1 Multiple Choice

ANSWER SHEET FOR MULTIPLE CHOICE:
Please fill the correct answers in the **answer sheet**. Answers outside the answer sheet will **NOT** be graded.

1	2	3	4	5	6
C	A	A	A	B	A
7	8	9	10	11	
D	A	ABE	ADE	BCE	

2 Writing Questions

2.1 Warm Up (15 points)

i) What is the average time complexity of Merge Sort and Insertion Sort? (3 points) Whether are they stable? (2 points)

ii) In principle, the time complexity of Selection Algorithms should be ____ ($\leq$ or $\geq$) the time complexity of Sorting Algorithms. (2 points) Why? (3 points)

iii) Draw the Binary Search Tree based on the insertion sequence "8, 3, 1, 6, 10, 4, 7". (3 points) If a proper binary tree has n 2-degree nodes, what is the number of its leaf nodes? (2 points)

Solution:

i) Merge Sort: $O(n \log n)$, Insertion Sort: $O(n^2)$. Both of them are stable.

ii) $\leq$. Because selection is easier than sorting. If an array is sorted, selection can be done in $O(1)$.

iii) 8
3 10
1 6 null null
null null 4 7
The number should be $n + 1$

2.2 Fix K-D Tree (14 points)

You must be very familiar k-d trees as you have passed the trial of Project 3. The pseudocode below is directly from that project, illustrating the deletion operation.

```
Function Delete(node, key, depth)
    if node is null then
        return null;
    end
    dimension ← depth mod k;
    if key = node.key then
        if node is a leaf then
            delete node directly;
            return null;
        else if node has right subtree then
            minNode ← the minimum node on dimension in the right subtree node.right;
            node.key ← minNode.key;
            node.value ← minNode.value;
            node.right ← Delete(node.right, key, depth + 1);
        else if node has left subtree then
            maxNode ← the maximum node on dimension in the left subtree node.left;
            node.key ← maxNode.key;
            node.value ← maxNode.value;
            node.left ← Delete(node.left, key, depth + 1);
        end
    else
        if key < node.key on dimension then
            node.left ← Delete(node.left, key, depth + 1);
        else
            node.right ← Delete(node.right, key, depth + 1);
        end
    end
    return node;
end
```

Algorithm 1: Deletion of node.

The following code, extracted from slides, illustrates the insertion operation of a k-d tree.

```cpp
void insert(node *&root, Item item, int dim) {
    if(root == NULL) {
        root = new node(item);
        return;
    }
    if(item.key == root->item.key) // equal in all
        return;                    // dimensions
    if(item.key[dim] < root->item.key[dim])
        insert(root->left, item, (dim+1)%numDim);
    else
        insert(root->right, item, (dim+1)%numDim);
}
```

For the simplicity, we omit the values of nodes, and only consider the keys.

i) Suppose you already have a 2-d tree shown in Fig. 1. Now please insert (0,1) and (0,5) into the tree, and draw the tree after the insertion operation. Notice that you should strictly follow the code provided. (2 points)

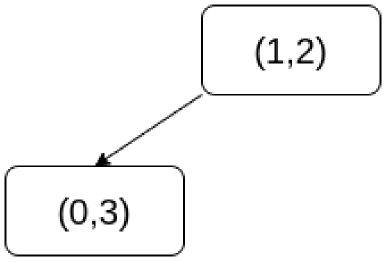


Figure 1: Initial K-D Tree

ii) Based on the results from part i), delete the root node (1,2) and draw the tree after the deletion operation. Notice that you should strictly follow the pseudocode provided.(2 points).

Hint: In the deletion operation, we need to find the **maximum** or **minimum** node on *dimension*. In cases where two nodes share identical keys within that *dimension*, a secondary comparison is conducted along the other dimension. To illustrate, consider the scenario where (1,4) represents the **maximum** node on the **first** dimension when compared to other nodes such as (1,2) and (1,0). Similarly, (1,0) stands as the **minimum** node on the **second** dimension when compared among nodes like (5,0), (1,0), and (9,0).

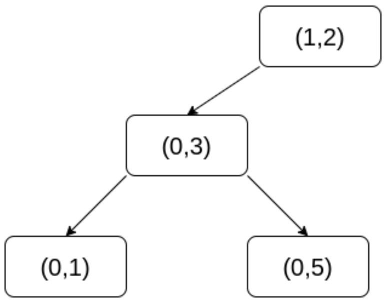
iii) Based on the results from part ii), insert (0,1) again and draw the tree after the insertion operation. Notice that you should strictly follow the code provided. (2 points).

iv) The tree seems a little bit strange now. Observe the tree, you will find one invariant of the k-d tree is violated. What is the violated invariant specifically? You should think about the relationship between one node and the nodes in its **subtree**. (2 points)

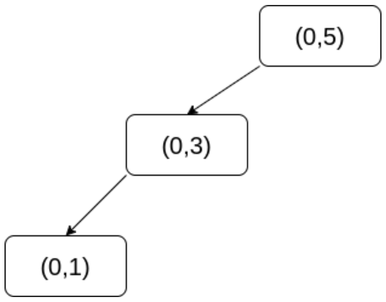
v) ____ (insertion or deletion) operation ruins the invariant. (2 points).

vi) Explain how does the operation you chose in part iv) violate the invariant. (1 points).

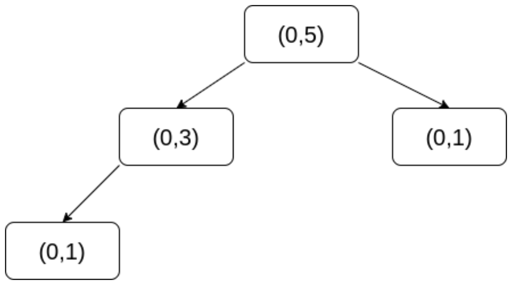
vii) How will you modify the operation to maintain the invariant? Notice that you will not get full points if your method has a bad time complexity. (3 points)



i)



ii)



iii)

iv) in any given node's left subtree, all nodes have keys smaller than the root node along a specified dimension.

v) Deletion

vi) The deletion operation involves replacing the root with a node from the left subtree. If multiple nodes in the left subtree share the same key as the selected node along the specified dimension, it results in a violation of the invariant that dictates all nodes in the left subtree must have keys smaller than the root node along that dimension, as some nodes are equal rather than strictly smaller.

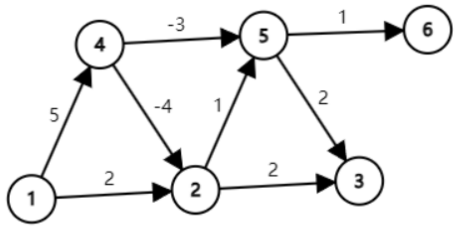
vii) remove all the nodes in that subtree and reinsert. (1 points)

Choose the minimum node on the dimension in the left subtree to replace the root, then move the left subtree to right subtree. (3 points)

2.3 Graph! Graph! Graph!(15 points)

2.3.1 Dijkstra(5 points)

You are given a directed graph G shown below, 1 is the source node and you want to calculate the shortest distance from 1 to all other nodes x, which is denoted as $d(x)$. Although you know it's wrong to use Dijkstra's algorithm here, but you still use it. Write the actual $d(x)$ and the $d(x)$ you get by Dijkstra's algorithm for each node x.



Solution:
Actual:

$d(1)$	$d(2)$	$d(3)$	$d(4)$	$d(5)$	$d(6)$
0	1	3	5	2	3

Dijkstra:

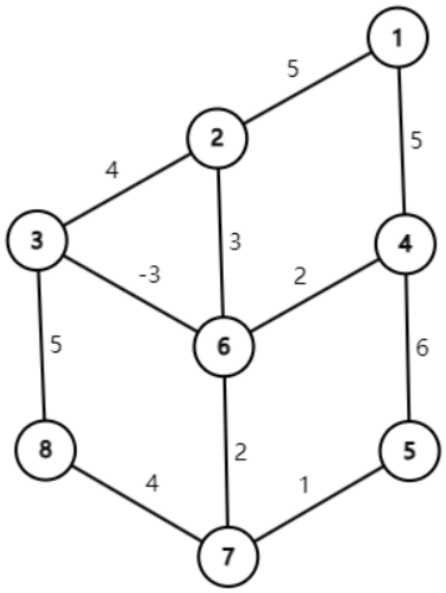
$d(1)$	$d(2)$	$d(3)$	$d(4)$	$d(5)$	$d(6)$
0	2	4	5	3	4

or

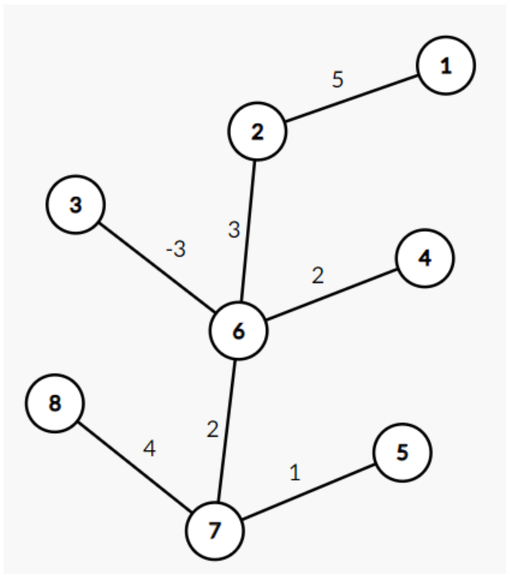
$d(1)$	$d(2)$	$d(3)$	$d(4)$	$d(5)$	$d(6)$
0	1	4	5	2	4

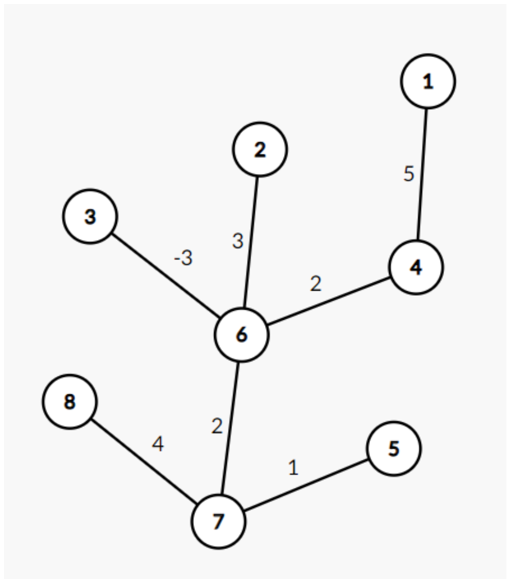
2.3.2 Prim(5 points)

You are given a undirected graph G shown below, apply Prim's algorithm to find the minimum spanning tree of the graph G. Illustrate step by step to get partial points.



Two possible MST:





2.3.3 Save Blue Tiger!(5 points)

After an expedition on an island, Mr. Blue Tiger finds him lost. He desires to know the shortest distance to another island, can you help him? Specifically, you are given an a matrix indicating a 2-D map, where 1 represents land and 0 represents water. An island is a 4-directionally connected group of 1's not connected to any other 1's. There are exactly two islands in grid.
Please complete the following code which uses BFS to find the shortest distance.

```cpp
int shortestBridge(vector<vector<int>>& grid) {
    int Dirs[4][2] = { { 0 , -1 } , { 0 , 1 } , { 1 , 0 } , { -1 , 0 } };
    int m = grid.size() , n = grid[0].size();
    queue<pair<int,int>> Q;
    vector<pair<int,int>> island1;

    for(int i = 0;i < m;++i)
        for(int j = 0;j < n;++j) if( grid[i][j] == 1 ){
            grid[i][j] = -1;
            Q.emplace(i,j); // same to Q.push_back(make_pair(i,j))
            while(!Q.empty()) {
                auto [x,y] = Q.front();
                Q.pop();
                ___1___ // TODO
                for(int k = 0;k < 4;++k) {
                    int nx = x + Dirs[k][0];
                    int ny = y + Dirs[k][1];
                    if( nx >= 0 && ny >= 0 && nx < m && ny < n
                        && grid[nx][ny] == 1 ) {
                        Q.emplace(nx,ny);
                        grid[nx][ny] = -1;
                    }
                }
            }
            // Find all lands for this island, then BFS to find another island
            ___2___ //TODO
            int distance = 0;
            while(!Q.empty()) {
                int size = Q.size();
                for(int i = 0; i < size; ++i) {
                    auto [x,y] = Q.front();
                    Q.pop();
                    for(int k = 0;k < 4;++k) {
                        int nx = x + Dirs[k][0];
                        int ny = y + Dirs[k][1];
                        if( nx >= 0 && ny >= 0 && nx < m && ny < n ){
                            if( grid[nx][ny] == 0 ) {
                                Q.emplace(nx,ny);
                                ___3___ // TODO
                            }
                            else if( grid[nx][ny] == 1 )
                                ___4___ // TODO
                        }
                    }
                }
                ___5___ //TODO
            }
        }
    return 0;
}
```

Solution:
①: island1.push_back(make_pair(x,y));

②: Q.push(make_pair(x,y));

③: grid[nx][ny]=-1;

④: return distance;

⑤: distance++;

2.4 Coin Change (15 points)

You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount.* If that amount of money cannot be made up by any combination of the coins, return `-1`.

You may assume that you have an infinite number of each kind of coin.

Example 1:

```
Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
```

Example 2:

```
Input: coins = [2], amount = 3
Output: -1
```

Example 3:

```
Input: coins = [1], amount = 0
Output: 0
```

A simple solution is to consider every possible subset of coins and calculate the total number of coins needed for each subset. Finally, return the minimum of all the values.

```cpp
int coinChange(vector<int>& coins, int amount) {
    // base case
    if (amount == 0) return 0;
    // initialize the result
    int res = INT_MAX;
    // consider every possible subset of coins
    for (int i = 0; i < coins.size(); i++) {
        // if the coin value is less than or equal to the amount
        if (coins[i] <= amount) {
            // calculate the result for the subset
            int sub_res = coinChange(coins, amount - coins[i]);
            // if the result is not INT_MAX and the result + 1
            // is less than the current result
            if (sub_res != INT_MAX && sub_res + 1 < res) {
                // update the result
                res = sub_res + 1;
            }
        }
    }
    // return the result
    return res;
}
```

The time complexity of this solution is $O(S^n)$.

This solution is not efficient enough. We can optimize this solution using ___**1**___.
Assume $F(S)$ is the minimum number of coins needed to make change for amount S. And C is the last coin in the optimal solution. Then we have the following relation:

$$F(S) = \_\_\_\mathbf{2}\_\_\_$$

However, we don't know what C is. So we need to try every possible coin and choose the one that leads to the minimum number of coins. In above recursive solution, many subproblems are solved again and again. We can avoid this by storing the results of subproblems.

Please complete the following code to implement the solution.

```
class Solution {
    vector<int>count;
    int dp(vector<int>& coins, int rem) {
        if (rem < 0) return -1;
        if (rem == 0) return 0;
        if (count[rem - 1] != 0) return ___3___; // TODO
        int Min = INT_MAX;
        for (int coin:coins) {
            int res = ___4___; // TODO
            if (res >= 0 && res < Min) {
                Min = res + 1;
            }
        }
        count[rem - 1] = ( ( ___5___ ) ? -1 : Min ); // TODO
        return count[rem - 1];
    }
public:
    int coinChange(vector<int>& coins, int amount) {
        if (amount < 1) return 0;
        count.resize(amount);
        return dp(coins, amount);
    }

};
```

The time complexity of this solution is ___**6**___, where S is the amount, n is count size.
There is also another solution.

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        int Max = amount + 1;
        vector<int> dp(amount + 1, Max);
        dp[0] = 0;
        for (int i = 1; i <= amount; ++i) {
            for (int j = 0; j < (int)coins.size(); ++j) {
                if (coins[j] <= i) {
                    dp[i] = min(dp[i], ___7___); // TODO
                }
            }
        }
        return dp[amount] > amount ? -1 : dp[amount];
    }
};
```

This solution is a bottom-up approach. The time complexity of this solution is ___**8**___, where S is the amount, n is count size.

Solution:

①: Dynamic Programming

②: $F(S-C)+1$

③: count[rem - 1]

④: dp(coins, rem - coin)

⑤: Min == INT_MAX

⑥: $O(Sn)$

⑦: dp[i - coins[j]] + 1

⑧: $O(Sn)$