Assignment 2 Problem 3

Consider a method *step-up-down* that given an array A of n non-repeating numbers, rearranges A into blocks of varying size: the first block is size 2, the second is size 3, and so on. Each block also orders the numbers it contains in the follow way: the first block contains numbers in decreasing order, the second block is in increasing order, the third block is in decreasing order, and so on. Adjacent blocks also have the property that the numbers in a even numbered block must all be smaller than the numbers in the immediately preceding odd numbered block.

For example, given

$$A : [8, 13, 2, 7, 10, 4, 9, 3, 1]$$

the output of *step-up-down* could be

$$A : [13, 9, 1, 2, 3, 10, 8, 7, 4]$$

Another possibly is

$$A : [13, 10, 1, 2, 3, 9, 8, 7, 4]$$

Note: Both outputs above are valid and for a given input there may be more than one valid output. Your algorithm needs to only output one valid output - any valid output is okay.

You are given two heaps (already implemented), one min-heap and one max-heap, that you may use in your implementation. You may use the heap methods from class without restating them here. The two heaps are independent (i.e. not linked) and you do not have access to their implementations.

Aside from A and the two heaps, you may only use a constant amount of space.

Give pseudocode for an efficient implementation of *step-up-down*. Also, briefly state and justify the correctness of your algorithm and its time complexity.

Solution:
The pseudocode for an implementation of step-up-down:

```
step-up-down(A)
A: The given array from input
MA: The given max-heap
MI: The given min-heap
n ← size of array
pre-len ← 0 // pre-len stores the legnth of the array that is already a part of output
current-block ← min(2,n) // current-block stores the length of the current block
nxt-len ← min(2,n) // next-len stores the length output before we sort the current block
i ← 0 while i < n do
```

```
if current-block even
    MA.insert(A[i])
if current-block odd
    MI.insert(A[i])
if i euqals nxt-len - 1
    if current-block even
        j ← pre-len up to i do
            A[j] ← MA.deleteMax()
    else // current-block odd
        j ← pre-len up to i do
            A[j] ← MI.deleteMin()
    pre-len ← i+1
    current-block increased by 1
    nxt-len ← min(n,nxt-len + current-block)
i increased by 1 // end loop
```

What this algorithm do is just look through the whole array once, and during this process, doing the following things: first identify which block are we current in, is it a increasing one or a decreasing one. Then insert the current entry into the correct heap. After that, we do a judgement, if after the current insert, we reach the target length of what we want to sort it now(ie. i = nxt-len), then we sort the current block using the heap, by deleting the max of heap. After we sort the current block, the heap is also cleared and waiting for the next time calling. This guarantee that we get the output which we want.

For the time complexity, since we go through the array one time, therefore, it must be n times something. For the inner part, we focus on each element, we firstly insert it into the heap, and then delete it somewhere else(in the alter iteration), and do nothing else. From question 2, the number of blocks is similar to the number of levels in pyramid, therefore, we know that number of block should be $\Theta(\sqrt{n})$. And the length of the longest block should also be $\Theta(\sqrt{n})$. And we know that from lecture, the time complexity for insert and deleteMax are both $O(\log n)$.

There fore, the total complexity for this algorithm should be $O(n \log \sqrt{n})$.