Algorithms -1
Tutorial 4
Solution Sketch

1. Show the min-heap that will be formed after inserting the elements 23, 1, 34, 4, 56, 77, 12, 33, 2, 12, 54, 2, 1, 34 one by one in an empty heap? Would the heap be the same if the elements are given at one time in an array and you use the Build-Heap procedure to construct a heap out of it?

   Routine

2. Given two arrays A1 and A2 (each of size n) of integers, design an O(nlgn) time algorithm for finding whether there exists a pair of elements, one from A1 and one from A2, such that their sum is some given integer x.

   Sort A2. For each element k in A1, do binary search for (x – k) in A2

3. Consider an array A of n integers such that $A[0] \leq A[1] \geq A[2] \leq A[3] \geq A[4] \leq A[5] \geq A[6] \leq \ldots$ Can you sort it in O(n) time?

   The answer is no. If we can do this, then this breaks the lower bound of comparison-based sort (see below).

   However, this problem has inadvertently been incorrectly specified. What we wanted to give is given an array of integers, can you create an array in O(n) time that satisfies the above constraints. Unfortunately, it got specified like this, sorry. There is an O(n) time algorithm for that (can you do it?).

   Then if there is an O(n) time algorithm for the above problem, given an arbitrary array, we can put it in the above form in O(n) time. Had it been possible to sort such an array in O(n) time, it will allow us to sort an arbitrary array in O(n) time, breaking the lower bound.

4. Given two heaps each with n elements, how can you merge them to create a single heap of 2n elements in O(n) time?

   Use an additional 2n-sized array. Put elements of first heap in the first n elements of the array, and elements of the second heap in the last n elements. Now build a heap out of the 2n elements in O(n) time. (This assumes that the heap is always given in array form. If the input is given in binary tree form, just do a traversal to get the elements into the array. What if the output is required in binary tree form?)

5. Suppose that you are given n integers in a min-heap (already formed), and an integer m. Design an algorithm to find all elements in the heap with value < m in O(k) time, where k is the number of elements in the hep that have value < m.

Do a preorder traversal of the heap. At each step, if the element looked at has value ≥ m, do not explore that subtree any further; otherwise print the element and continue the preorder traversal in that subtree

6. You are given k sorted arrays A_1, A_2,...,A_k. Design an algorithm to merge them in O(nlgk) time algorithm to merge the k sorted arrays into a single sorted array, where n is the total number of elements in all the arrays.

   Use a min-heap, First put the minimum elements from each array into the heap (along with their array id). Now repeat this n times:

   - extract the minimum from the heap and put it in the output array
   - If the minimum is from array A_k, insert the next element from A_k into the heap (if there are elements left in A_k; else do not insert anything)

7. You are given an integer n. You are also given at most n (x_i, v_i) pairs such that all x_i are distinct integers and $1 \leq x_i \leq n$. You need to design an ADT which supports the following operations:
   a. getMax(): Get the maximum value v among all the values present. Time : O(1)
   b. decreaseKey(x, v): Decrease the value of key x to v. You can assume v is always less than the current value of x. Time : O(log n)

   You are allowed to pre-process the data before the operations in O(n) time.

   Create a max-heap with the v values. Then getMax is easy. For decreaseKey, use the fact that the x values are distinct and between 1 to n. Specifically, keep an n-sized array A with A[i] storing the position in the heap of the value with x=i if present. Now, for decreaseKey(x,v), firts find position of x in the heap in O(1) time from A[x], then decrease that value and re-heapify in O(lg n) time.

8. Given an array of n integers, design an O(nlgn) algorithm to find the majority element (element that occurs more than n/2 time) in the array. Try two variations (i) if you are allowed to use sorting, and (ii) if you are not allowed to use sorting. Can you do it in O(n) time?

   The key is to note that if the majority exists, it must be the median.

   With sorting is easy. Sort, then take the median. Then count how many times the median occurs and check if it is the majority.

   Without sorting: See Pr. 9 below. There are other ways using Balanced BST also.

   O(n) time: Now that we know how to find the median in linear time, we can first find the median in O(n) time and then count how many times it appears in the array in another O(n) time to check if it is majority. However, since the median finding algorithm is not good in practice, this algorithm is not either. There is a beautiful simple algorithm for this:

- Initialize a variable candidate to first element of the array, with an associated count of 1.
- Now scan through the array from second to last, For each element, do the following:
    - If count = 0, candidate = this element
      Else If element = candidate, increment count
      Else If element not equal to candidate, decrement count.

If count not equal to 0 after scanning, the candidate variable contains a potential majority element. Just count how many times it occurs in the array in the array in another O(n) time.

The algorithm works because if candidate is truely a majority, and we cancel each occurrence of it by another different element, there will still be at least one occurrence left (not cancelled). On the other hand, if it is not a majority, its count can still be non-0 at the end (think of the array 12345, candidate = 5, count = 1 at the end but 5 is not a majority), so we count it once more to check.

9. Suppose that a stream of integers are coming one by one, and you have to find the median of the elements after inserting each integer in O(lgn) time. Design an algorithm to do this. You are not allowed to use any balanced BST.

   Use two heaps, a max-heap to store all elements to the left of the median and a min-heap to store all elements to the right. When a new integer comes, compare it with the roots to decide which heap it should go subject to the condition that the no. of elements in the two heap cannot differ by more than 1. To maintain this, if needed, transfer one element between heaps during insert (one more extractMin and Insert as part of the new integer insert). Give this condition, it is now easy to report the median (one of the two roots, depending on the size of the two heaps).

   Write a nice pseudocode for it.