CS 310 'Algorithms' **Assignment-2**
Due date: Thursday, October 31, 2019 at 11:55 PM. [Total Marks = 58]

---

**IMPORTANT:** Read the instructions at the end of this document and follow the naming conventions.

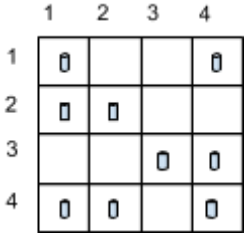
---

**Q1.** **[12 marks]**

You are given an `nxn` square game board and positive integers $row_1$, $row_2$, … $row_n$ and $col_1$, $col_2$, .. $col_n$. You have to place the game pieces in the squares in such a way that the number of game pieces in $i^{th}$ row is equal to $row_i$ and the number of game pieces in the $i^{th}$ column is equal to $col_i$.

For example, let n=4. You are given as input a blank 4x4 board and the following integers:
$row_1$=2, $row_2$=2, $row_3$=2, $row_4$=3
$col_1$=3, $col_2$=2, $col_3$=1, $col_4$=3

One possible arrangement of game pieces on the board, satisfying the above row and column values, is as follows:




1. Code an **efficient greedy** algorithm in C/C++ that solves the above problem. Your code will take as input the value of 'n' and the row and column values. See the input format below. [6+2]
2. Your algorithm should output the (row, column) indices where game pieces are placed. See format below. If it is not possible to satisfy the row and column values given in the input, then your algorithm should output "Not Possible". [2]
3. Give a clear description of your algorithm and data structures used to implement it in the comments at the beginning of your code. **What is the running time of your algorithm?** Your should include clear comments that explain your algorithm's time complexity. [2]
4. You should create at least three different input files to test your algorithm for different scenarios (see format below). Creating test cases helps you think about the problem and different corner cases. For testing your code, you may share your test cases with other classmates. Your code should be able to scale up to larger input sizes.

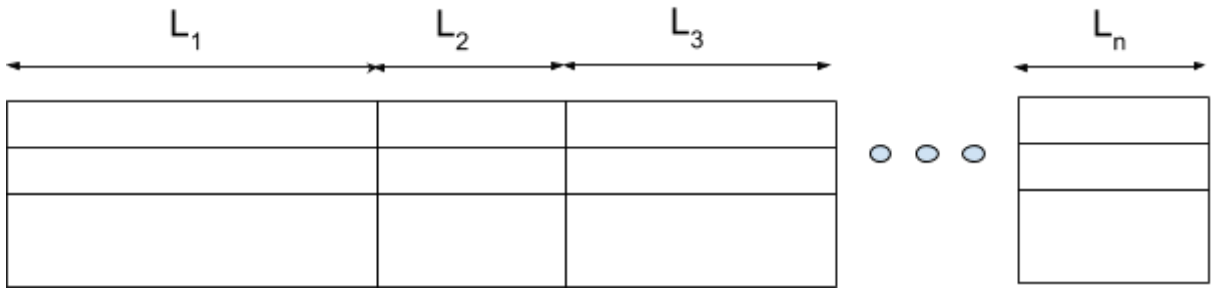
**Input :**
n 4
Row 2 2 2 3
Col 3 2 1 3

**Output :**

(1,1) (1,4)
(2,1) (2,2)
(3,3) (3,4)
(4,1) (4,2) (4,4)

**Q2.** **[12 marks]**

Consider a large warehouse where the inventory items for storage are stacked in 'n' racks. Assume all the racks are placed in a straight line as shown in the figure below. The racks are of different lengths $L_1$, $L_2$, .. $L_n$.




One type/category of items are placed together on one rack. Depending on the sales, different items require replenishment more often than others. Each of the racks have replenishment probabilities associated with them, which are denoted by $p_1$, $p_2$, … $p_n$.

A robot stands on the left of the first rack and when it receives a replenishment request, it goes to the appropriate rack and fills it up. We want to come up with an ordering of the racks, along a straight line, such that the **expected** replenishment time is **minimized**. That is, for all the 'n' racks, you have to decide which rack should be placed where, starting from the first position on the left to last on the right.

For example, suppose n=3 and $rack_3$ is placed first, followed by $rack_1$ and lastly $rack_2$. The expected replenishment time is given by: $p_3 \times L_3 + p_1 (L_3+L_1) + p_2 (L_3+L_1+L_2)$.

If we change the ordering in the above example to $rack_2$, $rack_3$, $rack_1$, then the expected replenishment time will be $p_2 \times L_2 + p_3 (L_2+L_3) + p_1 (L_2+L_3+L_1)$.

If the ordering is denoted by $O_1$, $O_2$, … $O_n$, then the general formula for expected time is given as follows:

$$\sum_{x=1}^{x=n} p_{Ox} \left( \sum_{y=1}^{x} L_{Oy} \right)$$

You have to come up with a **greedy** strategy for rack ordering that minimizes the expected replenishment time.

1. Code an **efficient greedy** algorithm in C/C++ that solves the above problem. Your code will take as input the value of 'n' and the lengths of the racks and their replenishment probabilities. Your algorithm should output the ordering of the racks as well as the expected time. See input and output formats below. [8+2]

2. Give a clear description of your algorithm and data structures used to implement it in the comments at the beginning of your code. **What is the running time of your algorithm?** Your should include clear comments that explain your algorithm's time complexity. [2]
3. You should create at least three different input files to test your algorithm for different scenarios (see format below). Creating test cases helps you think about the problem and different corner cases. For testing your code, you may share your test cases with other classmates. Your code should be able to scale up to larger input sizes.

**Input :**
n 4
L 12 2 8 3
p 0.3 0.2 0.4 0.1

**Output :**
2 3 4 1
Expected Time 13.2

**Q3a. [10 marks]**

Consider a railway network that connects a large number of cities across the country. A big oil company is laying pipelines along a rail network. The rail network has 'n' junctions and the oil company has set up booster stations at each junction. They want to **minimize** the total length of the pipes they have to lay such that there is a path for oil between every pair of junctions. The engineers at the oil company have constructed an undirected, weighted graph G(V,E) with 'n' vertices (i.e., junctions) and 'm' railway tracks. To minimize the total pipe lengths along the rail tracks they have constructed a minimum spanning tree 'T' of G and installed pipes along the edges of T.

One day, due to an earthquake, one of the railway tracks and also the pipe alongside it is permanently damaged. The engineers have to quickly construct a new minimum spanning tree $T_2$ after removing the damaged rail track from the graph G. You have to help the engineers. **Note: if you say rerun a minimum spanning tree (MST) algorithm on the rail network, you will not get any marks.**

A. Code an **efficient** algorithm in C/C++ to solve the above problem. The input to your program is a weighted undirected graph G (format is similar to one described in Assignment-1, except that the graph is weighted. See example below.). You will find T of G using any of the MST algorithms. Now delete an arbitrary edge 'e' from T and find the new MST ($T_2$) without re-running the MST algorithm. Your program should output the MST 'T', the edge 'e' you have removed from T, and the new MST $T_2$. You should also print the sum of all edge weights of the two MSTs. For outputting the MST, it is sufficient if you list all its edges. Note: it is possible that a new spanning tree $T_2$, including all the vertices of G, cannot be constructed after deletion of 'e' (think about why). In that case, simply print that $T_2$ cannot be constructed. [4+4]
B. Give a clear description of your algorithm and data structures used to implement it in the comments at the beginning of your code. **What is the running time of your**

**algorithm?** Your should include clear comments that explain your algorithm's time complexity. [2]

C. You should create at least three different input files to test your algorithm for different scenarios (see format below). Creating test cases helps you think about the problem and different corner cases. For testing your code, you may share your test cases with other classmates. Your code should be able to scale up to larger input sizes.

An example of a graph G is below.
n 5
0: 1;3 2;4
1: 0;3 3;5 4;2
2: 0;4 3;4
3: 1;5 2;4 4;2
4: 1;2 3;2

Output:
MST1: (0,1) (0,2) (1,4) (4,3)
Sum MST1: 11
Edge Removed: (1,4)
MST2: (0,1) (0,2) (2,3) (4,3)
Sum MST2: 13

**Q3b. [10 marks]**

This is a continuation of Q3a. You have the same initial rail network G(V,E) described in Q3a. This time there is no earthquake but the railway company has added a new railway track between two of the junctions (there was no direct track between these junctions previously). The railway engineers want to know what the new MST $T_2$ looks like after the addition of this new link. **As in Q3a, you are not allowed to rerun the MST algorithm to find $T_2$.**

A. Code an **efficient** algorithm in C/C++ to to find $T_2$. The input to your program is the same as in Q3a. Add a new link to G between two vertices that did not have an edge previously. Your program should output the original MST 'T' (reuse code from Q3a), the edge 'e' you have added to G and the new MST $T_2$. You should also print the sum of all edge weights of the two MSTs. For outputting the MST, it is sufficient if you list all its edges. [4+4]

B. Give a clear description of your algorithm and data structures used to implement it in the comments at the beginning of your code. **What is the running time of your algorithm?** Your should include clear comments that explain your algorithm's time and space complexity. [2]

C. Reuse the input files you created in Q3a to test your algorithm for different scenarios.

Output format for the above question is the same as Q3a.

**Q4. [14 marks]**

You are standing in a hallway that has 'n' rooms. Each of the rooms has a double door. To enter the room you can open either the right-side of the double door or the left-side or both

sides. Each of the sides of the double doors are labeled. See Figure-3 below, which shows three rooms in the hallway. The same label (for example da2) can appear on more than one door.

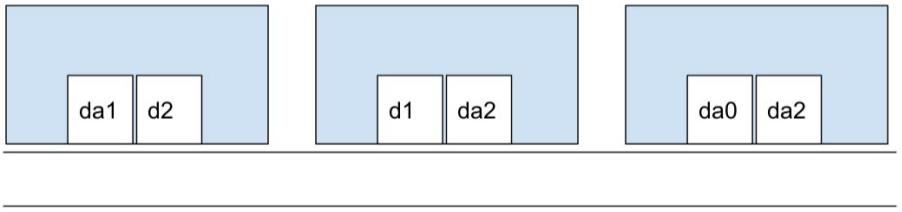
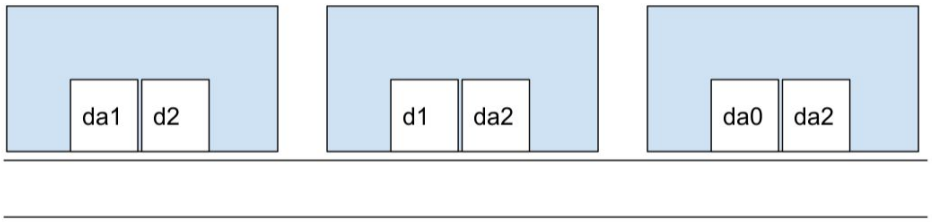



Figure-3

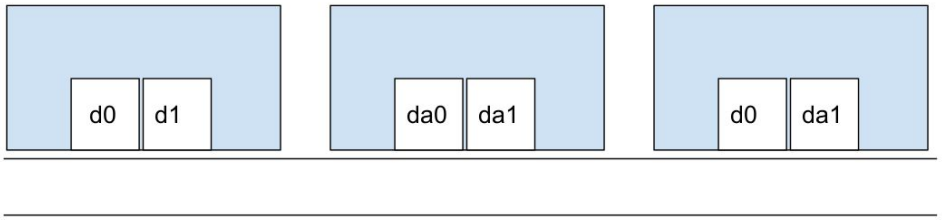
The architect, who designed the rooms loved puzzles, so he created what we'll call an "anti" door. In Figure-3, we denote the "anti" door by an extra symbol 'a'. If a door (say) d1 is opened then its anti-door (denoted by da1) will automatically lock and da1 cannot be opened until d1 is closed. Similarly, if da1 is opened, then d1 will close. Note that a door label can appear more than once, so if you open (say) da2 in one room then all door-sides with the same label in *other* rooms will also open automatically.

We want to find out if it's possible to open ***all*** rooms ***at the same time***. Note that opening a room means that *at least* one side of the double-door is open. See some examples below:



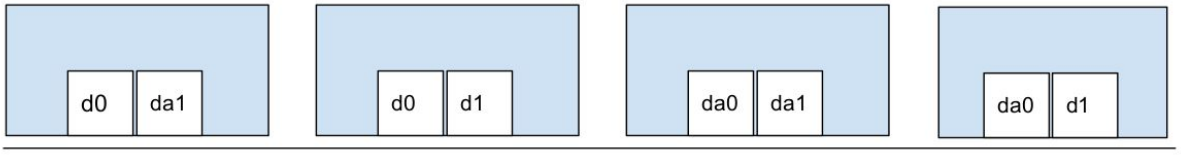

Example-1

In Example-1, we can open d2, d1 and da0, and hence all three rooms will be open at the same time.




Example-2

In Example-2, we can open d0 and da1, and hence all three rooms will be open at the same time.




Example-3

In Example-3, it is **not** possible to open all four rooms at the same time.

1. Code an **efficient** algorithm in C/C++ that determines if it's possible to open all 'n' doors at the same time. If yes, then you should identify which door-sides should be opened. See the given output format. [10+2]
2. Give a clear description of your algorithm and data-structures used to implement it in the comments at the beginning of your code. **What is the running time of your algorithm?** You should include clear comments that explain your algorithm's time complexity. [2]
3. You should create at least three different input files to test your algorithm for different scenarios (see format below). Creating test cases helps you think about the problem and different corner cases. For testing your code, you may share your test cases with other classmates. Your code should be able to scale up to larger input sizes.

As you can see in the examples above, integer values are used for the door labels starting from 0. E.g. in Example-1, we have three labels: 0, 1 and 2 to identify the doors and the corresponding anti-doors. In Example-2 we have two labels: 0 and 1. Let 'k' be the number of door labels.

Your code will read input from a text file. The file will have the value of n on the first line. The second line will have value of 'k'. This will be followed by n lines (one for each of the n rooms) and each line will contain the labels of its two doors. For example, Example-1 is represented as below.

**Input (for Example-1):**

```
n 3
k 3
da1 d2
d1 da2
da0 da2
```

**Output (for Example-1):**

```
Yes
0
1
1
```

**NOTE:** The above output prints values assigned to doors in ***increasing order of their labels***. (i.e., the values you have assigned to d0, d1 and d2 respectively, each on a separate line). A value of 0 to d0 means that d0 is closed (and hence da0 is open). A value of 1 to d1 means door d1 is open and similarly value of 1 to d2 means door d2 is open.

**Output (for Example-3):**

```
No
```

**Instructions and policies**

1. When submitting, please ***rename the folder*** according to your ***roll number***.
2. Do ***delete all executables*** and ***test files*** before submitting your assignment on LMS.
3. Folder convention ***should not be changed.*** If you make any changes, the auto grader will ***grade your assignment 0***.
4. You must submit your ***own*** work. You may discuss the problems with other classmates but must not reveal the solution to others or copy someone's work. Remember to acknowledge other classmates if discussions with them has helped you.
5. You should name your code files using the following convention: qx.cpp
6. If the assignment includes any theoretical questions, then type your answer to those questions and submit a separate pdf file for each using the naming convention above.
7. Upload all your files in the corresponding assignment folder on LMS. **There will be a 20% deduction for assignments submitted up to one day late (the late deduction is only applicable to the questions submitted late, not on the whole assignment). Assignments submitted 24 hours after the deadline will not be marked.**
8. There will be vivas during grading of the assignment. The TAs will announce a schedule and ask you to sign up for viva time slots. Failure to show up for vivas will result in a **70% marks reduction** in the assignment.
9. In the questions where you are asked to create test cases. Think carefully about good test cases that check different conditions and corner cases. The examples given in the assignment are for clarity and illustration purposes. You should not assume that those are the only test cases your code should work for. Your code should be able to scale up to larger input sizes and more complex scenarios.
10. Do not make arbitrary assumptions about the input or the structure of the problem without clarifying them first with the Instructor or the TAs.
11. **We will use automated code testing so pay close attention to the input and output formats.**
12. **Make sure that your code compiles and runs on Ubuntu. You may choose to develop your code in your favourite OS environment, however, the code must be properly tested on Linux before submission. During vivas, your code should not have any compatibility issues. It's a good idea to use gcc -Wall option flag to generate the compiler's warnings. Pay attention to those warnings as fixing them will lead to a much better and robust code.**
13. For full credit, comment your code clearly and state any assumptions that you have made. There are marks for writing well-structured and efficient code.
14. Familiarize yourself with LUMS policy on plagiarism and the penalties associated with it. **We will use a tool to check for plagiarism in the submissions.**