COL106: Solutions for the Major Examination

January 2021

1 Question 1

(a) The output sequence will be: 15, 2, 23, 15, 7, 10, 17, 11.

(b) $O(n)$. Even with multi-pop of $n-1$ elements, pushing $n-1$ elements will need $O(n)$ in the pushes, with each push being $O(1)$, plus $O(n)$ in the multi-pop.

(c) Returns the predecessor of the node (closest value to the node, less than the node) in the BST. Why: if node's left subtree is not empty, then it returns maximum of left subtree. Else, it returns the parent P of the first ancestor A of node (lying between root and node), which is a right child of P. Why while loop is needed: we keep moving up the ancestors, till we find A which is a right child of A's parent P. Or we hit null, which means node has no predecessor, it is the minimum node in the BST.

(d) In a red-black tree all the nodes can be black and hence the minimum r:b ratio can be 0. For the maximum ratio, the number of red nodes have to be maximized. This is possible for a fully balanced tree with all the red nodes on its leaves. In this case, every black node will have exactly two red children. Thus the maximum r:b is 2.

(e) **Part (a):** The given code first pushes the input tree on the stack. It then keeps popping the stack until it is not empty. After poping the top element from the stack, it first prints it and then pushes its right and left subrees respectively. Thus, **the algorithm is doing a pre-order traversal of the input tree**.

Part (b): For an in-order traversal, the stack needs to be modified to contain an indicator whether an element is a value to be printed or a tree to be traversed. The following is the modified code for in-order traversal.

```
function InOrder(Tree T)
        Stack S; Tree U;

        S ← new (Stack);
```

```
        S.push(T, 1); // 1 indicates a tree to be traversed
        while not empty(S) do
            (U, val) ← S.pop()
            if (U != null) then
                if (val == 2) // Print the value only
                    print(U.value)
                else
                    S.push(U.right, 1) // Tree to be traversed
                    S.push(U, 2) // Only value to be printed. No traversal
                    S.push(U.left, 1) // Tree to be traversed
                endif
            endif
        end while
    endFunction InOrder
```

2 Question 2

(a) The graph will be directed and weighted. There will be $P * Q$ nodes in the graph, one for each cell of the maze. The weight of the edge $w(u, v)$ = height(v). A droid can traverse between nodes only if the edge weight is >= its size. There will be an edge between all the nodes, whose corresponding cells are adjacent in the maze.

Alternative can be an undirected graph, with minimum of (u, v) and (v, u) as edge weight. Graph creation below will change accordingly.

(b)
```
function CreateGraph(arr[][], P, Q)
    adj[][] ← 2D array of size P * Q * P * Q;
    for i ← 1 to P * Q do
        for j ← 1 to P * Q do
            adj[i][j] ← 0;
        end for
    end for
    for i ← 1 to P do
        for j ← 1 to Q do
            //Check for out of bounds indeces
            adj[i * P + j][i * P + j − 1] = arr[i][j − 1];
            adj[i * P + j][i * P + j + 1] = arr[i][j + 1];
            adj[i * P + j][(i − 1) * P + j] = arr[i − 1][j];
            adj[i * P + j][(i + 1) * P + j] = arr[i + 1][j];
        end for
    end for
endFunction CreateGraph
```

(c) A droid can reach T if there is a path in the graph from S to T, containing no edge of weight less than the droid size. We can find such a path by a simple modification to bfs/dfs starting at S. While exploring an edge e in either of the algorithm, check if $width(e) \geq x$, where x is the current droid size being checked - call these admissible edges. Add vertices to the queue/stack if and only if they are explored via admissible edges. If T is explored at some point, we have a desired path. Otherwise we don't.

We can solve the given problem by using the above subroutine in conjunction with a binary search. Note that the array of droids is sorted. So, as usual, start at the middle droid and check if there is an $S - T$ path that can handle this droid size. If yes, then recurse on the right subarray, else recurse on the left subarray. The overall runtime is $O(E + V) \log d$ where $O(E + V)$ comes from the graph search and $\log d$ from the binary search.

```
function boolean isReachable(adj,size,s,t)
          visited ← boolean array of size P * Q;
          for i ← 0 to P * Q − 1 do
               visited[i] ← false;
          end for
          dfs(adj,size,s,visited);
          return visited[t];
endFunction isReachable

function void dfs(adj,size,s,visited)
          visited[s] ← true;
          for j ← 1 to P * Q do
               if adj[s][j] >= size do
                  dfs(adj,size,j,visited);
               end if
          end for
endFunction dfs

function int FindBiggestDroid(sortedarr,S,T)
          int s ← S_x * P + S_y; int t ← T_x * P + T_y;
          int l ← 0; int r ← d − 1;
          int m ← −1;
          while l <= r do
                m = l + (r − l)/2;
                if (!isReachable(adj, sortedarr[m], s, t) do
                   l ← m − 1;
                else
                      r ← m + 1;
                end if
          end while
          return m;
endFunction FindBiggestDroid
```

(d) Replace length of path in Dijkstra with minimum weight of an edge in that path. Let's call this bottleneck weight. For a droid to traverse a path, its size has to exceed the bottleneck weight of that path. Run Dijkstra to find path between S and T with maximum bottleneck weight B. Using a max-heap (instead of min-heap typically used for minimum length path in Dijkstra), this can be run in $((E+V)\log V)$ runtime to get B. Run binary search on sorted array to find value <= B. This will take $(\log d)$ time. So overall time $((E+V)\log V + \log d)$.

3 Question 3

(a) The minimum number of piles will be 1 when all the numbers arrive in strictly decreasing order. One such sequence could be $n, n-1, \ldots, 3, 2, 1$. The maximum number of piles will be n when all the n numbers arrive in strictly increasing order. One such sequence could be $1, 2, 3, \ldots, n-1, n$.

(b) A number is placed on top of a pile only if it is smaller than the current number at the top of the pile. Therefore, within each pile, the numbers will be sorted in ascending order from the top of the pile to the bottom of the pile. A list data structure could be used to store numbers within a pile.
The topmost numbers of each of the piles will be in increasing order (from left to right). In Phase 1, an array may be used to store the pointers to the list corresponding to the piles. In phase 2, the pointers to the list from this array may be inserted into a priority heap, using the topmost element value as the key.

(c) The insert function given below first checks if the value to be inserted is greater than the value on the top of the right most pile. If not, then it performs binary search on the piles using the topmost values on the piles. The total time for this will be $O(\log(n))$ for each insert operation, since maximum number of piles will be n. The pseudo code for the insert function is given below. This function may be called repeatedly to insert the elements into the piles.

```
function Insert(val, P) // Algorithm multi-pile sorting
    l ← 0;
    h ← rightmost non-empty pile;
    if (val > topval(P[h])) // Insert the val into a new pile on the right
        P[h+1] = new List(val)
        return
    endif
    // Now perform a binary search to find the right place to insert
```

```
    mid ← (l+h)/2
    while (mid > l)
        if (val > topval(P[mid]))
            l ← mid+1
        else
            h = mid
        endif  mid ← (l+h)/2
    end while
    if (val ≤ topval(P[l])
        P[l].insert(val)
    else
        P[l+1].insert(val)
    endif
endFunction Insert
```

(d) For the delete operations, all the piles are first inserted into a heap data structure, indexed by the value of the topmost element in the pile. Now the Extract function calls deletemin on the heap to get the pile with the smallest element on the top. This element is removed from the pile and the pile is re-inserted into the heap if it has more elements. The pseudo code is given below. The heap can have atmost n elements. Therefore the deletemin and the inset opetaions take at most $\log(n)$ time. All the other operations take a constant amount of time. Hence the Extract function takes $O(\log(n))$ time.

```
function Initialize(P) // Initialize Phase 2 of multi-pile sorting
    H ← new Heap()
    h ← rightmost non-empty pile;
    for i = 1 to h
        H.insert(topval(P[i]), P[i])
    end for
    return H
endFunction Initialize

function Extract(H) // Extract the smallest element in multi-pile sorting
    P ← H.deletemin()
    val ← P.val
    if (P.next != nil)
        H.insert(P.next)
    endif
    return val
endFunction Extract
```

4 Question 4

(a) Initialization rules: $d(i,j,1) = W(i,j)$ if $(i,j) \in E; d(i,j,1) = \infty$ if $(i,j) \notin E$.
Update rule (inefficient): $d(i,j,k) = \min_{u \in V} d(i,u,k-1) + d(u,j,1)$
Update rule (efficient): $d(i,j,2k) = \min_{u \in V} d(i,u,k) + d(u,j,k)$

(b) The algorithm runs for log(n) iterations to iteratively compute $d(i,j,2^{k-1})$ which is stored in the variables $d(i,j,k)$ using the efficient update rule defined above. It also assumes the availability of the tree data structure which stores the corresponding shortest paths in $P(i,j,k)$ using a hierarchical representation of the edges in the shortest paths. The in-order traversal of this tree will give the shortest paths. In the code below, it is assumed that the function Tree with one argument creates a tree with leaf node containing the corresponding edge and the Tree function with two arguments creates an internal node with corresponding two tree arguments as its subtrees. For internal nodes, there is no corresponding edge stored in the internal node. So an in-order traversal prints out edges stored in the leaf nodes only.

```
function APSP(V, E, W) // Algorithm All Pair Shortest Paths
   // Initializations
   n = size(V);
   for i = 1 to n
      for j = 1 to n
         if ((i,j) ∈ E)
            d(i,j,1) ← W(i,j)
            P(i,j,1) ← new Tree((i,j))
         else
            d(i,j,1) = ∞
            P(i,j) = nil
         endif
      endfor
   endfor
   // Upadate paths and path lengths iteratively using efficient update rule
   for k = 1 to ⌈log(n)⌉
      for i = 1 to n
         for j = 1 to n
            d(i,j,k+1) ← d(i,j,k)
            P(i,j,k+1) ← P(i,j,k)
            for u = 1 to n
               if (d(i,u,k) + d(u,j,k) < d(i,j,k+1))
                  d(i,j,k+1) ← d(i,u,k) + d(u,j,k)
                  P(i,j,k+1) ← new Tree(P(i,u,k), P(u,j,k))
```

```
            endif
          endfor
        endfor
      endfor
    endfor
endFunction APSP
```

(c) **Time complexity:** There are for nested for loops three of which go from 1 to n and fourth from 1 to $\lceil \log(n) \rceil$. In the innermost loop, the update of distances is a constant time operation. The trees can also be updated in constant time since there is no need to copy earlier path trees. Storing the pointers to the two subtrees suffices. Hence the total runtime of this algorithm is $O(n^3 \log(n))$.
Space complexity: The algorithm maintains $\log(n)$ all-pair distances taking the space $n^2 \log(n)$. It also maintains $n^2 \log(n)$ trees containing the shortest paths. Each of the node in the tree uses trees stored earlier and just uses new space of a single node to store the two sub-paths. Hence the space required is also $O(n^2 \log(n))$.

(d) **Proof of Correctness:** (Using the efficient update rule).
Claim: At the beginning of the k^{th} iteration of the outermost for loop (after the initialization phase) of the algorithm, $P(i, j, k)$ and $d(i, j, k)$ respectively represent the shortest path and its length from vertex i to vertex j while using less than or equal to 2^{k-1} edges.

Proof: The proof follows from induction on k. The base case is trivially true for $k = 1$ because if there is a direct edge from i to j then the path from i to j and its length is correctly initialized.

For induction step, we assume the induction hypothesis to be true for a given value of k and then we show that it is also true for $k + 1$. To see this, note that the shortest path from vertex i to vertex j of length less than 2^k can be decomposed into two paths (from i to u and from u to j) of lengths less than or equal to 2^{k-1} through an intermediate vertex u. The minimum of all such shortest paths (using less than or equal to 2^{k-1} edges) through the vertex u must give the required shortest path from vertex i to vertex j (using less than or equal to 2^k edges. The innermost loop on vertex u is indeed finding the best such path through the vertex u. Hence at the beginning of the $(k+1)^{th}$ iteration, the variables $P(i, j, k+1)$ and $d(i, j, k+1)$ give the shortest path (between vertices i and j using less than 2^k edges) and its length respectively.