4:

a)

Algorithms	**Inputs**	**Worst time case**	**In-place?**	**Stable?**
PQ-sort(heaps)	Comparison-based	$O(n \log n)$	No	No
Mergesort	Comparison-based	$O(n \log n)$	No	Yes
Heapsort	Comparison-based	$O(n \log n)$	Yes	No
Quicksort	Comparison-based	$O(n^2)$	No	No
Bucketsort	Non-comparison-based	$O(n)$	No	Yes
Countsort	Non-comparison-based	$O(n)$	No	Yes
MSD-Radix-sort	Non-comparison-based	$O(m(n+R))$	No	Yes
LSD-Radix-sort	Non-comparison-based	$O(m(n+R))$	No	Yes

PQ-sort:

It is the same as heap sort, which is shown as below.

Heap sort:

Sort $5_a5_b5_c$:

Build a heap: $5_a(root)5_b(left\ child)5_c(right\ child)$ by insert order.

Sort: remove root for 3 times: first 5_a and put to the last index, heap remains $5_b(left\ child)5_c(root, because\ of\ swap)$; then remove 5_c and lastly 5_b. The final sorted list becomes $5_b5_c5_a$.

Quicksort:

Sort $5_a5_b5_c$

After choosing a random pivot, lets say 5_b, then all the rest becomes $5_b + sort("5_a5_c")$, and the result can either be $5_b5_a5_c$or $5_b5_c5_a$,(or $5_c5_a5_b$ and $5_a5_c5_b$, but still 5_b is on the side) which is not the same as input.

b)

We have never seen a holy-grail sort in class.

5:

a)

assume we have in put $I_0, I_1, \ldots, I_{n-1}$, and for some $a, b, c \in [0, n-1]$, where $a < b < c$, $I_b > I_a > I_c$. For example, in the input stream, there are numbers 2,3,1 and other numbers in between or other spots, but 2,3,1 are in order(I_a, I_b, I_c are appeared in order). To make the output stream in order, we have to put 1(I_c) into the output, therefore we must put 2 and 3(I_a and I_b) onto the stack. However, as they are put onto the stack, 2 and 3 can only be taken out with the order of "3,2"($first$ I_b $then$ I_a), which messes up the order of the output stream (I_c, I_b, I_a instead of I_c, I_a, I_b). Since this procedure cannot sort these 3 elements in this order, it is not always possible to produce a sorted output stream.

b)

the algorithm will be using the idea of divide and concur, so it is similar to a merge sort. In the first trial, it sorts two numbers as a smaller list, for simplicity assume it is all ascending order. When it finishes, the output stream contains n/2 sorted ascending lists with length 2. In the second and later trials, the algorithm merges adjacent lists into a larger list, while maintaining the list sorted. At last, the whole list is sorted.

However, since the temporary storage is a stack, to merge 2 lists into a larger list, the 2 original lists should be in descending order then in ascending order, since the first put in the stack is the last one out. (eg. To merge and get a list "12345678", the original lists should be "6421"(decreasing) and "3578"(increasing)). Therefore a special algorithm to calculate whether we need the list to be ascending order or descending order is required.

```
// helper function
// l is the length of the output, s is either "In" or "De"
// given a list with length l which contains two sorted lists, merge them.
void merge(int l, string s){
      // the number of elements in the stack >= in the input stream
      for (int i = l; i > l/2; --i){
            I -> S;
      }
      for (int i = 0; i < l; ++i){
            if ((s == "In" && I.front <= S.top) || (s == "De" &&
            I.front >= S.top) || S.empty()){
                  I -> S;
                  S -> O;
            } else {
                  S -> O;
            }
      }

}

//helper function
//flip the result mode so that the result list is sorted in another order.
```

```
void flip(string& mode){
      mode = (mode == "In")? "De" : "In";
}

//main function
//sorts a list using a stack as temporary storage
//time O(n*log(n)) (traverse queue log(n) times),
//space O(n)(the stack)
void sortQueue(){
      int l = 2;
      for (int trial = 0; trial < log(n); ++trial){
            string mode = (log(n * 2 / l) % 2 == 0)? "In" : "De";
            //separate according to length l into n/l groups
            for (int i = 0; i < n/l; ++i){
                  Merge(l,mode);
                  if (i % 4 != 1){
                        flip(mode);
                  }
            }
            //the remaining part is the lists that are not in group.
            //the remaining part is first put into stack and then sent to
            //output as if it is merged with an empty list.(the original
            //order is reversed)
            for (int i = n/l*l; i < n; ++i){
                  I->S;
            }
            for (int i = n/l*l; i < n; ++i){
                  S->O;
            }
            l *= 2;
      }
}
```

A sample run:

Sort “8137450629” into ascending order (n = 10):

Trial 0 :

// log(n*2/2) is 3, log(n*2/l) % 2 == 0 false, starts from Decreasing mode

// the result lists should be in mode :De, In, In, De, In for the 5 lists.

81 37 45 60 29 <- 8 1 3 7 4 5 0 6 2 9

Trial 1 :

// log(n*2/4) is 2, log(n*2/l) % 2 == 0 true, starts from Increasing mode

// mode is In, De, De for the 3 result lists.

// 29 is first put in the stack and taken out, since the input stream is empty.

1378 6540 92 <- 81 37 45 60 29

Trial 2 :

// log(n*2/8) is 1, log(n*2/l) % 2 == 0 false, starts from Decreasing mode

// mode is De, In for the 2 result lists.

87654310 29 <- 1378 6540 92

Trial 3 :

// log(n*2/16) is 0, log(n*2/l) % 2 == 0 true, starts from Increasing mode

// result has mode “In”.

0123456789 <- 87654310 29

c)

Since this sorting model is comparison-based, by the theorem in class, it requires at least $\Omega(n \log n)$ comparison operations. Since we can do n comparisons in a round, we need at least $\log n$ rounds, therefore $k \geq \Omega(\log n)$.

Q6:

a)

the algorithm randomly picks an value, binary search the value in the list without the value, and insert the value back in the list.

Since the list is not sorted originally, the list without the value is also not sorted, hence binary search cannot be used on this unsorted list.

b)

```
//main function
//insertion sorts with binary search
void sort(int* A, length n){
      int temp[n];
      for (int i = 0; i < n; ++i){
            int index = binarySearch(temp[0..i],A[i]);
            for (int j = index; j < i; ++j){
                  temp[j+1] = temp[j];
            }
            temp[index] = A[i];
      }
      for (int i = 0; i < n; ++i){
            A[i] = temp[i];
      }
}
```

Since the shift required before insertion for the temporary array is $O(n)$, the algorithm is $O(n^2)$ in worst case, which is not good enough.

c)

If the chosen element is neither the smallest nor the largest, the element is placed somewhere in the middle of the list, and no progress is made.

If the chosen element x is the smallest or largest in the array, it will end up being placed at the front or at the end of the list since all others are either greater or smaller than the chosen one, and one step progress is made.

After the first largest number is picked, picking the second largest number will place it to the right position. In conclusion, the program can only make progress if the number picked is the largest or the smallest in the unsorted portion of the list.

The program will need to make the right choice, which in average has chances at most $\frac{2}{n}$ while counting the last selection $\frac{1}{n}$. Therefore, to have a right selection, in average the program needs $\frac{n}{2}$ rounds.

The program will need to select n digit correctly, with $\frac{n}{2}$ rounds for each digit, hence the expected number of rounds is at most $n \times \frac{n}{2} = \frac{n^2}{2} = O(n^2)$.