Advent of code 2024

Miembros equipo:
MªDolores Muñoz García
Jorge Márquez Morales

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

int main() {
	vector<string> tree = {
		"              *",
		"             ^",
		"            /^\\",
		"           / * \\",
		"          / ^ * \\",
		"         / * ^ * \\",
		"        / ^ * ^ * \\",
		"       / * ^ * ^ * \\",
		"      / ^ * ^ * ^ * \\",
		"             [ | | ]",
		"             [ | | ]",
		"      cout << \"Feliz Año Miguel!\" << endl;",
	};


	for (const string& line : tree) {
		cout << line << endl;
	}

	return 0;
}
```

Índice

1. Dia 1: Historian Hysteria

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>
#include <cstdlib> // Para usar atoi
#include <algorithm> // Para usar std::count

using namespace std;

// Función para convertir poder convertir las listas de string a enteros
int StringAEntero(const string &cadena) {
    return atoi(cadena.c_str());
}

// Función para leer fichero y convertir a vectores
// tambien es aqui donde separamos el fichero en dos listas distintas
void LeerLista(vector<int> &lista1, vector<int> &lista2) {

    ifstream archivo("lista.txt");
    string linea, data;

    while (getline(archivo, linea)) {
        stringstream iss(linea);
        iss >> data;
        lista1.push_back(StringAEntero(data));
        iss >> data;
        lista2.push_back(StringAEntero(data));
    }
}

// funcion merge y merge sort para ordenar la lista
void merge(vector<int> &arr, int left, int mid, int right) {
    // creamos dos vectores a partir de dividir las listas
    vector<int> leftArr(arr.begin() + left, arr.begin() + mid + 1);
    vector<int> rightArr(arr.begin() + mid + 1, arr.begin() + right + 1);

    int i = 0, j = 0, k = left;

    while (i < leftArr.size() && j < rightArr.size()) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k++] = leftArr[i++];
        } else {
            arr[k++] = rightArr[j++];
        }
    }
    while (i < leftArr.size()) {
        arr[k++] = leftArr[i++];
    }
    while (j < rightArr.size()) {
        arr[k++] = rightArr[j++];
    }
}

// funcion en la que hacemos la llamada recursiva de las divisiones y luego la union de los resultados
void mergeSort(vector<int> &arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);       // Divide la mitad izquierda
        mergeSort(arr, mid + 1, right);  // Divide la mitad derecha
        merge(arr, left, mid, right);    // Combina los resultados
    }
}
```

```cpp

// funcion que calcula la diferencia entre las dos listas
int EncontrarDiferencia(const vector<int> &lista1, const vector<int> &lista2) {
    int suma = 0;
    for (int i = 0; i < lista1.size(); i++) {
        cout << lista1[i] << " - " << lista2[i] << " = " << abs(lista2[i] - lista1[i]) << endl;
        suma += abs(lista2[i] - lista1[i]);
    }
    return suma;
}

// funcion para calcular la similitud
int calcularSimilitud(const vector<int> &lista1, const vector<int> &lista2, int left, int right) {
    if (left > right) {
        return 0;
    }
    if (left == right) {
        return lista1[left] * count(lista2.begin(), lista2.end(), lista1[left]);
    }
    // volvemos a aplicar DyV para ir recorriendo las listas para ir mirando la similitud
    int mid = left + (right - left) / 2;
    int leftSim = calcularSimilitud(lista1, lista2, left, mid);
    int rightSim = calcularSimilitud(lista1, lista2, mid + 1, right);
    return leftSim + rightSim;
}

int main() {
    vector<int> lista1, lista2;

    // Leer las líneas y añadirlas a los vectores
    LeerLista(lista1, lista2);

    // Ordenar vectores usando merge sort
    mergeSort(lista1, 0, lista1.size() - 1);
    mergeSort(lista2, 0, lista2.size() - 1);

    // Encontrar diferencia
    int suma = EncontrarDiferencia(lista1, lista2);

    // Calcular similitud usando divide y vencerás
    int similitud = calcularSimilitud(lista1, lista2, 0, lista1.size() - 1);


    cout << "El puntaje de similitud total es: " << similitud << endl;
    cout << "El valor total que deseas es: " << suma << endl;

    return 0;
}

```

Introducción

El reto del Día 1 de **Advent of Code** consiste en reconciliar dos listas de IDs numéricos representando ubicaciones. El objetivo es calcular la distancia total entre ambas listas tras emparejar sus elementos ordenados de menor a mayor. Para ello los pasos a seguir serán: leer los datos desde un archivo, ordenar las listas y luego calcular la distancia absoluta entre los elementos emparejados.

Para resolver este problema aplicamos un algoritmo basado en **Divide y Vencerás** para ordenar las listas y luego calcular la distancia total entre los elementos correspondientes de ambas listas.

Pasos a seguir

1. **Leer los datos desde un archivo**: El archivo contiene pares de números enteros que representan los elementos de las dos listas. El primer paso consistió en leer el archivo de entrada que contiene los pares de números, donde cada línea del archivo representa dos números enteros separados por un espacio. Los datos se almacenaron en un vector de pares de enteros.
2. **Separación de datos**: Los pares de números se separan en dos listas distintas. Una vez leídos los datos, estos se dividieron en dos listas

separadas. Cada lista representa una columna de IDs. Los elementos de cada par se asignaron a listas distintas, denominadas c1 y c2.

3. **Ordenación de las listas**: Ambas listas de números se ordenan de menor a mayor. Las listas obtenidas fueron ordenadas de forma ascendente utilizando el algoritmo **Merge Sort** implementado por la función std::sort. Esta ordenación es crucial, ya que garantiza que los elementos se emparejen de la manera más eficiente posible, con los valores más pequeños en las primeras posiciones de cada lista.
4. **Cálculo de distancias absolutas**: Se calcula la distancia absoluta entre los elementos emparejados de las listas. Una vez las listas estaban ordenadas, se recorrieron simultáneamente y se calculó la distancia absoluta entre los elementos emparejados. La distancia absoluta entre dos elementos se calculó con la función std::abs, y este valor se sumó a una variable que acumuló la distancia total.
5. **Suma de distancias**: Las distancias absolutas calculadas se suman para obtener la distancia total. El total de todas las distancias absolutas entre los elementos emparejados fue acumulado en una variable denominada distanciaTotal. Esta variable se actualizó en cada iteración del recorrido de las listas.
6. **Impresión del resultado**: Se muestra en pantalla la distancia total calculada Finalmente, una vez calculada la distancia total, el programa imprimió el valor de distanciaTotal, que representa la distancia total entre las dos listas después de emparejar sus elementos ordenados.

2. Día 5: Print Queue

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <sstream>
#include <unordered_map>
#include <unordered_set>
#include <queue>
#include <algorithm>
#include <fstream>


// funcion para leer los archivos rules y updates. lo hemos divido en 2 archivos para no tener todo en un
void parseInput(std::ifstream& ruleFile, std::ifstream& updateFile,
                std::vector<std::pair<int, int>>& rules, std::vector<std::vector<int>>& updates) {
    std::string line;
    // abrimos archivo de las reglas y leemos linea por linea.
    // al leer cada linea separamos los nuemros en x|y y los almacenamos como un par en la lista rules
    while (std::getline(ruleFile, line)) {
        std::stringstream ss(line);
        int x, y;
        char sep;
        ss >> x >> sep >> y;
        rules.emplace_back(x, y);
    }
    //abrimos archivo de updates y leemos linea por linea
    //convertimos cada linea en una lista y gardamos todas las listas en una lista llamada updates
    while (std::getline(updateFile, line)) {
        std::stringstream ss(line);
        std::vector<int> pages;
        int page;
        char sep;
        while (ss >> page) {
            pages.push_back(page);
            ss >> sep; // Consume ','
        }
        updates.push_back(pages);
    }
}

// funcion para hacer el grafico en funcion de las reglas
void buildGraph(const std::vector<std::pair<int, int>>& rules,
                std::unordered_map<int, std::vector<int>>& graph,
                std::unordered_map<int, int>& inDegree) {
  //
    for (const auto& rule : rules) {
        int from = rule.first;
        int to = rule.second;
        graph[from].push_back(to);
        inDegree[to]++;
        if (inDegree.find(from) == inDegree.end()) {
            inDegree[from] = 0;
        }
    }
}

// fucnion para ver si un update respeta las reglas comparando las posicion del elemento en la lista updates con el grafo con las reglas
bool isUpdateValid(const std::vector<int>& update, const std::vector<std::pair<int, int>>& rules) {
    std::unordered_map<int, std::vector<int>> graph;
    std::unordered_map<int, int> inDegree;
    // creamos el arbol y un mapa de grados de entrada basados en la reglas
    buildGraph(rules, graph, inDegree);

    // mapear las paginas de la actualizacion a sus posiciones
```

```cpp
    std::unordered_map<int, int> pagePosition;
    for (size_t i = 0; i < update.size(); ++i) {
        pagePosition[update[i]] = i;
    }

    // verificamos que al actualizar se cumplan las reglas
    for (const auto& rule : rules) {
      int x = rule.first;//pagina que debe ir primero
      int y = rule.second;// pagina que deberia de ir la segunda
      // comprobamos que ambas paginas estén en la actualizacion
        if (pagePosition.count(x) && pagePosition.count(y)) {
      if (pagePosition[x] >= pagePosition[y]) {// condicion que comprueba que se cumpla la regla
                return false;
          }
      }
    }
    return true;
}
// funcion para encontrar y devolver la pagina del medio en una actualizacion
int findMiddlePage(const std::vector<int>& update) {
    return update[update.size() / 2];
}

// Función para procesar las actualizaciones y calcular la suma de las páginas del medio
int processUpdates(const std::vector<std::pair<int, int>>& rules, const std::vector<std::vector<int>>& updates) {
    int sumOfMiddlePages = 0; // Inicializar la suma
    for (const auto& update : updates) {
        if (isUpdateValid(update, rules)) { // Verificar si la actualización es válida
            sumOfMiddlePages += findMiddlePage(update); // Agregar la página del medio a la suma
        }
    }
    return sumOfMiddlePages; // Devolver la suma total
}


int main() {
    // abrimos los archivos
    std::ifstream ruleFile("rules.txt");
    std::ifstream updateFile("updates.txt");

    if (!ruleFile.is_open() || !updateFile.is_open()) {
        std::cerr << "Error: Unable to open input files." << std::endl;
        return 1;
    }

    // analizmos los archivos
    std::vector<std::pair<int, int>> rules;
    std::vector<std::vector<int>> updates;
    parseInput(ruleFile, updateFile, rules, updates);

    // una vez que ya hemos leido todos los archivos y tenemos las listas con los datos modificados podemos cerrar los archivos
    ruleFile.close();
    updateFile.close();


    // Procesar las actualizaciones y calcular la suma de las páginas del medio
    int sumOfMiddlePages = processUpdates(rules, updates);

    // Output the result
    std::cout << "sumas de las paginas del medio de las actualizaciones correctas: " << sumOfMiddlePages << std::endl;

    return 0;
}
```

Introducción

Este problema consiste en procesar un conjunto de actualizaciones de páginas, siguiendo un conjunto de reglas de precedencia entre páginas. Para ello, se deben verificar si cada actualización cumple con las reglas establecidas y luego, en caso afirmativo, sumar las páginas que ocupan la posición intermedia en cada actualización. La solución está basada en la construcción de un **grafo dirigido** y en el procesamiento de las actualizaciones para asegurar que se respeten las reglas antes de realizar el cálculo empleando **hash**.

Pasos a seguir

1. **Leer las reglas y las actualizaciones desde archivos**: Los datos de entrada consisten en dos archivos: uno con las reglas de precedencia y otro con

las actualizaciones de páginas. La función parseInput se encarga de leer dos archivos: uno con las reglas (rules.txt) y otro con las actualizaciones (updates.txt).

- **Archivo de reglas**: Cada línea contiene una regla en la forma x|y, indicando que la página x debe ir antes que la página y. Estos pares de páginas se almacenan en un vector rules.
- **Archivo de actualizaciones**: Cada línea contiene una lista de números representando las páginas que se deben actualizar. Estas se almacenan como listas dentro de un vector updates.

2. **Construcción del grafo de reglas**: Representar las reglas como un grafo dirigido donde las páginas están conectadas según el orden de precedencia. La función buildGraph toma las reglas leídas y las utiliza para construir un grafo dirigido. En este grafo cada página es un nodo.

   Además, se mantiene un mapa de los grados de entrada (inDegree) para cada página, lo que ayudará más tarde al proceso de verificación.

3. **Verificación de actualizaciones**: Comprobar si cada actualización de páginas respeta las reglas de precedencia definidas en el grafo. La función isUpdateValid verifica si una actualización respeta las reglas de precedencia. Para ello, construye un grafo basado en las reglas y luego mapea las posiciones de las páginas en la actualización. Después, compara las posiciones de las páginas en la actualización con las reglas: si una página que debe ir antes de otra aparece en una posición posterior en la actualización, la actualización no es válida.

   Si la actualización cumple todas las reglas, se considera válida.

4. **Cálculo de las páginas del medio**: Para las actualizaciones válidas, determinar cuál es la página del medio y sumar su valor. La función findMiddlePage se encarga de determinar la página que ocupa la posición intermedia en una actualización, es decir, la página que se encuentra en el centro de la lista de páginas de la actualización. La función processUpdates procesa todas las actualizaciones y, para cada actualización válida, agrega el valor de la página intermedia a una suma total.
5. **Imprimir el resultado**: Mostrar la suma total de las páginas del medio de las actualizaciones válidas. Finalmente, el programa imprime la suma de las páginas del medio de todas las actualizaciones que cumplen con las reglas.

3. Dia 19: Linen Layout

```cpp
#include <fstream>
#include <iostream>
#include <map>
#include <sstream>
#include <string>
#include <vector>
#include <iterator>

struct TreeNode {
    std::string word;
    TreeNode *left = nullptr, *right = nullptr;
    TreeNode(const std::string& w) : word(w) {}
};

TreeNode* insert(TreeNode* root, const std::string& word) {
    if (!root) return new TreeNode(word);
    if (word < root->word) root->left = insert(root->left, word);
    else if (word > root->word) root->right = insert(root->right, word);
    return root;
}

bool search(TreeNode* root, const std::string& word) {
    while (root) {
        if (root->word == word) return true;
        root = word < root->word ? root->left : root->right;
    }
    return false;
}

bool canBuild(TreeNode* root, const std::string& word, std::map<std::string, bool>& cache) {
    if (auto it = cache.find(word); it != cache.end()) return it->second;
    if (search(root, word)) return cache[word] = true;
    for (size_t i = 1; i < word.size(); ++i) {
        if (canBuild(root, word.substr(0, i), cache) && canBuild(root, word.substr(i), cache))
            return cache[word] = true;
    }
    return cache[word] = false;
}

int main() {
    std::ifstream file("input.txt");
    if (!file) {
        std::cerr << "Cannot open file input.txt\n";
        return EXIT_FAILURE;
    }

    std::string line;
    std::getline(file, line);  // Leer la primera línea para el árbol
    std::istringstream iss(line);
    TreeNode* root = nullptr;
    for (std::string word; iss >> word; ) {
        if (word.back() == ',') word.pop_back();
        root = insert(root, word);
    }
```

```
63      int count = 1; //Inicializamos a 1 para contar con el nodo raíz
64      std::map<std::string, bool> cache;
65      for (size_t i = 2; i < lines.size(); ++i) {  // Procesar desde la segunda línea
66          if (canBuild(root, lines[i], cache)) ++count;
67      }
68
69      std::cout << count << '\n';
70      return EXIT_SUCCESS;
71  }
72
```

Introducción

El programa tiene como objetivo analizar si un conjunto de palabras puede construirse combinando otras palabras base, proporcionadas en un archivo de texto. Para ello, se utiliza una técnica basada en **árboles implícitos**, que explora todas las posibles divisiones de las palabras y determina si ambas partes pertenecen al conjunto de palabras base o pueden construirse recursivamente. Además, se implementa **memoización** para optimizar el rendimiento, evitando el análisis redundante de palabras ya evaluadas.

Pasos a seguir

1. L**ectura del Archivo:** Se comienza abriendo el archivo llamado input.txt. Si el archivo no se puede abrir, el programa informa del error y se detiene. A continuación, las líneas del archivo se leen y almacenan en un vector de cadenas para procesarlas más adelante. Este vector será la base para extraer las palabras base y las palabras que se evaluarán.
2. **Carga de Palabras Base:** Una vez leído el archivo, la primera línea contiene las palabras base separadas por comas. Estas palabras se procesan eliminando los caracteres no deseados (como comas al final de cada palabra) y se almacenan en un conjunto (set<string>). Este conjunto asegura que las palabras sean únicas y permite búsquedas rápidas para verificar si una palabra pertenece al grupo de palabras base.
3. **Inicialización de Variables:** Después de cargar las palabras base, se prepara un mapa (map<string, bool>) que actúa como una caché para optimizar el análisis de las palabras, evitando evaluaciones redundantes. También se inicializa un contador para llevar el registro de cuántas palabras son construibles a partir de las palabras base.
4. **Verificación de Palabras Construibles:** A partir de la tercera línea del archivo, el programa analiza cada palabra individualmente. Para ello, se utiliza una función recursiva llamada canBuild, que explora todas las posibles divisiones de la palabra en segmentos. Cada segmento se evalúa para verificar si pertenece al conjunto de palabras base o si puede construirse a partir de otras palabras ya verificadas. Si ambas partes de una división cumplen

alguna de estas condiciones, la palabra completa se considera construible y se almacena el resultado en la caché.

5. **Salida del Resultado:** Finalmente, el programa recorre todas las palabras del archivo y utiliza el contador para registrar cuántas de ellas son construibles. Una vez completado este proceso, el resultado (el total de palabras construibles) se imprime en pantalla.

4. Día 7: Bridge

```cpp
#include <iostream>
#include <vector>
#include <sstream>
#include <fstream>
#include <unordered_map>
#include <string>

using namespace std;

// Función auxiliar para evaluar expresiones de izquierda a derecha
// Esta función aplica los operadores + y * de izquierda a derecha en la lista de números
long evaluateLeftToRight(const vector<int>& nums, const vector<char>& ops) {
    long result = nums[0];  // Empezamos con el primer número
    for (size_t i = 0; i < ops.size(); ++i) {
        if (ops[i] == '+') {
            result += nums[i + 1];  // Si el operador es suma, sumamos el siguiente número
        } else if (ops[i] == '*') {
            result *= nums[i + 1];  // Si el operador es multiplicación, multiplicamos el siguiente número
        }
    }
    return result;  // Devolvemos el resultado final
}

// Función recursiva para explorar todas las combinaciones posibles de operadores
// La función intenta aplicar + o * entre los números hasta que se alcanza el último número
// Si una combinación de operadores produce el valor objetivo, devuelve true
bool isPossibleToReachTarget(const vector<int>& nums, int target, vector<char>& ops, int pos, unordered_map<string, bool>& memo) {
    // Si hemos llegado al último número en la lista
    if (pos == nums.size() - 1) {
        string key;
        for (char op : ops) key += op;  // Generamos una clave basada en la combinación de operadores
        if (memo.count(key)) return memo[key];  // Si ya hemos evaluado esta combinación, devolvemos el resultado guardado
        return memo[key] = (evaluateLeftToRight(nums, ops) == target);  // Evaluamos la expresión y guardamos el resultado
    }

    // Probamos agregar el operador '+' y exploramos el siguiente número
    ops.push_back('+');
    if (isPossibleToReachTarget(nums, target, ops, pos + 1, memo)) return true;
    ops.pop_back();  // Volvemos atrás si no encontramos una solución

    // Probamos agregar el operador '*' y exploramos el siguiente número
    ops.push_back('*');
    if (isPossibleToReachTarget(nums, target, ops, pos + 1, memo)) return true;
    ops.pop_back();  // Volvemos atrás si no encontramos una solución

    return false;  // Si no encontramos una solución con ninguna combinación de operadores
}
```

```cpp
// Función para procesar las ecuaciones y calcular la suma de los valores objetivos válidos
int processEquations(const vector<pair<int, vector<int>>>& equations) {
    int totalSum = 0;  // Inicializamos la suma total
    for (const auto& eq : equations) {
        int target = eq.first;  // El valor objetivo de la ecuación
        const vector<int>& nums = eq.second;  // Los números de la ecuación
        vector<char> ops;  // Lista de operadores a insertar entre los números
        unordered_map<string, bool> memo;  // Mapa para guardar las combinaciones evaluadas

        // Si solo hay un número y es igual al objetivo, se suma directamente
        if (nums.size() == 1 && nums[0] == target) {
            totalSum += target;
            continue;
        }

        // Llamamos a la función recursiva para verificar si es posible alcanzar el objetivo
        if (isPossibleToReachTarget(nums, target, ops, 0, memo)) {
            totalSum += target;  // Si es válido, sumamos el objetivo a la suma total
        }
    }
    return totalSum;  // Devolvemos la suma total de los objetivos válidos
}

// Función para leer el archivo de entrada y parsear las ecuaciones
vector<pair<int, vector<int>>> readInput(const string& filename) {
    ifstream file(filename);  // Abrimos el archivo para lectura
    vector<pair<int, vector<int>>> equations;  // Vector para almacenar las ecuaciones

    // Si no se puede abrir el archivo, mostramos un mensaje de error
    if (!file) {
        cerr << "Error: No se pudo abrir el archivo " << filename << endl;
        return equations;  // Devolvemos un vector vacío en caso de error
    }

    string line;
    // Leemos cada línea del archivo
    while (getline(file, line)) {
        stringstream ss(line);  // Creamos un stringstream para dividir la línea
        int target;  // Valor objetivo de la ecuación
        char colon;

        // Extraemos el objetivo y el carácter ':' (esperamos que esté presente)
        if (!(ss >> target >> colon)) {
            cerr << "Error: Línea mal formada: " << line << endl;
            continue;  // Si la línea está mal formada, la ignoramos
        }

        vector<int> nums;  // Lista de números de la ecuación
```

```
97          int num;
98          // Extraemos los números y los agregamos a la lista
99          while (ss >> num) {
100             nums.push_back(num);
101         }
102
103         // Guardamos la ecuación (objetivo y números) en el vector
104         equations.emplace_back(target, nums);
105     }
106
107     file.close();  // Cerramos el archivo después de leerlo
108     return equations;  // Devolvemos el vector con las ecuaciones leídas
109 }
110
111 int main() {
112     string filename = "input.txt";  // Definimos el nombre del archivo de entrada
113     vector<pair<int, vector<int>>> equations = readInput(filename);  // Leemos las ecuaciones del archivo
114
115     // Procesamos las ecuaciones y obtenemos la suma total de los objetivos válidos
116     int result = processEquations(equations);
117
118     // Imprimimos el resultado final
119     cout << "Suma total de los valores de calibración válidos: " << result << endl;
120
121     return 0;  // Terminamos el programa con éxito
122 }
123
```

Introducción

El objetivo de este proyecto es procesar un conjunto de ecuaciones de calibración contenidas en un archivo de entrada y calcular la suma de los valores de calibración válidos. Cada ecuación está formada por una serie de números, y el objetivo es verificar si existen combinaciones de operadores (+ y *) que permitan alcanzar un valor objetivo (target) para cada ecuación empleando **memoización** para almacenar las diferentes operaciones. Las combinaciones válidas de operaciones se evalúan de izquierda a derecha. El resultado final es la suma de los valores de calibración para los cuales se ha podido encontrar una combinación de operadores válida.

Pasos a seguir

1. **Leer las ecuaciones desde un archivo**: La función readInput se encarga de leer el archivo de entrada que contiene las ecuaciones. Cada línea del archivo consiste en un valor objetivo seguido de una serie de números, separados por espacios. Estos valores se almacenan como un par (target, nums), donde target es el valor objetivo y nums es un vector de enteros que representan los números de la ecuación.

2. **Explorar las combinaciones posibles de operadores**: Se deben considerar todas las combinaciones posibles de los operadores + y * entre los números de la ecuación para ver si se puede alcanzar el valor objetivo.

3. **Evaluar las ecuaciones**: Para cada ecuación, el programa intenta encontrar una combinación de operadores que haga que la evaluación de la ecuación sea igual al valor objetivo. La función isPossibleToReachTarget es la encargada de realizar este trabajo utilizando recursión. Se exploran todas las combinaciones de operadores entre los números, y la función evaluateLeftToRight evalúa cada combinación.

- Si una combinación de operadores da como resultado el valor objetivo, se considera una solución válida.
- Si una solución válida se encuentra, se suma el valor objetivo al total.

4. **Sumar los valores válidos**: Una vez que se han procesado todas las ecuaciones, la función processEquations devuelve la suma total de los valores de calibración válidos, es decir, aquellos valores objetivos que tienen una combinación de operadores que hace que la ecuación se evalúe correctamente.
5. **Optimización con memorización:** Para evitar realizar evaluaciones repetidas de las mismas combinaciones de operadores, se utiliza un mapa (unordered_map) para almacenar los resultados de las combinaciones evaluadas previamente. Esto mejora la eficiencia del programa y evita el cálculo innecesario.
6. **Impresión del resultado:** Al final, el programa imprime la suma total de los valores de calibración válidos que han sido encontrados en las ecuaciones procesadas.

5. Día 10: Hoof It

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <set>
#include <fstream> // Para el manejo de archivos

using namespace std;

// Direcciones para movernos hacia arriba, abajo, izquierda o derecha
const int dx[] = {0, 0, -1, 1}; // Movimientos en el eje X
const int dy[] = {-1, 1, 0, 0}; // Movimientos en el eje Y

// Función para verificar si una posición está dentro de los límites del mapa
// Asegura que no salgamos fuera de los límites de la cuadrícula del mapa
bool isValid(int x, int y, int rows, int cols) {
    return x >= 0 && x < rows && y >= 0 && y < cols;
}

// Función para realizar una búsqueda en anchura (BFS) y encontrar todas las posiciones alcanzables con un valor '9'
// Desde el punto de inicio, BFS sigue la restricción de "aumentar exactamente de 1" en cada paso
int bfs(const vector<vector<int>> &map, int startX, int startY) {
    int rows = map.size(); // Número de filas del mapa
    int cols = map[0].size(); // Número de columnas del mapa

    queue<pair<int, int>> q; // Cola para almacenar las posiciones a explorar
    set<pair<int, int>> visited; // Conjunto para seguir las posiciones visitadas

    // Inicializamos BFS con la posición de inicio
    q.push({startX, startY});
    visited.insert({startX, startY});

    int score = 0; // Variable para contar el número de '9' alcanzables

    // Mientras haya posiciones por explorar
    while (!q.empty()) {
        auto [x, y] = q.front(); // Obtenemos la posición actual
        q.pop(); // Quitamos la posición de la cola

        // Si la posición actual tiene altura 9, incrementamos el puntaje
        if (map[x][y] == 9) {
            score++;
            continue; // Saltamos la exploración desde esta posición
        }

        // Exploramos los vecinos en las 4 direcciones
        for (int i = 0; i < 4; i++) {
            int nx = x + dx[i]; // Nueva coordenada X
            int ny = y + dy[i]; // Nueva coordenada Y

            // Comprobamos si el vecino es válido, no ha sido visitado y cumple con la condición de aumento de 1
            if (isValid(nx, ny, rows, cols) && !visited.count({nx, ny}) && map[nx][ny] == map[x][y] + 1) {
                q.push({nx, ny}); // Añadimos el vecino a la cola
                visited.insert({nx, ny}); // Lo marcamos como visitado
            }
        }
    }
    return score; // Devolvemos el puntaje total de '9' alcanzables
}

// Función para leer el mapa desde un archivo
vector<vector<int>> readMapFromFile(const string &filename) {
    ifstream inputFile(filename); // Abrimos el archivo para lectura
    if (!inputFile) {
        cerr << "Error: ¡No se pudo abrir el archivo!" << endl;
        exit(1); // Terminamos el programa si no se puede abrir el archivo
    }

    vector<vector<int>> map; // Mapa donde se almacenará la cuadrícula
    string line;

    // Leemos el archivo línea por línea
    while (getline(inputFile, line)) {
        vector<int> row; // Fila del mapa
        for (char c : line) {
            if (isdigit(c)) {
                row.push_back(c - '0'); // Convertimos el carácter en número y lo agregamos a la fila
            }
        }
        if (!row.empty()) {
            map.push_back(row); // Añadimos la fila al mapa
        }
    }
    inputFile.close(); // Cerramos el archivo después de leerlo
    return map; // Devolvemos el mapa
}

// Función para calcular el puntaje total de todos los puntos de inicio (trailheads)
int calculateTotalScore(const vector<vector<int>> &map) {
    int rows = map.size(); // Número de filas del mapa
    int cols = map[0].size(); // Número de columnas del mapa
    int totalScore = 0; // Variable para almacenar el puntaje total

    // Buscamos todos los puntos de inicio (con altura 0) y calculamos su puntaje
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (map[i][j] == 0) { // Si la posición es un punto de inicio
                totalScore += bfs(map, i, j); // Añadimos el puntaje de ese punto de inicio
            }
        }
    }

    return totalScore; // Devolvemos el puntaje total
}

int main() {
    // Leemos el mapa desde el archivo
    vector<vector<int>> map = readMapFromFile("map.txt");

    // Calculamos el puntaje total para todos los puntos de inicio
    int totalScore = calculateTotalScore(map);

    // Imprimimos el puntaje total
    cout << "Puntaje total: " << totalScore << endl;

    return 0; // Terminamos el programa correctamente
}
```

Introducción

El reto consiste en un mapa de números enteros, donde cada celda tiene un valor que representa la altitud del punto en el mapa. Los "trailheads" son los puntos de partida del recorrido, definidos por las celdas de valor 0. El objetivo es realizar un recorrido desde estos trailheads mediante el uso de **grafos dirigidos** siguiendo un conjunto de reglas que permiten avanzar solo a celdas adyacentes que tengan una altura que aumente exactamente en 1 en cada paso.

Desde cada trailhead, debemos contar cuántos puntos de valor 9 son alcanzables, sumando el total de esos puntos alcanzados en el recorrido desde todos los trailheads presentes en el mapa.

Pasos a seguir:

1. **Entrada del problema:** Se proporciona un archivo de texto con el mapa. A continuación, se definen los desplazamientos posibles en el mapa: arriba, abajo, izquierda y derecha, para poder movernos entre las celdas del mapa. Para cada movimiento, se verifica que la nueva posición esté dentro de los límites del mapa mediante una función isValid.

2. **Búsqueda en Amplitud (BFS):** El algoritmo principal para recorrer el mapa es la búsqueda en amplitud (BFS). Desde cada trailhead (celda con valor 0), se exploran todas las celdas alcanzables que cumplen con la condición de que la altura de la celda destino debe ser exactamente 1 unidad mayor que la de la celda de origen. Si se encuentra una celda con altura 9, se cuenta y se incrementa el puntaje.

3. **Lectura del Mapa:** El mapa se lee desde un archivo de texto, donde cada línea representa una fila del mapa y cada número dentro de esa línea corresponde a una celda. Se procesa cada línea para almacenar el mapa en una estructura de datos adecuada.

4. **Cálculo del Puntaje Total:** La función principal del programa recorre todas las celdas del mapa. Cuando encuentra una celda con valor 0 (un trailhead), ejecuta la búsqueda en amplitud desde esa posición para calcular cuántos puntos de altura 9 son alcanzables. El puntaje total es la suma de todas las celdas alcanzables con altura 9 desde cada trailhead.

6. Recursos

Para la realización de este trabajo nos hemos ayudado de diversos recursos principalmente hemos utilizado:

- Diapositivas de clase para ayudarnos a estructurar los códigos antes de empezar a programar.
- Chat gpt (lo hemos usado para corrección de errores y dudas que nos hayan surgido a la hora de realizar alguna función)
- Canal de Reddit del advent of code. Cuando introducíamos una solución incorrecta el advent of code te da un enlace a Reddit con un foro para resolver preguntas.