CS 32 Worksheet 10

This worksheet is entirely **optional**, and meant for extra practice. Some problems will be more challenging than others and are designed to have you apply your knowledge beyond the examples presented in lecture, discussion or projects. All exams will be done on paper, so it is in your best interest to practice these problems by hand and not rely on a compiler. Solutions are written in red. The solutions for **programming** problems are not absolute, it is okay if your code looks different; this is just one way to solve the specific problem.

If you have any questions or concerns please go to any of the LA office hours, or contact raykwan@ucla.edu.

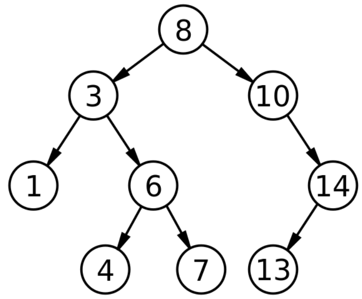
1. You are given the following STL data structure:
   hash_map<student, set<class>> studentClass

   There are S students and an average of C classes per student.

   a. What is the big-O complexity of printing all students alphabetically and for each student, listing each of their classes alphabetically?
   b. What is the big-O complexity of determining who's taking CS32?
   c. What is the big-O complexity of determining if Joe Smith is taking CS32?
   d. Suppose we have the data structure hash_map<string, set<int>> and we wish for a particular class L to print the ID numbers of all the students in that course in sorted order (assume there are S students in a class). What is the big-O complexity?

   a. Hash table is unordered => SlogS + S + SC => SlogS + SC
   b. Must look through EVERY student and check if CS 32 exists their classes => SlogC
   c. Find Joe Smith's classes in O(1) time, check if class exists in a SET of classes => logC
   d. Find class L in O(1) time, print all the students IDs in order in S => S

2.



a. What are the preorder and inorder traversal for the tree?
b. Using the simplest binary search tree (BST) insertion algorithm (no balancing), what is the resulting tree after inserting the nodes 80, 65, 71, 15, 39 and 25, then deleting 10, 6 and 39 in that order.
c. What is the postorder traversal for the resulting tree in part 2?
d. Given the following postorder and inorder traversal of a binary tree (not necessarily a BST), draw the corresponding tree:

INORDER: 1 5 0 8 4 3 7 6 2

POSTORDER: 1 8 4 0 5 2 6 7 3

a. Preorder: 8, 3, 1, 6, 4, 7, 10, 14, 13;

Inorder: 1, 3, 4, 6, 7, 8, 10, 13, 14

```
b.                    8
            3                  14
          1   4            13     80
                7               65
                              15    71
                                  25
```

c. Postorder: 1, 7, 4, 3, 13, 25, 15, 71, 65, 80, 14, 8

d. Solution:

```
                 3
          5             7
      1       0              6
                4               2
              8
```

3. Given the following code:

```
void func(int arr[], int n){

   set<int> s;

   for (int i=0;i<n;i++){
```

```
        for (int j=0;j<i;j++)

            s.insert(arr[i] * arr[j]);

        }
}
```

a. What is the time complexity of the code?
b. If we implement `s` with an `unordered_map` instead, what is the time complexity of the code?
c. What if we just implement it using an array that's sorted? What is its time complexity?

a. O(N^2 logN) b/c a set must maintain sorted order
b. O(N^2) b/c hash tables offer a O(1) insert
c. O(N^3) b/c the values must be inserted in the right order

4. a. Given the following array of integers `{-4, 10, 8, 2, -7, 3, 1, 0}`. Show the resulting array after three iterations of i) insertion sort, ii) bubble sort, and iii) quicksort with the first element as the pivot
b. What sorting algorithm is used for the iterations shown below?

i)

54	26	93	17	77	31	44	55	20
26	54	93	17	77	31	44	55	20
26	54	93	17	77	31	44	55	20
17	26	54	93	77	31	44	55	20
17	26	54	77	93	31	44	55	20

ii)

54	26	93	17	77	31	44	55	20
26	54	93	17	77	31	44	55	20
26	54	93	17	77	31	44	55	20
26	54	17	93	77	31	44	55	20
26	54	17	77	93	31	44	55	20

c. Now given the following array of integers `{-2, -1, 0, 3, 10, 5, 7, 9}`. Which sorting algorithm is most efficient for you to use? Insertion sort, merge sort or heapsort?

a) i) Insertion sort:
-4 10 8 2 -7 3 1 0
-4 8 10 2 -7 3 1 0
-4 2 8 10 -7 3 1 0
ii) Bubble sort:

-4 10 8 2 -7 3 1 0
-4 8 10 2 -7 3 1 0
-4 8 2 10 -7 3 1 0

iii) Quicksort (by swapping):

-7 -4 8 2 3 1 0 10 // initial pivot -4
-7 -4 2 3 1 0 8 10 // pivot1 = -7, pivot2 = 8
-7 -4 1 0 2 3 8 10 // pivot1 = -7, pivot2 = 2, pivot3 = 10

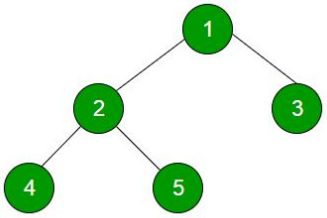
b) i) bubble sort; ii) insertion sort (the first part is always fully sorted)

c) Insertion sort is the best algo for nearly sorted arrays, and arrays with small number of elements

5. a. Given a binary tree, write a code that returns all root-to-leaf paths. Use the function header:

vector<string> rootToLeaf(Node* head)

Example:



All root-to-leaf paths are: ["1->2->4", "1->2->5", "1->3]

Note: Use this Node definition

```
struct Node {
        int val;
        Node* left, right;
};

void rootToLeaf(Node* root, vector<std::string>& paths, std::string curr) {
        if (root->left == nullptr  &&  root->right == nullptr) {
                paths.add(curr);
                return;
        }
        if (root->left != nullptr) {
                rootToLeaf(root->left, paths, curr + "->" +
                    std::to_string(root->left->val));
        }
        if (root->right != nullptr) {
                rootToLeaf(root->right, paths, curr + "->" +
                    std::to_string(root->right->val));
        }
}
```

```
// Use helper
vector<std::string> rootToLeaf(Node* root) {
        vector<std::string> paths;
        if (root != nullptr)
                rootToLeaf(root, paths, std::to_string(root->val));
        return paths;
}
```

b. What is the time complexity of this function for each recursive call?
O(1)/constant because the time complexity for vector insertion is constant time

6. Evaluate the following infix to postfix expression. Show how the stack looks after each operation.

A / (B * C) - (D / E ^ F) * G

		*				/	^			
	(	(			(	(	(		*	
/	/	/	/	-	-	-	-	-	-	
After /	After (	After +	After)	After -	After (	After -	After +	After)	After /	After end

Answer: A B C * / D E F ^ / G * -

7. Consider the following code:

```
void mystery(queue<int>& q) {
   if (q.empty()) {
        return;
    }

   int data = q.front();
   q.pop();
   mystery(q);
   q.push(data);
}
```

Given the queue containing [3, 7, 5, 9, 2, 0, 8]

a. What is the queue being passed in on the first and last recursive call?

Calls mystery on the queue [7,5,9,2,0,8] on the first recursive call, and the queue [8] on the last recursive call

b. What does this function do?
It reverses the queue.

8. Evaluate the following *prefix* expression using integer arithmetic. The answer is a single integer:

   + / * 2 - 8 2 4 * + 1 0 - 6 2

   To evaluate prefix expressions:
   - Reverse the expression to be postfix, i.e. 2 6 - 0 1 + * 42 8 - 2 * / +
   - Push every digit into the stack, and once you hit an operator, evaluate the first 2 digits on the stack and push the result back in the stack
   - Example: our stack should have 2 6 until we hit - and we evaluate 6-2 = 4 and then our stack would be 4, so on...

   FINAL ANSWER: 7

9. At UCLA, each department has its own set of classes. There are D departments and an average of C classes in a department. Give the following data structures:

   unordered_map<department, set<class>> SC1
   set<pair<department, set<class>> SC2

   a. Which one would be better (more efficient) if we want to print out all the classes in each department, with the departments and classes printed in alphabetical order? What is the big-O of the *less efficient* data structure for doing this?

      SC2 more efficient with big-O O(DC), SC1 is worse with time complexity O(DlogD + DC)

   b. What is the big-O of determining what department a specific class belongs to if we use the SC1 data structure?

      Have to iterate through each department -> O(D) + search the set -> O(logC) => O(DlogC)

   c. What is the *worst* data structure to use if we want to print out all the classes in a specific department? What is the big-O?

SC2 would be worse with time complexity O(logD + C) (have to find the department in the set, then iterate to print the classes)

d. What is the big-O for each data structure if we want to see if a certain class is in a specific department?

SC1: hash to department -> O(1) + search class in set -> O(logC)
=> O(logC)
SC2: search for dpt in set -> O(logD) then search class in set -> O(logC)
=> O(logD + logC)

e. Say we have another data structure that stores the student IDs of all the students enrolled in a class (with total of C classes, an average of S students in each class, and each student is enrolled in N classes) defined as unordered_map<class, list<studentIDs>>, what is the big-O if:

List is implemented as linked list so to find, need to iterate through all elements:

i. We want to print out all the students IDs enrolled in a specific class in an increasing order?

Hash to class -> O(1) + sort the list of IDs -> O(SlogS) + iterate and print -> O(S) => O(SlogS + S) = O(SlogS)

ii. We want to print out all the classes a specific student is enrolled in alphabetically?

Iterate through each class and find student -> O(CS) + sort the classes -> O(NlogN) => O(CS + NlogN)

iii. We want to find if a certain student is enrolled in a certain class?

Hash to class -> O(1) + find student in the list -> O(S) => O(S)

10. We say that a tree is balanced if the tree is empty, or the *weight* (defined to be the total of the vals in the nodes of the tree) of the left and right subtrees are equal, and each of those subtrees is itself in balance. Write a function called isInBalance that takes a pointer to the root node of a tree and returns true if the tree is balanced, or false otherwise. The node is defined as:

```
struct Node{
    int val;
    Node* left, right;
```

};

You may write additional functions, but you must not use any global variables, not any variables other than bool, bool&, int, int&, or Node*. You must not use the keywords while, for, goto or static. Your solution shouldn't be more than 30 lines of code. Use the function header: bool isInBalance(Node* root)

Hint: Your solution must be such that a call to isInBalance results in each node in each tree being visited no more than once.

```
// helper function that tells you if this subtree is in balance and also computes the
sum
bool isInBalanceHelper(struct node* root, int& sum) {
   if (root == nullptr) {// empty subtree is in balance by definition
      sum = 0;
      return true;
   }

   int leftSum = 0;
   int rightSum = 0;
   // each subtree must be in balance
   if (isInBalanceHelper(root->left, leftSum) && isInBalanceHelper(root->right,
rightSum)) {
      // this subtree is in balance if the sums of left and right subtrees are equal
      if (leftSum == rightSum) {
         sum = leftSum + rightSum + root->data; // propagate the sum of this subtree
         return true;
      }
   }

   return false;
}

bool isInBalance(struct node* root) {
   int sum;
   return isInBalanceHelper(root, sum);
}
```