Problem 4-1

(a) REVERSE-PARTITION (A, p, q, r)

moves all elements greater that A[q] before index q i.e., from i = 1 to q-1 and all elements lesser after index q i.e., from i = q+1 to r and A[q] moves to last ~~elem~~ index A[r]

(i) Array A will become

A = [10, 5, 6, 4, 1, 9]

(ii)

A = [13, 10, 11, 12, 5, 6, 4, 1, 9]

(iii)

A = [14, 10, 11, 12, 13, 5 6, 4, 1, 9]

So, in the modified array A, the larger values than the pivot come before and then the smaller values comes and then at last the pivot itself.

(b)

FAST-QS (A, p, r)

1. if (p == r):
2. return
3. mid = (p+r)/2
4. REVERSE-PARTITION (A, p, mid, r)

5. Arrange the array A[p ... r-1] by exchanging the values inplace so that A becomes pivoted array
6. FAST-QS(A, p, mid)
7. FAST-QS (A, mid+1, q-1)

Correctness of algorithm -

Basic step of induction -

When array A is of size 1, it just returns the only element and the algorithm correctness is justified for this trivial case.

Induction hypothesis -

Let us assume that FAST-QS works correctly for A[1...K]
So, this means that the modified array A[1...K] produces balanced subproblem sizes on QUICK SORT

Induction Step:

Now, we need to prove the correctness of algorithm for A[1...K+1]

Now, since A[1...K] follows accordingly as required and we arrange the array in such a way that A is again converted to a pivoted array

So, in recursive terms the array again follows the algorithm and A[1...K] again gets modified so that it produces balanced subproblems for QUICKSORT.

Recurrence relation –

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$
$$T(1) = 1$$

Our array divides into 2 subarrays A[p ... mid] and A[mid+1 ... r]
REVERSE-PARTITION takes $\Theta(n)$ time.
Also, the arranging of the array (inplace) so as to make it pivoted takes $\Theta(n)$ time.

So, by case 2 of Master Theorem

$$T(n) = \Theta(n \lg n)$$

(c)

The resulting array of FAST-QS makes QUICKSORT run in $\Theta(n \lg n)$ time. This is so because in every divide step the array will be divided into 2 subarrays

of same size (differ in size by at most 1).

But this will not be the case for INSERTION-SORT also.

Now, time complexity for insertion sort is $\theta(n+I)$ where I is the no. of inversions in the array.

Now, in our modified array the maximum no. of inversions for any index i will be $\frac{n}{2}$

and the minimum no of inversions be 1.

So, $I = \frac{n}{2} + \left(\frac{n}{2} - 1\right) + \cdots 1$

$$I = \frac{\frac{n}{2}\left(\frac{n}{2}+1\right)}{2}$$

So, $I = \Theta(n^2)$

So, in this case the runtime of INSERTION SORT will be $\Theta(n + n^2) = \Theta(n^2)$

Now, $\Theta(n^2)$ is both $O(n^2)$ and $\Omega(n^2)$

So, INSERTION-SORT has runtime of $\Omega(n^2)$.

(d) In REVERSE-PARTITION, the array A was modified such the element greater that the pivot come before and then the smaller elements come and then the pivot itself.

This algorithm took a runtime of $O(n)$.

But, instead, we can modify the array such that the elements which are smaller than the pivot come before and then the elements which are larger than the pivot comes and then the pivot itself.

So, this can be simply done by exchanging A[pivot] with A[r].

And now, this algorithm runs in O(1).

This algorithm will again produce the same results as before.

This is so because we only wanted a separtion of values larger that the pivot and the values smaller than

the pivot which is also been encountered in FAST- REVERSE- PARTITION.

~~The function of FAST is with the same~~

In this way the runtime of the algorithm is justified.

(e)

Now, the inversions encountered on applying INSERTION-SORT will be 1, 2, 4, 8, ... $\left(\frac{n}{2}\right)$

So, inversions $I = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots + 4 + 2 + 1$

$$I = n \lg n$$

So, $\theta(n+I) = \theta(n + n \lg n)$

Now, the runtime of INSERTION-SORT will be $\theta(n \lg n)$.

Problem 4-2

(a)

Algorithm-

1. Create a min heap and insert the first elements of all k lists ie., A_0.head.key, A_1.head.key, ... A_{k-1}.head.key.
2. start = last = NIL
3. while(! heap.empty()) {
4. current = heap.top()
5. heap.pop()
6. last.next = current
7. last = last.next

```
8.      if (current.next != NIL)
9.          heap.push(current.next)
10.     }
11.     return start
```

Approach:

We took a min-heap having k elements which are the head nodes of the k-lists. Now, as long as the heap is not empty, we remove the minimum element of this heap and add to a list X. Now, if there is any element next to this popped element, we insert that element into the heap.

Loop invariant:

The loop invariant will be that

After the ith iteration of the while loop in the lines 3 to 10, there will be i elements in the sorted list X.

Correctness of algorithm:

Initialization - If K is 1, then that is the only element in the list X and our loop invariant is trivially satisfied.

Maintenance - In every iteration of the while loop, some element is popped out of the min heap which gets inserted in the list X.

So, if after the ith iteration, we have a list of size i, then

after the $(i+1)$-th iteration we will have a list X of size $(i+1)$.

Termination — After the termination of the while loop, there will be a total of n popped elements which will result in the formation of the sorted list X.

So, our algorithm is correct.

The runtime of the algorithm is $O(n \log k)$.

(b) Algorithm:

* GENERATESUBARRAY(A[1, ..., n], n, k)

```
1.  for (i=0 ; i<K ; i++)
        A_i.head = A_i.tail = NIL

2.  for (i = 1 ; i<= n ; i++)
3.  {
4.      if (A_A[i].head == NIL)
5.      {
6.          A_A[i].head.key = [redacted] i
7.          A_A[i].tail.key = [redacted] i
8       }
9.      else {  A_A[i].tail.next.key = [redacted] i

10.         A_A[i].tail = A_A[i].tail.next

11.     }
12. }
```

We are just iterating over the array A and pushing the index i into the A[i]th linked list.

The runtime of algorithm is O(n).

Problem 4-3

(a)

K-PARENT (i)

1. return ceil((i-1)/k)

K-CHILD (i, j)

1. return $k(i-1)+j+1$

(b)

The index returned by K-CHILD (i, j) is $k(i-1)+j+1$ which will serve as an argument for the function k-PARENT and the value returned by it will be

$$\text{ceil}\left(\frac{K(i-1)+j+1-1}{K}\right)$$

$$= \text{ceil}\left(\frac{Ki-K+j}{K}\right) = \text{ceil}\left(\frac{iK+(j-K)}{K}\right)$$

Now j can lie between 1 and K as it represents the child no. of i.

So, $\quad j-K \leq 0$

$$\Rightarrow \quad \frac{j-K}{K} \leq 0$$

So, $iK \leq iK + (j-K) < (i+1)K$

$\Rightarrow \text{ceil}\left(\frac{iK+(j-K)}{K}\right) = \frac{iK}{K} = i$

So,

K-PARENT(K-CHILD(i, j)) = i

(c) K- MAX- HEAPIFY (A, i, K)

1. largest = i

2. { for j = 1 ; j <= K ; j++

3. t = K-CHILD(i, j)

4. if (t ≤ A. heapsize and A[t] > A[largest])

5. { largest = t }

6. }

7. if largest ≠ i

8. exchange A[i] with A[largest]

9. K-MAX- HEAPIFY (A, largest, K)

Approach:

Our approach is very similar to the max heapify used in a binary heap.

Instead of comparing the value of parent with only left and right child, we have to compare with all the k child.

(d) K-EXTRACT-MAX (A)

1. if A.heap-size < 1
2. error "heap underflow"
3. max = A[1]
4. A[1] = A[A.heap-size]
5. A.heap-size = A.heap-size - 1
6. K-MAX-HEAPIFY (A, 1, k)
7. return max

(c)

Height of the heap will be

$$\Theta(\log_K n)$$

Explanation:

Let us consider K-ary heap of height h having all levels filled.

So, $1 + K + K^2 + \ldots + K^{h-1} = n$

$$\Rightarrow \frac{K^h - 1}{K - 1} = n$$

$$\Rightarrow h = \log_K (n(K-1) + 1)$$

$$\Rightarrow h \approx \log_K n$$

$$\Rightarrow \boxed{h = \Theta(\log_K n)}$$

Since K-MAX-HEAPIFY runs from a node to straight down the heap. So, its runtime will be $O(h)$ which is $O(\log_K n)$.