ESO207A: Data Structures and Algorithms

Homework *2* **Due: September 13, 2017, end of class.**

Instructions.

1. Start each problem on a new sheet. For each problem, Write your name, Roll No., the problem number, the date and the names of any students with whom you collaborated.
2. For questions in which algorithms are asked for, your answer should take the form of a short write-up. The first paragraph should summarize the problem you are solving and what your results are (time/space complexity as appropriate). The body of the essay should provide the following:
   (a) A clear description of the algorithm in English and/or pseudo-code, where, helpful.
   (b) At least one worked example or diagram to show more precisely how your algorithm works.
   (c) A proof/argument of the correctness of the algorithm.
   (d) An analysis of the running time of the algorithm.

   Remember, your goal is to communicate. *Full marks will be given only to correct solutions which are described clearly.* Convoluted and unclear descriptions will receive *low marks.*

Problem 1. Heaps: Running Median. (30)

The median of a set of n numbers is the $(n+1)/2$th number, for n odd, and it is the $\lfloor (n+1)/2 \rfloor$th ranked member and the $\lceil (n+1)/2 \rceil$ th ranked member (i.e., two medians), for n even. The following operations on a dynamic set S of elements have to be supported efficiently: INSERT(x) in $O(\log n)$ time and MEDIANin $O(1)$ time. Give a correctness and complexity analysis of your algorithm.

(*Hint:* Let the rank 1 member be the smallest member of S and rank n member be the largest member of S, etc.. Keep the smaller half of the numbers (ranks $1 \ldots \lfloor (n+1)/2 \rfloor$) in one heap and the remaining higher ranked numbers in a second heap. Maintain this invariant.) (Correctness, arguments: 13, Pseudo-code: 12, Complexity Analysis: 5)

Solution. Keep two heaps, H_1 as a max-heap and H_2 as a min-heap. We will maintain the following invariants.

1. H_1 keeps the members with ranks between 1 and $\lfloor (n+1)/2 \rfloor$ inclusive. This is maintained as a max-heap.
2. H_2 keeps the members with ranks between $\lfloor (n+1)/2 \rfloor + 1, \ldots, n$. This is maintained as a min-heap.
3. Keep the parity bit $p = n \mod 2$.

Outline: The INSERT(x) procedure: We will consider two cases, depending on whether n was even or odd, that is, p is 0 or 1, respectively.

Case 1: p is 1, or n was odd. If $x < H_1$.MAXIMUM, then, x should be in the first half, i.e., in H_1. Since, currently, H_1 has $\lfloor (n+1)/2 \rfloor = \lfloor (n+2)/2 \rfloor$ elements, for n odd. Hence, (1) let $m_1 = H_1$.EXTRACT_MAX; that is, remove the largest element of H_1, (2) INSERT(x) into H_1, and (3) INSERT(m_1) into H_2. The size of H_1 remains unchanged, since we have deleted the largest element and added x, and the size of H_2 is increased by 1. If $x > H_1$.MAXIMUM, then x is inserted into the second half. Change p to 0, since now n is even. The maximum element of H_1 is the lower median of S (element with rank $\lfloor (n+1)/2 \rfloor$ and the minimium element is the upper median of S (element with rank $\lceil (n+1)/2 \rceil$.

Case 2: p is 0, that is n was even. If $x < H_1$.MAXIMUM, then, x should be inserted in the first heap H_1. If $x > H_1$.MAXIMUM, then,(1) H_2.INSERT(x) into H_2, (2) $m_2 = H_2$.EXTRACT_MIN: extract the minimum element from H_2, and, (3) H_1.INSERT(m_2) : insert m_2 into H_1. The pseudo-code is given below.

```
// Data Structures: 1. H1: max-heap, (2) H2: min-heap, and (3) a parity bit p.
// Assume that for an empty max-heap H, H.MAXIMUM is −∞ and
// for an empty min-heap H, H.MINIMUM is ∞.

INSERT(x)
1.   if  p == 1        // n is odd prior to insertion
2.        if  x < H1.MAXIMUM
3.             m1 = H1.EXTRACT_MAX
4.             H1.INSERT(x)
5.             H2.INSERT(m1)
6.        else
7.             H2. INSERT(x)
8.   else              // n is even prior to insertion
9.        if  x ≤ H1.MAXIMUM
10.            H1.INSERT(x)
11.       else
12.            H2.INSERT(x)
13.            m2 = H2.EXTRACT_MIN
14.            H1.INSERT(m2)
15.  p = (1 + p)  mod 2     //flip the bit

MEDIAN(x)
1.   if p == 1
2.        return  H1.MAXIMUM
3.   else
4.        return ( H1.MAXIMUM, H2.MINIMUM)
```

Time complexity analysis: The INSERT(x) procedure in any execution makes one call to H_1.MAXIMUM, which takes $O(\log(n+1)/2)$ time, one call to either H_1.EXTRACT_MAX or H_2.EXTRACT_MIN, each of which takes $O(\log(n+1)/2)$ time, and calls to H_1. INSERT or H_2.INSERT. The total time taken

is $O(\log n)$. The median retrieves either H_1. MAXIMUM for max-heap H_1 or H_2.MINIMUM for min-heap, each requiring time $O(1)$.

Problem 2. Stack (30)

Given an array $A[1,\ldots,n]$ containing positive integers, for each $1 \le i \le n$, let $B[i]$ denote the largest number of consecutive indices $i-B[i]+1, i-B[i]+2, \ldots, i$ such that $A[i-B[i]+j] \le A[i]$, for $1 \le j \le B[i]$. The problem is to compute the $B[1,\ldots,n]$ array, given $A[1,\ldots,n]$. For example, given $A[6] = \{9,5,6,11,2,13\}$, the output $B[6] = \{1,1,2,4,1,6\}$. Design an algorithm that makes a single pass over the array A and may keep $O(n)$ space extra storage.

(*Hint:* Define $H(i)$ to be the index $j \le i$ that is closest to i such that $A[j] > A[i]$. If $H(i)$ is computed, then, the answer $B(i) = i - H(i)$. Assume, $H(1) = 0$, and for convenience, $A[0] = \infty$. Maintain the invariant: after processing $A[i]$, the top of stack contains $H(i)$.

(Correctness and Pseudocode: 10 + 10, Analysis: 10)

Solution. Suppose we are processing an index $i+1$. Keep a stack whose top is $j_1 = H(i)$, that is, the index $j_1 \le i$ closest to i such that $A[j_1] > A[i]$. The previous entry on the stack is $H(j_1) = H(H(i))$, which is the smallest index j_2 such that $A[j_2] > A[j_1]$, and so on–this defines the stack. Initially, $H(1) = 0$, and assume $A[0] = \infty$.

To process index $i+1$, first compare $A[i+1]$ with $A[i]$. If $A[i+1] < A[i]$, then $H(i+1) = i$ and so i is pushed onto stack. If $A[i+1] \ge A[i]$, then, $A[H(i)+1,\ldots,i] \le A[i] \le A[i+1]$. Hence, we compare $A[i+1]$ with $A[H(i)]$. If $A[i+1] < A[H(i)]$, then, $H(i+1) = H(i)$ and our processing of index $i+1$ is done. Otherwise, $H(i)$ is popped from stack, exposing $H(H(i)) = H_2(i)$ say. Now $H_2(i)$ is the index closest to $H(i)$ such that $A[H_2(i)] > A[H(i)] > A[i]$. This popping procedure is repeated until we find an index r on the stack such that $A[r] > A[i+1]$. The output is $r - (i+1)$.

```
COMPUTEB(A[], n)
// A[0, 1, ..., n] is an array of numbers, assumes A[0] is ∞.
1.  let B[1, ..., n] be a new array for output
2.  stack S;   // stack S is initialized to empty
3.  for i = 1 to n
4.      if  A[i] < A[i − 1]
5.          S.push(i − 1)
6.      else
7.          j = S.TOP()
8.          while  A[j] ≤ A[i]
9.              S.pop()
10.             j = S.TOP()
11.     B[i] = i− S.TOP()
```

The correctness proof, as indicated above, is based on the invariant that if the stack contains $j_1, j_2, \ldots, j_t$ with j_1 on top and $j_t = 0$ at the bottom at any time, and $i+1$ is the current index, then, $j_1 = H(i), j_2 = H(j_1), j_3 = H(j_2), \ldots, j_t = H(j_{t-1})$.

Complexity Analysis: The time taken is $O(n)$ plus some constant times the number of times line 9–the *pop* statement is executed. Note that once an index is popped from the stack, it is never

pushed again or compared with any other element. If there are n items, then there can be a total of at most n pop operations. Hence, time complexity is $O(n)$.

Problem 3. Queues: Breadth first search in a maze. (40)

The input is a two dimensional $N \times N$ array $M[1 \ldots N, 1 \ldots, N]$ of tiles. A mouse is sitting initially at tile $M[1,1]$. From tile numbered $[i,j]$, the mouse can navigate in *one step* to any of the four neighboring tiles at $[i+1,j]$, $[i,j+1]$, $[i-1,j]$ and $[i,j-1]$, provided, in each case, the mouse does not step out of the grid. Additionally, associated with each tile $M[i,j]$ there is a status flag $M[i,j].status$, which is 1 if the tile is "normal" and 0 if the tile is defective. If the tile (i,j) is defective, then the mouse cannot navigate to it, that is, this tile cannot be reached from any of its neighbors. The exit from this maze is at position $[N,N]$. Given the input tile array M, design an algorithm that finds a shortest length path (in terms of number of steps) from the tile $[1,1]$ to $[N,N]$ without stepping onto any defective tile. You algorithm should run in time linear in the size of the input. Give a correctness proof and a complexity analysis of your algorithm.

(*Note:* This is an application of the graph search method called breadth-first-search.)

Solution.. As mentioned, this problem is a classic application of the breadth-first-search method to explore graphs.

We will keep a queue of tile indices (i.e., pairs $[i,j]$), that is initially empty. Associated with each tile $M[i,j]$, we will keep the following attributes.

1. $M[i,j].status$, as explained in the question.
2. $M[i,j].d$ represents the shortest distance (in terms of the number of hops) from the starting tile.
3. $M[i,j].pred$ is tile say $[j,k]$ from which $[i,j]$ was first visited.
4. $M[i.j].condition$ which takes one of three values $\{\text{UNIVISITED}, \text{VISITED}, \text{INQUEUE}\}$.

We will write a slightly more general routine $\text{BFS}([u,v])$ that computes a breadth-first-search of the entire grid starting at position (i,j). The top-level call is $\text{BFS}([1,1])$.

```
BFS([u, v]) {
1.   QUEUE Q initialized to φ
2.   Q.Enqueue([u, v])              // Insert [u, v] into Q.
3.   M[u, v].d = 0                  // visited at time 0
4.   M[u, v].pred = NIL       // [u, v] has no predecessor
5.   M[u, v].condition = INQUEUE
6.   while Q is not empty // some tiles still to be explored
7.        [i, j] = Q.DEQUEUE;
8.        for each of the possible neighbors [k, l] of [i, j]
9.             if M[k, l].status == 1 and (M[k, l].condition != INQUEUE
10.                              or M[k.l].condition != VISITED)
11.                  // [k, l] is a normal tile and is neither in queue nor explored completely
12.                  Q.ENQUEUE([k, l])
13.                  M[k, l].condition = INQUEUE
14.                  M[k, l].d = M[i, j].d + 1
                          //[k, l] is at a distance of 1 more than [i, j] from starting tile.
15.                  M[k, l].pred= [i, j]        //[k, l] was discovered from [i, j]
16.       M[i, j]. condition = VISITED       //explored completely
```

The algorithm makes the following assumptions on behalf of the mouse. For all tiles $[i, j]$, $M[i, j].d = \infty$, $M[i, j].$ *pred*= NIL and $M[i, j].$*condition* = UNVISITED. In a typical BFS code, these are assigned in the initialization code.

The following are the properties of the BFS($[u, v]$) call.

1. For any tile $[i, j]$, $[i, j]$ belongs to the queue Q iff $M[i, j]$. condition = ENQUEUE. Once $[i, j]$ is dequeued, $M[i, j].$*condition* is set to VISITED and remains so until the end of the program. Also, once dequeued, $[i, j]$ is never enqueued again.

2. If there is a path from starting tile to a tile $[i, j]$, then, upon termination $M[i, j].d$ is finite (i.e., not ∞).

3. There is a path from $[u, v]$ to $[i, j]$ iff $M[i, j].d$ is finite. Further, if $M[i, j].d$ is finite, then it is the length of some path from starting tile to $[i, j]$.

4. If at any time, the queue in enqueue order consists of sequence of tiles $(a_1, \ldots, a_k)$, where, a_1 is the head of the queue and a_k is the tail of the queue, then, $M[a_1].d \leq M[a_2].d \leq \ldots \leq M[a_k].d$ and $M[a_k].d - M[a_1].d \leq 1$.

5. Tiles $[i, j]$ are enqueued in non-decreasing order of $M[i, j].d$. That is, if $[i, j]$ is enqueued before $[k, l]$, then, $M[i, j].d \leq M[k, l].d$. Tiles are dequeued in non-decreasing order of $M[i, j].d$. (This follows from the previous sentence, since, for a queue, the dequeue order is the enqueue order–FIFO).

6. For any tile $[i, j]$, if $M[i, j].d$ is set to a value smaller than ∞, then, this value is the length of the shortest path from the starting tile $[u, v]$ to tile $[i, j]$ in terms of number of hops in the maze.

7. If $M[i,j].d$ is set to a value smaller than ∞, then the path obtained by traversing the *pred* field from $[i,j]$ to the starting vertex gives the shortest path from the starting vertex to $[i,j]$ in reverse and is of length $M[i,j].d$. That is, the sequence is $[i_1,j_1] = M[i,j].pred$, $[i_2,j_2] = M[i_1,j_1].pred$, ..., $[u,v] = M[i_r,j_r].pred$, and the path is

$$[u,v] \quad [i_{r-1},j_{r-1}] \quad \ldots [i_1,j_1] \quad [i,j]$$

of length $r = M[i,j].d$ and is a shortest path from the starting tile to $[i,j]$ (although there could be other shortest paths).

We will now prove the various properties of the invariant.

1 Note that the enqueue of tile $[k,l]$ happens along with setting $M[k,l].condition = \text{INQUEUE}$ in lines 12 and 13 of the same iteration of the for loop. Also note that once $M[k,l].condition$ is INQUEUE. Once condition is INQUEUE, it is not inserted into the queue again. The next change happens in line 16 when $[k,l]$ is dequeued and $M[k,l].condition$ switches to VISITED. In line 9, a tile $[k,l]$ is enqueued only of $M[k,l].condition \notin \{\text{VISITED}, \text{ENQUEUE}\}$. Hence, once dequeued, $[i,j]$ is never enqueued again. This proves the first property.

2 First it will be shown that if there is a path from the source tile to a tile $[i,j]$, then, upon termination $M[i,j].d$ is finite. Let $P = a_1, a_2, \ldots, a_k$ be a valid path of the maze, where, $a_1 = [u,v]$ the starting tile and $a_k = [i,j]$ is the last tile and a_i and a_{i+1} are adjacent and normal tiles (so that one can pass from a_i to a_{i+1}).

Suppose for sake of contradiction that $a_k.d = \infty$. Consider the earliest index i such that $a_i.d = \infty$. Now, $i > 1$, since we initialize $a_1.d = 0$. Now, a_{i-1} gets enqueued at some instant t and at a later instant w gets dequeued. (This is because the graph is finite). In the for loop of line 8 corresponding to a_{i-1}, the tile a_i is checked to see if $a_i.condition$ is INQUEUE and $a_i.status = 1$. Now, we know that $a_i.status$ is 1, since P is a valid path. Now $a_i.condition \notin \{\text{INQUEUE}, \text{VISITED}\}$ since, otherwise, $a_i.d$ is already finite. Note that once $a_i.d$ is set, it is never changed again. But having $a_i.d$ as finite is a contradiction since we assumed that $a_i.d = \infty$. Hence, we must assume that $a_i.condition \notin \{\text{INQUEUE}, \text{VISITED}\}$.

Hence, in the loop of line 8 corresponding to a_{i-1}, the tile a_i will be selected as one of the neighbors and it will be enqueued, and $a_i.d$ will be set to $a_{i-1}.d + 1$, which is finite. However, we had assumed that i is the smallest index in P such that $a_i.d = \infty$. This is a contradiction. Hence, there is no smallest index in P such that $a_i.d = \infty$. Hence, for every tile a_i in P, $a_i.d$ is finite. This proves the second part of the second invariant property.

3 We now try to prove the first property, that is, $M[i,j].d$ equals the length of a shortest path from starting tile to $[i,j]$. We will first prove a simpler one-sided property.

(2.1a) Let $1 \leq i,j \leq n$ and suppose, upon termination of the algorithm, $M[i,j].d$ is finite. Then, there is a path of P from the starting tile $[u,v]$ to $[i,j]$ of length at most d.

The proof is by induction on the value of $M[i,j].d$. The base case happens at initialization, when $M[u,v].d = 0$ and there is a path of 0 length from $[u,v]$ to $[u,v]$. Suppose the statement is true for all $[i,j]$ such that $M[i,j].d \leq d_0$. Now suppose $[k,l]$ is a tile such that $M[k,l].d = d_0 + 1$. Then, suppose $M[k,l].pred = [i,j]$, that is, $[k,l]$ was enqueued as a result of dequeuing $[i,j]$ and following the neighborhood of $[i,j]$. Hence, $M[i,j].d = d_0$ and by the induction hypothesis, there is a path P' of length d_0 starting at $[u,v]$ and ending at $[i,j]$. Hence, by

traversing from $[i, j]$ to $[k, l]$, we get a path P' of length $d_0 + 1$ starting at $[u, v]$ and ending at $[k, l]$. This proves the induction case.

Let $\delta([u, v], [i, j])$ denote the length of a shortest path from $[u, v]$ to $[i, j]$ (passing through normal tiles only).

Property 2.1a and 2.2 shows that there is a path from $[u, v]$ to $[i, j]$ iff $M[i, j].d$ is finite. Further, if $M[i, j].d$ is finite, then it is the length of some path from starting tile to $[i, j]$.

Collectively, this means that $\delta([u, v], [i, j]) \leq M[i, j].d$.

4 The proof is by induction on the number of enqueue operations of Q. Suppose that the statement holds for all the first n enqueue operations. The base case is when $n = 1$, that is, Q consists only of $[u, v]$, and the statement holds.

Now for the induction case, let the current state of the queue be $(a_1, a_2, \ldots, a_k)$. Now, a_1 is dequeued from the queue and all its neighbors b_j are checked and enqueued, provided they have not been enqueued or visited earlier. Let b_j is a newly discovered tile and adjacent from a_1. Then, b_j is enqueued with $M[b_j].d = M[a_1].d + 1$. Note that the new queue is $(a_2, a_3, \ldots, a_{k+1}, b_j)$ and $M[b_j].d = M[a_1].d + 1 \geq M[a_k].d$, ensuring that the $.d$ attribute of tiles in Q are in non-decreasing order. Also,

$$M[b_j].d - M[a_2].d = M[b_j].d - M[a_1].d + M[a_1].d - M[a_2].d = 1 + M[a_1].d - M[a_2].d \leq 1$$

proving the property.

As a consequence of property 4, the enqueue order is consistent with a non-decreasing order of $.d$ distances.

6 We would now like to show that $\delta([u, v], [i, j]) \geq M[i, j].d$, for each tile $[i, j]$. Suppose this claim is not true, that is, there is an $[i, j]$ such that $\delta([u, v], [i, j]) < M[i, j].d$. Among the set of all such $[i, j]$ satisfying this property, consider the one with the smallest value of $\delta([u, v], [i, j])$. Clearly, $[i, j] \neq [u, v]$, since, $M[u, v].d = 0$. Let $\delta([u, v], [i, j]) = d_0$. Let $a_0 = [u, v], a_1, \ldots, a_{d_0} = [i, j]$ be a shortest path from $[u, v]$ to $[i, j]$. Then, we have by the assumption that $a_{d_0-1} = [k, l]$ (say) satisfies $M[k, l].d \leq \delta([u, v], [k, l])$. Now consider the iteration of the while loop when $[k, l]$ is dequeued.

There are two cases, either $[i, j]$ is already enqueued, in which case, by property (4), $M[i, j].d \leq M[k, l].d + 1 = \delta([u, v], [k, l])$, proving the case. If $[i, j]$ is not enqueued, then, the for loop in line 8 will now enqueue it and set $M[i, j].d = M[k, l].d + 1 = \delta([u, v], [k, l])$, thus proving the property that $\delta([u, v], [i, j]) \geq M[i, j].d$.