Problem 1: Python file submitted

Problem 2: Purpose: Often, recursive function calls use up precious stack space and might lead to stack overflow errors. Tail call optimization is one method to avoid this problem by replacing certain recursive calls with an iterative control structure. Learn how this technique can be applied to QUICKSORT. (2 points each) Solve Problem 7-4, a-c on page 188 of our textbook.

Solution Problem 2:

a. Argue that TAIL-RECURSIVE-QUICKSORT(A, 1, A.length) correctly sorts the array A.
   - ➔ Induction can be used to prove this:
   1) Base Case: The array(A) has only one element, which makes it already sorted(p=r).
   2) Inductive Step: Assuming TAIL-RECURSIVE-QUICKSORT sorts an array(A) containing k elements for 1<=k<=n. Within the first iteration TAIL-RECURSIVE-QUICKSORT partitions the elements in A[p..r] and q as the pivot, first the left subarray i.e. A[1..q] is sorted because the left subarray definitely has less elements than n+1. Then, the right side of the array A[q+1..n] is sorted as it is of smaller size where p is updated p = q+1 and it is correctly sorted by induction hypothesis.

      The base case and inductive step, both are performed by Induction and hence TAIL-RECURSIVE-QUICKSORT (A,1,A.length) correctly sorts the array A.

b. Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is Θ(n) on an n-element input array.
   - ➔ The stack depth will be Θ(n) if the input array is already sorted. The right subarray will always have size 0 so before the while-condition is p < r is violated there will be (n-1) recursive calls.

c. Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is Θ(lg n). Maintain the O(n lg n) expected running time of the algorithm

   - ➔ To get Θ(lg n) as the stack depth, we can design the algorithm such that recursive call is always made on the smaller subproblem. At each recursion, the range of the array will reduce at least by half. Since the array size is reduced by half in each recursive call, the stack depth, is in the worst case O(lg n).

```
MODIFIED-TAIL_RECURSIVE-QUICKSORT(A,p,r)
while p < r
    q = PARTITION(A,p,r)
    if q < floor((r-p)/2)
            MODIFIED-TAIL-RECURSIVE-QUICKSORT (A,p,q-1)
            p = q+1
    else
            MODIFIED_QUICKSORT(A,q+1,r)
            r = q -1
```

Problem3: (9 points) Purpose: Practice algorithm design and the use of data structures. This problem was an interview question! To avoid deductions, please follow Eric Demaine's instructions about how to "give/describe" an algorithm. Consider a situation where your data is almost sorted—for example you are receiving time-stamped stock quotes and earlier quotes may arrive after later quotes because of differences in server loads and network traffic routes. Focus only on the time-stamps. Assume that each time-stamp is an integer, all time-stamps are different, and for any two time-stamps, the earlier time-stamp corresponds to a smaller integer than the later time-stamp. The time-stamps arrive in a stream that is too large to be kept in memory completely. The time-stamps in the stream are not in their correct order, but you know that every time-stamp (integer) in the stream is at most hundred positions away from its correctly sorted position. Design an algorithm that outputs the time-stamps in the correct order and uses only a constant amount of storage, i.e., the memory used should be independent of the number of time-stamps processed. Solve the problem using a heap.

Solution Problem 3 –

We are going to implement a priority queue and a min heap since the output has to be reported in increasing order. Consider, E as the total number of elements in the priority queue. Since every time stamp is located at most hundred steps away from its correctly sorted position when E<100 we do not extract any element but as soon as E>=100 we do EXTRACT_MIN_HEAP(E). So, the array size does not go beyond 101 and only a constant amount of storage is used.

Proof of Correctness:

Initialization: At the start, there is no element inserted in minimum heap.

Maintenance: If an element X in the output is smallest in the array of 101 elements in the priority queue S. If in future element Y which is smaller than X arrives, in this case Y is smaller than the 101 elements in the priority queue, hence it is more than 100 positions off its correct position. This calls for a contradiction to our assumption that no element is more than 100 positions away from its correct position. Whenever X is returned, it is the smallest elements in the remaining input stream. Hence X is in correct positions.

Termination: At the end, last 101 elements are larger than all previously reported elements. The elements are sorted by min-heap-sort and reported in increasing order. This guarantees that all elements are reported in their correct order.

Algorithm:

```
counter = 0
val = True
while val:
    if (time stamps arriving):
        Insert_heap(E, time_stamp)
        counter +=1
        If counter > 100:
            Extract_heap(E)
            counter -=1
    if (time stamps arrival finished):
        while(counter > 0):
            Extact_heap(E)
            counter -=1
    val = False
```

Complexity: All operations are insert and extract in minimum heap the time complexity is O(1).

Example:

Maximum element size = 2

Input stream = 20, 25, 10, 60, 80

Stream Input	Priority Queue, S	Output
20	20	-
20,25	20,25	-
20,25,10	10,20,25	10
20,25,10,60	20,25,60	10,20
20,25,10,60,80	25,60,80	10,20,25
-	60,80	10,20,25,60
-	80	10,20,25,60,80

4. Purpose: practice algorithm design, and dynamic programming. Consider a sequence of n distinct integers S[1], …, S[n]. The longest nondecreasing subsequence problem asks you to find a longest sequence (i1,…,ik) such that ij <i{j+1} and S[ij] ≤S[i{j+1}] for j ϵ{1,…,k1}. For example, consider the sequence: 3, 45, 23, 9, 3, 99, 108, 76, 12, 77, 16, 18, 4. A longest nondecreasing subsequence is 3, 3, 12, 16, 18, having length 5.
a) (6 points) Let S[1], …S[n] be a sequence of n integers. Denote li the length of a longest nondecreasing subsequence that ends in (and includes) S[i]. Use dynamic programming to compute li. Describe the algorithm and give an (additional) algorithm that generates the corresponding nondecreasing subsequence.
b) (1 point) Using l1,…,ln, how do you solve the longest nondecreasing subsequence problem?
c) (3 points) An alternative approach: Describe an algorithm that use the Longest Common Subsequence (LCS) Problem that we have discussed in class to solve the longest nondecreasing subsequence problem.
Solution 4:
a and c) Since, Dynamic Programming should be used, let us try to find out the subproblems involved. At index 0 the length of the subsequence is always 1, this is true for all positions. Therefore, we initialise the Length of the subsequence for all i at 1.
Therefore,
Array = S[1]…S[n]
Length of the longest non- decreasing subsequence (l)= [l1….ln] = [1,1,1,1…n]
Example:
For simplicity, let us assume an array [1,4,5,6,3,3,3,3]
We initiate the Length at - [1,1,1,1,1,1,1,1]
n = length of array S = 8
Iterative steps:

```
   j         i
S[2] >= S[1]    l[2] = max( l[2], l[1]+1) = 2

S[3] >= S[1]    l[3] = max( l[3], l[1]+1) = 2

S[3] >= S[2]    l[3] = max( l[3], l[2]+1) = 3

S[4] >= S[1]    l[4] = max( l[4], l[1]+1) = 2

S[4] >= S[2]    l[4] = max( l[4], l[2]+1) = 3

S[4] >= S[3]    l[4] = max( l[4], l[3]+1) = 4

S[5] >= S[1]    l[5] = max( l[5], l[1]+1) = 2

S[5] < S[2]    l[5] = No change …. Same no change till i = 4

S[6] >= S[1]    l[6] = max( l[6], l[1]+1) = 2

S[6] < S[2]    l[6] = No change …. Same no change till i = 4

S[6] >= S[5]    l[6] = max( l[6], l[5]+1) = 3
Similarly
S[7] >= S[6]    l[7] = max( l[7], l[6]+1) = 4
And,
```

S[8] >= S[7]   LNDS[8] = max(LNDS[8], LNDS[7]+1) = 5
So the final Length of the non-decreasing subsequence array is {1,2,3,4,2,3,4,5}, and the maximum length is 5.

So the algorithm goes like this,

```
1.  n = length(S)
2.  l = [1]*n
3.  for i ← 1 to n:
4.     for j ←  0 to i:
5.        If S[i] >= S[j] and l[i] < l[j] +1:
6.           l[i] = l[j]+1
7.  return max(l)             #return maximum of array l, which is the length of max subsequence
```

Modifications to return the subsequence:

```
1.  n = length(S)
2.  l = [1]*n
3.  new = [-1]*n              #initializing an array to store index values on longest subsequence
4.  for i ← 1 to n:
5.     for j ←  0 to i:
6.        if S[i] >= S[j] and l[i] < l[j] +1:
7.           l[i] = l[j]+1
8.           new[i] = j
9.  final_len = max(l)+1
10. final_arr = []                  #final_arr stores the longest subsequence
11. while final_len >=0:
12.    final_arr.append(arr[final_arr])
13.    final_len = new[final_len]     #start the iteration from the
14. final_arr.sort()    #sort by any algorithm which we learnt in class – for ex. merge sort etc.
15. return max(l), final_arr    #returns the length of the longest subsequence and the longest
    subsequence.
```

Complexity: The time complexity for this would be O(n^2)

Proof of Correctness:
Initialization: Initially, an array with 0 elements is already sorted.
Maintenance: Every j element is checked with all the i elements which are prior to it. If the j element is larger than or equal to i, the respective length is increased by 1. Similarly, if j is less than i, the length is kept the same. So, to generalize S[j] >= S[i]   LNDS[j] = max(LNDS[j], LNDS[i]+1) and S[j] < S[i]   LNDS[j] = No change. Incrementing i and j re-establishes the loop invariant for the next iteration.
Termination: At termination, final final_arr returns the array is non-decreasing order and also the maximum length of the subsequence.

b) As l1…ln stores the length of longest subsequence, we store the index position from the original array w.r.t l1 in a new array called "new". "new" is initialized with all elements as -1 so that we don't interfere with index positions from the original array. Start a while loop from "ln+1" = which is maximum length +1. Reduce it by calling "new". This fills out our final array which is the longest subsequence.