# PRÁCTICA 5:
# PROGRAMACIÓN DINÁMICA


***Integrantes del grupo:***

*Ana López Mohedano*
*Adrián Anguita Muñoz*
*Marina Jun Carranza Sánchez*
*Pedro Antonio Mayorgas Parejo*
*Rafael Jiménez Márquez*

# ÍNDICE



---

# 0. Aclaraciones previas

Para la resolución de problemas propuestos, se va a llevar a cabo el análisis de los siguientes algoritmos de exploración de grafos, cada cual se dividirá en dos apartados:

- **Diseño de componentes**: incluye el diseño de resolución por etapas, la ecuación recurrente, el diseño de la memoria y la verificación del P.O.B.
- **Diseño de algoritmos**: el algoritmo del cálculo de coste óptimo y el de la recuperación de la solución.
- **Ejecución del código**: los resultados por pantalla de ejecutar los algoritmos de solución.

**Pautas para probar el código con makefile:**

Se incluye una tarea “all” para compilar y probar todos los algoritmos, y también una orden de make de compilación y de ejecución para cada uno de ellos, por ejemplo: “make pX_compile” y “make pX_test”, siendo X el número del problema.

# 1. Problema de los viajes en canoa

## a) Diseño de componentes

### Resolución por etapas

El problema se puede dividir en varias etapas. En cada etapa se considera el costo de viajar desde una aldea *i* hasta una aldea *j*, utilizando cualquier número de transbordos intermedios en otras aldeas *k*.

### Ecuación recurrente

Partiendo de los siguientes **elementos**:

- ➢ `C(i,j)`: el costo mínimo conocido para viajar de la aldea i a la j, hasta el momento de la iteración actual.
- ➢ `C(i,k)`: el costo mínimo para viajar de la aldea i a una aldea intermedia k.
- ➢ `C(k,j)`: el costo mínimo para viajar desde la aldea intermedia k a la aldea j.

$$\mathbf{C(i,j) = mín_{i=1\ldots n}\{C(i,j),C(i,k)+C(k,j)\}} \text{ para todo k tal que } i<k<j$$

Por tanto en cada etapa i, hay dos posibles **decisiones**:

- ➢ Quedarnos con el camino más corto conocido hasta el momento, el que viene representado por `C(i,j)`.
- ➢ Considerar pasar por una aldea intermedia *k* que minimice el costo, optando por el recorrido `C(i,k)+C(k,j)`.

**Casos base**: Cuando i=j, el costo es 0 porque no se requiere viajar.
**Valor objetivo**: encontrar el costo mínimo `C(i,j)` para todos los pares (i,j) donde i<j.

### Diseño de la memoria

Para resolver este problema, se utilizarán dos **matrices**:

- ➢ **`dp[][]`**: almacena el costo mínimo de viajar de *i* a *j*. Es una matriz triangular superior derecha porque no se puede ir contra la corriente del río (restricción i < j).
  - **Filas/columnas**: tiene n filas y columnas, siendo n el número de aldeas.
  - **Celdas**: cada celda `dp[i][j]` contiene el costo mínimo para viajar de la aldea *i* a la aldea *j*. Si i=j, el costo es 0 ya que no se requiere viaje.
  - **Inicialización**: toma los valores de los costos directos proporcionados en la matriz del enunciado del problema.
  - **Actualización**: se va actualizando cada celda considerando todos los puntos intermedios posibles *k* para ver si dicho viaje ofrece un costo menor

- ➢ **`next[][]`**: almacena el índice del último punto intermedio *k* utilizado para encontrar el camino óptimo de *i* a *j*. Ayuda en la recuperación del camino óptimo.
  - **Filas/columnas**: tiene n (número de aldeas) filas y columnas

- **Celdas**: cada celda `next[i][j]` contiene el índice de la aldea intermedia *k* que fue usada para lograr el costo mínimo de *i* a *j*. Si el valor es -1, significa que el camino más barato es el directo.
- **Inicialización**: con el valor -1 en todas sus celdas.
- **Actualización**: al actualizar `dp[i][j]`, si se encuentra un nuevo costo mínimo mediante el uso de un punto intermedio *k*, `next[i][j]` se actualiza al valor de *k*. Esto permite reconstruir posteriormente el camino completo, al ir buscando recursivamente los puntos intermedios hasta llegar al valor de -1.

### Verificación del P.O.B.

Según el ***Principio de Optimalidad de Bellman***, si una ruta de *i* a *j* a través de *k* es óptima, entonces los subcaminos de *i* a *k* y *k* a *j* también deben ser óptimos.

En este algoritmo de programación dinámica, esta propiedad se verifica en cada paso de actualización de la matriz `dp`. Cuando encontramos que `dp[i][j]` puede ser reducido por `dp[i][k]+dp[k][j]`, estamos implícitamente afirmando que `dp[i][k]` y `dp[k][j]` ya representan los costos mínimos para esos segmentos. Es más, si hubiera una manera más barata de viajar de *i* a *k* o de *k* a *j*, esos costos menores ya estarían reflejados en `dp`.

## b) Diseño de los algoritmos

### Algoritmo de cálculo de coste óptimo

Inspirado en el algoritmo de ***Floyd-Warshall***, un método de programación dinámica que encuentra las distancias más cortas entre todos los pares de nodos en un grafo ponderado. Funciona inicializando una matriz de distancias con las conexiones directas entre nodos y luego actualizándola iterativamente considerando cada nodo como un punto intermedio potencial para mejorar las distancias actuales. Con una eficiencia de **$O(n^3)$**, es útil para aplicaciones que requieren la evaluación de rutas mínimas en redes densas.

Para cada *i* solo se consideran los *j* tal que *j* > *i* (debido a la restricción de la corriente del río y que no tiene sentido viajar de *i* a *i*), lo que se refleja en el i+1 del tercer bucle for.

### Pseudocódigo

```
Función calculaCosteMín(dp, next, n)
    Para cada aldea k desde 0 hasta n-1, hacer:
        Para cada fila i desde 0 hasta n-1, hacer:
            Para cada columna j desde i+1 hasta n-1, hacer:
                dp(i,j) = mín{dp(i,j), dp(i,k)+dp(k,j)}
                Actualizar next si se va por k
            Fin-Para
        Fin-Para
    Fin-Para
Fin-Función
```

### Algoritmo de recuperación de la solución

Esta función auxiliar devuelve la ruta (aldeas recorridas) de coste mínimo desde i hasta j, recorriendo recursivamente los valores de la matriz `next`, rellenada en la anterior función.

```
Función construirRutas(i, j, next)
    Si next[i][j] == -1, hacer:
        Devolver {i,j}
    Fin-Si

    intermedia = next[i][j]
    ruta = construirRutas(i, intermedia, next)
    Elimina el último elemento de "ruta" para evitar duplicados
    subruta = construirRutas(intermedia, j, next)
    Añadir subruta a la ruta
    Devolver ruta
Fin-Función
```

La siguiente función imprime la solución; es decir, el costo mínimo y su camino correspondiente para ir desde cualquier *i* hasta cualquier *j* que esté a favor de la corriente.

```
Función imprimirSolución(dp, next, n)
    Para cada fila i desde 0 hasta n-1, hacer:
        Para cada columna j desde 0 hasta n-1, hacer:
            Si i < j, hacer:
                Imprimir "Desde i hasta j":
                Si dp[i][j] == ∞, hacer:
                    Imprimir "No hay rutas disponibles"
                En otro caso, hacer:
                    ruta = construirRutas(i, j, next)
                    costo = dp[i][j]
                    Imprimir "costo" y "ruta"
                Fin-Si
            Fin-Si
        Fin-Para
    Fin-Para
Fin-Función
```

## c) Ejecución del código

Teniendo la siguiente matriz de costos iniciales del enunciado del ejercicio:

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 3 | 3 | ∞ | ∞ |
| 2 | - | 0 | 4 | 7 | ∞ |
| 3 | - | - | 0 | 2 | 3 |
| 4 | - | - | - | 0 | 2 |
| 5 | - | - | - | - | 0 |

La ejecución del algoritmo de solución muestra el siguiente resultado por pantalla:

```
executing the dynamic programming algorithm P1
./p1_canoa.bin
Rutas y costo mínimo de los viajes en canoa:
Desde 1 hasta 2: Costo: 3 | Camino: 1 -> 2
Desde 1 hasta 3: Costo: 3 | Camino: 1 -> 3
Desde 1 hasta 4: Costo: 5 | Camino: 1 -> 3 -> 4
Desde 1 hasta 5: Costo: 6 | Camino: 1 -> 3 -> 5
Desde 2 hasta 3: Costo: 4 | Camino: 2 -> 3
Desde 2 hasta 4: Costo: 6 | Camino: 2 -> 3 -> 4
Desde 2 hasta 5: Costo: 7 | Camino: 2 -> 3 -> 5
Desde 3 hasta 4: Costo: 2 | Camino: 3 -> 4
Desde 3 hasta 5: Costo: 3 | Camino: 3 -> 5
```

Y los valores finales de las dos matrices de memoria, `dp` y `next`, son los siguientes:

- **"-"**: equivale a ∞ e indica que no hay camino posible desde *i* hasta *j*.
- **"*"**: equivale a -1 e indica una conexión directa de *i* a *j*, sin viajes intermedios *k*.

```
Matriz de costos mínimos (DP)
0 3 3 5 6
- 0 4 6 7
- - 0 2 3
- - - 0 2
- - - - 0
Matriz de viajes intermedios (next)
* * * 3 3
* * * 3 3
* * * * *
* * * * *
* * * * *
```

# 2. Problema de los viajes aéreos de empresa

## a) Diseño de componentes

### Resolución por etapas

El problema se puede dividir en varias etapas. En cada etapa se considera el costo de viajar de una ciudad i a una ciudad j, utilizando cualquier número de escalas en otros aeropuertos k. En este ejercicio nos basaremos en el pseudocódigo e implementación del anterior para resolverlo, ya que es muy similar pero con distintas restricciones. La ecuación recurrente tendrá una modificación, pero el diseño de la memoria y la recuperación de la solución serán los mismos.

### Ecuación recurrente

Partiendo de los siguientes **elementos**:

- ➢ `C(i,j)`: el costo mínimo conocido para viajar de la ciudad i a la j, hasta el momento de la iteración actual.
- ➢ `C(i,k)`: el costo mínimo para viajar de la ciudad i a una ciudad intermedia k.
- ➢ `C(k,j)`: el costo mínimo para viajar desde la ciudad intermedia k a la ciudad j.
- ➢ E(k): tiempo de escala adicional en la ciudad k, en este ejercicio suponemos E(k) = 1 ∀k.

$$\mathbf{C(i,j) = mín_{i=1\ldots n}\{C(i,j),C(i,k)+C(k,j) + E(k)\}}$$ para todo k tal que `i<k<j`

Por tanto en cada etapa i, hay dos posibles **decisiones**:

- ➢ Quedarnos con el camino más corto conocido hasta el momento, el que viene representado por `C(i,j)`.
- ➢ Considerar pasar por una ciudad intermedia *k* que minimice el costo, optando por el recorrido `C(i,k)+C(k,j)+E(k)`.

**Casos base**: Cuando i=j, el costo es 0 porque no se requiere viajar.
**Valor objetivo**: encontrar el costo mínimo `C(i,j)` para todos los pares (i,j).

### Diseño de la memoria

Para resolver este problema, se utilizarán dos **matrices**:

- ➢ **`dp[][]`**: almacena el costo mínimo de viajar de *i* a *j*.
  - **Filas/columnas**: tiene n filas y columnas, siendo n el número de aldeas.
  - **Celdas**: cada celda `dp[i][j]` contiene el costo mínimo para viajar de la aldea *i* a la aldea *j*. Si i=j, el costo es 0 ya que no se requiere viaje.
  - **Inicialización**: toma los valores de los costos directos proporcionados en la matriz del enunciado del problema.
  - **Actualización**: se va actualizando cada celda considerando todos los puntos intermedios posibles *k* para ver si dicho viaje ofrece un costo menor.

- ➢ **`next[][]`**: almacena el índice del último punto intermedio *k* utilizado para encontrar el camino óptimo de *i* a *j*. Ayuda en la recuperación del camino óptimo.
  - **Filas/columnas**: tiene n (número de ciudades) filas y columnas
  - **Celdas**: cada celda `next[i][j]` contiene el índice de la ciudad intermedia *k* que fue usada para lograr el costo mínimo de *i* a *j*. Si el valor es -1, significa que el camino más barato es el directo.
  - **Inicialización**: con el valor -1 en todas sus celdas.
  - **Actualización**: al actualizar `dp[i][j]`, si se encuentra un nuevo costo mínimo mediante el uso de un punto intermedio *k*, `next[i][j]` se actualiza al valor de *k*. Esto permite reconstruir posteriormente el camino completo, al ir buscando recursivamente los puntos intermedios hasta llegar al valor de -1.

### Verificación del P.O.B.

Según el ***Principio de Optimalidad de Bellman***, si una ruta de *i* a *j* a través de *k* es óptima, entonces los subcaminos de *i* a *k* y *k* a *j* también deben ser óptimos.

En este algoritmo de programación dinámica, esta propiedad se verifica en cada paso de actualización de la matriz `dp`. Cuando encontramos que `dp[i][j]` puede ser reducido por `dp[i][k]+dp[k][j]`, estamos implícitamente afirmando que `dp[i][k]` y `dp[k][j]` ya representan los costos mínimos para esos segmentos. Es más, si hubiera una manera más barata de viajar de *i* a *k* o de *k* a *j*, esos costos menores ya estarían reflejados en `dp`.

## b) Diseño de los algoritmos

### Algoritmo de cálculo de coste óptimo

Inspirado en el algoritmo de ***Floyd-Warshall***, un método de programación dinámica que encuentra las distancias más cortas entre todos los pares de nodos en un grafo ponderado. Funciona inicializando una matriz de distancias con las conexiones directas entre nodos y luego actualizándola iterativamente considerando cada nodo como un punto intermedio potencial para mejorar las distancias actuales. Con una eficiencia de **$O(n^3)$**, es útil para aplicaciones que requieren la evaluación de rutas mínimas en redes densas.

### Pseudocódigo

```
Función calculaCosteMín(dp, next, n)
    Para cada aldea k desde 0 hasta n-1, hacer:
        Para cada fila i desde 0 hasta n-1, hacer:
            Para cada columna j desde 0 hasta n-1, hacer:
                dp(i,j) = mín{dp(i,j), dp(i,k)+dp(k,j)+E(k)}
                Actualizar next si se va por k
            Fin-Para
        Fin-Para
    Fin-Para
Fin-Función
```

### Algoritmo de recuperación de la solución

Esta función auxiliar devuelve la ruta (ciudades recorridas) de coste mínimo desde i hasta j, recorriendo recursivamente los valores de la matriz `next`, rellenada en la anterior función.

```
Función construirRutas(i, j, next)
    Si next[i][j] == -1, hacer:
        Devolver {i,j}
    Fin-Si

    intermedia = next[i][j]
    ruta = construirRutas(i, intermedia, next)
    Elimina el último elemento de “ruta” para evitar duplicados
    subruta = construirRutas(intermedia, j, next)
    Añadir subruta a la ruta
    Devolver ruta
Fin-Función
```

La siguiente función imprime la solución; es decir, el costo mínimo y su camino correspondiente para ir desde cualquier *i* hasta cualquier *j*.

```
Función imprimirSolución(dp, next, n)
    Para cada fila i desde 0 hasta n-1, hacer:
        Para cada columna j desde 0 hasta n-1, hacer:
            Si i != j, hacer:
                Imprimir “Desde i hasta j”:
                Si dp[i][j] == ∞, hacer:
                    Imprimir “No hay rutas disponibles”
                En otro caso, hacer:
                    ruta = construirRutas(i, j, next)
                    costo = dp[i][j]
                    Imprimir “costo” y “ruta”
                Fin-Si
            Fin-Si
        Fin-Para
    Fin-Para
Fin-Función
```

## c) Ejecución del código

Teniendo la siguiente matriz de costos iniciales del enunciado del ejercicio:

| $T[i,j]$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 2 | 1 | 3 |
| 2 | 7 | 0 | 9 | 2 |
| 3 | 2 | 2 | 0 | 1 |
| 4 | 3 | 4 | 8 | 0 |

La ejecución del algoritmo de solución muestra el siguiente resultado por pantalla:

```
Rutas y costo mínimo de los viajes en avión:
Desde 1 hasta 2: Costo: 2 | Camino: 1 -> 2
Desde 1 hasta 3: Costo: 1 | Camino: 1 -> 3
Desde 1 hasta 4: Costo: 3 | Camino: 1 -> 3 -> 4
Desde 2 hasta 1: Costo: 6 | Camino: 2 -> 4 -> 1
Desde 2 hasta 3: Costo: 8 | Camino: 2 -> 4 -> 1 -> 3
Desde 2 hasta 4: Costo: 2 | Camino: 2 -> 4
Desde 3 hasta 1: Costo: 2 | Camino: 3 -> 1
Desde 3 hasta 2: Costo: 2 | Camino: 3 -> 2
Desde 3 hasta 4: Costo: 1 | Camino: 3 -> 4
Desde 4 hasta 1: Costo: 3 | Camino: 4 -> 1
Desde 4 hasta 2: Costo: 4 | Camino: 4 -> 2
Desde 4 hasta 3: Costo: 5 | Camino: 4 -> 1 -> 3
```

Como en el ejercicio anterior, mostrará los caminos mínimos desde cada ciudad a cualquier otra ciudad, a diferencia del anterior, que solo mostraba desde una aldea hasta otra aldea que estuviese río abajo.

Y los valores finales de las dos matrices de memoria, `dp` y `next`, son los siguientes:

- **"-"**: equivale a ∞ e indica que no hay camino posible desde *i* hasta *j*.
- **"*"**: equivale a -1 e indica una conexión directa de *i* a *j*, sin viajes intermedios *k*.

```
Matriz de costos mínimos (DP)
0 2 1 3
6 0 8 2
2 2 0 1
3 4 5 0
Matriz de viajes intermedios (next)
* * * 3
4 * 4 *
* * * *
* * 1 *
```

En este ejemplo todas las ciudades pueden volar a cualquier otra, por lo que no tendremos ningún "-".

# 3. Problema de las bolsas de dinero

## a) Diseño de componentes

### Resolución por etapas

El problema se puede dividir en varias etapas. En cada etapa se considera el costo de moverse desde una casilla *i, j* hasta otra casilla i, *j*, permitiéndose sólo movimientos a la izquierda, abajo y izquierda-abajo (en diagonal) y obteniendo la mayor cantidad de dinero.

### Ecuación recurrente

Partiendo de los siguientes elementos:

-Fil sera el numero de filas

-Col será el número de columnas

-**M[i][j]** será el número de bolsas de dinero en la casilla (i,j), si la casilla no es transitable será -1

-**T[i][j]** será la cantidad máxima de dinero que podemos recolectar al llegar a la casilla (i,j)

Para cada casilla (i,j) la cantidad máxima de oro T[i][j] la podemos calcular considerando las 3 casillas desde las que podemos llegar a la casilla actual.

**T[i][j]** = **M[i][j]** + max(**T[i - 1][j]**, **T[i][j + 1]**, **T[i - 1][j + 1]**)

Por lo tanto para cada etapa hay 3 posibles decisiones:

**T[i - 1][j]:** corresponde al movimiento que hacemos desde arriba.

**T[i][j + 1]**: corresponde al movimiento que hacemos desde la derecha.

**T[i - 1][j + 1]**: corresponde al movimiento que hacemos desde la esquina superior derecha es decir el movimiento en diagonal.

**Casos Base:**

-Cuando el jugador está en la casilla de salida (0,Col - 1) la cantidad máxima de dinero que puede recolectar es simplemente el número de bolsas de esa casilla.

T[i][Col - 1] = M[i][Col - 1]

-Si i = 0 y j > 0:

T[0][j] = M[0][j] + T[0][j - 1]

-Si j = 0 y i > 0:

T[i][0] = M[i][0] + T[i-1][0]

**Valor objetivo:** El número máximo de dinero que puedes obtener desde la entrada a la salida.

## Diseño de la memoria

Matriz costes[ ][ ]

- **Filas / Columnas:** tendrá FIL filas y COL columnas las cuales son las filas y columnas del laberinto.
- **Celdas:** cada celda costes[i][j] contiene la cantidad máxima de oro que se puede recolectar al llegar a la casilla (i, j), si la casilla no es transitable el valor será -1.
- **Inicialización:**Se inicializa con los valores de dinero en cada casilla del laberinto proporcionados por la matriz laberinto[ ][ ].
- **Actualización:**Se actualiza considerando los movimientos posibles(anterior arriba, derecha o diagonal arriba) para calcular la cantidad máxima de dinero recolectable al llegar a cada casilla

Vector camino ya que recuperaremos el camino directamente desde la matriz costes.

Vector camino

- **Elementos:** contiene pares de (i,j) que representan las coordenadas de las casillas que forman el camino óptimo de recolección de dinero.
- **Inicialización:** Se inicializa en la casilla inicial (FIL - 1, 0) que esto representa la salida del laberinto ya que vamos a recuperar el camino hacia atrás.
- **Actualización:** Se actualiza mientras se recorre el camino desde la casilla final a la inicial utilizando la matriz costes para determinar la siguiente casilla en el camino óptimo.

## Verificación del P.O.B.

El ***Principio de Optimalidad de Bellman*** establece que una solución óptima global se puede construir a partir de subsoluciones óptimas. Para confirmar que el algoritmo cumple con el principio de optimalidad debemos verificar que cada subproblema es óptimo y estas soluciones forman una solución óptima.

**-Subproblemas óptimos:**
La cantidad de dinero recolectada al llegar a una casilla (i,j) depende solo de las cantidades máximas de dinero recolectadas en las anteriores. Esto significa que para llegar a (i,j) de manera óptima las decisiones tomadas en las casillas anteriores deben ser óptimas.

**-Solución global óptima**:
Al rellenar la matriz de coste[ ] [ ] el algoritmo nos asegura que en cada casilla se encuentra la cantidad máxima de dinero recolectable hasta esa casilla. Al recuperar el camino el algoritmo usa las decisiones óptimas para reconstruir el camino desde la salida hasta la entrada.

En conclusión el algoritmo si cumple con el ***Principio de Optimalidad de Bellman*** garantizando que la solución global obtenida es la óptima.

# b) Diseño de los algoritmos

## Algoritmo de cálculo de coste óptimo

Este algoritmo rellena la matriz de costes con la cantidad máxima de dinero que puedes obtener al llegar a una determinada casilla y rellena las casillas no transitables con su determinado coste.

### Pseudocódigo

```
Funcion rellenarMatrizCostes(laberinto, costes)

    Para i desde 0 hasta FIL - 1 hacer
        Para j desde FIL - 1 hasta 0 hacer
            Si laberinto[i][j] == -1 entonces
                costes[i][j] = -1
            sino
                Si laberinto[i][j] == 1 entonces
                    oro_actual = 1
                sino
                    oro_actual = 0
                Fin si

                Si i == 0 y j == COL - 1 entonces
                    costes[i][j] = oro_actual
                sino
                    max_oro_prev = -1

                    Si i > 0 y costes[i-1][j] != -1 entonces  // Anterior arriba
                        max_oro_prev = maximo(max_oro_prev, costes[i-1][j])
                    Fin si

                    Si j < COL - 1 y costes[i][j+1] != -1 entonces  // Derecha
                        max_oro_prev = maximo(max_oro_prev, costes[i][j+1])
                    Fin si

                    Si i > 0 y j < COL - 1 y costes[i-1][j+1] != -1 entonces  // Diagonal arriba
                        max_oro_prev = maximo(max_oro_prev, costes[i-1][j+1])
                    Fin si

                    Si max_oro_prev != -1 entonces
                        costes[i][j] = oro_actual + max_oro_prev
                    Fin si
                Fin si
            Fin si
        Fin para
    Fin para

    devolver costes
Fin funcion
```

## Algoritmo de recuperación de la solución

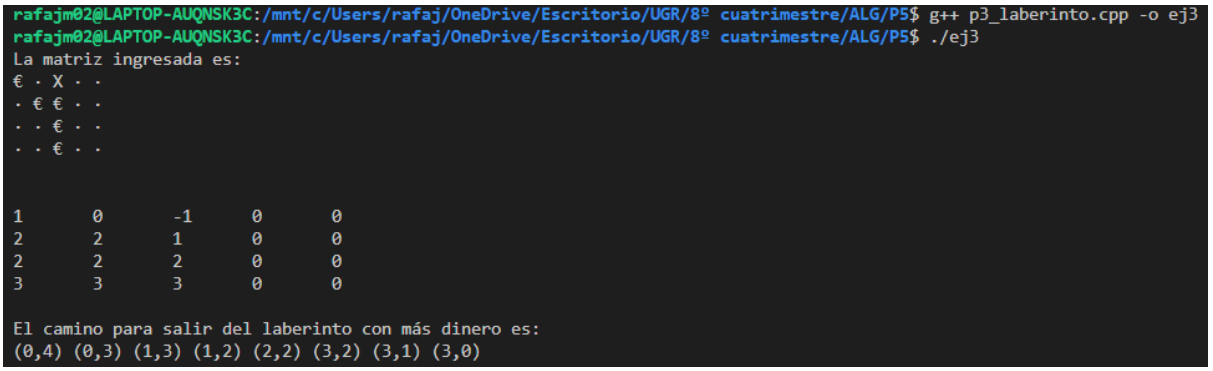
Este algoritmo lo que hace es recuperar el camino óptimo es decir recupera el camino por el cual se puede obtener el máximo de dinero recorriendo el laberinto desde la entrada hasta la salida.

```
Funcion recuperarCamino(laberinto, DP, camino)
    i = FIL - 1
    j = 0
    agregar (i, j) a camino

    mientras i > 0 o j < COL - 1 hacer
        max_oro = -1
        siguiente_casilla = null

        si i > 0 y DP[i-1][j] != -1 y DP[i-1][j] > max_oro entonces
            max_oro = DP[i-1][j]
            siguiente_casilla = (i-1, j)
        Fin si

        si j < COL - 1 y DP[i][j+1] != -1 y DP[i][j+1] > max_oro entonces
            max_oro = DP[i][j+1]
            siguiente_casilla = (i, j+1)
        Fin si

        si i > 0 y j < COL - 1 y DP[i-1][j+1] != -1 y DP[i-1][j+1] > max_oro entonces
            max_oro = DP[i-1][j+1]
            siguiente_casilla = (i-1, j+1)
        Fin si

        agregar siguiente_casilla a camino
        i = siguiente_casilla.primero
        j = siguiente_casilla.segundo
    Fin mientras

    devolver camino
Fin funcion
```

## c) Ejecución del código

Representaremos con un signo de € las casillas donde se encuentran las bolsas de dinero y con una “X” las casillas donde se encuentran los muros y las casillas con “.” lo que representa son las casillas transitables sin dinero y sin muro.

Luego en la matriz de abajo iremos indicando la máxima cantidad de bolsas de dinero con las que podemos llegar a cierta casilla.



# 4. Problema de Pixel Mountain

## a) Diseño de componentes

### Resolución por etapas

El problema se puede dividir en varias etapas. En cada etapa se considera la dificultad de moverse desde una casilla *i, j* de la fila inferior hasta otra casilla i, *j*, de la fila superior permitiéndose sólo movimientos arriba-izquierda (diagonal), arriba y arriba-derecha (diagonal) siempre que no nos salgamos del rocódromo y minimizando el esfuerzo hasta llegar a la fila superior.

### Ecuación recurrente

Partiendo de los siguientes elementos:

-Fil sera el numero de filas

-Col será el número de columnas

-**M[i][j]** será la dificultad (coste) que existe para llegar a esa casilla.

-**T[i][j]** será la dificultad mínima que requerirá llegar a la casilla (i,j)

Para cada casilla (i,j) la dificultad mínima T[i][j] la podemos calcular considerando la suma mínima de la dificultad de las casillas desde las que podemos llegar a la casilla actual.

**T[i][j]** = **M[i][j]** + min(**T[i + 1][j -1]**, **T[i +1][j]**, **T[i + 1][j + 1]**)

Por lo tanto para cada etapa hay 3 posibles decisiones:

**T[i + 1][j -1]:** corresponde al movimiento que hacemos hacia arriba a la izquierda.

**T[i +1][j]**: corresponde al movimiento que hacemos hacia arriba.

**T[i + 1][j + 1]**: corresponde al movimiento que hacemos hacia arriba a la derecha.

**Casos Base:**

El jugador empezará en una casilla de la última fila, empezando por la esquina inferior izquierda. Las dificultades de esa primera fila coincidirán con las dificultades iniciales de esas casillas, es decir, T[FIL-1][j] = M[FIL-1][j].

**Valor objetivo:** La dificultad mínima desde la base (última fila) hasta la cima (primera fila).

## Diseño de la memoria

Tendremos dos matrices y un vector de pares para almacenar la solución:

Matriz montaña[ ][ ]

- **Filas / Columnas:** tendrá FIL filas y COL columnas las cuales son las filas y columnas de la montaña.
- **Celdas:** cada celda montaña[i][j] contiene la dificultad requerida para llegar a la casilla(i, j) desde una casilla adyacente de la fila inferior.
- **Inicialización:** Se inicializa con valores de dificultades al azar para cada casilla de la montaña.

Matriz dificultad[ ][ ]

- **Filas / Columnas:** tendrá FIL filas y COL columnas las cuales son las filas y columnas de la montaña.
- **Celdas:** cada celda costes[i][j] contiene la dificultad mínima al llegar a la casilla(i, j).
- **Inicialización:** Se inicializa con valor 0 todas las casillas exceptuando las de la última fila que coincidirá con el valor que contiene la última fila de la matriz montaña.
- **Actualización:** Se actualiza considerando los movimientos posibles (superior arriba, diagonal superior izquierda y diagonal superior derecha) para calcular la dificultad mínima al llegar a cada casilla.

Vector camino ya que recuperaremos el camino directamente desde la matriz costes.

Vector camino

- **Elementos:** contiene pares de (i,j) que representan las coordenadas de las casillas que forman el camino óptimo para alcanzar la cima de la montaña.
- **Inicialización:** Se inicializa con una casilla de la primera fila (0, j), concretamente la que tenga el mínimo valor. Esto representa la cima de la montaña ya que vamos a recuperar el camino hacia arriba y hacia abajo.
- **Actualización:** Se actualiza mientras se recorre el camino desde la casilla final hasta una casilla de la última fila utilizando la matriz costes para determinar la siguiente casilla en el camino óptimo.

## Verificación del P.O.B.

El ***POB*** nos dice que una solución óptima global se puede construir a partir de subsoluciones óptimas. Para confirmar que el algoritmo cumple con el principio de optimalidad debemos verificar que cada subproblema es óptimo y estas soluciones forman una solución óptima.

**-Subproblemas óptimos:**
La dificultad requerida para llegar a una casilla (i,j) depende solo de las dificultades requeridas en las anteriores etapas. Esto significa que para llegar a (i,j) de manera óptima las decisiones tomadas en las casillas anteriores deben ser óptimas.

**-Solución global óptima**: Al rellenar la matriz de coste[ ][ ], el algoritmo nos asegura que en cada casilla está la dificultad mínima requerida hasta llegar a esa casilla. Al recuperar el camino el algoritmo usa las decisiones óptimas para reconstruir el camino desde la entrada hasta la salida.

Como conclusión, el algoritmo si cumple con el ***Principio de Optimalidad de Bellman*** garantizando que la solución global obtenida es la óptima.

# b) Diseño de los algoritmos

## Algoritmo de cálculo de coste óptimo

El algoritmo rellena la matriz de costes con la dificultad mínima requerida para llegar a una determinada casilla y rellena la última fila que con el valor que contiene la última fila de la matriz montaña.

### Pseudocódigo

```
Funcion rellenaPixelMountain(montania, dificultad):
    // Rellenamos la última fila
    para j desde 0 hasta COL - 1 hacer
        dificultad[FIL - 1][j] = montania[FIL - 1][j]
    Fin para

    // Rellenar la matriz de costos desde la penúltima fila hasta la primera
    para i desde FIL - 2 hasta 0 hacer
        para j desde 0 hasta COL - 1 hacer
            minDif = dificultad[i + 1][j]

            si j > 0 entonces
                minDif = minimo(minDif, dificultad[i + 1][j - 1])
            Fin si

            si j < COL - 1 entonces
                minDif = minimo(minDif, dificultad[i + 1][j + 1])
            Fin si

            dificultad[i][j] = montania[i][j] + minDif
        Fin para
    Fin para

    devolver dificultad
Fin funcion
```

## Algoritmo de recuperación de la solución

Este algoritmo recupera el camino óptimo, es decir, recupera el camino por el cual se puede escalar la montaña con la mínima dificultad Para ello, recorre la matriz de costes desde la cima de la montaña hasta la base siendo el primer valor el mínimo de las casillas de la primera fila(cima), continuando hacia abajo.

### Pseudocódigo

```
Funcion recuperaCamino(montania, dificultad, camino):

    minCoste = dificultad[0][0]
    posMin = 0

    // Encontrar la posición con el costo mínimo en la primera fila
    para j desde 0 hasta COL - 1 hacer
        si dificultad[0][j] < minCoste entonces
            minCoste = dificultad[0][j]
            posMin = j
        Fin si
    Fin para

    // Añadir la posición inicial al camino
    agregar (0, posMin) a camino

    // Recuperar el camino desde la segunda fila hasta la última
    para i desde 1 hasta FIL - 1 hacer
        sigMin = posMin
        minCoste = dificultad[i][posMin]

        si posMin > 0 y dificultad[i][posMin - 1] < minCoste entonces
            minCoste = dificultad[i][posMin - 1]
            sigMin = posMin - 1
        Fin si

        si posMin < COL - 1 y dificultad[i][posMin + 1] < minCoste entonces
            minCoste = dificultad[i][posMin + 1]
            sigMin = posMin + 1
        Fin si

        posMin = sigMin
        agregar (i, posMin) a camino
    Fin para

    devolver camino
Fin funcion
```

## c) Ejecución del código

```
rafajm02@LAPTOP-AUQNSK3C:/mnt/c/Users/rafaj/OneDrive/Escritorio/UGR/8º cuatrimestre/ALG/P5$ g++ p4_pixelmountain.cpp -o ej4
rafajm02@LAPTOP-AUQNSK3C:/mnt/c/Users/rafaj/OneDrive/Escritorio/UGR/8º cuatrimestre/ALG/P5$ ./ej4
La matriz ingresada es:
2 8 9 5 8
4 4 6 2 3
5 7 5 6 1
3 2 5 4 8


13      19      16      12      15
11      11      13      7       8
7       9       7       10      5
3       2       5       4       8

El camino para escalar la montaña con la mínima dificultad es:
(3,3) (2,4) (1,3) (0,3)
```