Soluciones OIFem 2020
Entrenamiento 3

Dividiendo

Teoría

- Vectores
- Factores

Solución

Si un número a es factor de otro número b, $a \leq b$. Por lo tanto, en el caso de que un número de la lista sea factor del resto, este debe de ser el elemento más pequeño del grupo. Encontramos este mínimo y vamos valor a valor del vector probando para ver si se da el caso de que divide a todos los demás.

Código

C++

```cpp
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

bool divide(vector<int> & numeros) {
    int n = (int) numeros.size();
    int minimo = *min_element(numeros.begin(), numeros.end()); // Usamos
    ↪ min_element para encontrar al número más pequeño de la lista
    for (int i = 0; i < n; i++) {
        if (numeros[i] % minimo != 0)
            return false;
    }
    // Sabemos que numeros[i] % minimo == 0 para todos los i entre 0 y n-1
    return true;
}
```

```cpp
int main() {
    int t, n;
    cin >> t;
    vector<int> numeros;
    while(t--) {
        cin >> n;
        numeros = vector<int>(n);
        for (int i = 0; i < n; i++)
            cin >> numeros[i];
        if (divide(numeros))
            cout << "SI\n";
        else
            cout << "NO\n";
    }
    return 0;
}
```

Pizzas

Teoría

- Vectores

Solución

Para cada caso de entrada, recogemos los datos de todos los comensales, guardando un total del número de trozos de pizza pedidos de cada tipo de topping. Comprobamos que el número pedido sea un múltiplo de 8 para cada topping, correspondiendo esto a un número entero de pizzas.

Código

C++

```cpp
#include <vector>
#include <iostream>
using namespace std;

bool sinSobras(vector<int> & trozos) {
    // comprobamos que para cada tipo de pizza haya un número de trozos pedido que corresponda a un entero de pizzas
    for (int cantidad: trozos) {
        if (cantidad % 8)
            return false;
    }
    return true;
}

int main() {
    int N, M, numTrozos, tipoDePizza;
    vector<int> totalTrozos;
    while(cin >> N >> M) {
        totalTrozos.assign(M, 0); // total de trozos pedidos de cada tipo
        for (int comensal = 0; comensal < N; comensal++) {
            // pedimos los datos de los trozos que quiere el comensal y los sumamos al total
            cin >> numTrozos;
            for (int porcion = 0; porcion < numTrozos; porcion++) {
                cin >> tipoDePizza;
                totalTrozos[tipoDePizza]++;
            }
        }
        if (sinSobras(totalTrozos)) cout << "SI\n";
        else cout << "NO\n";
    }
    return 0;
}
```

Palabras abcd

Teoría

- Método de la ventana deslizante

Solución

Utilizando el método de la ventana deslizante, llevamos la cuenta de los últimos 4 caracteres leídos de la palabra, sumando 1 al total cada vez que esta secuencia de 4 caracteres es alguna permutación de abcd.

Código

C++

```cpp
#include <vector>
#include <iostream>
using namespace std;

void rotar(string & secuencia) {
    // desplaza todos los caracteres de la secuencia 1 lugar hacia atrás
    for (int i = 0; i < (int) secuencia.length()-1; i++)
        secuencia[i] = secuencia[i+1];
}

int main() {
    int casos, substrings, N;
    vector<int> cantidadLetras;
    string secuencia;
    char letra;
    cin >> casos;

    for (int i = 0; i < casos; i++) {
        cin >> N;
        // reiniciamos las variables
        cantidadLetras.assign(4, 0); // [número de 'a', número de 'b', número de 'c', número de 'd']
        substrings = 0;
        secuencia.clear(); // guarda la ventana con los últimos 4 caracteres leídos

        // recogemos los primeros 4 caracteres (o menos si la palabra mide menos)
        for (int i = 0; i < min(4, N); i++) {
            cin >> letra;
            cantidadLetras[letra - 'a']++;
            secuencia.push_back(letra);
        }
        // hacemos la primera comprobación
        if (cantidadLetras[0] == 1 && cantidadLetras[1] == 1 && cantidadLetras[2] == 1 && cantidadLetras[3] == 1)
            substrings++;
```

```
        for (int i = 4; i < N; i++) {
            // vamos caracter a caracter actualizando la secuencia y comprobando
            ↪  si es buena
            cin >> letra;
            cantidadLetras[secuencia[0] - 'a']--;
            rotar(secuencia);
            secuencia[3] = letra;
            cantidadLetras[letra - 'a']++;
            // no hace falta comprobar si cantidadLetras[3] == 1 ya que la
            ↪  longitud de la ventana es 4 y 4-3 = 1
            if (cantidadLetras[0] == 1 && cantidadLetras[1] == 1 &&
            ↪  cantidadLetras[2] == 1)
                substrings++;
        }
        cout << substrings << '\n';
    }
    return 0;
}
```

Multiplicación de matrices

Teoría

- Programación dinámica

Solución

Este es un conocido clásico de programación dinámica. La solución se basa en poner paréntesis en todos los sitios posibles, guardando la mejor solución.

Para lograr esto, rompemos el problema mayor en problemas más pequeños. Empezamos con un vector de longitud k. Hay $k-1$ lugares donde poner unos paréntesis que separen la cadena en dos (por ejemplo: (0)(123), (01)(23) y (012)(3) son las 3 formas de partir esta cadena de longitud 4).

Las dimensiones de la matriz correspondiente a la primera sección son filas[0]*columnas[m-1] y las de la segunda sección son filas[m]*columnas[k-1], sin olvidarnos de que columnas[m-1] = filas[m]. Esto significa que, una vez obtenido el resultado de ambas secuencias, multiplicar estas dos matrices tiene un coste de filas[0]*columnas[m-1]*columnas[k-1]. Por lo tanto, el coste total C de partir esta cadena de longitud k en una de longitud m y una segunda de longitud $k-m$ es igual a:

$$C[0..k-1](m) = filas[0] * columnas[m-1] * columnas[k-1] + C[0..m-1] + C[m..k-1]$$

El coste $C[0..k-1]$ resultante para la cadena de longitud k es el mínimo $C[0..k-1](m)$ de las posibles formas de elegir m.

Código

C++

```cpp
#include <vector>
#include <iostream>
using namespace std;

int operaciones(vector<int> & dims) {
    // implementamos la DP explicada arriba
    int n = (int) dims.size(), j;
    vector<vector<int>> DP(n, vector<int>(n, 1e9));
    for (int i = 0; i < n; i++)
        DP[i][i] = 0;
    for (int longitud = 2; longitud < n; longitud++) {
        for (int i = 1; i < n - longitud + 1; i++) {
            j = i + longitud - 1;
            for (int k = i; k < j; k++) {
                DP[i][j] = min(DP[i][j], DP[i][k] + DP[k+1][j] +
                ↪  dims[i-1]*dims[k]*dims[j]);
            }
        }
    }
    return DP[1][n-1];
}
```

```cpp
int main() {
    int casos, n, a, b;
    cin >> casos;
    vector<int> dims;
    while(casos--) {
        cin >> n;
        // guardamos las n+1 dimensiones (el número de filas de todas y las
        ↪  columnas de la última)
        dims = vector<int>(n+1);
        for (int i = 0; i < n; i++) {
            cin >> a >> b;
            dims[i] = a;
        }
        dims[n] = b;
        cout << operaciones(dims) << '\n';
    }
    return 0;
}
```

Flipping Rectangles

Teoría

- Programación dinámica
- Operadores bitwise (&)

Solución

El color de una casilla con coordenadas (i, j), llamando a la esquina superior izquierda del tablero la coordenada (0, 0), solo puede ser alterado seleccionando una casilla con coordenadas (a, b), donde $a \leq i$ y $b \leq j$.

Por lo tanto, empezamos en la casilla con coordenadas (0, 0). Si esta necesita ser cambiada de color, efectuamos el cambio en esta misma, trasladando el resultado al resto de columnas con coordenadas superiores a 0. Pero esto afecta al resto de casillas de su fila, por lo que pasamos a comprobar el estado de la casilla (0, 1) y así sucesivamente hasta terminar la fila y pasar a la siguiente.

En todo momento, tenemos guardado en el vector *cambios* cuántos cambios se han hecho en una columna, pudiendo saber en cuanto recibimos el valor del color original de un recuadro, su color actual tras los cambios.

La respuesta final será el total de los cambios efectuados una vez lleguemos a la casilla (filas-1, columnas-1).

Código

C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

void movimientos(int filas, int columnas) {
    int recuadro, respuesta = 0;
    vector<int> cambios(columnas, 0);
    for (int f = 0; f < filas; f++) {
        for (int c = 0; c < columnas; c++) {
            cin >> recuadro;
            if ((cambios[c] + recuadro)&1) {
                // la casilla en la posición (f, c) necesita un cambio de color
                respuesta++; // incrementamos el total de cambios hechos
                for (int i = c; i < columnas; i++)
                    cambios[i]++; // trasladamos el cambio al resto de columnas
            }
        }
    }
    cout << respuesta << '\n';
}
```

```cpp
int main() {
    int t, n, m;
    cin >> t;
    while(t--) {
        cin >> n >> m;
        movimientos(n, m);
    }
    return 0;
}
```

Memory

Teoría

- Vectores
- Problemas interactivos

Solución

Para conseguir los 100 puntos, basta con una solución que no levante una carta más de 100 veces. Esto significa que cada carta se puede levantar hasta dos veces, lo que nos permite resolver el problema con una función que hace dos pases por las cartas.

Primero, levantamos cada carta una vez, guardando la letra que sale. Para saber qué dos cartas tienen cada letra, tenemos dos arrays, uno con una de las posiciones que tiene la carta y otro con la suma de las dos posiciones, de la cual podemos obtener mediante una simple resta la posición de la segunda carta. De esta forma, nos ahorramos el comprobar si ya se ha encontrado esa letra cada vez que levantemos una carta.

Después, hacemos otro pase por las cartas, levantando juntas las que sabemos que tienen la misma letra debajo. Así, en 100 llamadas exactas a `faceup (C)` aseguramos el resultado.

Código

C++

```cpp
#include "memory.h"
#include "grader.h"

void play() {
        int pos[25]; // pos[i] contiene una de las posiciones de esa letra
        int sumPos[25] = {0}; // sumPos[i] contiene la suma de las 2 posiciones de esa letra
        int curChar;
        for (int i = 1; i <= 50; i++) {
                // levantamos todas las cartas y guardamos el resultado
                curChar = faceup(i) - 'A';
                pos[curChar] = i;
                sumPos[curChar] += i;
        }
        for (int i = 0; i < 25; i++) {
                // levantamos las parejas de cartas que contienen la misma letra
                faceup(pos[i]);
                faceup(sumPos[i] - pos[i]);
        }
}
```

Combo

Teoría

- Strings
- Problemas interactivos

Solución

Primero, obtenemos el primer carácter de S presionando la secuencia `"AB"`. Tras este primer intento, hay tres posibilidades:

- `press("AB")` $= 2$: S empieza por `"AB"`
- `press("AB")` $= 1$: S empieza por `'A'` o `'B'`, por lo que resolvemos esta duda.
- `press("AB")` $= 0$: S empieza por `'X'` o `'Y'`, por lo que resolvemos esta duda.

Tras dos invocaciones a `press()`, ya conocemos el primer carácter de S, que sabemos que no se vuelve a repetir. Digamos que este primer carácter es `'Y'`. Una solución que nos daría 30 puntos de los 100 sería ir carácter a carácter probando si este es `'A'` y si es el caso proceder, repetir lo mismo para ver si este es `'B'` y si no asumir que es `'X'`.

Para llegar a los 100 puntos, hay que aprovecharnos del límite de `press()` de longitud $4N$. Guardamos en un vector `pos` los tres posibles caracteres para el resto de posiciones de S, en formato `string`, que nos será más cómodo de manejar. En cada iteración, construiremos la siguiente palabra: $S + pos[0] + pos[1] + S + pos[0] + pos[1] + S + pos[0] + pos[2] + S + pos[1]$, donde S corresponde al prefijo que ya hemos averiguado de la solución final.

Si el siguiente carácter es $pos[0]$, `press()` nos devolverá la longitud del prefijo S más dos, ya que hemos incluido las tres posibilidades con $pos[0]$ y el siguiente carácter.

Si el siguiente carácter es $pos[1]$, `press()` nos devolverá la longitud del prefijo S más uno.

Finalmente, si el siguiente carácter es $pos[2]$, `press()` nos devolverá la longitud del prefijo S, ya que no hemos mandado en ninguna parte de nuestra `string` la palabra $S + pos[2]$.

Repetimos este proceso hasta que solo falte el último carácter de S. En este momento, ya no podremos obtener el último carácter de esta forma, ya que no hay una posición siguiente de la que aprovecharnos. Obtenemos este carácter con ensayo y error.

Código

C++

```cpp
#include "combo.h"
#include <vector>
using namespace std;

string guess_sequence(int N) {
	string S;

	// Encontramos el primer carácter
	char first;
	int x = press("AB");
	if (x == 2) {
		S = "AB";
		first = 'A';
	} else if (x == 1) {
		x = press("A");
		if (x) {
			first = 'A';
			S = "A";
		}
		else {
			first = 'B';
			S = "B";
		}
	} else {
		x = press("X");
		if (x) {
			first = 'X';
			S = "X";
		} else {
			first = 'Y';
			S = "Y";
		}
	}
	vector<string> pos;
	if (first != 'A') pos.push_back("A");
	if (first != 'B') pos.push_back("B");
	if (first != 'X') pos.push_back("X");
	if (first != 'Y') pos.push_back("Y");
	// Procedemos con el resto de la palabra
	while((int) S.length() < N) {
		if (3*((int) S.length() + 2) + (int) S.length() + 1 <= 4*N) {
			x = press(S + pos[0] + pos[0] + S + pos[0] + pos[1] + S +
			↪  pos[0] + pos[2] + S + pos[1]);
			if (x == (int) S.length() + 2) S += pos[0];
			else if (x == (int) S.length() + 1) S += pos[1];
			else S += pos[2];
		} else {
		    // Encontramos el último carácter
			x = press(S + pos[0]);
			if(x == (int) S.length() + 1) {
				S += pos[0];
				continue;
			}
			x = press(S + pos[1]);
			if (x == (int) S.length() + 1) S += pos[1];
			else S += pos[2];
		}
	}
	return S;
}
```

Montañas rusas

Teoría

- Programación dinámica

Solución

Guardaremos un vector DP con todas las posibilidades que nos pueden pedir de altura y longitud. El número de montañas rusas de longitud $curL$, altura $curH$ y altura máxima $maxH$ (`DP[curL][curH][maxH]`) se compone de tres partes:

- `DP[curL-1][curH+1][maxH]`: el número de montañas rusas con un tramo menos de longitud, con una unidad más de altura y la misma altura máxima (que es el mismo número de montañas rusas que las de longitud $curL$ y altura $curH$ con un último tramo descendente.
- `DP[curL-1][curH-1][maxH]`: el número de montañas rusas con un tramo menos de longitud, con una unidad menos de altura y la misma altura máxima (que es el mismo número de montañas rusas que las de longitud $curL$ y altura $curH$ con un último tramo ascendente.
- `DP[curL-1][curH-1][maxH-1]`: el número de montañas rusas con un tramo menos de longitud, con una unidad menos de altura y una altura máxima una unidad inferior. Esta es la forma de tener en cuenta todas las montañas rusas con alturas que nunca llegan a $maxH$, ya que ya las habremos contado en las de altura que no supere $maxH - 1$.

Código

C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

int MOD = 1000000007;
vector<vector<vector<int>>> DP;

int recur(int curL, int curH, int maxH) {
    int res = 0;
    if (DP[curL][curH][maxH] == -1) {
        if (curL == 0)
            return 0;

        if (curH < maxH) {
            res += recur(curL-1, curH+1, maxH) % MOD; // tramo descendente
            res %= MOD;
        }

        if (curH > 0) {
            res += recur(curL-1, curH-1, maxH) % MOD; // tramo ascendente
            res %= MOD;
            if (curH == maxH) {
                res += recur(curL-1, curH-1, maxH-1) % MOD; // alturas máximas más pequeñas
                res %= MOD;
            }
        }
        DP[curL][curH][maxH] = res;
    }
    return DP[curL][curH][maxH];
}

int main() {
    int T, L, H;
    cin >> T;
    DP = vector<vector<vector<int>>>(401, vector<vector<int>>(201, vector<int>(201, -1)));
    DP[0][0][0] = 1;
    while(T--) {
        cin >> L >> H;
        cout << recur(2*L, 0, H) << "\n";
    }
    return 0;
}
```

Monedas 2

Teoría

- Programación dinámica

Solución

Vamos moneda a moneda actualizando las posibles sumas. Un valor k es alcanzable desde la moneda i, con valor x si la cantidad $k - x$ es alcanzable utilizando las primeras $i - 1$ monedas. Así, vamos moneda a moneda actualizando los valores alcanzables con un vector de booleanos, teniendo cuidado de ir "hacia atrás" en el bucle de la cantidad para evitar contar a la misma moneda dos veces en una suma.

Finalmente, contamos cuántas celdas dentro del vector de booleanos están marcadas como `true` y devolvemos ese valor.

Código

C++

```cpp
#include <vector>
#include <iostream>
using namespace std;

int sumas(vector<bool> & DP, vector<int> & monedas) {
    DP[0] = true;
    for (int i = 0; i < (int) monedas.size(); i++) {
        for (int j = (int) DP.size()-1; j >= monedas[i]; j--)
            // el total j es alcanzable desde la moneda i'ésima si lo era desde la
            ↪ (i-1)'ésima o si la cantidad j - el valor de la moneda i'ésima sí
            ↪ es alcanzable
            DP[j] = DP[j] || DP[j - monedas[i]];
    }
    int respuesta = 0;
    // devolvemos el número de sumas posibles
    for (int i = 1; i < (int) DP.size(); i++)
        respuesta += DP[i];
    return respuesta;
}
```

```cpp
int main() {
    int T, N, suma;
    vector<bool> DP;
    vector<int> monedas;
    cin >> T;
    while(T--) {
        cin >> N;
        monedas = vector<int>(N);
        suma = 0;
        for (int i = 0; i < N; i++) {
            cin >> monedas[i];
            suma += monedas[i];
        }
        DP.assign(suma+1, false);
        cout << sumas(DP, monedas) << '\n';
    }
    return 0;
}
```

Raisins

Teoría

- 2D Prefix Sums.
- Programación Dinámica.

Solución

Las ideas principales son:

- Utilizar 2D prefix sums para poder calcular el valor de todo subrectángulo en tiempo constante.
- Utilizar programación dinámica para determinar el mínimo valor a pagar. La formulación recursiva simplemente examina todos los posibles cortes, verticales y horizontales, y lleva la cuenta del mínimo.

Los detalles de cómo adaptarlos al problema en cuestión están comentados en el código.

Código

C++

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

using namespace std;

const int N = 51;  // el máximo valor, más 1 para operar en el rango 1..50.
const int M = 51;
vector<vector<int>> ps_v;  // 2D prefix sums de los valores (núm. de raisins).

// Retorna el mínimo coste de partir una tableta rectangular de dimensiones
// horizontales x..X y verticales y..Y
int f(int x, int X, int y, int Y, vector<vector<vector<vector<int>>>> & dp) {
  if (x + 1 == X && y + 1 == Y) return 0;  // caso base
  if (dp[x][X][y][Y]) return dp[x][X][y][Y];
  int minimo = INT_MAX; // Asignamos el mínimo a "infinito", que consideramos que
  ↪  es el máximo valor que entra en un int
  // Calculamos, de entre todos los cortes verticales y horizontales posibles,
  // aquel de coste mínimo.
  // Cortes horizontales
  for (int i = x + 1; i < X; i++) {
    minimo = min(minimo, f(x, i, y, Y, dp) /* cacho superior */ +
                         f(i, X, y, Y, dp) /* cacho inferior */);
  }
```

```
  // Cortes verticales
  for (int i = y + 1; i < Y; i++) {
    minimo = min(minimo, f(x, X, y, i, dp) /* cacho izquierdo*/ +
                         f(x, X, i, Y, dp) /* cacho derecho */);
  }
  // Combinamos el resultado mínimo obtenido de los subproblemas con el coste
  // del problema actual, utilizando 2D prefix sums.
  return dp[x][X][y][Y] =
             minimo + ps_v[X][Y] - ps_v[x][Y] - ps_v[X][y] + ps_v[x][y];
}

int main() {
  int n, m;
  cin >> n >> m;
  ps_v = vector<vector<int>>(n + 1, vector<int>(m + 1));
  vector<vector<vector<vector<int>>>> dp(N, vector<vector<vector<int>>>(M,
  ↪  vector<vector<int>>(N, vector<int>(M, 0))));
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      cin >> ps_v[i][j];
    }
  }
  // Cálculo de las 2D prefix sums
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      ps_v[i][j] += ps_v[i - 1][j] + ps_v[i][j - 1] - ps_v[i - 1][j - 1];
    }
  }
  cout << f(0, n, 0, m, dp) << '\n';
}
```

Miners

Teoría

- Programación Dinámica.

Solución

La idea es utilizar programación dinámica. Sea `M(n, estado1, estado2)` la mayor cantidad de carbón que puede ser producida tras n-1 envíos de comida. "estado1" describe el historial de envíos a la primera mina, "estado2" el de los enviados a la segunda. Sea también value(estado, comida) el valor que se obtiene cuando se envía "comida" a una mina en estado "estado".

La formulación recursiva sería:

```
enviando_a_mina1 = M(n+1, nuevo-estado1-si-se-envia-a-mina1, estado2)
enviando_a_mina2 = M(n+1, estado1, nuevo-estado2-si-se-envia-a-mina2)
valor_obtenido_al_enviar_a_mina1 = value(estado1, envio)
valor_obtenido_al_enviar_a_mina2 = value(estado2, envio)
M(n, estado1, estado2) =
   max( enviando_a_mina1 + valor_obtenido_al_enviar_a_mina1,
        enviando_a_mina2 + valor_obtenido_al_enviar_a_mina2 )
```

El último paso consiste en darnos cuenta de que `M(n+1, ...)` sólo depende del valor de `M(n, ...)`, con lo que sólo es necesario almacenar dos "filas" simultáneamente.

Código

C++

```cpp
#include <bits/stdc++.h>

using namespace std;

const int N = 1e5;

int food[N + 1];
// Dimensiones: n, penultima comida mina 1, ultima comida mina1,
//                 penultima comida mina 2, ultima comida mina2
int dp[2][4][4][4][4];
string str;

int nFood;
```

```cpp
int value(int a, int b, int c) {
  int ans = 0;
  if (a == 1 || b == 1 || c == 1) {
    ++ans;
  }
  if (a == 2 || b == 2 || c == 2) {
    ++ans;
  }
  if (a == 3 || b == 3 || c == 3) {
    ++ans;
  }
  return ans;
}

int main() {
  cin >> nFood;
  cin >> str;
  for (int i = 1; i <= nFood; ++i) {
    char foodchar = str[i - 1];
    // un switch es una alternativa a los bloques 'if' cuando todas las
    ↪  condiciones son sobre el valor de la misma variable
    switch(foodchar) {
      case 'M':
        food[i] = 1;
        break;
      case 'F':
        food[i] = 2;
        break;
      case 'B':
        food[i] = 3;
        break;
    }
  }

  for (int i = nFood + 1; i >= 1; --i) {
    for (int x1 = 0; x1 <= 3; ++x1) {
      for (int x2 = 0; x2 <= 3; ++x2) {
        for (int y1 = 0; y1 <= 3; ++y1) {
          for (int y2 = 0; y2 <= 3; ++y2) {
            int curr = i % 2;
            int next = curr ^ 1;
            if (i == nFood + 1) {
              dp[curr][x1][x2][y1][y2] = 0;
            } else {
              int send_to_mine1 =
                  value(x1, x2, food[i]) + dp[next][food[i]][x1][y1][y2];
              int send_to_mine2 =
                  value(y1, y2, food[i]) + dp[next][x1][x2][food[i]][y1];
              dp[curr][x1][x2][y1][y2] = max(send_to_mine1, send_to_mine2);
            }
          }
        }
      }
    }
  }
  cout << dp[1][0][0][0][0];
  return 0;
}
```

Bank

Teoría

- Programación dinámica con máscaras de bits
- Backtracking

Solución

La solución de Bank se compone de varias partes separadas. Primero, recogemos los salarios de los empleados y los valores de los billetes del usuario. Sabiendo esto, buscamos el mayor salario. Una vez encontrado este valor, calculamos la suma de cada posible combinación de billetes. Todas las que no superen este máximo salario pueden interesarnos y, por tanto, guardamos la máscara de bits (forma eficiente de guardar la combinación de billetes en cuestión) para acceder a ella más adelante.

En este momento, hacemos una comprobación rápida de que haya al menos una combinación de billetes que sume cada salario. Si no es el caso, sabemos que no hay solución posible sin dar más pasos y cortamos el proceso.

Si hay al menos una forma de pagar cada salario, procedemos así: conociendo las combinaciones de billetes que suman el salario de cada empleado, vamos empleado a empleado probando todas las combinaciones que suman su salario y yendo al siguiente de la lista. Repetimos el mismo proceso, disponiendo de los billetes no utilizados anteriormente. Hay dos posibilidades eventualmente: llegamos al final de la lista habiendo pagado a todos los empleados, en cuyo caso retornamos `true`, o llegamos a un punto muerto, momento en el que hacemos backtracking y volvemos al empleado anterior, cambiando la combinación de billetes usada para pagarle. Hacemos esto hasta encontrar una forma de pagar a todos o concluir tras probar todas las combinaciones posibles que no hay forma de pagar los N salarios con los M billetes disponibles.

Código

C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

bool posible(int indiceEmpleado, int mascara, vector<int> & salarios,
 ↪  vector<vector<int>> & formasDeLlegar, vector<vector<int>> & DP) {
        if (indiceEmpleado == (int) salarios.size()) {
                // Hemos conseguido pagar a todos los empleados
                return true;
        }
        if (DP[indiceEmpleado][mascara] != -1) {
                // Ya hemos comprobado esta posibilidad, por lo que retornamos el
                 ↪  valor sin tener que calcularlo nuevamente.
                return DP[indiceEmpleado][mascara];
        }
```

```
        int salarioActual = salarios[indiceEmpleado]; // guardamos el valor del
        ↪  empleado que estamos mirando
        bool respuesta = false; // esta variable nos dirá si hemos encontrado
        ↪  alguna posibilidad de pagar a este i'ésimo empleado y a los restantes
        ↪  con la combinación de billetes utilizada para los i-1 primeros.
        for (auto posibilidad: formasDeLlegar[salarioActual]) {
                if (!(mascara & posibilidad)) {
                        respuesta |= posible(indiceEmpleado + 1, mascara |
                        ↪  posibilidad, salarios, formasDeLlegar, DP); // vemos
                        ↪  si hay solución con esta posibilidad de pagar al
                        ↪  i'ésimo empleado
                        if (respuesta) {
                                // Ya hemos encontrado una solución
                                break;
                        }
                }
        }
        return DP[indiceEmpleado][mascara] = respuesta; // retornamos la respuesta
        ↪  y la guardamos para ahorrarnos futuros cálculos
}

void mascarasASumas(vector<vector<int>> & formasDeLlegar, vector<int> & billetes,
↪  int mayorSalario) {
        // Esta función calcula la suma de todas las combinaciones posibles de
        ↪  billetes y guarda qué combinaciones
        // llevan a cada suma para acceder a ellas rápidamente más adelante.
        int sumaActual, M = (int) billetes.size();
        for (int mascara = 0; mascara < (1<<M); mascara++) {
                sumaActual = 0;
                for (int i = 0; i < M; i++) {
                        if (mascara & (1<<i))
                                sumaActual += billetes[i];
                }
                if (sumaActual <= mayorSalario)
                        formasDeLlegar[sumaActual].push_back(mascara);
        }
}

bool salariosAlcanzables(vector<int> & salarios, vector<vector<int>> &
↪  formasDeLlegar) {
        // Comprobación rápida para ver si hay algún empleado cuyo salario no
        ↪  corresponde a ningún total de billetes
        for (auto salario: salarios) {
                if (formasDeLlegar[salario].empty())
                        return false;
        }
        return true;
}
```

```cpp
int main() {
        int N, M, mayorSalario;
        vector<int> salarios, billetes;
        vector<vector<int>> formasDeLlegar;
        vector<vector<int>> DP;
        cin >> N >> M;
        salarios = vector<int>(N);
        billetes = vector<int>(M);
        DP.assign(N, vector<int>(1<<M, -1));

        for (int i = 0; i < N; i++)
                cin >> salarios[i];
        for (int i = 0; i < M; i++)
                cin >> billetes[i];

        mayorSalario = *max_element(salarios.begin(), salarios.end()); // encontramos el salario más alto
        formasDeLlegar = vector<vector<int>>(mayorSalario+1, vector<int>());
        mascarasASumas(formasDeLlegar, billetes, mayorSalario); // guardamos las máscaras que llegan a cada total

        if (salariosAlcanzables(salarios, formasDeLlegar) && posible(0, 0, salarios, formasDeLlegar, DP))
                cout << "YES";
        else
                cout << "NO";

        return 0;
}
```