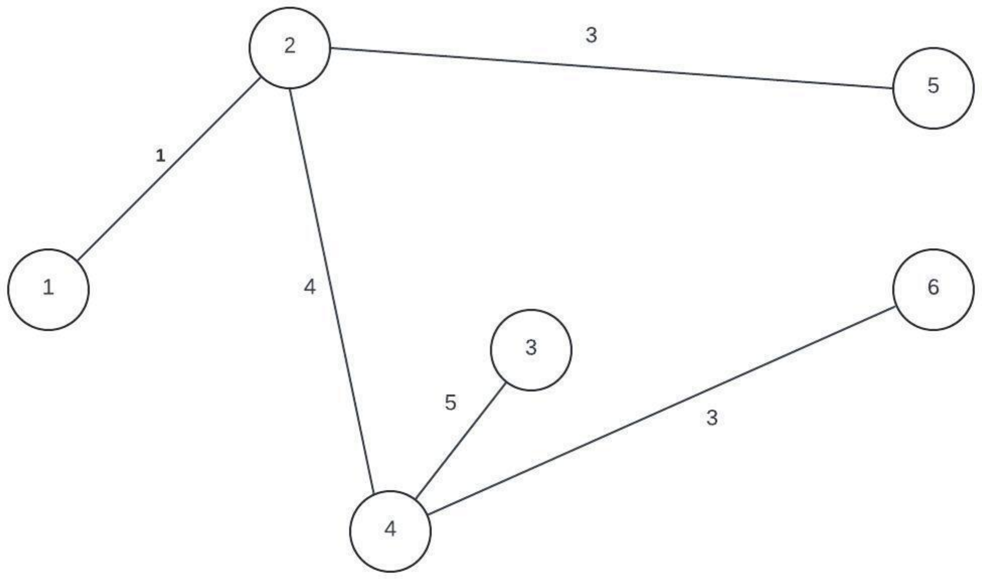
Task 1

1. Illustrate what is meant by minimum spanning tree and Dijkstra algorithms, apply both algorithms on the mentioned graph

- A **minimum spanning tree** (**MST**) or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

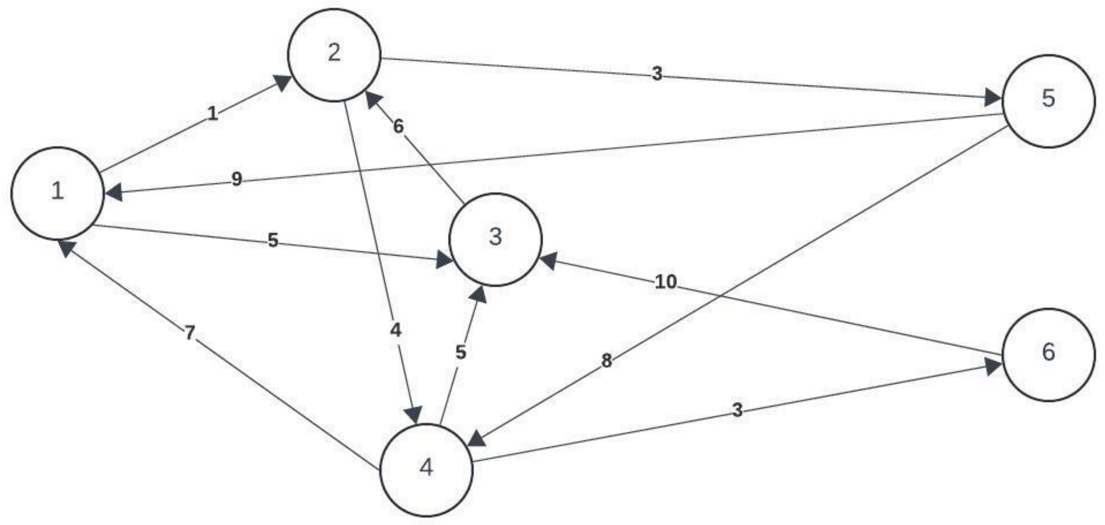
- **Dijkstra's algorithm** is a popular algorithms for solving many single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph.

- **Minimum Spanning Tree**



MST Cost = 16

- **Dijkstra's Algorithm**



Selected vertex 1

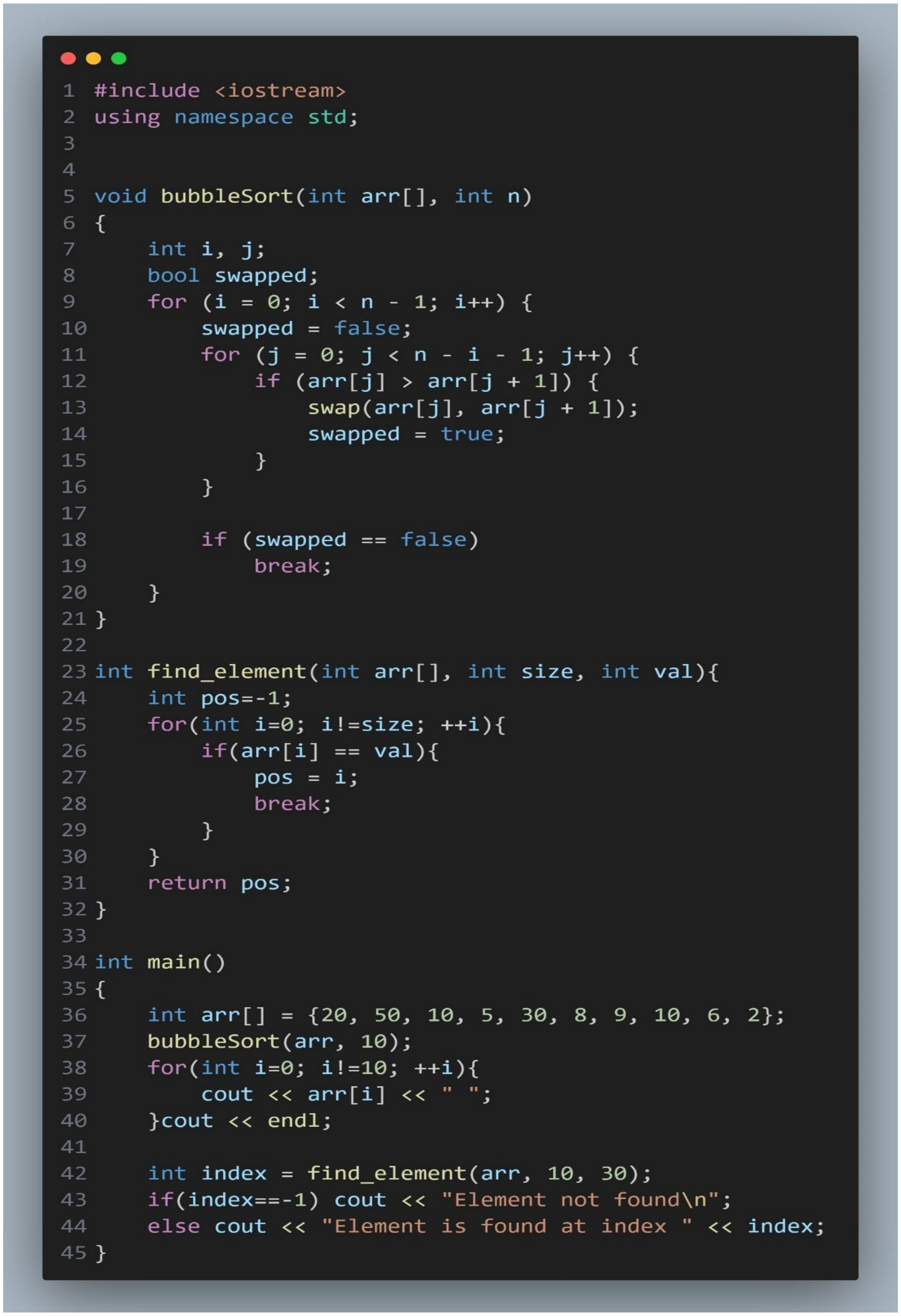
Select	1	2	3	4	5	6
1	0	1	5	∞	∞	∞
2	-	-	5	5	4	∞
5	-	-	5	5	-	∞
4	-	-	5	-	-	8
3	-	-	-	-	-	8
6	-	-	-	-	-	-

2. Differentiate between Graph and trees.

Trees are special cases of graphs. Graphs and trees are collections of nodes and edges, they differ in terms of connectivity, structure, presence of cycles, and applications. Trees are a special case of graphs with a hierarchical structure and no cycles.

Task 2

Assuming you have the above array, first apply any sorting technique to sort the array elements in C++ and apply a sequential search algorithm to search for the value 30



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


void bubbleSort(int arr[], int n)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }

        if (swapped == false)
            break;
    }
}

int find_element(int arr[], int size, int val){
    int pos=-1;
    for(int i=0; i!=size; ++i){
        if(arr[i] == val){
            pos = i;
            break;
        }
    }
    return pos;
}

int main()
{
    int arr[] = {20, 50, 10, 5, 30, 8, 9, 10, 6, 2};
    bubbleSort(arr, 10);
    for(int i=0; i!=10; ++i){
        cout << arr[i] << " ";
    }cout << endl;

    int index = find_element(arr, 10, 30);
    if(index==-1) cout << "Element not found\n";
    else cout << "Element is found at index " << index;
}
```

2. Illustrate how to calculate the complexity of the previous programs? And discuss how to evaluate the complexity of these algorithms?

```
int find_element(int arr[], int size, int val){
    int pos=-1; → 1
    for(int i=0; i!=size; ++i){ → n+1
        if(arr[i] == val){ → n
            pos = i;
            break;
        }
    }
    return pos;
}
```

<u>Space Complexity</u>

size	→ 1 word
val	→ 1 word
pos	→ 1 word
i	→ 1 word

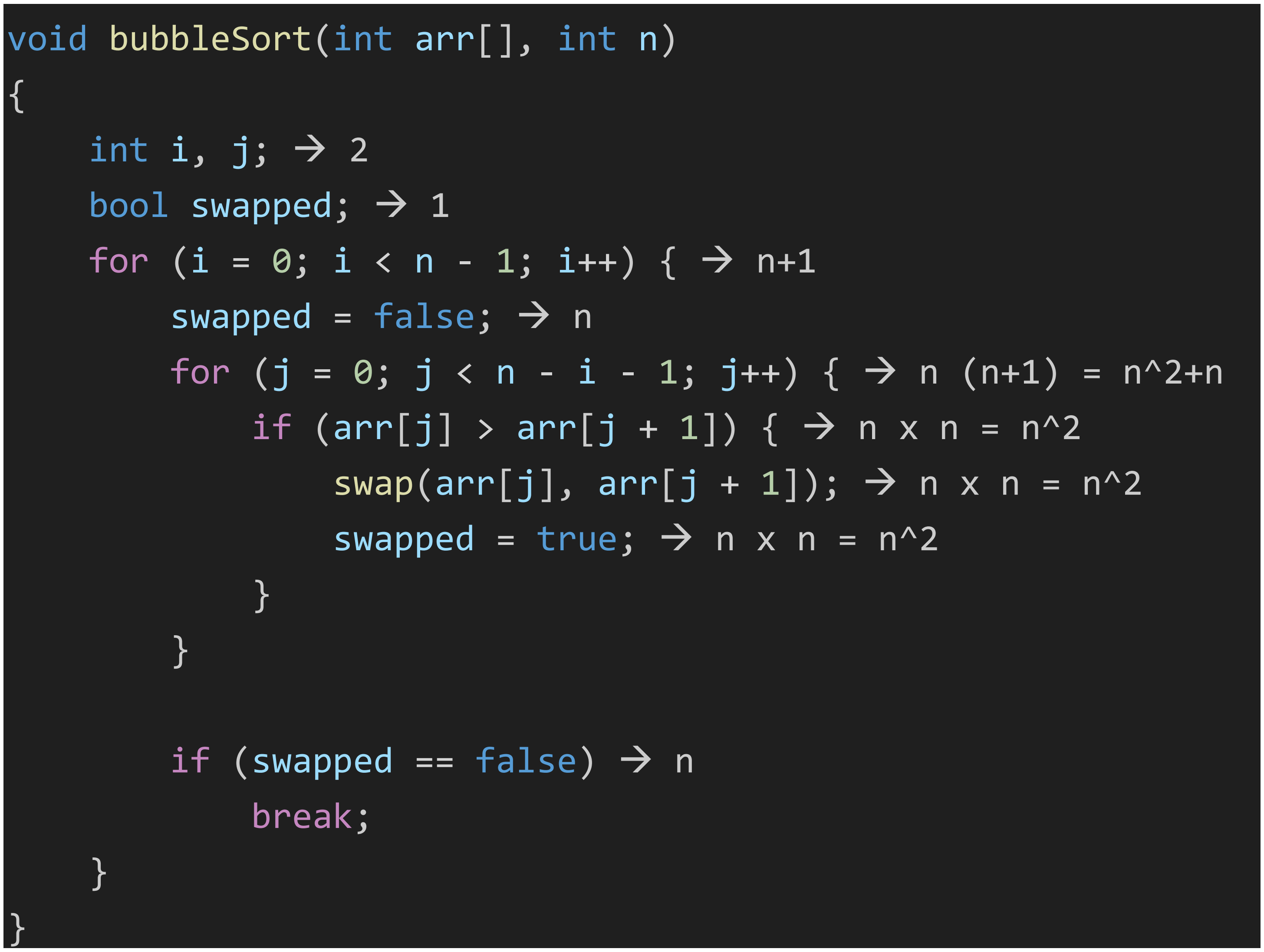
S (n) = 4

O (1)

<u>Time Complexity</u>

F (n) = 2n + 2

O (n)



```
void bubbleSort(int arr[], int n)
{
    int i, j; → 2
    bool swapped; → 1
    for (i = 0; i < n - 1; i++) { → n+1
        swapped = false; → n
        for (j = 0; j < n - i - 1; j++) { → n (n+1) = n^2+n
            if (arr[j] > arr[j + 1]) { → n x n = n^2
                swap(arr[j], arr[j + 1]); → n x n = n^2
                swapped = true; → n x n = n^2
            }
        }

        if (swapped == false) → n
            break;
    }
}
```

Space Complexity

n	→ 1 word
i	→ 1 word
j	→ 1 word
swapped	→ 1 word

S (n) = 4

O (1)

Time Complexity

$F(n) = 2+1+n+1+n+n^2+n+3n^2+n$

$= 4n^2 + 4n + 4$

$O(n^2)$

Discuss how to evaluate the complexity of these algorithms

1. **Bubble Sort**:
   - Time Complexity: The worst-case time complexity of Bubble Sort is O(n^2) where n is the number of elements in the array. This is because Bubble Sort involves nested loops where each iteration compares adjacent elements and swaps them if they are in the wrong order. In the worst case, the algorithm needs to iterate over the array multiple times.
   - Space Complexity: The space complexity of Bubble Sort is O(1) because it only uses a constant amount of extra space for temporary variables.
2. **Sequential Search**:
   - Time Complexity: The time complexity of Sequential Search is O(n) where n is the number of elements in the array. In the worst case, the algorithm may need to traverse the entire array to find the desired element.
   - Space Complexity: The space complexity of Sequential Search is O(1) because it only uses a constant amount of extra space for variables.

To evaluate the complexity of these algorithms, we consider:

- The number of operations performed as a function of the input size.
- How the algorithm's runtime scales with increasing input size.
- The presence of nested loops or recursive calls that affect the number of operations.
- Any optimizations or early termination conditions that may affect runtime.