Advanced Algorithms – Assignment 1

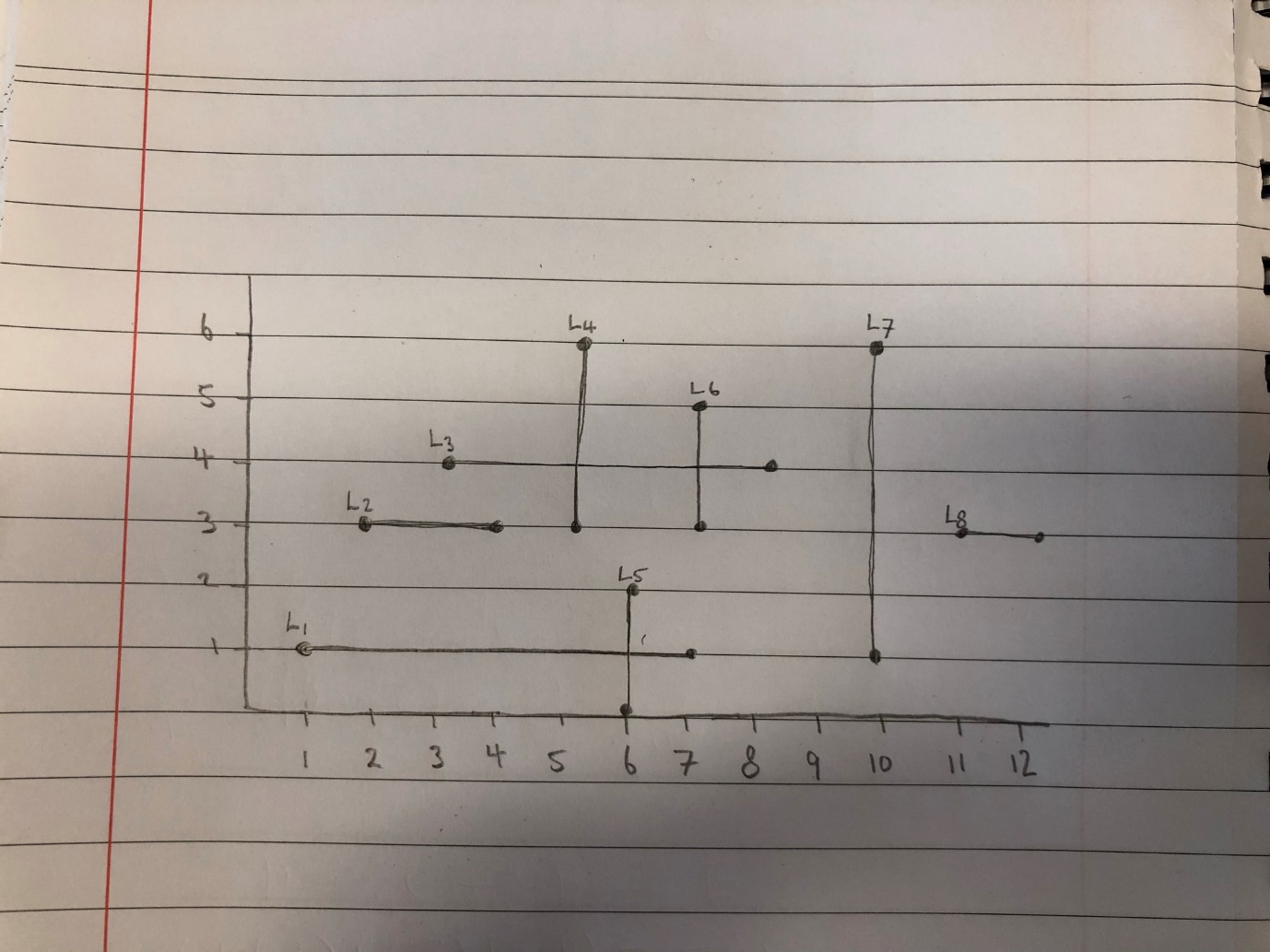
Lucas Geurtjens (s5132841)

# Q1. Sweep Line Algorithm

**Introduction:**

In this problem we created a Sweep Line algorithm in order to solve the Orthogonal Line Intersection problem in O(N log N) complexity.

This is the test input that was used:



**Overview:**

* Orthogonal lines are initialised with their two (x, y) points.
* Initialise a priority queue (to contain the x-position points of each line), and binary search tree (to contain the y-position of horizontal lines).
* Insert the X value of each point the priority queue.
* Sort the priority queue.
* Sweep over the points in the priority queue in ascending order.
  + If the point is vertical
    - Do a range search in the Binary Search Tree within the line’s y-positions.
    - If a line exists within this range, increment the number of intersections.
  + If the point is the first point of the line, insert it into a Binary Search Tree.
  + If the point is the last point of the line, remove it from the Binary Search Tree.
* Display the number of intersections.

**Output:**

After inputting to the algorithm the above lines, it provided the correct output:



**Complexity Analysis:**

…

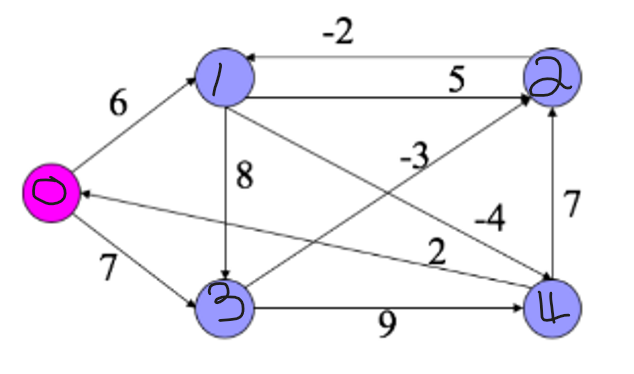
**References:**

* I had help understanding the Orthogonal line intersection problem from a Princeton University lecture on the topic (<https://youtu.be/dePDHVovJlE>).
* I had a huge help building the Binary Search Tree data structure from Paul Programming's video series on BST (<https://www.youtube.com/watch?v=sf_9w653xdE&list=PLTxllHdfUq4d-DE16EDkpeb8Z68DU7Z_Q>)
* I had help with the range search from Persistent Programmer's video on the topic (<https://www.youtube.com/watch?v=iQiEMgbptwQ>)

# Q2. Bellman-Ford Algorithm

**Introduction:**

A Bellman-Ford algorithm was created to solve the provided graph (as labelled by myself):

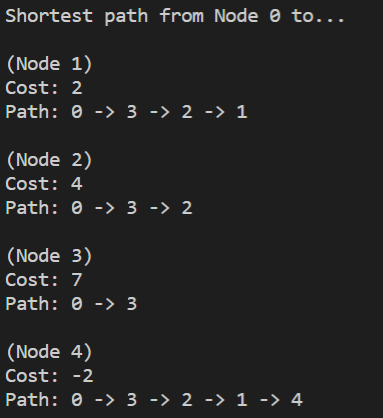


**Overview:**

* Edges are initialised with an input *from*, *to*, and *weight* value.
* A *best distance* array is initialised with each node having a cost of infinity (and parent node of -1).
* Set our starting node to a cost of 0 (and parent of itself).
* For number of nodes – 1 iterations:
* Iterate for the number of nodes -1 times:
  + For each node:
    - And each edge connected to the node:
      * Try to relax the cost of travelling to the node across the edge.
* Repeat the above iterations, except if a cost of travelling is able to be relaxed, set it to -infinity. (This is a negative cycle).
* Display the shortest paths.

**Output:**

After inputting to the algorithm the above graph, it provided the correct output:

****

# Q3. Graph Data Structure

# Q4. Unordered List Data Structure

**Introduction:**

A data structure was created for storing unordered lists of integers. It needed to be O(1) for adding, deleting, testing for being in the list, and displaying an integer when iterating through the list. It also needed to be O(k) (where k is the number of integers in the list) for clearing the list. Duplicates were not allowed in the list, and the integers had to be stored in the range 0 to N (where N was an upper bound value).

**Overview / Complexity Analysis**:

Data structure

The structure I used included a list array of the values, and a lookup map array that stored the index position of each value in the list. The lookup table was of size N + 1 and is initialised to all -1s. This example illustrates how it works:

* N=6
* list = {1, 3, 5}
* lookup = [ -1 , 0 , -1 , 1 , -1 , 2, -1 ].

So, in lookup [3] we get 1, which is 3’s index in the list.

Add

* We check if the value is within the range 0 – N. **O (1)**
* We check if it’s not a duplicate. **O (1)**
* Add the value to the head of the list. **O (1)**
* Add the value into the lookup map. (E.g., lookup[value] = list index we just inserted the value into). **O (1)**

Therefore, adding is **O (1)**

Remove a value

* Check if the value exists. **O (1)**
* Swap the value to delete with the last value of the list in the lookup map. **O (1)**
* Set the element we’re deleting to the last element in the list. **O (1)**
* Update the index of the value we swapped to in the lookup map. **O (1)**
* Set the index we deleted to -1 in the lookup map. **O (1)**
* Decrease our list size by 1. **O (1)**

An example removing 2 from list = { 2, 3, 5} lookup={ -1, -1, 0, 1, -1, 3 }:

1. lookup = { -1, -1, 3, 1, -1, 0 }
2. list = { 5, 3, 5}
3. lookup = { -1, -1, 0, 1, -1, -1 }
4. list = {2, 3}

Therefore, removing a value is **O (1)**

Exists (test if in the list)

* Plug the value into the lookup map and check if it’s not -1. **O (1)**

Therefore, testing if a value exists in the list is **O (1)**

Iterate over list

* Loop over each item in the list **O (K)** (which is **O (1)** per iteration which I’m assuming is what the O (1) in the question was referring to).

Therefore, to iterate over each element in the list is **O (1)** per iteration.

Clear (empty list)

* Set the list values in our lookup map to -1 **O (K)**
* Remove all items from the array **O (K)**

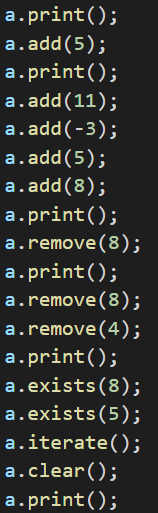
Therefore, to clear the list is **O (K)**

**Output:**

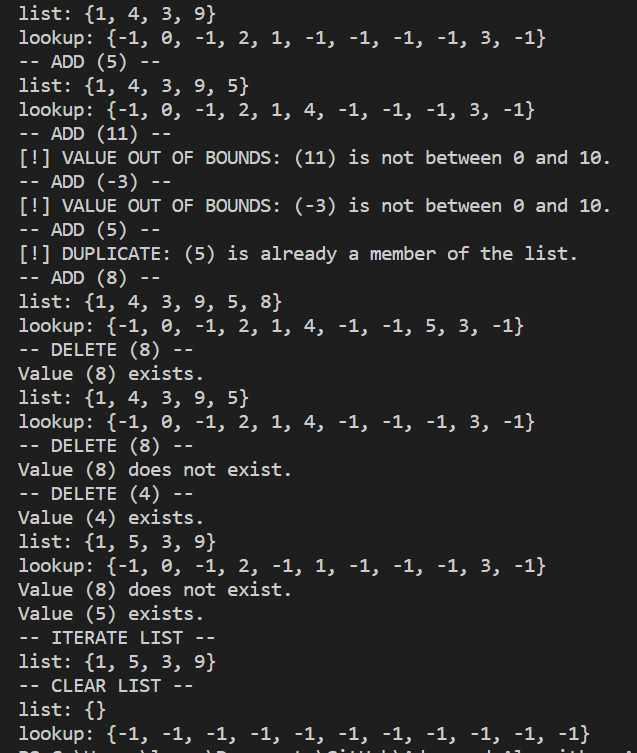
Inputting values = {1, 4, 3, 9} N = 10



The following commands were run:



Resulting in the following output, which executed successfully:



# Q5. Ladder-gram Algorithm

**Introduction:**

**Overview:**

Mention it can do 3 words easy, 4 words easy, but 5 words real hard.

**Output:**

Examples of words it can get to and from.

**References:**

* I had help understanding Ladder-gram algorithm from Tech Dose’s video on the topic (<https://www.youtube.com/watch?v=ZVJ3asMoZ18>).