**IQRA NATIONAL UNIVERSITY**

**Name: Nazish**

**ID NO: 6861**

**Degree: BS (SE)**

**Subject: Data Mining**

**Lecturer: Sir zain**

Q1. Implement a code of Genetic Algorithm in any language and show the output.

**import** java.util.Random;

//Main class

**public** **class** SimpleDemoGA {

Population population = **new** Population();

Individual fittest;

Individual secondFittest;

**int** generationCount = 0;

**public** **static** **void** main(String[] args) {

Random rn = **new** Random();

SimpleDemoGA demo = **new** SimpleDemoGA();

//Initialize population

demo.population.initializePopulation(10);

//Calculate fitness of each individual

demo.population.calculateFitness();

System.***out***.println("Generation: " + demo.generationCount + " Fittest: " + demo.population.fittest);

//While population gets an individual with maximum fitness

**while** (demo.population.fittest < 5) {

++demo.generationCount;

//Do selection

demo.selection();

//Do crossover

demo.crossover();

//Do mutation under a random probability

**if** (rn.nextInt()%7 < 5) {

demo.mutation();

}

//Add fittest offspring to population

demo.addFittestOffspring();

//Calculate new fitness value

demo.population.calculateFitness();

System.***out***.println("Generation: " + demo.generationCount + " Fittest: " + demo.population.fittest);

}

System.***out***.println("\nSolution found in generation " + demo.generationCount);

System.***out***.println("Fitness: "+demo.population.getFittest().fitness);

System.***out***.print("Genes: ");

**for** (**int** i = 0; i < 5; i++) {

System.***out***.print(demo.population.getFittest().genes[i]);

}

System.***out***.println("");

}

//Selection

**void** selection() {

//Select the most fittest individual

fittest = population.getFittest();

//Select the second most fittest individual

secondFittest = population.getSecondFittest();

}

//Crossover

**void** crossover() {

Random rn = **new** Random();

//Select a random crossover point

**int** crossOverPoint = rn.nextInt(population.individuals[0].geneLength);

//Swap values among parents

**for** (**int** i = 0; i < crossOverPoint; i++) {

**int** temp = fittest.genes[i];

fittest.genes[i] = secondFittest.genes[i];

secondFittest.genes[i] = temp;

}

}

//Mutation

**void** mutation() {

Random rn = **new** Random();

//Select a random mutation point

**int** mutationPoint = rn.nextInt(population.individuals[0].geneLength);

//Flip values at the mutation point

**if** (fittest.genes[mutationPoint] == 0) {

fittest.genes[mutationPoint] = 1;

} **else** {

fittest.genes[mutationPoint] = 0;

}

mutationPoint = rn.nextInt(population.individuals[0].geneLength);

**if** (secondFittest.genes[mutationPoint] == 0) {

secondFittest.genes[mutationPoint] = 1;

} **else** {

secondFittest.genes[mutationPoint] = 0;

}

}

//Get fittest offspring

Individual getFittestOffspring() {

**if** (fittest.fitness > secondFittest.fitness) {

**return** fittest;

}

**return** secondFittest;

}

//Replace least fittest individual from most fittest offspring

**void** addFittestOffspring() {

//Update fitness values of offspring

fittest.calcFitness();

secondFittest.calcFitness();

//Get index of least fit individual

**int** leastFittestIndex = population.getLeastFittestIndex();

//Replace least fittest individual from most fittest offspring

population.individuals[leastFittestIndex] = getFittestOffspring();

}

}

//Individual class

**class** Individual {

**int** fitness = 0;

**int**[] genes = **new** **int**[5];

**int** geneLength = 5;

**public** Individual() {

Random rn = **new** Random();

//Set genes randomly for each individual

**for** (**int** i = 0; i < genes.length; i++) {

genes[i] = Math.*abs*(rn.nextInt() % 2);

}

fitness = 0;

}

//Calculate fitness

**public** **void** calcFitness() {

fitness = 0;

**for** (**int** i = 0; i < 5; i++) {

**if** (genes[i] == 1) {

++fitness;

}

}

}

}

//Population class

**class** Population {

**int** popSize = 10;

Individual[] individuals = **new** Individual[10];

**int** fittest = 0;

//Initialize population

**public** **void** initializePopulation(**int** size) {

**for** (**int** i = 0; i < individuals.length; i++) {

individuals[i] = **new** Individual();

}

}

//Get the fittest individual

**public** Individual getFittest() {

**int** maxFit = Integer.***MIN\_VALUE***;

**int** maxFitIndex = 0;

**for** (**int** i = 0; i < individuals.length; i++) {

**if** (maxFit <= individuals[i].fitness) {

maxFit = individuals[i].fitness;

maxFitIndex = i;

}

}

fittest = individuals[maxFitIndex].fitness;

**return** individuals[maxFitIndex];

}

//Get the second most fittest individual

**public** Individual getSecondFittest() {

**int** maxFit1 = 0;

**int** maxFit2 = 0;

**for** (**int** i = 0; i < individuals.length; i++) {

**if** (individuals[i].fitness > individuals[maxFit1].fitness) {

maxFit2 = maxFit1;

maxFit1 = i;

} **else** **if** (individuals[i].fitness > individuals[maxFit2].fitness) {

maxFit2 = i;

}

}

**return** individuals[maxFit2];

}

//Get index of least fittest individual

**public** **int** getLeastFittestIndex() {

**int** minFitVal = Integer.***MAX\_VALUE***;

**int** minFitIndex = 0;

**for** (**int** i = 0; i < individuals.length; i++) {

**if** (minFitVal >= individuals[i].fitness) {

minFitVal = individuals[i].fitness;

minFitIndex = i;

}

}

**return** minFitIndex;

}

//Calculate fitness of each individual

**public** **void** calculateFitness() {

**for** (**int** i = 0; i < individuals.length; i++) {

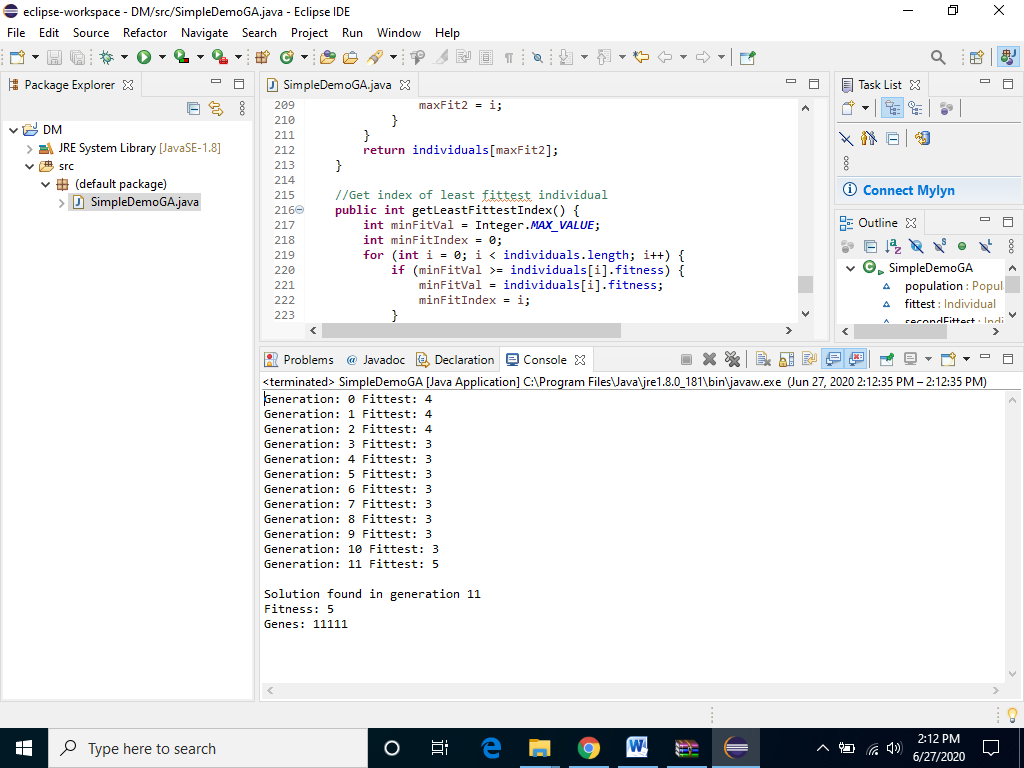
individuals[i].calcFitness();

}

getFittest();

}

}

**OUTPUT** 

Q2. Implement a code of Fuzzy logic in any language and show the output.

To implement a fuzzy system in C, the following types of data must be accommodated:

* System inputs.
* Input membership functions.
* Antecedent values.
* Rules.
* Rule-output strengths.
* Output membership functions.
* System outputs.

**Listing One** includes the C-code definition of these data structures:

/\*CODE is about General-purpose fuzzy inference engine supporting any number of system

inputs and outputs, membership functions, and rules. Membership functions can

be any shape defineable by 2 points and 2 slopes--trapezoids, triangles,

rectanlges, etc. Rules can have any number of antecedents and outputs, and can

vary from rule to rule. "Min" method is used to compute rule strength, "Max"

for applying rule strengths, "Center-of-Gravity" for defuzzification. This

implementation of Inverted Pendulum control problem has: System Inputs, 2

(pendulum angle and velocity); System Outputs, 1 (force supplied to base of

pendulum); Membership Functions, 7 per system input/output; Rules, 15 (each

with 2 antecedents & 1 output). If more precision is required, integers can

be changed to real numbers.\*/

#include <stdio.h>

#define MAXNAME 10          /\* max number of characters in names   \*/

#define UPPER\_LIMIT  255    /\* max number assigned as degree of membership \*/

/\* io\_type structure builds a list of system inputs and a list of system

outputs. After initialization, these lists are fixed, except for value field

which is updated on every inference pass. \*/

struct io\_type{

  char name[MAXNAME];        /\*  name of system input/output       \*/

  int value;                 /\*  value of system input/output      \*/

  struct mf\_type             /\*  list of membership functions for  \*/

    \*membership\_functions;   /\*     this system input/output       \*/

  struct io\_type \*next;      /\*  pointer to next input/output      \*/

  };

/\* Membership functions are associated with each system input and output. \*/

struct mf\_type{

  char name[MAXNAME]; /\* name of membership function (fuzzy set)    \*/

  int value;          /\* degree of membership or output strength    \*/

  int point1;         /\* leftmost x-axis point of mem. function \*/

  int point2;         /\* rightmost x-axis point of mem. function    \*/

  int slope1;         /\* slope of left side of membership function  \*/

  int slope2;         /\* slope of right side of membership function \*/

  struct mf\_type \*next;   /\* pointer to next membership function    \*/

  };

/\*  Each rule has an if side and a then side. Elements making up if side are

pointers to antecedent values inside mf\_type structure. Elements making up then

side of rule are pointers to output strength values, also inside mf\_type

structure. Each rule structure contains a pointer to next rule in rule base. \*/

struct rule\_element\_type{

  int \*value;                /\* pointer to antecedent/output strength value \*/

  struct rule\_element\_type \*next; /\* next antecedent/output element in rule \*/

  };

struct rule\_type{

  struct rule\_element\_type \*if\_side;     /\* list of antecedents in rule \*/

  struct rule\_element\_type \*then\_side;   /\* list of outputs in rule     \*/

  struct rule\_type \*next;                /\* next rule in rule base  \*/

  };

struct rule\_type \*Rule\_Base;             /\* list of all rules in rule base \*/

**Listing Two:**

main()

{

 initialize\_system();

 while(1){

  get\_system\_inputs();

  fuzzification();

  rule\_evaluation();

  defuzzification();

  put\_system\_outputs();

  }

}

**Listing Three:**

/\* Fuzzification--Degree of membership value is calculated for each membership

function of each system input. Values correspond to antecedents in rules. \*/

fuzzification()

{

 struct io\_type \*si;    /\* system input pointer        \*/

 struct mf\_type \*mf;    /\* membership function pointer \*/

for(si=System\_Inputs; si != NULL; si=si->next)

  for(mf=si->membership\_functions; mf != NULL; mf=mf->next)

    compute\_degree\_of\_membership(mf,si->value);

}

/\* Rule Evaluation--Each rule consists of a list of pointers to antecedents

(if side), list of pointers to outputs (then side), and pointer to next rule

in rule base. When a rule is evaluated, its antecedents are ANDed together,

using a minimum function, to form strength of rule. Then strength is applied

to each of listed rule outputs. If an output has already been assigned a rule

strength, during current inference pass, a maximum function is used to

determine which strength should apply. \*/

rule\_evaluation()

{

 struct rule\_type \*rule;

 struct rule\_element\_type \*ip;       /\* pointer of antecedents  (if-parts)   \*/

 struct rule\_element\_type \*tp;       /\* pointer to consequences (then-parts) \*/

 int strength;                /\* strength of  rule currently being evaluated \*/

 for(rule=Rule\_Base; rule != NULL; rule=rule->next){

  strength = UPPER\_LIMIT;                       /\* max rule strength allowed \*/

        /\* process if-side of rule to determine strength \*/

  for(ip=rule->if\_side; ip != NULL; ip=ip->next)

      strength = min(strength,\*(ip->value));

       /\* process then-side of rule to apply strength \*/

  for(tp=rule->then\_side; tp != NULL; tp=tp->next)

      \*(tp->value) = max(strength,\*(tp->value));

  }

}

/\* Defuzzification \*/

defuzzification()

{

 struct io\_type \*so;    /\* system output pointer \*/

 struct mf\_type \*mf;    /\* output membership function pointer \*/

 int sum\_of\_products;   /\* sum of products of area & centroid \*/

 int sum\_of\_areas;  /\* sum of shortend trapezoid area \*/

 int area;

 int centroid;

 /\* compute a defuzzified value for each system output \*/

for(so=System\_Outputs; so != NULL; so=so->next){

  sum\_of\_products = 0;

  sum\_of\_areas = 0;

  for(mf=so->membership\_functions; mf != NULL; mf=mf->next){

     area = compute\_area\_of\_trapezoid(mf);

     centroid = mf->point1 + (mf->point2 - mf->point1)/2;

     sum\_of\_products += area \* centroid;

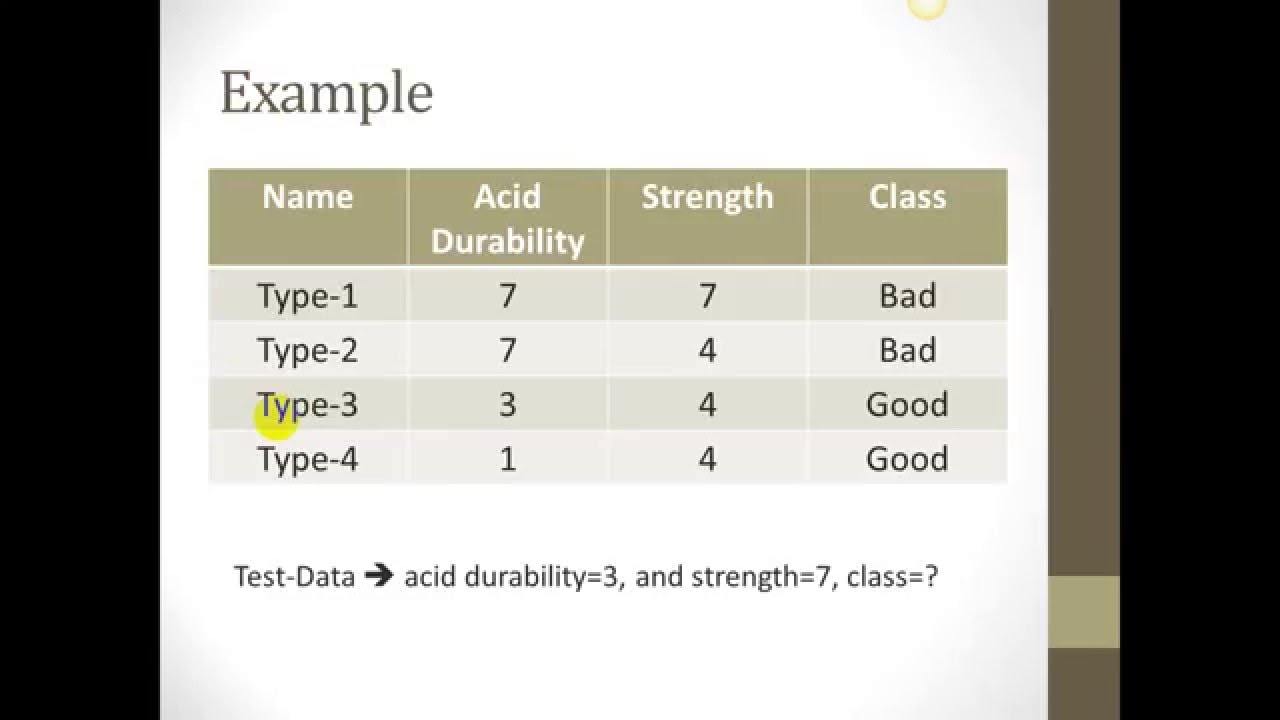
     sum\_of\_areas += area;

     }

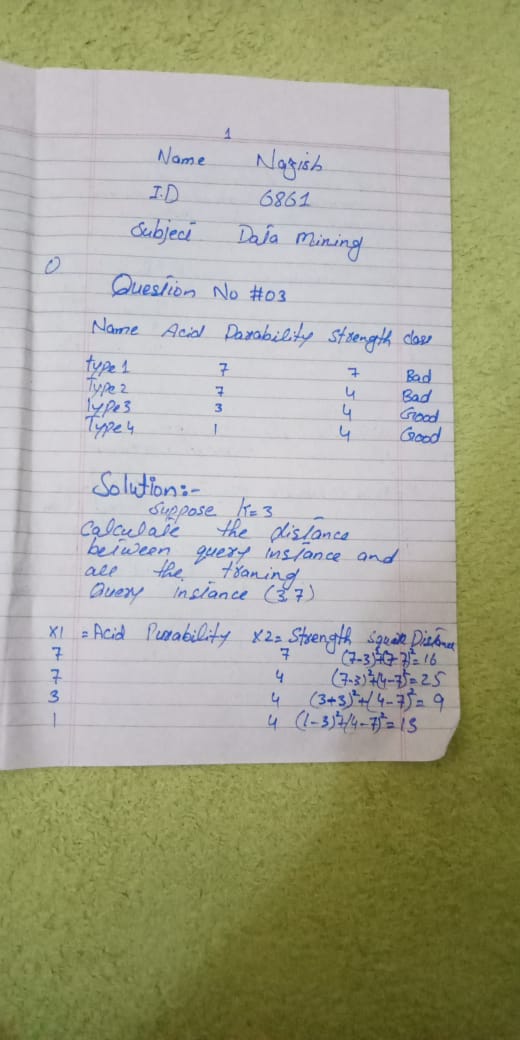
  so->value = sum\_of\_products/sum\_of\_areas;   /\* weighted average \*/

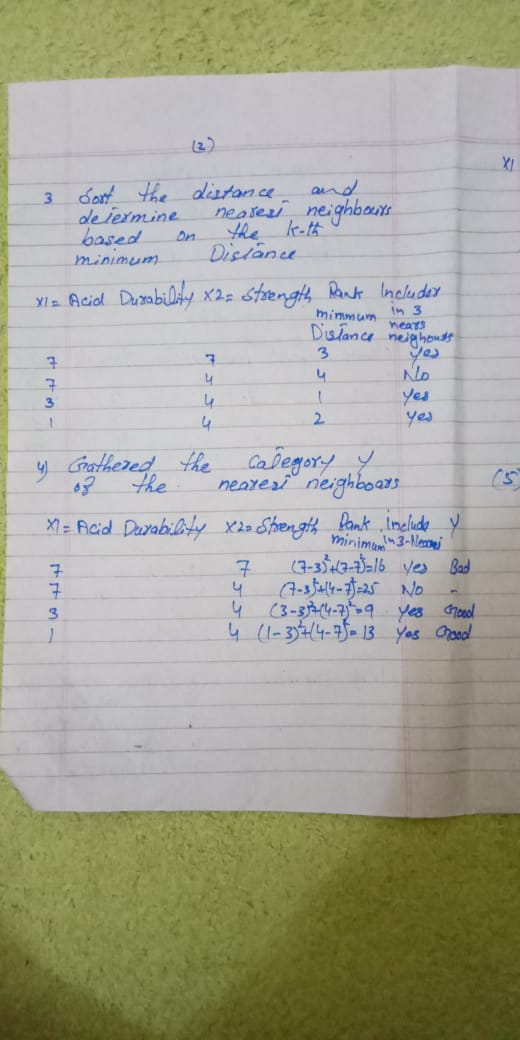
  }}

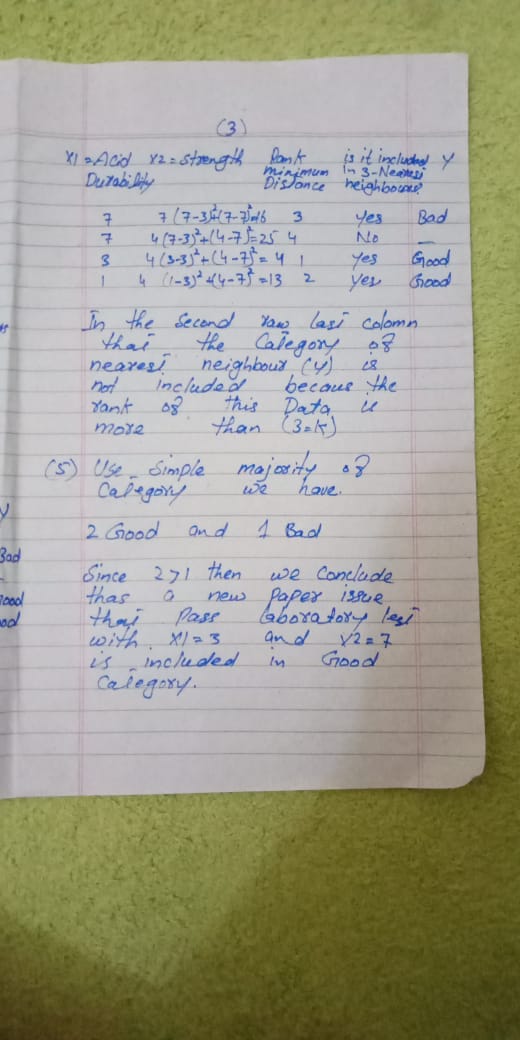
Q3. Solve this using KNN.



Answer





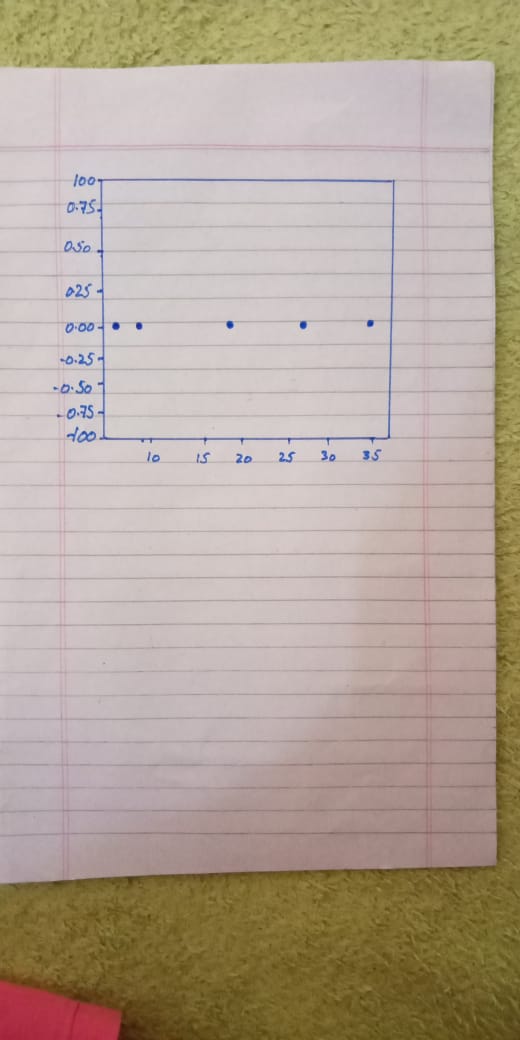


Q4. Give solved example of hierarchical Clustering. (15)

Answer

**Objective** : For the one dimensional data set **{7,10,20,28,35}**, perform hierarchical clustering and plot the dendogram to visualize it.

**Solution:** First, let’s visualize the data.

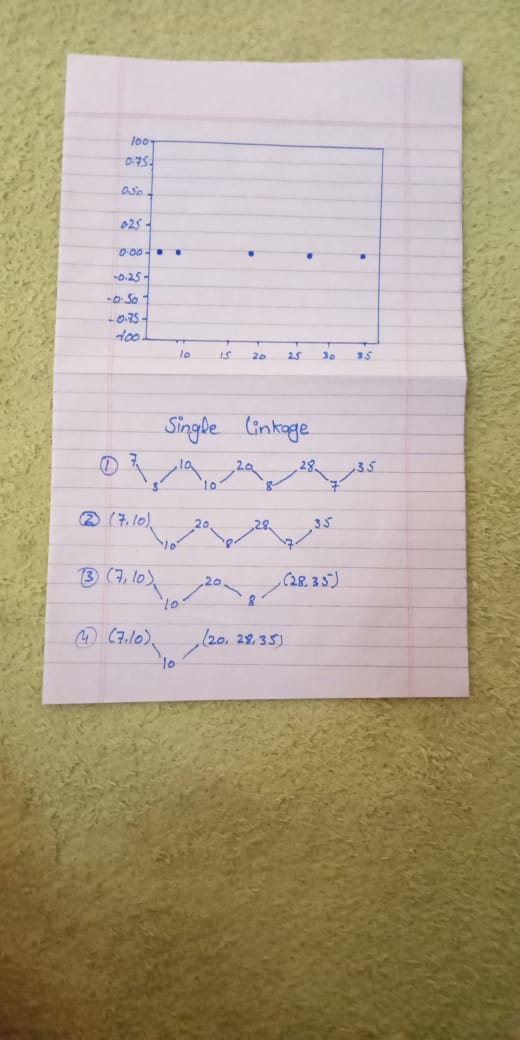


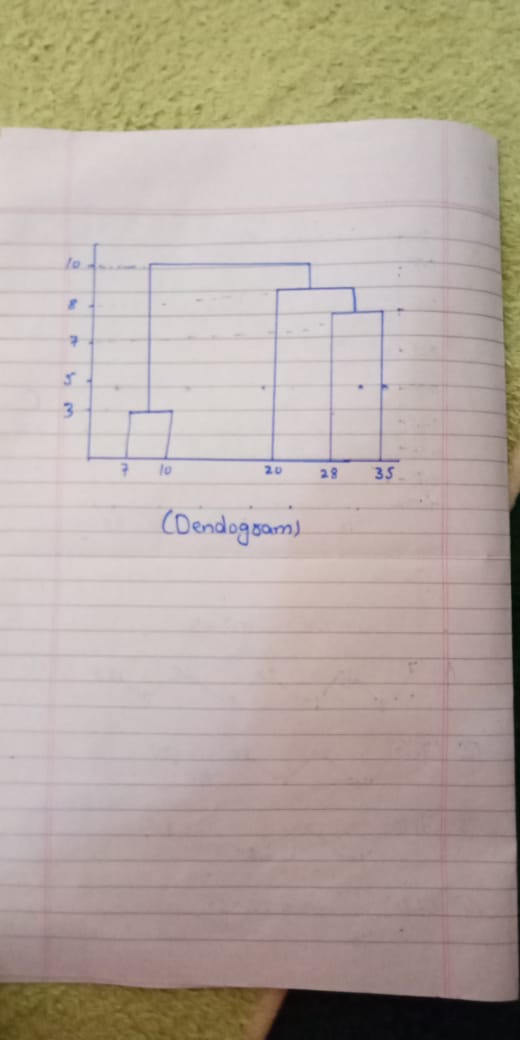
Observing the plot above, we can intuitively conclude that:

1. The first two points (7 and 10) are close to each other and should be in the same cluster
2. Also, the last two points (28 and 35) are close to each other and should be in the same cluster
3. Cluster of the center point (20) is not easy to conclude

Let’s solve the problem by hand using both the types of agglomerative hierarchical clustering :

1. **Single Linkage :**In single link hierarchical clustering, we merge in each step the two clusters, whose two closest members have the smallest distance.



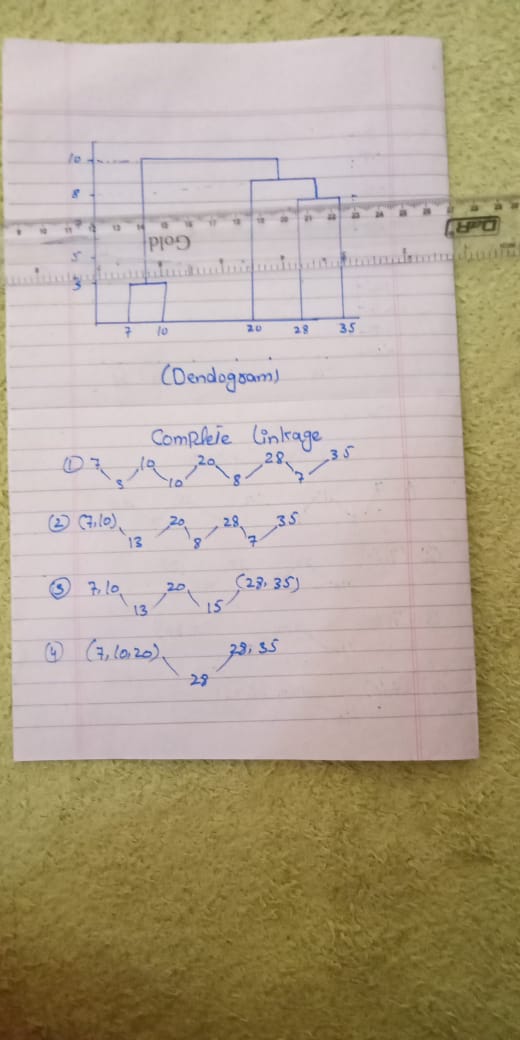


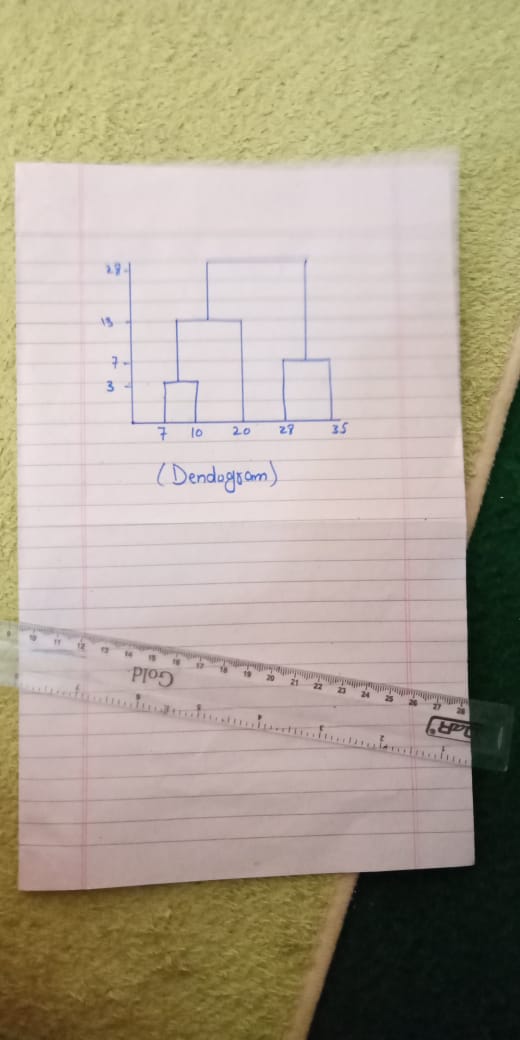
Using single linkage two clusters are formed :

Cluster 1 : (7,10)

Cluster 2 : (20,28,35)

1. **Complete Linkage :**In complete link hierarchical clustering, we merge in the members of the clusters in each step, which provide the smallest maximum pairwise distance.





Using complete linkage two clusters are formed :

Cluster 1 : (7,10,20)

Cluster 2 : (28,35)

**Conclusion :**Hierarchical clustering is mostly used when the application requires a hierarchy, e.g creation of a taxonomy. However, they are expensive in terms of their computational and storage requirements