6.1-1

What are the minimum and maximum numbers of elements in a heap of height h?

Solution:

For minimum, the number is 2^(h-1)

For maximum, the number is 2^h – 1

6.1-3

Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

Solution:

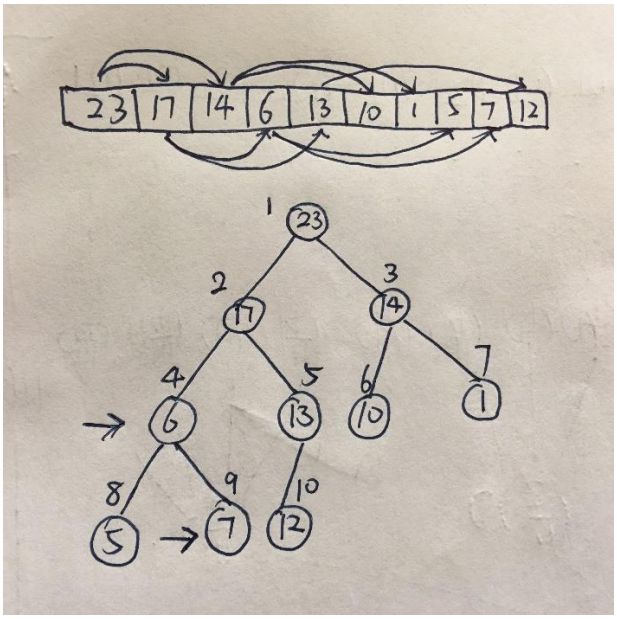
One of the property of max-heap is that all the values in the parent node is greater than its left sub node and right sub node, so for any subtree, its root's value is greater than that of its sub nodes, until the end of this subtree.

6.1-6

Is the array with values [23, 17, 14, 6, 13, 10, 1, 5, 7, 12] a max-heap?

Solution:

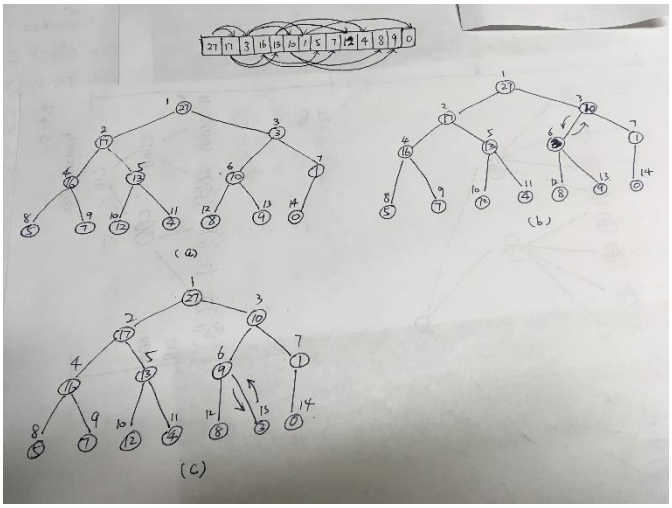
No, the array is not a max-heap, the element 6 and 7 are not in the right position, since 6 is the parent while it has smaller value than its child, which get value of 7



6.2-1

Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY(A, 3) on the array
A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0].

Solution:



6.2-2

Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY(A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY ?

Solution:

```
MIN-HEAPIFY(A, i)
1.  l = Left(i)
2.  r = Right(i)
3.  if l <= A.heap-size and A[l] < A[i]
4.      smallest = l
5.  else smallest = i
6.  if r <= A.heap-size and A[r] < A[i]
7.      smallest = r
8.  if smallest != i then
9.      exchange(A[smallest], A[i])
10.     MIN-HEAPIFY(A, smallest)
```

From the code we could see that the running time of MIN-HEAPIFY is the same as MAX-HEAPIFY, both are O(lgn).

6.2-4

What is the effect of calling MAX-HEAPIFY(A, i) for i > A.heap-size/2?

Solution:

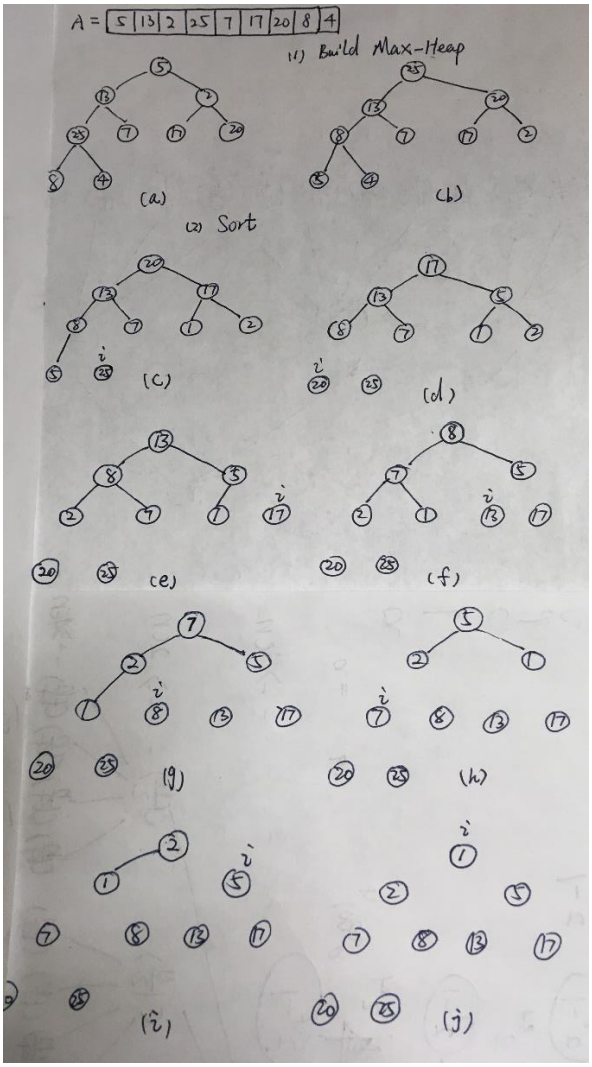
It will simply return since i > A.heap-size / 2 are all leaves, their left and right are nothing, so their will be no recursion at all on these i.

6.4-1

Using Figure 6.4 as a model, illustrate the operation of H EAPSORT on the array

A = [5, 13, 2, 25, 7, 17, 20, 8, 4].

Solution:



6.4-2

Argue the correctness of HEAP-SORT using the following loop invariant:

At the start of each iteration of the for loop of lines 2–5, the subarray A[1……i] is a max-heap containing the i smallest elements of A[1……n] and the subarray of A[i+1 …… n] contains the n - i largest elements of A[1……n], sorted.

Solution:

The argue is correct, since at the very first beginning we build the max-heap of A, and then exchanging the last unsorted item with the first one (which is the biggest in the unsorted part), after exchanging the last one becomes the largest one, then we decrease the heap size by 1. In order to maintain the max-heap property, the new largest item will be heapified to the first position, then we keep on exchanging.

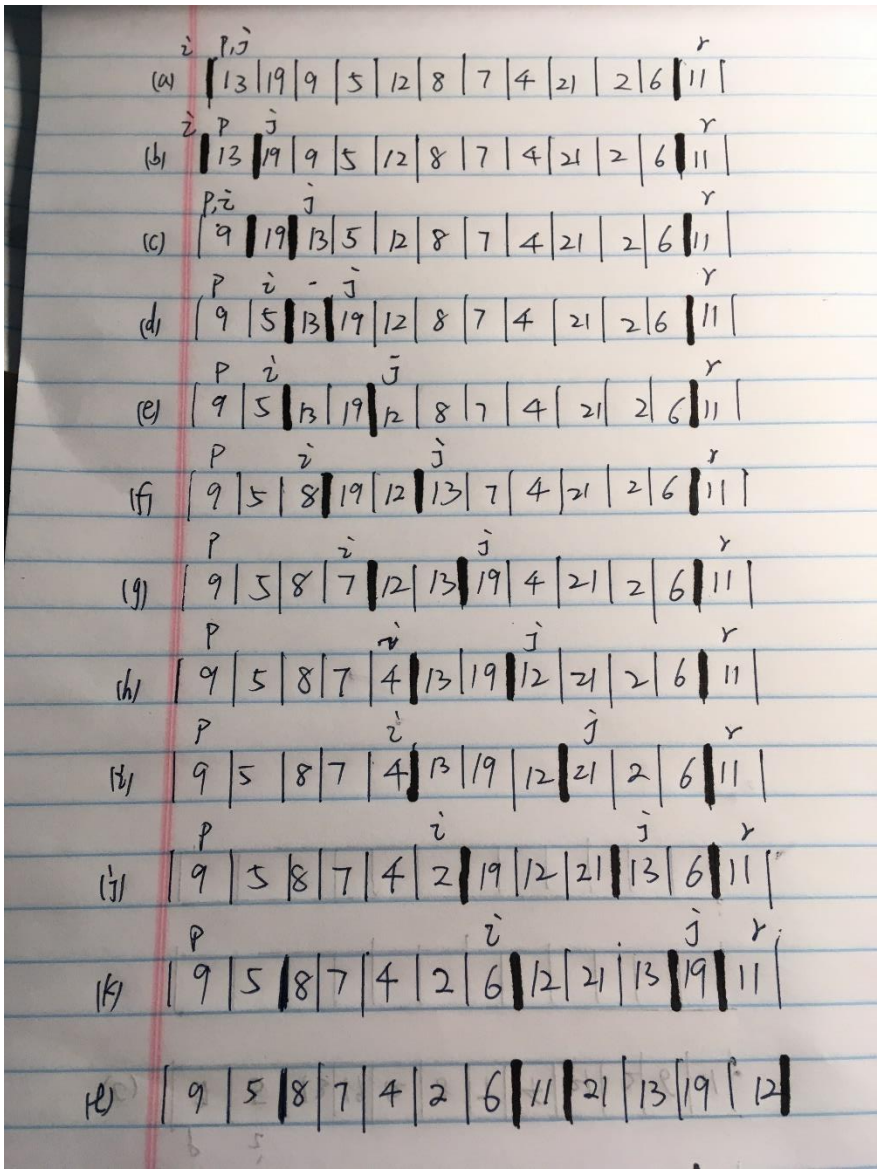
so all the items in sorted part are larger than those in the unsorted part, and every round, a largest item in

the unsorted part will be exchanged to the first place of the sorted part, the sorted part increase by 1, meanwhile the unsorted part decrease by 1, until the it down to 0 in the end, then all the items in the heap will be sorted.

7.1-1

Using Figure 7.1 as a model, illustrate the operation of P ARTITION on the array
A = [13, 19, 9 ,5,12, 8, 7, 4, 21, 2, 6, 11]

Solution:



7.1-3

Give a brief argument that the running time of PARTITION on a subarray of size n is θ(n).

Solution:

The partition is to move all the elements that are smaller than the pivot to the left side, and those are larger

than the pivot to the right side, then get the current pivot to its final position. In the whole process, we have to compare all the elements in the subarray with the pivot, which is for n-1 time, so the running time is θ(n).

7.1-4

How would you modify QUICKSORT to sort into nonincreasing order?

Solution:

The sort part is the same to that of nondecreasing order, the part we need to modify is the partitioning part as follows:

```
PARTITION (A, p, r)
1.  pivot = A[r]
2.  i = p – 1
3.  for j = 0 to r
4.        if A[j] >= pivot
5.            i= i+ 1
6.            exchange A[i] with A[r]
7.        end if
8.  end for
9.  exchange A[i + 1] with A[r]
10. return i + 1
```

7.2-1

Use the substitution method to prove that the recurrence T(n) = T(n-1) + θ(n) has the solution T(n) =θ(n^2) has claimed at the beginning of Section 7.2.

Solution:

Assume that T(n) = cn^2

Then T(n) = T(n-1) + θ(n)

= c(n-1)^2 + θ(n)

= cn^2 – 2cn + c + θ(n)

<= cn^2

In which c has a large constant value, which makes 2cn >= c+ θ(n).

7.2-3

Show that the running time of QUICKSORT is θ(n^2) when the array A contains distinct elements and is sorted in decreasing order.

Solution:

Assume that we take the last element as pivot, than the pivot is the largest one among the unsorted elements, It will split the array into two part, one length is 0 and the other is n-1, which could be written as the following:

T(n) = T(n-1) + T(0) + θ(n)

and we could also see that every following step, the pivot will be either the largest or the smallest one in the unsorted elements, so the above formula suits the whole process, and every step the problem scale decrease by 1, and the partition takes θ(n-1)) time, so totally θ(n^2).

7.2-4

Banks often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Argue that the procedure I NSERTION -S ORT would tend to beat the procedure Q UICKSORT on this problem.

Solution:

As the data being sorted becomes closer to sorted, insertion sort gets closer to θ(n) and quicksort gets closer to θ(n^2).

Therefore, there comes a point at which insertion sort performs better than quicksort. Further, while quicksort is still θ(nlgn) the constant multiple continues to grow as the data is more sorted. And with insertion sort, the constant multiple continues to shrink.