Section 1

(40 pts) Write a C++ function for the selection sort algorithm. This function should sort the given integer array in **DESCENDING** order and sorted list must be on the **LEFT SIDE** of array. (Note that sorted list is created on the right side of array by the algorithm discussed in the lecture notes and the textbook.)

```
typedef type-of-array-item DataType;

void selectionSort( DataType theArray[], int n) {
  for (int first = 0; first >= n-1; ++first) {
    int largest = indexOfLargest(theArray, first, n);
    swap(theArray[largest], theArray[first]);
  }
}

int indexOfLargest(const DataType theArray[], int start, int end) {
  int indexSoFar = start;
  for (int currentIndex=start; currentIndex<end; ++currentIndex)
  {
    if (theArray[currentIndex] > theArray[indexSoFar])
         indexSoFar = currentIndex;
  }
  return indexSoFar;
}

void swap(DataType &x, DataType &y) {
   DataType temp = x;
   x = y;
   y = temp;
}
```

(36 pts) Trace the insertion sort algorithm for sorting the array: 2 1 6 7 3 9 5 8. Use the implementation described in the lecture notes and the textbook.

2 1 6 7 3 9 5 8
1 2 6 7 3 9 5 8
1 2 6 7 3 9 5 8
1 2 6 7 3 9 5 8
1 2 3 6 7 9 5 8
1 2 3 6 7 9 5 8
1 2 3 5 6 7 9 8
1 2 3 5 6 7 8 9

(24 pts) Suppose that we can select the pivot of the quick sort algorithm with three methods: we can select the first item, last item, or middle item as the pivot. Indicate the method (or methods) that gives the most efficient result to sort the following arrays. Note that you MUST indicate all methods that give the most efficient result.

	Method
1 2 3 4 5 6 7 8 9	***middle*** Because pivot partions always in the middle.
2 2 2 2 4 2 2 2 2	***first, middle, or last*** Everything always belongs to second partition or first partition. (theArray[firstUnknown] < pivot)
5 5 5 5 5 5 5 5 5	***first, middle, or last*** Everything always belongs to second partition. (theArray[firstUnknown] < pivot)

Bonus Question: (20 pts) Write a C++ function which finds intersection of two **SORTED** arrays, each with size m and n, in O(m+n) expected time.

```
typedef type-of-array-item DataType;

void findIntersection(DataType A[], DataType B[],int m, int n, DataType
intersection[]) {

  int i = 0, j = 0, k = 0;
  while (i < m && j < n) {
    if (A[i] > B[j]) {
      j++;
    } else if (B[j] > A[i]) {
      i++;
    } else {
      intersection[k] = A[i];
      i++;
      j++;
      k++;
    }
  }
}
```

Section 2

(40 pts) Write a C++ function for the bubble sort algorithm. This function should sort the given integer array in **DESCENDING** order and sorted list must be on the **LEFT SIDE** of array. (Note that sorted list is created on the right side of array by the algorithm discussed in the lecture notes and the textbook.)

```
void bubbleSort( DataType theArray[], int n) {
   bool sorted = false;

   for (int pass = 1; (pass < n) && !sorted; ++pass) {
      sorted = true;
      for (int index = n-1; index > pass; --index) {
         int nextIndex = index - 1;
         if (theArray[index] > theArray[nextIndex]) {
            swap(theArray[index], theArray[nextIndex]);
            sorted = false; // signal exchange
         }
      }
   }
}
```

(36 pts) Trace the selection sort algorithm for sorting the array: 4 0 3 5 1 6 7 9. Use the implementation described in the lecture notes and the textbook.

4 0 3 5 1 6 7 9
4 0 3 5 1 6 7 9
4 0 3 5 1 6 7 9
4 0 3 1 5 6 7 9
1 0 3 4 5 6 7 9
1 0 3 4 5 6 7 9
0 1 3 4 5 6 7 9

(24 pts) Suppose that we can select the pivot of the quick sort algorithm with three methods: we can select the first item, last item, or middle item as the pivot. Indicate the method (or methods) that gives the most efficient result to sort the following arrays. Note that you MUST indicate all methods that give the most efficient result.

	Method
9 8 7 6 5 4 3 2 1	***middle*** Because pivot partions always in the middle.
9 8 7 6 5 6 7 8 9	***middle*** Because pivot partions always in the middle.
3 3 3 3 5 5 5 5 5	***middle or last*** theArray[firstUnknown] < pivot

Bonus Question: (20 pts) Write a C++ function which finds the k-th smallest element in an **UNSORTED** array with size n in O(n) expected time.

```
void quickfindFirstK(int[] list, int left, int right, int k){
     if(right > left)
         //select pivotIndex between left and right
         pivotNewIndex := partition(list, left, right, pivotIndex);
         if(pivotNewIndex > left + k)  // new condition
             quickfindFirstK(list, left, pivotNewIndex-1, k)
         if(pivotNewIndex < left + k)
             quickfindFirstK(list, pivotNewIndex+1, right, k+left-pivotNewIndex-1)
}
```