PRÁCTICA 5

PROGRAMACIÓN DINÁMICA

UNIVERSIDAD
DE GRANADA

Marino Fernández Pérez,
Mehdi Iabouten,
Jorge López Molina,
Gabriele Ruggeri
y Pablo Vega Romero

Problema 1: Viaje en canoa

-> Diseño de resolución por etapas

El problema de encontrar el costo mínimo para viajar en canoa por el río hasta recorrer las ciudades se puede resolver por etapas.
Cada una de las etapas corresponde con viajar a la siguiente ciudad, es decir, suponemos que nos encontramos en una casilla (i, j) correspondiente a una ciudad y buscamos el costo mínimo para llegar a esa casilla desde la ciudad anterior (teniendo en cuenta de que si partimos de una ciudad i a una ciudad j, el punto i será menor que j). Para la resolución de cada etapa, nos podemos mover hacia abajo, en función de las ciudades con las que se conecte aquella desde la que partimos, siempre y cuando las posiciones sean válidas, es decir, que el río conecte dichas ciudades(el coste de una ciudad consigo misma es 0).

-> Diseño de la Ecuación Recurrente

La solución depende de las etapas (las filas del mapa, denotadas como i) y de las conexiones con las distintas ciudades . Definimos el valor de coste mínimo para llegar a la casilla (i, j) como min_coste(i, j).

En cada etapa/casilla, tenemos varias opciones en función del número de ciudades que tengamos y de si la ciudad en la que estamos se conecta con alguna de ellas o no. Por tanto que la solución depende de dos factores:

- Los nodos intermedios que se incluyan como camino a la solución.
- Los pares de nodos para los cuales calculamos la distancia mínima.

Definimos min_coste(i,j) como la distancia mínima de la ciudad i a la ciudad j, considerando los nodos intermedios(si fuera necesario).

Dado que el problema es de minimización, podríamos expresar la **ecuación recurrente** de la siguiente manera:

d(i,j) = min(d(i,j), d(i, k) + d(k, j))

Siendo k un nodo intermedio por el que tenga que pasar si el coste es mejor que un viaje directo.

A la hora de identificar el caso base, corresponde con el caso en el que podemos calcular directamente el resultado, esto sucede cuando no se utilizan nodos intermedios por los que pasar, y el viaje es directo.
Si no se utilizan nodos intermedios:

- La distancia entre i y j es directamente el coste de viaje entre estas dos.
- min_coste[i][j] = costos[i][j]

-> Valor Objetivo

Se desean conocer todos los caminos con el menor coste posible de todas las ciudades i a todas las ciudades j.

- > Verificación del cumplimiento del P.OB

Para garantizar que la solución obtenida es óptima, debemos asegurarnos de que cualquier solución óptima está compuesta por subsoluciones óptimas. En este caso se cumple, ya que, en cada una de las etapas, se calcula el camino mínimo hacia la ciudad a la que se dirige.
El procedimiento corresponde con:

Para cada etapa:

- Se evalúan todos los posibles movimientos.
- Se escoge la opción que cuente con el menor coste posible para llegar a la ciudad deseada.

Este proceso garantiza que la solución final sea la más óptima ya que en cada paso, decidimos que camino debemos cruzar con la canoa para llegar a la ciudad correspondiente.

```
ermedi ~/Documentos/CUATRI->2/ALG/P5 ./camino
Matriz de costos mínimos:
0       3       3       5       6
INF     0       4       6       7
INF     INF     0       2       3
INF     INF     INF     0       2
INF     INF     INF     INF     0

Caminos más cortos:
Camino de 1 a 2: 1 2
Camino de 1 a 3: 1 3
Camino de 1 a 4: 1 3 4
Camino de 1 a 5: 1 3 5
Camino de 2 a 1: No hay camino
Camino de 2 a 3: 2 3
Camino de 2 a 4: 2 3 4
Camino de 2 a 5: 2 3 5
Camino de 3 a 1: No hay camino
Camino de 3 a 2: No hay camino
Camino de 3 a 4: 3 4
Camino de 3 a 5: 3 5
Camino de 4 a 1: No hay camino
Camino de 4 a 2: No hay camino
Camino de 4 a 3: No hay camino
Camino de 4 a 5: 4 5
Camino de 5 a 1: No hay camino
Camino de 5 a 2: No hay camino
Camino de 5 a 3: No hay camino
Camino de 5 a 4: No hay camino
```

Como vemos en esta ejecución, para cada ciudad i encuentra el camino(si es que existe) con coste mínimo hacia una ciudad j.

Hay que tener en cuenta que aquellas ciudades destino que tienen un identificador menor a las ciudades origen(por ejemplo, si vamos de la ciudad 5 a la 1), no tienen un camino, ya que no se puede remar en contracorriente del río.

-> Diseño de la memoria

- Se representan los posibles resultados con 2 matrices.
- En una matriz contiene las distancias mínimas entre todos los pares de nodos, es decir, guarda el valor que representa la distancia más corta desde una ciudad i a una ciudad j.
- En otra matriz se representan los siguientes nodos para llegar al camino más corto por cada pareja de nodos. Para verlo más claro:
  -Si next[i][j] es k(nodo intermedio), esto significa que para ir desde una ciudad i hacia una ciudad j a través del río, k es uno de los nodos por el que hay que pasar antes de llegar a j.
- Estas matrices se rellenan de la siguiente forma:
  - Inicialización:
    - Cada casilla de la matriz min_coste(min_coste[i][j]) se rellena con la matriz que pasamos como argumento (matriz denominada costos). Si hay un coste ya, min_coste[i][j] se iguala a costos[i][j], en caso contrario, se iguala a infinito.
    - Cada casilla de la matriz next se inicializa con j(quiere decir que hay un camino hacia la ciudad siguiente) y se queda con un -1 en esa posición si no hay camino.
  - Actualización:
    A lo largo del proceso de actualización se actualizan las matrices en cada iteración, donde se considera cada nodo k como un posible nodo intermedio para encontrar el camino más corto en cada par de ciudades (i,j).
    Se comprueba si el coste de i a j pasando por k es menor que el coste de ir directamente sin pasar por ninguna otra ciudad intermedia, en caso de que se cumpla, se actualizan los valores de las matrices.
- Una vez se hayan abarcado todas las iteraciones para todos los posibles trayectos de coste mínimo en canoa el programa termina.

Finalmente, y con este diseño, construimos el algoritmo de Programación Dinámica como sigue:

```
function min_coste(costos):
    tam = tamaño de costos
    INF = un valor muy grande que representa "infinito"

    // Inicialización de las matrices min_coste y next
    min_coste = matriz de tamaño tam x tam, inicializada con INF
    next = matriz de tamaño tam x tam, inicializada con -1

    for i desde 0 hasta tam - 1:
        for j desde 0 hasta tam - 1:
            if costos[i][j] != INF:
                min_coste[i][j] = costos[i][j]
                next[i][j] = j  // Inicialmente, el siguiente nodo en el camino de i a j es j
            end if
        end for
    end for

    for k desde 0 hasta tam - 1:
        for i desde 0 hasta tam - 1:
            for j desde 0 hasta tam - 1:
                if min_coste[i][k] != INF y min_coste[k][j] != INF y
                   min_coste[i][k] + min_coste[k][j] < min_coste[i][j]:
                    min_coste[i][j] = min_coste[i][k] + min_coste[k][j]
                    next[i][j] = next[i][k]  // Actualiza el siguiente nodo en el camino
                end if
            end for
        end for
    end for

    return (min_coste, next)
end function
```

-> Recuperación de la solución:

```
function reconstruir_camino(i, j, next):
    if next[i][j] == -1:
        return []  // No hay camino
    end if

    camino = [i]
    while i != j:
        i = next[i][j]
        camino.append(i)
    end while

    return camino
end function
```

Esta función se encarga de reconstruir y devolver los caminos dada una pareja de ciudades i y j, y la matriz "next" donde guardabamos los recorridos de una ciudad a otra.

-> Implementación final de los algoritmos

```
pair<vector<vector<int>>, vector<vector<int>>> min_coste(const vector<vector<int>>& costos) {
    int tam = costos.size();
    vector<vector<int>> min_coste(tam, vector<int>(tam, INF));
    vector<vector<int>> next(tam, vector<int>(tam, -1));

    for (int i = 0; i < tam; ++i) {
        for (int j = 0; j < tam; ++j) {
            if (costos[i][j] != INF) {
                min_coste[i][j] = costos[i][j];
                next[i][j] = j;  // Inicialmente, el siguiente nodo en el camino de i a j es j
            }
        }
    }

    for (int k = 0; k < tam; ++k) {
        for (int i = 0; i < tam; ++i) {
            for (int j = 0; j < tam; ++j) {
                if (min_coste[i][k] != INF && min_coste[k][j] != INF &&
                    min_coste[i][k] + min_coste[k][j] < min_coste[i][j]) {
                    min_coste[i][j] = min_coste[i][k] + min_coste[k][j];
                    next[i][j] = next[i][k];
                }
            }
        }
    }

    return {min_coste, next};
}
```

```
vector<int> reconstruir_camino(int i, int j, const vector<vector<int>>& next) {
    if (next[i][j] == -1) {
        return {};  // No hay camino
    }
    vector<int> camino = {i};
    while (i != j) {
        i = next[i][j];
        camino.push_back(i);
    }
    return camino;
}
```

```
int main() {
    vector<vector<int>> costos = {
        {0, 3, 3, INF, INF},
        {INF, 0, 4, 7, INF},
        {INF, INF, 0, 2, 3},
        {INF, INF, INF, 0, 2},
        {INF, INF, INF, INF, 0}
    };

    // Calcular la matriz de costos mínimos y la matriz de caminos
    auto [min_cost, next] = min_coste(costos);

    // Imprimir la matriz de costos mínimos
    cout << "Matriz de costos mínimos:" << endl;
    for (const auto& row : min_cost) {
        for (int cost : row) {
            if (cost == INT_MAX) {
                cout << "INF " << "\t";
            } else {
                cout << cost << "\t";
            }
        }
        cout << endl;
    }
```

```

    // Imprimir los caminos más cortos para cada par de aldeas
    cout << "\nCaminos más cortos:" << endl;
    for (int i = 0; i < next.size(); ++i) {
        for (int j = 0; j < next.size(); ++j) {
            if (i != j) {
                cout << "Camino de " << i + 1 << " a " << j + 1 << ": ";
                vector<int> camino = reconstruir_camino(i, j, next);
                if (camino.empty()) {
                    cout << "No hay camino" << endl;
                } else {
                    for (int nodo : camino) {
                        cout << nodo + 1 << " ";
                    }
                    cout << endl;
                }
            }
        }
    }

    return 0;
}
```

Versión que no guarda el recorrido:

```cpp
vector<vector<int>> min_coste(const vector<vector<int>>& costos) {
    int tam = costos.size();
    vector<vector<int>> min_coste(tam, vector<int>(tam, INT_MAX));

    for (int i=0 ; i<tam ; i++)
    {
        for (int j=i ; j<tam ; j++)
            min_coste[i][j] = costos[i][j];
    }

    for (int k = 0; k < tam; k++) {
        for (int i = 0; i < tam; i++) {
            for (int j = i + 1; j < tam; j++) {
                if (min_coste[i][k] != INT_MAX && min_coste[k][j] != INT_MAX) {
                    min_coste[i][j] = min(min_coste[i][j], min_coste[i][k] + min_coste[k][j]);
                }
            }
        }
    }

    return min_coste;
}
```

```cpp
int main() {
    // Matriz de costos proporcionada en el problema
    vector<vector<int>> costes = {
        {0, 3, 3, INT_MAX, INT_MAX},
        {INT_MAX, 0, 4, 7, INT_MAX},
        {INT_MAX, INT_MAX, 0, 2, 3},
        {INT_MAX, INT_MAX, INT_MAX, 0, 2},
        {INT_MAX, INT_MAX, INT_MAX, INT_MAX, 0}
    };

    // Calcular la matriz de costos mínimos
    vector<vector<int>> min_cost = min_coste(costes);

    // Imprimir la matriz de costos mínimos
    for (const auto& row : min_cost) {
        for (int cost : row) {
            if (cost == INT_MAX) {
                cout << "INF " << "\t";
            } else {
                cout << cost << "\t";
            }
        }
        cout << endl;
    }

    return 0;
}
```

Problema 2: Planificación de Viajes Aéreos

-> Diseño de resolución por etapas

Este problema se puede subdividir en etapas. Se saca un trayecto que aún no ha llegado al destino y, a partir de la ciudad en la que estemos, volará a la ciudad destino o tratará de hacer trasbordo en alguna de las 2 ciudades posibles en cada iteración. Esto es posible ya que dividimos en 3 contenedores distintos. En uno tenemos los trayectos inacabados pero pendientes de ser estudiados, en otro tenemos aquellos que pese a no haber llegado al destino ya no merece la pena seguir estudiando por acumular más coste del que supondría el propio viaje directo, por último guardamos las soluciones que cuentan con posibilidades de ser la solución del problema y recibirán un estudio exclusivo.

-> Diseño de la Ecuación Recurrente

La solución dependerá de la ciudad en la que comenzamos (llamada ini en la función) y de la ciudad destino (llamada fin). Además también dependerá de cuánto sea el coste del viaje directo entre estas dado que toda solución más costosa se dejará de analizar aunque no se haya llegado a la solución.

La ecuación dice que debemos minimizar costes luego se comparará el coste generado por los transbordos que se hayan realizado con el coste del viaje directo:

```
min(matriz[ini][fin], matriz[ini][k] + matriz[k][fin] + 1)
```

Finalmente sumamos 1 porque en el enunciado se nos especifica que hacer trasbordo implica un retraso de 1. Pese a parecer contraintuitivo en ocasiones resulta más beneficioso visitar todas las ciudades en un orden concreto que ir directamente del punto inicial al punto final.

El caso base para este problema sería aquel en el que el viaje más corto posible es el viaje directo por ello no se estudiaría ninguna otra solución posible. Esto ocurre gracias a una condición que expresa que el coste acumulado de la secuencia tiene que ser mejor que el coste del viaje directo para que siga desarrollándose esa secuencia.

-> Valor Objetivo

Se desean conocer las ciudades a visitar para que el coste de llegar de una ciudad inicial a una ciudad final sea mínimo.

- > Verificación del cumplimiento del P.OB

Cualquier solución óptima debe estar formada por subsoluciones óptimas. En este caso se verifica y lo voy a demostrar con un ejemplo:

```
va desde:1 hasta: 2
el coste mejor es: 8
el coste directo es: 9
pasa por 1, pasa por 3, pasa por 0, pasa por 2,
```

Para llegar de la ciudad 1 a la ciudad 2 vemos que la solución óptima sería pasando por las ciudades 3 y 0. El principio de bellman se cumple ya que el subtrayecto óptimo que une las ciudades 3 y 2 también es el 3 o 2 como se demuestra en la siguiente ejecución:

```
va desde:3 hasta: 2
el coste mejor es: 5
el coste directo es: 8
pasa por 3, pasa por 0, pasa por 2,
```

Esto se debe a que la distancia de mínimo coste que une las ciudades 3 y 2 es la misma siempre sin importar si viene de la ciudad 1 o comienza en la 3. (También se cumple para todos los pares de ciudades posibles, este es el ejemplo más claro posible).

-> Diseño de la memoria

- Se representan los posibles resultados con 3 matrices y un diccionario map.
- En una matriz se ponen las secuencias pendientes de estudiar.
- En otra matriz se ponen las secuencias que ya no serán estudiadas por no ser candidatas a óptimas.
- En la última matriz se ponen aquellas secuencias que han llegado a la solución con menor coste que el viaje directo.
- En el diccionario map se ponen las secuencias asociadas al coste que tienen para que si se precisa saber el coste de una secuencia ya calculada no tenga que volver a llamarse al método que calcula el coste de una secuencia. Bastaría con buscar el coste en el diccionario usando la secuencia como llave. En el caso de que no esté registrada, se llama al método que calcula el coste y se añade al diccionario.
- Estos contenedores son rellenados de la siguiente forma
    - se saca una secuencia de la matriz de secuencias pendientes.
    - Se le calculan los posibles vuelos desde la ciudad
    - Se devuelve a la matriz de pendientes aquellos que no hayan llegado al final pero se deban seguir estudiando.
    - Se pone a la matriz soluciones aquella que implique llegar a la solución y se postula como solución posible.
    - Si no compensa seguir esa secuencia se pone en la matriz de secuencias despreciables.
- Una vez no queden secuencias por estudiar pasamos a estudiar las secuencias de la matriz de soluciones posibles y se devuelve la secuencia de menor coste de esta matriz.

Finalmente el algoritmo que hemos planteado seguiría la siguiente estructura:

```
function caminoMasCorto(vector<vector<int>> matriz, int ini, int fin):
    coste_directo = coste del viaje directo entre la ciudad ini y la ciudad fin
    soluciones_parciales = matriz que almacena las soluciones que aun estan pendientes de estudio
    soluciones_desechadas = matriz de secuencias que no merece la pena seguir estudiando
    soluciones = matriz de secuencias posibles mejores soluciones
    secuencia = vector con el que vamos a manipular
    coste = variable entera con el que vamos a trabajar

    costes_ya_calculados = diccionario que almacena secuencias y su coste para evitar calcular lo mismo varias veces

    //inicializamos souciones_parciales, soluciones y costes_ya_calculados
    //añadiendo el viaje directo a soluciones y la primera ciudad a soluciones parciales

    while (soluciones_parciales tenga elementos):
        // sacamos una secuencia de soluciones parciales
        secuencia = soluciones_parciales.back();
        soluciones_parciales.pop_back();

        for i desde 0 hasta numero de filas de la matriz:
            if i distinto de la misma ciudad en la que estamos y distinto de la ciudad inicio:
                //recordar que la ciudad en la que nos encontramos es secuencia.back()

                añadimos i a la secuencia

                if no se habia calculado el coste de esa secuencia antes:
                    añadimos la secuencia y el coste al diccionario costes_ya_calculados
                else
                    lo sacamos del diccionario a partir de la llave secuencia
                end if

                if no pertenece a las soluciones desechadas:
                    if coste acumulado < coste que representa el viaje directo:
                        if i es la ciudad destino:
                            añadimos la secuencia a soluciones
                        else
                            añadimos la secuencia a soluciones_paricales y se la estudiará más tarde
                        end if
                    else
                        lo añadimos a soluciones_desechadas ya que por coste no merece la pena seguir analizandola
                    end if
                end if
            end if
        end for
    end while
```

```
    //fase de obtencion de la solución óptima entre las posibles calculadas

    secuencia_mejor = vector que albergará la mejor secuencia
    coste_mejor = variable entera que almacenará el coste de la secuencia mejor (no se pide pero por añadir información al resultado)

    for i = 0 hasta recorrer todas las soluciones:
        coste_secuencia = variable que almacenará el coste de la secuencia que obtendremos del diccionario costes_ya_calculados

        if es menor que el actualmente coste menor:
            actualizamos el valor de coste mejor y de secuencia menor
        end if
    end for

    mostramos por pantalla el coste del viaje mejor
    mostramos por pantalla el coste del viaje directo

    devolvemos secuencia_mejor
```

-> Implementación final de los algoritmos

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <map>

using namespace std;

int costeDeUnaSecuencia(vector<vector<int>> matriz, vector<int> secuencia){

    int solucion = 0;

    for (int i = 1 ; i < secuencia.size() ; i++){
        solucion += matriz[secuencia[i-1]][secuencia[i]];
    }

    if (secuencia.size() >= 2)
        solucion += secuencia.size() - 2; // por el coste de las escalas

    return solucion;
}

bool contenida(vector<vector<int>> soluciones, vector<int> secuencia){

    for (int i = 0 ; i < soluciones.size() ; i++){
        if (soluciones[i] == secuencia){
            return true;
        }
    }
    return false;
}
```

```cpp
vector<int> caminoMasCorto(vector<vector<int>> matriz, int ini, int fin) {
    int coste_directo = matriz[ini][fin];
    vector<vector<int>> soluciones_parciales;
    vector<vector<int>> soluciones_deshechadas;
    vector<vector<int>> soluciones;
    vector<int> secuencia;
    int coste;

    map<vector<int>,int> costes_ya_calculados;

    secuencia.push_back(ini);
    soluciones_parciales.push_back(secuencia);

    secuencia.push_back(fin);
    costes_ya_calculados[secuencia] = matriz[ini][fin];

    soluciones.push_back(secuencia);
```

```
51      while (!soluciones_parciales.empty())
52      {
53          secuencia = soluciones_parciales.back();
54
55          soluciones_parciales.pop_back();
56
57          for (int i = 0 ; i < matriz.size() ; i++){
58              if (i != secuencia.back() && i != ini){
59
60                  secuencia.push_back(i);
61
62                  if (costes_ya_calculados.find(secuencia) == costes_ya_calculados.end()){
63                      coste = costeDeUnaSecuencia(matriz,secuencia);
64                      costes_ya_calculados[secuencia] = coste;
65                  }
66                  else
67                  {
68                      coste = costes_ya_calculados[secuencia];
69                  }
70
71
72                  if (!contenida(soluciones_deshechadas,secuencia)){
73                      if (coste <= coste_directo){
74                          if (i == fin){
75                              soluciones.push_back(secuencia);
76                          }
77                          else
78                          {
79                              soluciones_parciales.push_back(secuencia);
80                          }
81
82                          secuencia.pop_back();
83                      }
84                      else
85                      {
86                          soluciones_deshechadas.push_back(secuencia);
87                      }
88                  }
```

```
93      vector<int> secuencia_mejor;
94      int coste_mejor = coste_directo;
95
96      for (int i = 0 ; i < soluciones.size(); i++){
97          int coste_secuencia = costes_ya_calculados[soluciones[i]];
98
99          if (coste_secuencia <= coste_mejor){
100             secuencia_mejor = soluciones[i];
101             coste_mejor = coste_secuencia;
102         }
103     }
104
105     cout << "el coste mejor es: " << costes_ya_calculados[secuencia_mejor] << endl;
106     cout << "el coste directo es: " << matriz[ini][fin] << endl;
107     return secuencia_mejor;
108 }
```

```cpp
int main(){

    vector<vector<int>> matriz;

    matriz.push_back({0,2,1,3});
    matriz.push_back({7,0,9,2});
    matriz.push_back({2,2,0,1});
    matriz.push_back({3,4,8,0});

    for (int i = 0 ; i < 4 ; i++){
        for (int j = 0 ; j < 4 ; j++){

            cout << "va desde:" << i << " hasta: " << j << endl;

            vector<int> solucion = caminoMasCorto(matriz,i,j);

            for (int i = 0 ; i < solucion.size() ; i++){
                cout << "pasa por " << solucion[i] << ", ";
            }

            cout << endl;
            cout << endl;
            cout << endl;
            cout << endl;
        }
    }
}

```

Problema 3: Máximo número de bolsas de oro

-> Diseño de resolución por etapas

El problema de encontrar el camino con el mayor número de bolsas de oro se puede resolver por etapas. Para ello, dividimos el recorrido hasta la meta en ellas.
Cada una de las etapas corresponde con bajar un nivel de nuestro mapa, es decir, obviando las columnas, suponemos que nos encontramos en una casilla (i, j) del mapa y buscamos el número de bolsas de oro máximo para llegar a esa casilla desde la fila superior(tomando como punto de partida la esquina superior derecha del mapa). Para la resolución de cada etapa, nos podemos mover hacia abajo, en diagonal hacia la izquierda, o en diagonal hacia la izquierda-abajo, siempre y cuando las posiciones sean válidas.

-> Diseño de la Ecuación Recurrente

La solución depende de las etapas (las filas del mapa, denotadas como i) y de la posición en la fila (las columnas del mapa , denotadas como j). Definimos el valor de coste mínimo para llegar a la casilla (i, j) como coste(i, j).

En cada etapa/casilla, tenemos tres opciones:

- moverse hacia abajo desde la casilla superior(i+1, j)
- moverse diagonalmente hacia la izquierda desde la casilla superior(i+1, j-1)
- moverse hacia la izquierda desde la casilla a la derecha(i, j-1).

Con estas tres decisiones posibles, y dado que el problema es de maximización, tendríamos que podríamos expresar la **ecuación recurrente** como:

dp(i,j) = oro + max({abajo, izquierda, abajoIzquierda});

A la hora de identificar el caso base y por definición para este algoritmo, corresponde con el caso en el que podemos calcular directamente el resultado sin tener que dividirlo en subproblemas. En este caso, cuando nos encontramos en la casilla de salida.

-> Valor Objetivo

Se desea conocer el máximo número de bolsas de oro que se pueden acumular llegando a la meta.

- > Verificación del cumplimiento del P.OB

Para garantizar que la solución obtenida es óptima, debemos asegurarnos de que cualquier solución óptima está compuesta por subsoluciones óptimas. En este caso se cumple, ya que, en cada una de las etapas, el máximo número de bolsas de oro que se puede acumular, probando todos los posibles movimientos.

El procedimiento corresponde con:

Para cada etapa:

- Se evalúan todos los posibles movimientos
- Se escoge la opción que cuente con el número de bolsas oro máximo hasta el momento

Este proceso garantiza que la solución final sea la más óptima ya que en cada paso, decidimos que movimiento nos da el número de bolsas de oro más grande hasta el momento.

-> Diseño de la memoria

- Para resolver el problema, dp(i, j) se representará como una matriz.
- Cada celda de la matriz dp(i, j) contendrá el máximo oro acumulado para llegar a la casilla (i, j)
- La memoria se rellenará de la siguiente forma:
    - La memoria (en este caso, la matriz dp) se rellenará de manera recursiva a medida que se llame a la función resolver. Aquí está el proceso paso a paso:

    - Cuando se llama a resolver(i, j), primero se verifica si la posición (i, j) está fuera del mapa o si es un muro. Si es así, se devuelve 0 y no se realiza ninguna actualización en dp.

    - Luego, se verifica si ya se ha calculado el resultado para la posición (i, j). Si es así, se devuelve el resultado almacenado en dp[i][j] y no se realiza ninguna actualización en dp.

    - Si no se ha calculado el resultado para la posición (i, j), se inicializa oro a 0 y luego se verifica si hay oro en la posición (i, j). Si es así, se cambia oro a 1.

Finalmente, y con este diseño, construimos el algoritmo de Programación Dinámica como sigue:

```
Función resolver(i, j):
    Comprobar si la posición (i, j) está fuera del mapa o si es un muro:
        Si es así, devolver 0.
    Comprobar si ya hemos calculado el resultado para la posición (i, j):
        Si es así, devolver el resultado almacenado en 'dp[i][j]'.
    Inicializar 'oro' a 0.
    Comprobar si hay oro en la posición (i, j):
        Si es así, cambiar 'oro' a 1.
    Llamar a la función resolver para la posición debajo de la actual (i + 1, j) y almacenar el resultado en 'abajo'.
    Llamar a la función resolver para la posición a la izquierda de la actual (i, j - 1) y almacenar el resultado en 'izquierda'.
    Llamar a la función resolver para la posición en diagonal abajo a la izquierda de la actual (i + 1, j - 1) y almacenar el resultado en 'abajoIzquierda'.
    Calcular la cantidad máxima de oro que se puede recoger desde la posición (i, j), que es 'oro' más el máximo de 'abajo', 'izquierda' y 'abajoIzquierda'.
    Almacenar este resultado en 'dp[i][j]'.
    Devolver 'dp[i][j]'.

Función principal:
    Llamar a la función resolver con la posición inicial (0, 3) y almacenar el resultado en 'resultado'.
    Imprimir 'resultado'.
    Devolver 0 para indicar que el programa ha terminado correctamente.
```

Para poder llevar a cabo la recuperación de la solución, es decir, el camino a seguir con el objetivo de la salida con el máximo número de oro, tenemos el siguiente algoritmo.

Recorremos la matriz direcciones desde la posición inicial (i, j) hasta que se sale del mapa. En cada paso, añade la posición actual a la lista pasos y luego actualiza (i, j) según la dirección almacenada en direcciones[i][j]. Al final, devuelve la lista de pasos, que contiene la secuencia de posiciones que llevan a la solución.

```
Función recuperarPasos(i, j):
    Crear una lista vacía 'pasos'.
    Mientras que 'i' y 'j' están dentro del mapa:
        Añadir '(i, j)' a 'pasos'.
        Si 'direcciones[i][j]' es 'abajo', incrementar 'i'.
        Si no, si 'direcciones[i][j]' es 'izquierda', decrementar 'j'.
        Si no, incrementar 'i' y decrementar 'j'.
    Devolver 'pasos'.
```

-> Implementación final de los algoritmos

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

vector<vector<char>> mapa = {
    {' ', 'O', 'M', ' '},
    {'O', ' ', ' ', ' '},
    {' ', 'O', ' ', ' '},
    {' ', 'O', ' ', ' '},
    {' ', 'O', ' ', ' '},
    {'S', 'O', 'O', ' '}
};

vector<vector<int>> dp(6, vector<int>(4, -1));
vector<vector<pair<int, int>>> direcciones(6, vector<pair<int, int>>(4, {-1, -1}));

int resolver(int i, int j) {
    if (i < 0 || i >= 6 || j < 0 || j >= 4 || mapa[i][j] == 'M') {
        return 0;
    }

    if (dp[i][j] != -1) {
        return dp[i][j];
    }

    int oro = 0;
    if (mapa[i][j] == 'O') {
        oro = 1;
    }

    int abajo = resolver(i + 1, j);
    int izquierda = resolver(i, j - 1);
    int abajoIzquierda = resolver(i + 1, j - 1);

    int maxOro = max({abajo, izquierda, abajoIzquierda});

```

```cpp
    if (maxOro == abajo) {
        direcciones[i][j] = {i + 1, j};
    } else if (maxOro == izquierda) {
        direcciones[i][j] = {i, j - 1};
    } else {
        direcciones[i][j] = {i + 1, j - 1};
    }

    dp[i][j] = oro + maxOro;

    return dp[i][j];
}

vector<pair<int, int>> recuperarPasos(int i, int j) {
    vector<pair<int, int>> pasos;

    while (i >= 0 && i < 6 && j >= 0 && j < 4) {
        pasos.push_back({i, j});
        pair<int, int> siguiente = direcciones[i][j];
        i = siguiente.first;
        j = siguiente.second;
    }

    return pasos;
}

int main() {
    int resultado = resolver(0, 3);  // posición inicial en la esquina superior derech

    cout << "Número máximo de bolsas de oro: " << resultado << endl;

    vector<pair<int, int>> pasos = recuperarPasos(0, 3);
    cout << "Pasos: ";
    for (const auto& paso : pasos) {
        cout << "(" << paso.first << ", " << paso.second << ") ";
    }
    cout << endl;

    return 0;
}
```

Problema 4: Escalada con Coste Mínimo

-> Diseño de resolución por etapas

El problema de encontrar el costo mínimo para escalar una montaña hasta la cima se puede resolver por etapas. Para ello, dividimos el ascenso a la montaña en ellas.
Cada una de las etapas corresponde con subir un nivel de la montaña, es decir, obviando las columnas, suponemos que tenemos una casilla (i, j) de la montaña y buscamos el costo mínimo para llegar a esa casilla desde la fila inferior (tomando como punto de partida la base de la montaña). Para la resolución de cada etapa, nos podemos mover hacia arriba, en diagonal hacia la izquierda, o en diagonal hacia la derecha, siempre y cuando las posiciones sean válidas.

-> Diseño de la Ecuación Recurrente

La solución depende de las etapas (las filas de la montaña, denotadas como i) y de la posición en la fila (las columnas de la montaña, denotadas como j). Definimos el valor de coste mínimo para llegar a la casilla (i, j) como coste(i, j).

En cada etapa/casilla, tenemos tres opciones:

- moverse hacia arriba desde la casilla inferior (coste(i+1, j))
- moverse diagonalmente hacia la izquierda desde la casilla inferior (coste(i+1, j-1))
- moverse diagonalmente hacia la derecha desde la casilla inferior (coste(i+1, j+1)).

Con estas tres decisiones posibles, y dado que el problema es de minimización, tendríamos que podríamos expresar la **ecuación recurrente** como:

coste(i,j) = min(coste(i+1, j), coste(i+1, j-1), coste(i+1, j+1)) + montaña[i][j] ("coste_casilla" en código)

El costo mínimo para llegar a la casilla (i, j) se calcula como el costo de la casilla actual más el mínimo entre estas tres opciones.

A la hora de identificar el caso base y por definición para este algoritmo, corresponde con el caso en el que podemos calcular directamente el resultado sin tener que dividirlo en subproblemas. En este caso, corresponde con la fila de partida de la montaña, donde el costo mínimo para alcanzar cualquier casilla en esa fila será el mismo valor que se encuentra en dicha casilla.

-> Valor Objetivo

Se desea conocer el coste mínimo (“coste_minimo”) para poder llegar a la cima de la montaña.
Para ello, queremos determinar el valor coste(0, j) para cada j posible, es decir, para todos los valores de la primera fila de la matriz, que nos determinará el costo mínimo para alcanzar la cima a partir de un camino concreto.

- > Verificación del cumplimiento del P.OB

Para garantizar que la solución obtenida es óptima, debemos asegurarnos de que cualquier solución óptima esté compuesta por subsoluciones óptimas. En este caso se cumple, ya que, en cada una de las etapas, se calcula el coste mínimo para llegar a esa casilla en particular, probando todos los posibles movimientos.
El procedimiento corresponde con:

Para cada etapa:

- Se evalúan todos los posibles movimientos
- Se escoge la opción que cuente con el coste más pequeño hasta el momento

Este proceso garantiza que la solución final sea la más óptima ya que en cada paso, decidimos que movimiento nos da el coste más pequeño que permita minimizar el costo total hasta el momento.

-> Diseño de la memoria

- Para resolver el problema, coste(i, j) se representará como una matriz.
- Tendrá n filas. Cada fila i corresponderá con un nivel de la montaña
- Tendrá N columnas. Cada columna i corresponderá con una columna de la montaña
- Cada celda de la matriz coste(i, j) contendrá el mínimo coste acumulado para llegar a la casilla (i, j)
- La memoria se rellenará de la siguiente forma:
    - En primer lugar, se rellenan las celdas correspondientes al caso base mencionado anteriormente, es decir, la última fila de esta matriz con los costes de la última fila de la matriz “montaña”
    - En segundo lugar, se rellenarán las filas restantes {n-2, ..., 0}, en orden secuencial
    - Cada fila se rellenará en orden creciente de columnas {0, ... N - 1}

Finalmente, y con este diseño, construimos el algoritmo de Programación Dinámica como sigue:

```
ALGORITMO escaladaCosteMinimo(montaña):
   montaña ← matriz que representa el coste asociado a la dificultad de llegar a cada una de sus casillas

   si montaña esta vacia entonces
      devolver 0
   end si

   coste ← matriz con dimensiones idénticas a montaña donde para cada par (i, j) que represneta una casilla, se encuentra almacenado el coste mínimo para su alcance

   filas = filas de montaña
   cols = columnas de montaña

   # caso base

   para cada columna de montaña (col):
      coste(ultima fila, col) = montaña(ultima fila, col)
   end para

   para cada fila desde la penultima hasta la primera (fil):
      para cada columna desde la primera hasta la ultima (col):
         coste_casilla = coste actual en montaña

         # movimiento ascendente

         coste_movimiento = coste(fil + 1, col)

         # diag. izq

         si col es mayor que 0 entonces
            coste_movimiento = min{coste_movimiento, coste(fil + 1, col - 1)}
         end si

         si col es menor que la ultima columna de montaña entonces
            coste_movimiento = min{coste_movimiento, coste(fil + 1, col + 1)}
         end si

         coste(i, j) = montaña(i, j) + coste_movimiento
      end para
   end para

   coste_minimo = INFINITO
   para cada columna de coste (col):
      si coste(0, col) es menor que el menor coste entonces
         coste_minimo = coste(0, col);

   devolver coste_minimo
end ALGORITMO
```

Para poder llevar a cabo la recuperación de la solución, es decir, el camino a seguir con el objetivo de alcanzar la cima de la montaña con el menor coste posible, tenemos el siguiente algoritmo.

Para su implementación, cabe destacar el uso de una matriz auxiliar “camino” que nos permitirá representar lo siguiente:

- En cada casilla (i, j) almacenaremos la columna solución que supone el movimiento de coste mínimo en cada una de las etapas para llegar a (i, j)

Además, el algoritmo tendrá como parámetro por referencia un vector “path” que finalmente contará con las posiciones de las columnas que corresponden con las casillas de coste mínimo a pisar para poder llegar a la cima con el menor costo posible. Para ayudarnos en su uso, se ha añadido una variable “posi_minima”, que almacenará la posición de la columna en la que se encuentra el mínimo coste.
A la hora de rellenar “path”:

- Comenzamos desde la cima de la montaña recorriendo las filas
- Seguimos el camino de coste óptimo hacia abajo, es decir, para cada nivel, vamos avanzando por cada una de las casillas cuya posición de columna corresponde con la que consigue el coste más óptimo.

```
ALGORITMO escaladaCosteMinimoConPlan(montaña, path):
   montaña ← matriz que representa el coste asociado a la dificultad de llegar a cada una de sus casillas
   path ← vector ha modificar que representará finalmente el camino a seguir para alcanzar la cima con el menor coste

   si montaña esta vacia entonces
      devolver 0
   end si

   coste ← matriz con dimensiones idénticas a montaña donde para cada par (i, j) que represneta una casilla, se encuentra almacenado el coste
   mínimo para su alcance

   camino ← matriz con dimensiones idénticas a montaña donde para cada par(i, j) que representa una casilla se encuentra almacenada la
   columna de menor coste para llegar a dicho par

   filas = filas de montaña
   cols = columnas de montaña

   # caso base

   para cada columna de montaña (col):
      coste(ultima fila, col) = montaña(ultima fila, col)
   end para

   para cada fila desde la penultima hasta la primera (fil):
      para cada columna desde la primera hasta la ultima (col):
         coste_casilla = coste actual en montaña
         posi_min = valor indefinido

         # movimiento ascendente

         coste_movimiento = coste(fil + 1, col)
         posi_min = col

         # diag. izq

         si col es mayor que 0 entonces
            coste_movimiento = min{coste_movimiento, coste(fil + 1, col - 1)}

            si coste_movimiento es igual a coste(fil + 1, col - 1)
               posi_min = col - 1
            end si
         end si

         si col es menor que la ultima columna de montaña entonces
            coste_movimiento = min{coste_movimiento, coste(fil + 1, col + 1)}

            si coste_movimiento es igual a coste(fil + 1, col + 1)
               posi_min = col + 1
            end si
         end si

         coste(i, j) = montaña(i, j) + coste_movimiento
         camino(i, j) = posi_min
      end para
   end para
```

-> Implementación final de los algoritmos

```cpp
void printEscalada(const vector<vector<int>> &montaña, const vector<int> &camino){
   int filas = montaña.size();
   int cols = montaña[0].size();

   vector<vector<char>> mostrar(filas, vector<char> (cols, ' '));

   // copiamos literalmente la matriz mmontaña

   for (int i = 0; i < filas; i++){
      for (int j = 0; j < cols; j++){
         mostrar[i][j] = montaña[i][j] + '0';
      }
   }

   //pintamos el camino

   for (int i = 0; i < camino.size(); i++){
      mostrar[i][camino[i]] = 'X';
   }

   // mostramos la matriz

   for (int i = 0; i < filas; i++){
      for (int j = 0; j < cols; j++){
         cout << mostrar[i][j] << " ";
      }

      cout << endl;
   }
}
```

```cpp
int escaladaCosteMinimo(const vector<vector<int>> &montaña){
    if (montaña.size() == 0) {
        return 0;
    }

    // encontrar la representacion para almacenar subsoluciones y por lo tanto,
    // no trabajar con valores repetidos

    // en cada una de las casillas (i, j) de la matriz se almacenara el coste minimo
    // para alcanzar dicha casilla

    vector<vector<int>> coste(montaña.size(), vector<int>(montaña[0].size(), INT_MAX));

    int filas = montaña.size();
    int cols = montaña[0].size();

    // identificacion del caso base:
    // caso en el que podemos calcular directamente el resultado sin tener que hacer
    // subproblemas. en este caso, el caso base corresponde con la fila de partida
    // donde para obtener el costo minimo basta con recorrer todos los elementos de
    // esta y añadir a la matriz coste

    for (int col = 0; col < cols; col++)
        coste[filas - 1][col] = montaña[filas - 1][col];

        // el coste para llegar a cualquiera de las casillas en la fila de partida
        // es el que tienen dado en un primer momento ya que, al no tener filas anteriores
        // a ella, el coste es el que es y ya.


    // movimientos posibles:
    // hacia arriba desde abajo [i-1][j]
    // diagonal izq [i-1][j-1]
    // diagonal dcha [i-1][j+1]

```

```cpp
    for (int fil = filas - 2; fil >= 0;  fil--){
        for (int col = 0; col < cols; col++){
            int coste_casilla = montaña[fil][col];

            // comprobamos el coste para cada uno de los movimientos posibles:

            // movimiento ascendente:

            int coste_movimiento = coste[fil + 1][col];

            // diagonal izquierda

            if (col > 0)
                coste_movimiento = min(coste_movimiento, coste[fil + 1][col - 1]);

            // diagonal derecha

            if (col < cols - 1)
                coste_movimiento = min(coste_movimiento, coste[fil + 1][col + 1]);

            // asignamos el menor coste

            coste[fil][col] = coste_casilla + coste_movimiento;
        }
    }

    int coste_minimo = INT_MAX;

    for (int col = 0; col < cols; col++){
        if (coste[0][col] < coste_minimo)
            coste_minimo = coste[0][col];
    }

    return coste_minimo;
}
```

```cpp
int escaladaCosteMinimoConPlan(const vector<vector<int>> &montaña, vector<int> &path){
    if (montaña.size() == 0) {
        return 0;
    }

    // encontrar la representacion para almacenar subsoluciones y por lo tanto,
    // no trabajar con valores repetidos

    // en cada una de las casillas (i, j) de la matriz se almacenara el coste minimo
    // para alcanzar dicha casilla

    vector<vector<int>> coste(montaña.size(), vector<int>(montaña[0].size(), INT_MAX));

    // en cada casilla de la matriz almacenamos la columna de menor coste para llegar
    // a la casilla (i, j ) de la matriz

    vector<vector<int>> camino(montaña.size(), vector<int>(montaña[0].size(), -1));

    int filas = montaña.size();
    int cols = montaña[0].size();

    // identificacion del caso base:
    // caso en el que podemos calcular directamente el resultado sin tener que hacer
    // subproblemas. en este caso, el caso base corresponde con la fila de partida
    // donde para obtener el costo minimo basta con recorrer todos los elementos de
    // esta y añadir a la matriz coste

    for (int col = 0; col < cols; col++)
        coste[filas - 1][col] = montaña[filas - 1][col];

        // el coste para llegar a cualquiera de las casillas en la fila de partida
        // es el que tienen dado en un primer momento ya que, al no tener filas anteriores
        // a ella, el coste es el que es y ya.


    // movimientos posibles:
    // hacia arriba desde abajo [i-1][j]
    // diagonal izq [i-1][j-1]
    // diagonal dcha [i-1][j+1]
```

```cpp
    for (int fil = filas - 2; fil >= 0;  fil--){
        for (int col = 0; col < cols; col++){
            int coste_casilla = montaña[fil][col];
            int posi_min = -1;

            // comprobamos el coste para cada uno de los movimientos posibles:

            // movimiento ascendente:

            int coste_movimiento = coste[fil + 1][col];
            posi_min = col;

            // diagonal izquierda

            if (col > 0){
                coste_movimiento = min(coste_movimiento, coste[fil + 1][col - 1]);
                if (coste_movimiento == coste[fil + 1][col - 1]) posi_min = col - 1;
            }

            // diagonal derecha

            if (col < cols - 1){
                coste_movimiento = min(coste_movimiento, coste[fil + 1][col + 1]);
                if (coste_movimiento == coste[fil + 1][col + 1]) posi_min = col + 1;
            }

            // asignamos el menor coste

            coste[fil][col] = coste_casilla + coste_movimiento;
            camino[fil][col] = posi_min;
        }
    }

    int coste_minimo = INT_MAX;
    int posi_minima = -1;

    for (int col = 0; col < cols; col++){
        if (coste[0][col] < coste_minimo){
            coste_minimo = coste[0][col];
            posi_minima = col;
        }
    }

    // recuperamos el plan
    // comenzamos insertando desde la cima los valores

    for (int i = 0; i < filas; i++){
        path.push_back(posi_minima);
        posi_minima = camino[i][posi_minima];
    }

    return coste_minimo;
}
```