CMP_SC 3050: Homework 2 answers

1. Consider the following computational problem.

   Input: A binary tree T.
   Output: deg_2_height(T).

   (a) Give a recursive algorithm that solves the above problem in $O(n)$ running time.

   Answer If both the left and right child of the tree are non-empty then deg_2_height(T) is 1 more than the maximum of deg_2_height of its left child and deg_2_height of its right child. If both the children are empty then deg_2_height(T) is 0. In only one child is empty then deg_2_height(T) is the deg_2_height of that child.

   So, this gives the following algorithm (the main calls the procedure deg_2_height_RECUR with $T.root$):

```
deg_2_height_RECUR(root)
 1  if root == NIL
 2       return 0
 3  elseif ((root.left == NIL) and (root.right == NIL))
 4       return 0
 5  else
 6       lheight = deg_2_height_RECUR(root.left)
 7       rheight = deg_2_height_RECUR(root.right)
 8       mxheight = MAX(lheight, rheight)
 9       if ((root.left ≠ NIL) and (root.right ≠ NIL))
10            mxheight = mxheight + 1
11       return mxheight
```

   The algorithm runs in $O(n)$ time since it visits each node and each edge at most twice and spends a constant amount of time during each visit.

   (b) Give a non-recursive algorithm that solves the above problem in $O(n)$ running time.

   Answer. We do a pre-order traversal of the tree by means of a stack S. The stack keeps pointers to those nodes whose children have not been visited. We maintain the deg_2_depth of internal nodes as we are traversing the tree as a separate attribute, where deg_2_depth of a node n is the number of degree 2 nodes on the unique simple path from the root to the node n. It is easy to see that deg_2_height of a tree is the maximum value of deg_2_depths of the leaf nodes. This gives us the following algorithm.

```
deg_2_height_NORECUR(root)
 1  if root == NIL
 2      return 0
 3  else
 4      STACK S = EMPTY
 5      mxheight = 0
 6      root.deg_2_depth = 0
 7      PUSH(S, root)
 8      while ISEMPTY(S) == FALSE
 9          n = POP(S)
10          if n.left == NIL and n.right == NIL
11              if n.deg_2_depth > mxheight
12                  mxheight = n.deg_2_depth
13          if n.left ≠ NIL and n.right ≠ NIL
14              n.left.deg_2_depth = n.deg_2_depth + 1
15              n.right.deg_2_depth = n.deg_2_depth + 1
16              PUSH(S, n.right)
17              PUSH(S, n.left)
18          if n.right ≠ NIL
19              n.right.deg_2_depth = n.deg_2_depth
20              PUSH(S, n.right)
21          if n.left ≠ NIL
22              n.left.deg_2_depth = n.deg_2_depth
23              PUSH(S, n.left)
24  return mxheight
```

The algorithm runs in $O(n)$ time since it visits each node and each edge at most twice and spends a constant amount of time during each visit.

2. Consider the following computational problem.

   Consider the following computational problem.

Input:	An array A of size n, with each entry in the array.
Output:	Array B of size n such that for each i, $B[i]$ is the length of the longest strictly increasing contiguous subsequence of the sequence $A[1], A[2], \ldots, A[i]$.

   Give an algorithm that solves the above computational problem in $O(n)$ running time. (3 points)

Answer. We read from left to right. Essentially we have to keep track of maximum length of contiguous increasing sequence we have seen so far, and the current length of contiguous increasing subsequence. As soon as the latter number exceeds the first one, we have to update the former number. As we read from left to right, we should copy the maximum length of contiguous increasing sequence into array B.

```
MAX-INC-SUBS-LENGTH(A, n)
 1  currentmax = 1
 2  currentlength = 1
 3  B[i] = 1
 4  for i = 2 to n
 5       if A[i] > A[i − 1]
 6            currentlength = currentlength + 1
 7            if currentlength > currentmax
 8                 currentmax = currentlength
 9       else
10            currentlength = 1
11       B[i] = currentmax
12  return B
```

It is easy to see that lines 1, 2, 3 and 12 get executed once, line 4 gets executed $n-1$ times and lines 5-11 are executed at most $n-2$ times, giving us a total time of $O(n)$.

3. We want to solve the following computational problem.

   Input: An integer array A of size n.
   Output: Array B such that $B[i]$ is the first-quartile of the first i integers of A.

   Give an algorithm that solves the above problem in $O(n \log n)$ running time.

Answer. Basically, we will scan the array A from left to right. Having read i numbers, we will keep the smallest $\lceil \frac{i}{4} \rceil$ numbers amongst $A[1:i]$ in a max-priority queue L and the rest of the numbers in a min-priority queue U. Thus, the first-quartile of the i numbers is the maximum element of L. So, the only thing to worry about is how to maintain L and U. Basically when we start reading read the i-th number, there are three possibilities depending on the value of the remainder when i is divided by 4 (represented thus by $i \mod 4$) :

(a) $i \mod 4$ is 0. In this case, U has exactly one element less than $\frac{3i}{4}$ thus far. Now, if $A[i]$ is greater than all the elements of L then we just need to insert $A[i]$ in U. Otherwise, we remove the maximum element of L; insert this element in U and insert $A[i]$ in L.

(b) $i \mod 4$ is 1. In this case, L has $\frac{i-1}{4}$ elements and U has $\frac{3(i-1)}{4}$ elements thus far. Now, if $A[i]$ is smaller than all the elements of U then we just need to insert $A[i]$ in L. Otherwise, we remove the minimum element of U; insert this element in L and insert $A[i]$ in U.

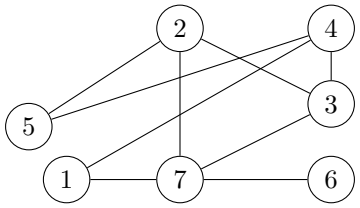
(c) $i \mod 4$ is 2. In this case, L has exactly $\lceil \frac{i}{4} \rceil$ elements thus far and U has $\lfloor \frac{3i}{4} \rfloor$ elements. Now, if $A[i]$ is greater than all the elements of L then we just need to insert $A[i]$ in U. Otherwise, we remove the maximum element of L; insert this element in U and insert $A[i]$ in L.

Observe that the first case and third case are similar. This gives the following pseudo-code:

```
FIRST-QUARTILE(A, n)
 1  B[1] = A[1]
 2  if n > 1
 3       // Initialize max-priority queue L and a min-priority queue U.
 4       // We will scan A from left to right.
 5       // After scanning the i-th number, L will store the
 6       //          smallest ⌈i/4⌉ numbers amongst A[1 : i].
 7       // After scanning the i-th number, U will store the
 8       //          numbers in A[1 : i] that are not in L.
 9       // Maximum size of A and B is ⌈n/4⌉.
10       if A[2] > A[1]
11            B[2] = A[1]
12            INSERT(L, A[1])
13            INSERT(U, A[2])
14       else
15            B[2] = A[2]
16            INSERT(L, A[2])
17            INSERT(U, A[1])
18       for i = 3 to n
19            if (i  mod 4 == 1)
20                 u = EXTRACT-MIN(U)
21                 if A[i] > u
22                      INSERT(L, u)
23                      INSERT(U, A[i])
24                 else
25                      INSERT(L, A[i])
26                      INSERT(U, u)
27            else
28                 l = EXTRACT-MAX(L)
29                 if A[i] > l
30                      INSERT(L, l)
31                      INSERT(U, A[i])
32                 else
33                      INSERT(L, A[i])
34                      INSERT(U, l)
35            B[i] = MAXIMUM(L)
36  return B
```

For the running time, note that lines 20-38 takes $O(\log n)$ time since each priority queue operation is $O(\log n)$ time. The **for** loop runs $n-2$ times and so the running time for the **for** loop is $O(n \log n)$. The other lines take $O(1)$ time and thus the running time is $O(n \log n)$ time.

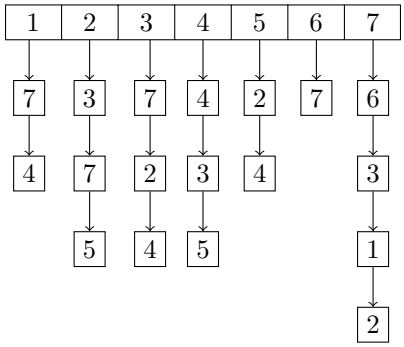
4. Consider the following undirected graph G:



(a) Give the adjacency matrix representation of G.

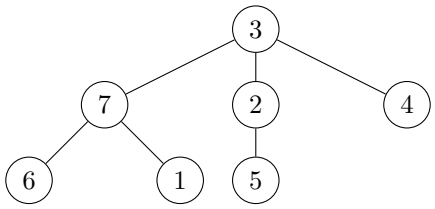
	1	2	3	4	5	6	7
1	0	0	0	1	0	0	1
2	0	0	1	0	1	0	1
3	0	1	0	1	0	0	1
4	1	0	1	0	1	0	0
5	0	1	0	1	0	0	0
6	0	0	0	0	0	0	1
7	1	1	1	0	0	1	0

(b) Give the adjacency list representation of G.



(c) Give the distances and the BFS tree generated by running the Breadth First Search algorithm on G with 3 as the source vertex.

Tree:



Distances:

$$\begin{aligned} 1.d &= 2 \\ 2.d &= 1 \\ 3.d &= 0 \\ 4.d &= 1 \\ 5.d &= 2 \\ 6.d &= 2 \\ 7.d &= 1 \end{aligned}$$