EE599 HW#6

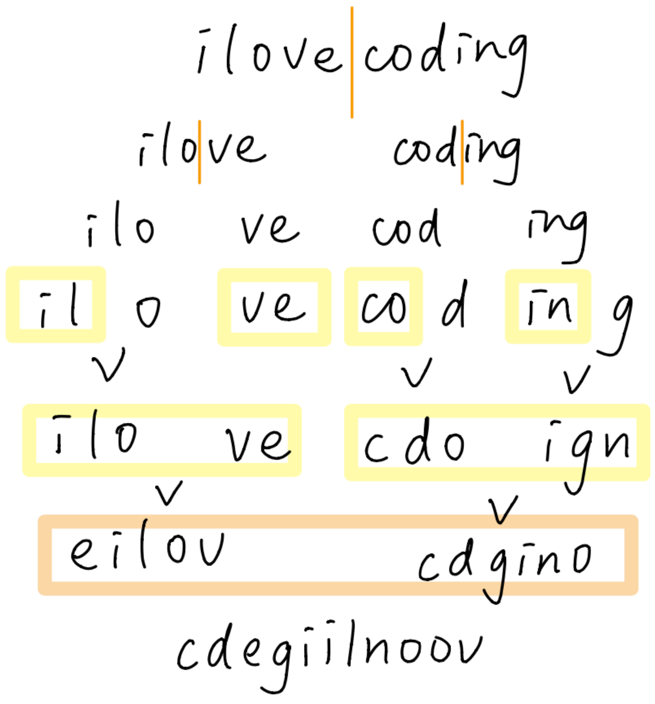
Runtime Analysis

Student Name: Shi Lin Chen
Student ID: 2991911997
Student Email: shilinch@usc.edu

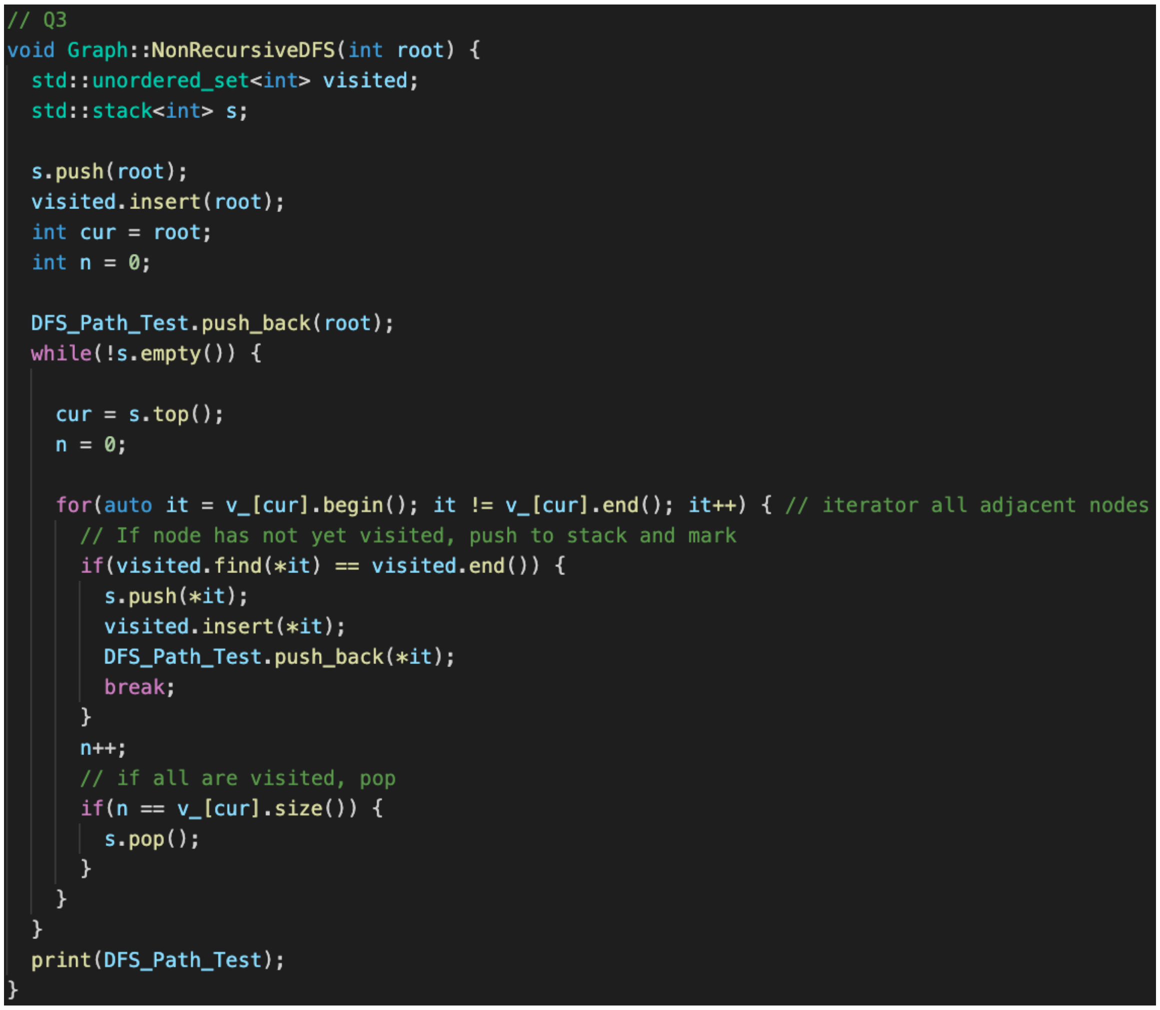
Q1

- A tree is an **[✓ undirected]/directed** graph.
- A tree is a **[✓ connected] / unconnected** graph.
- A tree is a **cyclic / [✓ acyclic]** graph.
- In a tree, there **[✓ is] / is not** a path from each vertex to all other vertices.
- A simple graph is a graph that without repeated vertices

Q2



Q3



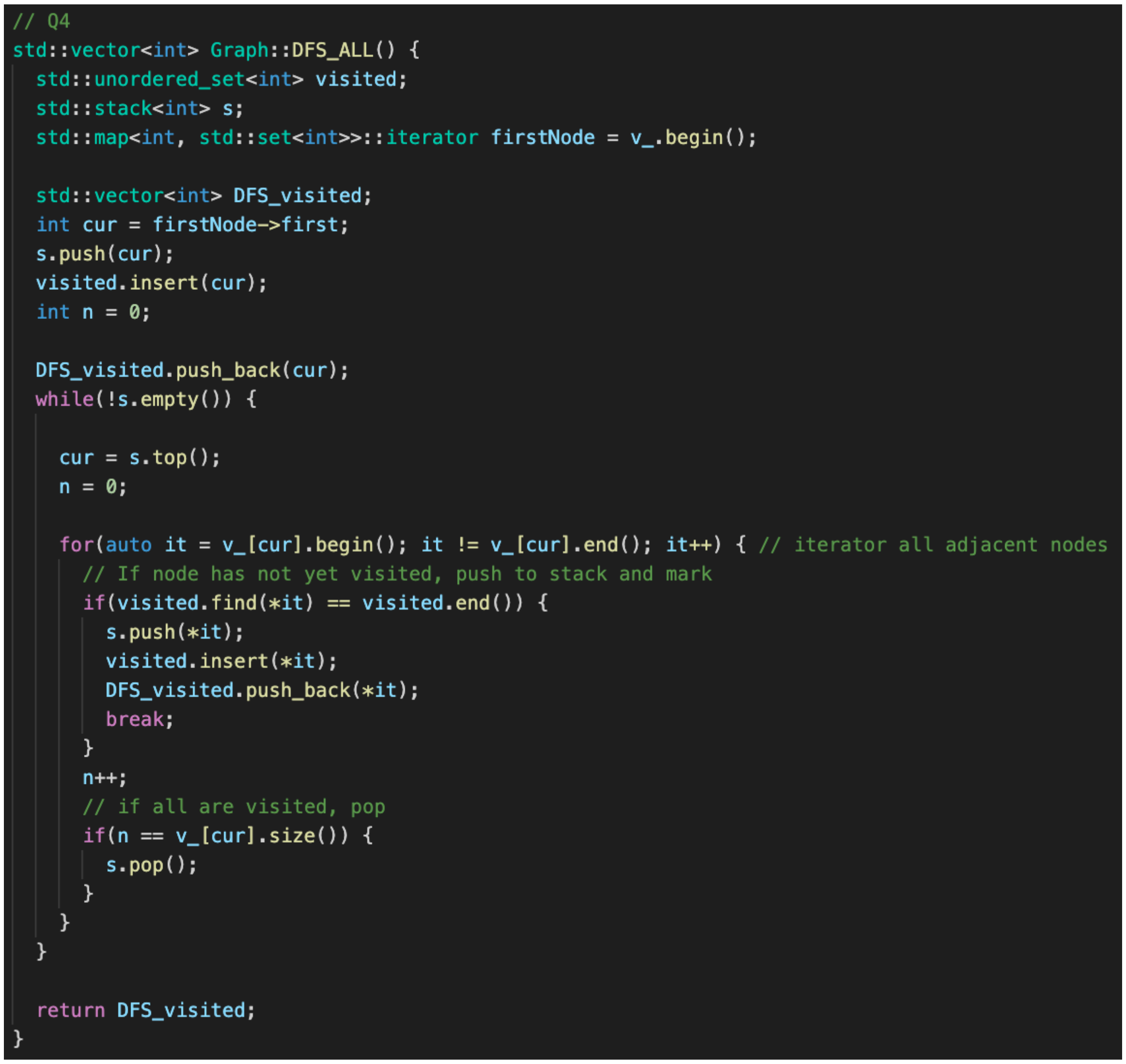
```
// Q3
void Graph::NonRecursiveDFS(int root) {
  std::unordered_set<int> visited;
  std::stack<int> s;

  s.push(root);
  visited.insert(root);
  int cur = root;
  int n = 0;

  DFS_Path_Test.push_back(root);
  while(!s.empty()) {

    cur = s.top();
    n = 0;

    for(auto it = v_[cur].begin(); it != v_[cur].end(); it++) { // iterator all adjacent nodes
      // If node has not yet visited, push to stack and mark
      if(visited.find(*it) == visited.end()) {
        s.push(*it);
        visited.insert(*it);
        DFS_Path_Test.push_back(*it);
        break;
      }
      n++;
      // if all are visited, pop
      if(n == v_[cur].size()) {
        s.pop();
      }
    }
  }
  print(DFS_Path_Test);
}
```

DFS needed O(V+E), V = Vertices and E = Edges

In my algorithm, I use stack to store the parent of nodes, if the node doesn't have children, then pop the element and let the current temp node equal top and mark the visited nodes in an unorder set. While the std::find in std::set need O(1), n is the # of elements in set, worst case might be O(n), we could see this finding method as the number of edges when checking if visited or not.

Therefore, T(n) = O(V+E) = O(n)

Q4

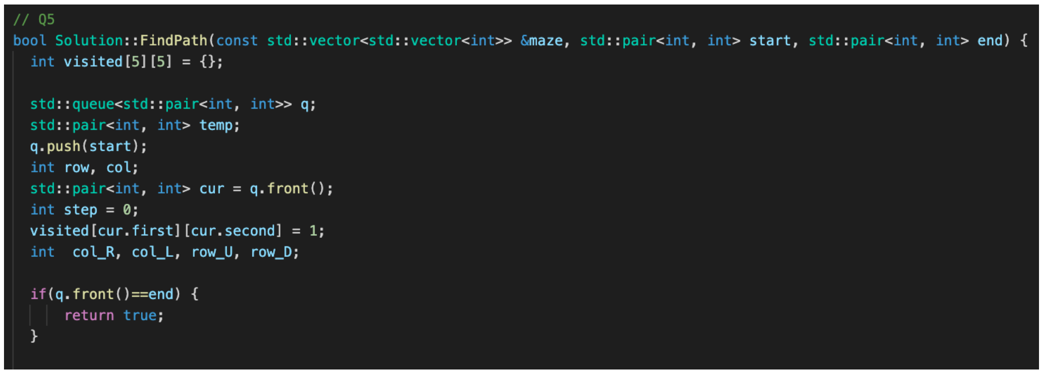


```cpp
// Q4
std::vector<int> Graph::DFS_ALL() {
  std::unordered_set<int> visited;
  std::stack<int> s;
  std::map<int, std::set<int>>::iterator firstNode = v_.begin();

  std::vector<int> DFS_visited;
  int cur = firstNode->first;
  s.push(cur);
  visited.insert(cur);
  int n = 0;

  DFS_visited.push_back(cur);
  while(!s.empty()) {

    cur = s.top();
    n = 0;

    for(auto it = v_[cur].begin(); it != v_[cur].end(); it++) { // iterator all adjacent nodes
      // If node has not yet visited, push to stack and mark
      if(visited.find(*it) == visited.end()) {
        s.push(*it);
        visited.insert(*it);
        DFS_visited.push_back(*it);
        break;
      }
      n++;
      // if all are visited, pop
      if(n == v_[cur].size()) {
        s.pop();
      }
    }
  }

  return DFS_visited;
}
```

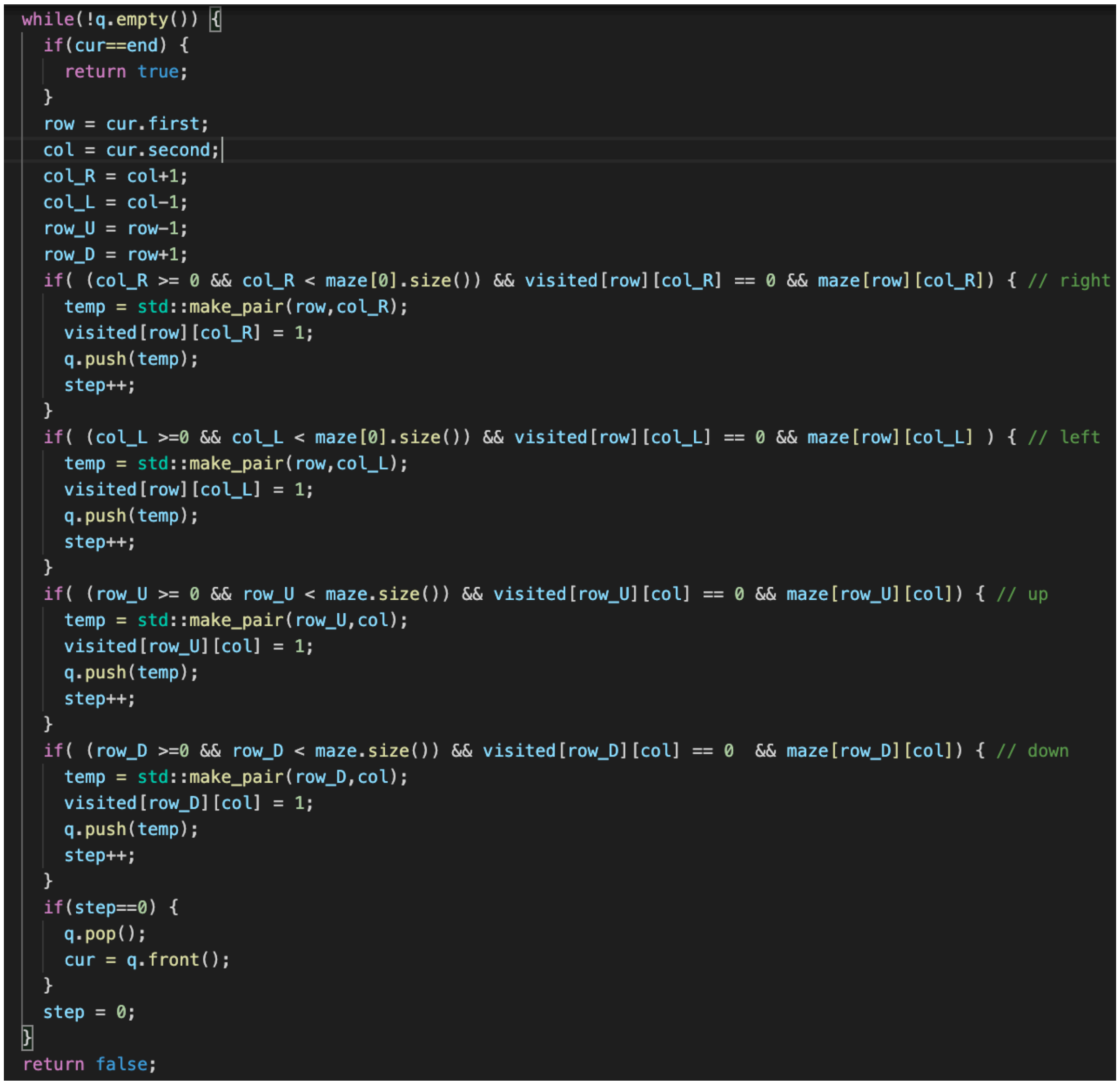
Q4 is almost the same function as Q3, instead of the return type.

Therefore, T(n) = O(V+E) = O(n)

Q5



```cpp
// Q5
bool Solution::FindPath(const std::vector<std::vector<int>> &maze, std::pair<int, int> start, std::pair<int, int> end) {
  int visited[5][5] = {};

  std::queue<std::pair<int, int>> q;
  std::pair<int, int> temp;
  q.push(start);
  int row, col;
  std::pair<int, int> cur = q.front();
  int step = 0;
  visited[cur.first][cur.second] = 1;
  int  col_R, col_L, row_U, row_D;

  if(q.front()==end) {
      return true;
  }
```

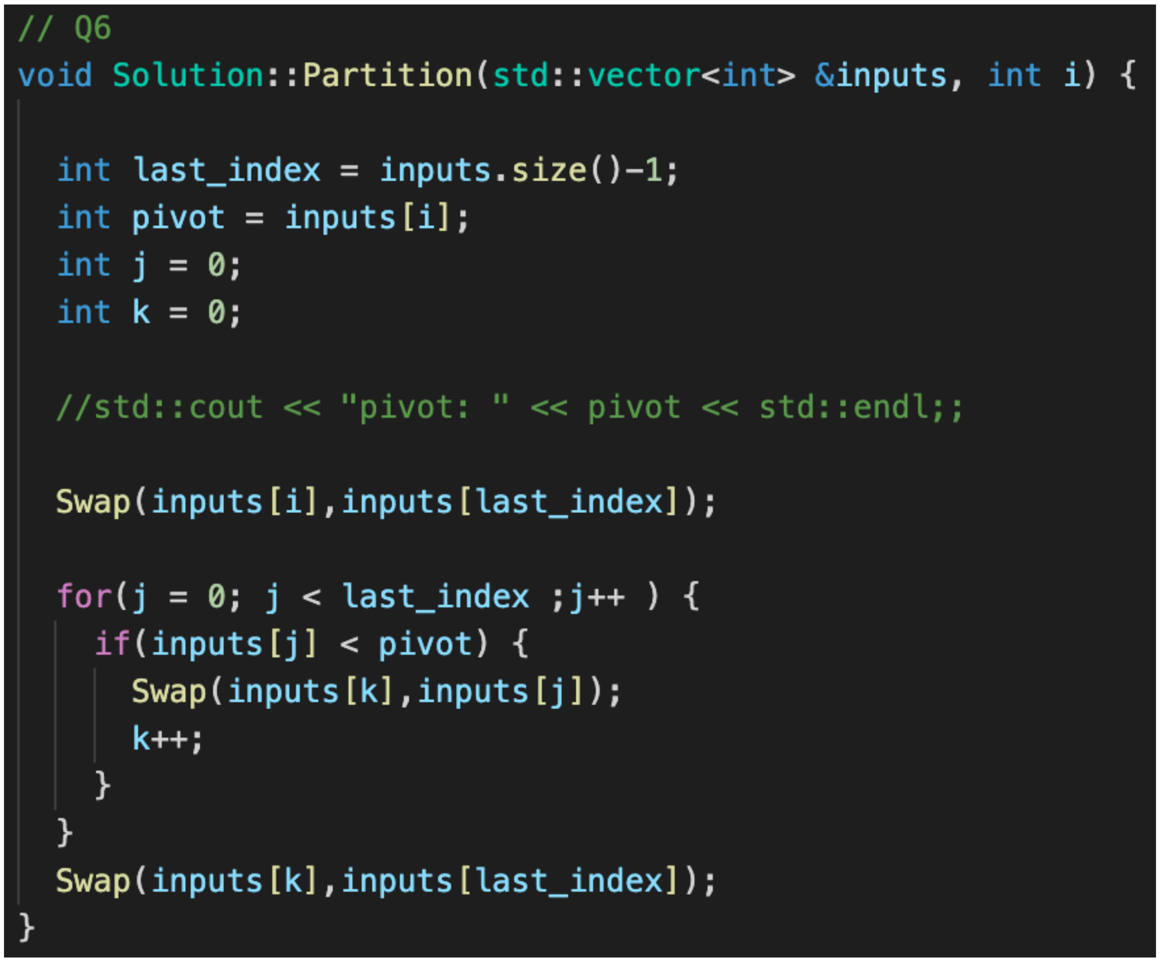


```
while(!q.empty()) {
  if(cur==end) {
    return true;
  }
  row = cur.first;
  col = cur.second;
  col_R = col+1;
  col_L = col-1;
  row_U = row-1;
  row_D = row+1;
  if( (col_R >= 0 && col_R < maze[0].size()) && visited[row][col_R] == 0 && maze[row][col_R]) { // right
    temp = std::make_pair(row,col_R);
    visited[row][col_R] = 1;
    q.push(temp);
    step++;
  }
  if( (col_L >=0 && col_L < maze[0].size()) && visited[row][col_L] == 0 && maze[row][col_L] ) { // left
    temp = std::make_pair(row,col_L);
    visited[row][col_L] = 1;
    q.push(temp);
    step++;
  }
  if( (row_U >= 0 && row_U < maze.size()) && visited[row_U][col] == 0 && maze[row_U][col]) { // up
    temp = std::make_pair(row_U,col);
    visited[row_U][col] = 1;
    q.push(temp);
    step++;
  }
  if( (row_D >=0 && row_D < maze.size()) && visited[row_D][col] == 0  && maze[row_D][col]) { // down
    temp = std::make_pair(row_D,col);
    visited[row_D][col] = 1;
    q.push(temp);
    step++;
  }
  if(step==0) {
    q.pop();
    cur = q.front();
  }
  step = 0;
}
return false;
```

The runtime of finding a path in a maze is the # of available cells. (n)

I use BFS to traverse the maze until all maze has been visited, if countering end, return true, if not, return false. The worst case will be O(n) since, we traverse all the cells in maze.

T(n) = O(n)



```
// Q6
void Solution::Partition(std::vector<int> &inputs, int i) {

  int last_index = inputs.size()-1;
  int pivot = inputs[i];
  int j = 0;
  int k = 0;

  //std::cout << "pivot: " << pivot << std::endl;;

  Swap(inputs[i],inputs[last_index]);

  for(j = 0; j < last_index ;j++ ) {
    if(inputs[j] < pivot) {
      Swap(inputs[k],inputs[j]);
      k++;
    }
  }
  Swap(inputs[k],inputs[last_index]);
}
```

Partition needed to iterate all the elements in input vector. Therefore, it takes linear time. T(n) = O(n)