ESO207: Theoretical Assignment 2 - Part 2

Debaditya Bhattacharya

March 22, 2021

Name: Debaditya Bhattacharya
Roll No: 190254
Email id: debbh@iitk.ac.in
Hackerrank Id: debbh922

Problem 4

Description:

This problem is a modified version of the count inversions algorithm that was discussed in class and in programming assignment 2. To solve this we, can simply borrow the same algorithm used earlier(modified merge sort, $T = \mathcal{O}(n \log n)$, and modify our input accordingly. Counting number of elements that are smaller to the right is equivalent to finding the number of elements which are greater and to the left in an new array which is formed by reversing the array, and taking its negative($A[i]$ swapped with $-A[N-1-i]$). This operation can be done in $\mathcal{O}(N)$. Hence the overall time complexity is $T = \mathcal{O}(N) + \mathcal{O}(N \log N) = \mathcal{O}(N \log N)$

Algorithm:

```
#A struct used for holding the index and the value together while sorting
struct data_struct:
    int index
    int value

#Merge and count function (T = O(L-R))
def merge_and_count_strict_inversions(data_structs, L, mid, R, counts):
    p <- L #index for left array
    j <- mid + 1 #index for right array
    copy <- empty array of size R-L+1
    r <- 0 #index for copy array

    #p should be in between L and mid, j should be in between mid and R.
    while p <= mid && j <= R:

        #value[p] <= value[j] -> Not a strict inversion. Copy value at p.
        if (data_structs[p].value <= data_structs[j].value):
            copy[r] <- data_structs[p]
            r++
            p++

        #value[p] < value[j] -> Strict Inversion. Copy value at j. Increment count j by number
     of smaller elements (mid-p+1)
        else:
            copy[r] <- data_structs[j]
            counts[data_structs[j].index] += mid - p + 1
            j++
            r++

    #Copy the rest of values in left subarray
    while p <= mid:
        copy[r] <- data_structs[p]
        r++
        p++

    #Copy the rest of the values in the right subarray
    while j <= L:
        copy[r] <- data_structs[j]
        r++;
        j++;

    #Overwrite data_structs with the temporary copy array
    for x from 0 to R-L:
        data_structs[i+x] <- copy[x]



#Please turn over
```

```
48
49 #Recursive sort and count algorithm T = O(N)
50 def sort_and_count_strict_inversions(data_structs, L, R, counts)
51
52     #Base case
53     if L == R:
54         return
55
56     else:
57         mid <- (L + R) / 2 #dividing the array (for divide and conquer)
58         #sort and count left half T = T(N/2)
59         sort_and_count_strict_inversions(data_structs, L, mid, counts)
60         #sort and count right half T = T(N/2)
61         sort_and_count_strict_inversions(data_structs, mid+1, R, counts)
62         #merge and count cross terms T = O(N)
63         merge_and_count_strict_inversions(data_structs, L, mid, R, counts)
64     return
65
66 #Driver function
67 def count_smaller_right(A):
68     N <- length(A) #Get the length
69     data_structs <- empty array of data_struct #data_struct is a tuple of values and tuples of
       the original array
70
71     #Array modification (T = O(N))
72     for i from 0 to N-1:
73         data_structs[N-1-i].value <- -A[i] #flip order and negate the original array
74         data_structs[i].index <- i
75
76     counts <- array of zeros of size N
77
78     #sort and count strict inversions algorithm (T = O(NlogN))
79     sort_and_count_strict_inversions(data_structs, 0, N - 1, counts)
80
81     output <- empty array of size N
82
83     #Reverse counts to get correct order. T = O(N)
84     for i from 0 to N-1:
85         output[i] <- counts[N-1-i]
86
87     return output
```

Listing 1: Count smaller to right

Time Complexity Analysis:

Count inversions can be implemented via a modified merge sort in $T = \mathcal{O}(N \log N)$ as we have seen in class. Modifying the input array A into a new array B via tha transformation $B[i] = A[N-1-i]$ takes $T = \mathcal{O}(N)$. Transforming the count array to our desired output array takes $T = \mathcal{O}(N)$. The overall time complexity of the algorithm is hence $T = \mathcal{O}(N \log N) + \mathcal{O}(N) + \mathcal{O}(N) = \mathcal{O}(N \log N)$

Problem 5

Description:

To solve this question we note that in a connected graph, if there are no cycles, then the number of edges in the graph is constricted by $V - 1$. Consider our graph to be made of i chunks of connected graphs. To find if a connected graph has a cycle, we can run a breadth / depth first search, and keep a track of which nodes have been visited, and upon finding a cycle terminate the program. If we do not find a cycle, return false.

Algorithm:

```
1     #Traverse via depth first traversal, and check for cycles
2     def traverse_and_check(adj_list, node, visited, parent):
3         stack <- Empty queue #Create an empty queue for bfs
4         stack.push(node) #insert first node
5         current <- NULL #create variable called current.
6
7         while !stack.isEmpty(): #Run until the stack is empty
8             current <- stack.pop() #Pop topmost element in stack.
9             adjacent <- adj_list[current] #Get head of adjacency list for current element
10
11            while adjacent.next != NULL: #Traverse adjacency list
```

```
12
13                  if visited[adjacent.value] == True:       #If adjacent element already visited
14                      if adjacent.value != parent[current]: #If it is not the parent of the
    current element
15                          return True                       #It must be a cycle. Return true
16
17                  else:                                     #If not visited, add to queue, make
    visited = 1, and assign current node as parent.
18                      stack.push(adjacent.value)
19                      parent[adjacent.value] <- current
20                      visited[adjacent.value] = 1
21
22                  adjacent <- adjacent.next                 #Go to next element in adjacency
    list
23
24      #Driver code
25      def find_cycle(V, adj_list):
26          visited <- array of zeros of size V
27          parents <- array of -1 of size V
28          has_cycle <- False #Base assumption
29          for node in V: #This loop has a runtime of T = O(V)
30              if visited[node] == 0: #If node is unvisited, run a traverse and check on it.
31                  has_cycle <- traverse_and_check(adj_list, node, visited)
32                  if has_cycle: #If has cycle, return true to terminate the program.
33                      return True
34
35          return False #All nodes have been checked and there are no cycles.
```

Listing 2: Check cycle

Time Complexity Analysis:

For a connected graph with V_i nodes and E_i edges the time complexity of a breadth first traversal is given as $T = \mathcal{O}(V_i + E_i)$. However for a connected graph with no cycles, E_i is upper bounded by $V_i - 1$. Hence to run a depth first traversal to find if a graph has cycle will take time $T = \mathcal{O}(V_i + V_i - 1) = \mathcal{O}(V_i)$. The given graph may be considered to be made up of k many individual connected graphs. The total number of nodes is given as $V = \sum_{i=0}^{k-1} V_i$. The running time for the loop in `find_cycle` is $\mathcal{O}(V)$ and hence the overall running time for the algorithm is $T = \mathcal{O}(V) + \mathcal{O}(V_0) + \mathcal{O}(V_1) + \cdots = \mathcal{O}(V)$ (Total edges are again still upper bounded by $V - 1$ for a fully connected graph with no cycles). Hence overall running time complexity is $T = \mathcal{O}(V)$

The data structures used in this question are arrays to store pointers, singly linked lists which stores the adjacency lists for the graph and a stack for the depth first traversal. Had we decided to go with a breadth first traversal, we would have required a queue instead of a stack.

Problem 6

Description:

The problem can be seen as a graph traversal problem with the requirement of finding the shortest path between nodes in a graph. To do this we employ the Breadth First Search (BFS) implemented via a queue.

Algorithm:

```
1   def valid_pos(pos,N): #Function to check if position is within board. #T(N) = O(1)
2       if 0 < pos[0] <= N and 0 < pos[1] <= N:
3           return True
4       else:
5           return False
6
7   def get_min_knight_moves(N, start, end): #Function to get minimum moves of knight.
8       distance <- zeros(N,N) #Matrix of N,N with all entries zeros #S(N) = O(N)
9
10      queue = [] #Initialize a queue for BFS
11      queue.append(start) #Add starting element to queue.
12
13      while len(queue) != 0: #Breadth First Search
14          current = queue.pop(0)
15          #Calculate all possible positions the knight can move to.
16          neighbours=[(current[0]+2, current[1]+1),
17                      (current[0]+2, current[1]-1),
18                      (current[0]-2, current[1]+1),
19                      (current[0]-2, current[1]-1),
```

```
                    (current[0]+1, current[1]+2),
                    (current[0]-1, current[1]+2),
                    (current[0]+1, current[1]-2),
                    (current[0]-1, current[1]-2)
                    ]

        good_neighbours = [] #Find neighbours which are valid positions on the board
        for neighbour in neighbours:
            if valid_pos(neighbour,N):
                good_neighbours.append(neighbour);

        for neighbour in good_neighbours:
            #If distance == 0, cell is unvisited. Calculate its distance and add it to the
    queue.
            if distance[neighbour[0]-1][neighbour[1]-1] == 0:
                queue.append(neighbour)
                distance[neighbour[0]-1][neighbour[1]-1] = distance[current[0]-1][current
    [1]-1] + 1
            #If we have reached the end, terminate the loop return the distance.
            if neighbour == end:
                return distance[neighbour[0]-1][neighbour[1]-1]
```

Listing 3: Get Minimum Knight Moves

Time and Space Complexity Analysis:

The algorithm used is a BFS algorithm. The time complexity of the algorithm arises from having to calculate all possible neighbours of a cell for each cell. Due to the rules binding to how the knight can move, the number of neighbours of a cell is upper bounded by 8, hence the time complexity for finding neighbours, checking if they exist, and checking if they have been visited, calculating their distance can all be upper bounded by $T(N) = \mathcal{O}(1)$. In the worst case scenario, all cells would have to be visited, and there are $N \times N$ cells. This would result in the overall worst case time complexity to be $T(N) = \mathcal{O}(N \times N) = \mathcal{O}(N^2)$.

The space complexity of the algorithm arises from the `distance` matrix and the size of the `queue`. `distance` is of size $S(N) = \mathcal{O}(N \times N)$ and the size of `queue` is upper bounded by $S(N) = \mathcal{O}(N \times N)$, as there are at max $N \times N$ nodes in the graph. All other auxillary arrays like `neighbours` and `good_neighbour` are upper bounded by $S = \mathcal{O}(1)$. Hence the overall space complexity is given by $S = \mathcal{O}(N^2)$