1. **Sorting**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

2. **BFS/DFS**

**BFS:** All possible full credit solutions:
- A, C, D, B, E, H, G, J, F
- A, C, D, B, E, H, J, G, F
- A, C, D, E, B, H, G, J, F
- A, C, D, E, B, H, J, G, F
- A, D, C, B, H, E, G, J, F
- A, D, C, B, H, E, J, G, F
- A, D, C, H, B, E, G, J, F
- A, D, C, H, B, E, J, G, F

**DFS:** All possible full credit solutions:
- A, C, B, E, D, H, J, G, F
- A, C, B, E, D, H, G, F, J
3. Heap

Circle: YES

4. MST

(order in list does not matter)

MST #1: AC, BC, BE, CE, BD, DE, DF
MST #2: AC, BC, BE, CE, BD, DE, DF
MST #3: AC, BC, BE, CE, BD, DE, DF
MST #4: AC, BC, BE, CE, BD, DE, DF
MST #5: AC, BC, BE, CE, BD, DE, DF
5. BST

Diagram after inserting (25,2):

Diagram after inserting (5,5):

This one is completed for you as a node formatting example.

Diagram after inserting (19,3):

Diagram after inserting (5,12):

2/2 pts if they update value of 5 regardless of 5's location in the tree

Diagram after inserting (40,5):

Diagram after inserting (3,1):

6. ADTs

void moveLeft(Grid<int> &board) {
    // For each [row][col], we consider if something from the right
    // should move into this place, and there are two cases of this:
    // (1) if we are non-zero, see if a matching number merges into us
    // (2) if we are blank, see if a number moves into this space
    for (int row = 0; row < board.numRows(); row++) {
        for (int col = 0; col < board.numCols(); col++) {
            // (1) if we are non-zero, see if a matching number merges
            if (board[row][col] != 0) {
                for (int i = col + 1; i < board.numCols(); i++) {
                    // matching number: merge
                    if (board[row][i] == board[row][col]) {
                        board[row][col] *= 2;
                        board[row][i] = 0;
                        break;
                    }
                    // non-matching number: end search
                }
            }
        }
    }
}
else if (board[row][i] != 0) {
    break;
}
}
}
// (2) if we are blank, see if a number moves into this space
else {
    for (int i = col + 1; i < board.numCols(); i++){
        if (board[row][i] != 0){
            board[row][col] = board[row][i];
            board[row][i] = 0;
            col--;
            break;
        }
    }
}

7. Graphs

// This solution uses a DFS helper that is shared between parts (a) and (b). It
// finds all itineraries and then parts (a) and (b) examine the output to answer
// their specific questions about them (number of different ones, and longest one).
// It is also possible to write custom DFS for each part, with no shared code. That
// gives two simpler functions, but less reuse.

// Shared DFS is in two functions: start is a wrapper and go is the actual recursive
// function.
void startDFS(BasicGraph &map, Vertex *dorm, Vector<Vector<Vertex*> > &itineraries) {
    Vector<Vertex*> currentItinerary;
    currentItinerary.add(dorm);
    for (Vertex *neighbor : map.getNeighbors(dorm)) {
        if (!neighbor->visited) {
            neighbor->visited = true;
            goDFS(map, neighbor, dorm, currentItinerary, itineraries);
            neighbor->visited = false;
        }
    }
}

void goDFS(BasicGraph &map, Vertex *current, Vertex *dorm, Vector<Vertex*> currentItinerary, Vector<Vector<Vertex*> > &itineraries) {
    currentItinerary.add(current);
    if (current == dorm) {
        itineraries.add(currentItinerary);
        return;
    }
    for (Vertex *neighbor : map.getNeighbors(current)) {
if (!neighbor->visited) {
    neighbor->visited = true;
    goDFS(map, neighbor, dorm, currentItinerary, itineraries);
    neighbor->visited = false;
}
}

// this is the required function for (a), relies on two DFS helpers above
int countItineraries(BasicGraph &map, Vertex *dorm) {
    if (dorm == NULL) error("Dorm is null!";
    Vector<Vector<Vertex*>> itineraries;
    startDFS(map, dorm, itineraries);
    return itineraries.size();
}

// this is the required function for (b), relies on two DFS helpers above
int longestItinerary(BasicGraph &map, Vertex *dorm) {
    if (dorm == NULL) error("Dorm is null!";
    Vector<Vector<Vertex*>> itineraries;
    startDFS(map, dorm, itineraries);
    int longest = -1;
    for (Vector<Vertex*> itinerary : itineraries) {
        if (itinerary.size() > longest) {
            longest = itinerary.size();
        }
    }
    return longest;
}

8. Trees

bool isSubtree(Node *tree1, Node *tree2) {
    if (isSame(tree1, tree2)) return true;
    if (tree1 == NULL) return false;
    if (isSubtree(tree1, tree2->left)) return true;
    if (isSubtree(tree1, tree2->right)) return true;
    return false;
}

bool isSame(Node *tree1, Node *tree2) {
    if (tree1 == NULL && tree2 == NULL) return true;
    if (tree1 == NULL || tree2 == NULL) return false;
    if (tree1->value != tree2->value) return false;
    return isSame(tree1->left, tree2->left) && isSame(tree1->right, tree2->right);