National University of Singapore

School of Computing

CS2040C - Data Structures and Algorithms

Final Assessment

(Semester 1 AY2018/19)

Time Allowed: 2 hours

INSTRUCTIONS TO CANDIDATES:

1. Do **NOT** open this assessment paper until you are told to do so.
2. This assessment paper contains THREE (3) sections and comprises SIXTEEN (16) printed pages, including this page.
3. This is an **Open Book Assessment**.
4. Answer **ALL** questions within the **boxed spaces** in this booklet.
   You can use either pen or pencil. Just make sure you write **legibly**!
5. Important tips: Pace yourself! Do **not** spend too much time on one (hard) question.
   Read all the questions first! Some questions might be easier than they appear.
6. You can use any **standard, non-modified** algorithm discussed in class by just mentioning its name.
7. Write your Student Number in the box below:

A	0							

This portion is for examiner's use only

Section	Maximum Marks	Your Marks	Remarks
A	20		
B	30		
C	50		
Total	100		

A Basics (20 marks)

A.1 Worst Case Time Complexity Analysis (10 marks)

Write down the *tightest*[1] *worst case* time complexity of the various data structure operations or algorithms below. Every correct answer is worth 1 mark.

The operations/algorithms referred to below are the **unmodified version**, as per discussion in class or as currently implemented in C++ STL. For graph-related operations, let n be the number of vertices and m be the number of edges. Otherwise, n denotes the size of data as usual. AM/AL/EL are the abbreviations for Adjacency Matrix/Adjacency List/Edge List, respectively. Unless specifically mentioned, all graph-related operations are performed on simple graphs stored in an AL data structure.

No	Operation	Complexity
1	Inserting an element into the *2nd last* position of a **circular singly** linked list.	$O(\_\_\_\_)$
2	Running DFS(t) where t is the *root* of a **tree**.	$O(\_\_\_\_)$
3	Running BFS(u) when the *out*-degree of vertex u is **0**.	$O(\_\_\_\_)$
4	Running Radix Sort on an *already sorted* array of d**-digit** integers.	$O(\_\_\_\_)$
5	Searching for a *key* in a C++ STL `unordered_map`.	$O(\_\_\_\_)$
6	Deleting $\frac{n}{2}$ items **one by one** from an AVL tree.	$O(\_\_\_\_)$
7	Adding a new **directed** edge into an Adjacency Matrix.	$O(\_\_\_\_)$
8	Running Dijkstra's algorithm on a **positive weighted** graph.	$O(\_\_\_\_)$
9	Running Bellman-Ford's algorithm on a graph to detect **negative weight** cycles.	$O(\_\_\_\_)$
10	Inserting n integers **one by one** into a C++ STL `set`.	$O(\_\_\_\_)$

[1] What we mean by tightest worst case time complexity is as follows: If an operation of the stipulated data structure/algorithm needs at best $O(n^3)$ if given the worst possible input but you answer higher time complexities than that, e.g. $O(n^4)$ – which technically also upperbounds $O(n^3)$, you will be marked wrong.

A.2 Fill in the Blanks (10 marks)

Each correct answer is worth 1 mark.

1. In a binary search tree, the **maximum** key is stored in the right-most leaf node, while in a max heap, the **maximum** value is stored in the root node.
2. A minimal AVL tree of height 6, i.e. n(6) as discussed in class, has 33 nodes.
3. In a C++ STL `set`, keys must be unique. If we need to store duplicate keys in our set, we can use C++ STL multiset.
4. A complete **undirected** graph of n vertices will have (V(V-1))/ 2 edges.
5. If we need to count the number of edges in a graph, the most appropriate graph data structure to use is an EdgeList, but to enumerate the neighbours of a vertex, we use an Adjacency List.
6. For checking if a graph is a bipartite graph, we can use the BFS/DFS algorithm.
7. One similarity between an AVL tree and a binary heap (min-heap or max-heap) is that they are both (you cannot answer trees or data structures) Complete.
8. The maximum number of topological sort orderings on a directed graph of n vertices is n!.

Section A Marks = _____ + _____ = _____

B Short Questions (30 marks)

B.1 Analysis (15 marks)

Prove (show that the statement is correct) or disprove (give a counter example) the statements below.

1. It is better to use hash table with separate chaining instead of open addressing techniques if there are many deletions and insertions.

LP/QP/DH is bad for repeated deleted and inserts as there is a long travel time as you would walk over many DEL nodes just to find a space to insert the number
Ie:
14 21 1 blank 18 blank blank
If I insert 2, it will start at cluster 1 (14, 21, 1) and move all the way to the end of the cluster in very few steps
For Closed Addressing, when you delete an item, we don't leave DEL markers in our hashtable, when we delete something, we pop it out of the list. If you look at HashTable_Demo.cpp, we use a vector of pairs to store key value pairs. Make sure table size is a prime to minimize collision and to make load factor small N/M that is usually 1 or 2 or ½

2. Inserting a new element that is greater than the maximum element currently in a binary max-heap with $N = 15$ elements will definitely incur more than 3 swaps.

True. As the height of a binary heap with N = 15 elements is height = 3, resulting in exactly 3 swaps if the bigger element bigger than the root is added to the leaf of the heap.

3. The number of nodes of a complete binary tree of height h (as defined in the lectures) can range from $2^{(h-1)} - 1$ to $2^h - 1$ inclusively.

The range of a complete binary tree of height h is 2^h to 2^(h+1) - 1 so this equation is completely wrong.

4. All Directed Acyclic Graphs (DAGs) have more than one topological sort order.

Not true, for a graph of (0, 1), (0, 2), (1, 2), there is only one topological sort order

5. Dijkstra's algorithm is the best algorithm for solving the single source shortest path problem given any *kind of graph*.

Dijkstra algo is in ((V+E) log V) time. However, for graphs like DAG, you can find the SSSP using a topological sort followed by relaxing all the neighbouring vertices in that topological sorted order in O(V+E) time which is faster

B.2 Short Questions (15 marks)

1. What is the time complexity of the following C++ function? Explain your answer.

```
int F(int n){
     int sum = 0;
     for (int i = 0; i < n; ++i)
         for (int j = 0; j < n; j += 2)
             for (int k = 0; k < 2; ++k)
                  sum++;
     return sum;
}
```

This is n * n/2 * 2 = n^2 = O(N^2)

2. Perform the following operations in the order given below using the following hash function and using quadratic probing for collision resolution. Show the resultant hash table provided below.

   Hash function is $H(key) = (key)\%13$

   Insert 89, Insert 45, Insert 92, Insert 58, Delete 45, Insert 13, Insert 67, Insert 51, Insert 26, Insert 71, Insert 39.

idx	0	1	2	3	4	5	6	7	8	9	10	11	12
val													

3. Show the adjacency list for the graph shown in figure 1.

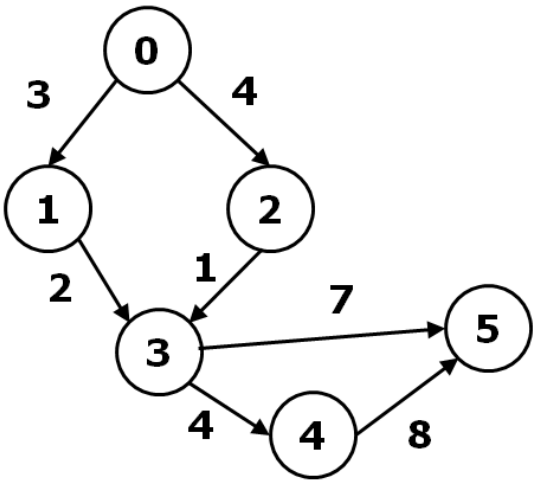

Figure 1: Directed Graph

0: <1,3>, <2, 4>
1: <3,2>
2: <3, 1>
3: <4,4>, <5,7>
4: <5,8>
5:

the pairs are <u, w>, with vertex u and weight value of w

Section B Marks = _____ + _____ = _____

C Applications (50 marks)

C.1 Stack ADT with 'remove' Operation (20 marks)

Assuming that we are dealing with a stack of names, let us define one new operation of a standard Stack like the one that we discussed in class. This new operation is `remove(string value)` that should remove the string 'value' from the stack. In addition, the following conditions are guaranteed:

1. The stack S starts off as empty.
2. All `push` operations will be called with ***distinct* names with length less than 80 characters**. In other words, the same name will not be pushed into the stack S more than once.
3. The `remove` operation will only be called with a name that exists in the stack S.
4. The `pop` and `top` operations will only be called when the stack S is not empty.

Show how you are going to implement the Stack ADT with this new operation in $O(1)$ while maintaining that the standard push, top, pop operations of a stack all remain $O(1)$ too. Implementations that perform each operation in $O(N)$ time complexity or higher will receive significantly lesser marks. You are allowed and encouraged to use C++ STL Stack, Linked List and other STL data structures. A skeleton C++ code has been written for you below. Please complete and/or modify it as you see fit. If you are not sure of the required C++ syntax, you can write your answer in pseudo-code for slightly lesser marks.

```
#include <bits/stdc++.h>
using namespace std;
class StackWithRemove {
private:
// variables and additional private functions if required
```

```
public:
  StackWithRemove() {
    using a std::map<int, string> with a counter int = 0 at the start. With each new addition ++start, and we
    insert into the map with (start, string), this means each addition is in O(log n) time however. but all removal
    and push etc is in O(log n) time, to find top of stack is *map.end()

    Another idea is to use unorderd_set<string> and a normal stack. When we delete from stack we first delete from the
    unordered_set. If the top() is not in the set, then pop() again until we find the correct value. But is this O(1)?

  }
  void push(string value) { // put ‘value’ at the top of the stack









  }
  string top() { // return the current string at the top of the stack









  }
  void remove(string value) { // remove the string ‘value’ from the stack









  }
  void pop() { // pop the topmost string from the stack









  }
};
```

```
int main() {
  StackWithRemove S;
  S.push("Gary"); S.push("JunAn"); S.push("Tapas");
  S.push("Ranald"); S.push("Sidhant"); S.push("Steven");
  cout << S.top() << endl;          // should be Steven
  S.pop();                          // pop Steven out
  cout << S.top() << endl;          // should be Sidhant now
  S.remove("Ranald");               // delete Ranald from the stack
  S.pop();                          // pop Sidhant out
  cout << S.top() << endl;          // should be Tapas now, instead of Ranald
  S.remove("Tapas");                // should be same as pop
  cout << S.top() << endl;          // should be JunAn now
  S.remove("Gary");                 // delete Gary from the stack
  cout << S.top() << endl;          // should still be JunAn
  S.push("CS2040C");                // add CS2040C into the stack
  cout << S.top() << endl;          // should be CS2040C now
  return 0;
}
```

C.2 Matching Rectangles (5 marks)

In a computer simulation of a world, entities can be represented as rectangles as shown in Figure 2. The sizes of the entities can vary from entity to entity. Rectangles are defined in the Cartesian co-ordinate space and are represented by two points corresponding to the lower left co-ordinates (L1) and upper right co-ordinates (R1) (rectangle A in the figure). Information exchange between two entities occurs only when their rectangles overlap each other. **Two rectangles are considered to overlap each other if there exists at least one point that lies inside both rectangles, i.e. the area of intersection is non-zero.** In Figure 2 , A and B overlap, so information is transmitted between them. C and D, however, do not overlap, so no exchange of information occurs between them.

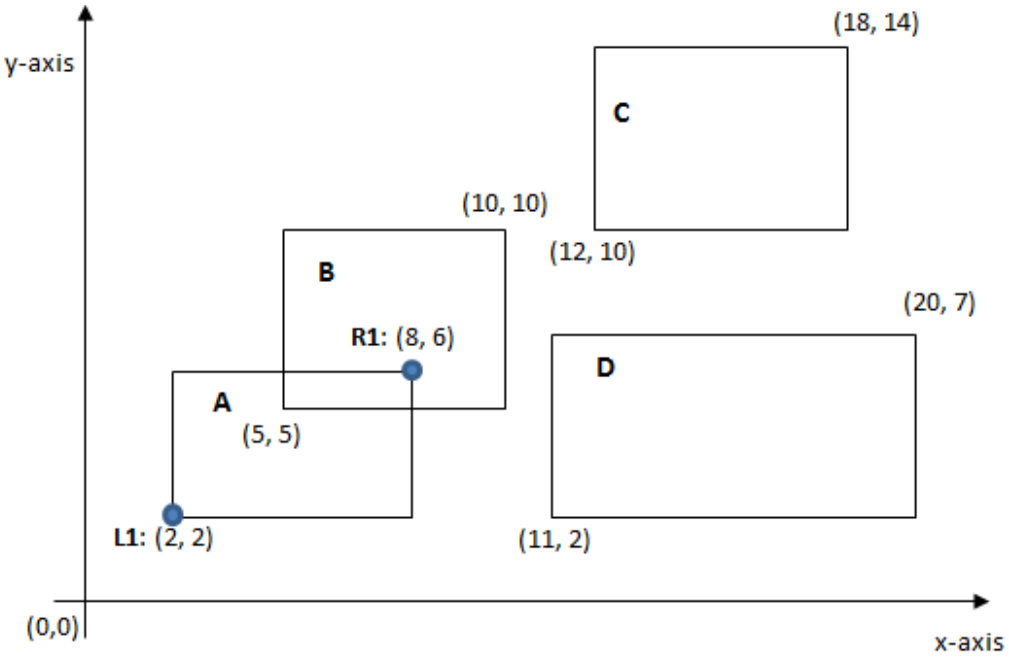

Figure 2: Four rectangles wth A and B overlapping

Your job is to read in N ($2 \leq N \leq 2\,000$) rectangles and determine how many overlaps there are in this world. If A overlaps with B as well as C, it is counted as two overlaps. The input is given as an integer representing N in the first line, followed by data for N rectangles, one line per rectangle. The rectangle data is represented by four integers, the first two representing the co-ordinates for the lower left point, and the remaining two representing the co-ordinates of the upper right point. The sample input given below shows the co-ordinates for the four rectangles in Figure 2: Rectangle A is {(2,2), (8,6)}, Rectangle B is {(5,5), (10,10)}, Rectangle C is {(12,10), (18,14)} and Rectangle D is {(11,2), (20,7)}.

Sample Input	→	Sample Output
4 2 2 8 6 5 5 10 10 12 10 18 14 11 2 20 7		Number of Overlaps: 1

This sample input tallies with Figure 2.

Below is a simple C++ code that is trying to compute the number of overlaps among the rectangles. The code is complete except for a function `overlap` that should determine whether two rectangles overlap. It should return **True** if the two rectangles overlap and **False** otherwise.
Your task is to complete the `overlap` function using C++ code. The function should run in $O(1)$ time. You can write your answer in pseudo-code for slightly lesser marks.

```
#include <bits/stdc++.h>
using namespace std;
struct Point {
  int x, y; // x and y coordinates of a point
};
struct Rectangle {
  Point l, r; //Bottom-left of Rectangle, Top-right of Rectangle
};
// overlap should return True if rectangles A and B (passed as parameters) overlap,
// otherwise, it should return False.
bool overlap (Rectangle A, Rectangle B) {
  // PS: A.r.x is the x-coordinate of rectangle A's top-right point









}
int main() {
  int N, count = 0;
  Rectangle rect[2000];
  cin >> N;   //N is the number of rectangles
  for (int i = 0; i < N; i++) {
    cin >> rect[i].l.x >> rect[i].l.y >> rect[i].r.x >> rect[i].r.y ;
    for (int k = 0; k < i; k++) {
      if (overlap(rect[k], rect[i]))  count++;
    }
  }
  cout << "Number of Overlaps: " << count << endl;
}
```

C.3 From Matching to Connectivity (20 marks)

In the previous problem (C.2), entities are represented as rectangles. Information exchange between two entities occurs only when their rectangles overlap each other.

In this problem, the N entities (rectangles) will be labelled from 0 to $N-1$, based on their order of input. The entity labelled S would like to transfer some information to another entity with label T. However, the rectangles representing entity S and T might not overlap each other. Hence, for the information from S to be transferred to T, the information might need to be passed through a series of other entities, before being able to reach entity T.

Your task is to find out the shortest amount of time required for the information to be transferred from entity S to entity T. For this problem, we will assume that each information exchange between two overlapping entities will take exactly 3 minutes. If it is not possible to transfer information from entity S to T, you should output `impossible`.

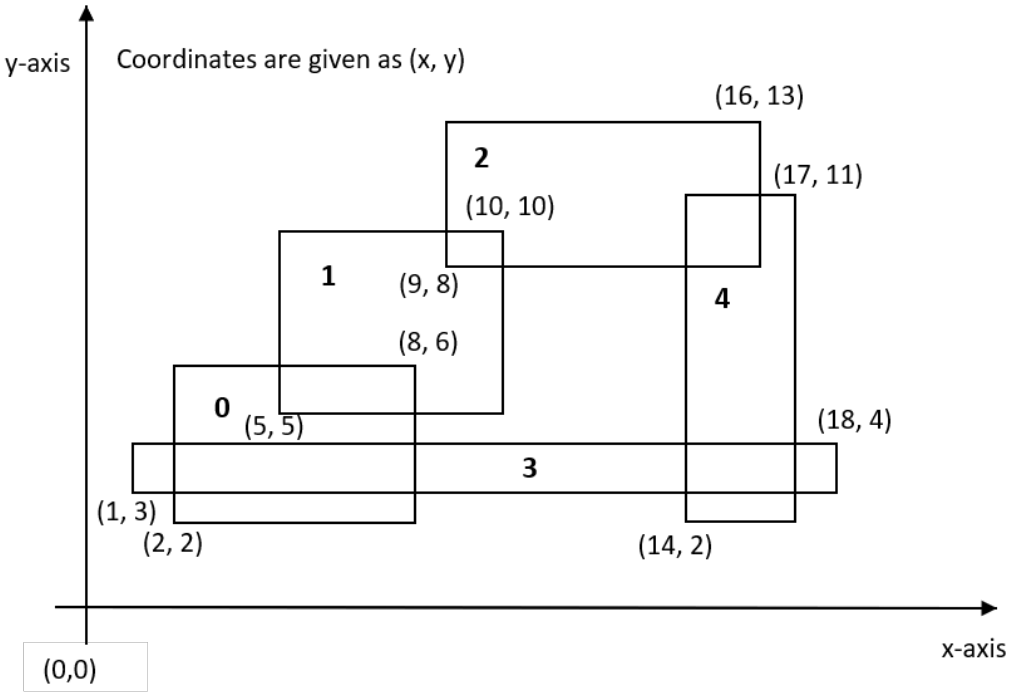

Figure 3: Five rectangles labelled from 0 to 4

In Figure 3, if $S = 1$ and $T = 4$, then the shortest amount of time needed to transmit information from entity 1 to entity 4 is 6 minutes: transferring information from entity 1 to entity 2 takes 3 minutes, and subsequently, transferring from entity 2 to entity 4 takes another 3 minutes.

To solve this, the world can be modelled as a Graph by letting each entity be one vertex in the graph. An **undirected** edge of **weight 3** exists between 2 vertices (entities) if the rectangles representing the entities overlap. Figure 4 below shows how the rectangles in Figure 3 are modelled as a graph.

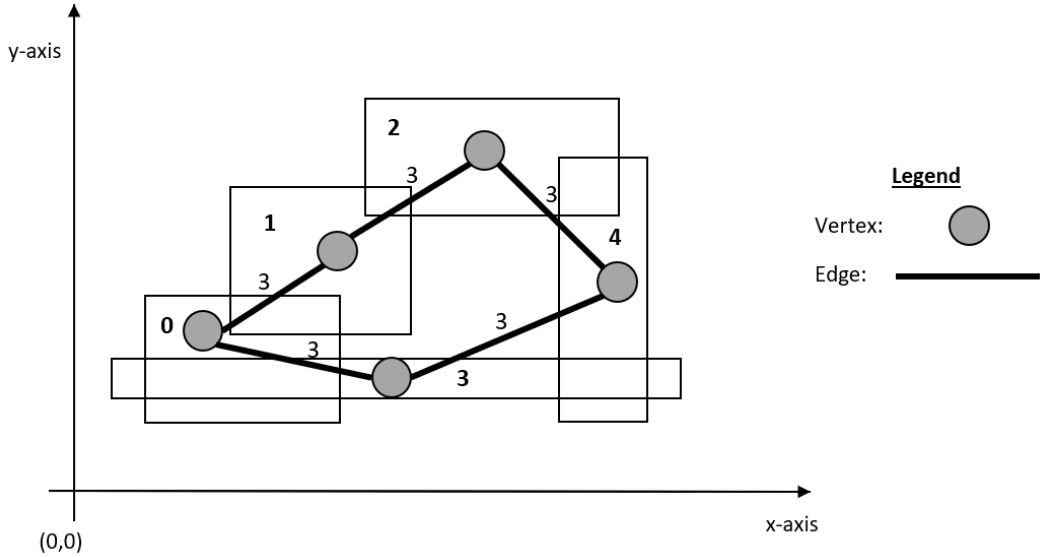

Figure 4: Graph modelled using entities in Figure 3

1. If the input contains N entities (rectangles), how many **vertices** will there be in the graph formed using this modelling? (1 mark)

2. If the input contains N entities (rectangles), what is the **maximum** number of **edges** in the graph formed using this modelling? (1 mark)

3. Which of the following would you choose to store the graph formed using this modelling: **Edge List**, **Adjacency Matrix** or **Adjacency List**? Explain your answer. (2 marks)

4. The problem we are trying to solve is similar to a graph problem that has been covered in CS2040C known as ... ? (1 mark)

You are tasked to write the **most efficient** C++ code that solves the problem described. It should input the N entities, S, T and output the shortest amount of time required for the information to be transferred from entity S to entity T. If it is not possible to transfer information from entity S to T, you should output `impossible` instead.

The input to this program is similar to problem C.2, with an additional line containing S and T. It is guaranteed that S and T are different valid labels and $2 \leq N \leq 2\,000$.

Sample Input 1	→	Sample Output 1		Sample Input 2	→	Sample Output 2
5 2 2 8 6 5 5 10 10 9 8 16 13 1 3 18 4 14 2 17 11 1 4		6		4 2 2 8 6 5 5 10 10 12 10 18 14 11 2 20 7 1 3		impossible

Sample Input 1 tallies with Figure 3 while Sample Input 2 tallies with Figure 2.

Write the C++ code that solves the problem in the space below. You may assume that the `overlap` function is **correctly implemented**. The code for reading input is already provided. (15 marks)

```
#include <bits/stdc++.h>
using namespace std;
struct Point {
  int x, y; // x and y coordinates of a point
};
struct Rectangle {
  Point l, r; //Bottom left of Rectangle, Top right of Rectangle
};
// overlap returns True if rectangles A and B overlap. False otherwise
bool overlap (Rectangle A, Rectangle B) {
  // Assume that it is already correctly implemented
   ...
}
```

```
int main() {
  int N, S, T;
  Rectangle rect[2000];
  cin >> N;   //N is the number of rectangles
  for (int i = 0; i < N; i++) {
    cin >> rect[i].l.x >> rect[i].l.y >> rect[i].r.x >> rect[i].r.y ;
  }
  cin >> S >> T; //Information needs to be transferred from entity S to entity T.









}
```

The time complexity of your algorithm is: O(_____).

C.4 Non-overlapping Rectangles (5 marks)

It is claimed that the answer to problem C.3 will always be 'impossible' if the answer to problem C.2 is 0. (i.e. there are no overlaps among the N rectangles.)

1. Is this claim true? (1 mark)

2. Provide **an outline** (in words) of an $O(N \log N)$ algorithm that can determine if there are any overlaps among the N rectangles. The algorithm just needs to print **'Yes' if there are overlaps** and **'No' if there are no overlaps**. You do not need to describe in detail how to implement the algorithm. (4 marks)

– END OF PAPER. ALL THE BEST –

Section C Marks = ____ +____ +____ +____ = _____