Advent of code

Programación Avanzada
Curso 2024-2025
Profesor: Miguel Rebollo
Realizado por:
Miquel Benlloch Armijo
Mario Alonso de la Torre

Índice

Day 1

Porque hemos elegido este problema:

Hemos elegido este día ya que vimos que podía encajar con Divide y Vencerás ya que utilizando este algoritmo hemos conseguido:

1. **Ordenación:** utilizamos DyV para ordenar la lista eficientemente mediante el algoritmo Merge Sort.
2. **Eficiencia comparativa:** al ordenar los elementos, se facilita la comparación directa de los números de ambas listas
3. **Robustez:** al ordenar las listas previamente al cálculo de distancias y frecuencias no solo conseguimos mejorar la eficiencia sino que además conseguimos que el código sea escalable para gran cantidad de datos

- Por estas razones pensamos que el uso de DyV para este problema es más que adecuado. Ya que nos **facilita la resolución del problema** y lo hace más **eficiente que un algoritmo de fuerza bruta.**

Dificultades:

1. La primera pequeña dificultad que tuvimos fue aprender a cómo **encarar el problema.** Era el primer día que hacíamos el advent of code y no sabíamos que esperarnos.
2. El segundo inconveniente fue **fallar con los índices** en el Mergesort, un despiste de nada nos costó que el primer código que aparentemente era fácil se alargará más de lo previsto.
3. **Implementación unordered_map,** nos vimos con fuerzas de quitar también está estructura además de DyV, pero al final decidimos que este ejercicio no era el correcto para abordar dos ejercicios de una. Aunque conseguimos implementar el unordered_map, pero hay otro día que nos centramos más en él

En resumen: al ser el primer día buscamos hacer todo más rápido y eso nos costó algunos errores, pero al final conseguimos abordar el ejercicio de una manera en nuestra opinión muy satisfactoria que incluía DyV y de regalo unordered_map.

Introducción a nuestro código: //parte 1

Nos llegan 2 pares de vectores, y hay que comparar la distancia entre ellos, la distancia es |left-right|, siendo left y right los correspondientes vectores. Entonces lo primero de todo es ordenarlos, nosotros hemos decidido implementar una mergesort, para este problema.

```
void mergesort(vector<int>& array, int ini, int fin) {
    if (ini < fin) {
        int medio = ini + (fin - ini) / 2; // Calcular el punto medio
        mergesort(array, ini, medio);     // Ordenar la mitad izquierda
        mergesort(array, medio + 1, fin); // Ordenar la mitad derecha
        merge(array, ini, fin);           // Fusionar ambas mitades
    }
}
```

Se va dividiendo el vector para que sea una ordenación más fácil y después se acaba uniendo. Para eso hemos implementado una función llamada merge, para complementar el proceso

```
// Función para fusionar dos subarreglos en orden
void merge(vector<int>& array, int ini, int fin) {
    int medio = ini + (fin - ini) / 2; // Punto medio del subarreglo
    int n1 = medio - ini + 1;         // Tamaño del subarreglo izquierdo
    int n2 = fin - medio;             // Tamaño del subarreglo derecho

    // Crear subarreglos temporales
    vector<int> left(n1), right(n2);

    // Copiar datos en los subarreglos
    for (int i = 0; i < n1; i++) {
        left[i] = array[ini + i];
    }
    for (int i = 0; i < n2; i++) {
        right[i] = array[medio + 1 + i];
    }

    // Fusionar los subarreglos ordenados
    int i = 0, j = 0, k = ini;
    while (i < n1 && j < n2) {
        if (left[i] <= right[j]) {
            array[k] = left[i];
            i++;
```

```
        } else {
            array[k] = right[j];
            j++;
        }
        k++;
    }

    // Copiar los elementos restantes del subarreglo izquierdo (si los hay)
    while (i < n1) {
        array[k] = left[i];
        i++;
        k++;
    }

    // Copiar los elementos restantes del subarreglo derecho (si los hay)
    while (j < n2) {
        array[k] = right[j];
        j++;
        k++;
    }
```

Una vez ordenados simplemente se comparan calcula distancia

```
    // Ordenar ambas listas usando mergesort
    mergesort(left_side, 0, left_side.size() - 1);
    mergesort(right_side, 0, right_side.size() - 1);

    // Calcular la distancia total entre ambas listas
    int distance = 0;
    for (size_t i = 0; i < left_side.size(); i++) {
        distance += abs(left_side[i] - right_side[i]);
    }
```

//Añadido de parte 2

```
std::unordered_map<int,recurrent> map;
int last_number = INT_MIN;
for (int number : left_side){
    if(number == last_number){
       map[number].number_of_times_left_side +=1;
    }else{
        map[number] = {1,0};
    }
    last_number = number;
}
for(int number : right_side){
    if(map.find(number) == map.end()){
        continue; // not exist this number in the left side
    }
    map[number].number_of_times_right_side += 1;
}
int similarity = 0;
for(const auto& pair : map){
    similarity += pair.second.number_of_times_left_side * pair.second.number_of_times_right_side * pair.first;
}
```

Hemos creado un unorderer_map, en este nos vamos guardando la cantidad de veces que sale ese número en la lista de la izquierda. Y luego otro pero en el de la derecha. Una vez acabado el proceso de relleno de la tabla hash calculamos la similitud entre las listas gracias a las tablas.

Day 8

Porque hemos elegido este problema:

Hemos elegido este día ya que vimos que podía encajar con tablas hash ya que utilizando esta estructura hemos conseguido:

1. **Búsqueda de los elementos (rápida y eficiente):** para marcar y verificar posiciones.
2. **Evitar duplicados (eficientemente):** al ser hash, las claves son únicas y por esto si se repite una posición solo se contará una única vez.
3. **Facilidad:** se evita usar cualquier lógica compleja para verificar las posiciones

- Todo esto hace que el código sea más **simple** y **comprensible**, nos permite hacer un seguimiento **eficiente y menos costoso**. Por ende creemos que el uso de las tablas hash es una elección más que adecuada para este día.

Dificultades:

1. Primero de todo, la primera dificultad para nosotros fue **comprender** correctamente lo que nos pedía el enunciado.
2. Una vez entendimos el problema, vimos que el punto clave era resolver el **problema con las direcciones y orientaciones**
3. La menor de las dificultades, pero también importante fue la **elección de la estructura** para tener control de las posiciones visitadas
4. Otro problema que encontramos al acabar la primera parte fue **identificar la dirección** en la que están alineadas las antenas, asegurarse que detecta correctamente las diagonales.

Introducción al código:

Primero de todo creamos un struct donde en el ponemos las coordenadas x e y

```
struct Coordinate{
    int x;
    int y;
    Coordinate(int i, int j): x(i), y(j){}
};
```

Este struct lo usamos en la función de anthinodes, a esta le pasamos varios parámetros: el vector que representa el mapa, el position_anthenes, la variable antenna, las coordenadas x e y, y overlaps.

```
for(const Coordinate& pos : position_anthenes[anthenna]){
    x_difference = x - pos.x;
    y_difference = y - pos.y;
```

Aquí vamos recorriendo las anthenas y vamos calculando la diferencia de posiciones entre las anthenas nuevas y las previas.

```
    x_pos = pos.x - x_difference;
    y_pos = pos.y - y_difference;
    if(x_pos >= 0 && y_pos >=0 && x_pos < map.size() && y_pos < map[x_pos].size() ){
        if(map[x_pos][y_pos] == '.'){
            map[x_pos][y_pos] = '#';
        }else if (map[x_pos][y_pos] != '#'){
            overlaps.insert(std::make_pair(x_pos,y_pos));
        }
    }
}
```

Después calculamos la nueva posición de los antinodos, una vez calculada comprobamos si se encuentra dentro del mapa. Una vez comprobado procedemos a colocar el antinodo # en el caso de que sea una posición vacía o válida, con un carácter entre '.' o '#'

```
position_anthenes[anthenna].push_back(Coordinate(x,y));
for(int i = x; i < map.size(); i++){
    for(int j = 0; j < map[x].size(); j++){
        if(i == x && j <= y) continue;
        if(map[i][j] == anthenna){
            x_difference = i - x;
            y_difference = j - y;
            x_pos = i + x_difference;
            y_pos = j + y_difference;
```

Luego vamos almacenando la posición de la nueva antena en este struct. Posteriormente vamos recorriendo el mapa desde la posición de la nueva antena. Una vez llegado a otra antena calculamos la posición de los siguientes antinodos. Lo que nos vuelve a llevar a la verificación de estos

```
            if(x_pos >= 0 && y_pos >=0 && x_pos < map.size() && y_pos < map[x_pos].size() ){
                if(map[x_pos][y_pos] == '.'){
                    map[x_pos][y_pos] = '#';
                }else if (map[x_pos][y_pos] != '#'){
                    overlaps.insert(std::make_pair(x_pos,y_pos));
```

Day 10

Porque hemos elegido este problema:

Hemos elegido este problema porque es idóneo para el uso de estructuras de grafos para representar y analizar las relaciones entre posiciones en un mapa bidimensional. A través de este enfoque, logramos resolver el desafío de forma eficiente y estructurada por las siguientes razones:

1. **Modelado natural del problema**: El mapa bidimensional se interpreta como un grafo, donde cada celda representa un nodo, y las transiciones válidas entre celdas consecutivas (en términos de valores) se convierten en aristas. Esta representación hace que las relaciones entre las posiciones sean claras y manejables.
2. **Eficiencia con grafos:** Utilizamos un algoritmo de búsqueda en anchura (BFS) para explorar todos los caminos posibles desde cada celda con un valor 0 hacia las celdas con el valor 9. Esto permite una exploración sistemática y eficiente de los caminos sin omitir ninguna ruta válida.
3. **Prevención de duplicados:** Durante la búsqueda, usamos un conjunto (unordered_set) para marcar los nodos visitados, evitando procesar repetidamente las mismas posiciones. Esto no solo optimiza el tiempo de ejecución, sino que también asegura que cada ruta se cuente una sola vez.
4. **Escalabilidad y modularidad:** La separación del código en funciones específicas, como la construcción del grafo y la búsqueda con BFS, hace que el enfoque sea modular y escalable. Esto facilita la comprensión del flujo del programa y permite extender la funcionalidad de forma sencilla si el problema se complica.

- **En resumen** hemos elegido este enfoque porque nos permite manejar casos complejos con mayor claridad y flexibilidad, al aprovechar la capacidad de los grafos para adaptarse a diferentes restricciones y condiciones del problema. Por eso, decidimos apostar por esta técnica, que no solo nos proporciona un resultado correcto, sino también un proceso optimizado y elegante para alcanzar la solución.

Dificultades:

1. Nuestro primer inconveniente como casi todo el rato es **saber llegar a la solución con el algoritmo pertinente.**
2. Otra dificultad fue diseñar correctamente el algoritmo para **evitar revisar los nodos ya explorados** y contar correctamente los caminos hacia las celdas con valor ‘9’

3. Considerar que todas **las direcciones estuviesen consideradas** a la hora de construir el grafo y recorrerlo

   En resumen: nos pareció un día más complicado por la poca costumbre de usar grafos, ya que además se nos hace mucho más fácil trabajar con tablas hash y árboles.

Introducción a nuestro código:

Estructuras y utilidades clave

Node

Representa un nodo en el grafo con coordenadas (x, y), que son las posiciones de una celda en el mapa.

Implementa:

- Un constructor para inicializar las coordenadas.
- Una sobrecarga del operador == para comparar nodos.

NodeHash

- Una función de hash personalizada para que los objetos Node puedan ser usados en contenedores como std::unordered_map y std::unordered_set.

Relación entre Node y NodeHash

- Los nodos del mapa son usados como claves en un unordered_map para representar el grafo.

```cpp
// Estructura para representar un nodo en el grafo.
struct Node {
    int x, y; // Coordenadas de la celda en el mapa.
    Node(int x, int y) : x(x), y(y) {}

    // Operador de comparación para que los nodos puedan ser comparados.
    bool operator==(const Node& other) const {
        return x == other.x && y == other.y;
    }
};

// Hash personalizado para usar Node en contenedores como unordered_map o unordered_set.
struct NodeHash {
    std::size_t operator()(const Node& node) const {
        // Combina los hashes de las coordenadas x e y.
        return std::hash<int>()(node.x) ^ (std::hash<int>()(node.y) << 1);
    }
};
```

Construcción del Grafo

Función: build_graph

- Entrada: Un vector de cadenas (map) que representa el mapa.
- Salida: Un grafo representado como un unordered_map<Node, std::vector<Node>>.

Descripción:

- Construye el grafo en base al mapa, donde cada celda se convierte en un nodo.
- Conecta nodos adyacentes si el valor de la celda destino es el consecutivo de la celda actual (map[ni][nj] == map[i][j] + 1).
- Usa las coordenadas (x, y) de cada celda como clave para representar las conexiones en el grafo.

```cpp
// Construye un grafo a partir del mapa.
std::unordered_map<Node, std::vector<Node>, NodeHash> build_graph(const std::vector<std::string>& map) {
    std::unordered_map<Node, std::vector<Node>, NodeHash> graph;
    int rows = map.size();
    int cols = map[0].size();
```

Exploración del Grafo

Función: bfs (Breadth-First Search)

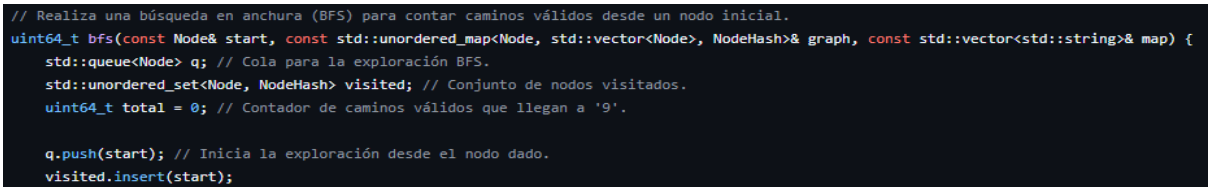
Entrada:

- Un nodo de inicio (Node start).
- El grafo (unordered_map<Node, std::vector<Node>>).
- El mapa (std::vector<std::string>).

Salida: La cantidad de caminos válidos desde la celda de inicio que terminan en una celda '9'.

Descripción:

- Utiliza una cola (std::queue<Node>) para realizar la búsqueda en anchura (BFS).
- Lleva un registro de los nodos visitados usando un unordered_set<Node>.
- Si se encuentra un nodo con el valor '9', se cuenta como un camino válido y no se exploran más vecinos desde allí.
- Antes de explorar los vecinos de un nodo, verifica que este exista en el grafo para evitar errores.



```
// Realiza una búsqueda en anchura (BFS) para contar caminos válidos desde un nodo inicial.
uint64_t bfs(const Node& start, const std::unordered_map<Node, std::vector<Node>, NodeHash>& graph, const std::vector<std::string>& map) {
    std::queue<Node> q; // Cola para la exploración BFS.
    std::unordered_set<Node, NodeHash> visited; // Conjunto de nodos visitados.
    uint64_t total = 0; // Contador de caminos válidos que llegan a '9'.

    q.push(start); // Inicia la exploración desde el nodo dado.
    visited.insert(start);
```

Day 11

Porque hemos elegido este problema:

Hemos elegido este día ya que vimos que podía encajar con árboles y programación dinámica ya que utilizando esta combinación hemos conseguido:

1. **Jerarquía**: el valor de un nodo depende de la operación que se haya utilizado, lo que nos lleva a esta estructura donde cada nodo puede llegar a tener uno o varios hijos.
2. **Eficiencia:** el uso de esta estructura nos permite representar de manera eficiente el problema, además de gracias a añadirle memorización estos pueden manejar subproblemas de manera efectiva
3. **Memorización:** La usamos para evitar cálculos innecesarios o redundantes, al usar una matriz de resultados memorizados, evitamos ese cálculo innecesario.
4. **Dinámica:** como tenemos crecimiento de nodos, la estructura no es estática. Por lo que es más eficiente ya que se pueden añadir stones de manera fácil
5. **Simplicidad:** Al utilizar memorización si hay algún cálculo repetido no hace volver a operar
6. **Eficiencia:** Al no tener que calcular todo el rato lo mismo, conseguimos que el código sea eficiente y veloz, aunque nosotros no percibamos esa diferencia temporal, hacemos que el código sea más robusto.

- Por estás razones pensamos que este algoritmo de programación dinámica encaja de buena manera, ya que conseguimos resolver el día once "**huyendo de la fuerza bruta**" y haciendo que el código sea computacionalmente más eficiente. Aunque un algoritmo de fuerza bruta pueda llegar a la misma solución aparentemente a la vez, no significa que este sea el más indicado, por eso nos hemos decantado por este enfoque. Además el enfoque dinámico con memoización es clave para la resolución de este problema.

Dificultades:

1. El primer problema como casi todo el rato es **saber enfocar el ejercicio** y buscar qué algoritmo o estructura le podía encajar.
2. Ajustar adecuadamente el problema para cubrir la **combinación** entre el algoritmo Pd y la estructura de los árboles posiblemente nos haya costado más que si hubiésemos hecho dos días. Pero una vez teníamos el enfoque no íbamos a desaprovechar la oportunidad de practicar
3. **Complejidad del código,** para nosotros este es el día más difícil de los que hemos entregado.

En resumen: nos pareció un día más complicado en comparación con el resto, pero pensamos que el resultado es satisfactorio porque al final el código funcionaba. Vamos notando el aumento de dificultad en comparación con los primeros días, pero vamos mejorando la dinámica

Introducción al código:

```
struct Node {
    long long engraving; // Valor del nodo
    Node* left = nullptr; // Hijo izquierdo
    Node* right = nullptr; // Hijo derecho
};
```

Primero de todo creamos el struct del árbol.

```
void generate_nodes(std::vector<std::unique_ptr<Node>>& nodes, std::unordered_map<long long, Node*>& seen_nodes) {
    while (!node_queue.empty()) {
        auto [node, level] = node_queue.front();
        node_queue.pop();

        // Limitar la profundidad para evitar procesamiento innecesario
        if (level >= 75) continue;

        // Calcular el número de dígitos del valor actual
        int num_digits = 0;
        {
            auto engraving = node->engraving;
            while (engraving > 0) {
                num_digits++;
                engraving /= 10;
            }
        }
```

vamos extrayendo de la cola elementos y luego con el pop lo eliminamos. Luego verificamos la profundidad a fin de limitar en caso de que sea necesario. Posteriormente buscamos contar el número de dígitos, lo inicializamos a 0 y vamos dividiendo entre 10, para comprobar cuántos dígitos tiene.

```
if (node->engraving == 0) {
    // Caso: valor 0 -> Crear nodo con valor 1
    auto new_node = std::make_unique<Node>();
    new_node->engraving = 1;

    if (seen_nodes.find(new_node->engraving) == seen_nodes.end()) {
        nodes.push_back(std::move(new_node));
        node->left = nodes.back().get();
        seen_nodes[nodes.back()->engraving] = nodes.back().get();
        node_queue.push({nodes.back().get(), level + 1});
```

Vamos creando nodos, con el make_unique, una manera diferente que la de usar new, pero en teoría más segura y eficiente. Si el nodo no ha sido visitado previamente se añade a la lista de nodos y se le añade a la parte izquierda. Luego se coloca en la cola de nodos y su valor se incluye en nodos visitados. Si este ya ha sido visitado se vuelve a usar ya el existente.

```
    } else {
        node->left = seen_nodes[new_node->engraving];
    }
} else if (num_digits % 2 == 0) {
    // Caso: número de dígitos par -> Dividir en dos mitades
    long long divisor = static_cast<long long>(std::pow(10, num_digits / 2));

    // Crear nodo izquierdo con la parte superior de los dígitos
    auto left_node = std::make_unique<Node>();
    left_node->engraving = node->engraving / divisor;
    if (seen_nodes.find(left_node->engraving) == seen_nodes.end()) {
        nodes.push_back(std::move(left_node));
        node->left = nodes.back().get();
        seen_nodes[nodes.back()->engraving] = nodes.back().get();
        node_queue.push({nodes.back().get(), level + 1});
    } else {
        node->left = seen_nodes[left_node->engraving];
    }
```

comprobamos si el nodo es par, en caso de que sea procedemos a dividirlo en dos mitades. Luego creamos un nuevo nodo de manera dinámica, y le ponemos el valor de del nodo actual dividido entre el divisor. Luego comprobamos si el valor ya ha sido visitado anteriormente, en caso de que no haya sido lo metemos a los ya visitados. Y esté se asigna como nodo izquierdo.

```
        // Crear nodo derecho con la parte inferior de los dígitos
        auto right_node = std::make_unique<Node>();
        right_node->engraving = node->engraving % divisor;
        if (seen_nodes.find(right_node->engraving) == seen_nodes.end()) {
            nodes.push_back(std::move(right_node));
            node->right = nodes.back().get();
            seen_nodes[nodes.back()->engraving] = nodes.back().get();
            node_queue.push({nodes.back().get(), level + 1});
        } else {
            node->right = seen_nodes[right_node->engraving];
        }
    } else {
        // Caso: número de dígitos impar -> Multiplicar por 2024
        auto new_node = std::make_unique<Node>();
        new_node->engraving = node->engraving * 2024;
```

nos vamos a la parte derecha del árbol. Y básicamente seguimos haciendo lo mismo que antes de comprobar los nodos visitados pero cambiando los valores que se le añaden al nodo, por ejemplo aquí tenemos la parte que multiplica a 2024.

```
std::unordered_map<long long, std::unordered_map<int, long long>> memo;

// Función para contar nodos hasta el nivel especificado
long long count_nodes(Node* node, int level) {
    if (level == MAX_LEVEL) return 1; // Llegamos al nivel máximo
    if (memo[node->engraving][level] != 0) return memo[node->engraving][level]; // Usar resultados memorizados

    long long total = 0;
    if (node->left != nullptr) total += count_nodes(node->left, level + 1);
    if (node->right != nullptr) total += count_nodes(node->right, level + 1);

    memo[node->engraving][level] = total; // Memorizar resultado
    return total;
}
```

aquí es donde nos almacenamos los datos, por así decirlo los memorizamos para evitar la redundancia de cálculos. Y utilizamos recursividad para ir contando los nodos y finalmente almacenamos el resultado en memo.