Programación Dinámica (1)

Introducción

- Utilización de una tabla (array multidimensional) donde se almacenan los resultados a subproblemas ya resueltos
- La tabla tiene tantas dimensiones como argumentos tiene la recurrencia
- El tamaño de cada dimensión coincide con los valores que puede tomar el argumento correspondiente
- Cada subproblema se asocia a una posición de la tabla

Programación Dinámica Descendente

- Mantiene el diseño recursivo.
- La función recibe como parámetro de entrada/salida, la tabla donde se almacenan las soluciones a subproblemas ya resueltos.

- Antes de resolver de manera recursiva un subproblema, se mira en la tabla por si ya se hubiera resuelto.
- Tras resolver un subproblema recursivo, su solución se almacena en la tabla.
- Necesidad de saber si un subproblema esta resuelto o no.

La solución para el problema del número de subconjuntos de cardinal r que tiene un conjunto de n elementos

```
// coste O(nr) y en espacio 0(nr)
int num_combi(int i, int j, Matriz<int> & C) {
    if (j == 0 || j == i) return 1;
    else if (C[i][j] != -1) return C[i][j];
    else {
        C[i][j] = num_combi(i-1, j-1, C) + num_combi(i-1, j, C);
        return C[i][j];
    }
  }
```

Programación Dinámica Ascendente

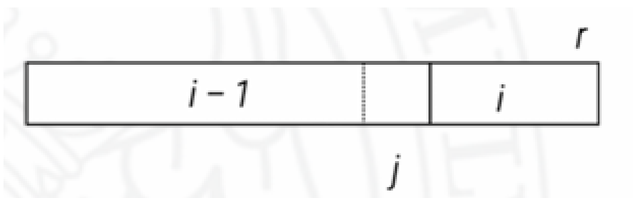
- Cambiar el orden en el que se resuelven los subproblemas
- Comenzar por resolver todos los subproblemas más pequeños que se puedan necesitar para ir combinándolos hasta llegar a resolver el problema original.
- Los subproblemas se van resolviendo recorriéndolos de menor a mayor tamaño
- Todos los posible subproblemas de tamaño menor tienen que ser resueltos antes de resolver uno de tamaño mayor.

La solución primera al problema sería:

```
 // coste O(nr) y en espacio O(nr)
int pascal(int n, int r) {
 Matriz<int> C(n+1,r+1,0);
 C[0][0] = 1;
 for (int i = 1; i <= n; ++i) {
     C[i][0] = 1; // inicialización de la primera fila
     for (int j = 1; j <= r; ++j)
         C[i][j] = C[i-1][j-1] + C[i-1][j];
 }
 return C[n][r];
}
```

	0	1	2	…	r
0	1	0	0	…	0
1	1	1	0	…	0
2	1	2	1	…	0
⋮	⋮	⋮	⋮	⋱	⋮
n	1	n	$\binom{n}{2}$	…	$\binom{n}{r}$

Sin embargo es posible utilizar un vector. En el presente problema solo requerimos de los números inmediatamente superior y el superior-izquierda. Por ello utilizamos la parte izquierda del vector como la fila superior.




```
int pascal2(int n, int r) {
    vector<int> C(r+1,0);
    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]
}
```

Esquema General

Para realizar este tipo de problemas es importante realizar primero la especificación. Los pasos son los siguientes

1. Determinar una función con la siguiente estructura

```
function(k, l) ⇒ nº de algo (puede ser tambiene minimo o maximo) para llegar
                                        con los k primeros objetos (festivales, cuerdas, tareas)
```

2. Determinar el caso base, normalmente

```
- function(k, 0) = cb1 ⇒ ¿se necesitan objetos para llegar al tamaño 0?
- function(0, l) = cb2 (donde l>0)⇒ ¿hay forma de llegar al tamaño l sin objeto
```

3. Determinar el caso recursivo, normalmente:

```
- Si No se puede usar porque no cabe para el tamño restante (objetos[k-1].len
    - function (k, l) = function (k-1, l)
- Si Si se puede usar, diferenciar entre usarlo o no usarlo
    - function (k, l) = function(k-1, l) o function (k-1, l - objetos[k-1].length) + (ob
```

4. Las columnas en la tabla serán `l` y `k` serán las filas.
5. Desarrollar el codigo, que sería algo así:

```
retType getFunction(vector<Objet> const& objetos, int L, int K) {
    vector<retType> function(L + 1, cb2);
    function[0] = cb1;

    for (int k = 1; k <= K; ++k) {
        for (int l = L; l >= objetos[k - 1].length; l--) {
            function[l] = function(k-1, l) o function (k-1, l - objetos[k-1].length) + (ob
        }
    }
    return function[L];
}
```

6. Ejemplos

```
/*
*   minNum (k, l) = nº de cuerdas necesarias para formar una cuerda de tamañ
*
*   CASO BASE
*       - minNum (k, 0) = 0 => no se ncesitan cuerdas para una cuerda de tamañ
*       - minNum (0, l) = inf  donde l > 0 => no hay forma de construir una cuerd
*
*   CASO RECURSIVO
*       - No se puede usar cordel => cuerdas(k-1).longitud > l
*           - minNum (k, l) = minNum (k - 1, l)
*       - El minimo numero de cordeles para componer la cuerda es el minimo d
*            - minNum (k, l) = min(minNum(k-1, l), minNum(k-1, l - cuerdas[k - 1].lo
*/

EntInf minimoNumero(vector<Cuerda> const& cuerdas, int L, int N) {
    vector<EntInf> minNum(L + 1, Infinito);
    minNum[0] = 0;

    for (int k = 1; k <= N; ++k) {
        for (int l = L; l >= cuerdas[k - 1].longitud; l--) {
            minNum[l] = min(minNum[l], minNum[l - cuerdas[k - 1].longitud] + 1);
        }
    }
    return minNum[L];
}
```

```
/*
*   coste ( k, l) = minimo coste de componer la cuerda de tamaño l con los k pr
*
*   CASO BASE
*       - coste (k, 0) = 0 => no hay coste en componer una cuerda de tamaño 0
*       - coste (0, l) = inf  donde l > 0 => no hay coste de contruir una cuerda de
*
*   CASO RECURSIVO
*       - No se puede usar cordel => cuerdas(k-1).longitud > l
*           - coste (k, l) = coste (k - 1, l)
*       - El coste de componer la cuerda es el minimo coste de usar o no k
*           - coste (k, l) = min(coste(k-1, l), coste(k-1, l - cuerdas[k - 1].longitud) +
*
*/

EntInf minimoCoste(vector<Cuerda> const& cuerdas, int L, int N) {
    vector<EntInf> coste(L + 1, Infinito);
    coste[0] = 0;
    for (int k = 1; k <= N; ++k) {
        for (int l = L; l >= cuerdas[k - 1].longitud; l--) {
            coste[l] = min(coste[l], coste[l - cuerdas[k - 1].longitud] + cuerdas[k-1]
        }
    }
    return coste[L];
}
```

Algoritmo de Floyd

- Dado un grafo valorado, calcular el camino de coste mínimo entre cada par de vértices.
- Si los pesos son positivos y el grafo es disperso, utilizando el algoritmo de Dijkstra V veces, obtenemos un algoritmo con coste en O(V A log V).
- El algoritmo de Floyd resuelve el caso general, con pesos posiblemente negativos, con coste en O(V^3).

Definición de Recurrencia

- C^k(i, j) = coste mínimo para ir de i a j pudiendo utilizar como vértices intermedios aquellos entre 1 y k.
- Caso recursivo
    - mínimo de ir directamente de i a j, o la suma por caminos intermedios

$$C^k(i,j) = \min(C^{k-1}(i,j), C^{k-1}(i,k) + C^{k-1}(k,j))$$

$$C^0 = G$$

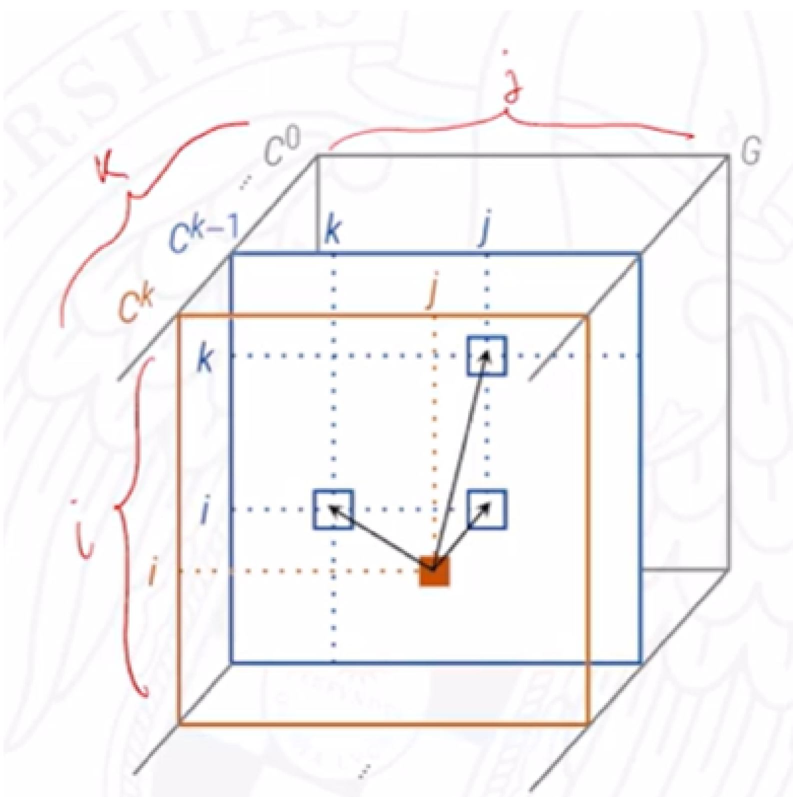
  - G == matriz de adyacencia

- Caso Base

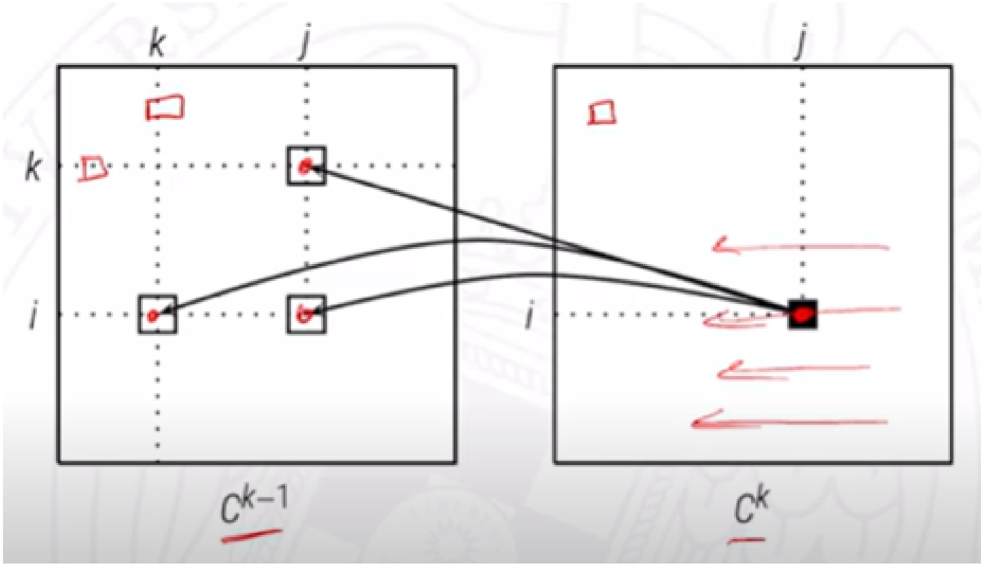
$$G[i][j] = \begin{cases} 0 & \text{si } i = j \\ coste & \text{si hay arista de } i \text{ a } j \\ +\infty & \text{si no hay arista de } i \text{ a } j \end{cases}$$

Tabla

- Necesitaremos una tabla de tres dimensiones, para la k, la i y la j.



- La matriz con k=0 se rellenará con la matriz de adyacencia de G.
- La k irá avanzando y siempre tendrá a su anterior completado.
- Como solo necesitamos la matriz anterior, podremos mejorar el espacio adicional a 2 matrices.
- Incluso se podrá solo necesitar una matriz recorriendola de abajo a arriba y de derecha a izquierda. El problema sería calcular el valor dentro de las columnas y filas k, pero esos valores son 0 como indica la recursión.



$$C^k(k,j) = \text{mín}(C^{k-1}(k,j), \underbrace{C^{k-1}(k,k)}_{0} + C^{k-1}(k,j)) = C^{k-1}(k,j)$$

Reconstrucción

$A^k(i,j)$ = vértice anterior a j en el camino mínimo de i a j pudiendo utilizar como vértices intermedios aquellos entre 1 y k

$$A^0(i,j) = \begin{cases} -1 & \text{si } i = j \vee G[i][j] = +\infty \\ i & \text{si } i \neq j \wedge G[i][j] < +\infty \end{cases}$$

$$A^k(i,j) = \begin{cases} A^{k-1}(i,j) & \text{si } C^{k-1}(i,j) \leq C^{k-1}(i,k) + C^{k-1}(k,j) \\ A^{k-1}(k,j) & \text{si } C^{k-1}(i,j) > C^{k-1}(i,k) + C^{k-1}(k,j) \end{cases}$$

Código

```
bool Floyd(Matriz<EntInf> const& G, Matriz<EntInf>& C, Matriz<int>& A) {
   int V = G.numfils(); // número de vértices de G
   // inicialización
   C = G;
   A = Matriz<int>(V, V, -1);
   for (int i = 0; i < V; ++i) {
      for (int j = 0; j < V; ++j) {
         if (i != j && G[i][j] != Infinito)
            A[i][j] = i;
      }
   }

   // actualizaciones de las matrices
```

```
    for (int k = 0; k < V; ++k) {
        for (int i = 0; i < V; ++i) {
            for (int j = 0; j < V; ++j) {
                auto temp = C[i][k] + C[k][j];
                if (temp < C[i][j]) { // es mejor pasar por k
                    C[i][j] = temp;
                    A[i][j] = A[k][j];
                }
            }
            if (C[i][i] < 0) return false;
        }
    }
    return true;
}

using Camino = std::deque<int>;
Camino ir_de(int i, int j, Matriz<int> const& A) {
    Camino cam;
    while (j != i) {
        cam.push_front(j);
        j = A[i][j];
    }
    cam.push_front(i);
    return cam;
}
```

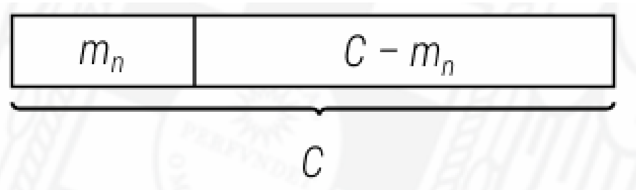
Problemas

Cambio de Monedas

- Conjunto finito M de tipos de monedas, donde cada mi es un número natural.
- Existe una cantidad ilimitada de monedas de cada valor.
- Se quiere pagar una cantidad C > 0 utilizando el menor número posible de monedas.

Al no funcionar una estrategia voraz, tenemos que considerar diferentes alternativas hasta encontrar la mejor.

- Las soluciones son multiconjuntos de monedas.
- Podemos fijar el orden en el que vamos considerando los tipos de monedas, sin que eso afecte el resultado final
- Con la moneda de valor n podemos:
    1. Usar la moneda para pagar y ver lo que queda



2. Probar que podemos hacer con el resto de monedas sin contar n

- En cada caso tendremos un subproblema porque o bien porque queda menos por pagar o bien por que tenemos menos tipos de monedas.
- Teniendo `monedas(i, j)` = número mínimo de monedas para pagar la cantidad j considerando los tipos de monedas del 1 al i

> Principio de optimalidad de Bellman ⇒ para conseguir una solución óptima basta con considerar subsoluciones óptimas. Este principio se cumple si una solución óptima a una instancia del problema siempre contiene soluciones óptimas a todas sus subinstancias.

Definición Recursiva

- Casos recursivos

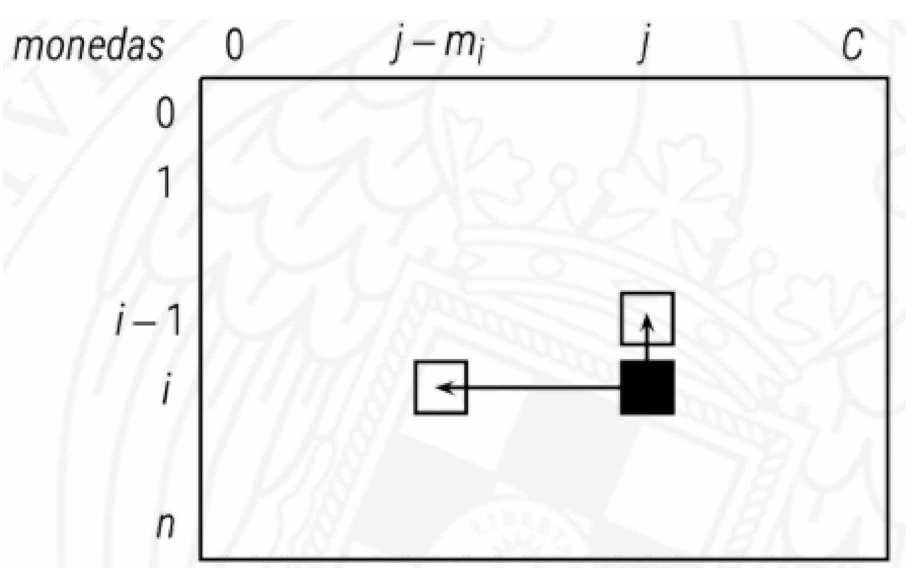
$$monedas(i,j) = \begin{cases} monedas(i-1,\ j) & \text{if } m^i > j \\ min(monedas(i-1,\ j), monedas(i,\ j-m^i)) & \text{if } m^i \leqslant j \end{cases}$$

- Casis básicos (no me queda nada más por pagar o no me queden más tipos de monedas):

$$\begin{array}{rc} monedas(i,0) = 0 & 0 \leqslant i \leqslant n \\ monedas(0,j) = +\infty & 1 \leqslant j \leqslant C \end{array}$$

- Llamada inicial ⇒ `monedas(n, C)`

Tabla

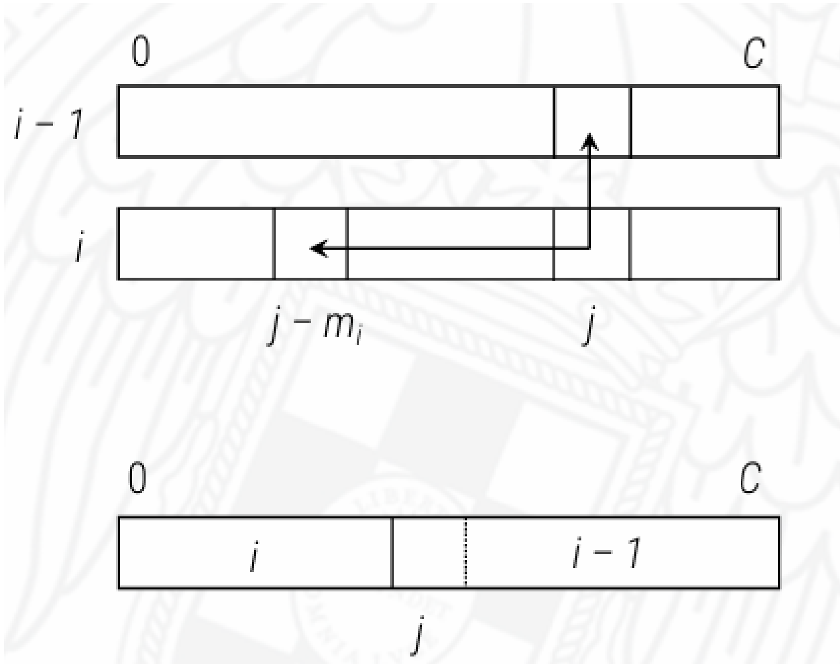


El tamaño de la tabla será las n tipo de monedas por la cantidad C a pagar. Tamaño de la tabla = n+1 y c+1

Código

```
vector<int> devolver_cambio(vector<int> const& M, int C) {
    //rellenar la matriz de monedas
     int n = M.size();
     Matriz<EntInf> monedas(n+1, C+1, Infinito);
     monedas[0][0] = 0;
     for (int i = 1; i <= n; ++i) {
         monedas[i][0] = 0;
         for (int j = 1; j <= C; ++j){
             if (M[i-1] > j)
                 monedas[i][j] = monedas[i-1][j];
             else
                 monedas[i][j] = min(monedas[i-1][j], monedas[i][j - M[i-1]] + 1);
        }
     }
     //obtener la solución
      vector<int> sol;
         if (monedas[n][C] != Infinito) {
             int i = n, j = C;
                 while (j > 0) { // no se ha pagado todo
                     if (M[i-1] <= j && monedas[i][j] != monedas[i-1][j]) {
                     // tomamos una moneda de tipo i
                     sol.push_back(M[i-1]); j = j - M[i-1];
                     } else // no tomamos más monedas de tipo i
                         --i;
             }
         }
         return sol;
 }
```

Se puede reducir el coste en espacio adicional si solamente se quiere calcular el número de monedas de la solución óptima,. Para rellenar la fila i solo necesitamos la fila i-1. Para calcular el valor de j solo se necesita el valor de j en i-1, por lo que una vez calculado se puede reemplazar. Por lo que solo se necesitará un vector de dimensión C.



Normalmente se necesita una matriz para recuperar la solución

```
vector<int> devolver_cambio(vector<int> const& M, int C) {
    int n = M.size();
    vector<EntInf> monedas(C+1, Infinito);
    monedas[0] = 0;
    // calcular la matriz sobre el propio vector
    for (int i = 1; i <= n; ++i) {
        for (int j = M[i-1]; j <= C; ++j) {
            monedas[j] = min(monedas[j], monedas[j - M[i-1]] + 1);
        }
    }
    vector<int> sol;
    if (monedas[C] != Infinito) {
        int i = n, j = C;
        while (j > 0) { // no se ha pagado todo
            if (M[i-1] <= j && monedas[j] == monedas[j - M[i-1]] + 1) {
                // tomamos una moneda de tipo i
                sol.push_back(M[i-1]);
                j = j - M[i-1];
            } else // no tomamos más monedas de tipo i
                --i;
        }
    }
    return sol;
}
```

Problema de la Mochila Entera

- Hay n objetos, cada uno con un peso (entero) pi > 0 y un valor (real) vi > 0
- La mochila soporta un peso total (entero) máximo M > 0
- El problema consiste en maximizar el valor sin pasarse de peso.

- Las soluciones no importa el orden en el que los objetos son introducidos en la mochila.
- Las soluciones dependen de los objetos que tengamos disponible para introducir en la mochila y del peso que soporte esta

Definición Recursiva

```
/*
*   mochila(k, l) = máximo valor que podemos poner en una mochila de peso máxi
*
*   CASO BASE:
*       - grupos (k, 0) = 0 no tenemos mas hueco en la mochila
*       - grupos (0, l) = 0 no tenemos objetos disponible
*   CASO RECURSIVO
*       - no cabe el objeto en la mochila
*           - mochila (k, l) = mochila(k-1, l)
*       - el maximo valor consistirá en cojer o no cojer el objeto
*           - mochila (k, l) = max( mochila(k-1, l), mochila(k -1 , l - pk) + vk)
*   LLAMADA
*       - mochila(n, M)
*/
```

Tabla

Si quisiesemos recuperar los valores necesitariamos una tabla de M * n, siendo n el numero de objetos y M el peso máximo de la mochila.

Si el problema no requiere recuperar los valores podremos usar un array y donde las j primeras posiciones son los resultados y de j a M son el resto.

Código

```
 struct Objeto { int peso; double valor; };
 double mochila_rec(vector<Objeto> const& obj, int k, int l,
                                    Matriz<double> & mochila) {
     if (mochila[k][l] != -1) // subproblema ya resuelto
         return mochila[k][l];
     if (i == 0 || j == 0) mochila[k][l] = 0;
     else if (obj[k-1].peso > l)
         mochila[k][l] = mochila_rec(obj, k-1, l, mochila);
     else
         mochila[k][l] = max(mochila_rec(obj, k-1, l, mochila),
                                          mochila_rec(obj, k-1, l - obj[k-1].peso, mochila)
                                          + obj[k-1].valor);
     return mochila[k][l];
}


 double mochila(vector<Objeto> const& objetos, int M, vector<bool> & sol) {
```

```
    int n = objetos.size();
    Matriz<double> mochila(n+1, M+1, -1);
    double valor = mochila_rec(objetos, n, M, mochila);

    int k = n, l = M;
    sol = vector<bool>(n, false);
    while (k > 0 && l > 0) {
        if (mochila[k][l] != mochila[k-1][l]) {
            sol[k-1] = true; l = l - objetos[k-1].peso;
        }
        --k;
    }
  return valor;
```

Tiro Al Palíndromo

- Dado una palabra , el objetivo es quitando algunas letras (si es necesario) formar el palíndromo más largo.
- Una palíndromo es una palabra que se lee igual de izquierda a derecha como de derecha a izquierda.

Definición Recursiva

```
/*
*   palindromo (k, l) = longitud del palíndromo más largo que podemos obtener co
*
*   CASO BASE:
*       - palindromo (k, k) = 1 ⇒ tenemos dos letras iguales
*       - palindromo (k, l) = 0 si k > l ⇒ no tenemos mas letras disponibles
*
*   CASO RECURSIVO (k < l)
*       - las letras k y l son iguales (letras[k] == letras[l])
*           - palindromo (k, l) = palindromo(k + 1, l -1) + 2
*       - si son distintas, entonces habra que comprobar si quitamos k o quitamos l
*           - palindromo (k, l) = max( palindromo(k+1, l), palindromo(k, l-1))
*   LLAMADA
*       - palindromo(0, n-1)
*/
```

Código

```
int patin_rec(string const& patitos, int i, int j, Matriz<int>& patin) {
    int& res = patin[i][j];
    if (res == -1) {
        if (i > j) res = 0;
        else if (i == j) res = 1;
```

```
        else if (patitos[i] == patitos[j])
            res = patin_rec(patitos, i + 1, j - 1, patin) + 2;
        else
            res = max(patin_rec(patitos, i + 1, j, patin),
                patin_rec(patitos, i, j - 1, patin));
    }
    return res;
}

void reconstruir(string const& patitos, Matriz<int> const& patin, int i, int j, string&
    if (i > j) return;
    if (i == j) sol.push_back(patitos[i]);
    else if (patitos[i] == patitos[j]) {
        sol.push_back(patitos[i]);
        reconstruir(patitos, patin, i + 1, j - 1, sol);
        sol.push_back(patitos[j]);
    }
    else if (patin[i][j] == patin[i + 1][j])
        reconstruir(patitos, patin, i + 1, j, sol);
    else
        reconstruir(patitos, patin, i, j - 1, sol);
}
```

Multiplicación de Matrices

- El producto de una matriz A*pxq* y una matriz de B*qxr* es una matriz de C*pxr*.
- Se necesitan *pqr* multiplicaciones entre números para calcular C
- Queremos multiplicar una secuencias de matrices M1, M2, ... Mn donde Mi tiene dimension di-1 x di.
- El producto de matrices no es conmutativa (xyz ≠ xzy) pero sí asociativa ((xy)z = x(yz)), es decir importa el orden pero no importa cuales agrupamos.
- ¿Cuál es la mejor forma de colocar paréntesis en la secuencia de multiplicaciones de forma que el número de multiplicaciones entre números sea mínimo?

$$A_{13\times 5} \cdot B_{5\times 89} \cdot C_{89\times 3} \cdot D_{3\times 34}$$

$$\underbrace{\underbrace{(\underbrace{(A\cdot B)}_{5785}\cdot C)}_{3471}\cdot D}_{1326} \rightsquigarrow 10582 \qquad \underbrace{\underbrace{(A\cdot \underbrace{(B\cdot C)}_{1335})}_{195}\cdot D}_{1326} \rightsquigarrow 2856$$

Definición Recursiva

El objetivo será definir cual será el ultimo. Si ese entre las matrices k y k+1 entonces habrá que definir el producto de las matrices de la 1 a la k y de k+1 a n.

- matrices(i, j) = número mínimo de multiplicaciones básicas para realizar el producto matricial Mi....Mj.
- todas las posibilidades de decidir cuál es el último producto:

$$
\begin{gathered}
M_i \cdot (M_{i+1} \cdot \ldots \cdot M_j) \\
(M_i \cdot M_{i+1}) \cdot (M_{i+2} \cdot \ldots \cdot M_j) \\
\ldots \\
(M_i \cdot \ldots \cdot M_{j-1}) \cdot M_j
\end{gathered}
$$

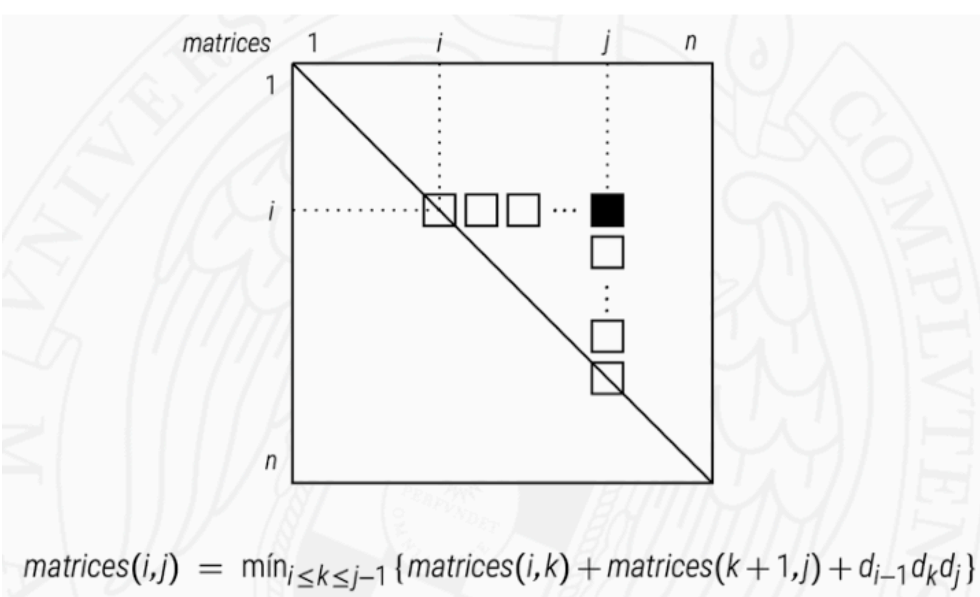
- Caso Recursivo, i < j

  k tiene muchos valores , por lo que habrá que buscar cuales son mejores. Consistirá en cojer las matrices de antes de k, con las de después de k, con la multiplicación de k.

$$
matrices(i,j) \;=\; \text{mín}_{i \leq k \leq j-1} \{ matrices(i,k) + matrices(k+1,j) + d_{i-1} d_k d_j \}
$$

- Casos Básicos:
  - matrices (i, i) = 0
- Llamad inicial
  - matrices(1, n)

Tabla



El recorrido sería desde la esquina superior izquierda a la inferior derecha. En este problema no se puede mejorar el espacio adicional porque se necesitan todas las diagonales para resolver el problema.

Reconstrucción

Habría que tener una matriz paralela que guarde la posición k que dio el menor resultado. En la posición [1][n]

Código

```
// el coste es de O(n^3)
//no se necesitan las matrices sino sus dimensiones
// d[0] es el número de filas de la primera matriz
// d[1] es el número de columnas de la primera matriz y el numero de filas de la se
int multiplica_matrices(vector<int> const& D, Matriz<int>& P) {
    int n = D.size() - 1;
    Matriz<int> matrices(n + 1, n + 1, 0); P = Matriz<int>(n + 1, n + 1, 0);
    for (int d = 1; d <= n - 1; ++d)  // recorre diagonales
        for (int i = 1; i <= n - d; ++i) { // recorre elementos de diagonal
            int j = i + d;
            matrices[i][j] = INF;
            for (int k = i; k <= j - 1; ++k) {
                int temp = matrices[i][k] + matrices[k + 1][j] + D[i - 1] * D[k] * D[j];
                if (temp < matrices[i][j]) { // es mejor partir por k
                    matrices[i][j] = temp; P[i][j] = k;
                }
            }
        }
    return matrices[1][n];
}

void escribir_paren(int i, int j, Matriz<int> const& P) {
    if (i == j)
        cout << "M" << i;
    else {
        int k = P[i][j];
        if (k > i) {
            cout << "("; escribir_paren(i, k, P); cout << ")";
        }
        else cout << "M" << i;
        cout << "*";
        if (k + 1 < j) {
            cout << "("; escribir_paren(k + 1, j, P); cout << ")";
        }
        else cout << "M" << j;
    }
}
```

Justificación de un Texto

- Dadas n palabras de longitudes l1…ln distribuidas en un párrafo con líneas de longitud L.
- Si una línea tiene las palabras de la i a la j, el numero de espacios extra es

$$L-(j-i)-\sum_{k=i}^{j} l_k.$$

- Añadir dichos espacios tiene la siguiente penalización:

$$penaliza(i,j)=\left(L-(j-i)-\sum_{k=i}^{j} l_k\right)^3$$

Solución 1

- La solución podría pasar por la pregunta ¿que hacer con cada palabra? o se pone (si cabe) o la pasamos a la siguiente línea.
- parrado(i, j) = penalización mínima al formatear las palabras de la i a la n empezando en una línea con j espacios libres