HOMEWORK 4

Name: Tianyu Gu Collaborator: AngLi

Q1:

The problem can be solved by dynamic programming:

For the node v_n, in order to find if there is a path from v_1 to v_n with k edges, we could check all the paths from v_i to v_n, if exists. For example, if there is a path from v_m to v_n , this problem becomes if there is a path from v_1 to v_m with exactly k-1 edges. This algorithm will run in O(n!) times at the worst case. Therefore, we need to memorize all the internal computations, by which we could decrease the computation dramatically.

Pseudo code:

```
1.  Matrix_Opt[N][k] = Nil  // declare a two dimensional array
2.  OPT(G,k):               // the function used to find the path. Suppose the node in G is arranges from V1
                               to Vn //
3.      if(k > n):
4.          return 0        // Apparently,there is no path when k > n.
5.      if(k == 0 and Vn != V1):
6.          return 0        // this means it cannot reach V1 with k edges.
7.      if(k == 0 and Vn == v1):
8.          return Vn       // this means a valid path has been found.
9.
10. // the above are base cases//
11.     else:
12.         while the path (Vi,Vn) exists:        //search all the paths ends with Vn. i is range(1,n)
13.             if Matrix_Opt[Vi][k-1] != Nil:
14.                 temp = Matrix_opt[Vi][k-1]
15.             else:
16.                 G' = G with only V1 to Vi      //cut the graph and keep the node V1 to Vi, since in
    this line-graph, there is no path backward.
17.                 temp = OPT(G',k-1)             // the problem has become a subproblem with smaller v and
                                                      k.
18.                 Matrix_Opt[n][k] =temp         // record the OPT(Gn,k), since this is the first time of
19. Computing this.
20.                 if temp != 0:                  // temp != 0 means that we found a path
21.                     Vn.π = Vi                  // this record the actual path by recording the last node in the
22.  Path.
23.                     return Vn                  // return the node we found
24.
25.         end while
```

```
26.         return 0                    // since we did not find any path from Vi to Vn with edges k, then return 0
27.                                     // which means the path with k edges cannot be found.
```

Correctness:

Let Path (v_n , k) gives the path from v_1 to v_n with edges k, otherwise Nil. we could find that, for any points in this line-graph, let's say v_m. If there is a path from v_1 to v_m with k edges, we can know that it could be expressed as:

$$\text{Path}(v_m, k) = \text{Path}(v_i, k-1) + v_i \rightarrow v_m$$

$$\text{for } i \in (1, m) \text{ that edge } (v_i, v_m) \text{ exists.}$$

Therefore, the existence of $\text{Path}(v_m, k)$ really depends on if there is an i makes $\text{Path}(v_i, k-1)$ true.

Runtime analysis:

In the worse case, the number of edges k will equal to n.

Therefore, in this case, in order to solve the $\text{Path}(v_n, n)$, we need to solve $\frac{n^2}{2}$ sub problems, ranging from $\text{Path}(v_1, 1)$ to $\text{Path}(v_n, n)$. Since we use a memorization method, which enables the algorithm only solve each sub problem once. For each problem, in the worse case, we need n times of iteration to solve (line 12 ~ line 25). That is to say: for solving $\text{Path}(v_1, 1)$ we need 1 unit of computation, for $\text{Path}(v_2, 1)$ and $\text{Path}(v_2, 2)$ we need 2 unit of computation each.

Therefore, the total running time is bounded by:

$1^2 + 2^2 + \ldots + n^2$ This running time, which can be proved inductively, equals to $O(n^3)$.

Q2.
(a).

We can use the similar algorithm discussed in the class, with several small changes:

1. Delete the constant trade-off factor λ.
2. Add a new factor k, to count the numbers of partitions.

Therefore, for the points $p_1\ p_2\ p_3 \ldots\ldots p_j$, and k segments, the problems can be reduced by the rules:

$$\mathrm{OPT(j,k)} = \min\left(e_{i,j} + OPT(i-1, k-1)\right)\ for\ 1 \le i \le j$$

Therefore, we could establish the whole OPT table increasingly by iteration.

Pseudo code:

```
Function (j, k):
    Array OPT(j,k) =Nil
    Set OPT(1,i) = 0 for all 1 ≤ i ≤ k
    For all pairs i ≤ j :
        Compute the least squares error e_{i,j} for the segment
 p1 p2 p3 …… pj
    End for
    // the iteration used to compute the error i j//
    For j = 1, 2…n:
        For i=1,2,..k:
            Use the recurrence above to compute OPT(j,i)
        Endfor
    Endfor
    Return OPT (n,k)
// the iteration used to update the table opt//
```

```
// the function used to find the exacted segments//
Find-Segments (n, k, opt (n, k)):
    If n=0 or k=0 then:
        Output nothing
    Else:
        Find an i that minimize e_{i,j} + OPT(i − 1, k − 1)
        Return the i and Find-segments(i-1,k-1)
```

(b).

The program below has been implemented by java.

My answer for segmentation is: [1, 37, 59, 82, 100], and the error is 71.11648043771159.

There is a slightly difference between the real implementation and the pseudo code, in the program, I use a 2d matrix path to keep tracking i that minimize $e_{i,j} + OPT(i - 1, k - 1)$ for opt (n, k), so that the find-segments function is not necessary now.

It's a trade-off between space efficiency and time efficiency, since the problem only give n=100, it will not introduce any problems.

Code:

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.Scanner;

import static java.lang.Math.pow;    //only used to compute power

public class hw4_2{
    // the function used to read in the files hw4test.txt.
   public static double[][] readfile() throws FileNotFoundException {
      double[][] vertices= new double[100][2];
        Scanner in = new Scanner(new FileReader("hw4test.txt"));
      int n = 0;
      double temp_x,temp_y;
      while(in.hasNextLine()){
         temp_x= in.nextDouble();
         temp_y = in.nextDouble();
```

```
            vertices[n][0] =temp_x;
            vertices[n][1] = temp_y;
            n++;
        }
        return vertices;

    }
    //this function is used to compute the a in linear regression for points from
m to n.
    public static double compute_a(int m,int n,double[][] vertices){
        double sum_x=0;
        double sum_y=0;
        double sum_xy=0;
        double sum_x2=0;
        double number = n-m+1;
        for(int i=m-1;i<n;i++){
            sum_x = sum_x + vertices[i][0];
            sum_y = sum_y + vertices[i][1];
            sum_xy = sum_xy + vertices[i][0] * vertices[i][1];
            sum_x2 = sum_x2 + pow(vertices[i][0],2);
        }
        double numerator = number * sum_xy - sum_x * sum_y;
        double denominator = number * sum_x2 - pow(sum_x,2);
        return  numerator/denominator;

    }
     //this function is used to compute the b in linear regression for points
from m to n.
    public static double compute_b(int m,int n,double[][] vertices){
        double sum_x=0;
        double sum_y=0;
        double number = n-m+1;
        double a = compute_a(m,n,vertices);
        for(int i=m-1;i<n;i++){
            sum_x = sum_x + vertices[i][0];
            sum_y = sum_y + vertices[i][1];
        }
        double numerator = sum_y - a * sum_x;
        return numerator/number;
    }
    //this function is used to compute the error r for points m to n.
    public static double compute_r(int m,int n,double[][] vertices){
        double error = 0;
        double difference = 0;
```

```
      double a = compute_a(m,n,vertices);
      double b = compute_b(m,n,vertices);
      if(m==n) {return 0;}
      for (int i = m-1;i<n;i++){
         difference = vertices[i][1]-a*vertices[i][0]-b;
         error = error + pow(difference,2);
      }
      return error;
   }
   //this function is used to calculate the OPT tables
   public static double OPT(int n,int k,double[][] vertices) {
      double[][] Matrix_Opt = new double[n+1][k];
      double error[][] = new double[n+1][n+1];
      double temp = 0;
      double min = 1000; // a number to assign min to infinite, so we can update
it.
      int[][] path = new int[n+1][k]; //used to keep tracking the points to
segment.
      int previous = 0;
        int[] trace = new int[k];// a small matrix used to print the segmentation
        int num = k;
        //compute the error matrix Eij for all i and j
      for (int i = 1; i < 100; i++) {
         for (int j = i; j <= 100; j++) {
            error[i][j] = compute_r(i, j, vertices);
         }
      }
      // iteratively compute the OPT matrix
      for (int j = 1; j <= 100; j++) {
         for (int i = 0; i < k; i++) {
            if (j == 1 || j == 2) {    //initialize the value
               Matrix_Opt[j][i] = 0;
            }
            else {
                // when k=0, it means only one line left, the OPT value is the
same with error value.
               if (i == 0) {
                  Matrix_Opt[j][i] = error[1][j];
               }
               else {
                   // follow the dynamic rules to update the table
                  for (int m = 1; m < j; m++) {
                     temp = Matrix_Opt[m-1][i - 1] + error[m][j];
                     if (temp < min) {
```

```
                    min = temp;
                    previous = m;
                }
            }
            Matrix_Opt[j][i] = min;
            path[j][i] = previous;
            min=1000;
        }
      }
    }
    //print out the exact segmentation points
      trace[num - 1] = path[n][num - 1];
      while(num>1) {
          System.out.println(trace[num - 1]);
          trace[num - 2] = path[trace[num - 1]][num - 2];
          num--;
      }
    return Matrix_Opt[n][k-1];

  }
  public static void main(String[] args) throws FileNotFoundException {
    double[][] vertices;
    vertices = readfile();
    double temp = OPT(100,4,vertices);
    System.out.println(temp);

  }
}
```

Q3:

First, according to the question, we can safely assume the total number of precincts n is even number.

Secondly, we can assume the total registered voters for party A is larger than B. therefore, we can only focus on party A, since it's impossible for party B to have majority in both district.

Notations:

Use $p_1\ p_2\ p_3 \ldots . . p_n$ to represent the precincts.

Use $a_1\ a_2\ a_3 \ldots \ldots a_n$ to represent the number of registered voters for party A in each precinct.

Use m to denote the total number of registered voters in each precinct.

The total voters for A will be denoted as sum_a and we know $sum_a \geq nm/2$

Use q to represent the total number of precincts in one district.

Use s_q to denote the sum of a_i which belongs to the district of s with q numbers of precincts.

Algorithm:

The problem can be solved by dynamic programming, the function $OPT(n, q, s_q)$ gives whether for $p_1 \sim p_n$, there exists a district with q precincts and s_q voters.

Therefore, we could know that:

$$OPT(n, q, s_q) = OPT(n-1, q, s_q) \text{ or } OPT(n-1, q-1, s_q - a_n)$$

That means, in order to find a district with m precincts and s_q voters in $p_1 \sim p_n$, it has to be in one of the following situations:

3. We can find a district with q precincts and s_q voters in $p_1 \sim p_{n-1}$.
4. We can find a district with q-1 precincts and $s_q - a_n$ voters in $p_1 \sim p_{n-1}$, so that we can put p_n in this district.

The base case is simple:

$OPT(1,1,a_1)$ is True.

Therefore we can build the table iteratively, based on the recursion rules above

for all $n, q \in \left(1, \frac{n}{2}\right), s_q \in (a_1, sum_a)$.

Solution:

We know that, in one district, if voters for party A want to dominate, the a_q in this area must larger than nm/4. Moreover, since A need dominate in both districts, so that $sum_a - a_q$ also need to be larger than nm/4.

Therefore, after we compute the whole OPT table, we could check all the values among $OPT(n, n/2, a_{n/2})$ where $a_{n/2} \in (\frac{nm}{4}, sum_a - \frac{nm}{4})$. Then we can know that, it is susceptible for gerrymandering if any of these values is true.

Runtime Analysis:

For computing each entry in this OPT table, only constant time is needed, since you compute it increasingly using iteration. Therefore, the whole running time is bounded by number of entries, which is $O\left(n * \frac{n}{2} * sum_a\right) = O(n^3 m)$