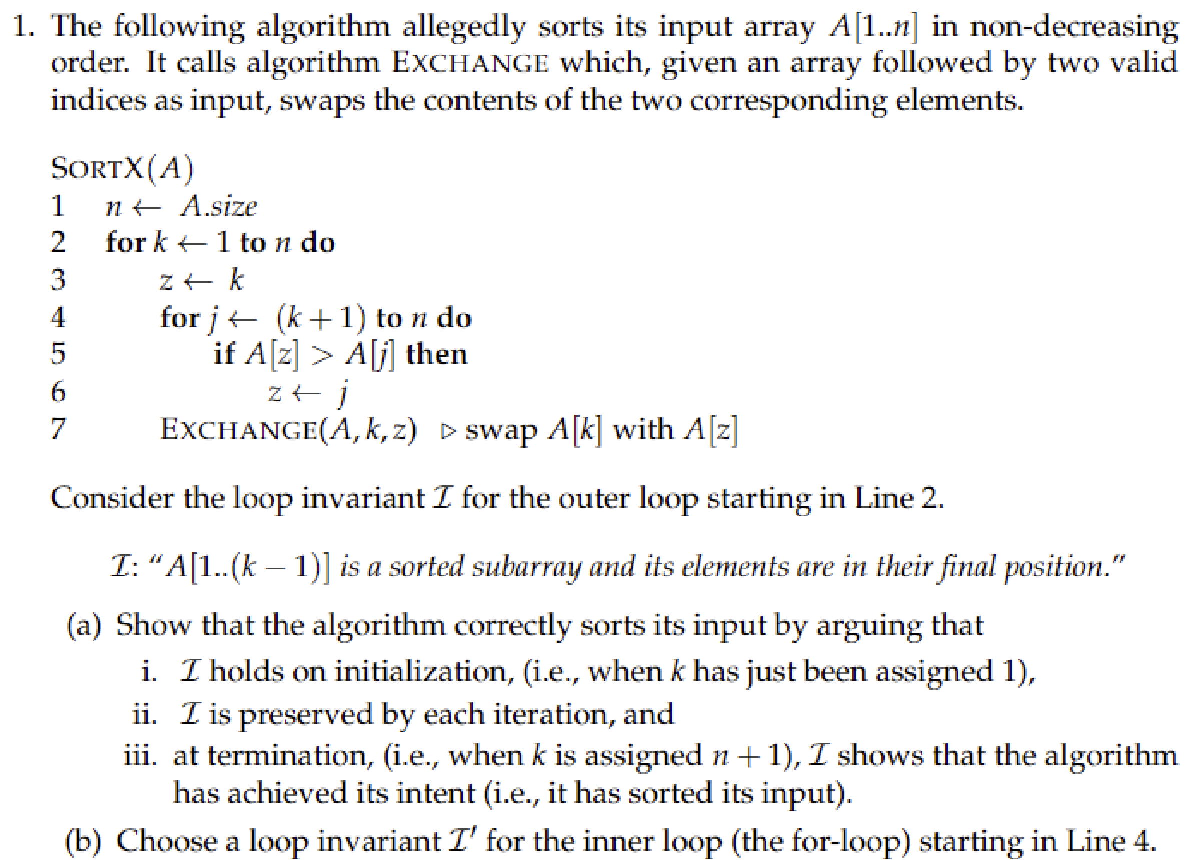
Julian Garcia
Hw2

I certify that every answer in this assignment is the result of my own work; that I have neither copied off the Internet nor from any one else's work; and I have not shared my answers or attempts at answers with anyone else.



1. The following algorithm allegedly sorts its input array $A[1..n]$ in non-decreasing order. It calls algorithm EXCHANGE which, given an array followed by two valid indices as input, swaps the contents of the two corresponding elements.

```
SORTX($A$)
1  $n \leftarrow A.size$
2  for $k \leftarrow 1$ to $n$ do
3      $z \leftarrow k$
4      for $j \leftarrow (k+1)$ to $n$ do
5          if $A[z] > A[j]$ then
6              $z \leftarrow j$
7      EXCHANGE($A, k, z$)  ▷ swap $A[k]$ with $A[z]$
```

Consider the loop invariant $\mathcal{I}$ for the outer loop starting in Line 2.

$\mathcal{I}$: "*$A[1..(k-1)]$ is a sorted subarray and its elements are in their final position.*"

(a) Show that the algorithm correctly sorts its input by arguing that

i. $\mathcal{I}$ holds on initialization, (i.e., when k has just been assigned 1),

ii. $\mathcal{I}$ is preserved by each iteration, and

iii. at termination, (i.e., when k is assigned $n+1$), $\mathcal{I}$ shows that the algorithm has achieved its intent (i.e., it has sorted its input).

(b) Choose a loop invariant $\mathcal{I}'$ for the inner loop (the for-loop) starting in Line 4.

(a) i. On initialization, since k = 1, the array is A[1 … 0], which means this ranges over an empty set, which implies that the set is technically sorted since it is empty and thus our Invariant holds on initialization.

ii. To see that after each iteration the invariant holds, observe how z is assigned the value of k and j is assigned the value of j+1, then the comparison 'if A[z] > A[j]' is run, which is really checking to see if A[k] is greater than A[k+1], if A[k] is greater than A[k+1], the values of the two are switched using EXCHANGE, this continues up to n, thus ensuring that A[k] is always LESS THAN each value in the subarray A[(k+1) … n]. Which means that the next iteration and each iteration after that holds true to the invariant that A[1… (k - 1)] is a sorted subarray.

iii. At termination, since k=n+1, by the loop invariant the array is A[1 … (n + 1 - 1)] = A[1 … n] Since the length of the array is of size n, and the invariant assures that all values of the array from 1 to n are sorted, the invariant has achieved the intent of sorting the entire array.

(b) I_2: "A[k] is less than or equal to each value in the subarray A[(k + 1) … n]"

2. Design an algorithm K-WAY-MERGE that makes use of a heap to merge k sorted non-empty arrays $A_1, \ldots, A_k$, of possibly unequal lengths, into one array B of length $n = A_1.length + \ldots + A_k.length$. Your algorithm's worst-case time complexity must be $O(n \lg k)$.

   (a) Either write pseudo-code or present an outline of your algorithm clearly in English in numbered steps. You will lose points if your grader does not understand your strategy.

   (b) Next, argue why the desired complexity is attained.

(a) To make this work, we need two heap algorithms for insertion and deletion:

For ***insertion*** we can follow these steps:
First we insert a node containing the insertion value in the "farthest left location" of the lowest level of the heap

Then we filter the inserted node up using this algorithm:

```
while (  inserted node's value  <  parent's node value )
{
  swap the values of the respective node;
}
```

For ***deletion*** we can follow these steps:
First copy over the roots value to the end of our array B
Then we copy the last value in the array to the root
We then decrease the heap's size by 1
We then sort down root's value. Sorting is done as following:
if the current node has no children, sorting is over;
if the current node has one child, we run a check, if heap property is broken, then we swap current node's value and child value, then we move down the child;
If the current node has two children, we find the smallest of them. If the heap property is broken, we then swap current node's value and selected child value; move down the child.

To make this work, we need to store the smallest element in each of the lists in a heap, and each element you store in that heap should be changed by the index of the list it comes from (perhaps add index to value after storing into heap). Then we can perform our delete min algorithm on the heap and insert the next element from the deleted element's corresponding list. After going through each value of every list and performing enough delete min's the resulting array B will be sorted and contain every value of all k arrays.

(b) Going through each step:
To build the heap, the algorithm would take O(k)
It would take the algorithm O(lg k) to perform DeleteMinimum on every element.
It would take the algorithm O(lg k) to insert the next element from the same list.
So since there are n total elements, it would take O(nlgk + k) = O(nlgk) time to perform this algorithm.