Semana 11.- Algoritmos voraces (II)

Problema de la mochila versión entera

- Hay n objetos, cada uno con un peso pi>0 y un valor vi>0.
- La mochila soporta un peso total máximo M>0
- El problema consiste en maximizar el valor con la restricción de que la suma de los pesos debe ser menos que M.

Aspectos relevantes

- En la solución no importa el orden en el que los objetos son introducidos.
- Las soluciones dependen de los objetos que tengamos disponibles para introducir en la mochila y el peso que esta soporte.
- Definimos la siguiente función:
    - mochila(i,j) = máximo valor que podemos poner en una mochila de peso máximo j considerando los objetos del 1 al i.
- Tiene que cumplir el principio de optimalidad de Bellman.

Resolución

- Casos recursivos:

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

con $1 \leq i \leq n$ y $1 \leq j \leq M$

- Casos básicos:

$$\begin{aligned} mochila(0,j) &= 0 & 0 \leq j \leq M \\ mochila(i,0) &= 0 & 0 \leq i \leq n \end{aligned}$$

La solución pasará por resolver el problema primero cogiendo el objeto si cabe y siguiendo la solución por ese camino, o no cogiéndolo y siguiendo por ese (Algoritmo de vuelta atrás).

Para reconstruir la solución:

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

Funcionamiento de la reconstrucción:

- **Caso base**:
  - Si el subproblema ya fue resuelto (`mochila[i][j] != -1`), simplemente retorna su valor almacenado, evitando recalcularlo.
  - Si `i == 0` (sin objetos) o `j == 0` (capacidad de la mochila es 0), no se puede obtener ningún valor. Por tanto, `mochila[i][j] = 0`.
- **Exclusión del objeto actual**:
  - Si el peso del objeto actual (`obj[i-1].peso`) es mayor que la capacidad restante de la mochila (`j`), el objeto no puede incluirse. Entonces esto significa que el máximo valor lo determinan únicamente los `i-1` objetos restantes.

    ```
    cpp
    Copiar código
    ```

```
mochila[i][j] = mochila_rec(obj, i-1, j, mochila);
```

- **Incluir o excluir el objeto actual**:
  - Si el objeto cabe en la mochila (`obj[i-1].peso <= j`), se considera incluirlo o no:
- Aquí se evalúan dos escenarios:
  - No incluir el objeto: El valor máximo es el obtenido con los primeros `i-1` objetos y la misma capacidad `j`.
  - Incluir el objeto: Se suma el valor del objeto actual (`obj[i-1].valor`) al máximo valor obtenido con capacidad restante `j - obj[i-1].peso` y considerando los `i-1` objetos restantes.

Guardar el resultado:

- Después de calcular el máximo valor para el subproblema actual, se guarda en `mochila[i][j]` para usarlo en futuros cálculos.

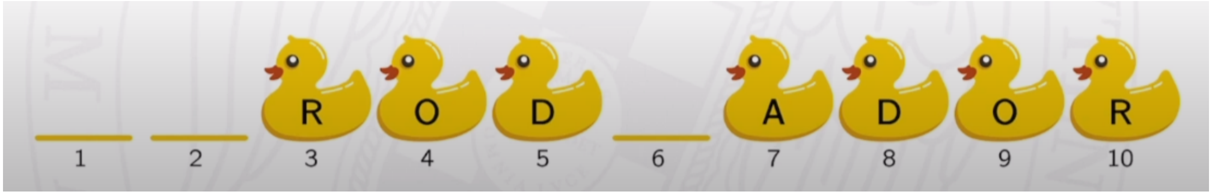
Complejidad

- **Temporal**: O(n×W), donde n es el número de objetos y W es la capacidad de la mochila.
- **Espacial**: O(n×W) por la matriz de memoización.

Tiro al patíndromo

Tenemos un número de patos cada uno con una letra:

El objetivo es conseguir el palíndromo más largos tirando alguno de los patos si es necesario:



Resolución

Al principio tenemos dos opciones:

1. Quitar la primera letra y buscar el palíndromo más largo: RODADOR.
2. Quitar la última letra y buscar el palíndromo más largo: ODADO.

Nos quedaremos con la mejor opcion: RODADOR.

Si las letras de principio y final son iguales el palíndromo deberá utilizarlas sin considerar otras opciones.

Aspectos relevantes

- patíndromo(i,j) = longitud del palíndromo más largo obtenible con patitos[i,j].

- Casos recursivos ($i < j$):

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

- Casos básicos:

$$\begin{aligned} patíndromo(i,i) &= 1 \\ patíndromo(i,j) &= 0 \quad \text{si } i > j \end{aligned}$$

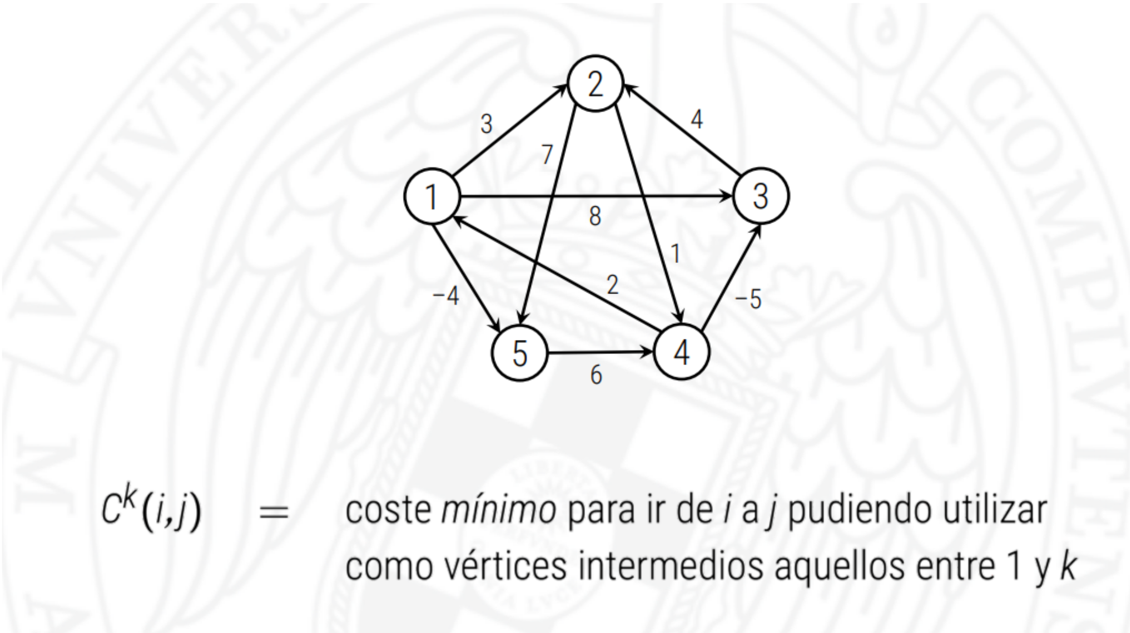
- Llamada inicial: $patíndromo(0, n-1)$

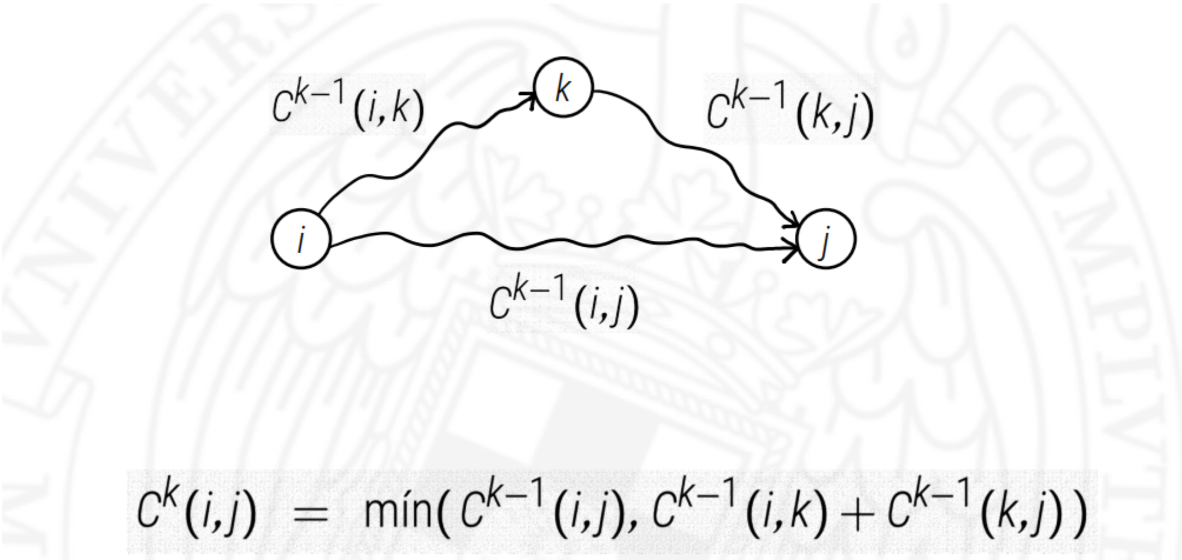
La tabla que necesitamos es nxn.

Camino mínimo entre par de vertices

- Dado un digrafo valorado, calcular el camino de coste mínimo entre cada par de vertíces.

- Si los pesos son positivos y el grafo disperso utilizando Dijkstra: O(VAlogV).
- Con Floyd y pesos negativos: O(V^3).





En este caso necesitaremos de una tabla tridimensional

Complejidad

Para la reconstrucción O(V^3) siendo V el número de vértices.

Cuestionario preguntas

- Cuando resolvemos el algoritmo de Floyd podemos ir rellenando la matriz de cualquier forma.
- El algoritmo de Floyd se puede utilizar en aristas con coste negativo siempre y cuando no haya ciclos de coste negativo.
- La diagonal principal siempre vale 0.
- En el algoritmo de la mochila con enteros la capacidad de la mochila influye en el coste.