Programación dinámica

Alberto Verdejo

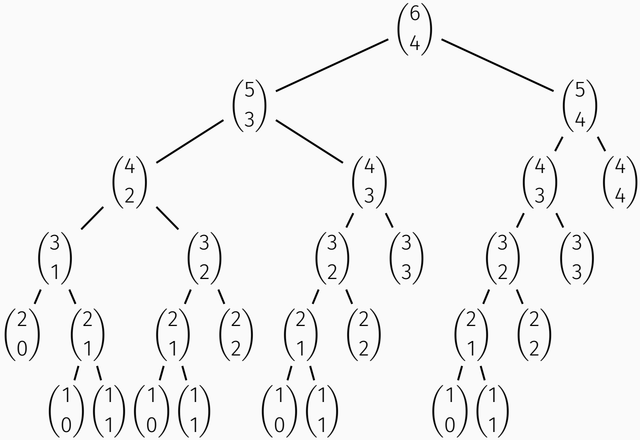
Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Programación dinámica. Motivación

$$\binom{n}{r} = \begin{cases} 1 & \text{si } r = 0 \ \lor \ r = n \\ \binom{n-1}{r-1} + \binom{n-1}{r} & \text{si } 0 < r < n \end{cases}$$

Programación dinámica. Motivación

$$\binom{n}{r} = \begin{cases} 1 & \text{si } r = 0 \lor r = n \\ \binom{n-1}{r-1} + \binom{n-1}{r} & \text{si } 0 < r < n \end{cases}$$



Programación dinámica *top-down*

Utilizar una tabla para guardar los valores ya calculados.

```
int C[MAX+1][MAX+1];

int comb(int i, int j) {
    if (j == 0 || j == i)
        return 1;
    if (C[i][j] != -1)
        return C[i][j];
    return C[i][j] = comb(i-1, j-1) + comb(i-1, j);
}
```

La tabla se inicializa con un valor distinto, por ejemplo -1.

```
memset(C, -1, sizeof(C));

cout << comb(n, r) << '\n';
```

Programación dinámica *bottom-up*

	0	1	2	...	r
0	1	0	0	...	0
1	1	1	0	...	0
2	1	2	1	...	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
n	1	n	$\binom{n}{2}$	...	$\binom{n}{r}$

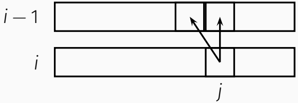
```c
int C[MAX+1][MAX+1];

int pascal(int n, int r) {
    memset(C, 0, (r+1)*sizeof(C[0][0]));
    C[0][0] = 1;
    for (int i = 1; i <= n; ++i) {
        C[i][0] = 1;
        for (int j = 1; j <= r; ++j)
            C[i][j] = C[i-1][j-1] + C[i-1][j];
    }
    return C[n][r];
}
```

Mejorar espacio adicional

Dejando aparte los casos básicos, para calcular cada entrada (i, j) en la tabla se necesitan las entradas $(i-1, j-1)$ e $(i-1, j)$ de la fila anterior, por lo que el espacio adicional se puede reducir a un vector que se rellena de derecha a izquierda.

Mejorar espacio adicional

Dejando aparte los casos básicos, para calcular cada entrada (i, j) en la tabla se necesitan las entradas $(i-1, j-1)$ e $(i-1, j)$ de la fila anterior, por lo que el espacio adicional se puede reducir a un vector que se rellena de derecha a izquierda.




```
int C[MAX+1];
int pascal2(int n, int r) {
    memset(C, 0, (r+1)*sizeof(C[0]));
    C[0] = 1;
    for (int i = 1; i <= n; ++i)
        for (int j = r; j >= 1; --j)
            C[j] = C[j-1] + C[j];
    return C[r];
}
```

Coste: en tiempo $\Theta(nr)$
en espacio $\Theta(r)$

Problema de la mochila (versión entera)

- En la cueva hay n objetos, cada uno con un peso (natural) $p_i > 0$ y un valor (real) $v_i > 0$ para todo i entre 1 y n.
- La mochila soporta un peso total (natural) máximo $M > 0$.
- El problema consiste en maximizar

$$\sum_{i=1}^{n} x_i v_i$$

  con la restricción

$$\sum_{i=1}^{n} x_i p_i \leqslant M,$$

  donde $x_i \in \{0, 1\}$ indica si hemos cogido (1) o no (0) el objeto i.

Problema de la mochila (versión entera)

Definimos una función

$$mochila(i, j) \quad = \quad \textit{máximo} \text{ valor que podemos poner en la mochila de peso máximo } j \text{ considerando los objetos del 1 al } i.$$

Problema de la mochila (versión entera)

Definimos una función

$$mochila(i,j) \quad = \quad \textit{máximo} \text{ valor que podemos poner en la mochila de peso máximo } j \text{ considerando los objetos del 1 al } i.$$

Definición recursiva

$$mochila(i,j) = \begin{cases} mochila(i-1,j) & \text{si } p_i > j \\ \text{máx}(\, mochila(i-1,j), mochila(i-1,j-p_i) + v_i\,) & \text{si } p_i \leqslant j \end{cases}$$

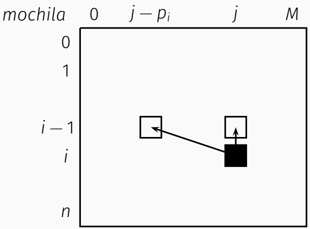
con $1 \leqslant i \leqslant n$ y $1 \leqslant j \leqslant M$.

Los casos básicos son:

$$\begin{array}{lcll} mochila(0,j) & = & 0 & 0 \leqslant j \leqslant M \\ mochila(i,0) & = & 0 & 0 \leqslant i \leqslant n. \end{array}$$

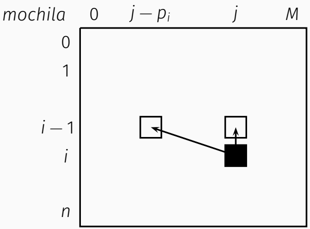
Problema inicial: $mochila(n, M)$.

Problema de la mochila (versión entera)



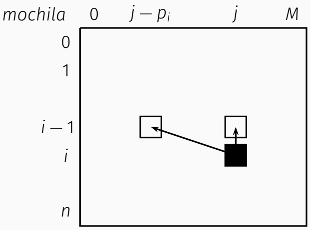
Recorrer la matriz por filas de arriba abajo y cada fila de izquierda a derecha.

Problema de la mochila (versión entera)



Recorrer la matriz por filas de arriba abajo y cada fila de izquierda a derecha.
Si solo quisiéramos el valor máximo alcanzable, podríamos optimizar el espacio adicional utilizando solo un vector que recorreríamos de derecha a izquierda.

Problema de la mochila (versión entera)



Recorrer la matriz por filas de arriba abajo y cada fila de izquierda a derecha.

Si solo quisiéramos el valor máximo alcanzable, podríamos optimizar el espacio adicional utilizando solo un vector que recorreríamos de derecha a izquierda.

Si queremos devolver la solución óptima no podemos optimizar, porque en ese caso las comparaciones que hacemos para llenar una posición siempre se refieren a posiciones de la fila anterior:

$$mochila(i,j) = \text{máx}\big(\underbrace{mochila(i-1,j)}_{\text{no cogemos el objeto } i},\ \underbrace{mochila(i-1,j-p_i)+v_i}_{\text{sí cogemos el objeto } i}\big)$$

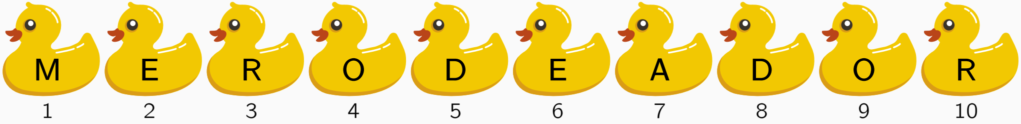
Problema de la mochila (versión entera), implementación

```
int p[NMAX]; int v[NMAX]; // 0-based
int n; int M;
int valor; bool cuales[NMAX];
int pd[NMAX+1][MMAX+1];

void mochila() {
    memset(pd, 0, (M+1)*sizeof(pd[0][0]));
    for (int i = 1; i <= n; ++i) {
        pd[i][0] = 0;
        for (int j = 1; j <= M; ++j) {
            if (p[i-1] > j)
                pd[i][j] = pd[i-1][j];
            else
                pd[i][j] = max(pd[i-1][j], pd[i-1][j - p[i-1]] + v[i-1]);
        }
    }
    valor = pd[n][M];
```

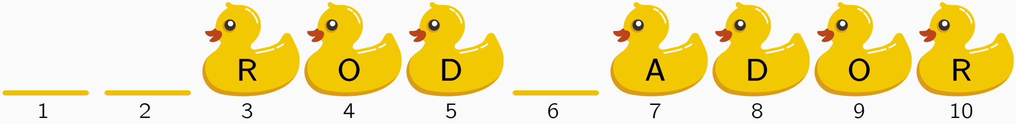
```
    // cálculo de los objetos
    int resto = M;
    for (int i = n; i >= 1; --i) {
        if (pd[i][resto] == pd[i-1][resto]) {
            // no cogemos objeto i
            cuales[i] = false;
        } else { // sí cogemos objeto i
            cuales[i] = true;
            resto = resto - p[i-1];
        }
    }
}
```

Coste: $\Theta(nM)$ tanto en tiempo como en espacio adicional.

Tiro al patíndromo



Conseguir el palíndromo más largo tirando (si es necesario) algunos de los patitos:



Tiro al patíndromo

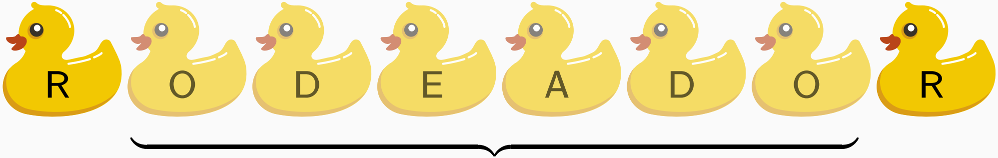
Si los patitos de los extremos coinciden

lo mejor es seleccionarlos y buscar el mejor palíndromo con el resto de patitos:

Tiro al patíndromo

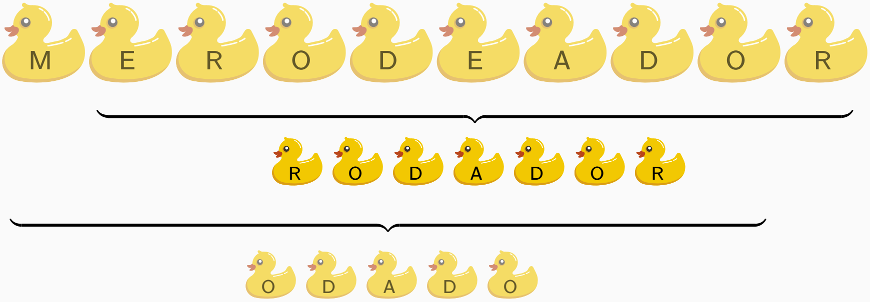
Si los patitos de los extremos coinciden

lo mejor es seleccionarlos y buscar el mejor palíndromo con el resto de patitos:



Tiro al patíndromo

Si los patitos de los extremos no coinciden, entonces habrá que quitar al menos uno de ellos, buscar recursivamente en cada opción, y quedarse con la mejor:



Tiro al patíndromo

$$\textit{patíndromo}(i, j) \quad = \quad \text{longitud del palíndromo más largo que podemos obtener con } \textit{patitos}[i..j]$$

Tiro al patíndromo

$$\begin{array}{rcl} patíndromo(i,j) & = & \text{longitud del palíndromo más largo que podemos} \\ & & \text{obtener con } patitos[i..j] \end{array}$$

Definición recursiva ($i < j$):

$$patíndromo(i,j) = \begin{cases} patíndromo(i+1, j-1) + 2 & \text{si } patitos[i] = patitos[j] \\ \text{máx}(patíndromo(i+1, j), patíndromo(i, j-1)) & \text{si } patitos[i] \neq patitos[j] \end{cases}$$

Casos básicos:

$$\begin{array}{rcl} patindromo(i,i) & = & 1 \\ patindromo(i,j) & = & 0 \quad \text{si } i > j \end{array}$$

Tiro al patíndromo, implementación

```
#define MAXLONG 1000
int pd[MAXLONG][MAXLONG];
string patitos;  // 0-based

int patin(int i, int j) {  // patitos[i..j]
   int & res = pd[i][j];
   if (res == -1) {
      if (i > j) res = 0;
      else if (i == j) res = 1;
      else if (patitos[i] == patitos[j])
         res = 2 + patin(i+1,j-1);
      else
         res = max(patin(i+1, j), patin(i, j-1));
   }
   return res;
}
```

Tiro al patíndromo, implementación

Reconstruir la solución a partir de la tabla

```
void print(int i, int j) {
   if (i > j) return;
   if (i == j) cout << patitos[i];
   else if (patitos[i] == patitos[j]) {
      cout << patitos[i];
      print(i+1, j-1);
      cout << patitos[j];
   } else if (pd[i][j] == pd[i+1][j])
      print(i+1, j);
   else
      print(i, j-1);
}
```

Tiro al patíndromo, implementación

```
bool resuelveCaso() {
   cin >> patitos;

   if (!cin) return false;

   int N = patitos.length();
   for (int i = 0; i < N; ++i)
      for (int j = i; j < N; ++j)
         pd[i][j] = -1;

   patin(0, N-1);

   print(0, N-1); cout << '\n';

   return true;
}
```

Tiro al patíndromo, implementación

```
bool resuelveCaso() {
   cin >> patitos;

   if (!cin) return false;

   int N = patitos.length();
   for (int i = 0; i < N; ++i)
      for (int j = i; j < N; ++j)
         pd[i][j] = -1;

   patin(0, N-1);

   print(0, N-1); cout << '\n';

   return true;
}
```

ACR 322 - Tiro al patíndromo