MATCOM

DAA

Proyecto Final

Ejercicio 2- Ciel and Gondolas. 321E

Autor:

Karen Dianelis Cantero López (C411) karen.canterolopez@gmail.com

2024

Contents

1 Definición formal del problema

Fox Ciel is in the Amusement Park. And now she is in a queue in front of the Ferris wheel. There are n people (or foxes more precisely) in the queue: we use first people to refer one at the head of the queue, and n-th people to refer the last one in the queue.

There will be k gondolas, and the way we allocate gondolas looks like this:

- When the first gondolas come, the q_1 people in head of the queue go into the gondolas.
- Then when the second gondolas come, the q_2 people in head of the remain queue go into the gondolas.
- ...
- The remain q_k people go into the last (k-th) gondolas.

Note that $q1, q2, ..., qk$ must be positive. You can get from the statement that $\sum_{i=1}^{k} q_i = n$ and $q_i > 0$.

You know, people don't want to stay with strangers in the gondolas, so your task is to find an optimal allocation way (that is find an optimal sequence q) to make people happy. For every pair of people i and j, there exists a value u_{ij} denotes a level of unfamiliar. You can assume $u_{ij} = u_{ji}$ for all i, j ($1 \leq i, j \leq n$) and $u_{ii} = 0$ for all i ($1 \leq i \leq n$). Then an unfamiliar value of a gondolas is the sum of the levels of unfamiliar between any pair of people that is into the gondolas.

A total unfamiliar value is the sum of unfamiliar values for all gondolas. Help Fox Ciel to find the minimal possible total unfamiliar value for some optimal allocation.

1.1 Input

The first line contains two integers n and k ($1 \leq n \leq 4000$ and $1 \leq k \leq min(n, 800)$) — the number of people in the queue and the number of gondolas. Each of the following n lines contains n integers — matrix u, ($0 \leq u_{ij} \leq 9, u_{ij} = u_{ji}$ and $u_{ii} = 0$).

1.2 Output

Print an integer — the minimal possible total unfamiliar value.

2 Reformulando el problema

En esencia, el problema es encontrar una partición óptima de la cola en "k" grupos de modo que se minimice la "distancia" total (desconocimiento) entre los miembros de cada grupo, manteniendo al mismo tiempo la restricción de que cada grupo debe contener al menos un miembro.

3 Solución

3.1 Algoritmo

El algoritmo que proponemos para abordar el problema de dividir n individuos en k grupos minimizando la falta de familiaridad total dentro de cada grupo utiliza una combinación de la técnica DP, y una optimización de divide y vencerás.

Inicialmente, construimos una matriz de suma acumulativa para calcular de manera eficiente la falta de familiaridad total entre cualquier subconjunto de individuos. Este paso le permite al algoritmo determinar rápidamente la falta de familiaridad dentro de cualquier grupo.

Utilizaremos una tabla de programación dinámica f donde cada entrada, $f[i][s]$, representa la falta de familiaridad mínima al dividir los primeros i $(0 \dots i)$ individuos en s grupos. La tabla se inicializa con valores altos para indicar soluciones inicialmente inviables, excepto para casos base que son trivialmente válidos.

Para evaluar el costo de agrupar subconjuntos de individuos, se usarán las sumas acumulativas ya precalculadas, lo que permite cálculos de costos rápidos. La solución emplea una estrategia de divide y vencerás, descomponiendo recursivamente el problema en subproblemas más pequeños. Para cada punto de partición potencial, calcula el costo mínimo considerando todas las divisiones posibles, refinando así la solución de manera eficiente y evitando cálculos redundantes.

Finalmente, después de procesar todas las configuraciones posibles, el algoritmo recupera la falta de familiaridad total mínima para particionar la cola en k grupos de la tabla DP.

4 Demostración de correctitud

4.1 Precálculo Correcto de la falta de familiaridad acumulada

La matriz ‘sum[i][j]‘ representa la suma de los valores de falta de familiaridad en la submatriz desde ‘(1, 1)‘ hasta ‘(i, j)‘ en la matriz de entrada original. La fórmula utilizada para calcular esto es:

$$sum[i][j] = sum[i-1][j] + sum[i][j-1] - sum[i-1][j-1] + x$$

Donde ‘x‘ es el valor de desconocimiento en la posición ‘(i, j)‘ en la entrada.

Caso base de inducción (i = 1, j = 1)

Para el primer elemento ‘(1, 1)‘, la fórmula se simplifica a:

$$sum[1][1] = 0 + 0 - 0 + x = x$$

Esto es correcto, ya que ‘sum[1][1]‘ debería ser igual al valor del elemento único en la posición ‘(1, 1)‘ en la matriz de entrada.

Hipótesis inductiva

Supongamos que para todas las posiciones hasta ‘(i-1, j-1)‘, la matriz de suma acumulativa se calcula correctamente. Ahora probaremos que la fórmula es válida para ‘(i, j)‘.

Paso inductivo

Para la posición ‘(i, j)‘:

- sum[i-1][j]: Contiene la suma de la submatriz de ‘(1, 1)‘ a ‘(i-1, j)‘. Esto representa la suma de todos los elementos en la submatriz de arriba de la fila actual.
- sum[i][j-1]: Contiene la suma de la submatriz de ‘(1, 1)‘ a ‘(i, j-1)‘. Esto representa la suma de todos los elementos en la submatriz a la izquierda de la columna actual.
- sum[i-1][j-1]: Contiene la suma de la submatriz de ‘(1, 1)‘ a ‘(i-1, j-1)‘, que es la superposición entre los dos términos anteriores.

Por lo tanto, al sumar ‘sum[i-1][j]‘ y ‘sum[i][j-1]‘, restar ‘sum[i-1][j-1]‘ (para evitar el doble conteo) y, finalmente, sumar el valor ‘x‘ en la posición ‘(i, j)‘, se obtiene la suma total de la submatriz desde ‘(1, 1)‘ hasta ‘(i, j)‘.

Esto demuestra que la fórmula calcula correctamente la suma acumulada para cada posición ‘(i, j)‘.

4.2 Funcion calc

En nuestra solución, hacemos uso de una función 'calc(l,r)' la cual tiene como objetivo calcular la falta de familiaridad en un grupo que comienza en l y termina en r.

La función luce:

$$calc(l, r) = (sum[r][r] - sum[l-1][r] - sum[r][l-1] + sum[l-1][l-1])//2$$

- sum[r][r]: Esta expresión representa la suma acumulada de todos los elementos desde la esquina superior izquierda (1,1) hasta la posición (r,r). Básicamente, incluye todo lo que se ha sumado hasta la posición r en ambas dimensiones.
- sum[l-1][r]: Esta resta elimina de la suma acumulada cualquier valor que esté por encima de la fila l. Es decir, elimina la parte de la suma que no está incluida en el subgrupo de personas desde l hasta r.
- sum[r][l-1]: Similarmente, esta resta elimina los valores que están a la izquierda de la columna l, lo que asegura que estamos calculando solo la suma de los valores dentro del rango de columnas l a r.
- sum[l-1][l-1]: Esta parte se ha restado dos veces en las dos restas anteriores (una vez en la fila y una vez en la columna), por lo que es necesario sumarla de nuevo para corregir la doble resta.

Dividimos entre dos ya que, como la matriz es simétrica, hemos sumado la falta de familiaridad entre los pares [l,r] y [r,l]

Concluimos que la función calc(l, r) calcula correctamente la suma de familiaridad para el subgrupo [l,r], ya que usa correctamente la matriz de sumas acumuladas para extraer la suma de la submatriz de manera eficiente, y divide por 2 para corregir la doble cuenta de los pares de familiaridad, dado que la matriz es simétrica.

4.3 Subestructura Óptima

Supongamos que existe una solución óptima .

Sea $f[i][s]$ la solución óptima para dividir las primeras i personas en s grupos. Esta solución óptima divide la secuencia en s grupos, donde el último grupo comienza en alguna posición p y llega hasta la persona i.

Por lo tanto, la falta de familiaridad óptima para esta configuración es:

$$f[i][s] = f[p-1][s-1] + calc(p, i)$$

Aquí, $f[p-1][s-1]$ es la falta de familiaridad óptima para dividir las primeras $p-1$ personas en $s-1$ grupos, y $calc(p, i)$ es la falta de familiaridad para el grupo de p a i.

Si la división general es óptima, entonces la división de las primeras $p-1$ personas en $s-1$ grupos también debe ser óptima; ya que si hubiera una mejor manera de dividir a las primeras $p-1$ personas en $s-1$ grupos, entonces podríamos reemplazar $f[p-1][s-1]$ con un mejor valor, lo que llevaría a una menor falta de familiaridad general para $f[i][s]$. Esto contradeciría la suposición de que nuestra solución era óptima.

Por lo tanto, la división óptima para las primeras i personas en s grupos se basa en la división óptima de las primeras $p-1$ personas en $s-1$ grupos.

Al considerar cada posible punto de división p, nos aseguramos de encontrar la mejor manera posible de dividir a las primeras i personas en s grupos. La solución para $f[i][s]$ se basa, por lo tanto, en soluciones óptimas para subproblemas más pequeños ($f[p-1][s-1]$), lo que demuestra la propiedad de subestructura óptima.

4.4 Subproblemas Superpuestos

La relación de recurrencia para el DP es:

$$f[i][s] = f[p-1][s-1] + calc(p, i)$$

Esto significa que para calcular $f[i][s]$, necesitamos examinar múltiples subproblemas $f[p-1][s-1]$, donde $1 \leq p \leq i$.

Ahora, consideremos los diferentes valores de f[i][s] que podrían requerir que se calcule el mismo subproblema f[p-1][s-1].

Supongamos que estamos calculando f[i][s], que utiliza los subproblemas f[p-1][s-1] para varios valores de p (donde $1 \leq p \leq i$). Simultáneamente, podríamos estar calculando f[i+1][s], que también requerirá los subproblemas f[p-1][s-1] para $1 \leq p \leq i+1$. En particular, tanto f[i][s] como f[i+1][s] podrían utilizar el subproblema f[p-1][s-1] para el mismo valor de p.

Ejemplificación

Supongamos que tenemos que calcular f[5][3] (las primeras 5 personas en 3 góndolas) y f[6][3] (las primeras seis personas en 3 góndolas):

Para calcular f[5][3], podríamos tener que evaluar f[4][2], f[3][2], f[2][2], y así sucesivamente. Para calcular f[6][3], también tendremos que evaluar f[5][2], f[4][2], f[3][2], y así sucesivamente. Observe que los subproblemas f[4][2] y f[3][2] aparece ntanto en el cálculo de f[5][3] como de f[6][3]. Esto significa que f[4][2] y f[3][2] son subproblemas superpuestos porque se reutilizan en múltiples evaluaciones diferentes de f[i][s] para diferentes valores de i.

Queda demostrado que nuestro problema contiene subproblemas superpuestos.

4.5 Correctitud de DP

Para demostrar la correctitud de nuestra solución que usa DP, necesitamos demostrar que la recurrencia modela correctamente el problema y que la solución calculada es óptima.

Caso base:

En el caso base, cuando $s = 0$ (es decir, no se forman grupos), la falta de familiaridad no está definida, por lo que establecemos:

$$f[i][0] = \infty \quad \forall i \geq 1$$

Esto tiene sentido porque es imposible dividir i personas en 0 grupos.

Cuando $i = 0$ y $s = 0$, definimos:

$$f[0][0] = 0$$

Esto refleja el hecho de que si no tenemos personas ni grupos, la falta de familiaridad es 0.

Relación de recurrencia:

La fórmula de recurrencia es:

$$f[i][s] = \min_{1 \leq p \leq i} (f[p-1][s-1] + \text{calc}(p, i))$$

Estamos determinando la falta de familiaridad óptima para dividir las primeras i personas en s grupos considerando cada posible punto de partida p para el último grupo. La falta de familiaridad total para esta configuración es $f[p-1][s-1] + calc(p, i)$. Al probar todos los posibles p, se garantiza que tomemos el mínimo de todos para encontrar la partición óptima.

Hipótesis inductiva:

Supongamos que la recurrencia se cumple para todos los valores de i' y s' tales que $i' < i$ y $s' < s$. Queremos demostrar que la recurrencia también se cumple para f[i][s].

Por la hipótesis inductiva, los subproblemas f[p-1][s-1] ya se calcularon correctamente y representan la falta de familiaridad mínima para dividir las primeras $p-1$ personas en $s-1$ grupos.

Dado esto, calculamos $f[i][s]$ considerando todas las formas posibles de formar el último grupo (comenzando en la posición p y terminando en la posición i), y tomamos el mínimo de estas posibilidades. Dado que la falta de familiaridad del último grupo se calcula correctamente con $calc(p, i)$ y la falta de familiaridad de las primeras $p-1$ personas se calcula correctamente con $f[p-1][s-1]$, la relación de recurrencia proporciona la falta de familiaridad mínima correcta para dividir las primeras i personas en s grupos.

Optimalidad:

Supongamos que existe una forma óptima de dividir las primeras i personas en s grupos. En esta solución óptima, supongamos que el último grupo comienza en la posición p. Entonces, la solución óptima se puede escribir como:

$$f[i][s] = f[p-1][s-1] + \text{calc}(p, i)$$

Por la hipótesis inductiva, sabemos que $f[p-1][s-1]$ es la falta de familiaridad óptima para dividir las primeras $p-1$ personas en $s-1$ grupos. Dado que estamos considerando cada valor posible de p en la recurrencia, el valor p es uno de los candidatos para el mínimo, y la recurrencia lo seleccionará porque conduce al menor grado de desconocimiento.

Por lo tanto, la recurrencia calcula correctamente el grado de desconocimiento mínimo para dividir las primeras i personas en s grupos.

Terminación

El algoritmo termina porque evalúa cada subproblema posible para cada par

$$(i, s)$$

donde $1 \leq i \leq n$ y $1 \leq s \leq k$. Dado que estamos construyendo la solución para i y s mayores resolviendo subproblemas más pequeños, la tabla DP se llena en una cantidad finita de tiempo y finalmente calculamos $f[n][k]$, que da el grado de desconocimiento mínimo para dividir todas las n personas en k grupos.

Por lo tanto, la solución de programación dinámica es correcta y $f[n][k]$ proporciona el nivel mínimo óptimo de desconocimiento para dividir n personas en k grupos.

4.6 Divide Y Vencerás

Para demostrar formalmente que la optimización de divide y vencerás se utiliza correctamente en la solución, debemos demostrar que la optimización reduce la complejidad temporal sin afectar la correctitud de la solución ya demostrada.

La optimización se aplica para acelerar el cálculo de la recurrencia de DP:

$$f[i][s] = \min_{1 \leq p \leq i} (f[p-1][s-1] + \text{calc}(p, i))$$

Sin optimización, esta recurrencia requeriría iterar sobre todos los puntos de partición posibles p, lo que llevaría a una complejidad temporal de $O(n^2k)$, donde n es el número de personas y k es el número de grupos.

La optimización de divide y vencerás reduce esta complejidad temporal a $O(kn \log n)$ al explotar la **propiedad de monotonía de la recurrencia**. Esto es posible ya que el punto de división óptimo p para $f[i][s]$ tiene una estructura que permite restringir la búsqueda del mínimo, utilizando el método divide y vencerás.

Probaremos la correctitud de la optimización divide y vencerás estableciendo lo siguiente:

1. Propiedad de monotonía de los puntos de división óptimos.
2. La recursión divide y vencerás mantiene la correctitud y reduce el tamaño del problema.

4.6.1 Propiedad de Monotonía

La clave para aplicar la optimización de divide y vencerás radica en demostrar que la recurrencia tiene la propiedad de monotonía de los puntos de partición óptimos.

Para la recurrencia:

$$f[i][s] = \min_{1 \leq p \leq i} (f[p-1][s-1] + \text{calc}(p, i))$$

Sea $opt(i, s)$ el punto de partición óptimo para $f[i][s]$. La propiedad de monotonía establece que:

$$opt(i, s) \leq opt(i+1, s)$$

En otras palabras: el punto de partición óptimo para ‘f[i][s]‘ no es mayor que el punto de partición óptimo para ‘f[i+1][s]‘. Esto significa que a medida que nos movemos de ‘i‘ a ‘i+1‘, el punto de inicio óptimo para el último grupo solo puede moverse hacia la derecha o permanecer igual.

De manera intuitiva, el problema tiene monotonía ya que:

- Se mantiene que $calc(p, i) < calc(p, i + 1)$ porque cuantas más personas incluimos en el último grupo, la falta de familiaridad o aumenta o se mantiene igual.
- Debido a que agregar más personas aumenta el costo de falta de familiaridad o lo mantiene invariante, el punto de división óptimo para f[i+1][s] generalmente está en el punto de división para f[i][s] o más allá de él. Esto significa que una vez que se determina el mejor lugar para dividir para un subproblema más pequeño (i personas), el punto de división óptimo para un subproblema más grande (i+1 personas) generalmente se moverá hacia adelante. El punto de partición óptimo no ”saltará hacia atrás”, ya que aumentaría el costo.

De manera formal, podemos hacer una diferenciación de casos:

1. Caso 1: La persona i+1 está en el mismo grupo que i

   Si $i + 1$ se incluye en el grupo de i, entonces $opt(i, s) = opt(i + 1, s)$

2. Caso 2: La persona i+1 pertenece a un nuevo grupo

   El costo de este nuevo grupo depende de la cantidad de personas que hayan en este. Para minimizar el costo de este nuevo grupo, el nuevo punto de partición debe ser lo más cercano a $i + 1$ como sea posible. Esto es debido a que mientras más personas se agreguen a un grupo, el costo total de falta de familiaridad tiende a aumentar. Esto implica que $opt(i + 1, s) > opt(i, s)$

Como resultado, el punto de partición óptimo para ‘f[i+1][s]‘ no se puede mover a la izquierda del punto de partición óptimo para ‘f[i][s]‘, lo que garantiza que se mantenga la propiedad de monotonía.

4.6.2 Recursión de divide y vencerás

La optimización de divide y vencerás aprovecha la propiedad de monotonía para restringir la búsqueda del punto de partición óptimo. En lugar de examinar todos los ‘p‘ posibles para cada ‘i‘, utilizamos un enfoque de divide y vencerás para encontrar los puntos de partición óptimos de manera más eficiente.

Estructura de divide y vencerás:

La tabla DP se llena utilizando el siguiente enfoque recursivo:

```
void solve(int s, int l, int r, int L, int R){
    if (l > r) return;

    int mid = (l + r) >> 1;
    int p_optimal = L;
    f[mid][s] = INF;

    // Busque el punto de partición óptimo dentro del rango restringido [L, R]
    for (int p = L; p <= min(mid, R); p++) {
        int ans = f[p-1][s-1] + calc(p, mid);
        if (ans < f[mid][s]) {
            f[mid][s] = ans;
            p_optimal = p;
        }
    }

    // Resuelva recursivamente las mitades izquierda y derecha
    solve(s, l, mid - 1, L, p_optimal);
    solve(s, mid + 1, r, p_optimal, R);
}
```

- La función ‘solve(s, l, r, L, R)‘ calcula el valor de DP ‘f[mid][s]‘ para ‘mid = (l + r) / 2‘ buscando el punto de partición óptimo ‘p‘ en el rango ‘[L, R]‘.

- Después de encontrar el punto de partición óptimo ‘p_optimal‘ para ‘mid‘, la función calcula recursivamente los valores de DP para la mitad izquierda (‘l‘ a ‘mid-1‘) y la mitad derecha (‘mid+1‘ a ‘r‘), restringiendo el rango de búsqueda de puntos de partición según la propiedad de monotonía.

Por qué la recursión es correcta:

- Restricción de búsqueda:

  La propiedad de monotonía asegura que si ‘p_optimal‘ es el punto de partición óptimo para ‘f[mid][s]‘, entonces el punto de partición óptimo para la mitad izquierda (‘f[l..mid-1][s]‘) debe estar en el rango ‘[L, p_optimal]‘, y el punto de partición óptimo para la mitad derecha (‘f[mid+1..r][s]‘) debe estar en el rango ‘[p_optimal, R]‘.

- Divide y vencerás:

  la recursión divide el problema en subproblemas más pequeños (resolviendo para ‘mid‘, luego resolviendo para las mitades izquierda y derecha), lo que garantiza que toda la tabla DP se complete correctamente al considerar todos los puntos de partición posibles dentro de los rangos restringidos.

- Caso base:

  la recursión termina cuando $l > r$, lo que significa que no hay más personas para particionar en el subproblema actual. El caso base garantiza que la recursión finalmente se complete y llene la tabla DP para todos los ‘i‘ y ‘s‘.

4.6.3 Reducción de la complejidad temporal

Sin la optimización de divide y vencerás, la recurrencia requeriría evaluar cada punto de partición posible ‘p‘ para cada combinación de ‘i‘ y ‘s‘, lo que lleva a una complejidad temporal de $O(n^2k)$.

Con divide y vencerás, los puntos de partición óptimos se encuentran en un tiempo de $O(\log n)$ para cada subproblema porque estamos dividiendo a la mitad el espacio de búsqueda para cada llamada recursiva. Por lo tanto, la complejidad temporal general se reduce a $O(nk \log n)$.

4.6.4 Preservación de correctitud

La correctitud de la solución DP se conserva porque:

1. El enfoque de dividir y vencer garantiza que todos los puntos de partición posibles se consideren mediante recursión.

2. Los rangos de búsqueda restringidos son válidos debido a la propiedad de monotonía, lo que garantiza que no se pase por alto ningún punto de partición óptimo.

Por lo tanto, la optimización de divide y vencerás no afecta la correctitud de la solución, sino que simplemente reduce la complejidad temporal al explotar la estructura del problema.

5 Complejidad

5.1 Temporal

1. Preprocesamiento: El cálculo de las sumas acumulativas 'sum[i][j]' lleva $O(n^2)$ tiempo.
2. Inicialización de la tabla de programación dinámica: La inicialización de la tabla 'f' lleva $O(n)$ tiempo.
3. Función recursiva 'solve': La función 'solve' tiene una complejidad temporal de $O(n \log n)$. Esto se debe a que:
   - El bucle interno itera desde 'L' hasta 'min(mid, R)', que es $O(n)$ en el peor de los casos.
   - Dentro del bucle interno, hay operaciones de tiempo constante.
   - Las llamadas recursivas a 'solve' reducen a la mitad el tamaño del intervalo, pero como es una llamada recursiva, es menos trivial ver la complejidad.
   - Su $T(n)$ es $T(n) = 2T(\frac{n}{2}) + n$.
     Usando el Teorema Maestro:
     - a=2
     - b=2
     - $f(n) = n$
     - c=1
     - $f(n)$ sigue la estructura $f(n) = \theta(n^c \log^k n)$ con c=1 y k=0
       Veamos si cumple la condición del segundo caso:

       $$\log_b a = \log_2 2 = 1 = c$$

       Por lo tanto, del segundo caso del teorema maestro se deduce:

       $$T(n) = \theta(n^{\log_b a} \log^{k+1} n) = \theta(n \log n)$$

   La función 'solve' se llama k veces

Por lo tanto, la complejidad temporal total del código es $O(n^2) + O(n) + O(knlogn)$, que se simplifica a $O(n^2 + knlogn)$.

5.2 Espacial

La complejidad espacial del código es $O(n^2)$.

- Matriz *sum*: Esta matriz tiene dimensiones n x n, por lo que requiere $O(n^2)$ de espacio.
- Matriz f: Esta matriz tiene dimensiones n x k, por lo que requiere $O(nk)$ de espacio.
- Otras variables: Las variables 'n', 'k', 'INF', 'mid', 'p', 'ans' y 'x' requieren espacio constante.

Dado que $O(n^2)$ domina a $O(nk)$, la complejidad espacial general del código es $O(n^2)$.