Práctica 4:
Algoritmos de Programación Dinámica

Santiago Carbó García y Carlos García Segura

Universidad de Granada, España

1. Ejercicio 1

1.1. Diseño de resolución por etapas y ecuación recurrente

- El problema **se puede resolver por etapas**. Para encontrar el camino mínimo desde un vértice i a otro j en un grafo G, veríamos qué vértice debe ser el segundo, cuál tercero, etc., hasta alcanzar j. Una sucesión optimal de decisiones proporcionará entonces el camino de longitud mínima. Antes de ello hemos definido un grafo dirigido $G = (N, A)$ en el que N es el conjunto de nodos y A el de sus arcos.
- Los idiomas son los nodos y los diccionarios son las aristas.
- A continuación, para la ecuación recurrente:
  - Sea A_i el conjunto de vértices adyacentes al vértice i.
  - Para cada vértice $k \epsilon A_i$ sea Γ_k el camino mínimo desde k hasta j.
  - Entonces el camino más corto desde i hasta j es el más corto de los caminos del conjunto $\{i, \Gamma_k \,/\, k \epsilon A_i\}$.
  - A continuación, para resolver operativamente el problema, supondremos que los nodos están numerados de 1 a n, $N = \{1, 2, \ldots n,\}$ y que L da la longitud de cada arco, de modo que $L(i,i) = 0$, $L(i,j) \geq 0$ si *i* es distinto de *j*, y $L(i,j) = \infty$ si no existe el arco (i,j).
  - Tras la iteración k, la matriz D dará la cadena de traducciones de longitud mínima usando como nodos intermedios los del conjunto {1, 2, …, k}.
  - La **ecuación recurrente** puede expresar el caso base como:
    - $D_0(i,j) = D(i,j)$
    - Esta ecuación expresa que el camino de i a j es directo, ya que no es necesario cruzar ningún nodo.
  - La **ecuación recurrente** puede expresar el caso general como:
    - $D_k(i,j) = min\{D_{k-1}(i,j),\ D_{k-1}(i,k) + D_{k-1}(k,j)\}$
    - Esta ecuación expresa que el camino de i a j puede cruzar algún nodo, dependerá de cuál es el camino mínimo entre los dos. Si el mínimo es el camino directo ($D_{k-1}(i,j)$) utilizaremos ese, si el camino es más corto pasando por k utilizaremos ese.
- El **valor objetivo** sería conocer el valor de $D_k(i,j)$, es decir, el camino mínimo de i a j.

1.2. Diseño de la memoria

- Para resolver el problema, $D_k(i,j)$ se representará como una matriz.
- La matriz tendrá *n* filas y *n* columnas, siendo *n* el número de idiomas disponibles.
- Cada celda de la matriz $D_k(i,j)$ contendrá la cadena mínima de traducciones entre el idioma *i* y el idioma *j*.
- La memoria se rellenará de la siguiente forma:
    - En primer lugar, se rellenan las celdas correspondientes a los casos base, es decir, en cada celda habrá un 1 si existe diccionario entre ambos idiomas o un 0 si no existe. La matriz *L* contiene los casos base. Se realizan n iteraciones.
    - Posteriormente, se rellena la diagonal principal con ceros, ya que no tiene sentido que haya conexión entre un idioma y él mismo.
    - Finalmente, en la iteración k, se comprueba para cada par de nodos (i,j) si existe o no un camino que pase a través de k que sea mejor que el actual camino minimal que solo pasa a través de los nodos {1,2,...,k-1}.

1.3. Verificación del P. O. B

- Sea $i, i_{1,}, \ldots, i_{k}, j$ el camino mínimo desde i hasta j.
- Comenzando en el vértice inicial i, se ha tomado la decisión de ir al vértice i_1.
- Como resultado, ahora el estado del problema está definido por el vértice i_1, y lo que se necesita es encontrar un camino mínimo entre i_1 y j.
- Está claro que la sucesión $i_{1,}, \ldots, i_{k}, j$ debe constituir un camino mínimo entre $i_{1,}$ y j. Sí no:
    - Sea $i, i_{1,}, r_{1,}, \ldots, r_{q}, j$ un camino más corto entre $i_{1,}$ y j.
    - Entonces $i, i_{1,}, r_{1,}, r_{2,}, \ldots, r_{q}, j$ es un camino entre i y j que es más corto que el camino $i, i_{1,}, r_{1,}, \ldots, r_{q}, j$.
    - Como eso es una contradicción, se verifica el P. O. B.
- Esta demostración por reducción al absurdo ha servido para verificar el Principio de Optimalidad de Bellman.
- De esta forma, si k es un nodo en el camino mínimo que une i con j, entonces la parte de ese camino que va desde i hasta k, y la del que va de k hasta j, es también optimal.

1.4. Diseño del algoritmo de cálculo de coste óptimo

El procedimiento utilizado ha sido el Algoritmo de Floyd-Warshall:

```
Algoritmo Cálculo de coste óptimo – Traductor-Textos
Begin
      For i := 1 to n do
            For j := 1 to n do begin
                  D[i,j] := L[i,j];
                  P[i,j] := 0;
            End;

      For i := 1 to n do
            D[i,i] := 0;
      For k := 1 to n do
            For i := 1 to n do
                  For j := 1 to n do
                        If D[i,k] + D[k,j] < D[i,j] then begin
                              D[i,j] := D[i,k] + D[k,j];
                              P[i,j] := k;
                        End;
End Procedure
```

1.5. Diseño del algoritmo de recuperación de la solución

El procedimiento utilizado es el siguiente:

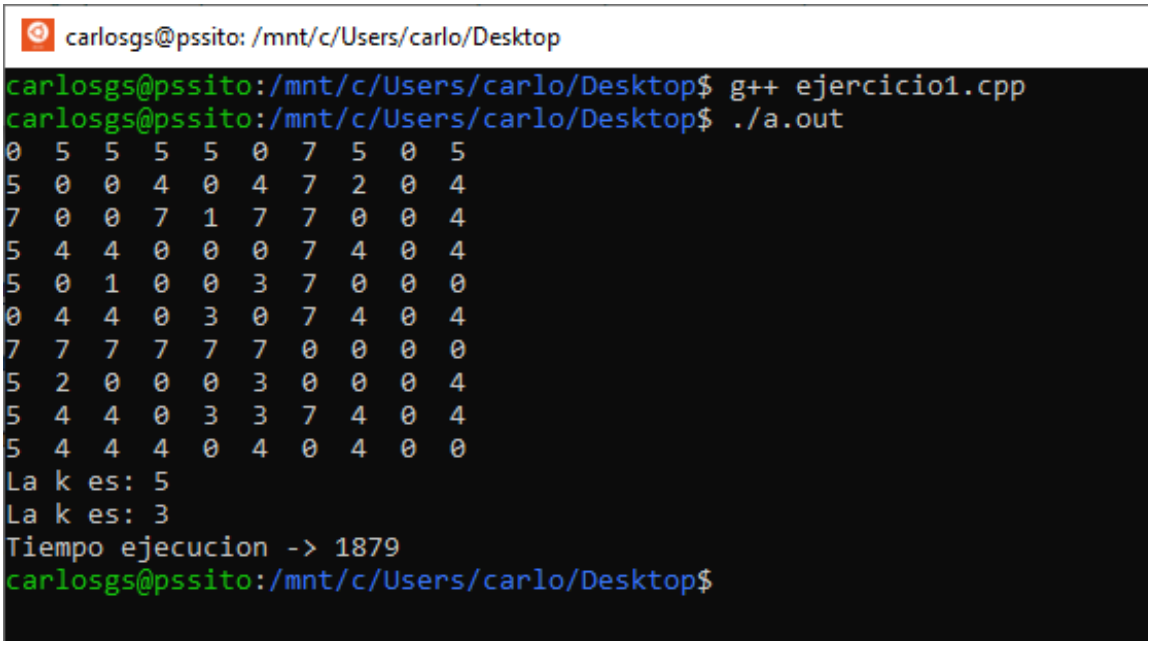
```
Algoritmo Recuperación de la solución – Traductor-Textos (nodo i, nodo j)
Begin
      k := P[i,j];
      If k := 0 then
            Return;
      Algoritmo Recuperación de la solución – Traductor-Textos(i,k);
      Writeln(k);
      Algoritmo Recuperación de la solución – Traductor-Textos(k,j);
End Procedure
```

La orden "**Writeln()**" hace alusión a imprimir el contenido en pantalla.

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

```
13  void calculoCosteOptimoEj1 (const vector<vector<int>>  &L, vector<vector<int>>  &D, vector<vector<int>>  &P,  int N)
14  {
15
16      for (int i = 0; i < N ; i++){
17          for (int j = 0; j < N; j++){
18              D[i][j] = L[i][j];
19              P[i][j] = 0;
20          }
21      }
22
23      for (int i = 0; i < N; i++){
24          D[i][i] = 0;
25      }
26
27      for (int k = 0; k < N; k++){
28          for (int i = 0; i < N; i++){
29              for (int j = 0; j < N; j++){
30                  if (D[i][k]+D[k][j] < D[i][j]){
31                      D[i][j] = D[i][k] + D[k][j];
32                      P[i][j] = k;
33                  }
34              }
35       }
36      }
37
38  }
39
40
41  void RecuperacionEj1 (int i, int j, vector<vector<int>>  &P)
42  {
43
44     int k = P[i][j];
45
46
47     if (k != 0){
48
49
50          RecuperacionEj1(i, k, P);
51          cout << "La k es: " << k << endl;
52          RecuperacionEj1(k, j, P);
53  }
54
55
56  }
57
```

En esta imagen podemos observar la implementación de los algoritmos encargados de calcular la matriz de cálculo coste óptimo y de recuperación de soluciones.



En esta ejecución se quiere traducir del lenguaje 0 al lenguaje 4. La matriz mostrada en la ejecución corresponde a la matriz de costes calculada en el método calculoCosteOptimoEj1. En esta matriz observamos que para traducir del lenguaje 0 al lenguaje 4 (observando la posición de la matriz 0,4) tenemos que usar el diccionario del lenguaje 5. Ahora comprobamos cómo podemos traducir del lenguaje 5 al lenguaje 4 (posición 5,4) y vemos que tenemos que usar el lenguaje 3, desde el lenguaje 3 vemos que hay conexión directa con el lenguaje 4 así que los lenguajes intermedios devueltos son el 5 y el 3.

2. Ejercicio 2

2.1. Diseño de resolución por etapas y ecuación recurrente

Este problema es similar al problema del "Turista de Manhattan". No es exactamente el mismo problema, ya que el problema del "Turista de Manhattan" considera que el valor de una casilla depende de la dirección en la que se venga. En este problema no ocurre, lo único relevante es que una casilla tenga una bolsa de dinero o no, no de qué dirección se viene.

- El problema **se puede resolver por etapas**. Esto es debido a que, a la hora de calcular el coste de avanzar hacia una nueva casilla, tenemos que tener en cuenta el coste de las casillas que hemos calculado anteriormente, vinculándose con una naturaleza n-etápica. El problema empieza conformando, inicialmente, el origen, y terminará conformando la matriz entera.
- A continuación, para la ecuación recurrente.
    - El valor de cada casilla es el que se expresa en la ecuación recurrente, es decir, el máximo de la suma de su propio valor y el valor de su casilla inmediatamente arriba, inmediatamente a la derecha o inmediatamente en la esquina superior derecha.
    - La **ecuación recurrente** para el caso base puede expresarse como:
        - $s_v = L_v$
        - En el caso base, el valor de la casilla es directamente el de la matriz de valores iniciales, es decir, no tiene en cuenta el valor de las anteriores casillas.
    - La **ecuación recurrente** para el caso general puede expresarse como:
        - $s_v = max\{s_{u1} + peso\ de\ la\ arista\ entre\ u_1\ y\ v,\ s_{u2} + peso\ de\ la\ arista\ entre\ u_2\ y\ v,\ s_{u3} + peso\ de\ la\ arista\ entre\ u_3\ y\ v\}$
        - En el caso general, esta ecuación expresa el beneficio máximo de avanzar una casilla y, como puede avanzar en tres direcciones, calculamos el beneficio máximos entre la suma del valor individual de la casilla y el valor de la casilla inmediatamente superior ($peso\ de\ la\ arista\ entre\ u_1$), inmediatamente a la derecha ($peso\ de\ la\ arista\ entre\ u_2$) e inmediatamente en la esquina superior derecha ($peso\ de\ la\ arista\ entre\ u_3$).
- El **valor objetivo** sería conocer el valor de s_v, es decir, el máximo beneficio de haber avanzado hasta dicha casilla.

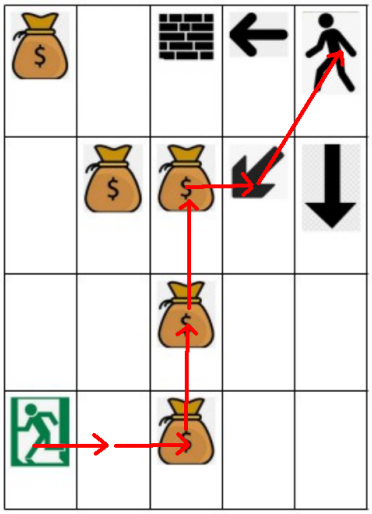
2.2. Diseño de la memoria

- Para resolver el problema, se construirá una matriz donde cada casilla será de la forma s_v.
- La matriz tendrá *n* filas y *m* columnas, de forma que no tiene por qué ser cuadrada. También existe una matriz de pesos que contiene el valor de cada casilla de manera individualizada.
- Cada celda de la matriz contendrá el beneficio acumulativo desde el origen, es decir, conforme se va avanzando en la matriz desde el origen hacia el destino el beneficio de cada casilla es mayor, ya que tiene en cuenta el beneficio de las casillas anteriores.
- La memoria se rellenará de la siguiente forma:
    - En primer lugar, se rellenan las celdas correspondientes a la primera fila y la primera columna, empezando desde el origen del problema, es decir, la fila *0*, rellenándose de arriba a abajo (ya que el origen se encuentra en la esquina superior derecha), y la columna *(columnas-1)*, rellenándose de derecha a izquierda.
    - Posteriormente, se empieza a rellenar la matriz desde la fila 1 y la columna (*columnas-2)*, de forma que se va rellenando columna por columna y fila por fila hasta que toda la matriz tiene valores.

2.3. Verificación del P. O. B

El Principio de Optimalidad de Bellman define: “una política es óptima si en un periodo dado, cualesquiera que sean las decisiones precedentes, las decisiones que quedan por tomar constituyen una política óptima independientemente de los resultados de las decisiones precedentes”.

Todas las subpolíticas son óptimas ya que, aplicando el algoritmo de recuperación de la solución desde la casilla final hasta la casilla inicial, nuestra visión es únicamente la casilla superior, las casilla de la derecha y la casilla de la esquina superior derecha. En el ejemplo mostrado se puede comprobar que, partiendo desde el final, el camino en cada etapa es óptimo:



2.4. Diseño del algoritmo de cálculo de coste óptimo

El procedimiento utilizado es el siguiente:

```
Algoritmo – Cálculo del coste – Videojuego
Begin

P[0][columns-1] = 0;
For i := 1 to (rows-1) do
      P[i][columns-1] := P[i-1][columns-1] + L[i][columns-1];
End;

For j := (columns-2) to 0 do
      P[0][j] := P[0][j+1] + L[0][j];
End;

For i := 1 to (rows-1) do
      For j := (columns-2) to 0 do
            P[i][j]  :=  MÁX(P[i-1][j]  +  L[i][j],  P[i][j+1]  +  L[i][j],
P[i-1][j+1] + L[i][j]);
      End;
End;

End Procedure
```

2.5. Diseño del algoritmo de recuperación de la solución

El procedimiento utilizado es el siguiente:

```
Algoritmo Recuperación de la solución – Videojuego
Begin
      i := (rows-1);
      j := 0;
      pair.first = i;
      pair.second = j;
      stack.push(pair);

      While i != 0 OR j != (columns-1)
            If i > 0 AND j < (columns-1) then begin
                  If P[i-1][j] >= P[i][j+1] AND P[i-1][j] >= P[i-1][j+1]
      then begin
                        pair.first = i-1;
                        pair.second = j;
                  Else  If  P[i][j+1]  >  P[i-1][j]  AND  P[i][j+1]  >
P[i-1][j+1] then begin
                        pair.first = i;
                        pair.second = j+1;
                  Else
                        pair.first = i-1;
                        pair.second = j+1;
                  End;
            Else If i > 0 then begin
                        pair.first = i-1;
                        pair.second = j;
            Else
                        pair.first = i;
                        pair.second = j+1;
            End;
            stack.push(pair);
            i := pair.first;
            j := pair.second;
      End;
```

```
      While !stack.empty() do
            Writeln(stack.pop());
      End;
End Procedure
```

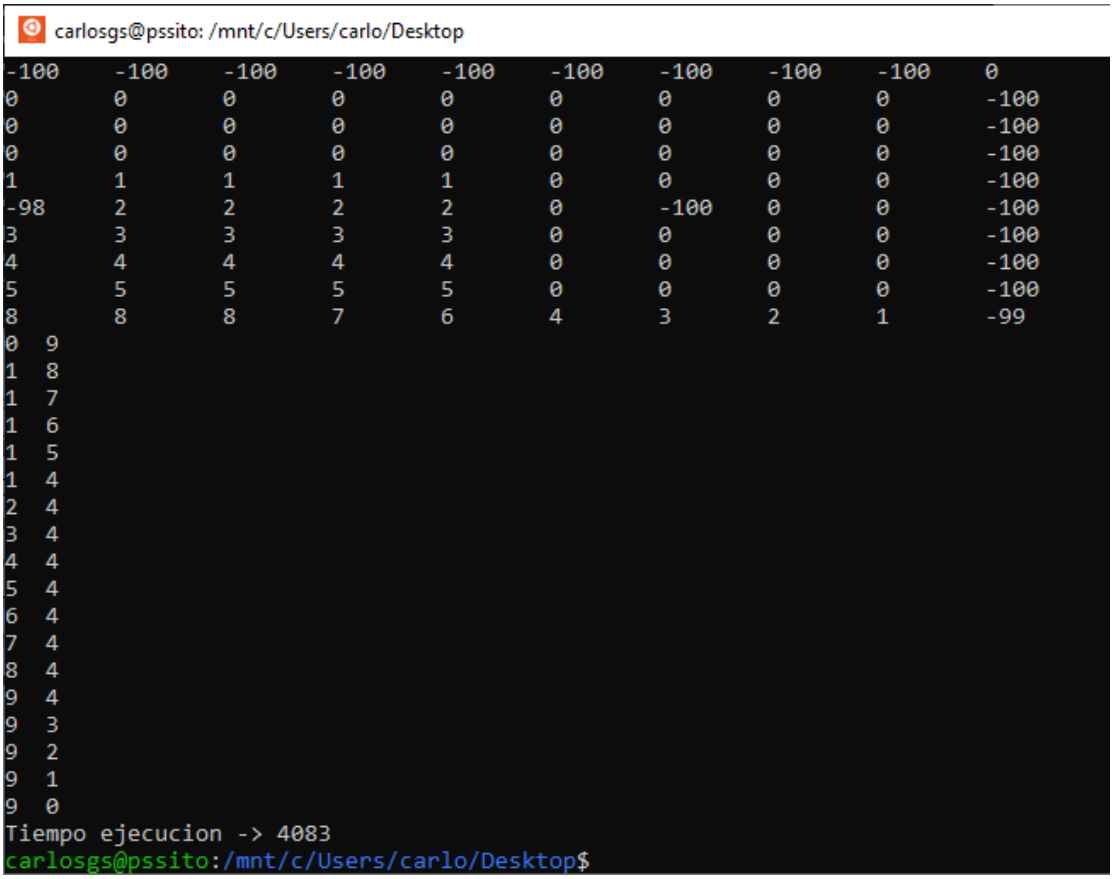
La orden “**WriteIn()**” hace alusión a imprimir el contenido en pantalla.

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

```
void CalculoCosteEj2(const vector<vector<int>> &L, vector<vector<int>> &P, int F, int C)
{

    P[0][C-1] = 0;

    for(int i = 1;i < F; i++)
    {
        P[i][C-1] = P[i-1][C-1] + L[i][C-1];
    }

    for (int j = C-2; j >= 0; j--)
    {
        P[0][j] = P[0][j+1] + L[0][j];
    }

    for (int i = 1; i < F; i++)
    {
        for (int j = C-2; j >= 0; j--)
        {
           P[i][j] = max(max(P[i-1][j] + L[i][j], P[i][j+1] + L[i][j]), P[i-1][j+1] + L[i][j]);
        }
    }
}
```

```
void RecuperacionEj2 (stack<pair<int,int>> pila, vector<vector<int>> &P, int F, int C)
{

    int i = F-1;
    int j = 0;
    pair<int,int> aux;
    aux.first = i;
    aux.second = j;
    pila.push(aux);

   while(i != 0 || j != (C-1)){
      if (i > 0 && j < (C-1)){
          if (P[i-1][j] >= P[i][j+1] && P[i-1][j] >= P[i-1][j+1]){
              aux.first = i-1;
              aux.second = j;
          } else if (P[i][j+1] > P[i-1][j] && P[i][j+1] > P[i-1][j+1]){
              aux.first = i;
              aux.second = j+1;
          }else{
              aux.first = i-1;
              aux.second = j+1;
          }
      } else if(i > 0){
          aux.first = i-1;
          aux.second = j;

      }else{
          aux.first = i;
          aux.second = j+1;
      }

       pila.push(aux);
       i = aux.first;
       j = aux.second;

   }
```

En estas imágenes podemos observar la implementación de los algoritmos encargados de calcular la matriz de cálculo coste óptimo y de recuperación de soluciones.



En esta ejecución podemos observar la matriz de cálculo de coste óptimo en la que se encuentran todos el “beneficio” máximo obtenible en cada casilla siendo las casillas de valores negativos muros que no se pueden recorrer.

Las líneas siguientes a la matriz nos muestran el camino escogido desde la esquina superior derecha hasta la esquina inferior izquierda para maximizar el beneficio.

3. Ejercicio 3

3.1. Diseño de resolución por etapas y ecuación recurrente

Este problema es similar al problema del "Turista de Manhattan". No es exactamente el mismo problema, ya que el problema del "Turista de Manhattan" considera que el valor de una casilla depende de la dirección en la que se venga. En este problema no ocurre, lo único relevante es el coste que tiene dicha casilla en términos del tipo de terreno, pero no es relevante la dirección de la que se proviene.

- El problema **se puede resolver por etapas**. Esto es debido a que, a la hora de calcular el coste de avanzar hacia una nueva casilla, tenemos que tener en cuenta el coste de las casillas que hemos calculado anteriormente, vinculándose con una naturaleza n-etápica. El problema empieza conformando, inicialmente, el origen, y terminará conformando la matriz entera.
- A continuación, para la ecuación recurrente:
    - El valor de cada casilla es el que expresa la ecuación recurrente, es decir, el mínimo de la suma de su propio valor y el valor de su casilla inmediatamente debajo, inmediatamente a la izquierda o inmediatamente en la esquina inferior izquierda.
    - La **ecuación recurrente** para el caso base puede expresarse como:
        - $s_v = L_v$
        - En el caso base, el valor de la casilla es directamente el de la matriz de valores iniciales, es decir, no tiene en cuenta el valor de las anteriores casillas.
    - La **ecuación recurrente** para el caso general puede expresarse como:
        - $s_v = min\{s_{u1} + peso\ de\ la\ arista\ entre\ u_1\ y\ v,\ s_{u2} + peso\ de\ la\ arista\ entre\ u_2\ y\ v,\ s_{u3} + peso\ de\ la\ arista\ entre\ u_3\ y\ v\}$
        - En el caso general, esta ecuación expresa el coste mínimo de avanzar una casilla y, como puede avanzar en tres direcciones, calculamos el coste mínimo entre la suma del valor individual de la casilla y el valor de la casilla inmediatamente inferior ($peso\ de\ la\ arista\ entre\ u_1$), inmediatamente a la izquierda ($peso\ de\ la\ arista\ entre\ u_2$) e inmediatamente en la esquina inferior izquierda ($peso\ de\ la\ arista\ entre\ u_3$).
- El **valor objetivo** sería conocer el valor de s_v, es decir, el mínimo coste de haber avanzado hasta dicha casilla.

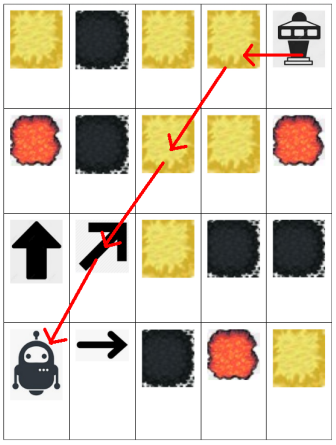
3.2. Diseño de la memoria

- Para resolver el problema, se construirá una matriz donde cada casilla será de la forma s_v.
- La matriz tendrá *n* filas y *m* columnas, de forma que no tiene por qué ser cuadrada. También existe una matriz de pesos que contiene el valor de cada casilla de manera individualizada.
- Cada celda de la matriz contendrá el coste acumulativo desde el origen, es decir, conforme se va avanzando en la matriz desde el origen hacia el destino el coste de cada casilla es mayor, ya que tiene en cuenta el coste de las casillas anteriores.
- La memoria se rellenará de la siguiente forma:
    - En primer lugar, se rellenan las celdas correspondientes a la primera fila y la primera columna, empezando desde el origen del problema, es decir, la fila *(filas-1)*, rellenándose de abajo a arriba (ya que el origen se encuentra en la esquina inferior izquierda), y la columna *0*, rellenándose de izquierda a derecha.
    - Posteriormente, se empieza a rellenar la matriz desde la fila *(filas-2)* y la columna *1*, de forma que se va rellenando columna por columna y fila por fila hasta que toda la matriz tiene valores.

3.3. Verificación del P. O. B

El Principio de Optimalidad de Bellman define: “una política es óptima si en un periodo dado, cualesquiera que sean las decisiones precedentes, las decisiones que quedan por tomar constituyen una política óptima independientemente de los resultados de las decisiones precedentes”. En el ejemplo que mostrará a continuación las casillas con lava gastan más energía que las casillas negras, las casillas negras gastan más energía que las casillas amarillas y las casillas amarillas gastan más energía que las casillas blancas.

Todas las subpolíticas son óptimas ya que, aplicando el algoritmo de recuperación de la solución desde la casilla final hasta la casilla inicial, nuestra visión es únicamente la casilla superior, las casilla de la derecha y la casilla de la esquina superior derecha. En el ejemplo mostrado se puede comprobar que, partiendo desde el final, el camino en cada etapa es óptimo:



3.4. Diseño del algoritmo de cálculo de coste óptimo

El procedimiento utilizado es el siguiente:

```
Algoritmo – Cálculo del coste – Sonda Marciana
Begin

P[rows-1][0] = 0;
For i := (rows-2) to 0 do
      P[i][0] := P[i+1][0] + L[i][0];
End;

For j := 1 to (columns-1) do
      P[rows-1][j] := P[rows-1][j-1] + L[rows-1][j];
End;

For i := (rows-2) to 0 do
      For j := 1 to (columns-1) do
            P[i][j]  :=  MIN(P[i+1][j]  +  L[i][j],  P[i][j-1]  +  L[i][j],
P[i+1][j-1] + L[i][j]);
      End;
End;

End Procedure
```

3.5. Diseño del algoritmo de recuperación de la solución

El procedimiento utilizado es el siguiente:

```
Algoritmo Recuperación de la solución – Sonda Marciana
Begin
      i := 0;
      j := (columns-1);
      pair.first = i;
      pair.second = j;
      stack.push(pair);

      While i != (rows-1) OR j != 0
            If i < (rows-1) AND j > 0 then begin
                  If P[i+1][j] <= P[i][j-1] AND P[i+1][j] <= P[i+1][j-1]
      then begin
                        pair.first = i+1;
                        pair.second = j;
                  Else   If   P[i][j-1]   <   P[i+1][j]   AND   P[i][j-1]   <
P[i+1][j-1] then begin
                        pair.first = i;
                        pair.second = j-1;
                  Else
                        pair.first = i+1;
                        pair.second = j-1;
                  End;
            Else If i < (rows-1) then begin
                        pair.first = i+1;
                        pair.second = j;
            Else
                        pair.first = i;
                        pair.second = j-1;
            End;
            stack.push(pair);
            i := pair.first;
            j := pair.second;
      End;
```

```
      While !stack.empty() do
            Writeln(stack.pop());
      End;
End Procedure
```

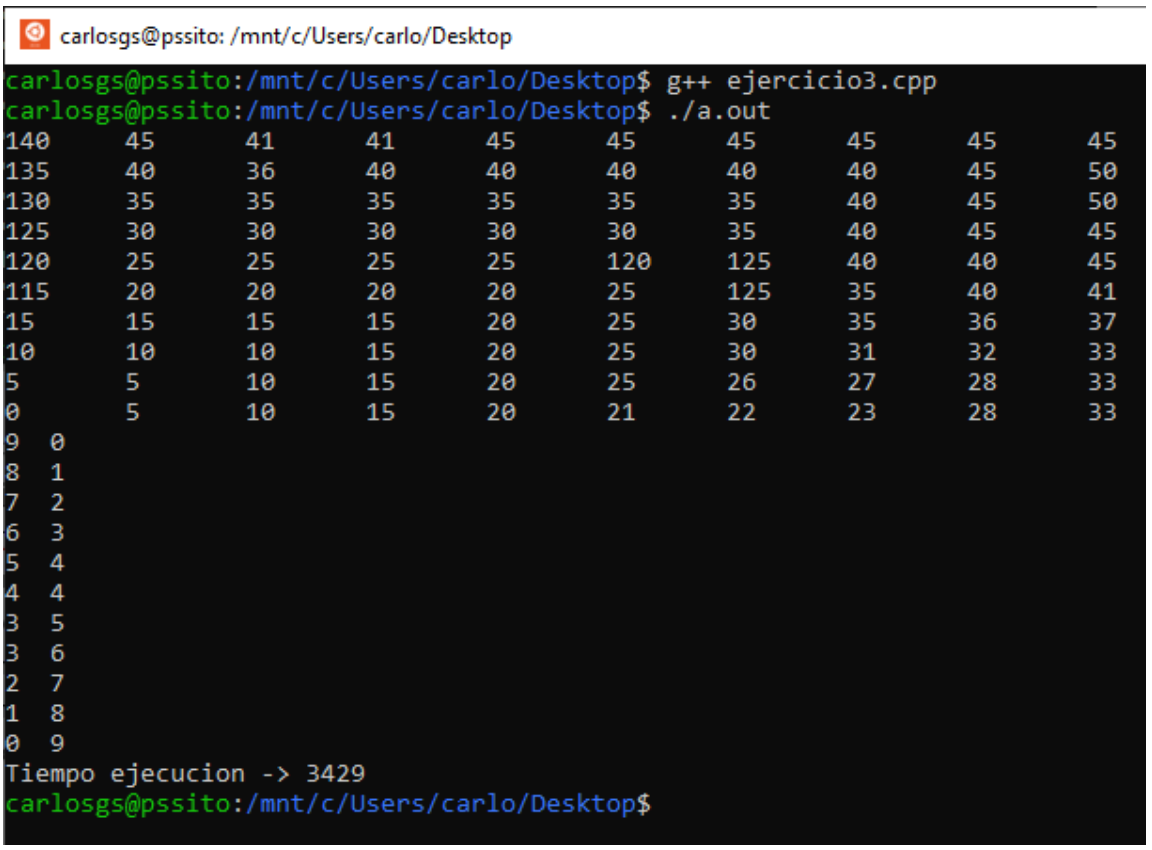
La orden “**Writeln()**” hace alusión a imprimir el contenido en pantalla.

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

```
void CalculoCosteEj3(const vector<vector<int>> &L, vector<vector<int>> &P, int F, int C)
{

    P[F-1][0] = 0;

    for (int i = (F-2); i >= 0; i--)
    {
        P[i][0] = P[i+1][0] + L[i][0];
    }

    for (int j = 1; j < C; j++)
    {
        P[F-1][j] = P[F-1][j-1] + L[F-1][j];
    }

     for (int i = (F-2); i >= 0; i--)
    {
        for (int j = 1; j < C; j++)
        {
           P[i][j] = min(min(P[i+1][j] + L[i][j], P[i][j-1] + L[i][j]), P[i+1][j-1] + L[i][j]);
        }
    }

}
```

```
void RecuperacionEj3 (stack<pair<int,int>> pila, vector<vector<int>> &P, int F, int C)
{

    int i = 0;
    int j = C-1;
    pair<int,int> aux;
    aux.first = i;
    aux.second = j;
    pila.push(aux);

    while(i != (F-1) || j != 0){
       if (i < (F-1) && j > 0){
          if (P[i+1][j] <= P[i][j-1] && P[i+1][j] <= P[i+1][j-1]){
             aux.first = i+1;
             aux.second = j;
          } else if (P[i][j-1] < P[i+1][j] && P[i][j-1] < P[i+1][j-1]){
             aux.first = i;
             aux.second = j-1;
          }else{
             aux.first = i+1;
             aux.second = j-1;
          }
       } else if(i < (F-1)){
          aux.first = i+1;
          aux.second = j;

       }else{
          aux.first = i;
          aux.second = j-1;
       }

        pila.push(aux);
        i = aux.first;
        j = aux.second;

    }
```

En estas imágenes podemos observar la implementación de los algoritmos encargados de calcular la matriz de cálculo coste óptimo y de recuperación de soluciones.



En esta ejecución podemos observar la matriz de cálculo de coste óptimo en la que se encuentran todos los costes mínimos con el que se puede llegar a cada casilla.

Las líneas siguientes a la matriz nos muestran el camino escogido desde la esquina inferior izquierda hasta la esquina superior derecha para maximizar el beneficio.