Semana 1 - Ejercicios de grupo

Métodos algorítmicos en resolución de problemas II
Facultad de Informática - UCM

	Nombres y apellidos de los componentes del grupo que participan	**ID Juezs31**
1	Borja Aday Guadalupe Luis	MAR241
2	Óscar Morujo Fernández	MAR262
3	Roberto Plaza Hermira	MAR267
4	Pablo Martínez	MARP258

Instrucciones:

1. Para editar este documento, es necesario hacer una copia de él. Para ello:
    - Alguien del grupo inicia sesión con la cuenta de correo de la UCM (si no la ha iniciado ya) y accede a este documento.
    - Mediante la opción *Archivo* → *Hacer una copia*, hace una copia del documento en su propia unidad de *Google Drive*.
    - Abre esta copia y, mediante el botón *Compartir* (esquina superior derecha), introduce los correos de los demás miembros del grupo para que puedan participar en la edición de la copia.

2. La entrega se realiza a través del Campus Virtual. Para ello:
    - Alguien del grupo convierte este documento a PDF (dándole como nombre el número del grupo, 1.pdf, 2.pdf, etc…). Desde *Google Docs*, puede hacerse mediante la opción *Archivo* → *Descargar* → *Documento PDF*.
    - Esta misma persona sube el fichero PDF a la tarea correspondiente del *Campus Virtual*. Solo es necesario que uno de los componentes del grupo entregue el PDF.

Juego de tablero

El objetivo de hoy es resolver el **problema 3 Juego de tablero**, del juez automático. Ahí podéis encontrar el enunciado completo del problema.

Se dispone de un tablero cuadrado con casillas que contienen números naturales. Un recorrido válido en el juego empieza en una casilla de la última fila y llega a una casilla de la primera fila teniendo en cuenta que desde una casilla se puede avanzar a una de las tres casillas que tiene justo encima en vertical o diagonal sin salirse del tablero. La puntuación obtenida será la suma de los valores de las casillas por las que se ha pasado en el recorrido. Se desea saber cuál es el valor del recorrido válido con mayor valor y en qué casilla de la última fila comienza.

Solución: (Plantead aquí la recurrencia que resuelve el problema y explicad las razones por las que es mejor resolver el problema utilizando programación dinámica que una implementación recursiva sin memoria.)

Definición: maxBeneficio(i, j) = valor entero que almacena el beneficio máximo al movernos en vertical o diagonal teniendo en cuenta la submatriz de dimensión i x j.

Casos base:
Sea N la dimensión de la matriz o tablero:

maxBeneficio(0,j) = tablero(0, j) 0 <= j < N

Casos recursivos:

- j > 0 y j < N - 1: Nos podemos mover en todas las direcciones.

maxBeneficio(i,j) = max(maxBeneficio(i - 1, j - 1) , maxBeneficio (i - 1, j), maxBeneficio(i - 1, j + 1)) + Cij

- j > 0 y j >= N - 1: No podemos movernos en la diagonal derecha.

maxBeneficio(i,j) = max(maxBeneficio(i - 1, j - 1) , maxBeneficio (i - 1, j)) + Cij

- j <= 0 y j < N - 1: No podemos movernos en la diagonal izquierda.

maxBeneficio(i,j) = max(maxBeneficio (i - 1, j) , maxBeneficio(i - 1, j + 1)) + Cij

Donde Cij es el beneficio de la casilla (i, j) del tablero, es decir tablero[i][j].

Llamada inicial :

maxBeneficio (N - 1, k) donde 0 <= k < N

En este caso al ser recurrencia y no programación dinámica, al llamar a maxBeneficio(N,N) no tendremos todos los subproblemas resueltos, por lo que no sabremos en qué índice de la última fila nos conviene terminar y hay que calcularlo para todos.

Como en el resto de problemas que hemos realizado con programación dinámica, el objetivo está en evitar el cálculo repetido de subproblemas. En este caso, si lo hacemos de forma recursiva repetiremos algunos subproblemas y además podemos provocar un fallo de pila si esta se llena. Es por ello que utilizamos memoria para tener guardados los resultados de los subproblemas y además nos quitamos las llamadas recursivas que pueden llegar a llenar la pila.

Solución: (Escribid aquí las explicaciones necesarias para contar de manera comprensible la implementación del algoritmo de programación dinámica. En particular, explicad si es posible reducir el consumo de memoria y cómo hacerlo. Incluid el código y el coste justificado de la función que resuelve el problema. Extended el espacio al que haga falta.)

Dados lo movimientos posibles, nos podremos desplazar una fila arriba y a 3 diferentes posiciones (2 si estamos en la columna 0 o N-1 de la tabla).
Así, para encontrar el máximo beneficio posible, llenaremos la tabla que contiene el beneficio acumulado empezando desde el destino (última fila) y veremos cual de las posiciones desde la que podríamos haber llegado nos da mayor beneficio.
Una vez hemos llegado al punto de inicio (primera fila), tendremos que recorrerla entera para encontrar el beneficio acumulado máximo y su posición.
De esta manera, vamos resolviendo los subproblemas de manera ascendente, para llegar a la solución final óptima.
A continuación mostramos los algoritmos de programación dinámica ascendente y descendente

Versión 1 : matriz NxN

La complejidad temporal es O(N^2) ya que tenemos que recorrer la matriz entera.

La complejidad espacial es O(N^2) ya que almacenamos el beneficio acumulado de cada posición.

```
void resolver(Matriz<int> tablero, int N) {
    Matriz<int> tabla(N, N);
    for (int i = 1; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (j > 0 && j < N - 1)
                tabla[i][j] = max(max(tabla[i - 1][j - 1], tabla[i - 1][j]), tabla[i - 1][j + 1]) + tablero[i][j];
            else if (j > 0)
                tabla[i][j] = max(tabla[i - 1][j - 1], tabla[i - 1][j]) + tablero[i][j];
            else
                tabla[i][j] = max(tabla[i - 1][j], tabla[i - 1][j + 1]) + tablero[i][j];
        }
    }

    int max = 0;
    int pos = 0;
    for (int j = 0; j < N; j++) {
        if (tabla[N - 1][j] > max) {
            max = tabla[N - 1][j];
            pos = j + 1;
        }
    }

    cout << max << " " << pos << '\n';
}
```

Versión 2: matriz 2xN

Para reducir el uso de memoria, ya que solo nos hace falta la fila actual y fila anterior, podríamos descartar las demás filas y usar una matriz de 2xN lo que reduciría la complejidad espacial respecto a la versión anterior.

La complejidad temporal es O(N^2) ya que tenemos que recorrer la matriz entera.
La complejidad espacial es O(N) .

```
void resolver(Matriz<int> tablero, int N) {
    //vector<int> tabla(N);
    Matriz<int> tabla(2, N);
    for (int j = 0; j < N; j++)
    {
        tabla[0][j] = tablero[0][j];
    }

    for (int i = 1; i < N; i++)
    {
        //int aux1, aux2;
        for (int j = N - 1; j >= 0; j--)
        {
            //aux1 = tabla[j];
            if (j > 0 && j < N - 1)
                //tabla[j] = max(max(tabla[j], tabla[j - 1]), aux2) + tablero[i][j];
                tabla[i % 2][j] = max(max(tabla[(i + 1) % 2][j - 1], tabla[(i + 1) % 2][j]), tabla[(i + 1) % 2][j + 1]) + tablero[i][j];
            else if (j > 0)
                //tabla[j] = max(tabla[j], tabla[j - 1]) + tablero[i][j];
                tabla[i % 2][j] = max(tabla[(i + 1) % 2][j - 1], tabla[(i + 1) % 2][j]) + tablero[i][j];
            else
                //tabla[j] = max(tabla[j], aux2) + tablero[i][j];
                tabla[i % 2][j] = max(tabla[(i + 1) % 2][j], tabla[(i + 1) % 2][j + 1]) + tablero[i][j];
            //aux2 = aux1;
        }
    }

    int max = 0;
    int pos = 0;
    for (int j = 0; j < N; j++) {
        if (tabla[0][j] > max) {
            max = tabla[0][j];
            pos = j + 1;
        }

    }

    cout << max << " " << pos << '\n';
}
```

Versión 3: array de N elementos

La complejidad temporal es O(N^2) ya que tenemos que recorrer la matriz entera.
La complejidad espacial es O(N) también, pero ya solo necesitamos una sola fila y 2 variables auxiliares para guardar los valores necesarios para resolver el siguiente subproblema antes de cambiarlo.

```
void resolver(Matriz<int> tablero, int N) {
    vector<int> tabla(N);
    //Matriz<int> tabla(2, N);
    for (int j = 0; j < N; j++)
    {
        tabla[j] = tablero[0][j];
    }

    for (int i = 1; i < N; i++)
    {
        int aux1, aux2;
        for (int j = N - 1; j >= 0; j--)
        {
            aux1 = tabla[j];
            if (j > 0 && j < N - 1)
                tabla[j] = max(max(tabla[j], tabla[j - 1]), aux2) + tablero[i][j];
                //tabla[i % 2][j] = max(max(tabla[(i + 1) % 2][j - 1], tabla[(i + 1) % 2][j]), tabla[(i + 1) % 2][j + 1]) + tablero[i][j];
            else if (j > 0)
                tabla[j] = max(tabla[j], tabla[j - 1]) + tablero[i][j];
                //tabla[i % 2][j] = max(tabla[(i + 1) % 2][j - 1], tabla[(i + 1) % 2][j]) + tablero[i][j];
            else
                tabla[j] = max(tabla[j], aux2) + tablero[i][j];
                //tabla[i % 2][j] = max(tabla[(i + 1) % 2][j], tabla[(i + 1) % 2][j + 1]) + tablero[i][j];
            aux2 = aux1;
        }
    }

    int max = 0;
    int pos = 0;
    for (int j = 0; j < N; j++) {
        if (tabla[j] > max) {
            max = tabla[j];
            pos = j + 1;
        }

    }

    cout << max << " " << pos << '\n';
}
```

Resolved el problema completo del juez, que ahora debe ser ya muy sencillo.

Número de envío: s31695