f(n)$  el número de formas se puede armar una pared de  $2 \times n$  usando rectángulos de tamaño  $2 \times 1$

$$f(1) = 1$$

# Problema C - Brick Wall Patterns

Sea  $f(n)$  el número de formas se puede armar una pared de  $2 \times n$  usando rectángulos de tamaño  $2 \times 1$

$$f(1) = 1$$

$$f(2) = 2$$

# Problema C - Brick Wall Patterns

Sea  $f(n)$  el número de formas se puede armar una pared de  $2 \times n$  usando rectángulos de tamaño  $2 \times 1$

$$f(1) = 1$$

$$f(2) = 2$$

$$f(3) = 3$$

# Problema C - Brick Wall Patterns

Sea  $f(n)$  el número de formas se puede armar una pared de  $2 \times n$  usando rectángulos de tamaño  $2 \times 1$

$$f(1) = 1$$

$$f(2) = 2$$

$$f(3) = 3$$

$$f(4) = 5$$

# Problema C - Brick Wall Patterns

Sea  $f(n)$  el número de formas se puede armar una pared de  $2 \times n$  usando rectángulos de tamaño  $2 \times 1$

$$f(1) = 1$$

$$f(2) = 2$$

$$f(3) = 3$$

$$f(4) = 5$$

.

.

.

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



# Implementación I

---

```
1  int f[51];
2
3  int main(){
4      f[1] = 1;
5      f[2] = 2;
6      for (int i = 3; i < 51; i++){
7          f[i] = f[i-1] + f[i-2];
8      }
9      int n;
10     while(cin >> n){
11         if (n == 0) break;
12         cout << f[n] << endl;
13     }
14     return 0;
15 }
```

---

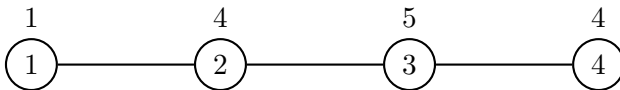
# Contenido

## 2 Motivación

# Problema

## Entrada

- Un grafo lineal  $G = (V, E)$  (grafo con dos nodos de grado 1 y el resto de nodo de grado 2 y un solo camino entre cualquier par de nodos)
- Un valor de peso para cada nodo  $v \in V$



## Objetivo

- Hallar el máximo peso total que se puede lograr de un subconjunto de nodos en el que no hay nodos adyacentes

# Ideas

## Ideas

- Existen dos posibilidades para cada nodo: está en el subconjunto óptimo o no está
- Si un nodo está en el óptimo ninguno de sus vecinos está

## Preguntas

- ¿Cuál es la mejor solución para un grafo de 1 nodo?
- ¿Cuál es la mejor solución para un grafo de 2 nodos?
- ¿Cuál es la mejor solución para un grafo de 3 nodos?

# Ideas

## Ideas

- Existen dos posibilidades para cada nodo: está en el subconjunto óptimo o no está
- Si un nodo está en el óptimo ninguno de sus vecinos está

## Preguntas

- ¿Cuál es la mejor solución para un grafo de 1 nodo?
- ¿Cuál es la mejor solución para un grafo de 2 nodos?
- ¿Cuál es la mejor solución para un grafo de 3 nodos?
- ¿La solución óptima para un grafo de 3 nodos se puede obtener de una solución óptima para grafos de 1 y 2 nodos?

# Ideas

## Ideas

- Existen dos posibilidades para cada nodo: está en el subconjunto óptimo o no está
- Si un nodo está en el óptimo ninguno de sus vecinos está

## Preguntas

- ¿Cuál es la mejor solución para un grafo de 1 nodo?
- ¿Cuál es la mejor solución para un grafo de 2 nodos?
- ¿Cuál es la mejor solución para un grafo de 3 nodos?
- ¿La solución óptima para un grafo de 3 nodos se puede obtener de una solución óptima para grafos de 1 y 2 nodos?
- ¿Cuál es la solución para un grafo de  $n$  nodos?

# Solución

## Solución

Sea  $f(n)$  el máximo peso total que se puede lograr con los nodos  $[1 \dots n]$

$f(0) = 0$  porque el conjunto de los nodos está vacío

$$f(1) = w(1)$$

$$f(2) = \max\{w(2), w(1)\} = \max\{f(0) + w(2), f(1)\}$$

$$f(3) = \max\{f(1) + w(3), f(2)\}$$

.

.

.

$$f(n) = \max\{f(n-2) + w(n), f(n-1)\}$$

# ¿Cómo computar la solución?

La solución usando la fórmula recursiva es

$$f(0) = 0$$

$$f(1) = w(1)$$

$$f(i) = \max\{f(i-2) + w(i), f(i-1)\} \text{ para } 2 \leq i \leq n$$

¿Qué pasa si pregunta por  $f(4)$ ?

¿Cuáles son las llamadas que se hacen recursivamente?

¿Hay llamadas que se hacen repetidas veces?

¿Hay alguna forma de evitar llamar más de una vez la misma función?



# Memoización

La memoización consiste en guardar los valores de la función que ya se hayan computado anteriormente y así no volverlos a computar en caso de volverlos a necesitar.

---

```
1  int memo[MAXN]; //Arreglo de memoria inicializado en -1
2  int w[MAXN];    //Arreglo con los pesos de cada nodo
3
4  int f(int n){
5      if (n == 0) return 0;
6      if (n == 1) return w[1];
7
8      if (memo[n] == -1){
9          memo[n] = max( f(n-2) + w[n], f(n-1) );
10     }
11     return memo[n];
12 }
```

---

# Implementación bottom-up

De acuerdo a la función que se construyó, se puede ver que para computar  $f(n)$  es necesario conocer los resultados de  $f(0)$  hasta  $f(n-1)$ .

Se pueden entonces computar cada uno de los  $f(n)$  empezando desde  $f(0)$ .

---

```
1  int f[MAXN]; //Arreglo con el valor de la funcin
2  int w[MAXN]; //Arreglo con los pesos de cada nodo
3
4  int main(){
5      int n; cin >> n;
6      for (int i = 1; i <= n; ++i) cin >> w[i];
7      f[0] = 0;
8      f[1] = w[1];
9      for (int i = 2; i <= n; ++i){
10         f[i] = max( f[i-2] + w[i], f[i-1] );
11     }
12     return 0;
13 }
```

# Complejidad

## Preguntas

- ¿Cuántos subproblemas (valores de  $f[i]$ ) hay que calcular?
- ¿Cuánto se demora calcular cada subproblema?

# Complejidad

## Preguntas

- ¿Cuántos subproblemas (valores de  $f[i]$ ) hay que calcular?
- ¿Cuánto se demora calcular cada subproblema?

## Complejidad

La complejidad de este algoritmo es  $O(n)$

# Contenido

## 3 Programación dinámica

# Programación dinámica

Las características principales que tiene un problema de programación dinámica son:

- Se puede hallar la solución a un número de subproblemas triviales
- La solución a los demás subproblemas se puede hallar usando la información de subproblemas más pequeños.

Usualmente los problemas de programación dinámica se pueden expresar en forma de una función recursiva.

# Programación dinámica

Cuando se identifica un problema como de programación dinámica se deben identificar los siguientes elementos:

- Relación recursiva entre los problemas
- Casos base
- Verificar que los casos base sí sean suficientes para construir todos los valores recursivamente

# Programación dinámica

Una vez hallada la relación entre los subproblemas, se crea una tabla que tenga capacidad para almacenar todos los subproblemas (tamaño para cada una de las variables).

## Ejemplo

$w(i, j) = w(i + 1, j + 1) + w(i + 1, j - 1)$  para  $1 \leq i, j \leq 100$

Se debe crear la tabla `w[105][105]`



# Programación dinámica

Luego de crear la tabla hay que hallar la forma de llenarla.

Primero se llenan los casos base.

Luego se llenan los casos recursivos.

El orden en el que se llenan los casos recursivos es importante ya que cuando se acceda a algún valor en la tabla, este ya se debe haber calculado con anterioridad.

# Ejemplo

Casos base:

$$w(i, 0) = 0 \quad \text{para } 0 \leq i \leq 100$$

$$w(100, j) = j \quad \text{para } 0 \leq j \leq 100$$

Caso recursivo:

$$w(i, j) = w(i + 1, j) + w(i, j - 1) \quad \text{para } 0 \leq i \leq 99, 1 \leq j \leq 100$$

Algoritmo:

---

```
1  int w[105][105];
2  for (int i = 0; i <= 100; ++i) w[i][0] = 0;
3  for (int j = 0; j <= 100; ++j) w[100][j] = j;
4
5  for (int i = 99; i >= 0; --i){
6      for (int j = 1; j <= 100; ++j){
7          w[i][j] = w[i+1][j] + w[i][j-1];
8      }
9  }
```

---

# Contenido

## 4 Problema de la mochila

# Problema da la mochila - Knapsack Problem

Un ladrón quiere robar una casa.

Él conoce los elementos que hay en la casa, su valor y su tamaño.

Sin embargo, tiene una bolsa de capacidad limitada, por lo que no puede robar todos los elementos.

¿Cuáles elementos debe robar para obtener la mayor ganancia posible y no superar la capacidad de su bolsa?



# Problema da la mochila - Knapsack Problem

## Entrada

- $n$  elementos cada uno con
  - Valor  $v_i$  no negativo
  - Tamaño  $w_i$  entero
- Capacidad  $W$  de la mochila

## Objetivo

Encontrar un subconjunto  $S \subset 1, 2, \dots, n$  tal que:

- $\sum_{i \in S} v_i$  se máxima
- $\sum_{i \in S} w_i \leq W$

# Preguntas

Se quiere hallar la mejor ganancia usando los elementos  $1 \dots n$  y teniendo una capacidad máxima de  $W$

## Preguntas

- ¿Se puede hallar esa solución basándose en la solución que usa los elementos  $1 \dots n - 1$ ?

# Preguntas

Se quiere hallar la mejor ganancia usando los elementos  $1 \dots n$  y teniendo una capacidad máxima de  $W$

## Preguntas

- ¿Se puede hallar esa solución basándose en la solución que usa los elementos  $1 \dots n - 1$ ?
- Si se tiene la solución para  $n - 1$  elementos y una capacidad  $W$  y se quiere hallar la solución para el  $n$ -ésimo elemento ¿qué opciones se tienen con ese elemento?

# Preguntas

Se quiere hallar la mejor ganancia usando los elementos  $1 \dots n$  y teniendo una capacidad máxima de  $W$

## Preguntas

- ¿Se puede hallar esa solución basándose en la solución que usa los elementos  $1 \dots n - 1$ ?
- Si se tiene la solución para  $n - 1$  elementos y una capacidad  $W$  y se quiere hallar la solución para el  $n$ -ésimo elemento ¿qué opciones se tienen con ese elemento?
- Si no se decide agregar el  $n$ -ésimo elemento ¿cuál es la solución?
- Si se decide agregar el  $n$ -ésimo elemento ¿cuál es la solución?



# Knapsack

Sea  $dp[i][j]$  la máxima ganancia que se puede obtener si se usan los elementos  $[1 \dots i]$  y se tiene una mochila de capacidad  $j$ .

$$dp[0][j] = 0 \quad \text{para } 0 \leq j \leq W$$

$$dp[i][j] = \max \left\{ \begin{array}{ll} dp[i-1][j] & \\ dp[i-1][j-w[i]] + v[i] & \text{si } j \geq w[i] \end{array} \right\} \quad \text{para } \begin{array}{l} 1 \leq i \leq n \\ 0 \leq j \leq W \end{array}$$

# Implementación

---

```
1  int dp[MAXN][MAXW];
2  // Casos base
3  for (int j = 0; j <= W; ++j) dp[0][j] = 0;
4  // Casos recursivos
5  for (int i = 1; i <= n; ++i){
6      for (int j = 0; j <= W; ++j){
7          dp[i][j] = dp[i-1][j];
8          if (j - w[i] >= 0){
9              dp[i][j] = max(dp[i][j], dp[i-1][j-w[i]] + v[i]);
10         }
11     }
12 }
```

---

# Complejidad

## NP-Completo

El problema de la mochila es NP-Completo, es decir que no se conoce algoritmo que lo resuelva en tiempo polinomial.

# Complejidad

## NP-Completo

El problema de la mochila es NP-Completo, es decir que no se conoce algoritmo que lo resuelva en tiempo polinomial.

## Preguntas

¿Cuántos subproblemas hay que resolver para hallar la solución a un problema con  $n$  artículos y una mochila de tamaño  $W$ ?

# Complejidad

## NP-Completo

El problema de la mochila es NP-Completo, es decir que no se conoce algoritmo que lo resuelva en tiempo polinomial.

## Preguntas

¿Cuántos subproblemas hay que resolver para hallar la solución a un problema con  $n$  artículos y una mochila de tamaño  $W$ ?

## Complejidad

La solución de programación dinámica al problema de la mochila con  $n$  artículos y una mochila de tamaño  $W$  tiene una complejidad de  $O(n \times W)$

# Contenido

## 5 Longest Common Subsequence

# Longest Common Subsequence (LCS)

## Entrada

Dos secuencias  $x_1, x_2, \dots, x_n$  y  $y_1, y_2, \dots, y_m$ .

## Objetivo

Encontrar la longitud de la subsecuencia común más larga entre  $x$  y  $y$ .

Una subsecuencia de una secuencia  $s$  es una secuencia que se puede obtener de  $s$  al borrarle algunos de sus elementos (probablemente todos) sin cambiar el orden de los elementos restantes.

- $(1, 6, 5)$  es una subsecuencia de  $(3, 1, 6, 7, 5, 2)$

# Ejemplos LCS

## Ejemplos

- La subsecuencia común más larga entre  $(6, 4, 1)$  y  $(3, 6, 1, 8)$  es  $(6, 1)$ .
- La subsecuencia común más larga entre  $(2, 5, 1)$  y  $(3, 0, 8)$  es  $()$ .
- La subsecuencia común más larga entre “ACAATCC” y “AGCATGC” es “ACATC”.



# Preguntas

Se quiere hallar la solución usando los caracteres  $1 \dots n$  de  $x$  y  $1 \dots m$  de  $y$ .

## Preguntas

- ¿Qué posibilidades se tienen si los caracteres  $x_n$  y  $y_m$  son diferentes?
- ¿Qué posibilidades se tienen si los caracteres  $x_n$  y  $y_m$  son iguales?

# Longest Common Subsequence

Sea  $dp[i][j]$  la longitud de la subsecuencia común más larga entre las secuencias  $x_1, x_2, \dots, x_i$  y  $y_1, y_2, \dots, y_j$

$$dp[0][j] = 0 \quad \text{para } 0 \leq j \leq m$$

$$dp[i][0] = 0 \quad \text{para } 0 \leq i \leq n$$

$$dp[i][j] = \max \left\{ \begin{array}{ll} dp[i-1][j] & \text{(No usar } x_i) \\ dp[i][j-1] & \text{(No usar } y_j) \\ dp[i-1][j-1] + 1 & \text{si } x_i = y_j \end{array} \right\} \quad \text{para } \begin{array}{l} 1 \leq i \leq n \\ 1 \leq j \leq m \end{array}$$

# Implementación

---

```
1  int dp[MAXN][MAXM]
2  for (int j = 0; j <= m; ++j) dp[0][j] = 0;
3  for (int i = 0; i <= n; ++i) dp[i][0] = 0;
4
5  for (int i = 1; i <= n; ++i){
6      for (int j = 1; j <= m; ++j){
7          dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
8          if (x[i] == y[j]){
9              dp[i][j] = max(dp[i][j], dp[i-1][j-1] + 1);
10         }
11     }
12 }
```

---

# Complejidad

## Preguntas

¿Cuántos subproblemas hay que resolver para hallar la solución a un problema con las secuencias  $x_1, x_2, \dots, x_n$  y  $y_1, y_2, \dots, y_m$ ?

# Complejidad

## Preguntas

¿Cuántos subproblemas hay que resolver para hallar la solución a un problema con las secuencias  $x_1, x_2, \dots, x_n$  y  $y_1, y_2, \dots, y_m$ ?

## Complejidad

La solución de programación dinámica al problema de LCS con dos secuencias de tamaño  $n$  y  $m$  tiene una complejidad de  $O(n \times m)$

# Contenido

## 6 Longest Increasing Subsequence

# Longest Increasing Subsequence (LIS)

## Entrada

Una secuencia  $x_1, x_2, \dots, x_n$

## Objetivo

Encontrar la longitud de la subsecuencia creciente más larga que hay en  $x$ .

## Ejemplo

- La subsecuencia creciente más larga de  $(-7, 10, 9, 2, 3, 8, 8, 1)$  es  $(-7, 2, 3, 8)$

# Pregunta

## Pregunta

¿Cómo se puede resolver este problema usando LCS?



# Pregunta

## Pregunta

¿Cómo se puede resolver este problema usando LCS?

## Respuesta

Hallar la solución a LCS usando la secuencia  $x$  y  $x$  ordenado de menor a mayor.

Complejidad  $O(n^2)$

# Pregunta

## Pregunta

¿Cómo se puede resolver este problema usando LCS?

## Respuesta

Hallar la solución a LCS usando la secuencia  $x$  y  $x$  ordenado de menor a mayor.

Complejidad  $O(n^2)$

## Otras soluciones

Existen otras soluciones para el problema de LIS, incluyendo una solución en  $O(n \log n)$

# Contenido

## 7 Tarea

# Tarea

## Tarea

Registrarse en las páginas:

- <http://www.codeforces.com/>
- <http://www.spoj.com/>
- <http://ahmed-aly.com/> - Ingresar los usuarios de codeforces, spoj y UVa.

Resolver los problemas de

<http://ahmed-aly.com/Contest.jsp?ID=4312>

# Ayudas

## Problema 2

Una forma de hacerlo es con la siguiente programación dinámica:

$dp[i][j][k]$  es el mínimo peso que se puede llevar si se usan los tanques  $1 \dots i$  y se quieren llevar **al menos**  $j$  litros de oxígeno y  $k$  litros de nitrógeno.

Pensarlo como una modificación de knapsack:

- Ver cómo modificar knapsack para que sea llevar al menos un volumen de  $j$  y no un volumen de máximo  $j$ .
- Luego de ver cómo hacer esas modificaciones tratar de hallar la fórmula recursiva y los casos base para el problema.

# Ayudas

## Problema 4

Cuidado que si la entrada dice 2 3 4 1 significa que en la secuencia el 1 está en la segunda posición, el 2 en la tercera, el 3 en la cuarta y el 4 en la primera.

Esto quiere decir que la secuencia es 4 1 2 3