COMP3121/3821/9101/9801 18s1
Assignment 1

Jia Jun Luo — z5115679

1. (a) **[5 marks]** Describe an $O(nlogn)$ algorithm (in the sense of the worst case performance) that, given an array S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x. (5 pts)

   (b) **[5 marks]** Describe an algorithm that accomplishes the same task, but runs in $O(n)$ expected (average) time.

   Note that brute force does not work here, because it runs in $O(n^2)$ time.

---

a) An $O(nlogn)$ algorithm (in the sense of the worst case performance) can be achieved by initially sorting the provided array into ascending order using MergeSort. This process will be $O(nlogn)$. Then, the algorithm should iterate through the sorted array once, which is an $O(n)$ process, and perform a binary search of the result of $x - S[n]$, which will be $O(logn)$. If the result is found to be existent in the array S, then the algorithm returns true (that is, there exists two elements in S whose sum is exactly x). If the algorithm does not find any element in the array which matches this condition, then the algorithm will return false.

Example of finding $x = 7$ in the following array:

1	3	2	5	4

After MergeSort:

1	2	3	4	5

Iterating through each $S[n]$ where n ranges from 0 to $n - 1$ and performing binary search on the result: Observing $S[0] = 1 \therefore 7 - 1 = 6$, performing binary search on 6 results in false, therefore we move to observe $S[1] = 2 \therefore 7 - 2 = 5$, where performing binary search on 5 reveals the other number in the required pair is when $n = 4$, returning true and ending the algorithm.

b) An $O(n)$ algorithm can be achieved by inserting all the elements in the array S into a hash table A, which takes $O(n)$ time. The index of the elements will be equal to the value of the elements themselves. Then, using a similar approach to part a, we will go through the original array to find the result m of $x - S[n]$. If the result m exists in A, which can be checked by simply going to $A[m]$, where m is the value in $A[m]$ in $O(1)$ time, the algorithm returns true. If not, it will return false.

Example of finding $x = 100$ in the following array:

50	-30	192	0	33	7

After inserting into a hash table:

Value/Index	-30	...	0	...	7	...	33	...	50	...	192

The algorithm first observes $S[0]$ to be 50 and then calculates $100 - S[0] = 50$. Then it goes straight to $H[50]$, which gives us the value 50, therefore immediately returning true. In another situation where this is not true, the algorithm continues to observe each integer in S until the end, where it will return false.

---

2. **[10 marks]** Youre given an array of n integers, and must answer a series of n queries, each of the form: how many elements of the array have value between L and R?, where L and R are integers. Design an $O(nlogn)$ algorithm that answers all of these queries. (10 pts)

An $O(nlogn)$ algorithm can be achieved by first putting all the values of the array into an AVL tree. This process of insertion takes up $O(logn)$ for each insertion of n items, hence the process of building up the tree is $O(nlogn)$. However, each "node" of the AVL tree should not only store the corresponding value of the array but also the amount of "nodes" with values less than that of the current node.

The total number of elements in the array that have a value between L and R is given by:

TOTAL = 1 (if L is found) + 1 (if R is found) + (number of nodes with values less than R, or the next smallest element than R - number of nodes with values less than L, or the next highest element than L)

So, for each query, we will search for the node containing the value L. If L is not found, then the next highest value is searched for. If L is found, then the

TOTAL is incremented. We do the same for R, but now we will search for the next lowest value if R is not found. The TOTAL is incremented if R is found.

Now, we have a pointer to L (or the next highest of L), and a pointer to R (or the next lowest of R). Therefore, we also have the amount of nodes with lesser values than that of L (let this amount be a) and R (let this amount be b). We perform the operation $b - a$, as shown previously in the equation for TOTAL, to obtain the TOTAL.

Searching for a node in the AVL tree is of order $O(logn)$, and since we do this twice to obtain the TOTAL, the order is of $O(2logn) \implies O(logn)$. Because we do this for n queries, the order becomes of $O(nlogn)$. Overall:

$$T(n) = \underbrace{nlogn}_{\text{AVL creation}} + \overbrace{nlogn}^{\text{search of } n \text{ queries}} = 2nlogn$$

And therefore, is of $O(nlogn)$. The full algorithm in pseudo-code is:

1. While we have not reached the end of the array, add item to AVL tree.
    i. If item is inserted to the left of current node, increment the number of nodes that has less value than that of current node in current node.
    ii. If item is inserted to the right of current node, the newly inserted item should have a "nodes less than" count of 1 + current nodes' "nodes less than" count.

    TOTAL = 0;
2. For n queries: Search for L in AVL tree & search for R in AVL tree.
    i. If L is found, **TOTAL** += 1;
    ii. Else if L is not found, go to next highest node.
    iii. Grab a
    iv. If R is found, **TOTAL** += 1;
    v. Else if R is not found, go to next lowest node.
    vi. Grab b
    vii. Compute $b - a$ and therefore **TOTAL**;
3. Return **TOTAL** for each query. Reset **TOTAL** = 0 after each query.

3. **[10 marks]** There are N teams in the local cricket competition and you happen to have N friends that keenly follow it. Each friend supports some subset (possibly all, or none) of the N teams. Not being the sporty type but wanting to fit in nonetheless you must decide for yourself some subset of teams (possibly all, or none) to support. You dont want to be branded a copycat, so your subset must not be identical to anyone elses. The trouble is, you dont know which friends support which teams, so you can ask your friends some questions of the form Does friend A support team B? (you choose A and B before asking each question). Design an algorithm that determines a suitable subset of teams for you to support and asks as few questions as possible in doing so.

An algorithm which can do this takes use of Cantor's Diagonal Argument, which states that "there are infinite sets which cannot be put into one to one correspondence with the infinite set of natural numbers". We can apply this theorem to this situation as such:

Friend 1 = {**1**, 0, 0, 0, ...}
Friend 2 = {0, **0**, 1, 1, ...}
Friend 3 = {1, 1, **0**, 1, ...}
Friend 4 = {0, 0, 0, **0**, ...}
...
Friend N = ...

Where the sets of each individual are represented, and whether or not a friend supports the team N from index 1 to N is denoted by a 1 for yes and 0 for no.

By analyzing the diagonal of this matrix, and by using Cantor's Diagonal Argument, we can form the set **{0, 1, 1, 1, ...}**, which is the complement of the values in the diagonal of the matrix. This set is unique and hence, the conditions that are required for this algorithm is satisfied in $O(n)$.

Hence, the full algorithm in pseudo-code is:

1. For every N^{th} friend in an $N * N$ matrix:
   i. Ask if he/she supports team team number N.
   ii. Record 1 for no, 0 for yes in an array. (As length of diagonal of an $N * N$ matrix is N, this is $O(n), n = N$).
2. Return array

The time complexity of this algorithm is: $O(n)$.

4. **[10 marks]** Given n numbers $x_1, ..., x_n$ where each x_i is a real number in the interval $[0, 1]$, devise an algorithm that runs in linear time that outputs a permutation of the n numbers, say $y_1, ..., y_n$, such that

$$\sum_{i=2}^{n} |y_i - y_{i-1}| < 2$$

Hint: this is easy to do in $O(nlogn)$ time: just sort the sequence in ascending order. In this case,

$$\sum_{i=2}^{n} |y_i - y_{i-1}| = \sum_{i=2}^{n} (y_i - y_{i-1}) = y_n - y_1 \leq 1 - 0 = 1$$

Here $|y_i - y_{i-1}| = y_i - y_{i-1}$ because all the differences are non-negative, and all the terms in the sum except the first and the last one cancel out. To solve this problem, one might think about tweaking the BUCKETSORT *algorithm.*

To solve this problem, we will use the BucketSort algorithm to sort x_n to form y_n, starting by separating the interval $[0, 1]$ into k segments. These segments will form our individual buckets. BucketSort will work by iterating through all of the n numbers in x_n, placing the numbers in their respective buckets (the segments $[0, 1/k), [1/k, 2/k), ..., [(k-1)/k, 1]$).

The maximum difference between numbers in each of the segments is now $1/k$, and between adjacent numbers, the difference is now observed from the hint above to be less than or equal to 1. These numbers now form the output numbers y_n.

However, we still have to determine what k is, and this can be determined depending on the n number of input numbers. The sum is bound by $((n - k + 1)/k) + ((k - 1)/k)$ and letting this be less than 2 gives:

$$n/2 < k$$

Therefore $k = n/2 + 1$ for the sum of adjacent y_i numbers to be less than 2. The full algorithm in pseudo-code is:

1. Calculate k using $k = n/2 + 1$
2. Perform BucketSort by splitting into k segments.
3. Iterate through x_i and place into respective buckets. Sort until in ascending order.
4. **Return** the newly sorted numbers y_i.

5. You are at a party attended by n people (not including yourself), and you suspect that there might be a celebrity present. A celebrity is someone known by everyone, but does not know anyone except herself/himself. (Of course everyone knows herself/himself)

   Your task is to work out if there is a celebrity present, and if so, which of the n people present is a celebrity. To do so, you can ask a person X if they know another person Y (where you choose X and Y when asking the question). (10 pts)

   a) [**10 marks**] Show that your task can always be accomplished by asking no more than $3n - 3$ such questions, even in the worst case.

   b) [**5 marks**] Show that your task can always be accomplished by asking no more than $3n - \lfloor log_2 n \rfloor - 2$ such questions, even in the worst case.

---

a) We can implement this algorithm using a stack with these two facts: **1)** If person X asks person Y but X does not know Y, Y is not a celebrity but X may be. **2)** If person X asks person Y and X knows Y, it may be that X is not a celebrity.

We start by placing all n people on a stack and continually popping the top two people off, and asking the above questions with those two people. In the first situation, Y is discarded and X is placed back on top of the stack. In the second situation, X is discarded and Y is placed back on top of the stack. This is done until the stack is empty, or until only the celebrity remains.

The full algorithm in pseudo-code is (algorithm is continued onto the next page):

1. Place everyone on a stack.
2. While celebrity has not been found:
   i. Pop 2 people off.
   ii. Ask if X knows Y and vice-versa. (2 questions for each pair of people $= 2(n-1)$ questions)
   iii. Remove X if X knows, else remove Y if Y knows.
   iv. Push the remaining person back onto the stack.
3. Confirm the remaining person is known by everyone else but does not know anyone except herself/himself. This means asking if the remaining person knows everyone else and therefore another $(n-1)$ questions are required (Asking everyone except himself/herself).
   i. If the remaining person knows no-one else, the remaining person is a celebrity.

ii. Else, the remaining person is popped off the stack and no-one is a celebrity.

Therefore, even in the worst case, the amount of questions asked is

$$2(n-1) + (n-1) = 3(n-1) = 3n - 3.$$

b) We can adjust the algorithm even further to reduce the number of questions asked by simply changing the ADT from a stack into a queue while still applying the same two situations as stated previously.

The full algorithm in pseudo-code is now:

1. Place everyone in a queue.
2. While celebrity has not been found:
   i. Take the first 2 people off the queue.
   ii. Ask if X knows Y and vice-versa. (Again, $2(n-1)$ questions, but this time we actively try to validate if the celebrity if he/she exists.)
   iii. Remove X if X knows, else remove Y if Y knows.
   iv. Place the remaining person at the end of the queue.

Because of this change in implementation, we do not repeat any questions already asked in the while loop, and since the celebrity is actively involved during the elimination phase, **at least** $log_2 n$ questions (It is $log_2 n$ as after each loop, half the people are eliminated and therefore the amount of questions asked including the celebrity are continually halved) or n (the maximum number of questions that could be asked including the celebrity, **which is inclusive of the celebrity so it is not** $(n-1)$ **like the stack implementation**) $-\lfloor log_2 n \rfloor$ questions are asked involving the celebrity.

Therefore, even in the worst case, the amount of questions asked is:

$$2(n-1) + n - \lfloor log_2 n \rfloor = 3n - \lfloor log_2 n \rfloor - 2$$

6. You are conducting an election among a class of n students. Each student casts precisely one vote by writing their name, and that of their chosen classmate on a single piece of paper. However, the students have forgotten to specify the order of names on each piece of paper - for instance, Alice Bob could mean Alice voted for Bob, or Bob voted for Alice!

   a) **[2 marks]** Show how you can still uniquely determine how many votes each student received.

   b) **[1 mark]** Hence, explain how you can determine which students did not receive any votes. Can you determine who these students voted for?

   c) **[2 marks]** Suppose every student received at least one vote. What is the maximum possible number of votes received by any student? Justify your answer.

   d) **[7 + 3 = 10 marks]** Using parts (b) and (c), or otherwise, design an algorithm that constructs a list of votes of the form X voted for Y consistent with the pieces of paper. Specifically, each piece of paper should match up with precisely one of these votes. If multiple such lists exist, produce any. An $O(n^2)$ algorithm earns 7 marks, and an $O(n)$ algorithm earns an additional 3 marks.

   Hint: first, use part (c) to consider how you would solve it in the case where every student received at least one vote. Then, apply part (b).

We assume a student cannot vote for himself/herself and everyone casts 1 vote. Let's consider an example of 3 students: A, B and C, and all the possible voting combinations. This would be: 1. [AB,BA,CA], 2. [AB,BA,CB], 3. [AB,BC,CB], 4. [AB,BC,CA], 5. [AC,BA,CA], 6. [AC,BA,CB], 7. [AC,BC,CB], 8. [AC,BC,CA].

Observe the first set. Let's define a pair to be two votes where both of the names on the votes are the same, and let's say AB means A voted for B and BA means B voted for A (in the context of pairs). AB and BA occur in the first set, meaning A voted for B and B voted for A. This is the same if it were AB & AB or BA & BA as no one can vote twice. Now, looking at CA, we recognize that AC (or A voted for C) doesn't make sense as we have already accounted for A's vote. That means C voted for A. Hence, A received 2 votes, B received 1 vote and C received no votes. From this, we can deduce that if two votes match (in other words they both have the same contents), both members in the votes were voted for by the other member. This result is similar in sets 2, 3, 5, 7 and 8. It is interesting to note that A has 3 occurrences and 2 votes. B has 2 occurrences and 1 vote. C has 1 occurrence

and no votes. As such, it is interesting to observe that:

The number of votes $= m - 1$, where m is the occurrence of the name. ***(Eqn1)***

Now, observe sets 4 and 6. These votes are unique and no other vote occurs with the same contents. This means everyone was voted for once. Again, the number of votes $= m - 1$ where m is the occurrence of the name.

a) As seen from the previous example, we can uniquely determine how many votes each person received by observing if there are pairs of votes (votes which have the same 2 names), which means that both members receive a vote. We build on this fact as this means that both members have voted, and any other vote containing their name (of either member in the pairs of votes) means that the member gets an additional vote. We can also use ***(Eqn1)***.

b) Students who did not receive any votes are the ones who voted for another person who was already in a pair. We can determine who they voted for simply by observing these pairs, as seen from the previous example. We can also use ***(Eqn1)***: the names of students who did not receive any votes are only mentioned once.

c) If not student can vote for themselves, and each student can only vote once, the maximum possible number of students received by any student is one. This can also be observed from the previous example. This is because in this situation, no pair exists, meaning the person a student voted for also voted for someone else. We can also use ***(Eqn1)***: each name was mentioned twice in the votes.

d) An $O(n^2)$ algorithm can be made by representing all the classmates as vertices and all the votes to be edges in an undirected multigraph. The pseudo-code for this algorithm is as follows:

1. Build an undirected multrigraph using that is represented with a matrix.
2. Place all the students in a queue.
3. While the queue is not empty:
    i. Pop a student A off the queue.
    ii. Observe the student and his/her connected components in the matrix.
        A. If there are 2 connections from another student B, this means that A voted for B and B voted for A.
        B. Else if there are no connections from another student B this means that no votes are counted.
        C. Else if there is 1 connection from another student C:

a) If that student C is no longer in the queue, this means the votes for that student is already accounted for and therefore A voted for C.
b) Else if the student is still in the queue, it is still inconclusive and no votes are counted.