ESO207A: Data Structures and Algorithms	
Homework *2: Heaps, Stacks*	HW Due Date: Sep 7, 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. (5)

   Solution. As discussed in the text, the procedure *Max-Heap-Insert*(A, k, v) takes time $O(\log k)$, where, k is the current heapsize. Thus, the time taken is as follows.

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

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. Let $A = \{1, 2, 3, \ldots, n\}$ in increasing sorted order. Consider the state when the call *Max-Heap-Insert*$(A, heapsize, k)$ is made, with $heapsize = k - 1$. 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,

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

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 $A[1,1]$ which is remembered and finally returned. The spot $A[1,1]$ is vacant and is replaced by the minimum of $A[2,1]$ and $A[1,2]$, its below and right neighbors. This creates a vacant spot, which is propagated by continuing the algorithm. Assume that for any finite number a, $a < \infty$ and $\infty \leq \infty$.

   DOWN(i,j) { **return** $(i, j+1)$ }
   RIGHT(i,j) { **return** $(i+1, j)$ }
   EXTRACT-MIN(A, m, n)
   1. $min = A[1,1]$
   2. $(x,y) = (1,1)$
   3. $done =$ FALSE
   4. **while not** *done*
   5. $\quad (mx, my) = undefined$
   6. // (mx, my) is the index of the minimum of $A[\text{RIGHT}(x,y)]$ and $A[\text{DOWN}(x,y)]$
   7. // some corner cases are checked to ensure that RIGHT(x,y) is a legal entry
   8. // and if DOWN(x,y) is a legal entry. Their minimum (or whichever exists) is taken
   9. // and is exchanged with $A[x,y]$. (x,y) is set to the index of this minimum,
   10. // and iterate.
   11. $\quad$ **if** DOWN$(i,j) \leq (m,n)$ $(mx, my) =$ DOWN(x,y)
   12. $\quad$ **if** RIGHT$(i,j) \leq (m,n)$
   13. $\quad\quad$ **if** $(mx, my) ==$ DOWN(x,y)
   14. $\quad\quad\quad$ **if** $A[mx, my] > A[\text{RIGHT}(x,y)]$ $(mx, my) =$RIGHT(x,y)
   15. $\quad\quad$ **else** $(mx, my) =$ RIGHT(x,y)
   16. $\quad$ **if** $(mx, my) ==$ UNDEFINED $done =$ TRUE
   17. $\quad$ **else** *swap* $A[mx, my]$ with $A[x,y]$
   18. $\quad\quad$ $(x,y) = (mx, my)$
   19. 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 current 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 $A[m \times n]$
   1. $A[m,n] = v$
   2. $(x,y) = (m,n)$

```
3.   done =FALSE
4.   while not done
5.        (mx, my) = (x, y)
6.        if LEFT(x, y) ≥ (1, 1) and A[LEFT(x, y)] > A[x, y]
7.             (mx, my) =LEFT(x, y)
8.        if  UP(x, y) ≥ (1, 1) and  A[UP(x, y)] > A[mx, my]
9.             (mx, my) =UP(x, y)
10.       if (mx, my) == (x, y) done = TRUE
11.       else swap A[x, y] with A[mx, my]
12.            (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 $A[m \times n]$ (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 $A[i, 1] \leq v$. 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 $A[i_1, 2] \leq v$. 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)
1.  j = 1          // j is current column index
2.  i = m          // i is current row index
3.  in_stairs = TRUE
4.  found = FALSE
5.  while in_stairs and not found
6.        while ( i ≥ 1 and A[i, j] > v) i = i − 1
7.        if i == 0 in_stairs = FALSE
8.        elseif (A[i, j] == v) found = TRUE
9.        else j = j + 1
10.               if j > n in_stairs = FALSE
11. if found return (i, j)
12. else return NOTFOUND
```

Problem 3. *Stack: a span problem.* You are given an array $P[1, \ldots, n]$ giving the daily price quotes for a stock for n consecutive days. The span of the stock's price on a given day i is the maximum number of consecutive days that the stock's price is less than or equal to its price on day i. (This includes day i, so span is at least 1). For example, suppose P is the array $\{50, 45, 35, 40, 60, 50, 55\}$. Then, the span array for these 7 consecutive days is $\{1, 1, 1, 2, 5, 1, 2\}$. An $O(n^2)$ algorithm follows in a straightforward way from the definition. Design an $O(n)$ (linear-time) algorithm. Analyze your algorithm.

(*Hint*: For $i = 1, 2, \ldots, n$, let $h(i)$ be the highest index less than or equal to i such $P[h(i)] > P[i]$, if such an index exists, otherwise, it is 0. The span for index i is $i - h(i)$. Store $i, h(i), h(h(i)), \ldots$ downwards in the stack. Note that this constitutes the unique ascending sequence $P[i] < P[h(i)] < P[h(h(i))] < \ldots$. The economy of computation is obtained as follows. Given $P[i + 1]$, if $P[i] > P[i+1]$, then, $h(i+1) = i$, but if $P[i] \leq P[i+1]$, then, we need only compare $P[i+1]$ with $P[h(i)]$, since, all elements j such that $h(i) < j \leq i$ have $P[j] \leq P[i]$. So values at indices $i - 1$ down to $h(i) + 1$ need not be stored or compared with, as they are clearly no larger than $P[i]$. So on, if $P[h(i)] \leq P[i + 1]$, then, consider $P[h(h(i))]$, and so on.)

Solution. We will follow the idea outlined in the hint. We will keep a stack, that, when we have finished processing the value at index i, has $i, h(i), h(h(i)), \ldots,$ downwards in the stack, where, $h(i)$ is as defined in the Hint paragraph. We will maintain this invariant for the problem. In particular, this is the unique ascending sequence of values, that is, $P[i] < P[h(i)] < P[h(h(i))] <$.

Now suppose we wish to maintain the invariant for index $i+1$. If $P[i] > P[i+1]$, then, $h(i+1) = i$. We can push $i + 1$ onto the stack and return 1. Otherwise, $P[i] \leq P[i + 1]$ and we can pop i from

the stack. The closest index to i that is larger than $P[i]$ is $h(i)$, so we check $P[i+1]$ versus $h(i)$ which, after the pop operation, is the current top element of the stack. If $P[i+1] < P[h(i)]$, then, $h(i+1) = h(i)$ and we can push $i+1$ onto the stack and return $i+1-h(i)$. Otherwise, we continue the above in a loop. The loop also terminates when the stack becomes empty, which happens when $P[i+1]$ is the current largest element among $P[1, \ldots, i+1]$. If this happens, we insert $i+1$ into the top of the now empty stack.

```
procedure ComputeSpan(P, n) // P[1, ..., n] is the given array of values
1.   Stack S            // S is a stack
2.   N[1, ..., n] is the span array to be computed
3.   S.MakeEmpty()     // Make S to be empty stack
4.   for i = 1 to n
5.        while not S.IsEmpty() and  P[S.Top()] < P[i]
6.             S.Pop()
7.        if  S.IsEmpty()   N[i] = i
8.        else  N[i] = i - S.Top()
9.        S.Push(i)
```

Time complexity. Each item is pushed exactly once in the code, after some constant amount of work (lines 7-9). It may therefore be popped at most once (or may be not). For each item popped, there is a constant amount of work done, namely the boolean condition checking in the while loop of line 6. Thus, the total time is $\Theta(1) \times n = \Theta(n)$.