Advanced Algorithms – Assignment 1

Lucas Geurtjens (s5132841)

# Q1. Sweep Line Algorithm

**Introduction:**

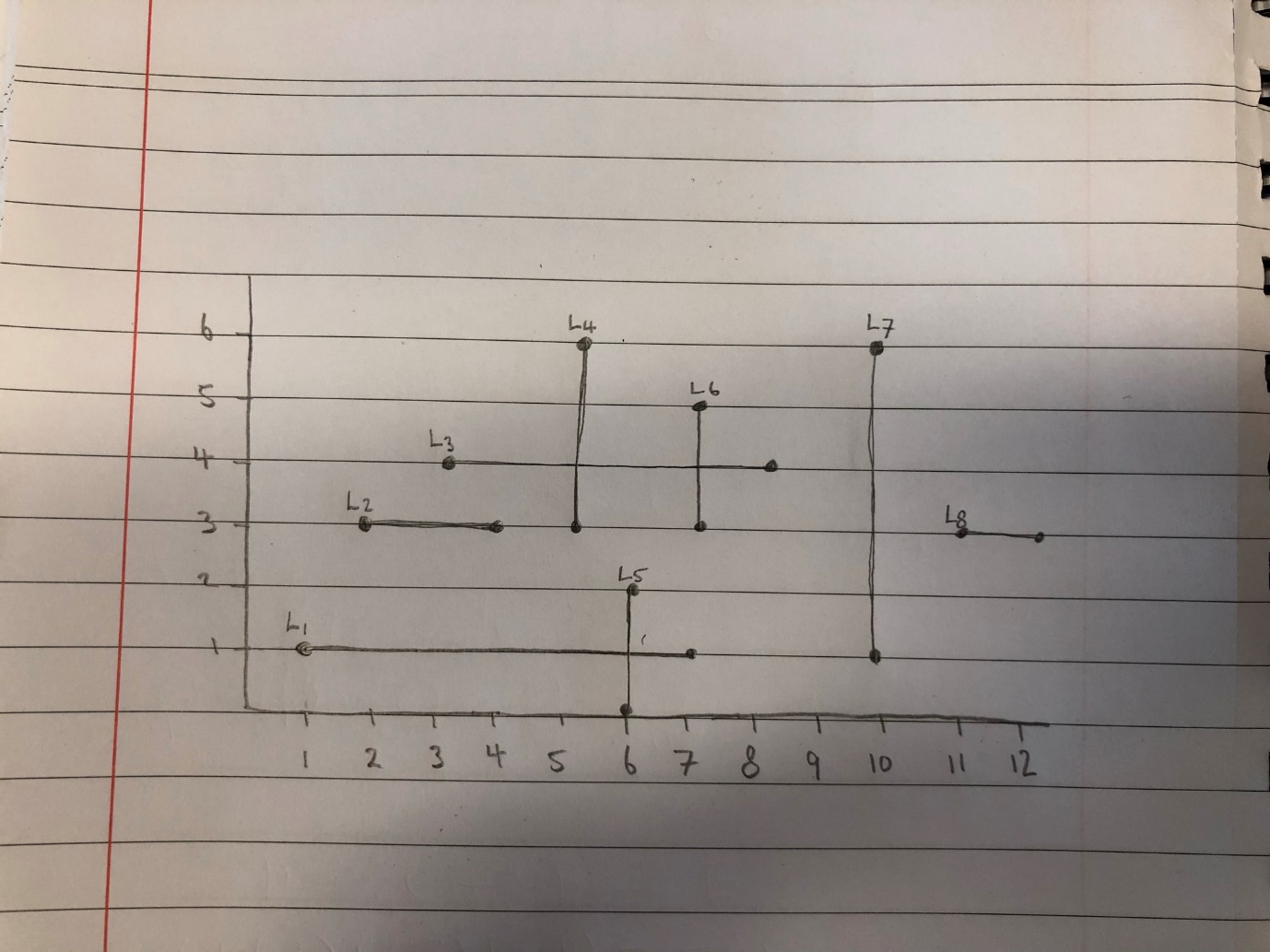
In this problem we created a Sweep Line algorithm in order to solve the Orthogonal Line Intersection problem in O (N log N) complexity.

**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.

**Input/Output:**

This is the test input that was used:



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



**Complexity Analysis:**

* Initialising priority queue with points: **O (N)** (where N is the number of lines).
* Stable sort of the priority queue: **O (N log N)** (NOTE: This sorting algorithm has been used from the <algorithms> c++ library. It’s referenced as being O (N log N) (see: <https://www.cplusplus.com/reference/algorithm/stable_sort/>).
* Sweeping over the priority queue: **O (N)**
  + BST (Binary Search Tree) insert: **O (log N)**
    - If empty set as root pointer: **O (1)**
    - Choose which side of the node the value should belong
      * If a node already exists in that position, run the algorithm with this as the root. This will run recursively, and assuming we have a balanced tree will run **O (log N)** where N is the height of the tree.
      * Otherwise, create a node in that position. **O (1)**
  + BST remove: **O (log N)**
    - Case 1: removing node with no children
      * Set this node to NULL: **O (1)**
    - Case 2: removing node with 1 child
      * Swap the child node with its parent: **O (1)**
      * Delete the child node: **O (1)**
    - Case 3: removing node with 2 children
      * Find the smallest node from its right child (check the right node and all it’s left most children until you reach the end of the children). Assuming we have a balanced tree, this will be **O (log N)** where N is the height of the tree from that node.
      * Swap the found node with the node we are deleting: **O (1)**
      * Delete the found node: **O (1)**
  + BST range search: **O (log N)**
    - Check if the current node exists (initially the root): **O (1)**
    - If the node’s value is within the range increment the number of intersections: **O (1)**
    - If the value is still within the left bounds, keep searching its left nodes recursively. Assuming it’s a balanced tree, this will take **O (log N)**
    - If the value is still within the right bounds, keep searching its right nodes recursively. Assuming it’s a balanced tree, this will take **O (log N)**

Therefore, the priority queue initialisation **O (N)** + priority queue stable sort + **O (N log N)** + Binary Search Tree Sweeping (assuming a balanced tree) **O (N log N)** gives a total time complexity of **O (N log N)**

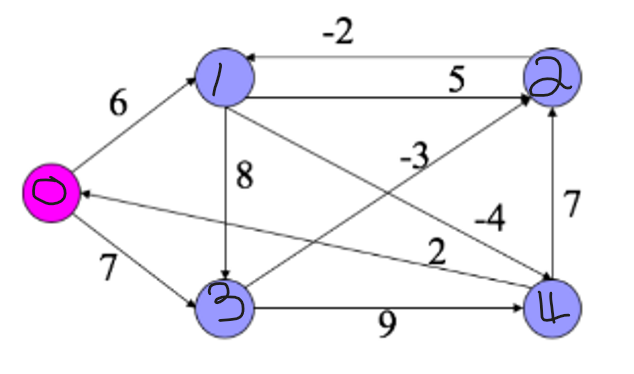
**References:**

* 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):

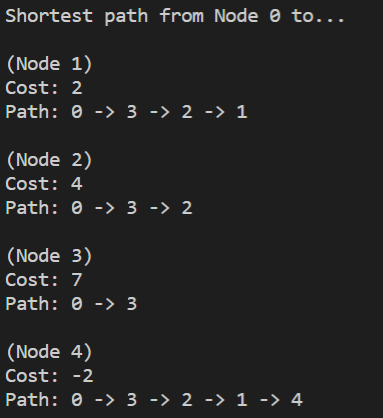


**Overview:**

* 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.

**Input/Output:**

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

****

# Q3. Graph Data Structure

**Introduction:**

A data structure for storing large graphs was attempted. It was able to check if two vertices are connected, and produce a list of all vertices connected to a given vertex. It is made as a variant of an adjacency list (which takes O(E) space complexity, where E is the number of edges, compared to an adjacency matrix approach which takes O(E2)).

**Overview:**

* Initialise a 2D array, storing the node number and a list of nodes it’s connected to.

Check if two nodes are connected

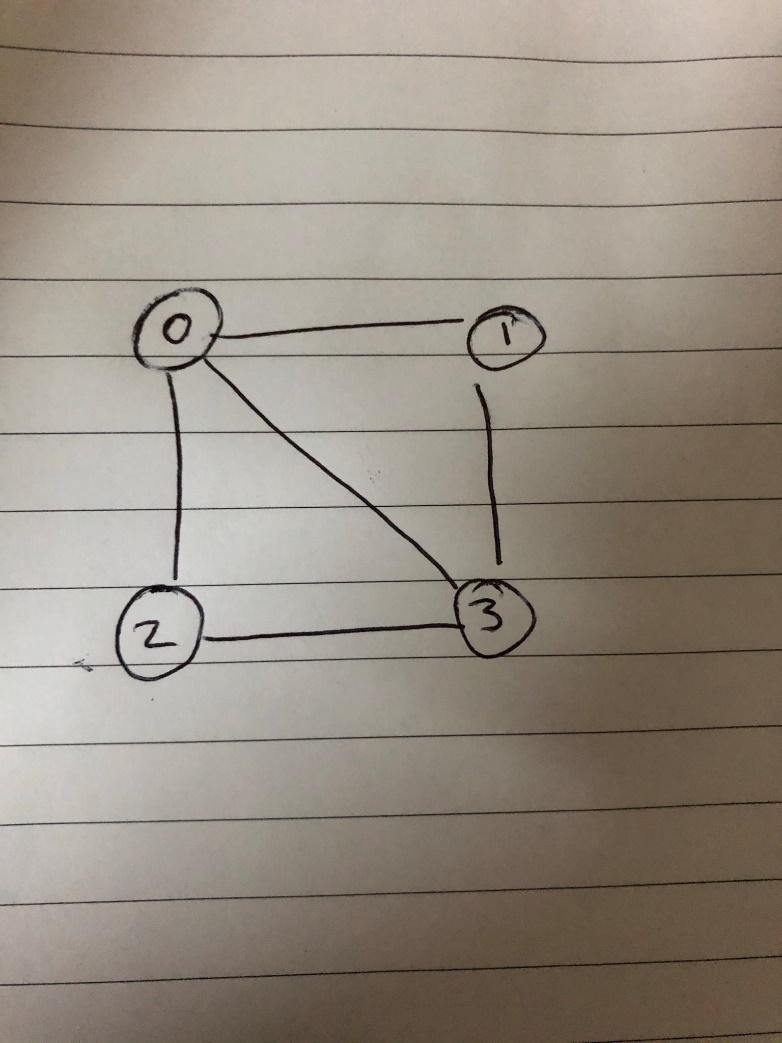
* Loop through the list of the node’s connected nodes
  + If it’s found in the connected nodes, display that it exists
  + Otherwise, display it doesn’t exist.

Check which nodes are connected to a given node

* Loop through the node’s connected nodes
  + Display each node

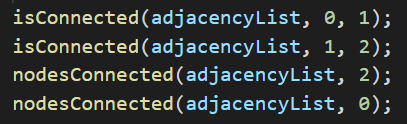
**Input/Output:**

This is a simple input we’re using as a graph:

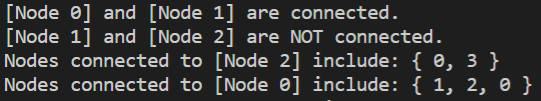


Which is represented as: {{1, 2, 0}, {0, 3}, {0, 3}, {1, 2, 0}} where the index of the list corresponds to the node value (e.g., index 0 corresponds to the edges connected to node 0).

We can execute our methods like so:



Which gives the correct output:



# 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)**

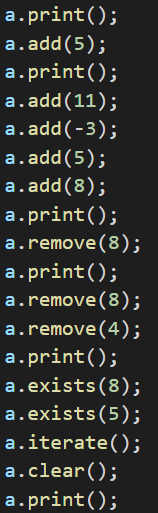
Therefore, to clear the list is **O (K)**

**Input/Output:**

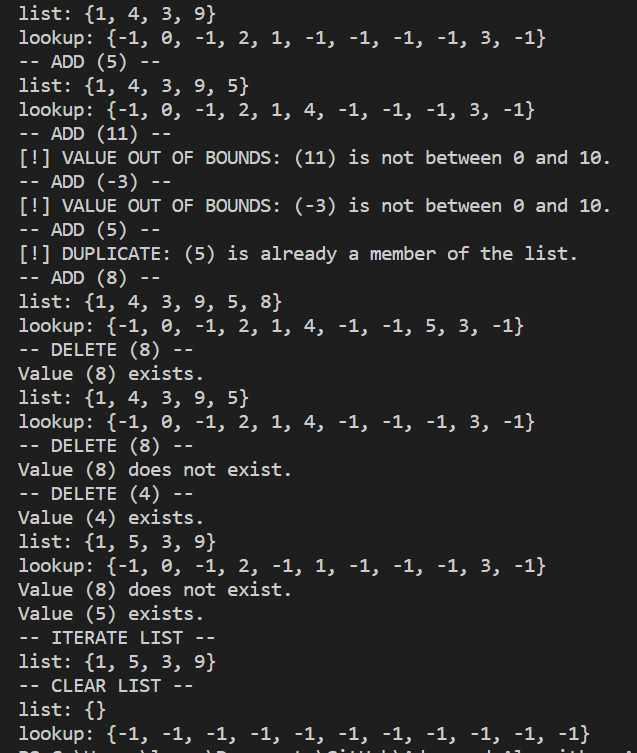
Inputting values = {1, 4, 3, 9} N = 10



The following commands were run:



Resulting in the following output, which executed successfully:



# Q5. Ladder-gram Algorithm

**Introduction:**

An algorithm was created to solve the ladder-gram problem. Using a dictionary of words, a start and end word (of equal length) are chosen. The algorithm then tries to form the end word by making character changes that create another word in the dictionary.

**Overview:**

* Two words of equal length from the dictionary are input.
* The dictionary is read.
* Words in the dictionary of equal length to the start/end words are stored in a word list.
* We initialise a queue with the start word at its beginning.
* We also initialise how many steps have been taken to change the word (what level of the BFS it’s on).
* And we initialise the number of words to explore on the given level (initially 1) and how many nodes to explore on the next level (initially 0).
* While the queue has words to explore:
  + Pop a word from the queue.
  + Loop through the letters of the word
    - Try a different letter in its place (from a – z not including the original letter)
      * If it makes a valid word:
        + Check if it’s the end node, if so, return the number of steps to get to this word + 1 (because it would be on the next level down).
        + Otherwise, insert the word into the queue, increment our next to explore, and remove it from the word list.
  + Decrement current words to explore
  + If the current words to explore is 0, increase the step by 1 and set the current to explore to the next to explore, resetting the next to explore afterwards.
* If we don’t find the end word, print a message to the user.

**Input/Output:**

Example input and output:

|  |  |  |
| --- | --- | --- |
| **Source word** | **End word** | **Number of steps** |
| umm | sip | 4 |
| abbe | umbo | Cannot reach end word. |
| char | suer | 5 |
| unco | undo | 1 |
| under | uncut | Cannot reach end word. |
| whine | whisk | 5 |
| ripsaw | toping | Cannot reach end word. |
| delays | deject | 11 |
| batfish | baskets | Cannot reach end word. |
| toolbox | topcoat | Cannot reach end word. |
| deformed | deformer | 1 |
| degraded | defusing | Cannot reach end word. |
| simulations | singularity | Cannot reach end word. |

**References:**

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

# Q6. Word Sequence Algorithm

**Introduction:**

This algorithm takes in a dictionary, splits it up into groups of words that are the same size, and then tries to find sequences across the words such that the 2nd and 3rd letter of a current word are equal to the 3rd and 2nd letter of a previous word.

**Overview:**

* The dictionary is read in.
* Then the words are partitioned into lists relating to the size of the words. (Thus, we have lists of words size 4 grouped together, words of size 5 grouped together etc.)
* For each word group…
  + Find the **best sequence** (see *finding the best sequence*)

Finding the best sequence

* Given a word group list
* Initialise a best sequence value set to 0
* Loop through each word in the list, using that word as the starting word of the sequence and try to **find a sequence** (see *finding a sequence*)
  + If it’s better than the previous sequence score, update the score.

Finding a sequence

* Initialise a sequence queue with a starting word as its first index.
* Initialise a sequence count of 0
* Remove this start word from the word list
* Looping until no sequence is found:
  + Set a parent word
  + For each word in the word list:
    - Check if the word’s 2nd character equals the parent word’s 3rd last character and if the words 3rd character equals the parent word’s 2nd last character.
      * If it does, add it to our sequence queue, increment the sequence count by 1, and remove the word from the word list.
  + If there was no sequence found in the previous iteration, return the sequence count.

**Input/Output:**

After inputting the dictionary.txt file, the following output was created:

|  |  |
| --- | --- |
| Word Length | Sequences found |
| 4 | 92 |
| 5 | 50 |
| 6 | 126 |
| 7 | 156 |
| 8 | 77 |
| 9 | 41 |
| 10 | 46 |
| 11 | 24 |
| 12 | 47 |
| 13 | 15 |
| 14 | 11 |
| 15 | 11 |