CS566 HW3 | Jiankun Dong

(1) Idea: use a sorting algorithm O(nlogn) to first sort the array. then peak max and min elements in the array (O(n)).

Code:

```
def findPair(intArr):
    intArr.sort() // sort array in O(nlogn)
    result = []
    i = 0
    j = len(intArr) - 1
    while (i <= j):
        curr = [intArr[i], intArr[j]]
        result.append(curr)
        i += 1
        j -= 1
    return result.
```

(2) ① Doesn't matter.
For max-heap, the root is the max element.
For sorted array, either first or last (deferedly on how you sort it) is the max element. Both are O(1)

② Max-heap O(log n) because you just need to traverse the height of the heap.
It's better than array's O(n)

③ Max heap: Time to creat: O(nlogN)
Array: O(nlogn)
Doesn't matter.

④ Sorted array is better O(1).
max-heap takes way longer.

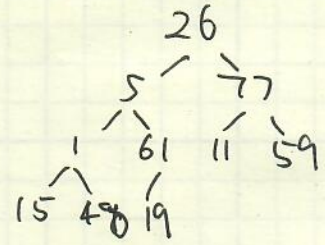
(3) See code.

Because merge sort is already spliting the array into segments.

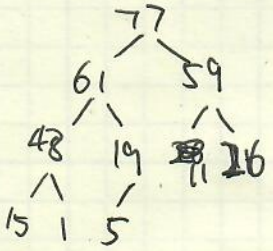
Heapsort and quicksort need to store all data in one place to sort the array.

(4) 26, 5, 77, 1, 61, 11, 59, 15, 48, 19.

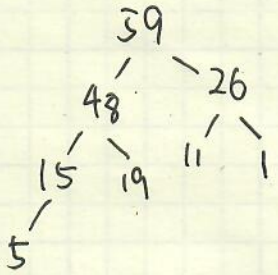
(a). Before heap:



(b) Initial heap:



(c)



new root is 59

(5) 72, 20, 73, 42, 11, 80, 39, 30, 100, 46, 88, 32, 21 (8 inversions)

(a) Initial swap result:

20, 42, 11, 39, 30, 46, 32, 21, 72, 73, 80, 100, 88 (5 inversions).

They are not inorder.

(b) 11, 20, 42, 39, 30, 46, 32, 21, 72, 73, 80, 100, 88 (5 inversions)

11 20 39 30 32 21 42 46 72 73 80 100 88. (3 inversions)

11 20 30 32 21 39 42 46 72 73 80 88 100 (1 inversion)

11 20 21 30 32 39 42 46 72 73 80 88 100 (0 inversions)

The initial ~~swap~~ sort reduced most inversions from 8 to 5.

For 1 swap & compare, because we swap the pivot at the end of each sort, the swapping of pivot should result in most inversion resolved.