Solution - 2

1.

(a) 91, 90, 15, 77, 60, 11, -2, 16, 21, 6

(b) 90, 77, 15, 21, 60, 11, -2, 16, 6

(c) 91, 90, 15, 21, 77, 11, -2, 16, 6, 60

2.

Loop Invariant: A[1….A.heap-size] satisfies the heap order property except possible A[i] is larger than its parent

Initialization: Prior to the first iteration,

- Assume that i is a leaf node. Then the trees rooted at LEFT(i) and RIGHT(i) are heaps containing 0 element, which are min heaps trivially.
- Assume that i an internal node. Since A is a min heap, the children of i will be min heaps by the property of min heaps.

Maintenance: Every iteration changes the value of i to its parent after swapping the parent's value with its own if the A[parent(i)] is greater than the value of A[i]. Therefore, before each iteration, the trees rooted at LEFT(i) and RIGHT(i) are min heaps.

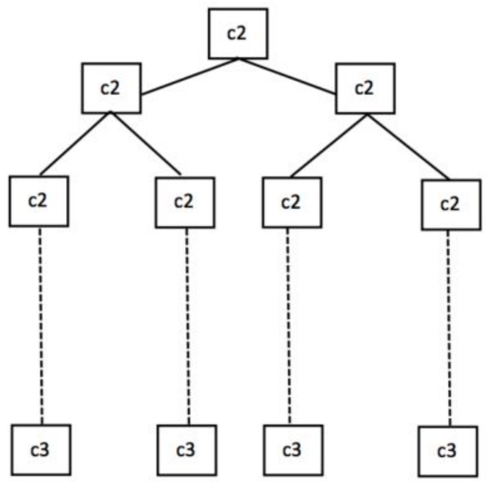
Termination: The loop terminated when either i = 1 or A[Parent(i)] <= A[i].

- When i = 1, the trees rooted at LEFT(i) and RIGHT(i) are min heaps and the previous swap puts the lower value to the root. This makes the trees rooted at LEFT(i) and RIGHT(i) to be min-heaps and A[i] to be lower than LEFT(i) and RIGHT(i). Therefore, A is a min heap.
- When A[Parent(i)] <= A[i], the trees rooted at LEFT(i) and RIGHT(i) are min heaps, and A as a whole is a min-heap.

3.

a. $T(n) = \begin{cases} c3 & n = 1 \\ c2 + 2T(n/2) & \text{otherwise} \end{cases}$

b.



# per level	cost
2^0	$2^0 . c2$
2^1	$2^1 . c2$
2^2	$2^2 . c2$
$2^{\log n}$	$2^{\log n} . c3$

Running time = c3(n) + c2(n)
= O(n)

4.

a) quad max-heap array representation:
   i) Parent(i) = $\lfloor (i - 2) / 4 \rfloor + 1$, where i > 1 or Parent(i) = $\lceil (i - 1) / 4 \rceil$ or $\lfloor (i + 2) / 4 \rfloor$
   ii) Children(i, j) = 4(i - 1) + j + 1, where $1 \leq i \leq$ heap_size and $1 \leq j \leq 4$

b) $\theta(\log_4 n)$ or $\lceil \log_4(3n + 1) \rceil - 1$

c)

```
HEAP_EXTRACT_MAX(A)
if A.heap_size < 1
   error "heap underflow"
max = A[1]
A[1] = A[A.heap_size]
A.heap_size = A.heap_size -1
MAX_HEAPIFY(A,1)
return max
```

```
MAX_HEAPIFY(A, i)
    largest = i
    for j = 1 to 4
        child = CHILDREN(i, j)
        if child ≤ A.heap_size and A[child] > A[largest]
            largest = child
    if largest ≠ i
        exchange A[i] with A[largest]
        MAX_HEAPIFY(A, largest)
```

The running time of HEAP-EXTRACT-MAX depends on the time taken by MAX-HEAPIFY. For MAX HEAPIFY to check all the child nodes, it takes O(4) time for a 4-ary tree and the total number of levels to check is $O(\log_4 n)$.

Therefore: Running time of HEAP-EXTRACT-MAX = $O(4\log_4 n) = O(\log_4 n)$.

d)

```
MAX_HEAP_INSERT(A, key)
    A.heap_size = A.heap_size + 1
    A[A.heap_size] = -∞
    HEAP_INCREASE_KEY(A, A.heap_size, key)
```

e)

```
HEAP_INCREASE_KEY(A, i, key)
    if key < A[i]
        error "new key is smaller than current key"
    A[i] = key
    while i > 1 and A[PARENT(i)] < A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

The run time of max heap insert is the same as that of a binary heap, as no extra comparisons are made. The run time of MAX-HEAP-INSERT depends on the running time of HEAP-INCREASE-KEY.

Therefore: Running time of MAX-HEAP-INSERT = O(height of the tree) = $O(\log_4 n)$

5. The question requires an algorithm that is (small) o(n^2) so we need to find an algorithm that is theta(n*log(g)) or theta(n)..and so on and not theta(n^2).

We can use merge sort to sort the array in ascending order theta(n*log(n)) and then remove equal elements adjacent to each other to get rid of duplicates in one pass theta(n). Making our algorithm - theta(n*log(n))

// **MergeSort(nums)**

```
int sol[] = new int[nums.length];
int i=0;
int index=0;
while(i<nums.length){

    while(i+1<nums.length && nums[i+1]==nums[i])
            i++;

    sol[index] = nums[i];
    index++;
    i++;
}
```

Note: 2 loops != O(n^2) while this code has two loops it is still one pass over the array making it theta(n)

6.

```
def  top_k(A):

        min_heap = build_heap(first k elements in A):

        for i = k + 1 to A.length:

                if min_heap[1] < A[i]:

                        min_heap.extract_min()

                        min_heap.heappush(A[i])

        return min_heap
```

7.

In this problem, we have k queues sorted by arrival time. We only have one gate, and only one player is allowed to enter at a time. We want the person with earliest arrival time to enter first.

This is essentially asking us to merge K sorted list. We can use a Min Heap to to store the heads of each list. Hence, we only need O(k) space and each person is processed in O(logk) time.

Pseudocode:

```
PROCESS_SHIPS(List<Line> lines) {
    result = [];        // initialize output list
    heads = BUILD_MIN_HEAP(lines);    // sort by arrival time, O(k log k)
    while heads is not empty:
        nextLine = EXTRACT_MIN_HEAP(heads);    // get min arrival time, O(log k)
        append(result, nextLine.ship);
        if (nextLine has more ships) {     // update the line
            INSERT_MIN_HEAP(heads, nextLine.next);
        }
    return result;
}
```

8.

The number of leaves of a nearly complete binary heap is $\lceil n/2 \rceil$

Basis step: Consider a binary heap of height h where $h = 0$.

When the height of a binary heap is 0, the number of nodes in the tree is 1.

∴ The number of leaves in such a tree is: $\left\lceil \frac{1}{2} \right\rceil = \left\lceil \frac{1}{2^{h+1}} \right\rceil = \left\lceil \frac{1}{2^{0+1}} \right\rceil = 1$

Inductive step: Let us assume that the above also holds true for nodes of height (h - 1). Suppose we have a binary heap with n nodes and we remove all the leaves of this new binary heap.

∴ The number of remaining nodes = $n - \left\lceil \frac{n}{2} \right\rceil = \left\lfloor \frac{n}{2} \right\rfloor$

By removing the leaf nodes, the height of the heap decreases by 1. Thus, the nodes with height initially h, now are of height (h - 1) in the modified heap.

By strong induction, the number of nodes with height (h - 1) is:

$$\left\lceil \frac{\left\lfloor \frac{n}{2} \right\rfloor}{2^{h-1+1}} \right\rceil \leq \left\lceil \frac{\frac{n}{2}}{2^{h}} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

9. https://leetcode.com/problems/find-median-from-data-stream/solution/

Approach 3 and 4 are accepted solutions as they fall within the complexity constraints