FINAL EXAM A

SECOND SEMESTER OF ACADEMIC YEAR 2019 – 2020

SOLUTION & SCORING CRITERION

1. Linked Lists (15pts).

// Sample solution 1

```
ListNode *curr = front->next;              // 3 pts
front->next = front->next->next;           // 2 pts
delete curr;                               // 2 pts
curr = front->next->next->next;            // 2 pts
front->next->next->next = nullptr;         // 2 pts
delete curr->next;                         // 2 pts
delete curr;                               // 2 pts
```

// Sample solution 2

```
delete front->next->next->next->next->next;   // 3 pts
delete front->next->next->next->next;         // 3 pts
ListNode *temp = front->next;                 // 2 pts
front->next = front->next->next;              // 2 pts
delete temp;                                  // 2 pts
front->next->next->next = nullptr;            // 2 pts
```

2. Graphs (20pts).

2.1 (16pts). (One path one point)

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
A, C, E, B, D, H, J, G, F
A, C, E, B, D, H, G, F, J
A, D, H, J, G, F, C, E, B
A, D, H, J, G, F, C, B, E
A, D, H, G, F, J, C, E, B
A, D, H, G, F, J, C, B, E

2.2 (4pts). The answer has nothing to do with the order of edges.

MST edges: AC CD AB DF AE

3. Sorting (15pts).

// 8 sequences in total： 2 points for each of the first seven sequences, 1 point for the last

9	4	8	5	1	2	3	7	6
4	9	8	5	1	2	3	7	6
4	8	9	5	1	2	3	7	6
4	5	8	9	1	2	3	7	6
1	4	5	8	9	2	3	7	6
1	2	4	5	8	9	3	7	6
1	2	3	4	5	8	9	7	6
1	2	3	4	5	7	8	9	6
1	2	3	4	5	6	7	8	9

4. Big-O (15pts).

a) O(N) // 5pts

b) O(N^2) // 5pts

c) O(N) // 5pts

5. ADTs (17pts).

```
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++) {              // 2pts
    for (int col = 0; col<board.numCols(); col++) {              // 2pts
      // (1) if we are non-zero, see if a matching number merges
      if (board[row][col] != 0) {                                // 2pts
        for (int i = col + 1; i < board.numCols(); i++) {        // 2pts
          //matching number: merge
          if (board[row][i] == board[row][col]) {                // 3pts
            board[row][col] *= 2;
            board[row][i] = 0;
            break;
          //non-matching number: end search
          } else if (board[row][i] != 0){                        // 2pts
            break;
          }
        }
      }
      // (2) if we are blank, see if a number moves into this space
      else {
        for (int i = col + 1; i < board.numCols(); i++){         // 4pts
          if (board[row][i] != 0){
            board[row][col] = board[row][i];
            board[row][i] = 0;
            col--;
```

```
              break;
            }
          }
        }
      }
  }
}
```

6. Trees (18pts).

```
/* APPROACH #1 observes that we really wish to return two values: (1) boolean subtree
* is valid, and (2) int sum of leaves in this subtree. It achieves this using a
* helper with two pass-by-reference parameters. This wrapper calls a helper. The sum
* is not needed by the wrapper (only needed inside the recursion).
*/
bool isValidSumTree_Approach1(TreeNode *tree){                    //  2pts
  int sum = 0;
  bool isValid = false;
  isValidHelper(tree, sum, isValid);
  return isValid;
}


/* APPROACH #1 helper:
* This recursive helper has these two parameters. It performs a post-order traversal
* to gather the sum and then check self for validity.
*
* sum and isValid are used as OUTPUT ONLY (essentially return values)
*/
void isValidHelper(TreeNode *tree, int &sum, bool &isValid) {     //  2pts
  // empty tree is trivially valid
  if (tree == NULL) {
     isValid = true;
     sum = 0;
     return;
  }

  // leaf
  if (tree->left == NULL && tree->right == NULL) {               //  1pts
     // leaf cannot have -1 (or any non-negative) key
```

```
    if (tree->key < 0) {                                  //  2pts
      isValid = false;
      sum = 0;
      return;
    }
    // any other key is fine for leaf
    isValid = true;                                       //  2pts
    sum = tree->key;
    return;
  }

  // post-order traversal for non-leaves
  bool leftIsValid = false;                               //  2pts
  bool rightIsValid = false;
  int leftSum = 0;
  int rightSum = 0;
  isValidHelper(tree->left, leftSum, leftIsValid);        //  4pts
  isValidHelper(tree->right, rightSum, rightIsValid);
  sum = leftSum + rightSum;
  // check for problems
  if (!leftIsValid || !rightIsValid /* subtree invalid */ //  2pts
      || (tree->key != -1 && tree->key != sum) /* sum is wrong */) {
    isValid = false;
    sum = 0;
    return;
  }

  isValid = true;                                         //  1pts
}
```

------------------------------- End of APPROACH #1 --------------------------------------

```
/* APPROACH #2 just re-traverses the tree to gather the descendent leaves' sum at
* every node. Inefficient but easy to write. */
bool isValidSumTree_Approach2(TreeNode *tree){            //  1pts
  // empty tree is trivially valid
  if (tree == NULL) {                                     //  1pts
    return true;
  }

  // leaf is valid if key is non-negative
  if (tree->left == NULL && tree->right == NULL) {        //  2pts
    return tree->key >= 0;
  }

  // check our own key for problems
  if ((tree->key != -1
    && tree->key != sumLeaves(tree->left) + sumLeaves(tree->right))) {    //  3pts
```

```
    return false;
  }

  // recursively check our subtrees for problems
  if (!isValidSumTree_Approach2(tree->left)                     // 3pts
    || !isValidSumTree_Approach2(tree->right)) {
    return false;
  }

  return true;                                                  // 1pts
}

/* APPROACH #2 helper calculates the sum of descendant leaves. It assumes
* the tree is valid, so that must be checked separately. */
int sumLeaves(TreeNode *tree) {                                 // 1pts
  // null contributes nothing to sum
  if (tree == NULL) return 0;                                   // 1pts

  // is leaf?
  if (tree->left == NULL && tree->right == NULL) return tree->key;   // 2pts

  // traversal to sum leaves (ignore own key since we are not leaf)
  return sumLeaves(tree->left) + sumLeaves(tree->right);        // 3pts
}

------------------------------- End of APPROACH #2 -------------------------------------


/* APPROACH #3 overwrites -1 keys with the actual sum, making checking children by
* the parent trivial. */
bool isValidSumTree_Approach3(TreeNode *tree) {                 // 2pts
  // empty tree is trivially valid
  if (tree == NULL) {
    return true;
  }

  // leaf is valid if key is non-negative
  if (tree->left == NULL && tree->right == NULL) {              // 2pts
    return tree->key >= 0;
  }

  // traversal of children will set both left and right children's keys
  // to the actual sums (overwriting -1 if necessary), and also check
  // left and right subtrees for validity
  if (!isValidSumTree_Approach3(tree->left)
       || !isValidSumTree_Approach3(tree->right)) {             // 4pts
    return false;
  }
```

```
  // check our own key for problems
  int leftSum = 0;                                            // 2pts
  int rightSum = 0;
  if (tree->left != NULL) leftSum = tree->left->key;          // 2pts
  if (tree->right != NULL) rightSum = tree->right->key;       // 2pts
  if (tree->key != -1 && tree->key != leftSum + rightSum) {   // 2pts
     return false;
  }

  // make sure our key is actual sum (not -1), to make parent's job of checking
  // its key easier
  tree->key = leftSum + rightSum;                             // 2pts

  return true;
}
```

-------------------------------- End of APPROACH #3--------------------------------------