Callum Johnson, 910519, Tute Fri 4.15pm Tobias

1. **function** evaluate_algorithm(expression)

while there are tokens to be read **do**:

if the token is a number **then**:

push the number to the value stack.

if the token is an open parenthesis '(' **then**:

push the token onto the operator stack.

if the token is a close parenthesis ')' **then**:

if there is more than one token in the operator stack **then:**

while the top element of the operator stack is not '(' **do:**

if the value stack has less than 2 elements **then:**

return NOTWELLFORMED.

else:

pop the top element of the operator stack and the top 2 elements of the value stack

perform the operation on the popped values in the correct order.

push the result onto the value stack

pop the top element of the operator stack and discard it (open parenthesis)

else:

return NOTWELLFORMED

if the token is an operator **then:**

if there are no tokens in the operator stack **then:**

if there are no values in the value stack **then:**

return NOTWELLFORMED

else:

push the operator onto the operator stack

else:

if the top element of the operator stack is an open parenthesis '(' **then:**

push the current token onto the operator stack.

else:

while the top element of the operator stack is higher precedence than the current token **do:**

pop the top element of the operator stack and the top 2 elements of the value stack

perform the operation on the popped values in the correct order

push the result onto the value stack

else:

return NOTWELLFORMED

while there are tokens in the operator stack **do:**

pop the top element of the operator stack and the top 2 elements of the value stack

perform the operation on the popped values in the correct order

push the result onto the value stack

pop and return the top value of the value stack.

Design of Algorithms Assignment 1 – Written questions

Callum Johnson, 910519, Tute Fri 4.15pm Tobias

3. B)

The input stage of the program runs in O(n + m + 1) time. 'parkranger.c' works by first initializing an array of empty deques equal to the number of trees in the problem plus an extra deque to represent the starting position of the run, this process runs in O(n + 1) time. Next the program reads in the viable routes into the deques previously initialized, this process costs O(m) time.

The next stage of the program is the topological sorting stage, which is a dfs algorithm with a worst-case time complexity O(m). This part of the algorithm explores possible paths from the start position 0, marking nodes as it backtracks into a topologically sorted array. This part of the program works by recursively calling 'dfs_search' and keeping track of visited nodes so as not to explore them more than once.

The next stage involves checking to see if the tree trimmer can go from the top of the mountain to the bottom going from one node to the next in the topological ordering given by dfs_sort. Since dfs_sort only explores from node 0, we first check to see if all the nodes were visited in the sort - with costs O(n) time. If they weren't the program returns false. Next the program checks for each node in the topological sort, whether the previous node has an outgoing edge to the current one, this process has a nested for loop but only costs O(n + m) time since nodes are not revisited and also due to the design decision to use an adjacency list instead of matrix.

Finally, freeing the memory associated with running the program is bounded by O(n) time due to freeing each deque in our adjacency list.

Hence the program runs in linear time, since each of its constituent stages run in linear time leading to an overall time complexity of O(n + m) for large values of n and m.