1. Solve the following recurrence relations. You do not need to give a Θ() bound for (a) and (b); it suffices to give the O() bound that results from applying the Master theorem. You may assume that T(n) = O(1) for n = O(1).

(a) $T(n) = 2T(n/3) + 1$.

a = 2, b=3, d=0, c=0.63, c>d: T(n) = O (n^0.63)

(b) $T(n) = 7T(n/7) + n$.

a = 7, b=7, d=1, c=1, c=d: T(n) = O (n.log n)

(c) $T(n) = T(n - 1) + 2$.

1. $T(n) = T(n - 1) + 2$.
2. $T(n)= T(n-2) +4$
3. $T(n)=T(n-3) +6$
4. $T(n)=T(n-4) +8$

.

.

i. $T(n)=T(n-i) +2.i$

n-i=1, i=n-1

$T(n) = T(1)+2n-2 = \Theta(n)$

2. Give a recursive version of the algorithm Insertion-Sort based on the following paradigm: to sort A[1..n], we first sort A[1..n−1] recursively and then insert A[n] in its appropriate position. Write a pseudocode for the recursive version of Insertion-Sort and analyze its running time by giving a recurrence relation for the running time and then solving it.

```
Insertion-Sort (A,n)
  if n <=1 then return; end
  Insertion-Sort (A,n-1);
  last = A[n-1];
  i = n-2;
  while (i >= 0 and A[i] > last) do
        A[i+1] = A[i];
        i = i – 1;
  end
  A[i + 1] = last;
```

Time Complexity:

1. *T(n) = T(n-1) + (n – 1)*
2. *T(n) = T(n-2) + (n – 2) + (n – 1)*

   .

   .

   .
3. *T(n) = T(n-i)+(n-i)+(n-i+1))...+ (n – 2) + (n – 1)*

n-i=1, i=n-1
T(n)= T(1)+1+2+….+(n – 2) + (n – 1) = Θ(n^2)

3. (15 points) Show how to determine in O(n^2lg n) time whether any three points in a set of n points are collinear.

```
import java.util.Scanner;
import java.util.Arrays;
import java.util.Stack;
import java.util.Comparator;

class Points implements Comparable<Points>
{
   public final Comparator<Points> AngleOrder = new AngleOrder();

   private final int x;
   private final int y;

   public Points(int x, int y)
   {
       this.x = x;
       this.y = y;
   }

   public int x()
   {
       return x;
   }

   public int y()
   {
       return y;
   }


   private double angleTo(Points that)
   {
       int dx = that.x - this.x;
       int dy = that.y - this.y;
       return Math.atan2(dy, dx);
   }

   public static int ccw(Points a, Points b, Points c)
   {
       double area2 = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
       if (area2 < 0)
           return -1;
       else if (area2 > 0)
           return +1;
       else{
           return 0;
           }
   }

   public int compareTo(Points that)
   {
       if (this.y < that.y)
```

```
            return -1;
        if (this.y > that.y)
            return +1;
        if (this.x < that.x)
            return -1;
        if (this.x > that.x)
            return +1;
        return 0;
    }


    private class AngleOrder implements Comparator<Points>
    {
        public int compare(Points q1, Points q2)
        {
            double angle1 = angleTo(q1);
            double angle2 = angleTo(q2);

            if (angle1 < angle2)
                return -1;
            else if (angle1 > angle2)
                return 1;
            else
                return 0;
        }
    }

    public String toString()
    {
        return "(" + x + ", " + y + ")";
    }

}

public class Collinear
{
    private Stack<Points> stack = new Stack<Points>();
```

```
public Collinear(Points[] points)
{
   int len = points.length;
   Points origin = new Points(0,0);
   Arrays.sort(points, 1, len, origin.AngleOrder);
   boolean found=false;
   for (int i = 0; i < len-2; i++){
      if (Points.ccw(points[i], points[i+1], points[i+2]) == 0){
         found=true;
         System.out.println("Collinear points:");
         System.out.println(points[i].toString());
         System.out.println(points[i+1].toString());
         System.out.println(points[i+2].toString());
         break;
      }

   }

   if(!found) System.out.println("No collinear points found");

}

public Iterable<Points> print()
{
   Stack<Points> s = new Stack<Points>();
   for (Points p : stack)
      s.push(p);
   return s;
}

 public static void main(String[] args)
{
   Scanner sc = new Scanner(System.in);
   System.out.println("Enter the no. of points:");
   int num = sc.nextInt();
```

```
        Points[] points = new Points[num];
        System.out.println("Enter the point coordinates:");
        for (int i = 0; i < num; i++)
        {
           int x = sc.nextInt();
           int y = sc.nextInt();
           points[i] = new Points(x, y);
        }
       Collinear col = new Collinear(points);
       for (Points p : col.print())
           System.out.println(p);

        sc.close();
    }

}
```

4. Give an algorithm that takes as input a positive integer n and a number x, and computes x^n (i.e., x raised to the power n) by performing O(lg n) multiplications. Your algorithm CANNOT use the exponentiation operation, and may use only the basic arithmetic operations (addition, subtraction, multiplication, division, modulo). Moreover, the total number of basic arithmetic operations used should be O(lg n).

```
import java.util.Scanner;
class Exponent {
        static int power(int num, int exp)
        {
                if (exp == 0)  return 1;
                int x = power(num, exp / 2);
                if (exp % 2 == 0) return x * x;
                else return num * x * x;
        }

        public static void main(String[] args)
        {
           Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number:");
        int num = sc.nextInt();
        System.out.println("Enter the exponent:");
        int exp = sc.nextInt();
        System.out.printf("Result: %d", power(num, exp));
        sc.close();


    }
}
```

5. Although merge sort runs in Θ(n lg n) worst-case time and insertion sort runs in Θ(n^2) worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to coarsen the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

(a) Show that insertion sort can sort the n/k sublists, each of length k, in Θ(nk) worst-case time.

The total cost of insertion sort for an array of length k is
Best case: O(k)
Worst case: O(k^2)

There are n/k such sublists *each of length k, so the total cost of insertion sort is*
*Best case: (n/k) * k = O(n)*
*Worst case: (n/k) * k^2 = O(nk)*

Hence the insertion sort can sort the n/k sublists, each of length k, in Θ(nk) worst-case time.

(b) Show how to merge the sublists in Θ(n lg (n/k)) worstcase time.

Merging n/k sublists of length k

```
k       k               k       k               k       k               k       k

  2k                      2k                      2k                      2k

              4k                      4k                      4k
                                      .
                                      .
                                      .

      (2^i)k                          (2^i)k                          (2^i)k
```

We merge till (2^i)k =n; i= log(n/k)
Merge a total of n elements takes Θ(n) time
*Total time = n*log(n/k)*
So the worst-case time to merge the sublists is Θ(n lg (n/k))

```
mergeSort(int a[],int l, int u){
   if (u - l <= k)
      insertionSort(a, l, u);
   else
   {
      int mid = (l+u)/2;
      mergeSort(a,l,mid);
      mergeSort(a,mid,u);
      merge(a,l,mid,u);
   }
}

insertionSort (int a[], int l, int u){
 for (int i = l; i <= u; i++){
          int j = i;
         int temp = a[i];
         while ((j > 0) && (a[j-1] > temp)){
                  a[j] = a[j-1];
```

```
                    j--;
            }
            a[j] = temp;
    }
}

merge(int a[], int l, int mid, int u) {

  int a1 = mid - l + 1;
  int a2 = u - mid;

  int left[a1], right[a2];

  for (int i = 0; i < a1; i++)
    left[i] = a[l + i];
  for (int j = 0; j < a2; j++)
    right[j] = a[mid + 1 + j];

  int i, j, k;
  i = 0;  j = 0;  k = l;

  while (i < a1 && j < a2) {
    if (left[i] <= right[j]) {
      a[k] = left[i];
      i++;
    } else {
      a[k] = right[j];
      j++;
    }
    k++;
  }

  while (i < a1) {
    a[k] = left[i];
    i++;
    k++;
  }
  while (j < a2) {
    a[k] = right[j];
    j++;
    k++;
  }
}
```

(c) Given that the modified algorithm runs in Θ (nk +n lg(n/k)) worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ-notation?

For the modified algorithm to have the same running time as standard merge sort, Θ(nk +n lg(n/k)) should be same as Θ(n logn), so k cannot grow faster than logn (upper bound) , k= Θ(logn)

Θ(nk +n lg(n/k))
Θ(nlogn+nlog(n/logn))
Θ(nlogn +nlogn – nlog(logn))
Θ(2nlogn-nlog(logn))
~ Θ(nlogn)

6. (20 points) A group of n Ghostbusters is battling n ghosts. Each Ghostbuster carries a proton pack, which shoots a stream at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits the ghost. The Ghostbusters decide upon the following strategy. They will pair off with the ghosts, forming n Ghostbuster-ghost pairs, and then simultaneously each Ghostbuster will shoot a stream at his chosen ghost. As we all know, it is very dangerous to let streams cross, and so the Ghostbusters must choose pairings for which no streams will cross. Assume that the position of each Ghostbuster and each ghost is a fixed point in the plane and that no three positions are collinear.
(a) (10 points) Argue that there exists a line passing through one Ghostbuster and one ghost such that the number of Ghostbusters on one side of the line

equals the number of ghosts on the same side. Describe how to find such a line in O(n lg n) time.

1. *Read Ghostbuster and Ghost point coordinates*
2. *Find the lowest Ghostbuster point (as in Graham scan algorithm, least y-coordinate and x-coordinate)*
3. *Sort all the points in increasing order of angle w.r.t to the lowest point*
4. *Traverse through the sorted list of points by their angle order and flag the visited points*
5. *When the difference between the number of visited Ghostbuster and Ghost points is -1, connect that point to the lowest point*

The algorithm majorly takes time for sorting, so time complexity is O(n logn).

(b) Give an O(n^2lg n)-time algorithm to pair Ghostbusters with ghosts in such a way that no streams cross.

The algorithm from (a) gets the first pair of Ghostbuster and Ghost points.
On each side of the line formed from here, the algorithm should be used recursively, traversing through the pairs of Ghostbuster and Ghost points and finding the other set of points where the lines doesn't intersect.
The worst case scenario would be when there are all the remaining points are only on one side of the line, with no points on the other, we have to iterate the loop for n/2 times to find all the pairings, which implies O(n^2 logn) time complexity.

```
import java.util.Scanner;
import java.util.Arrays;
import java.util.Stack;
import java.util.Comparator;
import java.util.ArrayList;
import java.util.List;

public class GrahamAlg
{
   private Stack<Points> stack = new Stack<Points>();
```

```
public static void main(String[] args)
{
   Scanner sc = new Scanner(System.in);
   System.out.println("Enter the no. of pairs of points:");
   int num = sc.nextInt();

   Points[] Busterpoints = new Points[num];
   Points[] Ghostpoints = new Points[num];
   System.out.println("Enter the Ghostbuster point coordinates:");
   for (int i = 0; i < num; i++)
   {
      int x = sc.nextInt();
      int y = sc.nextInt();
      Busterpoints[i] = new Points(x, y);
   }
    System.out.println("Enter the Ghost point coordinates:");
   for (int i = 0; i < num; i++)
   {
      int x = sc.nextInt();
      int y = sc.nextInt();
      Ghostpoints[i] = new Points(x, y);
   }

   GrahamAlg graham = new GrahamAlg(Busterpoints,Ghostpoints);
   System.out.println("Pairs:");
   for (Points p : graham.getPointPairs())
   {
      System.out.println(p);
   }
   sc.close();
}

public GrahamAlg(Points[] b, Points[] g)
{
   int num = b.length;
   Points[] points = new Points[2*num];
   Arrays.sort(b);
```

```
        for (int i = 0; i < num; i++)
        {
           points[i] = b[i];
        }
        Arrays.sort(g);
        for (int i = 0; i < num; i++)
        {
           points[num+i] = g[i];
        }
        int len=points.length;
       stack.push(points[0]);
        Arrays.sort(points,1,len-1, points[0].SortbyAngle);

        getPairs(points);


    }
 Public void getPairs(Points[] points){
Int num=points.length;
        for(int i=0;i<num;i++){
           int busterCount=0;
           int ghostCount=0;
           if(Arrays.binarySearch(b,points[i])>=0){
              busterCount++;
           }
           else {
               ghostCount++;

           }
           if(busterCount-ghostCount==-1){
              stack.push(points[i]);
             Points[]p1=new Point[num/2];
            Points[]p2=new Point[num/2];
            System.arrayCopy(points,0,p1,i);
            getPairs(p1);
           System.arrayCopy(points, i+1,p2,num-i-1);
```

```
          getPairs(p2);
           }
        }

  }
     public Stack<Points> getPointPairs()
     {
        Stack<Points> hull = new Stack<Points>();
        for (Points p : stack){
           hull.push(p);
        }
        return hull;
     }


  }

  class Points implements Comparable<Points>
  {
     public final Comparator<Points> SortbyAngle = new SortbyAngle();
     private final int x;
     private final int y;

     public Points(int x, int y)
     {
        this.x = x;
        this.y = y;
     }
     public int x()
     {
        return x;
     }
     public int y()
     {
        return y;
     }
     public String toString()
```

```
    {
        return x + ", " + y;
    }
    public int compareTo(Points p)
    {
        //Comparing for the lowest point
        if (this.y > p.y) return 1;
        else if (this.y < p.y) return -1;
        else if (this.x > p.x) return 1;
        else if (this.x < p.x) return -1;
        else return 0;
    }
    private class SortbyAngle implements Comparator<Points>
    {
       public int compare(Points q1, Points q2)
        {
            double angle1 = angleTo(q1);
            double angle2 = angleTo(q2);

            if (angle1 < angle2)
                return -1;
            else if (angle1 > angle2)
                return 1;
            else
                return 0;
        }

    }

    private double angleTo(Points that)
    {
        double dx = that.x - this.x;
        double dy = that.y - this.y;
        return Math.atan2(dy, dx);
    }

}
```