9801 assignment 1

z5100764

Chunnan Sheng

Six questions, marked out of a total of **70 marks**.
For submitting your assignment, please read and follow the instruc tions given in course homepage otherwise, your submission will not be considered as a valid submission for the marking process. (http://cgi.cse.unsw.edu.au/ cs3121/assignments.php)

1. (a) **[5 marks]** Describe an $O(n \log n)$ 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)

```
// find_the_other is a recursive search function
// using divide and conquer mechanism
bool find_the_other(Array, the_other, start, end, index)
{
    if (
        (the_other == Array[start] && (start != index))
        ||
        (the_other == Array[end] && (end != index))
        )
    {
        return true;
    }

    if (end - start == 1)
    {
        return false;
    }
    else
    {
        mid = (start + end) / 2;

        if (the_other < Array[mid])
        {
            return find_the_other(Array, the_other, start, mid, index);
        }
        else
        {
            return find_the_other(Array, the_other, mid, end, index);
        }
    }
}

// ------------------------------------------
// This is the main function of this algorithm
// ------------------------------------------
bool search(Array, x)
{
    merge_sort(Array); // within O(n * log(n)) time;
    // Result of this sort is in increasing order

    n = Array.length();

    for (index from 0 to n - 1) // Each cycle will be log(n) time
    {
        the_other = x - Array[index];

        // if value of the_other is already outside the array
        if (the_other < Array[0] || the_other > Array[n - 1])
        {
            return false;
        }

        // if value of the_other is within the array
        if (find_the_other(Array, the_other, 0, n - 1, index))
        {
            return true;
        }
    }

    return false;
}
```

(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.

```
bool search(Array, x)
{
    // Define an empty hash set
    Hashset = create_hash_set<int>();

    // Put all integers of this array into a hash set
    for (each element in Array)
    {
        // Insert an element into the hash set
        // duplicate elements are merged to one element
        Hashset.insert(element);
    }

    n = Array.length();

    for (index from 0 to n - 1)
    // Each cycle will be roughly constant time
    {
        // remove current element of this array from the hash set
        Hashset.remove(Array[index]);
        the_other = x - Array[index];

        if (Hashset.find(the_other))
        {
            return true;
        }

        // Return the current element to the hash set
        Hashset.insert(Array[index]);
    }

    return false;
}
```

2. **[10 marks]** You're 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 $O(n \log n)$ algorithm that answers all of these queries. (10 pts)

```
// Search for position of the value in a specified range
// from start to end.
// Time complexity of this search is O(log(n))
(int, int) search_for_position(Array, value, start, end)
{
    if (end - start == 1)
    {
        return (start, end);
    }
    else
    {
        mid = (start + end) / 2;

        if (value <= Array[mid])
        {
            return search_for_position(Array, value, start, mid);
        }
        else
        {
            return search_for_position(Array, value, mid, end);
        }
    }
```

```
}

// Search for position of the value in a sorted array
(int, int) search_for_position(Array, value)
{
    if (value < Array[0])
    {
        return (-1, 0);
    }

    n = Array.length();

    if (value > Array[n - 1])
    {
        return (n - 1, n);
    }

    return search_for_position(Array, value, 0, n - 1);
}

// -----------------------------------------
// This is the main function of this algorithm
// -----------------------------------------
vector<int> answer(Array, Queries)
{
    merge_sort(Array); // within O(n * log(n)) time;
    // Result of this sort is in increasing order

    answers = create_vector<int>();

    for (each query in Queries)
    // each cycle of for loop uses O(log(n)) time
    {
        // Determine the position of L which is between a1 and a2
        (a1, a2) = search_for_position(Array, query.L); // O(log(n))

        // Determine the position of R which is between b1 and b2
        (b1, b2) = search_for_position(Array, query.R); // O(log(n))

        // The following rows try to determine how many
        // elements are between these two positions

        integers_between = b1 - a2 + 1;

        if (a1 >= 0 && query.L == Array[a1])
        {
            ++integers_between;
        }

        if (b2 < Array.length() && query.R == Array[b2])
        {
            ++integers_between;
        }

        answers.push_back(integers_between);
    }

    return answers;
}
```

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 don't want to be branded a copycat, so your subset must not be identical to anyone else's. The trouble is, you don't 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.

```
// ------------------------------------------
// This is the main function of this algorithm
//          Time complexity: O(n)
// ------------------------------------------
vector<cricket_team> ask_questions(Friends, Cricket_Teams)
{
    // We assume that number of Friends equals to
    // number of Teams
    n = Friends.length();

    my_teams = create_vector<cricket_team>();

    // Go through each friend and
    // ask Friend(A) if Friend(A) supports Cricket_Teams(A)
    for (index from 0 to n - 1)
    {
        if (!do_you_support(Friend[index], Cricket_Teams[index]))
        {
            // Add the team that the friend does not support to my favorite teams
            my_teams.push_back(Cricket_Teams[index]);
        }
    }
    return my_teams;
}
```

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 $O(n \log n)$ *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.*

```
//*******************************
//
//     Question 4 solution 1
//
//*******************************

// Each bucket may contain multiple numbers.
// We do not need to sort these numbers within one bucket.
class bucket
{
    vector<double> numbers;
}
```

```
// Drop all the numbers into buckets.
// This function will spend O(n) time.
vector<bucket> drop_numbers_into_buckets(Numbers, int number_of_buckets)
{
    buckets = create_vector<bucket>(number_of_buckets);

    for (each number in Numbers)
    {
        // Figure out ID of the bucket
        bucket_ID = int(number * number_of_buckets);

        // Sometimes there is a boundary problem
        if (bucket_ID == number_of_buckets)
        {
            --bucket_ID;
        }

        // push the new number into this bucket
        buckets[bucket_ID].numbers.push_back(number);
    }

    return buckets;
}


// ------------------------------------------------
// This is the main function of this algorithm
//
// Time complexity will be exactly O(n) despite how
// these real numbers are distributed in (0, 1)
// ------------------------------------------------
vector<double> sort(Numbers)
{
    // If number of buckets equals to amount of numbers,
    // sum(abs(Numbers[i] - Numbers[i - 1])) is guaranteed to be less than 2
    number_of_buckets = Numbers.len();

    buckets = drop_numbers_into_buckets(Numbers, number_of_buckets);

    reordered_numbers = create_vector<double>();

    // Pick up numbers from buckets
    m = buckets.length();
    for (i from 0 to m - 1)
    {
        n = size_of(bucket[i].numbers);
        for(j from 0 to n - 1)
        {
            reordered_numbers.push_back(bucket[i].numbers[j]);
        }
    }

    return reordered_numbers;
}
```

```
//*******************************
//
//     Question 4 solution 2
//
//*******************************

//-------------------------------------------------------------------------
// The question is:
// Is it OK to convert real numbers into
// integral numbers that can be used by radix sort?
//
// A floating number (real number) usually contains 3 parts
// from higher binary digits to lower binary digits as follows:
//
// sign part
// exponent part
// fraction part
//
// In this case, all real numbers are >= 0, so we don't care about this part.
//
// If the sign part of two real numbers are the same, the real number with
// bigger exponent will be larger.
//
// If the exponent part of two real numbers are the same, the real number with
// bigger fraction part will be larger.
//
// Therefore, if we convert real numbers into integral numbers, size relationships
// between these numbers DO NOT change.
//
// Thus, it is OK to convert real numbers into integral numbers for radix sort
//
// -------------------------------------------------------------------------

// Each bucket may contain more than one number
class bucket
{
    vector<double> numbers;
}

// Drop all the numbers into buckets.
// This function will spend O(n) time.
void drop_numbers_into_buckets(Numbers, buckets, index)
{
    for (each number in Numbers)
    {
        // Convert the real number into an array of integral bytes
        byte_array = convert_real_number_into_byte_array(number);

        // Figure out ID of the bucket
        // which is the [index]-th byte of the byte array
        bucket_ID = byte_array[index];

        // push the new number into this bucket
        buckets[bucket_ID].numbers.push_back(number);
    }

    return buckets;
}

// Pickup numbers from buckets
// This function will spend O(n) time
void pickup_numbers_from_buckets(Numbers, buckets);
{
    n = size_of(buckets);

    for (i from 0 to n - 1)
    {
```

```
        m = size_of(buckets[i].numbers)
        for (j from 0 to m - 1)
        {
            Numbers.push_back(buckets[i].numbers[j]);
        }
        buckets[i].numbers.clear();
    }
}

// ----------------------------------------------------
// This is the main function of this algorithm
//
// Time complexity will be precisely O(m * n) where
// m is number of digits of each number,
// n is number of the real numbers.
// If m is considered as a constant coefficient, then the
// time complexity of this algorithm is O(n)
// ----------------------------------------------------
vector<double> sort(Numbers)
{
    // Each digit represent a byte (from 0 to 255)
    number_of_buckets = 256;
    buckets = create_vector<bucket>(number_of_buckets);

    // Number of digits is size of the floating number in bytes
    number_of_digits = size_of(double);

    // Radix sort
    // The real numbers will be converted into integral bytes
    for (index from 0 to number_of_digits - 1)
    {
        drop_numbers_into_buckets(Numbers, buckets, index);

        Numbers.clear();

        pickup_numbers_from_buckets(Numbers, buckets);
    }

    return Numbers;
}
```

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.

```
// Entire time complexity of this algorithm is O(n)
Person find_celebrity(People)
{
    // Push all the people into a stack
    stack = create_stack<Person>();
    for (each person in People)
    {
        stack.push(person);
    }

    // This while loop will ask n - 1 questions.
    while (stack.size() > 1)
    {
        person1 = stack.top();
        stack.pop();

        person2 = stack.top();
        stack.pop();

        // Ask person1 a question whether person1 knows person2
        if (knows(person1, person2))
        {
            // person1 is NOT a celebrity
            // push person2 back into the stack
            stack.push(person2);
        }
        else
        {
            // person2 is NOT a celebrity
            // push person1 back into the stack
            stack.push(person1);
        }
    }

    // The last person in this stack is potentially a celebrity
    celebrity = stack.top();
    // Remove the last person from the container of people
    // We assume that the container is a hash set, so that removing
    // one element only costs O(1) time.
    People.remove(celebrity);

    // This for loop will ask at most 2(n - 1) questions
    for (each person in People)
    {
        if (!knows(person, celebrity))
        {
            // There is at least one person who does not
            // know this celebrity, then there is no celebrity
            return NULL;
        }
```

```
        if (knows(celebrity, person))
        {
            // The celebrity knows somebody else, then these is
            // no celebrity
            return NULL;
        }
    }

    return celebrity;
}
```

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

```
// Entire time complexity of this algorithm is O(n)
Person find_celebrity(People)
{
    to_delete = create_vector<Person>();
    people_set = create_hashset<Person>();
    asked_questions = create_hash_map<Question, bool>();

    // Insert all the people into a hash set
    for (each person in People)
    {
        people_set.insert(person);
    }

    // This loop will ask n - 1 questions
    while (people_set.length() > 1)
    {
        for (each 2 persons in people_set)
        {
            person1 = persons[0];
            person2 = persons[1];
            // Ask person1 a question whether person1 knows person2
            // We should execute the question so that this question is asked
            if (Question(person1, person2).execute())
            {
                // person1 is NOT a celebrity
                to_delete.push_back(person1);

                asked_questions.insert(Question(person1, person2), true);
            }
            else
            {
                // person2 is NOT a celebrity
                to_delete.push_back(person2);

                asked_questions.insert(Question(person1, person2), false);
            }
        }

        // Delete the people who are not celebrity
        for (each person in to_delete)
        {
            people_set.remove(person);
        }
    }

    // The last person in this hash set is potentially a celebrity
    celebrity = *(people_set.begin());
    // Remove the last person from the container of people
    // We assume that the container is a hash set, so that removing
    // one element only costs O(1) time.
    People.remove(celebrity);

    // This loop will ask 2 * n - log(n) - 1 questions
```

```
    for (each person in People)
    {
        // Find answer to this question in asked questions
        answer = asked_questions.find(Question(person, celebrity))
        if (NULL == answer)
        {
            // If answer is not found, execute the question
            answer = Question(person, celebrity).execute();
        }

        if (!answer)
        {
            // There is at least one person who does not
            // know this celebrity, then there is no celebrity
            return NULL;
        }

        // Find answer to this question in asked questions
        answer = asked_questions.find(Question(celebrity, person))
        if (NULL == answer)
        {
            // If answer is not found, execute the question
            answer = Question(celebrity, person).execute();
        }

        if (answer)
        {
            // The celebrity knows somebody else, then these is
            // no celebrity
            return NULL;
        }
    }

    return celebrity;
}
```

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.

```
/*
Suppose there is a student A in these students.
Then there should be at least one piece of paper where name of A is written
because every student should vote another student.
If there are no students voting A, then, there is only one piece of paper with A's name.
If there are students voting A, then, there should be more than one piece of paper with A's name.

Therefore, it is determined that there are (number_of_paper_with_A - 1) students voting A.
*/
```

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

```
/*
If there is only one piece of paper writing A's name, then no people vote A.
Then, it is determined the other name on this piece of paper is the student who A votes 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.

```
/*
Since each student can only vote one person who is not him/herself, then the total votes
should be equal to the number of students.
Therefore, if every student receives at least one vote, then every student receives exactly one
vote.
*/
```

(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).

```
class StudentInfo
{
    int votes;
    hash_set<NameCard> cards;
}
```

```
// O(n) time complexity
// A queue and two hash maps are used
hash_map<string, string> find_solution(name_cards)
{
    // This hash map stores all the information of students
    name_info_pairs = create_hash_map<string, StudentInfo>();

    // This hash map is the answer to return
    name_name_pairs = create_hash_map<string, string>();

    // This queue stores students who have zero votes
    zero_vote_students = create_queue<string>();

    // Go through all the cards and
    // accumulate votes received by each student
    // Time complexity of this loop is O(n)
    for(each name_card in name_cards)
    {
        for (each name in name_card)
        {
            if (!name_info_pairs.find(name))
            {
                // If the name does not exist in name_info_pairs, insert this person.
                // Use 0 as an initial value of number of votes.
                StudentInfo info;
                info.votes = 0;
                // Insert this card to this student's card set.
                info.cards.insert(name_card);

                name_info_pairs.insert(name, info);
            }
            else
            {
                // If this name already exists in name_info_pairs,
                // Plus his/her votes by one
                ++name_info_pairs[name].votes;
                // Add this card to this student's card set
                name_info_pairs[name].cards.insert(name_card);
            }
        }
    }

    // Push all students who have zero votes into a queue
    // Time complexity: O(n)
    for(each k_v_pair in name_info_pairs)
    {
        if (0 == k_v_pair.value.votes)
        {
            zero_vote_students.push_back(k_v_pair.key);
        }
    }

    // This while loop spends O(n) time
    while (!zero_vote_students.is_empty())
    {
        // Get name of the student from the queue
        to_delete = zero_vote_students.front();
        // Remove this student from the queue
        zero_vote_students.pop_front();

        // To find the other person this student votes for costs constant time
        vote_for = (*(name_info_pairs[to_delete].cards.begin())).the_other_name(to_delete);

        // Delete this student from the hash map
        name_info_pairs.remove(to_delete);

        // Insert this student and the person the student votes for as a pair
        // into the answer
        name_name_pairs.insert(to_delete, vote_for);
```

```
        // Decrease number of votes of the other person because
        // the other person has lost his voter because his voter was deleted
        --name_info_pairs[vote_for].votes;

        // If votes of the other person is zero,
        // push the other person into the queue as well
        if (0 == name_info_pairs[vote_for].votes)
        {
            zero_vote_students.push_back(vote_for);
        }

        // Remove the voter's information from the other student
        // Order of names on the card may vary
        if (!name_info_pairs[vote_for].cards.remove(NameCard(to_delete, vote_for)))
        {
            name_info_pairs[vote_for].cards.remove(NameCard(vote_for, to_delete));
        }
    }


    // Go through all the other students
    // Right now, every student has at least one vote
    // Time complexity of this loop is O(n)
    for(each k_v_pair in name_info_pairs)
    {
        voter = k_v_pair.key;
        vote_for = (*(k_v_pair.value.cards.begin())).the_other_name(name);

        // If the voter already exists in name_name_pairs,
        // swap names of the voter and the person the voter votes for.
        if (name_name_pairs.find(voter))
        {
            tmp = voter;
            voter = vote_for;
            vote_for = tmp;
        }

        // Insert the voter and the person the voter votes for as a pair
        name_name_pairs.insert(voter, vote_for);
    }

    return name_name_pairs;
}
```