CS 32 Worksheet 8

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.

Concepts

Hash Tables, Heaps

1. Given an array arr[0..n-1] of distinct elements and a range [low, high], use a hash table to find all numbers that are in the range, but not in the array. Print out the missing elements in sorted order.

```
For example:
Input:     arr[] = {10, 12, 11, 15}, low = 10, high = 15
Output:    13, 14

Input:     arr[] = {1, 14, 11, 51, 15}, low = 50, high = 55
Output:    50, 52, 53, 54
```

```
#include <unordered_set>
#include <iostream>
using namespace std;

void inRange(int arr[], int size, int low, int high)
{
    // Insert all elements of arr[] in set
    unordered_set<int> set;
    for (int i=0; i<size; i++)
        set.insert(arr[i]);

    // Traverse through the range and print all
    // missing elements
    for (int x=low; x<=high; x++)
        if (set.find(x) == set.end()) //or if (set.count(x) ==
0)
```

```
            cout << x << " ";
    }
```

We use unordered_set here because it stores unique values not in a particular order to allow fast retrieval of values.

2. Given an array of integers and a target sum, determine if two integers in the array can be added together to equal the sum. The time complexity of your solution should be O(n) where n is the number of elements in the array. In other words, you cannot use the brute force method in which you compare each element with every other element using nested for loops. Example:

   Array: 4 8 3 7 9 2 5
   Target: 15

You can take 8 and 7 from the array, and their sum equals the target of 15. Thus, the function we will write will return true. Use the following function header to get started:

```
bool twoSum(const int arr[], int n, int target);
```

```
bool twoSum(const int arr[], int n, int target) {
    unordered_set<int> numsFound;

    // We will add every number to a set as we iterate
    // through the array. If our set ever contains the
    // 'complement' of the number we are looking at, we
    // have found a pair of numbers whose sum is the target
    // and we will return true. Otherwise, if we reach the
    // end of the array, return false.
    for (int i = 0; i < n; i++) {

        int complement = target - arr[i];

        if (numsFound.find(complement) != numsFound.end()) {
            return true;
        }
        else {
            numsFound.insert(arr[i]);
        }
    }
    return false;
```

```
}
```

3. Given a vector of strings, group anagrams together. An anagram is a word formed by rearranging the letters of another, such as *cinema* formed by *iceman*. Return a vector of vectors of strings. Solve this problem using a hash table. You may assume only lower case letters.

   ```
   vector<vector<string>> groupAnagrams(vector<string> strs)
   ```

   For example, given: ["eat", "tea", "tan", "ate", "nat", "bat"],
   Return:
   [
    ["ate", "eat","tea"],
    ["nat","tan"],
    ["bat"]
   ]

```
        // Given a string, compute its hash value based on prime
        // numbers

        int calculateHash(string word) {
  int primes[26] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
         43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101};
           int hash_val = 1;
           for(int i = 0; i < word.size(); i++) {
              // Multiplying by prime numbers ensures that all
words
              // with the same characters will have the same
product
               hash_val *= primes[word[i]-'a'];
           }
           return hash_val;
        }
        // Force collisions into buckets based on primality hash

    vector<vector<string>> groupAnagrams(vector<string> strs){
           unordered_map<int, vector<string>> anagrams;
           for(int i = 0; i < strs.size(); i++) {
               int key = calculateHash(strs[i]);
               anagrams[key].push_back(strs[i]);
           }
  unordered_map<int, vector<string>>::iterator it =
        anagrams.begin();
           vector<vector<string>> res;
```

```
            // Loop through Hash Table
            while(it != anagrams.end()) {
// Only care about the vectors of strings (i.e. anagrams)
                res.push_back(it->second);
                it++;
            }
            return res;
        }
```

4. Given a string, find the first non-repeating character in it and return its index. If it doesn't exist, return -1. You may assume the string contain only lowercase letters. Use a hashtable to solve this problem.
   Examples:
   s = "leetcode"
   return 0
   s = "loveleetcode",
   return 2

```
int firstUniqueChar(std::string s) {
        // Map character to the frequency of occurrence
        unordered_map<char, int> counter;
        for(int i = 0; i < s.size(); i++) {
            counter[s[i]]++;
        }
        for (int i = 0; i < s.size(); i++) {
            if (counter[s[i]] == 1) return i;
        }
        return -1;
}
```

5. Implement the following function:

```
bool isMaxHeap(const int arr[], int len);
```

   This function takes in an array *arr* of length *len* and returns whether or not that array represents a binary max heap. In other words, *arr* must follow the max heap property, where a parent is greater than or equal to its children.

```
bool isMaxHeap(const int arr[], int len) {
  for (int x = 0; x < len; x++) {
    int left = 2 * x + 1;
    int right = left + 1;
```

```
    if (left < len && arr[left] > arr[x])
      return false;
    if (right < len && arr[right] > arr[x])
      return false;
  }
  return true;
}
```

6. Implement the following function, given the following data structure:

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

bool isMinHeap(const Node* head);
```

This function takes in the head of a binary tree and returns whether or not that binary tree represents a binary min heap. In other words, this tree must follow the min heap property, where a parent is less than or equal to its children. (Bonus: Check for the completeness property, where every level of the tree except possibly for the lowest is completely filled, and the lowest level's nodes must be as far left as possible.)

```
// Precondition: head points to a complete tree
bool isMinHeap(const Node* head) {
    if (head == nullptr)
      return true;

    Node* left = head->left;
    Node* right = head->right;
    if (left != nullptr && head->val > left->val)
      return false;
    if (right != nullptr && head->val > right->val)
      return false;

    return isMinHeap(head->left) && isMinHeap(head->right);
}
```

7. Given an array of *n* integers that is guaranteed to satisfy the max heap property, write a function that returns a pointer to a binary tree representing the same binary max heap as the array. (MV)

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

Node* makeMaxHeap(const int a[], int n);

Node* makeMaxHeap(const int a[], int n) {
  return makeMaxHeapHelper(a, n, 0);
}

Node* makeMaxHeapHelper(const int a[], int n, int i) {
  if (i >= n) {
    return nullptr;
  }
  Node* root = new Node;
  root->val = a[i];
  root->left = makeMaxHeapHelper(a, n, 2 * i + 1);
  root->right = makeMaxHeapHelper(a, n, 2 * i + 2);
  return root;
}
```

8. Write a function, sum3, that given an array of integers finds if some three elements of the array sum to 0. Return true if three such elements exist and false if not. Your function must run faster than the brute force O(n^3). You can assume that all the elements of the array will be unique (i.e., no repeated values).

i.e [1,2,3,4,5,6] -> False
[1,-1,2,-2] -> False
[1,2,-3, 6, 8] -> True

```
bool sum3(const int arr[], int n);
```

```
#include <iostream>
#include <unordered_set>
using namespace std;
```

```
bool sum3(const int arr[], int n) {
  //create hash table
  unordered_set<int> hashedArr;
  for(int i = 0; i < n; i++){
    hashedArr.insert(arr[i]);
  }

  //search for opposite of every pair
  for(int i = 0; i < n; i++) {
    for(int j = i+1; j < n; j++) {
      int oppSum = (arr[i] + arr[j])*-1;
      if(oppSum != arr[i] && oppSum != arr[j] &&
     hashedArr.find(oppSum) != hashedArr.end()) {
        return true;
      }
    }
  }

  return false;
}
```

9. You are working at a credit card company and need to store account balances. Each account holder has an integer userId number. Each user/userid can have as many bank accounts as they want specified by an integer accountid. Write a class that supports insertion of a deposit and search of a given user and account id. Insert should update the amount if the given `accountId` and `userId` already exists. The company wants to process a high volume of transactions so they demand search and insertion work in O(1) time, i.e they do not depend on the number of users or bank accounts.
   Hint: Consider an STL container in an STL container

```
class Bank {
public:
void insert (int amount, int userId, int accountId);
int search (int userId, int accountId);
…
}

i.e
Bank B;
B.insert(10, 765, 937)
B.search(765, 937) // returns 10
```

```
class Bank {
     public:
       void insert(int amount, int userId, int accountId);
       int search(int userId, int accountId);
     private:
       unordered_map<int, unordered_map<int, int>> database;
};

void Bank::insert(int amount, int userId, int accountId) {
  auto userMap = database.find(userId);
  if(userMap == database.end()) {
    unordered_map<int, int> userAccountMap;
    userAccountMap.insert({accountId, amount});
    database.insert({userId, userAccountMap});
  } else {
    auto userAccountMap = userMap->second;
    //unspecified what to do when inserting multiple accounts
with same id
    userAccountMap.insert({accountId, amount});
    //alternate way of interacting with unordered_map that
updates and creates
    database[userId] = userAccountMap;
  }
}

int Bank::search(int userId, int accountId) {
  auto userMap = database.find(userId);
  if(userMap == database.end()) {
    return -1;
  }
  unordered_map<int, int> userAccountMap = userMap->second;
  auto account = userAccountMap.find(accountId);
  if(account != userAccountMap.end()) {
    return account->second;
  } else {
    //unspecified what to do here
    return -1;
  }
}
```

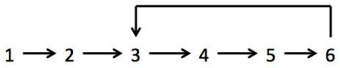
10. Write a function `travelItinerary` which takes in a list of tickets and prints the corresponding travel itinerary. There is only one ticket from every city, besides the final destination and the list of tickets is not cyclic. The maximum time complexity of your algorithm should be O(n). Hint: you will need to use two hash tables.

```
void travelItinerary(map<string, string> tickets);

Input:
Bali → Tokyo
London → Bangkok
Bangkok → Dubai
Dubai → Bali

Output:
London → Bangkok
Bangkok → Dubai
Dubai → Bali
Bali → Tokyo
```

```
void travelItinerary(map<string, string> tickets)
{
    unordered_map<string, string> reversemap;
    unordered_map<string, string>::iterator it;

    // Fill in the reverse map (Each value maps to its key)
    // Dubai --> Bali will now be Bali --> Dubai
    for (it = tickets.begin(); it != tickets.end(); it++)
        reversemap[it->second] = it->first; //

    // Start of itinerary
    string start;

    for (it = tickets.begin(); it != tickets.end(); it++)
    {
        if (reversemap.find(it->first) == reversemap.end())
        {
            start = it->first;
            break;
        }
    }

    // With starting point, we need to go next,
```

```
    // next of next using given hash map
    it = tickets.find(start);
    while (it != tickets.end())
    {
        cout << it->first << "->" << it->second << endl;
        it = tickets.find(it->second);
    }

}
```

11. Given a linked list, determine if it has a cycle in it. This can be done by starting from the head of the linked list and traversing it until you reach a node you have already seen or the end of the list. The time complexity of your solution should be O(n) where n is the number of nodes in the list. Example:



This list has a cycle - Node 6 points back to Node 3

Use the following Node definition and function header to get started:

```
struct Node {
   int val;
   Node* next;
}

bool hasCycle(const Node* head);
```

```
bool hasCycle(const Node* head) {
    unordered_set<const Node*> nodesSeen;
    Node* temp = head;

    while (temp != nullptr)
    {
        if (nodesSeen.find(temp) != nodesSeen.end()) {
           // we've seen this node already, there is a cycle!
            return true;
        }
        else {
           // add this node to the set of ones we've already seen
            nodesSeen.insert(temp);
            temp = temp->next;
        }
    }
```

```
    // we saw all nodes only once, so there is no cycle
    return false;
}
```