ESO207A: Data Structures and Algorithms

Homework *2: Heaps, QuickSort* HW Due Date: Sep 3, 2018

Problem 1. *Heap.* The following is an alternative routine to build a max-heap by permuting the elements of an array. It uses the subroutine *Max-Heap-Insert*$(A, heapsize, v)$. This routine takes $A[1,\ldots,heapsize]$ and inserts v into the max-heap. It also increments *heapsize* by 1 (*heapsize* is a reference parameter).

procedure Build-Max-Heap1(A, n) // Build Max Heap for the elements in $A[1,\ldots,n]$
1. *heapsize* $= 1$
2. **for** $i = 2$ **to** n
3. *Max-Heap-Insert*$(A, heapsize, A[i])$

1. Show that this procedure takes $O(n \log n)$ time. (10)

   Solution. Inserting an item into a Max-Heap consisting of k elements *Max-Heap-Insert*(A, k, v) takes time $O(\log k)$. Thus, the time by above procedure is:

$$\sum_{k=2}^{n} O(\log k) = \sum_{k=2}^{n} O(\log n) = O(n \log n) \enspace .$$

2. Show a worst case input and analyze it to show that on that input the procedure takes $\Omega(n \log n)$ time. Hence this algorithm takes $\Theta(n \log n)$ time worst-case.

   Solution. Insert $1, 2, \ldots, n$ in sorted order into the Max-Heap. Consider the state of the heap when the call *Max-Heap-Insert*$(A, heapsize, k)$ is made, with *heapsize* $= k-1$, for $k \geq 2$. Then k should finally reach the root of the heap, and this will require $\Omega(\log k)$ operations. Thus, we obtain the complexity as $\Omega(\sum_{k=2}^{n} \log(k))$. We have,

$$\begin{aligned}
\sum_{k=2}^{n} \log(k) &\geq \sum_{k=\lceil n/2 \rceil}^{n} \log_2(k) \\
&\geq \sum_{k=\lceil n/2 \rceil}^{n} \log_2(n/2) \\
&\geq (n/2)\log_2(n/2) \\
&\geq (n/2)\log_2 n - (n/2) \\
&= \Omega(n \log n) \enspace .
\end{aligned}$$

Problem 2. *CLRS 6-3 Young Tableau.* An $m \times n$ Young tableau is an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries may be ∞ that are treated as non-existent items. Thus a Young tableau can be used to hold $r \leq mn$ finite numbers. Note that an $m \times n$ tableau Y is empty if $Y[1,1] = \infty$ and is full if $Y[m,n] < \infty$. An example Young tableau is $\begin{bmatrix} 2 & 4 & 8 & 12 \\ 3 & 5 & 9 & \infty \\ 12 & 14 & 16 & \infty \end{bmatrix}$

1. Give an algorithm for EXTRACT-MIN on a non-empty $m \times n$ Young tableau that runs in time $O(m+n)$. (*Hint*: Follow Min-Heapify.)

   Solution. Let $A[m \times n]$ be the given tableau. The minimum element is clearly $A[1,1]$. Suppose we extract $A[1,1]$ to make the spot $A[1,1]$. Now replace it by the minimum of its two adjacent neighbors $A[1,2]$ (right neighbor) and $A[2,1]$ (down neighbor). This creates a new vacant spot, which is propagated by continuing this algorithm. The algorithm resembles a Heapify procedure. Assume that for any finite number a, $a < \infty$ and $\infty \le \infty$.

```
DOWN(i, j) { return (i, j + 1) }
RIGHT(i, j) { return (i + 1, j) }
EXTRACT-MIN(A, m, n)
min = A[1, 1]
(x, y) = (1, 1)
done = FALSE
while not done
      (mx, my) = undefined
// (mx, my) is the index of the minimum of A[RIGHT(x, y)] and A[DOWN(x, y)]
// some corner cases are checked to ensure that RIGHT(x, y) is a legal entry
// and if DOWN(x, y) is a legal entry. Their minimum (or whichever exists) is taken
// and is exchanged with A[x, y]. (x, y) is set to the index of this minimum,
// and iterate.
      if DOWN(i, j) ≤ (m, n) (mx, my) = DOWN(x, y)
      if RIGHT(i, j) ≤ (m, n)
            if (mx, my) == DOWN(x, y)
                  if A[mx, my] > A[RIGHT(x, y)] (mx, my) =RIGHT(x, y)
            else (mx, my) = RIGHT(x, y)
      if (mx, my) == UNDEFINED done = TRUE
      else swap A[mx, my] with A[x, y]
            (x, y) = (mx, my)
return min
```

   Time complexity. Each iteration of the while loop takes $\Theta(1)$ time. In each iteration, either the variable *done* is set to TRUE and the loop terminates. Otherwise, (x, y) moves either one position to the right, or one position down. There are only m positions to go down before the tableau ends and at most n positions to go right, so the time complexity is at most $O(m+n)$.

2. Show how to insert a new element into a non-full $m \times n$ Young tableau in $O(m+n)$ time.

   Solution. Let v be the value of the new element. Place it in the lower-most rightmost position $A[m, n]$. Since A is non-full, $A[m, n] = \infty$. If $A[m, n]$ is smaller than its left neighbor or its upper neighbor, then, it is exchanged with the maximum of these two neighbors. This process is iterated until the element is lodged in its correct place.

```
UP(i, j) { return (i, j − 1) }
LEFT(i, j) { return (i − 1, j) }
INSERT(A, m, n, v) // insert value v into non-full Young tableau A[m × n]
A[m, n] = v
(x, y) = (m, n)
```

```
done = FALSE
while not done
    (mx, my) = (x, y)
    if LEFT(x, y) ≥ (1, 1) and A[LEFT(x, y)] > A[x, y]
        (mx, my) = LEFT(x, y)
    if UP(x, y) ≥ (1, 1) and A[UP(x, y)] > A[mx, my]
        (mx, my) = UP(x, y)
    if (mx, my) == (x, y) done = TRUE
    else swap A[x, y] with A[mx, my]
        (x, y) = (mx, my)
```

Time complexity. In each step of the while loop, there is some constant amount of processing done $\Theta(1)$. After this processing, either the variable *done* is set to TRUE after which the while loop terminates, or, (x, y) either moves left or up from its current position. Since, x, y can only move left m times and up n times, before ending up at the boundary of the tableau, the complexity is $O(m + n)$.

3. Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau to sort n^2 numbers in $O(n^3)$ time.

   Solution. Create an empty $n \times n$ tableau B and initialize it with all ∞. Now insert the elements of the given tableau $A[n \times n]$ (in any order) one by one using the INSERT procedure above. Each insertion takes $O(n)$ time. Hence, the tableau B with the Young tableau property is created in time $O(n) \times n^2 = O(n^3)$.

   Repeatedly call the routine EXTRACT-MIN and place the numbers extracted in sequence in an array. This sequence is sorted since the current minimum is returned and placed. Each call to EXTRACT-MIN takes $O(n)$ time. Hence, the time taken is $n^2 \times O(n) = O(n^3)$.

4. Given an $O(m + n)$-time algorithm to determine whether a given number is stored in a given $m \times n$ tableau.

   Solution. . A simple solution idea going column-wise is as follows. A symmetric row-wise solution can also be formulated.

   Suppose we are searching for the value v. Start the search at position $A[m, 1]$ (left bottom) and go up sequentially along the first column. Let i be the largest row index such that $A[i, 1] \le v$. So if $A[i, i] = v$, then the search ends. Otherwise, suppose $A[i, 1] < v$. All elements in the submatrix $A[i + 1 \ldots m, 1 \ldots n]$ are at least as large as $A[i, 1]$, and so can be eliminated from the search. Thus, if v exists it must be in the sub-matrix $A[1 \ldots i, 2 \ldots n]$. We have now reduced the problem by one column and by $m - i$ rows. Thus, if $T(m, n)$ is the time for searching in the $m \times n$ tableau, then, we have,

   $$T(m, n) = \Theta(m - i) + T(i, n - 1) \ .$$

   Unrolling, the solution to this is $O(m + n)$.

   Another way to look at this is that we will repeat the procedure after the first column search, starting at row i of column 2. We repeat, searching for the largest row index starting from i backwards such that $A[i_1, 2] \le v$. If equality holds we are done, otherwise, the value v if it exists, lies in the sub-matrix $A[1 \ldots i_1, 3 \ldots n]$. For simplicity, Let $i_0 = m + 1$. At iteration

j, we start at index i_{j-1} and follow backwards in column j to find the largest index i_j such that $A[i_j, j] \leq v$. This takes time $O(i_{j-1} - i_j)$. If the item is not found, then the iteration stops at an index j when $i_j = 1$. At this time, we can complete a sequential search through the remainder of row 1, that is $A[1, j \ldots n]$, which takes time $O(n - j + 1) = O(n)$. The total time complexity is therefore

$$O(\sum_{j=0}^{J}(i_{j-1} - i_j)) + O(n) = O(i_0 - i_J) + O(n) = O(m) + O(n)$$

since, $i_0 = m$ and i_J is 0 or 1.

SEARCH(A, m, n, v)

$j = 1$ // j is current column index
$i = m$ // i is current row index
in_stairs = TRUE
found = FALSE
while *in_stairs* **and** **not** *found*
    while ($i \geq 1$ **and** $A[i, j] > v$) $i = i - 1$
    if $i == 0$ *in_stairs* = FALSE
    elseif ($A[i, j] == v$) *found* = TRUE
    else $j = j + 1$
        if $j > n$ *in_stairs* = FALSE
if *found* **return** (i, j)
else return NOTFOUND

Problem 3. *Selection.* Consider a generalized version of the selection problem where, given an array $A[1 \ldots n]$ of numbers and a number k, we wish to return the first to the kth smallest elements in *some* order, not necessarily in increasing order. That is, we want to return the smallest element, the second smallest element, ..., the kth smallest element, and these elements only, in some order. Give an efficient algorithm with expected time $O(n)$ for this problem, that is, without fully sorting the array. Give an analysis of your algorithm. (25 points)

Solution. Use the SELECT procedure that calls RANDOMIZED_PARTITION as done in the class and has expected time complexity $O(n)$. The procedure below makes a call to SELECT, taking expected $O(n)$ time, and one call to Partition, taking $\Theta(n)$ time, for a total expected time $O(n)$.

SELECTFIRSTK(A, k)

1. j = SELECT(A, k)
2. (m, q) = PARTITION$(A, 1, n, A[j])$ // Partition $A[1 \ldots n]$ around pivot $A[l]$.
3. Print $A[1 \ldots k]$

First, the algorithm finds the index j of the kth smallest element (line 1). Now call PARTITION on $A[1 \ldots n]$ using this element as the pivot. Since it is the kth smallest element, after this call, the first k elements are the k smallest elements. (Note that this is true even if there are repeated elements who are equally kth smallest).

Solution. [Partial Alternative.] Using Min-Heaps:

SelectFirstK(A, k) // returns the k smallest elements in A

1. Let $S[1 \ldots k]$ be an array
2. Build-Heap(A, n) // Transform A into a heap
3. **for** $i = 1$ **to** k
4. $\quad$ $S[i] =$ExtractMin(A)

Time Complexity. Note that the time taken by Build-Heap is $O(n)$, followed by k calls to $ExtractMin$ which takes $O(\log n)$ each. So total time is $O(n + k \log n)$, which for small values of k is $O(n)$. There is no randomization and is not a general solution either as it works only for small k, $k = O(n/\log n)$.