Preparação para o teste de DA

Conteúdo

1. Asymptotic Complexity and Basic Data Structures:

1.1. Upper bound, lower bound and tight bound definitions. Given an analytical function that determine its asymptotic complexity. Compare two functions and determine which one will dominate for larger and small instance sizes.

In algorithm analysis, we use asymptotic notation to describe the performance of an algorithm. Asymptotic notation refers to how an algorithm behaves as the size of the input grows without limit. The three most common types of asymptotic notation are:

1. **Big O notation**: This represents the upper bound or worst-case scenario for the running time of an algorithm. It gives an upper limit on the number of operations required by the algorithm to solve the problem for any input of size n.

2. **Big Omega notation**: This represents the lower bound or best-case scenario for the running time of an algorithm. It gives a lower limit on the number of operations required by the algorithm to solve the problem for any input of size n.

3. **Big Theta notation**: This represents the tight bound or average-case scenario for the running time of an algorithm. It gives an estimate of the number of operations required by the algorithm to solve the problem for any input of size n.

Given an analytical function, we can determine its asymptotic complexity by looking at the growth rate of the function as the input size increases. We typically ignore lower-order terms and constants and focus on the dominant term or terms that grow the fastest.

To compare two functions and determine which one will dominate for larger and small instance sizes, we can look at their growth rates. If one function has a higher growth rate than the other, it will dominate for larger input sizes. For smaller input sizes, the dominating function may depend on the specific values of the input. We can also use the limit definition of asymptotic notation to compare two functions.

For example, consider the following two functions:

f(n) = 2n^2 + 5n + 1; g(n) = n^3 + 10n.

As n approaches infinity, the dominant term in f(n) is 2n^2, while the dominant term in g(n) is n^3. Therefore, g(n) will dominate f(n) for large input sizes. For small input sizes, however, f(n) may dominate g(n) depending on the specific values of n.

1.2. Size of an instance given its implementation data structure for a selected operation. For example, using adjacency matrix, determine the asymptotic complexity of a DFS traversal.

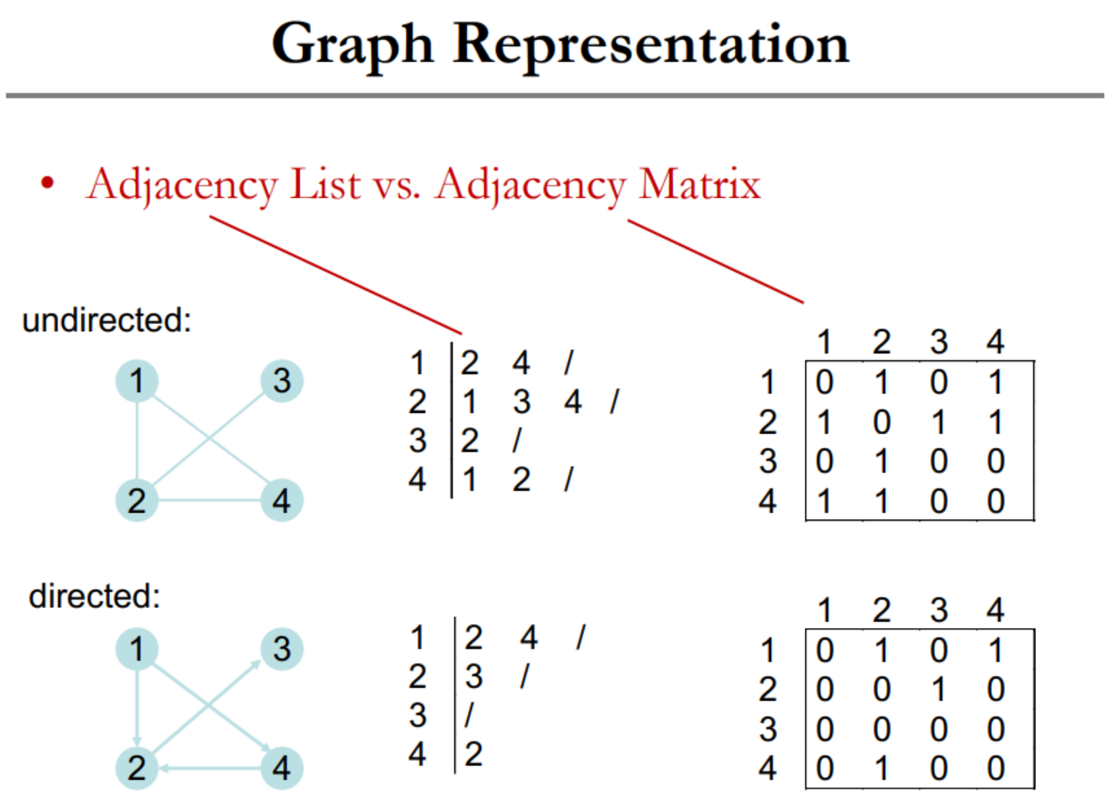
In computer science, the size of an instance refers to the amount of input data required to perform a given operation. For example, the size of an instance for a sorting algorithm may be the number of elements to be sorted, or for a graph traversal algorithm like DFS, it may be the number of vertices in the graph.

The adjacency matrix is a way of representing a graph as a matrix where the rows and columns correspond to the vertices, and the entries correspond to the edges. Specifically, if the graph has n vertices, then the adjacency matrix is an n-by-n matrix, where the entry in row i and column j is 1 if there is an edge from vertex i to vertex j, and 0 otherwise.

To analyze the time complexity of a DFS traversal using an adjacency matrix, we need to count the number of basic operations performed by the algorithm as a function of the size of the instance. The basic operations in a DFS traversal are visiting a vertex, marking it as visited, and recursively visiting its unvisited neighbors.

If we use a standard depth-first search algorithm, the time complexity of the algorithm can be expressed in terms of the number of vertices and edges in the graph. In particular, the time complexity of DFS on a graph represented by an adjacency matrix is O(V^2), where V is the number of vertices in the graph. This is because each vertex needs to be visited at most once, and for each vertex, we need to examine all V adjacent vertices in the matrix.

It's worth noting that if the graph is sparse, i.e., it has relatively few edges compared to the number of vertices, then it may be more efficient to use an adjacency list representation instead of an adjacency matrix. In that case, the time complexity of DFS would be O (V + E), where E is the number of edges in the graph.



Graph Representation

- Adjacency List vs. Adjacency Matrix

undirected:

```
1 |2  4  /
2 |1  3  4  /
3 |2  /
4 |1  2  /
```

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 0 & 1 & 0 & 0 \\ 4 & 1 & 1 & 0 & 0 \end{array}$$

directed:

```
1 |2  4  /
2 |3  /
3 |/
4 |2
```

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 \end{array}$$

1.3. The min-heap as a key data structure. Complexity of its key operations *siftDown* and *siftUp* (you do not need by heart the pseudo-code of these basic operations, but need to understand how they operate). Given a sequence of value insertion determine the configuration of a minheap or max-heap and reason about the complexity of the sequence of operations.

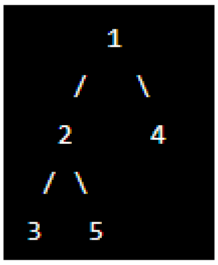
A min-heap is a binary tree in which the parent node is always less than or equal to its children. It is a key data structure used for sorting and priority queue implementations. The siftUp and siftDown operations are used to maintain the heap property after a node is inserted or deleted.

The siftUp operation is used after an element is inserted into the heap. It compares the element with its parent and swaps them if the parent is larger. This operation is repeated until the parent is smaller or until the root is reached. The complexity of the siftUp operation is O(log n), where n is the number of nodes in the heap.

The siftDown operation is used after an element is deleted from the heap. It swaps the deleted node with its smallest child until the heap property is restored. This operation is repeated until the smallest child is greater than the node or until the leaf is reached. The complexity of the siftDown operation is also O(log n).

To determine the configuration of a min-heap or max-heap after a sequence of value insertions, we start with an empty heap and add each element one at a time using the siftUp operation. The final configuration of the heap depends on the order in which the elements were inserted.

For example, if we insert the elements [3, 2, 4, 1, 5] into an empty min-heap, the resulting heap would be:



```
      1
    /   \
   2     4
  / \
 3   5
```

The complexity of this sequence of insertions is O (n log n), where n is the number of elements inserted, since each insertion requires a siftUp operation with a complexity of O (log n).

Here is the pseudocode for the siftDown and siftUp operations in a min-heap:

```
siftDown(array, index):
    leftChildIndex = 2 * index + 1
    rightChildIndex = 2 * index + 2
    smallestIndex = index

    if leftChildIndex < array.length and array[leftChildIndex] < array[smallestIndex]:
        smallestIndex = leftChildIndex

    if rightChildIndex < array.length and array[rightChildIndex] < array[smallestIndex]:
        smallestIndex = rightChildIndex

    if smallestIndex != index:
        swap(array[index], array[smallestIndex])
        siftDown(array, smallestIndex)

siftUp(array, index):
    parentIndex = (index - 1) / 2

    if parentIndex >= 0 and array[parentIndex] > array[index]:
        swap(array[index], array[parentIndex])
        siftUp(array, parentIndex)
```

And here's the pseudocode for inserting values into a min-heap:

```
minHeapInsert(array, value):
    array.push(value)
    siftUp(array, array.length - 1)
```

The complexity of the sequence of operations depends on the number of elements inserted and the order in which they are inserted. The worst-case time complexity for inserting n elements into a min-heap is O(nlog(n)), as each element may require sifting up the entire height of the tree. However, if the elements are inserted in a partially sorted order (e.g., already sorted or reverse sorted), the complexity can be reduced to O(n) as each element only needs to be sifted up one level.

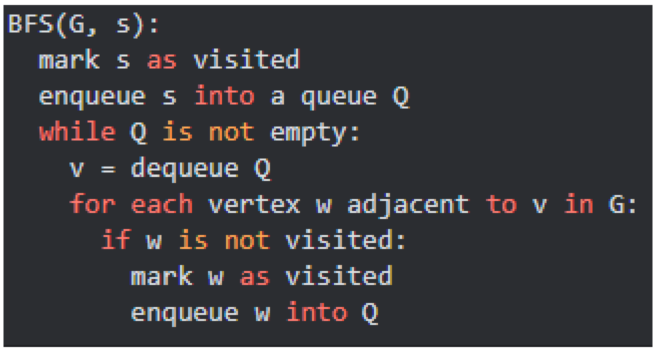
2. Graph Algorithms:

2.1. BFS and DFS labelling of nodes and asymptotic complexity. Given a graph understand the order in which the nodes are visited and the edges are explored.

BFS and DFS are two algorithms used for traversing graphs. Both algorithms visit each node in the graph, but they visit them in different orders. BFS stands for Breadth-First Search, while DFS stands for Depth-First Search.

In BFS, we start at a node, visit all its neighbors, then visit all the neighbors of those neighbors, and so on. We keep track of the nodes we have visited so that we don't visit them again. We use a queue data structure to keep track of the order in which we visit nodes. The order in which we visit nodes using BFS is such that nodes at the same level in the graph are visited before nodes at the next level.

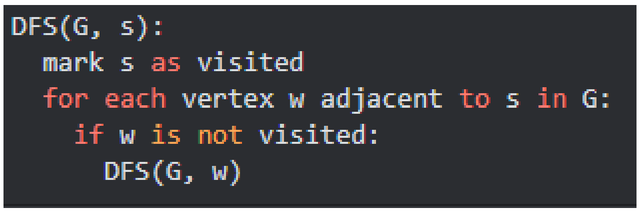
Here's the pseudo code for BFS traversal:



```
BFS(G, s):
  mark s as visited
  enqueue s into a queue Q
  while Q is not empty:
    v = dequeue Q
    for each vertex w adjacent to v in G:
      if w is not visited:
        mark w as visited
        enqueue w into Q
```

In DFS, we start at a node, visit one of its neighbors, then visit one of that neighbor's neighbors, and so on. We keep track of the nodes we have visited so that we don't visit them again. We use a stack data structure to keep track of the order in which we visit nodes. The order in which we visit nodes using DFS is such that we go as deep as possible before backtracking.

Here's the pseudo code for DFS traversal:



```
DFS(G, s):
  mark s as visited
  for each vertex w adjacent to s in G:
    if w is not visited:
      DFS(G, w)
```

The asymptotic complexity of both algorithms depends on the size of the graph and the way it is represented. For example, if the graph is represented using an adjacency list, BFS and DFS both have a time complexity of O(V + E), where V is the number of vertices and E is the number of edges in the graph. If the graph is represented using an adjacency matrix, BFS and DFS both have a time complexity of O(V^2).

In terms of the order in which nodes are visited and edges are explored, BFS visits nodes in order of their distance from the starting node, while DFS explores the edges of the graph in a depth-first manner, meaning it visits all the edges of one branch before moving on to another branch.

2.2. Topological sorting and its complexity. Given a directed graph, without cycles, determine the order of a possible topological sorting and its asymptotic complexity.

The function takes a directed acyclic graph represented as an adjacency list **graph** as input and returns a vector **sorted** representing a possible topological sorting of the vertices in the graph.

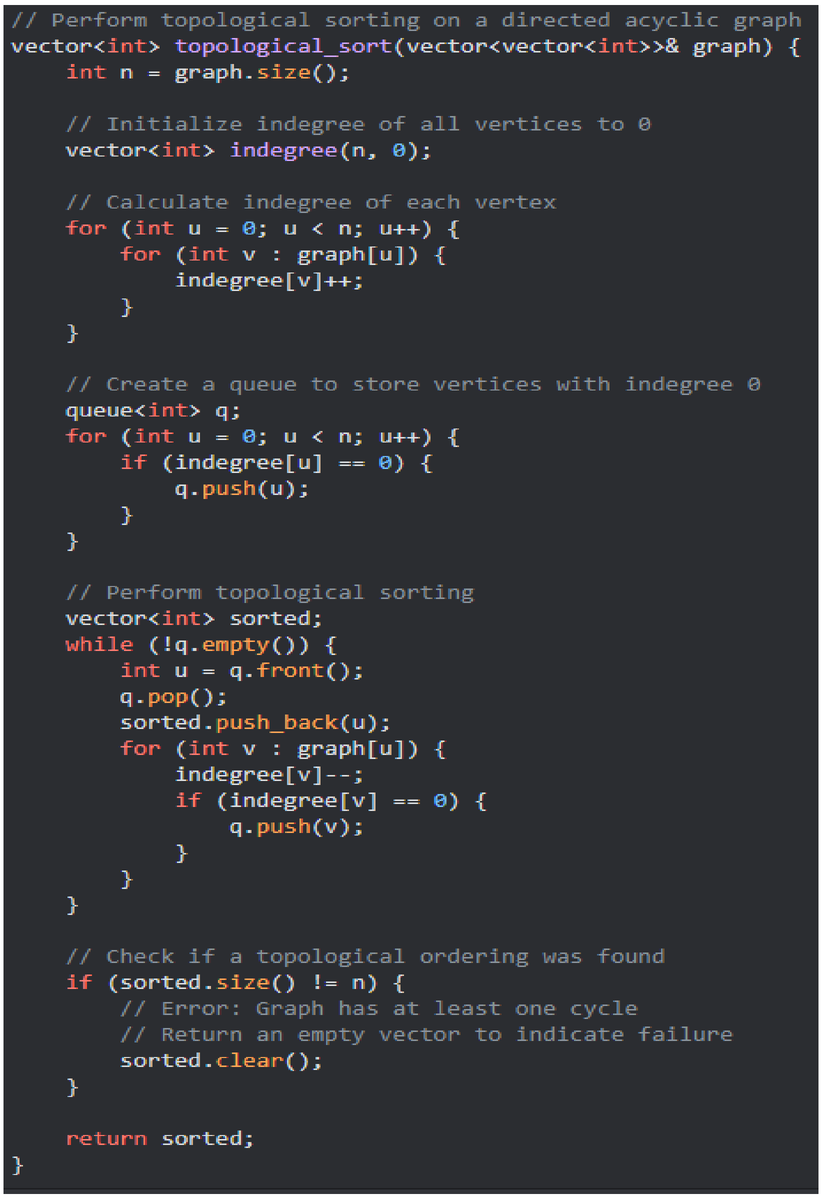
The algorithm works by first initializing the indegree of all vertices to 0 and then calculating the indegree of each vertex by iterating through the edges in the graph. The algorithm then creates a queue to store vertices with indegree 0, i.e., vertices that have no incoming edges.

The algorithm then performs a BFS-like traversal of the graph by repeatedly dequeuing a vertex **u** with indegree 0, adding it to the sorted list, and then reducing the indegree of its neighbors by 1. If any of the neighbors of **u** now have indegree 0, they are added to the queue.

The algorithm continues until all vertices have been added to the sorted list or until the queue is empty. If a topological sorting is found, the function returns the sorted list. If the sorted list has fewer than **n** vertices, then the graph contains at least one cycle, and the function returns an empty vector to indicate failure.

The asymptotic complexity of the algorithm is O(V + E), where V is the number of vertices and E is the number of edges in the graph, as the algorithm needs to visit each vertex and each edge exactly once.

Here's the pseudo code in C++ for topological sorting with comments explaining what's happening:

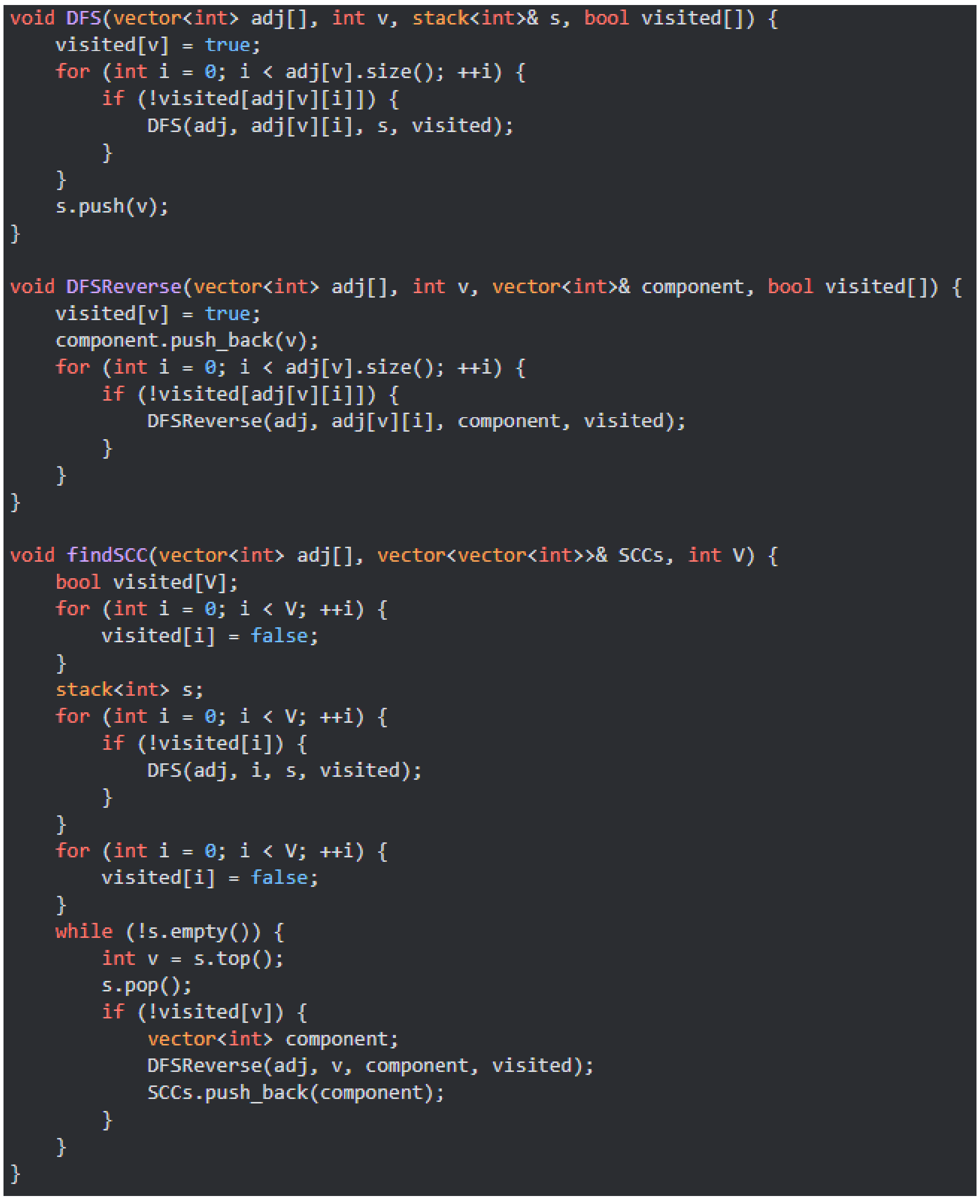


```
// Perform topological sorting on a directed acyclic graph
vector<int> topological_sort(vector<vector<int>>& graph) {
    int n = graph.size();

    // Initialize indegree of all vertices to 0
    vector<int> indegree(n, 0);

    // Calculate indegree of each vertex
    for (int u = 0; u < n; u++) {
        for (int v : graph[u]) {
            indegree[v]++;
        }
    }

    // Create a queue to store vertices with indegree 0
    queue<int> q;
    for (int u = 0; u < n; u++) {
        if (indegree[u] == 0) {
            q.push(u);
        }
    }

    // Perform topological sorting
    vector<int> sorted;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        sorted.push_back(u);
        for (int v : graph[u]) {
            indegree[v]--;
            if (indegree[v] == 0) {
                q.push(v);
            }
        }
    }

    // Check if a topological ordering was found
    if (sorted.size() != n) {
        // Error: Graph has at least one cycle
        // Return an empty vector to indicate failure
        sorted.clear();
    }

    return sorted;
}
```

2.3. Understand and apply the Strongly-Connected components algorithm that replies on a double pass over the graph (and its Reverse variant) and determine the overall asymptotic complexity. Application of this algorithm to an example graph.

The Strongly-Connected components algorithm is used to identify the strongly-connected components of a directed graph. A strongly-connected component is a subgraph where every vertex is reachable from every other vertex in the subgraph. The algorithm works by performing a depth-first search (DFS) on the graph, and then performing a second DFS on the reverse of the graph, which is the same graph with all edges reversed. The vertices are then sorted in order of their finishing times from the second DFS, and this order is used to group the vertices into strongly-connected components.

Here is the pseudocode for the Strongly-Connected components algorithm in C++:



```cpp
void DFS(vector<int> adj[], int v, stack<int>& s, bool visited[]) {
    visited[v] = true;
    for (int i = 0; i < adj[v].size(); ++i) {
        if (!visited[adj[v][i]]) {
            DFS(adj, adj[v][i], s, visited);
        }
    }
    s.push(v);
}

void DFSReverse(vector<int> adj[], int v, vector<int>& component, bool visited[]) {
    visited[v] = true;
    component.push_back(v);
    for (int i = 0; i < adj[v].size(); ++i) {
        if (!visited[adj[v][i]]) {
            DFSReverse(adj, adj[v][i], component, visited);
        }
    }
}

void findSCC(vector<int> adj[], vector<vector<int>>& SCCs, int V) {
    bool visited[V];
    for (int i = 0; i < V; ++i) {
        visited[i] = false;
    }
    stack<int> s;
    for (int i = 0; i < V; ++i) {
        if (!visited[i]) {
            DFS(adj, i, s, visited);
        }
    }
    for (int i = 0; i < V; ++i) {
        visited[i] = false;
    }
    while (!s.empty()) {
        int v = s.top();
        s.pop();
        if (!visited[v]) {
            vector<int> component;
            DFSReverse(adj, v, component, visited);
            SCCs.push_back(component);
        }
    }
}
```

Here is a brief explanation of what each function does:

- **DFS**: performs a DFS on the graph to find the vertices in the order of their finishing times.
- **DFSReverse**: performs a DFS on the reverse of the graph to find the strongly-connected component.
- **findSCC**: performs the entire algorithm by first performing a DFS on the graph to find the finishing times, and then performing a DFS on the reverse of the graph to find the strongly-connected components.

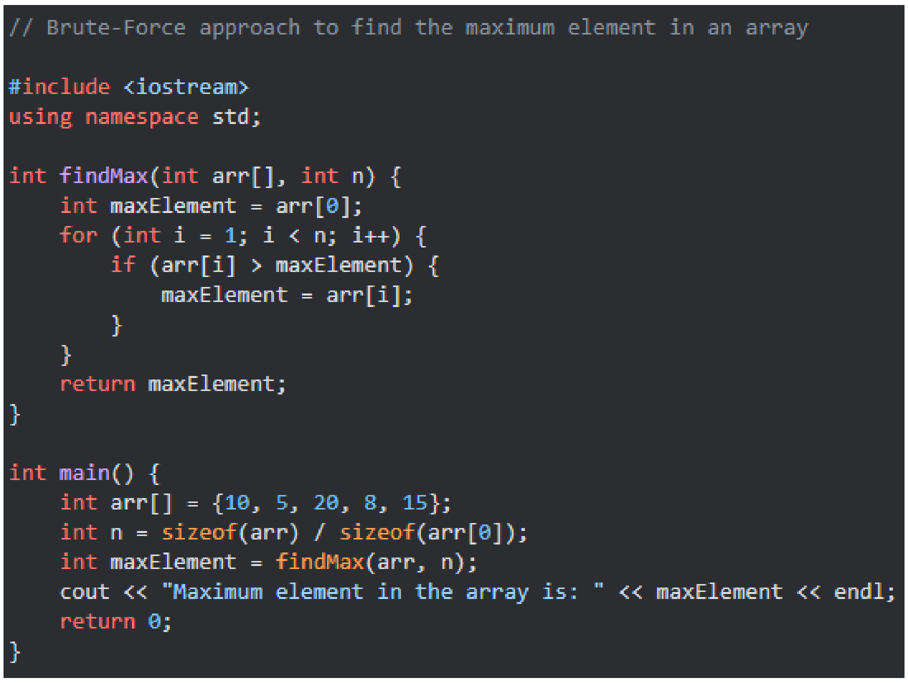
The overall asymptotic complexity of this algorithm is O(V+E), where V is the number of vertices and E is the number of edges in the graph. This is because we perform two DFS traversals over the graph and its reverse, each taking O(V+E) time.

To apply this algorithm to an example graph, we can create an adjacency list representation of the graph and pass it to the **findSCC** function. This will return a vector of strongly-connected components, where each component is represented as a vector of vertices.

3. Brute-Force Approach:

3.1. You will need to reason about the use of the Brute-Force approach in the development of an algorithm derive a simple pseudo-code implementation and determine its asymptotic complexity for a selected choice of a data structure. (see final note on pseudo-code).

The Brute-Force approach is a simple and straightforward way of solving problems that involves trying all possible solutions and selecting the best one. Here's an example of how it can be applied to find the maximum element in an array:



```
// Brute-Force approach to find the maximum element in an array

#include <iostream>
using namespace std;

int findMax(int arr[], int n) {
    int maxElement = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > maxElement) {
            maxElement = arr[i];
        }
    }
    return maxElement;
}

int main() {
    int arr[] = {10, 5, 20, 8, 15};
    int n = sizeof(arr) / sizeof(arr[0]);
    int maxElement = findMax(arr, n);
    cout << "Maximum element in the array is: " << maxElement << endl;
    return 0;
}
```

In this code, we define a function **findMax** that takes an array **arr** and its size **n** as input and returns the maximum element in the array. We start by initializing a variable **maxElement** with the first element of the array. We then loop through the rest of the array and compare each element with the **maxElement** variable. If we find an element that is greater than the **maxElement**, we update **maxElement** with the new element. Finally, we return the **maxElement**.

The time complexity of this Brute-Force approach is O(n), where n is the size of the array since we need to compare each element in the array with the **maxElement** variable.

Note that this is just a simple example of how the Brute-Force approach can be applied to solve a problem. In general, this approach may not be the most efficient or practical way of solving complex problems, but it can be a useful starting point for developing more optimized algorithms.

4. Greedy Algorithms:

4.1. The two properties of an optimal greedy choice and optimal solution to a sub-problem. Understanding the two properties what how to prove them for a selected given greedy algorithm (Note: you will not be asked detailed proofs of these greedy properties, but rather you will be asked to reason about them for selected covered algorithms or a sample algorithm provided in the test).

The two properties of an optimal greedy choice and optimal solution to a sub-problem are as follows:

1. Greedy choice property: A globally optimal solution can be obtained by making a locally optimal (greedy) choice.
2. Optimal substructure property: A problem can be solved optimally by breaking it down into subproblems and solving each subproblem optimally.

To prove these properties for a given greedy algorithm, we need to demonstrate that the algorithm satisfies these properties.

Let's take the example of the Activity Selection problem to illustrate this. In this problem, we are given a set of activities, each with a start time and an end time, and we need to select the maximum number of non-overlapping activities that can be performed.

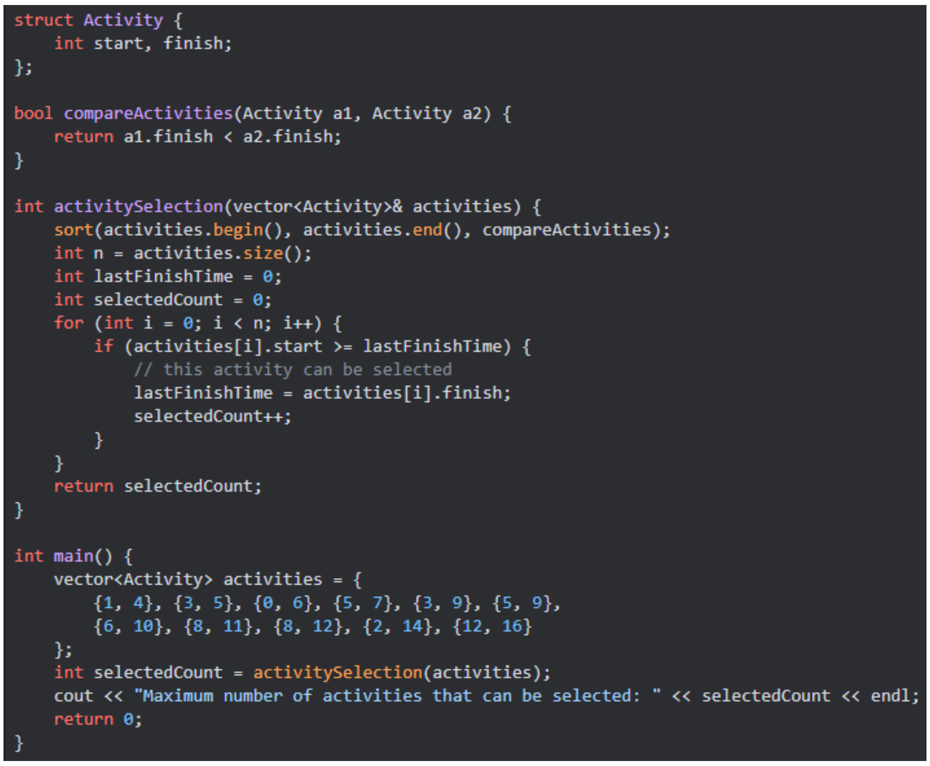
Here's the greedy algorithm for solving this problem:

1. Sort the activities by their end times.
2. Select the first activity in the sorted list.
3. For each subsequent activity in the sorted list, if its start time is greater than or equal to the end time of the previously selected activity, select it and update the end time.

To prove the two properties of optimal greedy choice and optimal substructure for this algorithm, we can reason as follows:

1. **Greedy choice property**: At each step, the algorithm selects the activity that has the earliest end time among the remaining activities. This is a locally optimal choice, as it ensures that we can select the maximum number of non-overlapping activities starting from the current activity. By selecting the activity with the earliest end time, we leave room for selecting other activities that start later. We can show that this locally optimal choice leads to a globally optimal solution by contradiction. Suppose there exists a globally optimal solution that does not include the activity with the earliest end time. We can always replace this activity with the one with the earliest end time and obtain a new solution that is at least as good as the previous one.
2. **Optimal substructure property**: We can break the problem of selecting the maximum number of non-overlapping activities into subproblems by considering the activities starting from the second one and selecting the maximum number of non-overlapping activities from this subset. This is a subproblem of the original problem, and we can solve it optimally using the same algorithm recursively. The optimal solution to the original problem can then be obtained by adding the first activity to the solution of the subproblem.

Here's the pseudo code implementation of the Activity Selection problem using the above greedy algorithm:



```cpp
struct Activity {
    int start, finish;
};

bool compareActivities(Activity a1, Activity a2) {
    return a1.finish < a2.finish;
}

int activitySelection(vector<Activity>& activities) {
    sort(activities.begin(), activities.end(), compareActivities);
    int n = activities.size();
    int lastFinishTime = 0;
    int selectedCount = 0;
    for (int i = 0; i < n; i++) {
        if (activities[i].start >= lastFinishTime) {
            // this activity can be selected
            lastFinishTime = activities[i].finish;
            selectedCount++;
        }
    }
    return selectedCount;
}

int main() {
    vector<Activity> activities = {
        {1, 4}, {3, 5}, {0, 6}, {5, 7}, {3, 9}, {5, 9},
        {6, 10}, {8, 11}, {8, 12}, {2, 14}, {12, 16}
    };
    int selectedCount = activitySelection(activities);
    cout << "Maximum number of activities that can be selected: " << selectedCount << endl;
    return 0;
}
```

The **Activity** struct represents an activity with a start and finish time. The **compareActivities** function is used to sort the activities in increasing order of finish time.

The **activitySelection** function takes a vector of activities and returns the maximum number of activities that can be selected. It first sorts the activities by finish time using the **sort** function. It then initializes **lastFinishTime** to 0 and **selectedCount** to 0. It then iterates over the sorted activities and selects each activity whose start time is greater than or equal to **lastFinishTime**. For each selected activity, it updates **lastFinishTime** to the activity's finish time and increments **selectedCount**. Finally, it returns **selectedCount**.

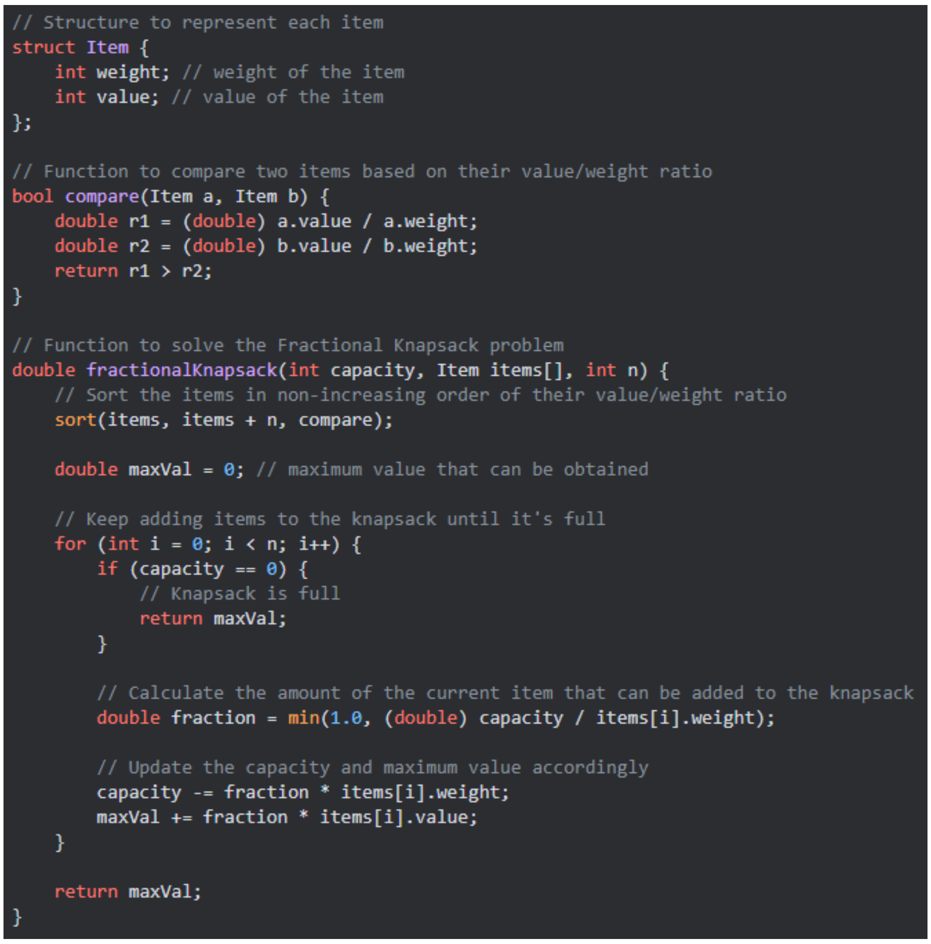
In the **main** function, we create a vector of sample activities and call the **activitySelection** function to determine the maximum number of activities that can be selected.

4.2. Fractional Knapsack algorithm and why it produces an optimal solution. Given an example, determine the outcome of the algorithm in terms of the sequence of selected items. Illustrate why the same greedy algorithm discussed in class does not work optimally for the case of the integer Knapsack problem.

Here's how the algorithm works:

1. The items are sorted in non-increasing order of their value/weight ratio using the **compare** function.
2. The maximum value that can be obtained is initialized to 0.
3. For each item, the amount of the item that can be added to the knapsack is calculated as the minimum of 1 and the remaining capacity divided by the item's weight.
4. The capacity and maximum value are updated accordingly.
5. The algorithm returns the maximum value that can be obtained.

The Fractional Knapsack algorithm produces an optimal solution because it always selects the item with the highest value/weight ratio, which maximizes the value per unit of weight. However, the same greedy algorithm does not work optimally for the case of the integer Knapsack problem because it may select an item with a lower value/weight ratio if it fits better in the knapsack, which may not result in the optimal solution.



```cpp
// Structure to represent each item
struct Item {
    int weight; // weight of the item
    int value; // value of the item
};

// Function to compare two items based on their value/weight ratio
bool compare(Item a, Item b) {
    double r1 = (double) a.value / a.weight;
    double r2 = (double) b.value / b.weight;
    return r1 > r2;
}

// Function to solve the Fractional Knapsack problem
double fractionalKnapsack(int capacity, Item items[], int n) {
    // Sort the items in non-increasing order of their value/weight ratio
    sort(items, items + n, compare);

    double maxVal = 0; // maximum value that can be obtained

    // Keep adding items to the knapsack until it's full
    for (int i = 0; i < n; i++) {
        if (capacity == 0) {
            // Knapsack is full
            return maxVal;
        }

        // Calculate the amount of the current item that can be added to the knapsack
        double fraction = min(1.0, (double) capacity / items[i].weight);

        // Update the capacity and maximum value accordingly
        capacity -= fraction * items[i].weight;
        maxVal += fraction * items[i].value;
    }

    return maxVal;
}
```

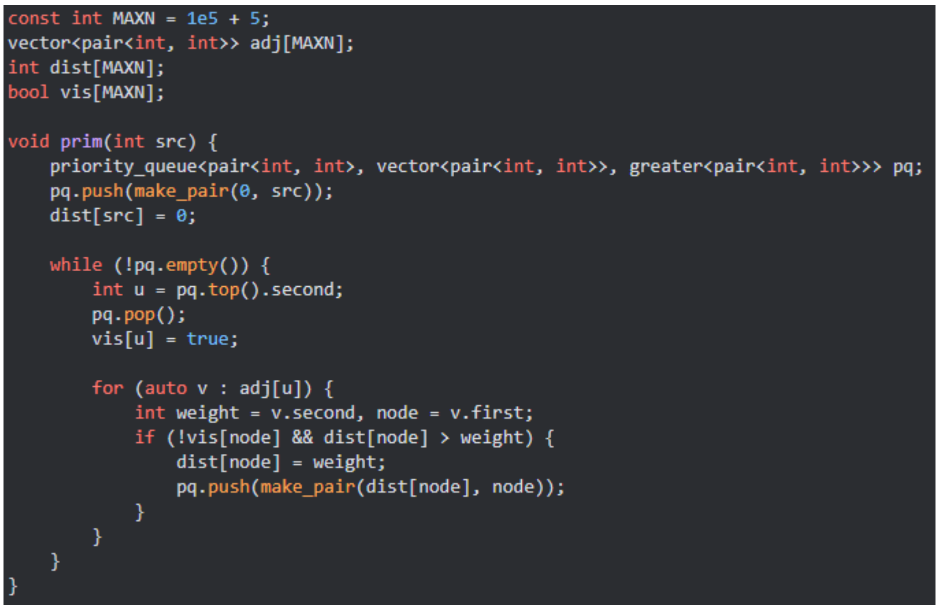
4.3. Minimum-Cost Spanning Tree algorithms (Prim and Kruskal) and their implementation time complexity as well as examples of application in restricted graphs. Properties of the algorithms in terms of data structures that best suit their implementation. Proof of correctness of the construction of the MST (choice of the lightest and safe edge) as well as regarding the two greedy algorithm properties. For restricted graphs, explore or develop possibly simpler algorithms and reason about their correctness and asymptotic time complexity.

Minimum-Cost Spanning Tree (MCST) algorithms aim to find the tree with the minimum total weight in each connected and weighted graph. Two well-known algorithms for finding the MCST are Prim's algorithm and Kruskal's algorithm.

1. Prim's Algorithm:

- Choose an arbitrary vertex from the graph and add it to the tree.
- From the set of vertices not yet included in the tree, choose the one that has the minimum distance to the tree and add it to the tree.
- Repeat step 2 until all vertices are included in the tree.

Here's the C++ implementation of Prim's Algorithm using a priority queue and adjacency list:

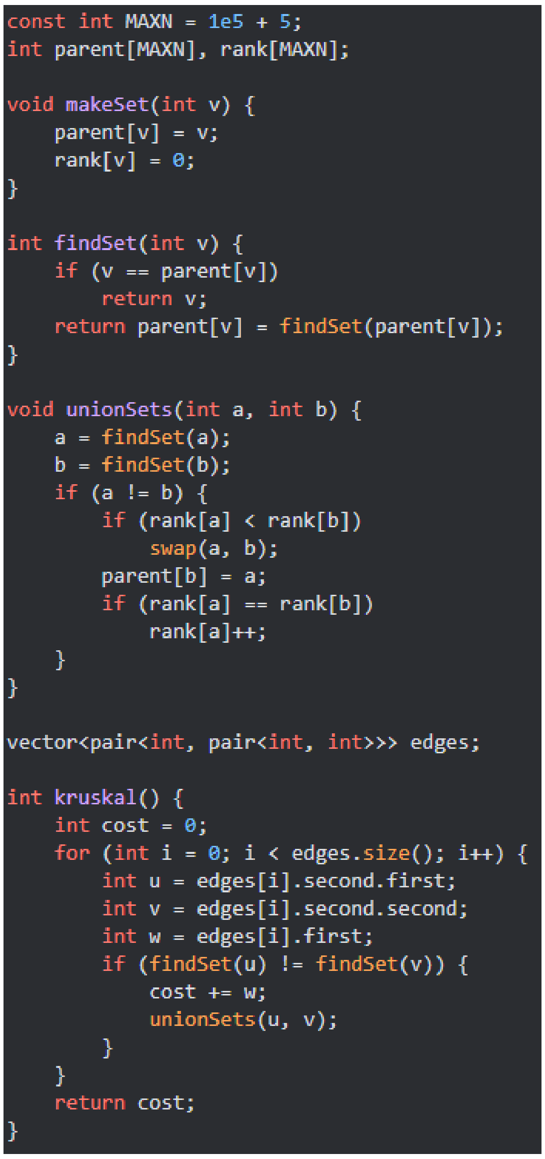


```cpp
const int MAXN = 1e5 + 5;
vector<pair<int, int>> adj[MAXN];
int dist[MAXN];
bool vis[MAXN];

void prim(int src) {
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    pq.push(make_pair(0, src));
    dist[src] = 0;

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
        vis[u] = true;

        for (auto v : adj[u]) {
            int weight = v.second, node = v.first;
            if (!vis[node] && dist[node] > weight) {
                dist[node] = weight;
                pq.push(make_pair(dist[node], node));
            }
        }
    }
}
```

2. Kruskal's Algorithm:

- Sort all the edges in non-decreasing order of their weight.
- Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge.
- Repeat step 2 until there are (V-1) edges in the spanning tree.

Here's the C++ implementation of Kruskal's Algorithm using the Union-Find algorithm:



```cpp
const int MAXN = 1e5 + 5;
int parent[MAXN], rank[MAXN];

void makeSet(int v) {
    parent[v] = v;
    rank[v] = 0;
}

int findSet(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = findSet(parent[v]);
}

void unionSets(int a, int b) {
    a = findSet(a);
    b = findSet(b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}

vector<pair<int, pair<int, int>>> edges;

int kruskal() {
    int cost = 0;
    for (int i = 0; i < edges.size(); i++) {
        int u = edges[i].second.first;
        int v = edges[i].second.second;
        int w = edges[i].first;
        if (findSet(u) != findSet(v)) {
            cost += w;
            unionSets(u, v);
        }
    }
    return cost;
}
```

The time complexity of Prim's algorithm for finding the minimum spanning tree of a graph is O(V^2) with an adjacency matrix representation and O(E log V) with an adjacency list representation, where V is the number of vertices and E is the number of edges in the graph.

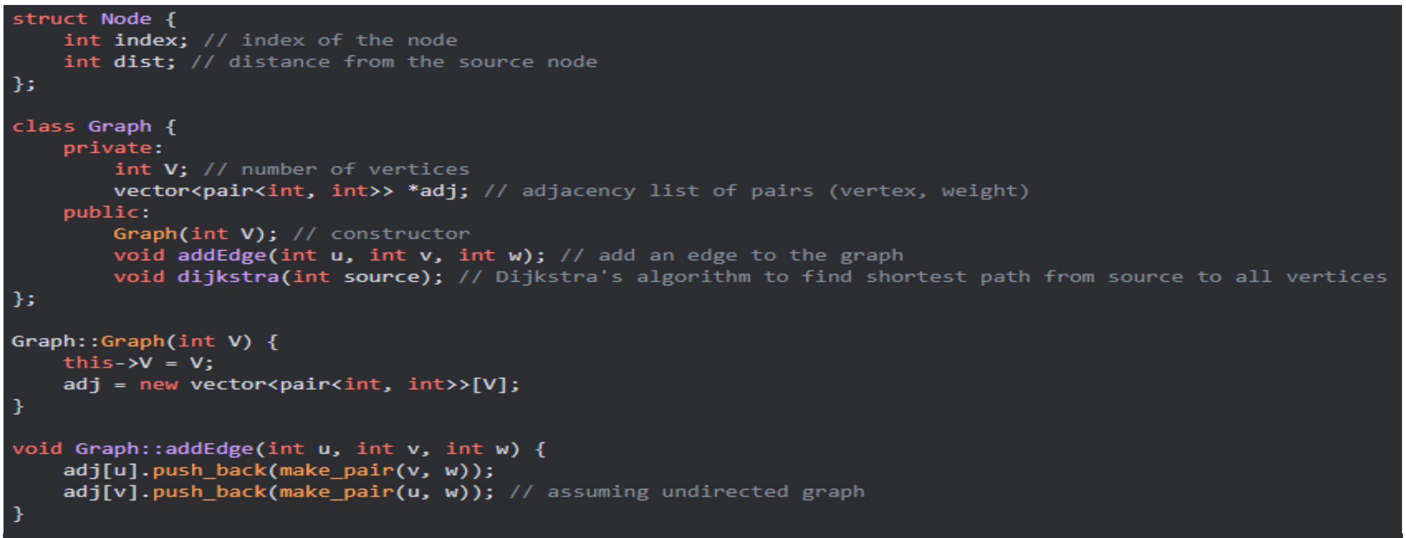
On the other hand, Kruskal's algorithm has a time complexity of O(E log E) or O(E log V), whichever is smaller, where E is the number of edges in the graph and V is the number of vertices.

In general, Kruskal's algorithm is faster than Prim's algorithm when the graph is sparse (i.e., has relatively few edges compared to the number of vertices), while Prim's algorithm is faster when the graph is dense (i.e., has relatively many edges compared to the number of vertices). However, the actual time complexity depends on the implementation and the specific characteristics of the input graph.

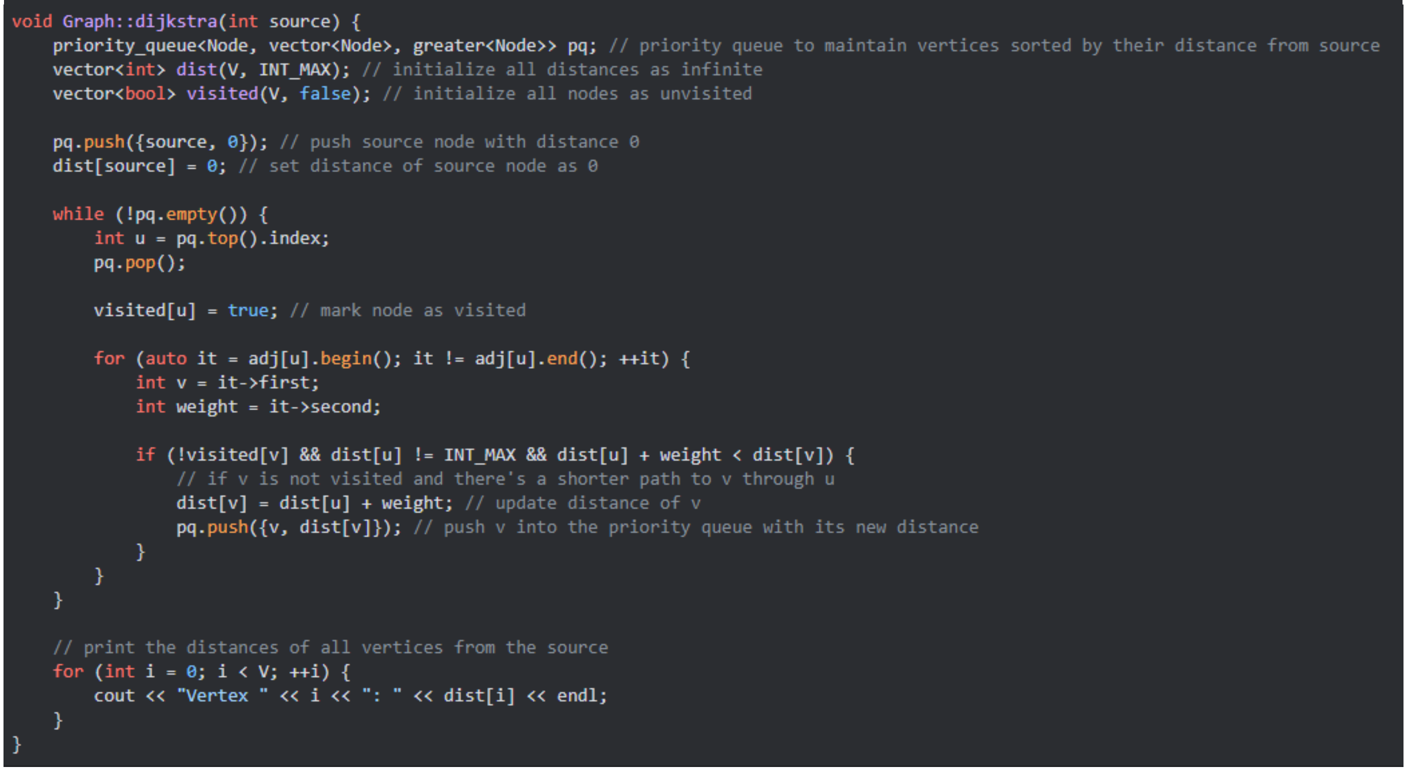
4.4. Dijkstra Shortest path algorithm and why it produces an optimal shortest path from a single source. Examples of application in special cases for DAGs or graphs with restricted edge weights where simpler algorithm could be developed. Correctness and proof of greedy properties in general or in specific (restricted graphs).

Dijkstra's Algorithm is a greedy algorithm that solves the shortest path problem for a weighted directed graph with non-negative edge weights, producing a shortest path tree. It works by maintaining a set of vertices whose shortest distance from the source node is known, and a set of vertices whose shortest distance is unknown. At each step, it selects the vertex with the smallest known distance from the source node and updates the distances of its adjacent vertices if a shorter path is found.

Here's the pseudocode:



```
struct Node {
    int index; // index of the node
    int dist; // distance from the source node
};

class Graph {
    private:
        int V; // number of vertices
        vector<pair<int, int>> *adj; // adjacency list of pairs (vertex, weight)
    public:
        Graph(int V); // constructor
        void addEdge(int u, int v, int w); // add an edge to the graph
        void dijkstra(int source); // Dijkstra's algorithm to find shortest path from source to all vertices
};

Graph::Graph(int V) {
    this->V = V;
    adj = new vector<pair<int, int>>[V];
}

void Graph::addEdge(int u, int v, int w) {
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w)); // assuming undirected graph
}
```



```
void Graph::dijkstra(int source) {
    priority_queue<Node, vector<Node>, greater<Node>> pq; // priority queue to maintain vertices sorted by their distance from source
    vector<int> dist(V, INT_MAX); // initialize all distances as infinite
    vector<bool> visited(V, false); // initialize all nodes as unvisited

    pq.push({source, 0}); // push source node with distance 0
    dist[source] = 0; // set distance of source node as 0

    while (!pq.empty()) {
        int u = pq.top().index;
        pq.pop();

        visited[u] = true; // mark node as visited

        for (auto it = adj[u].begin(); it != adj[u].end(); ++it) {
            int v = it->first;
            int weight = it->second;

            if (!visited[v] && dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
                // if v is not visited and there's a shorter path to v through u
                dist[v] = dist[u] + weight; // update distance of v
                pq.push({v, dist[v]}); // push v into the priority queue with its new distance
            }
        }
    }

    // print the distances of all vertices from the source
    for (int i = 0; i < V; ++i) {
        cout << "Vertex " << i << ": " << dist[i] << endl;
    }
}
```

In the above code, we define a **Node** struct to store the index of the node and its distance from the source node. We also define a **Graph** class with a constructor to initialize the number of vertices, an **addEdge** function to add edges to the graph, and a **dijkstra** function to implement Dijkstra's Algorithm to find the shortest path from a source node to all other vertices.

In the **dijkstra** function, we initialize a priority queue to maintain the vertices sorted by their distance from the source node, a vector to store the distances of all vertices from the source (initialized as **INT_MAX**), and a boolean array to keep track of the vertices visited so far (initialized as false). We also set the distance of the source vertex to itself as 0.

Then, we start a loop while the priority queue is not empty. In each iteration of the loop, we extract the vertex with the minimum distance value from the priority queue. We mark this vertex as visited and iterate over its adjacent vertices. For each adjacent vertex, we update its distance value in the distances vector if it can be improved by going through the current vertex.

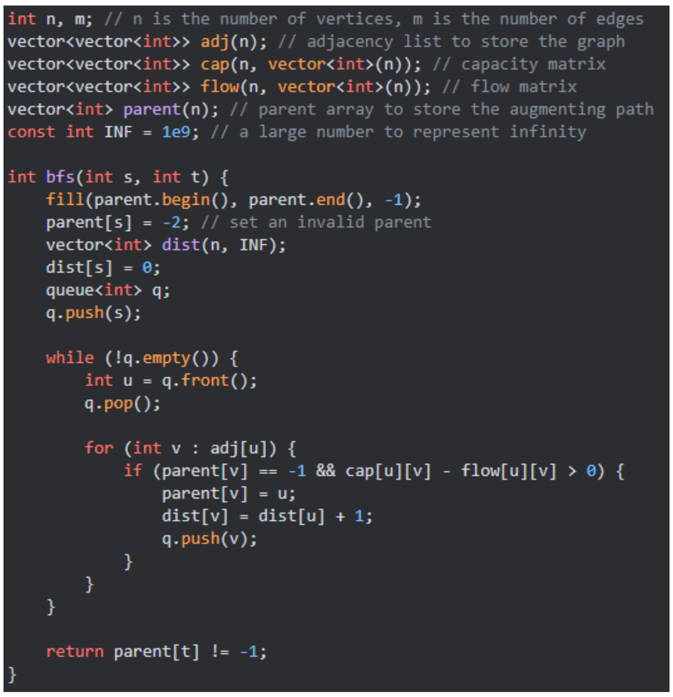
To do this, we calculate the tentative distance as the sum of the current vertex's distance and the weight of the edge connecting the current vertex to the adjacent vertex. If this tentative distance is less than the current distance of the adjacent vertex, we update its distance value and add it to the priority queue.

Finally, when all vertices have been visited, the distances vector contains the shortest path distances from the source vertex to all other vertices in the graph.

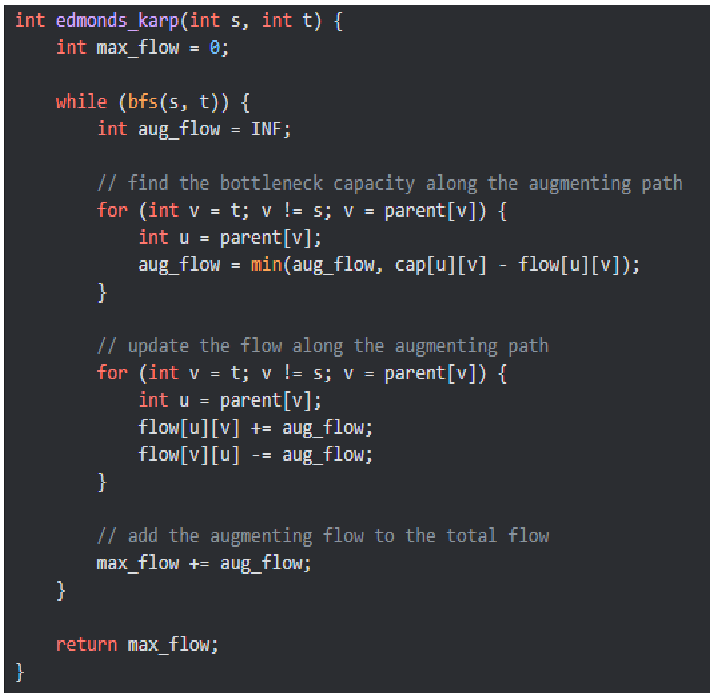
The time complexity of Dijkstra's algorithm is O((E+V)log(V)), where E is the number of edges and V is the number of vertices in the graph. This is due to the use of the priority queue data structure in the algorithm.

4.5. Max-Flow Ford-Fulkerson Algorithm – The Edmonds-Karp implementation using BFS for selection of augmentation paths. Examples and selected cases for DAG. Understanding extreme cases of time complexity for fractional and irrational edge weights. Implementation complexity. Use of Max-Flow in Maximal Bipartite Matching and reasoning about their correctness. Understanding the Min-Cut/Max-Flow Theorem and apply it to sample case to determine the impossibility of specific flow values. For the Max-Flow algorithm you will not be asked to proof the correctness of this greedy algorithm.

Here is the pseudo code implementation of the Edmonds-Karp algorithm for finding maximum flow in a network:



```
int n, m; // n is the number of vertices, m is the number of edges
vector<vector<int>> adj(n); // adjacency list to store the graph
vector<vector<int>> cap(n, vector<int>(n)); // capacity matrix
vector<vector<int>> flow(n, vector<int>(n)); // flow matrix
vector<int> parent(n); // parent array to store the augmenting path
const int INF = 1e9; // a large number to represent infinity

int bfs(int s, int t) {
    fill(parent.begin(), parent.end(), -1);
    parent[s] = -2; // set an invalid parent
    vector<int> dist(n, INF);
    dist[s] = 0;
    queue<int> q;
    q.push(s);

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v : adj[u]) {
            if (parent[v] == -1 && cap[u][v] - flow[u][v] > 0) {
                parent[v] = u;
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }

    return parent[t] != -1;
}
```



```
int edmonds_karp(int s, int t) {
    int max_flow = 0;

    while (bfs(s, t)) {
        int aug_flow = INF;

        // find the bottleneck capacity along the augmenting path
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            aug_flow = min(aug_flow, cap[u][v] - flow[u][v]);
        }

        // update the flow along the augmenting path
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            flow[u][v] += aug_flow;
            flow[v][u] -= aug_flow;
        }

        // add the augmenting flow to the total flow
        max_flow += aug_flow;
    }

    return max_flow;
}
```

In this algorithm, we first initialize a graph represented by an adjacency list, a capacity matrix to store the capacity of each edge, a flow matrix to store the flow through each edge, and a parent array to store the augmenting path. We then define a function **bfs** that takes a source node and a target node as input and returns **true** if there exists an augmenting path from the source to the target, and **false** otherwise. This function uses a Breadth-First Search algorithm to find the shortest augmenting path and stores the parent of each node in the **parent** array. We then define the **edmonds_karp** function that takes a source node and a target node as input and returns the maximum flow from the source to the target. This function repeatedly calls the **bfs** function to find augmenting paths and updates the flow along the augmenting paths until no more augmenting paths exist.

The time complexity of the Edmonds-Karp algorithm is O(VE^2), where V is the number of vertices and E is the number of edges. This is because in the worst case, we may have to update the flow along every edge in the graph for every iteration of the algorithm. However, in practice, the algorithm usually performs much better than this worst-case bound.

The Edmonds-Karp algorithm can be used to solve many types of network flow problems, including the Maximal Bipartite Matching problem. In this problem, we are given a bipartite graph and we want to find a matching that covers as many vertices as possible. A matching is a set of edges such that no two edges share a common vertex.

To use the Edmonds-Karp algorithm for this problem, we first convert the bipartite graph into a flow network by adding a source node and a sink node and connecting the source node to one partition of the bipartite graph and the other partition to the sink node. We then set the capacity of each edge to 1, indicating that each vertex can be matched with at most one other vertex. The maximum flow in this network corresponds to the maximum matching in the bipartite graph.

The correctness of the Edmonds-Karp algorithm can be proved using the Max-Flow Min-Cut theorem. This theorem states that the maximum flow in a network is equal to the minimum cut, where a cut is a partition of the vertices into two disjoint sets such that the source is in one set and the sink is in the other set. The minimum cut corresponds to the minimum number of edges that need to be removed to disconnect the source from the sink.

In the Edmonds-Karp algorithm, we always choose the shortest augmenting path from the source to the sink, which corresponds to a cut in the residual graph. Therefore, the flow along this path is equal to the capacity of the minimum cut. By repeating this process, we eventually find the maximum flow, which is equal to the capacity of the minimum cut.

The time complexity of the Edmonds-Karp algorithm can be improved to O((VE)log(U)) using the Dinic's algorithm, where U is the maximum capacity of any edge in the network. However, this improvement is not necessary for most practical applications.

5. Divide and Conquer Algorithms:

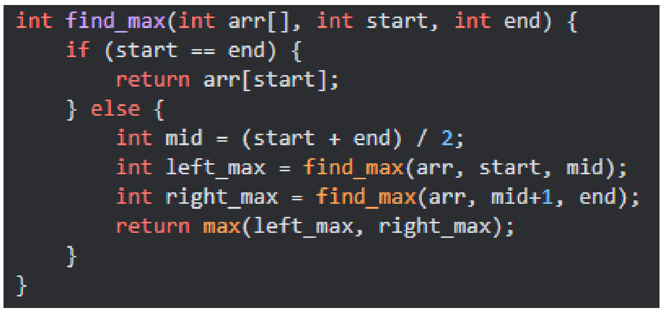
5.1. Formulation of recurrence and the Master Theorem. Deriving a recurrence from a code and “solving” it. How to improve time complexity by analysis of the terms in the Master Theorem for a specific code example, or recurrence example. For instance, you will need to know if changing a specific section of the code, will lead to a change in its asymptotic complexity, or by how much you need to make a change in which section of the recursive function implementation that will lead to changes in the time complexity of the overall algorithm.

Divide and conquer algorithms are a class of algorithms that solve a problem by breaking it down into smaller subproblems, solving those subproblems independently, and then combining the solutions to the subproblems to obtain the solution to the original problem. The key idea is to divide the problem into smaller subproblems that are easier to solve, and then combine the solutions to these subproblems to obtain the solution to the original problem.

One of the most important aspects of divide and conquer algorithms is the formulation of a recurrence relation that describes the running time of the algorithm. The recurrence relation expresses the running time of the algorithm as a function of the size of the input.

One of the most famous tools for analyzing the time complexity of divide and conquer algorithms is the Master Theorem, which provides a way to classify the running time of a class of divide and conquer algorithms based on their recurrence relation.

Here's an example of a divide and conquer algorithm to find the maximum element in an array:



```
int find_max(int arr[], int start, int end) {
    if (start == end) {
        return arr[start];
    } else {
        int mid = (start + end) / 2;
        int left_max = find_max(arr, start, mid);
        int right_max = find_max(arr, mid+1, end);
        return max(left_max, right_max);
    }
}
```

In this code, the **find_max** function takes an array **arr** and two indices **start** and **end** that specify the range of the array to search for the maximum element. If the range consists of a single element, the function returns that element. Otherwise, the function recursively calls itself on the left and right halves of the array and returns the maximum of the two values.

To analyze the time complexity of this algorithm, we can use a recurrence relation. Let **$T(n)$** be the running time of the algorithm on an array of size **n**. Then, we have:

$$T(n) = 2T(n/2) + O(1)$$

This is because the algorithm recursively calls itself twice on subproblems of size $n/2$, and the time required to compute the maximum of the two values is $O(1)$.

To solve this recurrence relation, we can use the Master Theorem. The Master Theorem states that if a recurrence relation has the form:

$$T(n) = aT(n/b) + f(n)$$

where **a** is the number of subproblems, each of size **n/b**, and **$f(n)$** is the time required to divide the problem and combine the solutions to the subproblems, then the running time of the algorithm is given by:

$$T(n) = O(n^{\log_b(a)})$$

if **$f(n) = O(n^{\log_b(a)})$**, and

$$T(n) = O(f(n))$$

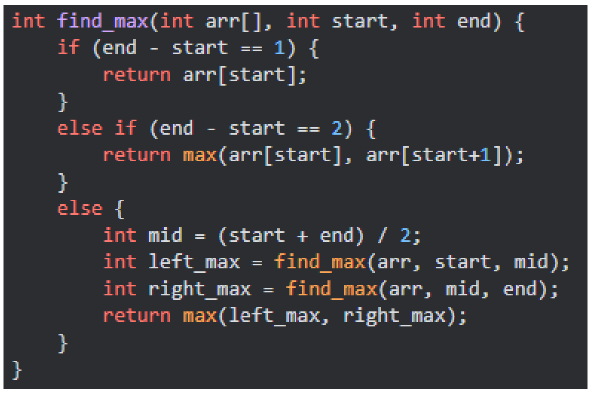
if **$f(n) = \Omega(n^{\log_b(a)})$**.

In our example, we have **$a = 2$**, **$b = 2$**, and **$f(n) = O(1)$**, so we can apply case 2 of the Master Theorem, which gives us:

$$T(n) = O(n^{\log_2(2)}) = O(n)$$

So, the running time of the **find_max** function is **O(n)**.

We can improve the time complexity of this algorithm by using the fact that the **max** function is associative and commutative. This means that we can compute the maximum of multiple elements in parallel, rather than recursively computing the maximum of two elements at a time. Here's an improved implementation of the **find_max** function that takes advantage of this property:



```
int find_max(int arr[], int start, int end) {
    if (end - start == 1) {
        return arr[start];
    }
    else if (end - start == 2) {
        return max(arr[start], arr[start+1]);
    }
    else {
        int mid = (start + end) / 2;
        int left_max = find_max(arr, start, mid);
        int right_max = find_max(arr, mid, end);
        return max(left_max, right_max);
    }
}
```

In this improved implementation, we first check if there are only two elements, and return the maximum of those two elements. Otherwise, we divide the array into two halves, and recursively compute the maximum of the left half and the maximum of the right half. Finally, we return the maximum of these two maximums.

The running time of this improved algorithm can be analyzed using the Master Theorem. In this case, we have a = 2 (since we are dividing the array into two halves), b = 2 (since each subproblem has half the size of the original problem), and d = 1 (since the time complexity of computing the maximum of two elements is O(1)). Therefore, we have:

$$T(n) = 2T(n/2) + O(1)$$

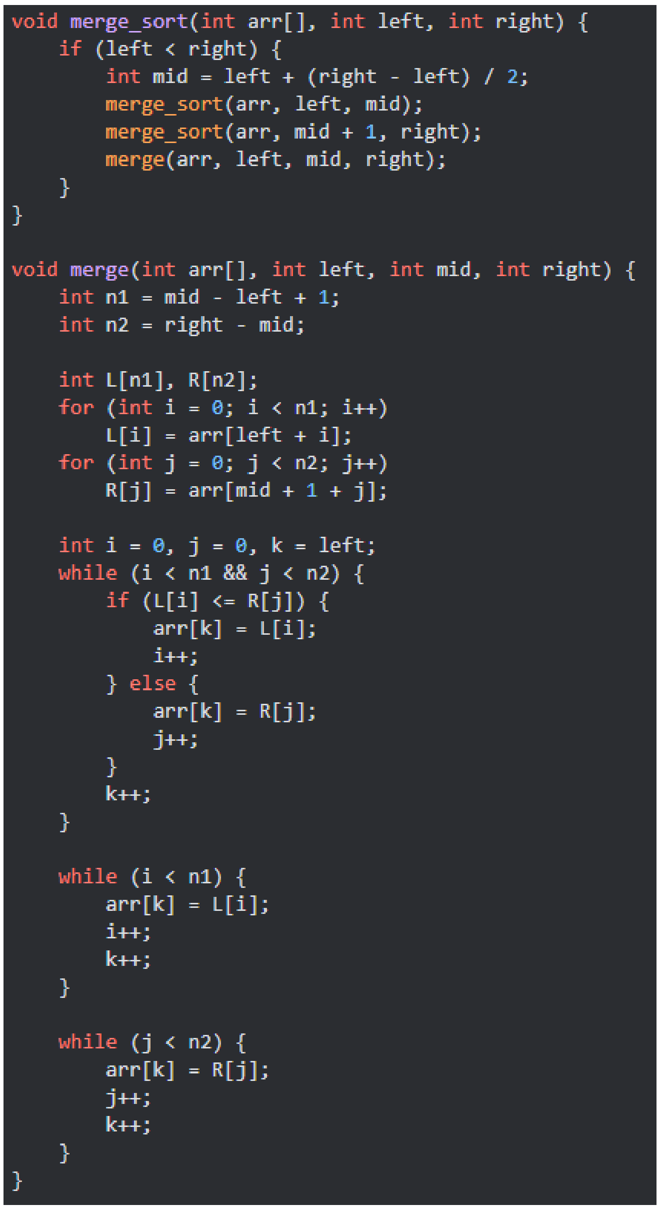
According to the Master Theorem, this recurrence has a running time of O(nlog(n)), which is much faster than the previous implementation.

By analyzing the terms in the Master Theorem for a specific code example or recurrence, we can identify which parts of the algorithm contribute most to the running time, and how we can modify the algorithm to improve its time complexity. Changing a specific section of the code can lead to changes in the asymptotic complexity, and we need to carefully analyze the impact of these changes to ensure that the algorithm remains correct and efficient.

5.2. Examples of the recurrences of the Merge-Sort and Hanoi Towers algorithms and their recurrence solutions.

5.2.1. Merge-Sort Algorithm:

The Merge-Sort algorithm is a popular sorting algorithm that uses the divide and conquer approach to sort an array. The algorithm consists of two main steps - dividing the array into two halves and recursively sorting them, and merging the sorted halves to produce the final sorted array. Here's the pseudocode for the Merge-Sort algorithm:



```
void merge_sort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        merge_sort(arr, left, mid);
        merge_sort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

In the merge_sort function, we first check if the left index is less than the right index. If it is, we calculate the middle index and recursively call the merge_sort function on the left and right halves of the array. Once both halves are sorted, we merge them using the merge function.

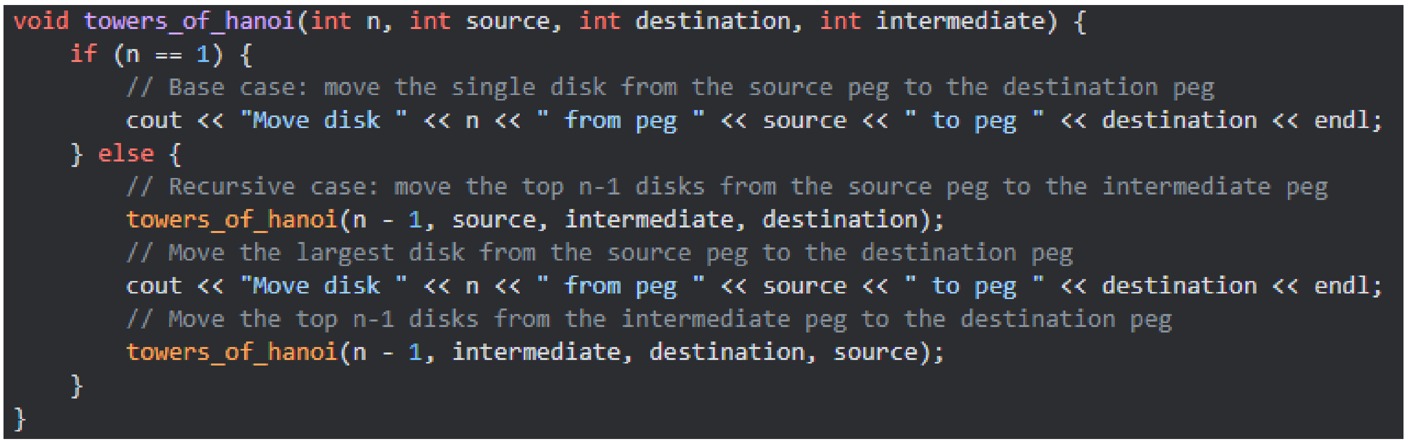
The recurrence relation for the Merge-Sort algorithm can be derived as follows:

$$T(n) = 2T(n/2) + O(n)$$

where T(n) represents the time taken by the algorithm to sort an array of size n. The first term on the right-hand side of the equation represents the time taken to sort the two halves of the array recursively, and the second term represents the time taken to merge the two sorted halves. Using the Master Theorem, we can determine the time complexity of the Merge-Sort algorithm to be O(nlog(n)).

5.2.2.Towers of Hanoi Algorithm:

The Towers of Hanoi problem is a classic problem that involves moving a tower of disks of different sizes from one peg to another, with the constraint that only one disk can be moved at a time and a larger disk cannot be placed on top of a smaller disk. The problem can be solved using the recursive approach, where we recursively move the smaller tower to an intermediate peg, move the largest disk to the destination peg, and then move the smaller tower from the intermediate peg to the destination peg. Here's the pseudocode for the Towers of Hanoi algorithm:



```
void towers_of_hanoi(int n, int source, int destination, int intermediate) {
    if (n == 1) {
        // Base case: move the single disk from the source peg to the destination peg
        cout << "Move disk " << n << " from peg " << source << " to peg " << destination << endl;
    } else {
        // Recursive case: move the top n-1 disks from the source peg to the intermediate peg
        towers_of_hanoi(n - 1, source, intermediate, destination);
        // Move the largest disk from the source peg to the destination peg
        cout << "Move disk " << n << " from peg " << source << " to peg " << destination << endl;
        // Move the top n-1 disks from the intermediate peg to the destination peg
        towers_of_hanoi(n - 1, intermediate, destination, source);
    }
}
```

The input arguments to the function are **n**, the number of disks in the tower, **source**, the peg the tower is currently on, **destination**, the peg we want to move the tower to, and **intermediate**, the third peg that we can use to move the tower.

The base case of the recursion is when we have only one disk left, in which case we simply move it from the source peg to the destination peg. In the recursive case, we first move the top **n-1** disks from the source peg to the intermediate peg using the destination peg as the temporary peg. Then, we move the largest disk from the source peg to the destination peg. Finally, we move the top **n-1** disks from the intermediate peg to the destination peg using the source peg as the temporary peg.

The time complexity of the Towers of Hanoi algorithm is O(2^n), since in the worst case we need to make 2^n - 1 moves to move a tower of n disks.