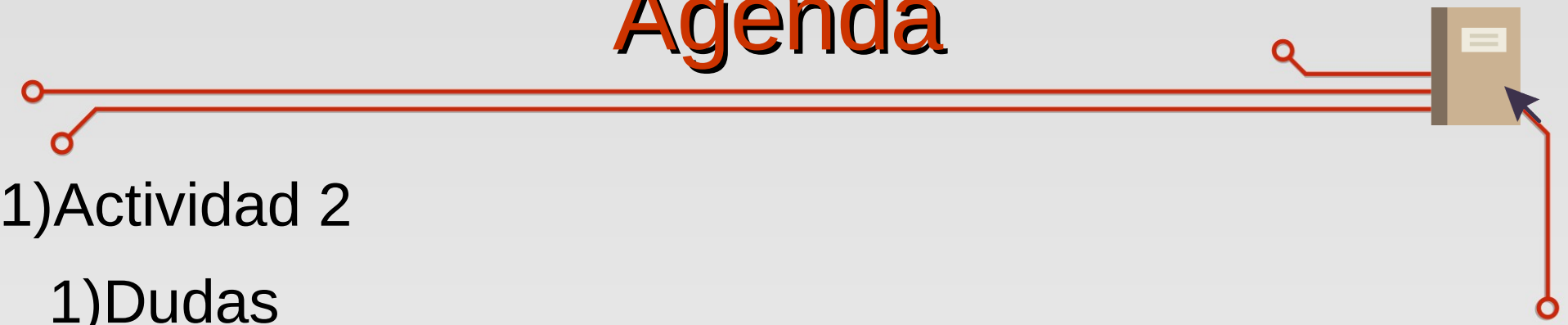
Agenda



1)Actividad 2

   1)Dudas

2)Recursividad

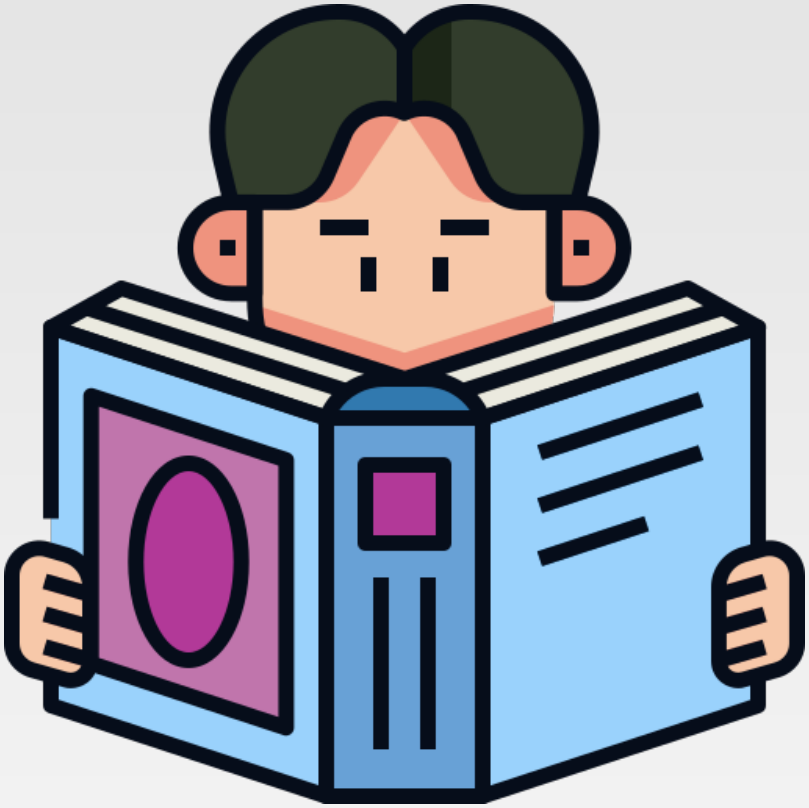
3)Para los grupos interesados, se les dará retroalimentación del UML del proyecto durante el transcurso de la clase.

4)Recordatorio:

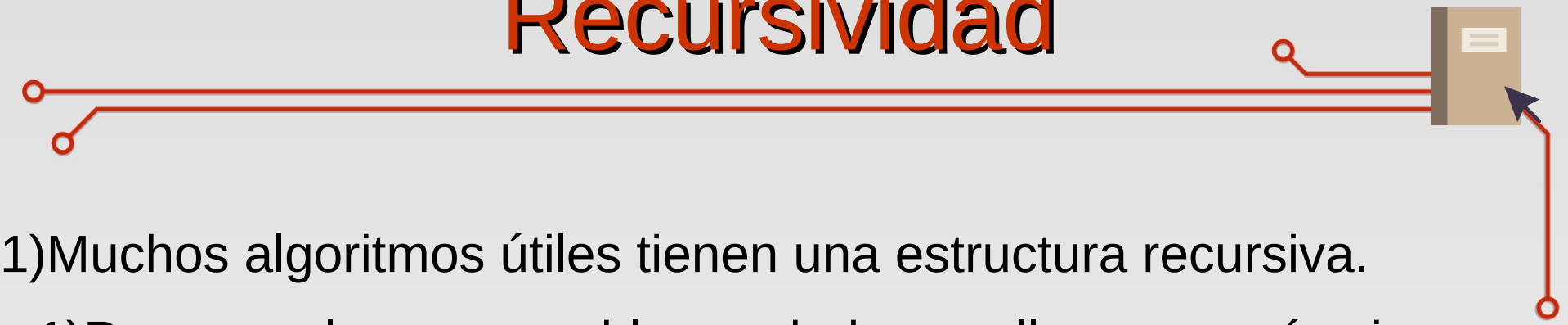
   1)Mañana, realización del I Examen: 8 am.

   2)Traer hojas blancas o rayadas, lápiz, lapicero.

Dudas Actividad 2



Recursividad



1)Muchos algoritmos útiles tienen una estructura recursiva.

   1)Para resolver un problema dado, se llaman a sí mismos recursivamente una o más veces para lidear con subproblemas estrechamente relacionados.

2)Estos algoritmos suelen seguir un enfoque de divide y vencerás (divide-and-conquer).

3)Dividir el problema en varios subproblemas que son similares al problema original, pero de menor tamaño, resuelve los subproblemas recursivamente y luego combina estas soluciones para crear una solución al problema original.

El enfoque divide-and-conquer

El paradigma divide y vencerás implica tres pasos en cada nivel de la recursividad:

1) Dividir el problema en varios subproblemas que son instancias más pequeñas del mismo problema.

2) Conquista los subproblemas resolviéndolos recursivamente. Si los tamaños de los subproblemas son lo suficientemente pequeños, simplemente, resuelve los subproblemas de una manera directa.

3) Combina las soluciones de los subproblemas en la solución del problema original.

Ejemplo 1 de Recursividad

Cálculo de factorial

Se desea escribir un programa para calcular el factorial de un número dado. El factorial de un número es el resultado de multiplicar ese número por todos los números menores a él. Por ejemplo, el factorial de 5 es 5 x 4 x 3 x 2 x 1 = 120. El programa debe utilizar la recursividad para calcular el factorial.

El usuario ingresará un número entero y el programa deberá calcular su factorial mediante una función recursiva llamada factorial(int n) y mostrar el resultado en pantalla.

Nota: El factorial de 0 es 1.

Factorial (solución recursiva)

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

int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}

int main() {
    int num;
    scanf_s("%d", &num);
    printf("%d\n", factorial(num));
    return 0;
}
```

Factorial (solución iterativa)

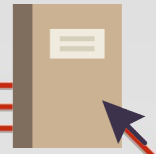
```
#include <iostream>
using namespace std;

int factorial(int num) {
    int result = 1;
    for (int i = 1; i <= num; i++) {
        result *= i;
    }
    return result;
}

int main() {
    int num;
    scanf_s("%d", &num);
    printf("%d\n", factorial(num));
    return 0;
}
```

Ejemplo 2 de Recursividad

Algoritmo Merge Sort

Es un algoritmo de ordenamiento que utiliza la técnica de dividir y conquistar. El algoritmo funciona dividiendo una lista en dos mitades y ordenando cada mitad de forma recursiva. Una vez que las dos mitades están ordenadas, se combinan para formar una lista ordenada completa.

Ejemplo 2 de Recursividad



Algoritmo Merge Sort

El algoritmo funciona de la siguiente manera:

1)Si el tamaño de la lista es 1 o 0, la lista está ordenada.

2)De lo contrario, se divide la lista en dos mitades.

3)Se llama al algoritmo de forma recursiva para ordenar cada mitad.

4)Se combinan las dos mitades ordenadas para formar una lista ordenada completa.

Ejemplo 2 de Recursividad

El algoritmo Merge Sort es considerado como uno de los algoritmos de ordenamiento más eficientes, ya que tiene una complejidad temporal de O(n log n) en promedio y en el peor de los casos, lo cual lo hace una excelente opción para ordenar grandes cantidades de datos. Aunque tiene un costo adicional de memoria ya que se utilizan arreglos auxiliares para realizar las mezclas.

Merge Sort (solución)

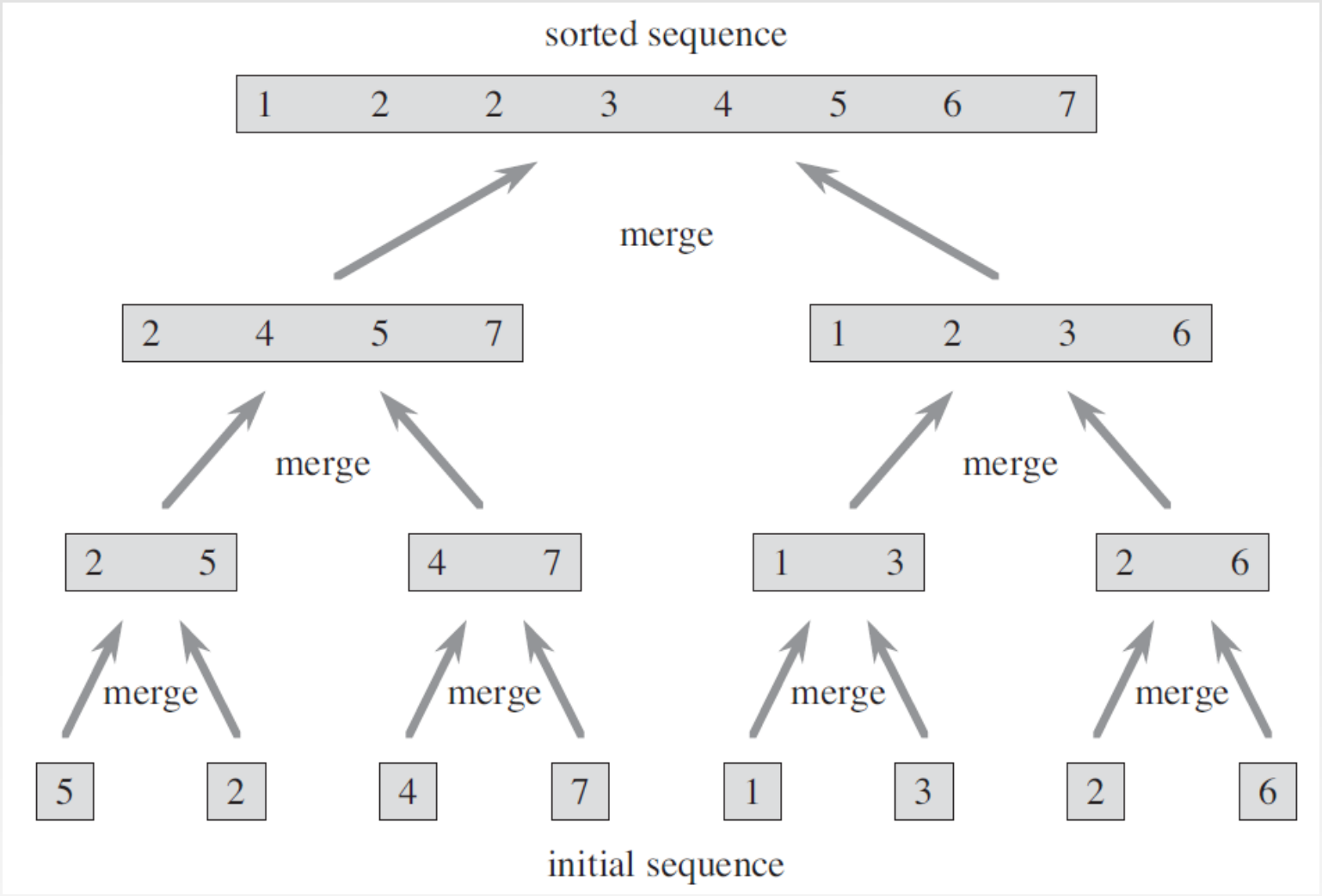
```
void mergeSort(int arr[], int start, int end) {
    if (start < end) {
        int middle = start + (end - start) / 2;
        mergeSort(arr, start, middle);
        mergeSort(arr, middle + 1, end);
        merge(arr, start, middle, end);
    }
}

int main() {
    int array[MAX] = { 5, 2, 4, 7, 1, 3, 2, 6 };

    mergeSort(array, 0, MAX - 1);

    for (int i = 0; i < MAX; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
    return 0;
}
```

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

const int MAX = 8;
```

Merge Sort (solución)

```
void merge(int arr[], int start, int middle, int end)
{
    int leftSize = middle - start + 1;
    int rightSize = end - middle;
    int* leftArray = new int[leftSize];
    int* rightArray = new int[rightSize];

    for (int i = 0; i < leftSize; i++) {
        leftArray[i] = arr[start + i];
    }
    for (int i = 0; i < rightSize; i++) {
        rightArray[i] = arr[middle + 1 + i];
    }

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

    while (i < leftSize && j < rightSize) {
        if (leftArray[i] <= rightArray[j]) {
            arr[k] = leftArray[i];
            i++;
        }
        else {
            arr[k] = rightArray[j];
            j++;
        }
        k++;
    }

    while (i < leftSize) {
        arr[k] = leftArray[i];
        i++;
        k++;
    }

    while (j < rightSize) {
        arr[k] = rightArray[j];
        j++;
        k++;
    }
    delete[] leftArray;
    delete[] rightArray;
}
```

Merge Sort (solución)



Otros ejemplos de recursividad

Algoritmo de búsqueda binaria (binary search)

El algoritmo de búsqueda binaria se utiliza para encontrar un valor específico en una lista ordenada. Funciona dividiendo la lista en dos mitades y comparando el valor buscado con el valor en el medio de la lista. Si el valor buscado es menor que el valor del medio, se busca en la mitad izquierda de la lista, y si es mayor, se busca en la mitad derecha. El proceso se repite hasta que el valor es encontrado o se determina que no está presente en la lista.

Al igual que Merge Sort, la búsqueda binaria tiene una complejidad temporal de O(log n), lo cual la hace muy eficiente para buscar valores en grandes cantidades de datos.

Otros ejemplos de recursividad

Algoritmo de la multiplicación de matrices

También conocido como el algoritmo de Strassen, es una técnica para multiplicar dos matrices A y B de una forma más eficiente que el algoritmo tradicional. El algoritmo divide cada una de las matrices en cuatro submatrices de tamaño n/2 x n/2 y utiliza un conjunto de 7 multiplicaciones de matrices más pequeñas en lugar de n^3 multiplicaciones elementales para calcular la matriz resultante C.

Acotación

Algoritmo tradicional de multiplicación de matrices

Consiste en calcular el producto de dos matrices A y B, donde cada elemento c[i][j] de la matriz resultante C es el resultado de multiplicar cada elemento de la fila i de la matriz A por cada elemento de la columna j de la matriz B, y sumando los resultados. El algoritmo se basa en aplicar la definición matemática de la multiplicación de matrices, se realiza de forma iterativa.

La complejidad temporal del algoritmo tradicional de multiplicación de matrices es O(n^3), lo que significa que el tiempo de ejecución aumenta significativamente a medida que las matrices se vuelven más grandes.

Multiplicación de matrices

El algoritmo funciona de la siguiente manera:

1)Divide cada una de las matrices A y B en cuatro submatrices de tamaño n/2 x n/2

2)Utilizando las submatrices de A y B, se realizan 7 multiplicaciones de matrices más pequeñas.

3)Se suman y restan las matrices resultantes para calcular 4 matrices intermedias.

4)Se combinan las matrices intermedias para calcular la matriz resultante C.

Multiplicación de matrices

La complejidad temporal del algoritmo de la multiplicación de matrices utilizando la técnica de dividir y conquistar es de O(n^log2(7)) que es menor que el algoritmo tradicional O(n^3) y en teoría es más eficiente que el algoritmo tradicional. Sin embargo, en la práctica, el algoritmo de Strassen solo es más eficiente para matrices de tamaño grande debido al costo adicional de la recursión y la necesidad de realizar operaciones adicionales para dividir y combinar las matrices.

Recursividad

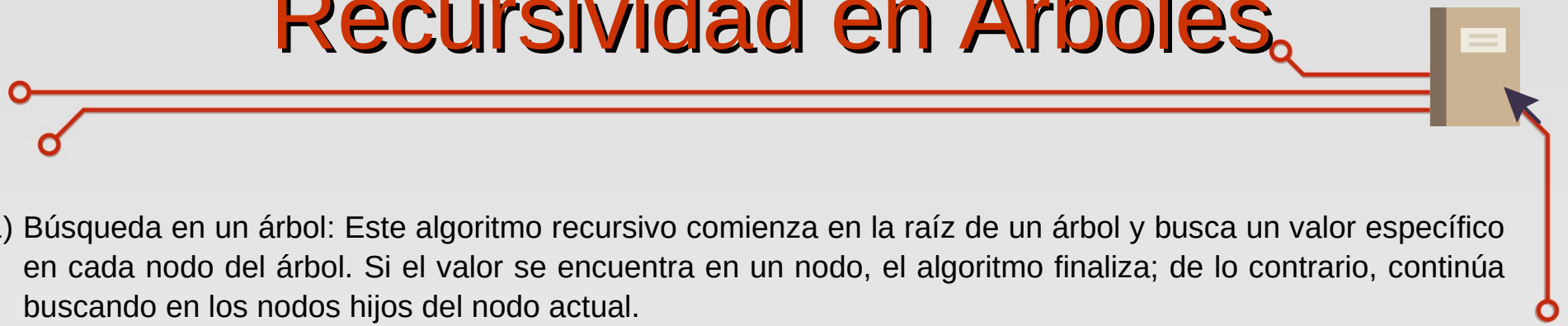
Anteriormente, se han mostrado ejemplos de recursividad aplicados a estructuras vectoriales y matriciales, pero

¿En dónde más se puede aplicar?

Recursividad en Grafos

1) Recorrido en profundidad de un grafo: Este algoritmo recursivo comienza en un nodo específico y visita todos los nodos adyacentes antes de regresar al nodo de inicio. El objetivo es recorrer todos los nodos del grafo sin visitar ninguno dos veces.
2) Recorrido en anchura de un grafo: Este algoritmo recursivo comienza en un nodo específico y visita todos los nodos adyacentes antes de continuar con los nodos adyacentes a los nodos visitados anteriormente. El objetivo es recorrer todos los nodos del grafo sin visitar ninguno dos veces.
3) Ordenamiento topológico: Este algoritmo recursivo se utiliza para ordenar los nodos de un grafo acíclico dirigido (DAG) de tal manera que si existe un arco desde el nodo A al nodo B, entonces B aparecerá después de A en el orden.
4) Caminos más cortos en un grafo: Este algoritmo recursivo se utiliza para encontrar el camino más corto entre dos nodos en un grafo utilizando el algoritmo de Dijkstra o Bellman-Ford.

Recursividad en Árboles



1) Búsqueda en un árbol: Este algoritmo recursivo comienza en la raíz de un árbol y busca un valor específico en cada nodo del árbol. Si el valor se encuentra en un nodo, el algoritmo finaliza; de lo contrario, continúa buscando en los nodos hijos del nodo actual.
2) Recorrido pre-orden en un árbol binario: Este algoritmo recursivo comienza en la raíz de un árbol binario y primero visita el nodo raíz antes de visitar recursivamente los nodos hijos izquierdo y derecho.
3) Recorrido post-orden en un árbol binario: Este algoritmo recursivo comienza en la raíz de un árbol binario y primero visita recursivamente los nodos hijos izquierdo y derecho antes de visitar el nodo raíz.
4) Recorrido in-orden en un árbol binario: Este algoritmo recursivo comienza en la raíz de un árbol binario y primero visita recursivamente el nodo hijo izquierdo, luego visita el nodo raíz y finalmente visita el nodo hijo derecho.
5) Encuentra la altura de un árbol binario: Este algoritmo recursivo se utiliza para encontrar la altura de un árbol binario.
6) Algoritmo de Kruskal para encontrar un árbol recubridor mínimo: Este algoritmo recursivo se utiliza para encontrar un subconjunto de las aristas de un grafo no dirigido que forman un árbol conectando todos los vértices y con peso total mínimo. El algoritmo funciona recursivamente uniendo los conjuntos de vértices conectados por las aristas de menor peso hasta que todos los vértices estén conectados.

Actividad 3

Pensar y desarrollar la propuesta de un programa que permita resolver una mazmorra en un juego de rol utilizando una matriz para representar el mapa de la mazmorra y una pila para almacenar los movimientos del jugador. En este se puede considerar lo siguiente:

1) Representar el mapa de la mazmorra mediante una matriz de enteros, caracteres u objetos, donde cada casilla de la matriz representa una celda o una habitación en la mazmorra.
2) Utilizar una pila para almacenar los movimientos del jugador en cada momento, permitiendo al jugador deshacer sus pasos si es necesario.
3) Utilizar un algoritmo de búsqueda como DFS (Profundidad de Primero) o BFS (Amplitud de Primero) para recorrer la matriz y encontrar la salida de la mazmorra.
4) A medida que el algoritmo recorre el mapa, se marcan las celdas visitadas para evitar ciclos infinitos.
5) El programa también podría incluir la implementación de reglas del juego, como la posibilidad de abrir puertas con llaves, luchar contra monstruos, etc.
6) Una vez que el algoritmo ha encontrado la salida, el programa utiliza la pila para reconstruir los pasos del jugador y mostrar el camino correcto.