

GRADO EN INGENIERÍA INFORMÁTICA

CURSO 2023-2024



UNIVERSIDAD DE GRANADA

ALGORÍTMICA

Práctica 5 - Programación Dinámica

Miguel Martinez Azor

Ángel Rodríguez Faya

Alejandro Botaro Crespo

Alberto Parejo Bellido

Alejandro Ocaña Sánchez

28/05/2024

ÍNDICE

Problema 1	2
1.1.-Diseño de resolución por etapas.	2
1.2.-Ecuación recurrente.	2
1.3.-Valor objetivo.	3
1.4.-Verificación del POB.	3
1.5.-Diseño de la memoria.	4
1.6.-Diseño del algoritmo de cálculo de coste óptimo.	5
1.7.-Diseño del algoritmo de recuperación de la solución.	5
Problema 2	7
2.1.-Diseño de resolución por etapas.	7
2.2.-Ecuación recurrente.	7
2.3.-Valor objetivo.	8
2.4.-Verificación del POB.	8
2.5.-Diseño de la memoria.	8
2.6.-Diseño del algoritmo de cálculo de coste óptimo.	9
2.7.-Diseño del algoritmo de recuperación de la solución.	10
2.8.-Implementación de los algoritmos de cálculo de coste óptimo y recuperación de la solución.	10
2.10.-Ejemplo de ejecución.	12
Problema 3	13
3.1.-Diseño de resolución por etapas.	14
3.2.-Ecuación recurrente.	14
3.3.-Valor objetivo.	14
3.4.-Verificación del POB.	14
3.5.-Diseño de la memoria.	14
3.6.-Diseño del algoritmo de cálculo de coste óptimo.	15
3.7.-Diseño del algoritmo de recuperación de la solución.	16
3.8.-Ejemplo de ejecución.	17
Problema 4	18
4.1.-Descripción del problema.	18
4.2.-Diseño de resolución por etapas.	18
4.3.-Ecuación recurrente.	19
4.4.-Valor objetivo.	19
4.5.-Verificación del POB.	20
4.6.-Diseño de la memoria.	20
4.7.-Diseño del algoritmo de cálculo de coste óptimo.	20
4.8.-Diseño del algoritmo de recuperación de la solución.	21
4.9.-Implementación de los algoritmos de cálculo de coste óptimo y recuperación de la solución.	23
4.10.-Ejemplo de ejecución.	25

Problema 1

A lo largo de un río hay n aldeas. En cada aldea se puede alquilar una canoa para viajar a otras aldeas que están a favor de la corriente (resulta casi imposible remar a contracorriente). Para todo posible punto de partida i y para todo posible punto de llegada y más abajo en el río ($i < j$), se conoce el coste de alquilar una canoa para ir desde i hasta j , $c(i,j)$ (si ese trayecto concreto no existe, entonces $c(i,j)=\text{infinito}$). Sin embargo, puede ocurrir que el coste del alquiler desde i hasta j sea mayor que el coste total de una serie de alquileres más breves. En tal caso, se puede devolver la primera canoa en alguna aldea k situada entre i y j y seguir el camino alquilando una nueva canoa (no hay costes adicionales por cambiar de canoa de esta manera).

Diseñe un algoritmo basado en programación dinámica para determinar el coste mínimo del viaje en canoa desde todos los puntos posibles de partida i a todos los posibles puntos de llegada j ($i < j$). Aplique dicho algoritmo en la resolución del caso cuya matriz de costos es la siguiente:

	1	2	3	4	5
1	0	3	3	inf	inf
2	-	0	4	7	inf
3	-	-	0	2	3
4	-	-	-	0	2
5	-	-	-	-	0

1.1.-Diseño de resolución por etapas.

El problema se puede resolver por etapas. En cada etapa se elegiría si se puede viajar desde la aldea i hasta la aldea j pasando o no por una aldea intermedia k . Así en cada etapa se considera si dividir el trayecto en subtrayectos cuyo coste sería $c(i,k)$ o hay una posible solución hasta la aldea final cuyo coste sería $c(i,j)$

1.2.-Ecuación recurrente.

La solución al problema puede estar compuesta por varias etapas. En cada etapa se considera el coste de viajar desde una aldea inicial i hasta una aldea destino j pasando o no por aldeas intermedias k . Llamaremos a $c(i,j)$ como el coste mínimo entre una aldea inicial i y una aldea final j .

En una instancia inicial, es decir en la aldea i , para viajar hasta la aldea destino j se consideran las siguientes posibilidades:

- Realizar un viaje directo desde la aldea i a la aldea j , donde el coste total sería $c(i,j)$.
- Realizar un viaje desde una aldea i hasta una aldea j pasando por una aldea intermedia k , donde el coste sería $c(i,j) + c(k,j)$.
- Por lo tanto $C(i,j) = \min[C(i,j), C(i,k)+C(k,j)]$

Los casos base para este problema de minimización son:

- No hay viaje posible,
 - ◆ o bien cuando $i \geq j$ (en cuyo caso sería como viajar en contracorriente),
 - ◆ o bien cuando $c(i,j) = \infty$ (en cuyo caso no hay camino directo posible entre una aldea i y una j),
 - ◆ o bien cuando $c(i,j) = 0$ (lo cual sería como viajar desde una aldea inicial i hasta una aldea final i).
- Si no se cumplen ninguna de las anteriores condiciones, significa que hay viaje directo entre la aldea i y la aldea j por lo tanto el coste sería $c(i,j)$

1.3.-Valor objetivo.

Se desea conocer el coste mínimo $c(i,j)$ de ir desde una aldea i a una aldea j . Dentro de esto hay que tener en cuenta que dicho coste puede ser directo o la acumulación de subcostes de ir pasando de una aldea i a una aldea k hasta llegar a la aldea destino j . Hay que contemplar también que el coste directo puede ser mayor que un determinado coste indirecto, por lo tanto la solución sería la suma de las diferentes subetapas por la que pase el problema.

1.4.-Verificación del POB.

Cualquier solución óptima $c(i,j)$ para viajar de la aldea i a la aldea j (donde $i < j$) debe ser una composición de subsoluciones óptimas. Es decir si una ruta óptima entre i y j pasa por una aldea k ($i < k < j$), entonces el coste de viajar desde i a k , $c(i,k)$ y el coste de viajar desde k a j , $c(k,j)$ también serán óptimos ya que sino fuera así la ruta entre i y j no sería óptima tampoco. Este hecho no se daría en una solución directa de i a j donde $c(i,j)$ sería el coste mínimo pero aun así seguiría siendo la solución óptima por no haber ninguna otra posibilidad de menor coste.

1.5.-Diseño de la memoria.

- Para resolver el problema, $C(i,j)$ sera la matriz asociada que almacena los costos de ir de una aldea i a una aldea j .
- Esta matriz tendrá n filas y n columnas. Es una matriz triangular superior con su diagonal en 0, ya que cada posición i comprobará el camino hacia la posición j siempre que esta sea mayor que i . Que sea 0 indica que se quiere ir desde la posición i a la posición j cuando $i=j$, lo cual no se puede.
- Cada celda de la matriz contendrá 4 valores posibles:
 - ◆ Valor nulo (-), para cada $\text{costes}(i,j)$ donde $i>j$ ya que esto implica ir a contracorriente del rio.
 - ◆ Valor 0, solo en la diagonal de la matriz lo que indica el coste de ir desde una aldea i a una aldea j donde $i=j$.
 - ◆ Valor numérico, el cual indica el coste ($\text{costes}(i,j)$) de ir desde una aldea i a una aldea j siempre que $i<j$.
 - ◆ Valor ∞ , que el camino de i a j no es posible por lo que habria que añadir una aldea intermedia.
- La memoria será rellenada de la siguiente manera:
 - ◆ Casos Base:
 - Si $i \geq j$, entonces $C(i,j) = \infty$
 - Si $i < j$ y hay un costo directo dado, entonces $C(i,j) = c(i,j)$
 - ◆ Rellenar las Celdas:
 - Iterar sobre todas las posibles aldeas desde i hasta j .
 - Para cada posible aldea intermedia k ($i < k < j$), Actualizar el coste mínimo $C(i,j)$.

1.6.-Diseño del algoritmo de cálculo de coste óptimo.

```
i6
i7 ALGORITMO RioAldeas(origen, destino, n, costes)
i8
i9   C ← inf : Matriz De Costes Minimos
i10
i11   //caso baso
i12   FOR i HASTA n HACER
i13     C[i][i] ← 0
i14   END
i15
i16   // Cálculo del costo mínimo
i17   FOR i HASTA n HACER
i18     FOR j HASTA n HACER
i19       C[i][j] ← CosteMinimo(i, j, n, costes)
i20     END
i21   END
i22
i23   devolver C[origen][destino]
i24 END
i25
i26 FUNCTION CostoMinimo( origen, destino, n, costes)
i27
i28   aux, k: INT
i29
i30   aux ← inf
i31   // Buscar el costo mínimo de viajar desde el origen hasta el destino
i32   FOR k ← origen HASTA n HACER
i33     aux ← min(aux, costes[origen][i] + costes[i][destino])
i34   END
i35
i36   devolver aux
i37 END
```

1.7.-Diseño del algoritmo de recuperación de la solución.

Para recuperar la solución comenzamos recorriendo la matriz desde la aldea destino. Mientras no se haya llegado al origen, se recorre todas las posibles aldeas intermedias, si las hay, que minimiza el coste total del viaje y si el coste total del viaje directo es mayor o igual que la suma de costes desde el origen a la aldea intermedia y la aldea intermedia y el destino, añade una aldea intermedia por la que pasar. Finalmente se añade dicha aldea a la ruta y se continúa comprobando nuevas posibilidades.

El resultado será un vector ruta con las aldeas por las que se llega a ese coste mínimo en cada posición del mismo.

Ejemplo:

Origen = 1 Destino=5 → Coste mínimo = 6 Ruta=[1,3,5]

```

0 FUNCTION Ruta(C, origen, destino)
1     aldeaActual=destino
2     ruta.add(origen)
3
4     while aldeaActual != origen
5         aldeaIntermedia = origen
6
7
8         FOR k = origen hasta aldeaActual
9             if C[origen][aldeaActual] >= C[origen][k] + C[k][aldeaActual]
10                 aldeaIntermedia = k
11             end
12         END
13
14         ruta.add(aldeaIntermedia)
15         aldeaActual=aldeaIntermedia
16     END
17     DEVOLVER ruta
18 END
1

```

Problema 2

Una empresa realiza planificaciones de viajes aéreos entre n ciudades. Para ir de una ciudad i a una ciudad j puede ser necesario coger varios vuelos distintos. El tiempo de un vuelo directo de i a j será (si existe) $T(i,j)$ (que puede ser distinto de $T(j,i)$). Hay que tener en cuenta que si cogemos un vuelo de i a k y después otro de k a j , será necesario esperar un tiempo de escala adicional $E(k)$ en el aeropuerto de k , con lo que el tiempo de ese viaje de i a j será de $T(i,k)+T(k,j)+E(k)$.

Se desea conocer la forma de volar desde cualquier ciudad i hasta cualquier otra j en el menor tiempo posible. Diseñad e implementad un algoritmo de Programación dinámica que resuelva este problema. Aplicado para resolver el problema, para $n = 4$, cuya matriz de tiempos es la siguiente, suponiendo que $E(k) = 1 \forall k$.

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

2.1.-Diseño de resolución por etapas.

El problema se puede resolver por etapas. En cada etapa se elegiría si se puede viajar más rápido desde la ciudad i y hasta la ciudad j sin pasar por una ciudad intermedia que añadiría un tiempo extra k . Así en cada etapa se considera si dividir el trayecto en subtrayectos cuyo coste sería $T(i,k)+T(k,j)+E(k)$ o hay una posible solución hasta la ciudad j cuyo coste sería $T(i,j)$.

2.2.-Ecuación recurrente.

La solución al problema puede estar compuesta por varias etapas. En cada etapa se considera el tiempo de viajar desde la ciudad inicial i hasta la ciudad destino j pasando o no por otras ciudades intermedias k . Llamaremos a $T(i,j)$ como el coste mínimo entre la ciudad inicial i y la ciudad final j .

En una instancia inicial, es decir, en la ciudad inicial i para viajar a la ciudad destino j , se consideran las siguientes posibilidades:

- Realizar el viaje directo desde la ciudad i hasta la ciudad j , donde el coste sería $T(i,j)$.

- Realizar un viaje desde la ciudad i hasta la ciudad j pasando por una ciudad intermedia k , donde el coste sería $T(i,k)+T(k,j)+E(k)$.

2.3.-Valor objetivo.

Se desea conocer el tiempo mínimo $T(i,j)$ de ir desde una ciudad i a una ciudad j . Dentro de esto hay que tener en cuenta que dicho tiempo puede ser directo o la acumulación de tiempos de ir pasando de una ciudad i a una ciudad k hasta llegar a la ciudad destino j . Hay que contemplar también que el tiempo directo puede ser mayor que un determinado tiempo indirecto, por lo tanto la solución sería la suma de las diferentes subetapas por la que pase el problema.

2.4.-Verificación del POB.

Cualquier solución óptima debe estar formada por subsoluciones óptimas. Es decir si una ruta óptima entre i y j pasa por una ciudad k ($i < k < j$), entonces el coste de viajar desde i a k , $T(i,k)$ y el coste de viajar desde k a j , $T(k,j)$ también serán óptimos ya que sino fuera así la ruta entre i y j no sería óptima tampoco. Este hecho no se daría en una solución directa de i a j donde $T(i,j)$ sería el tiempo mínimo pero aun así seguiría siendo la solución óptima por no haber ninguna otra posibilidad de menor coste.

2.5.-Diseño de la memoria.

- Para resolver el problema, $T(i,j)$ será la matriz asociada que almacena los tiempos de ir de una ciudad i a una ciudad j .
- Esta matriz tiene n filas y n columnas. Cada fila tendrá una ciudad que está asociada a una columna con otra ciudad. La diagonal está rellena de 0 puesto que son la misma ciudad.
- Cada celda de la matriz $T(i,j)$ contendrá el tiempo que tarda el viaje de i a j . Los de la misma ciudad será 0 puesto que es el mismo inicio que destino.

2.6.-Diseño del algoritmo de cálculo de coste óptimo.

Inicialización: $dist[i][j]$ se inicializa con $T[i][j]$ si hay un vuelo directo, infinito si no hay vuelo directo, y 0 para $dist[i][i]$. $sig[i][j]$ se inicializa con j si hay un vuelo directo, y -1 si no hay vuelo directo.

Floyd con tiempos de espera en escalas: Se considera cada ciudad k como posible escala. Si la ruta pasando por k es más corta que la ruta directa, se actualiza $dist[i][j]$ y se ajusta $sig[i][j]$ para indicar que la mejor ruta pasa por k .

```
Algoritmo Floydvuelos(n, T, E)
  Entrada:
    n: Número de ciudades
    T: Matriz de tiempos de vuelo directos (0 si no hay vuelo directo)
    E: Vector de tiempos de espera en cada ciudad

  Salida:
    dist: Matriz de tiempos mínimos entre todas las ciudades
    sig: Matriz para recuperar las rutas

  // Inicialización de la matriz de distancias y la matriz de recuperación
  Para i desde 0 hasta n-1 hacer
    Para j desde 0 hasta n-1 hacer
      Si i == j entonces
         $dist[i][j] \leftarrow 0$ 
         $sig[i][j] \leftarrow -1$ 
      Sino Si  $T[i][j] \neq 0$  entonces
         $dist[i][j] \leftarrow T[i][j]$ 
         $sig[i][j] \leftarrow j$ 
      Sino
         $dist[i][j] \leftarrow INF$ 
         $sig[i][j] \leftarrow -1$ 
      FinSi
    FinPara
  FinPara

  // Floyd con tiempos de espera en escalas
  Para k desde 0 hasta n-1 hacer
    Para i desde 0 hasta n-1 hacer
      Para j desde 0 hasta n-1 hacer
        Si  $dist[i][k] \neq INF$  y  $dist[k][j] \neq INF$  y  $dist[i][j] > dist[i][k] + dist[k][j] + E[k]$  entonces
           $dist[i][j] \leftarrow dist[i][k] + dist[k][j] + E[k]$ 
           $sig[i][j] \leftarrow sig[i][k]$ 
        FinSi
      FinPara
    FinPara
  FinPara

  Retornar dist, sig
FinAlgoritmo
```

2.7.-Diseño del algoritmo de recuperación de la solución.

```
1 Algoritmo RecuperarRuta(next, i, j)
2   Entrada:
3     next: Matriz para recuperar las rutas
4     i: Ciudad de inicio
5     j: Ciudad de destino
6
7   Salida:
8     ruta: Lista de ciudades que componen la ruta óptima de i a j
9
10  Si sig[i][j] == -1 entonces
11    Retornar "No hay ruta disponible"
12  FinSi
13
14  ruta ← Lista vacía
15  actual ← i
16  Mientras actual ≠ j hacer
17    Añadir actual a ruta
18    actual ← sig[actual][j]
19  FinMientras
20  Añadir j a ruta
21
22  Retornar ruta
23 FinAlgoritmo
```

2.8.-Implementación de los algoritmos de cálculo de coste óptimo y recuperación de la solución.

```

7
8 using namespace std;
9
10 // Constante para representar infinito
11 const int INF = numeric_limits<int>::max();
12
13 // Algoritmo de Floyd con tiempos de espera en escalas
14 void floydAviones(int n, vector<vector<int>>& T, vector<int>& E, vector<vector<int>>& dist, vector<vector<int>>& sig) {
15     // Inicializar la matriz de distancias y la matriz de recuperación
16     for (int i = 0; i < n; ++i) {
17         for (int j = 0; j < n; ++j) {
18             if (i == j) {
19                 dist[i][j] = 0;
20                 sig[i][j] = -1;
21             } else if (T[i][j] != 0) {
22                 dist[i][j] = T[i][j];
23                 sig[i][j] = j;
24             } else {
25                 dist[i][j] = INF;
26                 sig[i][j] = -1;
27             }
28         }
29     }
30
31     // Floyd-Marshall con tiempos de espera en escalas
32     for (int k = 0; k < n; ++k) {
33         for (int i = 0; i < n; ++i) {
34             for (int j = 0; j < n; ++j) {
35                 if (dist[i][k] != INF && dist[k][j] != INF && dist[i][j] > dist[i][k] + dist[k][j] + E[k]) {
36                     dist[i][j] = dist[i][k] + dist[k][j] + E[k];
37                     sig[i][j] = sig[i][k];
38                 }
39             }
40         }
41     }
42 }
43
44 // Algoritmo de recuperación de la ruta óptima
45 vector<int> recupera(vector<vector<int>>& sig, int i, int j) {
46     vector<int> camino;
47     if (sig[i][j] == -1) {
48         return camino; // No hay ruta disponible
49     }
50     int current = i;
51     while (current != j) {
52         camino.push_back(current);
53         current = sig[current][j];
54     }
55     camino.push_back(j);
56     return camino;
57 }

```

```

58
59 int main() {
60     // Número de ciudades
61     int n = 4;
62
63     // Matriz de tiempos de vuelo directos (0 si no hay vuelo directo)
64     vector<vector<int>> T = {
65         {0, 2, 1, 3},
66         {7, 0, 9, 2},
67         {2, 2, 0, 1},
68         {3, 4, 8, 0}
69     };
70
71     // Tiempo de espera en cada ciudad (escalas)
72     vector<int> E = {1, 1, 1, 1};
73
74     // Matriz de distancias y matriz de recuperación
75     vector<vector<int>> dist(n, vector<int>(n, INF));
76     vector<vector<int>> sig(n, vector<int>(n, -1));
77
78     // Ejecutar el algoritmo de Floyd-Marshall
79     floydAviones(n, T, E, dist, sig);
80
81     // Imprimir la matriz de tiempos mínimos entre todas las ciudades
82     cout << "La matriz de tiempos mínimos entre todas las ciudades es:" << endl;
83     for (int i = 0; i < n; ++i) {
84         for (int j = 0; j < n; ++j) {
85             if (dist[i][j] == INF) {
86                 cout << "INF ";
87             } else {
88                 cout << dist[i][j] << " ";
89             }
90         }
91         cout << endl;
92     }
93
94     // Recuperar y mostrar la ruta óptima entre dos ciudades específicas
95     int ciudadInicio = 1;
96     int ciudadDestino = 0;
97     vector<int> ruta = recupera(sig, ciudadInicio, ciudadDestino);
98
99     if (ruta.empty()) {
100         cout << "No hay ruta disponible de " << ciudadInicio << " a " << ciudadDestino << endl;
101     } else {
102         cout << "La ruta óptima de " << ciudadInicio << " a " << ciudadDestino << " es: ";
103         for (int ciudad : ruta) {
104             cout << ciudad << " ";
105         }
106         cout << endl;
107     }
108 }

```

2.10.-Ejemplo de ejecución.

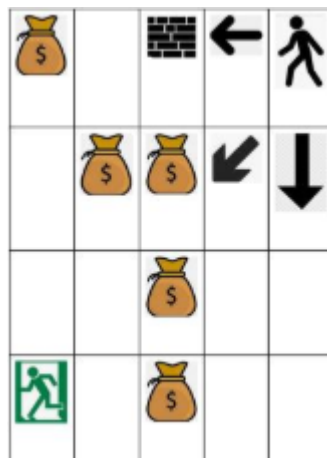
```
alberto@alberto-VirtualBox:~/Descargas/ALGP1$ make
Pruebe a ejecutar el algoritmo iterativo con ./Burbuja.bin salida.txt 12345 1000
2000 3000
Pruebe a ejecutar el algoritmo recursivo con ./MergeSort.bin salida.txt 12345 10
00 2000 3000
./Quicksort.bin quicksort.txt 12345 1000 2000 3000 4000 5000 6000 7000 8000 9000
10000
./Tornillodyv.bin tornillodyv..txt 12345 500 1000 1500 2000 2500 3000 3500 4000
4500 5000
./Tornilloiter.bin tornilloiter..txt 12345 500 1000 1500 2000 2500 3000 3500 400
0 4500 5000
./Duplicados.bin duplicados..txt 12345 500 1000 1500 2000 2500 3000 3500 4000 45
00 5000
./Duplicadosdyv.bin duplicadosdyv..txt 12345 500 1000 1500 2000 2500 3000 3500 4
000 4500 5000
./Aviones.bin aviones..txt
```

```
alberto@alberto-VirtualBox:~/Descargas/ALGP1$ ./Aviones.bin
La matriz de tiempos mínimos entre todas las ciudades es:
0 2 1 3
6 0 8 2
2 2 0 1
3 4 5 0
La ruta óptima de 1 a 0 es: 1 3 0
```

```
@echo ./Burbuja.bin burbuja..txt
@g++ -oAviones.bin aviones.cpp -std=c++11
@echo ./Aviones.bin aviones..txt
```

Problema 3

Un videojuego se juega por turnos y se representa en un mapa cuadrulado bidimensional de f filas y c columnas. El jugador siempre entra al mapa por la esquina superior derecha, y sale por la esquina inferior izquierda. En cada turno, los posibles movimientos del jugador son: ir 1 casilla a la izquierda, ir 1 casilla abajo, o moverse 1 posición a la casilla inferior izquierda. Cada casilla del mapa puede estar vacía, contener un muro, o contener una bolsa de oro. Todas las casillas son transitables salvo las que tienen muros. El objetivo consiste en llegar a la salida pudiendo recoger tanto oro como sea posible (pasar por tantas casillas que contengan una bolsa como se pueda). En el ejemplo siguiente, el jugador puede conseguir un máximo de 3 bolsas de oro con los movimientos permitidos.



3.1.-Diseño de resolución por etapas.

El problema se puede resolver por etapas. En cada etapa se selecciona que movimiento hacer. Los movimientos permitidos son a la izquierda, abajo y la diagonal abajo-izquierda y supondremos que las casillas con muros son .1, las vacías 0 y las con monedas 1.

3.2.-Ecuación recurrente.

La solución depende de las etapas(de los movimientos que se escojan) entre la casilla inicial i y la casilla j que será la casilla siguiente en el camino a la casilla de salida. Denominaremos $C(i,j)$ a la cantidad máxima de bolsas de oro entre la casilla i y la j .

En una instancia inicial para llegar desde i hasta j se considera estas posibilidades:

- se decidirán los movimientos conforme a cual tiene bolsa de oro. podría ser :
 - la casilla $i,j-1$
 - la casilla $i+1,j$
 - la casilla $i+1,j-1$

por lo que el beneficio $C(i,j)$ sería: $\max\{C(i,j-1), C(i+1,j), C(i+1,j-1)\}$

el caso base para esta ecuación sería:

- no hay oro en la cercanía: $C(i,j)=0$, da igual las posibilidades que hagamos que no hay beneficio
- hay oro en todos los posibles movimientos: $C(i,j)=1$, da igual las posibilidades que hagamos que hay beneficio.

3.3.-Valor objetivo.

Se desea conocer la cantidad de bolsas máximas $C(i,j)$ de ir desde una casilla i a una j . Dentro de esto hay que tener en cuenta que no se puede ir a casillas obstáculos y que hay que elegir la casilla con bolsa de oro o la más cercana a ella.

3.4.-Verificación del POB.

Si las soluciones son óptimas sus subsoluciones también tienen que serlo. En este caso es así dado que en cada etapa $C(i,j)$ siempre va a ser el máximo entre todas las posibilidades $\max\{C(i,j-1), C(i+1,j), C(i+1,j-1)\}$. Por lo que siempre se va a elegir la opción que lleva a mayor cantidad de bolas de oro.

3.5.-Diseño de la memoria.

- Para resolver el problema $C(i,j)$ será representado como una matriz bidimensional

- Esta matriz tendrá n filas y columnas y casilla de inicio será la 0, n-1 y la casilla de salida la n-1,0
- Cada casilla contendrá qué es un obstáculo(-1), que está vacía (0) o que tiene bolsa de oro(1).
- la matriz se rellenará de la siguiente manera:
 1. el caso base, la primera etapa
 2. El resto de casillas se rellenará en orden ascendente en filas y descendentes en columna sumando el valor que tengan al de la casilla de la que viene.

3.6.-Diseño del algoritmo de cálculo de coste óptimo.

Con este diseño el algoritmo de Programación dinámica sería de la siguiente manera:

```

ALGORITMO ColeccionarOro(Mapa, Fil, Col)
  M ← matriz de Fil filas y Col columnas

  Para cada fila i en {0...Fil-1}, hacer:
    Para cada columna j en {0...Col-1}, hacer:
      M[i][j] ← 0

  M[0][Col-1] ← 0

  Para cada fila i en {0...Fil-1}, hacer:
    Para cada columna j en {Col-1...0}, hacer:
      Var ← -1

      Si Mapa[i][j] = -1 Entonces
        M[i][j] ← -1
      Sino
        Si j+1 < Col Entonces
          Var ← máximo(Var, M[i][j+1])
        Fin Si

        Si i-1 >= 0 Entonces
          Var ← máximo(Var, M[i-1][j])
        Fin Si

        Si j+1 < Col y i-1 >= 0 Entonces
          Var ← máximo(Var, M[i-1][j+1])
        Fin Si

        Si Mapa[i][j] = 1 y Var ≠ -1 Entonces
          Var ← Var + 1
        Fin Si
      Fin Si
    Fin Para
  Fin Para

  Devolver M

```


3.7.-Diseño del algoritmo de recuperación de la solución.

La solución se recupera desde la casilla de salida y va cogiendo las casillas con mayor valor y va guardando la secuencia de posiciones en un vector formando así el camino de mayores bolsas de oro.

```
ALGORITMO SacarSolucion(Mapa, Fil, Col)
  Camino ← lista de pares
  i ← Fil - 1
  j ← 0
  Mientras (i ≠ 0 o j ≠ Col - 1), hacer:
    Insertar (i, j) al inicio de Camino
    Var ← -1
    Si Mapa[i][j] ≠ -1 Entonces
      Si j + 1 < Col Entonces
        Var ← Mapa[i][j + 1]
        Siguierte_i ← i
        Siguierte_j ← j + 1
      Fin Si
      Si i - 1 ≥ 0 Entonces
        Si Var < Mapa[i - 1][j] Entonces
          Var ← Mapa[i - 1][j]
          Siguierte_i ← i - 1
          Siguierte_j ← j
        Fin Si
      Fin Si
      Si j + 1 < Col y i - 1 ≥ 0 Entonces
        Si Var < Mapa[i - 1][j + 1] Entonces
          Var ← Mapa[i - 1][j + 1]
          Siguierte_i ← i - 1
          Siguierte_j ← j + 1
        Fin Si
      Fin Si
      i ← Siguierte_i
      j ← Siguierte_j
    Fin Si
    Insertar (i, j) al inicio de Camino
  Fin Mientras
  Devolver Camino
FIN ALGORITMO
```

3.8.-Ejemplo de ejecución.

Para ejecutar este problema debemos de compilarlo como:

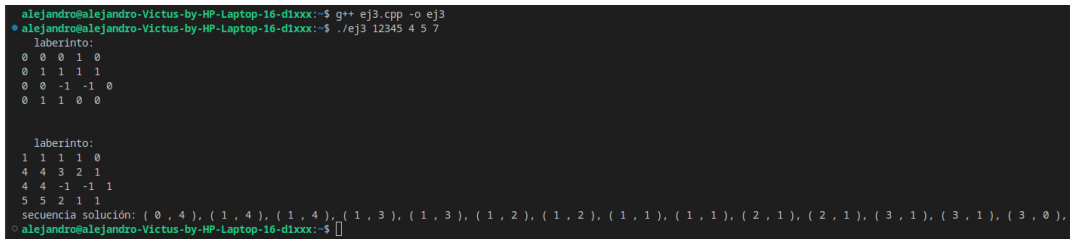
```
g++ ej3.cpp -o ej3
```

Y ejecutarlo como:

```
./ej3 semilla n°fil, n°col,n°bolsas de oro
```

```
./ej3 12345 4 5 7
```

Aquí dejamos una imagen de como quedaría:

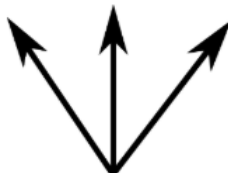


```
alejandro@alejandro-Victus-by-HP-Laptop-16-d1xxx:~$ g++ ej3.cpp -o ej3
alejandro@alejandro-Victus-by-HP-Laptop-16-d1xxx:~$ ./ej3 12345 4 5 7
laberinto:
0 0 0 1 0
0 1 1 1 1
0 0 -1 -1 0
0 1 1 0 0

laberinto:
1 1 1 1 0
4 4 3 2 1
4 4 -1 -1 1
5 5 2 1 1
secuencia solución: ( 0 , 4 ), ( 1 , 4 ), ( 1 , 4 ), ( 1 , 3 ), ( 1 , 3 ), ( 1 , 2 ), ( 1 , 2 ), ( 1 , 1 ), ( 1 , 1 ), ( 2 , 1 ), ( 2 , 1 ), ( 3 , 1 ), ( 3 , 1 ), ( 3 , 0 ),
alejandro@alejandro-Victus-by-HP-Laptop-16-d1xxx:~$
```

Problema 4

Tenemos que ponernos en forma, y hemos decidido empezar subiendo la “Pixel Mountain”. Para ello, tenemos que ascender desde una posición baja hasta la cumbre. Evidentemente, una parte muy importante de toda ascensión es decidir por dónde deberíamos subir. En nuestra ascensión siempre subimos (sin desfallecer) pero podemos movernos directamente hacia arriba, o



desplazarnos a la izquierda o derecha en diagonal:

En cada posible posición de la montaña tenemos un coste asociado de la dificultad que tiene llegar a esa posición. El objetivo es calcular el recorrido con menor dificultad:

Un ejemplo:

2	8	9	5	8
4	4	6	2	3
5	7	5	6	1
3	2	5	4	8

De esta montaña 5x4 se puede observar que el mejor camino (el más fácil) es el marcado en negrita, es decir, el que implica la dificultad de $4+1+2+5 = 12$. El plan (secuencia de pasos) serían las posiciones 3-4-3-3 (si numeramos desde 0).

4.1.-Descripción del problema.

Dada una matriz C de m filas y n columnas que representa la dificultad de cada posición en la montaña, queremos encontrar el camino desde cualquier posición en la primera fila hasta la última fila con el menor costo posible. Podemos movernos directamente hacia arriba, o en diagonal hacia la izquierda o derecha.

4.2.-Diseño de resolución por etapas.

El problema se puede resolver por etapas. En cada etapa se decide desde qué celda de la fila anterior se ha llegado a la celda actual. Para cada celda (i, j) en

la fila i , evaluamos tres posibles celdas en la fila $i - 1$ a las que podemos subir, justo encima de la que estamos $(i-1, j)$, en diagonal a la derecha $(i-1, j+1)$ y en diagonal a la izquierda $(i-1, j-1)$. El costo mínimo para llegar a (i, j) se obtiene sumando el costo de la celda actual al mínimo costo acumulado de la siguiente celda que escojamos. Repetimos este proceso para cada celda en cada fila, acumulando los costos mínimos hasta llegar a la última fila.

4.3.-Ecuación recurrente.

Sea $T(i, j)$ el costo mínimo para llegar a la celda (i, j) . Entonces, para cada celda (i, j) en la fila i , el costo mínimo para llegar a esa celda depende del costo mínimo de las celdas desde las que se puede llegar a (i, j) .

Dicho esto, volvemos a poner sobre la mesa los 3 posibles movimientos o decisiones que se pueden tomar cuando nos encontramos en una celda (i, j) :

- Subir recto: En tal caso, el costo mínimo sería el calculado hasta la celda (i, j) más el costo de ir a la celda $(i-1, j)$.
- Subir en diagonal a la derecha: Nos encontramos que el costo mínimo sería el calculado hasta la celda (i, j) más el costo de ir a la celda $(i-1, j+1)$.
- Subir en diagonal a la izquierda: En esta situación, el costo mínimo inicial sería igual que el de las dos anteriores, el calculado hasta la celda (i, j) más el costo de ir a la celda $(i-1, j-1)$.

Con estas tres situaciones posibles y ante un problema de minimización como es este, expresamos la ecuación recurrente como:

$$T(i, j) = C(i, j) + \min(T(i-1, j-1), T(i-1, j), T(i-1, j+1)).$$

Los casos base para esta ecuación serían:

- Para las celdas en el borde izquierdo ($j = 0$):

$$T(i, 0) = C(i, 0) + \min(T(i-1, 0), T(i-1, 1))$$
- Para celdas en el borde derecho ($j = n-1$):

$$T(i, n-1) = C(i, n-1) + \min(T(i-1, n-2), T(i-1, n-1))$$
- Para celdas interiores ($0 < j < n-1$):

$$T(i, j) = C(i, j) + \min(T(i-1, j-1), T(i-1, j), T(i-1, j+1))$$

4.4.-Valor objetivo.

Se desea conocer el mínimo costo en la última fila: $\min T(m-1, j)$, que representa la menor dificultad para llegar a la cumbre desde cualquier posición en la base de la montaña. Destacar que este valor será acumulativo, por lo que será el camino con menor coste desde la base hasta la última fila.

4.5.-Verificación del POB.

Cualquier solución óptima debe estar formada por subsoluciones óptimas. En este caso, cada decisión en la ecuación recurrente se elige minimizando la dificultad total, lo que garantiza que cualquier solución encontrada utilizando esta ecuación será óptima, por lo que cumplirá con el Principio de la Optimalidad de Bellman.

4.6.-Diseño de la memoria.

- Para resolver el problema, $T(i, j)$ será representada como una matriz bidimensional.
- Esta matriz tendrá 'm' filas y 'n' columnas. La casilla de inicio puede ser cualquiera posición en la última fila ($m = m-1$), representando la base de la montaña. La casilla de salida puede ser cualquier posición en la primera fila ($m = 0$), que representa la cima de la montaña.
- Cada casilla contendrá solamente el coste asociado de la dificultad que tiene llegar a esa posición.

4.7.-Diseño del algoritmo de cálculo de coste óptimo.

Con este diseño, construimos el algoritmo de Programación Dinámica como:

FUNCION encontrar_minimo_costo(montana, m, n)

T = nueva matriz de tamaño $m \times n$, inicializada con INT_MAX

 // Caso base: la última fila es igual a los costos de la última fila de la montaña

 PARA cada j desde 0 hasta $n-1$ HACER

$T[m-1][j] = \text{montana}[m-1][j]$

 FIN PARA

 // Rellenamos la matriz T con los valores mínimos de dificultad acumulada

 PARA cada i desde $m-2$ hasta 0 HACER

 PARA cada j desde 0 hasta $n-1$ HACER

 SI $j-1 \geq 0$ ENTONCES

$\text{izquierda} = T[i+1][j-1]$

 SINO

$\text{izquierda} = \text{INT_MAX}$

 FIN SI

```

        arriba = T[i+1][j]

        SI j+1 < n ENTONCES
            derecha = T[i+1][j+1]
        SINO
            derecha = INT_MAX
        FIN SI

        T[i][j] = montaña[i][j] + minimo(izquierda, arriba, derecha)
    FIN PARA
FIN PARA

DEVOLVER T
FIN FUNCION

```

Esta función devuelve una matriz donde cada celda es el mínimo costo que hay para llegar a esa celda desde las anteriores. Por lo que en la primera fila ($m=0$, simulando la cima de la montaña) estará el costo mínimo para cada camino. Nos va a interesar el menor valor

4.8.-Diseño del algoritmo de recuperación de la solución.

Este algoritmo comienza desde la primera fila superior de la matriz T, que esta fila se corresponde con la cima de la montaña y la dificultad acumulada desde la base para llegar a ella, por lo que va a coger el menor valor y va a empezar a construir el camino desde esa casilla, es decir, va hacer lo contrario que el algoritmo anterior puesto que en este caso va a comenzar desde la cima hasta llegar a la base(lo contrario al anterior).

```

FUNCION recuperar_camino(T, pos_min_dificultad)
    m = tamaño de filas de T
    n = tamaño de columnas de T
    camino = nuevo vector de tamaño m
    j = pos_min_dificultad

    camino[0] = j

    PARA cada i desde 0 hasta m-2 HACER
        SI j-1 >= 0 ENTONCES

```

```

        izquierda = T[i+1][j-1]
    SINO
        izquierda = INT_MAX
    FIN SI

    arriba = T[i+1][j]

    SI j+1 < n ENTONCES
        derecha = T[i+1][j+1]
    SINO
        derecha = INT_MAX
    FIN SI

    SI izquierda <= arriba y izquierda <= derecha ENTONCES
        j -= 1
    SINO SI derecha <= arriba y derecha <= izquierda ENTONCES
        j += 1
    FIN SI

    camino[i+1] = j

FIN PARA
DEVOLVER camino
FIN FUNCION

```

4.9.-Implementación de los algoritmos de cálculo de coste óptimo y recuperación de la solución.

```
1
2  /**
3   * @file: ej4_p5_algoritmica.cpp
4   *
5   * @brief: Solución al problema 4 de la práctica 5 de algoritmica.
6   *
7   * Autores:
8   * - Miguel Martínez Azor
9   * - Ángel Rodríguez Faya
10  * - Alejandro Botaro Crespo
11  * - Alberto Parejo Bellido
12  * - Alejandro Ocaña Sánchez
13  *
14  * ALGORÍTMICA CURSO 2023/2024
15  * 2o de Ingeniería Informática
16  * Universidad de Granada
17  *
18  */
19
20
21 #include <iostream>
22 #include <vector>
23 #include <climits>
24 #include <algorithm> // Para std::min
25
26 using namespace std;
27
28 // Función para encontrar el mínimo costo de ascenso
29 vector<vector<int>> encontrar_minimo_costo(vector<vector<int>>& montana, int m, int n) {
30     // Inicializar la matriz de dificultad mínima con infinito
31     vector<vector<int>> T(m, vector<int>(n, INT_MAX));
32
33     // Caso base: la última fila es igual a los costos de la última fila de la montaña
34     for (int j = 0; j < n; ++j) {
35         T[m-1][j] = montana[m-1][j];
36     }
37
38     // Rellenar la matriz T con los valores mínimos de dificultad acumulada
39     for (int i = m-2; i >= 0; --i) {
40         for (int j = 0; j < n; ++j) {
41
42             // Si estamos en la primera columna, no podemos ir a la izquierda
43             int izquierda = (j-1 >= 0) ? T[i+1][j-1] : INT_MAX;
44
45             int arriba = T[i+1][j];
46
47             // Si estamos en la última columna, no podemos ir a la derecha
48             int derecha = (j+1 < n) ? T[i+1][j+1] : INT_MAX;
49
50             T[i][j] = montana[i][j] + min({izquierda, arriba, derecha});
51         }
52     }
53
54     return T;
55 }
```



```

57 // Función para recuperar el camino con el mínimo costo
58 vector<int> recuperar_camino(const vector<vector<int>>& T, int pos_min_dificultad) {
59     int m = T.size();
60     int n = T[0].size();
61     vector<int> camino(m);
62     int j = pos_min_dificultad;
63
64     camino[0] = j;
65
66     for (int i = 0; i < m-1; ++i) {
67         int izquierda = (j-1 >= 0) ? T[i+1][j-1] : INT_MAX;
68         int arriba = T[i+1][j];
69         int derecha = (j+1 < n) ? T[i+1][j+1] : INT_MAX;
70
71         if (izquierda <= arriba && izquierda <= derecha) {
72             j -= 1;
73         } else if (derecha <= arriba && derecha <= izquierda) {
74             j += 1;
75         }
76
77         camino[i+1] = j;
78     }
79
80     return camino;
81 }
82
83 int main() {
84     vector<vector<int>> montana = {
85         {2, 8, 9, 5, 8},
86         {4, 4, 6, 2, 3},
87         {5, 7, 5, 6, 1},
88         {3, 2, 5, 4, 8}
89     };
90
91     int m = 4, n = 5;
92
93     vector<vector<int>> T = encontrar_minimo_costo(montana, m, n);
94
95     // Encontrar el costo mínimo en la primera fila
96     int min_dificultad = INT_MAX;
97     int pos_min_dificultad = -1;
98     for (int j = 0; j < T[0].size(); ++j) {
99         if (T[0][j] < min_dificultad) {
100             min_dificultad = T[0][j];
101             pos_min_dificultad = j;
102         }
103     }
104
105     vector<int> camino = recuperar_camino(T, pos_min_dificultad);
106
107     cout << "Dificultad mínima: " << min_dificultad << endl;
108     cout << "Camino: ";
109     for (int i = camino.size()-1; i >= 0; --i) {
110         cout << camino[i] << " ";
111     }
112     cout << endl;
113
114     return 0;
115 }

```

4.10.-Ejemplo de ejecución.

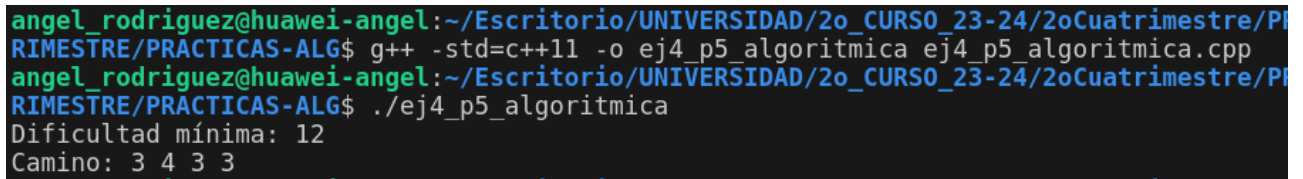
Para ejecutar este problema debemos de compilarlo como:

```
g++ -std=c++11 -o ej4_p5_algoritmica ej4_p5_algoritmica.cpp
```

Y ejecutarlo como:

```
./ej4_p5_algoritmica
```

Aquí dejamos una imagen de como quedaría:



```
angel_rodriguez@huawei-angel:~/Escritorio/UNIVERSIDAD/2o_CURSO_23-24/2oCuatrimestre/PR  
RIMESTRE/PRACTICAS-ALG$ g++ -std=c++11 -o ej4_p5_algoritmica ej4_p5_algoritmica.cpp  
angel_rodriguez@huawei-angel:~/Escritorio/UNIVERSIDAD/2o_CURSO_23-24/2oCuatrimestre/PR  
RIMESTRE/PRACTICAS-ALG$ ./ej4_p5_algoritmica  
Dificultad mínima: 12  
Camino: 3 4 3 3
```