Written Questions:

1) For the following, write a loop invariant that can be used to prove correctness of the loop, prove the loop invariant is true at initialization, maintenance and termination and then use that to prove the correctness of the loop.

(a) (10 points) The following is a loop in selection sort that is used to find the index of the smallest value in the remaining elements.

```
// j is the variable of the outer loop in selection sort
min_index = j
for i=j+1 to n
     if A[i] < A[min_index]
          min_index = i
```

The loop invariant is: min_index is the index of the smallest element in A[j…i-1]

Init:

At initialization, minx_index is the index of the smallest element in A[1...i-1]= A[1...1], min_index=A[1]

Maint:

Assume true at iteration K, we show it's also true for k+1

min_index= A[1...k-1]

we have two cases when the loop iterates.

1)A[k] >= min_index

Min_index doesn't change and since min_index contains the index of smallest from A[1…k] that means the invariant holds true and we incremnt to k+1

2)A[k] < min_index

We change min_index to A[k] which makes the invariant true from A[1…k]

Termination:

When the loop terminates j=n, minx_index will be the index of the smallest element in A[i..n-1] therefore the invariant holds true.

Therefore this proves the loop invariant is correct

(b) (10 points) The following is a loop in insertion sort before inserting the value of key into sorted list.

// j is the variable of the outer loop in insertion sort

//key is the value we are sorting

```
i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

The loop invariant is: At each iteration A[1…j-1] contains the first j-1 elements in sorted order

Init:

At initialization, the array A[1…j-1] = A[1..2-1] = A[1…1] holds true since it contains only one element and a single element is always sorted.

Maint:

The element A[j] is positioned properly into A[1…j-1], since A[1…j-1] is sorted, inserting the element A[j] to the proper position makes A[1…j] a sorted sub array.

Termination:

The loop terminates when j = 1 + A.length, since our invariant is A[1…j-1] at termination A[1…A.length+1-1]=[1…A.length] therefore the entire specified array is sorted and our loop invariant holds true for the whole array.

Therefore this proves the loop invariant is correct (Supposing that we have an outer loop with j=2, if we don't include the outer loop then the invariant isn't correct, not sure if that's part of the question or not so I made the invariant with the outer loop in mind)

(10 points) Design an algorithm that takes an array of positive and negative elements and rearranges them such that all negative numbers proceed all the positive numbers. Your solution must do this in place so no new arrays are allocated. (Note the elements don't have to be sorted). What's the running time of your algorithm? Is it optimal (cannot be improved)? (Hint: think of how **partition** algorithm works)

```
   public static void zeroPartition(int[] a)
{
    int positionPos = 0;
    int temp;
    for (int i = 0; i <= a.length-1; i++)
    {
        if (a[i] >= 0)
        {
            temp=a[i];
            a[i]=a[positionPos];
            a[positionPos]=temp;
            positionPos++;
        }
    }
   }
```

The running time is O(n) since it has to iterate throught the whole array, if it finds a positive number it puts it in the beginning of the array which ends with all the negative numbers being in the end of the array.

The algorithim is not optimal and it can be improved. If we were required to make as optimal of an algorithim as we can I'd use two pivots beginning and end and iterate both at the same time making the algorithim O(n/2) or a recirsove quicksort that runs at O(lgn)instead.

2) (10 points) Design an algorithm that takes an array that is filled with characters that can be A, B or C. Design an algorithm that will rearrange the elements so that all A's proceed all the B's and all B's proceed all the C's. Your solution must do this in place so no new arrays are allocated and perform it in one pass over the values. (Hint: You might want to think of how **partition** works and modify it to work for this problem).

```
	public static void charPartition(char[] c)
{
	int startPos = 0;
	int lastPos=c.length-1;
	int currentPos=0;
	char temp;

	while(lastPos>=startPos && currentPos<=lastPos) {
		if (c[currentPos] == 'A')
		{
			lastPos=charPartitionHelper(c,currentPos,lastPos);
			temp=c[currentPos];
			c[currentPos]=c[lastPos];
			c[lastPos]=temp;
			lastPos--;
		}
		if (c[currentPos] == 'C')
		{
			temp=c[currentPos];
			c[currentPos]=c[startPos];
			c[startPos]=temp;
			startPos++;
		}
		currentPos++;
	}
}
public static int charPartitionHelper(char[] c,int current,int count) {
	while(c[count]=='A' && count>current) {
		count--;
	}
	return count;
}
```

Programming Questions:

3) (60 points) **Convex Hull (Reference slides for lecture 6)**

OVERVIEW:

In this question, you will implement and compare runtime for two convex hull algorithms. The brute force algorithm and quick hull on a set of randomly generated points.

- You will need to generate a set of points to compute the convex hull. You might want to make the points with integer coordinates x and y.
- You can use java.awt.Point
- In order to test that your algorithm works, use StdDraw for drawing the set of points. You could draw the points with a certain color and then draw the points of the convex hull

with a different color and you should be able to verify if your algorithm worked by looking at the output.

IMPLEMENTATION:

(a) (20 points) Implement the brute force algorithm described in class to compute the convex hull of a set of points in 2D space.

(b) (20 points) Implement the Quickhull algorithm described in class to compute the convex hull of a set of points in 2D space. Create a method that takes an ArrayList of Point objects and returns an ArrayList of Points on the convex hull. This method should

- find the points with minimum and maximum x coordinates (a, b)
- create two ArrayLists: upper containing points above line ab and lower containing points below line ab
- create an ArrayList of Points that will store the points of the convex hull
- make two calls to a recursive method that computes the upper and lower hulls (upper is the input to compute the upper hull and lower to compute the lower hull).
- See hints for more details on the recursive method.

(c) (20 points) Test your implementation to make sure your algorithms work and also verify expected running time by generating arrays of Point objects of increasing sizes (think about a good range of sizes for $\Theta(n^3)$ algorithm). The points should have a random x and y coordinate values. (note that you should **not** need to draw the output in main when you are trying to determine running time since the arrays will be of large size and drawing will take a long time).

Hints:

- Here is some pseudocode for the recursive method that computes upper hull:

```
QuickHull(S, a, b, Result)
      Furthest = furthest away from line ab
      Add Furthest to Result
      Create two arrays to store points: left and right
      for each point p in S
           Add p to left array if p is to the left of line from a to furthest
           Add p to right array if p is to the right of line from b to furthest
     QuickHull(left, a, furthest, result)
```

```
QuickHull(right, furthest, b, result)
```

- How do we determine the distance from point p to line ab? Notice that the points a, b, p form a triangle. The area of the triangle is 0.5 * base * height. Also note that height is the distance from point p to ab. Since all points share the same base ab, the triangle with the largest area is indeed the point furthest away from line ab (see figure). Cross product can be used to compute a value that corresponds to twice the area of triangle:

```
valueBasedOnLineDistance(a, b, p)
        v1x = b.x - a.x
        v1y = b.y - a.y

        v2x = p.x - a.x
        v2y = p.y - a.y

        return abs(v1x * v2y - v1y * v2x)
```

- (Optional) If you want to draw the lines of the convex hull (it looks better) then instead of storing all points of the convex hull in the same array, create two arrays (upperHull and lowerHull) then after you compute the two hulls:
  1. sort the points in the result arrays with respect to x-coordinate.
  2. Draw a line between each two consecutive points in the upper hull and each two consecutive points in the lower hull. This will draw the convex polygon containing the set of points

DELIVERABLES:

1) Please submit a document that has your written answer for questions 1, 2 and 3
2) Please submit a zip file of your implementation of the convex hull algorithms. Your implementation should have two driver files with two main methods:
   (a) The first main method has code demonstrating that the algorithms work on a randomly generated array of size 50 and draws the points of the convex hull in different
   (b) The second main method has the runtime results with increasing array size.

3) Please submit experimental results for runtime of merge sort and insertion sort. You are free to choose the medium that illustrates result (spread sheet or code output).

Insetion sort of arrays of random numbers of increasing size:

229.0

829.0

1044.0

4240.0

Insetion sort of arrays of sorted values:

0.0

0.0

1.0

2.0

Merge sort of arrays of randomly generated values:

11.0

21.0

28.0

49.0