

Name: **Ahmad Ali**

ID NO: **13473**

Degree: BS (SE)

Subject: Data Mining

Lecturer**: Mr.Zain shaukat**

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

Answer

// C++ program to create target string, starting from

// random string using Genetic Algorithm

#include <bits/stdc++.h>

using namespace std;

// Number of individuals in each generation

#define POPULATION\_SIZE 100

// Valid Genes

const string GENES = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP"\

"QRSTUVWXYZ 1234567890, .-;:\_!\"#%&/()=?@${[]}";

// Target string to be generated

const string TARGET = "I love GeeksforGeeks";

// Function to generate random numbers in given range

int random\_num(int start, int end)

{

    int range = (end-start)+1;

    int random\_int = start+(rand()%range);

    return random\_int;

}

// Create random genes for mutation

char mutated\_genes()

{

    int len = GENES.size();

    int r = random\_num(0, len-1);

    return GENES[r];

}

// create chromosome or string of genes

string create\_gnome()

{

    int len = TARGET.size();

    string gnome = "";

    for(int i = 0;i<len;i++)

        gnome += mutated\_genes();

    return gnome;

}

// Class representing individual in population

class Individual

{

public:

    string chromosome;

    int fitness;

    Individual(string chromosome);

    Individual mate(Individual parent2);

    int cal\_fitness();

};

Individual::Individual(string chromosome)

{

    this->chromosome = chromosome;

    fitness = cal\_fitness();

};

// Perform mating and produce new offspring

Individual Individual::mate(Individual par2)

{

    // chromosome for offspring

    string child\_chromosome = "";

    int len = chromosome.size();

    for(int i = 0;i<len;i++)

    {

        // random probability

        float p = random\_num(0, 100)/100;

        // if prob is less than 0.45, insert gene

        // from parent 1

        if(p < 0.45)

            child\_chromosome += chromosome[i];

        // if prob is between 0.45 and 0.90, insert

        // gene from parent 2

        else if(p < 0.90)

            child\_chromosome += par2.chromosome[i];

        // otherwise insert random gene(mutate),

        // for maintaining diversity

        else

            child\_chromosome += mutated\_genes();

    }

    // create new Individual(offspring) using

    // generated chromosome for offspring

    return Individual(child\_chromosome);

};

// Calculate fittness score, it is the number of

// characters in string which differ from target

// string.

int Individual::cal\_fitness()

{

    int len = TARGET.size();

    int fitness = 0;

    for(int i = 0;i<len;i++)

    {

        if(chromosome[i] != TARGET[i])

            fitness++;

    }

    return fitness;

};

// Overloading < operator

bool operator<(const Individual &ind1, const Individual &ind2)

{

    return ind1.fitness < ind2.fitness;

}

// Driver code

int main()

{

    srand((unsigned)(time(0)));

    // current generation

    int generation = 0;

    vector<Individual> population;

    bool found = false;

    // create initial population

    for(int i = 0;i<POPULATION\_SIZE;i++)

    {

        string gnome = create\_gnome();

        population.push\_back(Individual(gnome));

    }

    while(! found)

    {

        // sort the population in increasing order of fitness score

        sort(population.begin(), population.end());

        // if the individual having lowest fitness score ie.

        // 0 then we know that we have reached to the target

        // and break the loop

        if(population[0].fitness <= 0)

        {

            found = true;

            break;

        }

        // Otherwise generate new offsprings for new generation

        vector<Individual> new\_generation;

        // Perform Elitism, that mean 10% of fittest population

        // goes to the next generation

        int s = (10\*POPULATION\_SIZE)/100;

        for(int i = 0;i<s;i++)

            new\_generation.push\_back(population[i]);

        // From 50% of fittest population, Individuals

        // will mate to produce offspring

        s = (90\*POPULATION\_SIZE)/100;

        for(int i = 0;i<s;i++)

        {

            int len = population.size();

            int r = random\_num(0, 50);

            Individual parent1 = population[r];

            r = random\_num(0, 50);

            Individual parent2 = population[r];

            Individual offspring = parent1.mate(parent2);

            new\_generation.push\_back(offspring);

        }

        population = new\_generation;

        cout<< "Generation: " << generation << "\t";

        cout<< "String: "<< population[0].chromosome <<"\t";

        cout<< "Fitness: "<< population[0].fitness << "\n";

        generation++;

     }

     cout<< "Generation: " << generation << "\t";

    cout<< "String: "<< population[0].chromosome <<"\t";

    cout<< "Fitness: "<< population[0].fitness << "\n";

OUTPUT

Generation: 1 String: tO{"-?=jH[k8=B4]Oe@} Fitness: 18

Generation: 2 String: tO{"-?=jH[k8=B4]Oe@} Fitness: 18

Generation: 3 String: .#lRWf9k\_Ifslw #O$k\_ Fitness: 17

Generation: 4 String: .-1Rq?9mHqk3Wo]3rek\_ Fitness: 16

Generation: 5 String: .-1Rq?9mHqk3Wo]3rek\_ Fitness: 16

Generation: 6 String: A#ldW) #lIkslw cVek) Fitness: 14

Generation: 7 String: A#ldW) #lIkslw cVek) Fitness: 14

Generation: 8 String: (, o x \_x%Rs=, 6Peek3 Fitness: 13

.

.

.

Generation: 29 String: I lope Geeks#o, Geeks Fitness: 3

Generation: 30 String: I loMe GeeksfoBGeeks Fitness: 2

Generation: 31 String: I love Geeksfo0Geeks Fitness: 1

Generation: 32 String: I love Geeksfo0Geeks Fitness: 1

Generation: 33 String: I love Geeksfo0Geeks Fitness: 1

Generation: 34 String: I love GeeksforGeeks Fitness: 0

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

**Ans No 2:**

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;

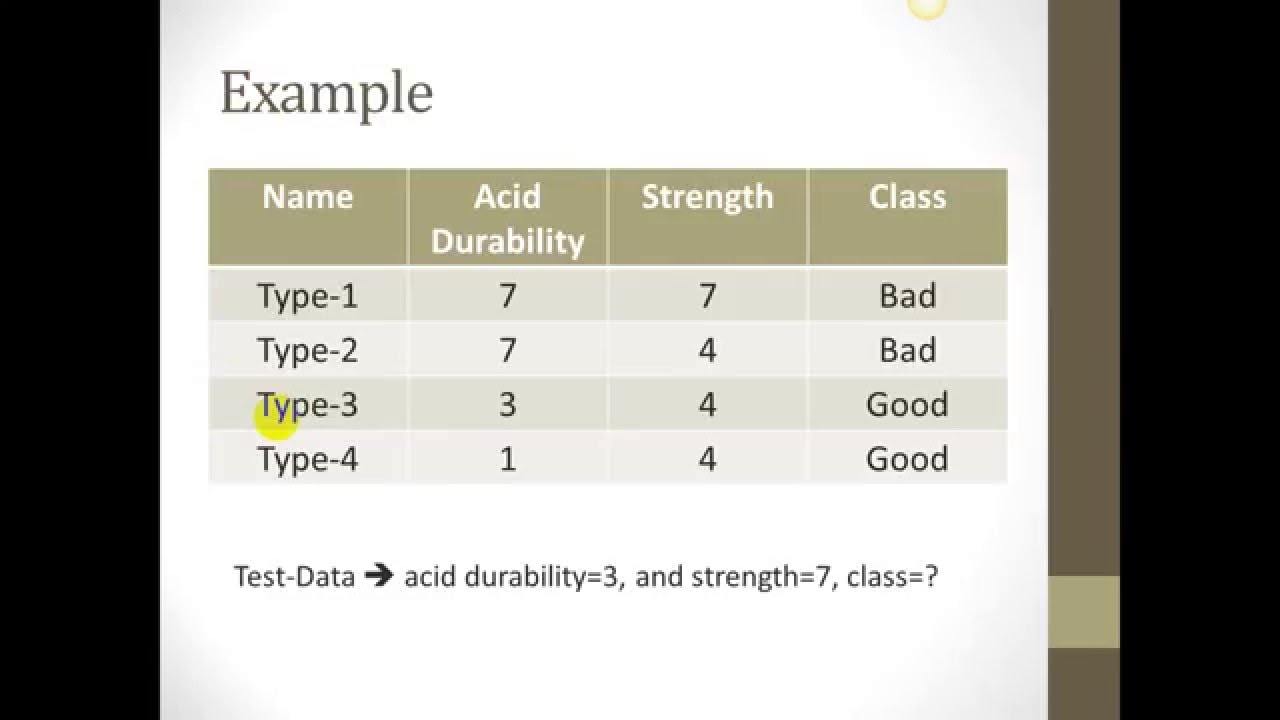
     sum\_of\_areas += area;

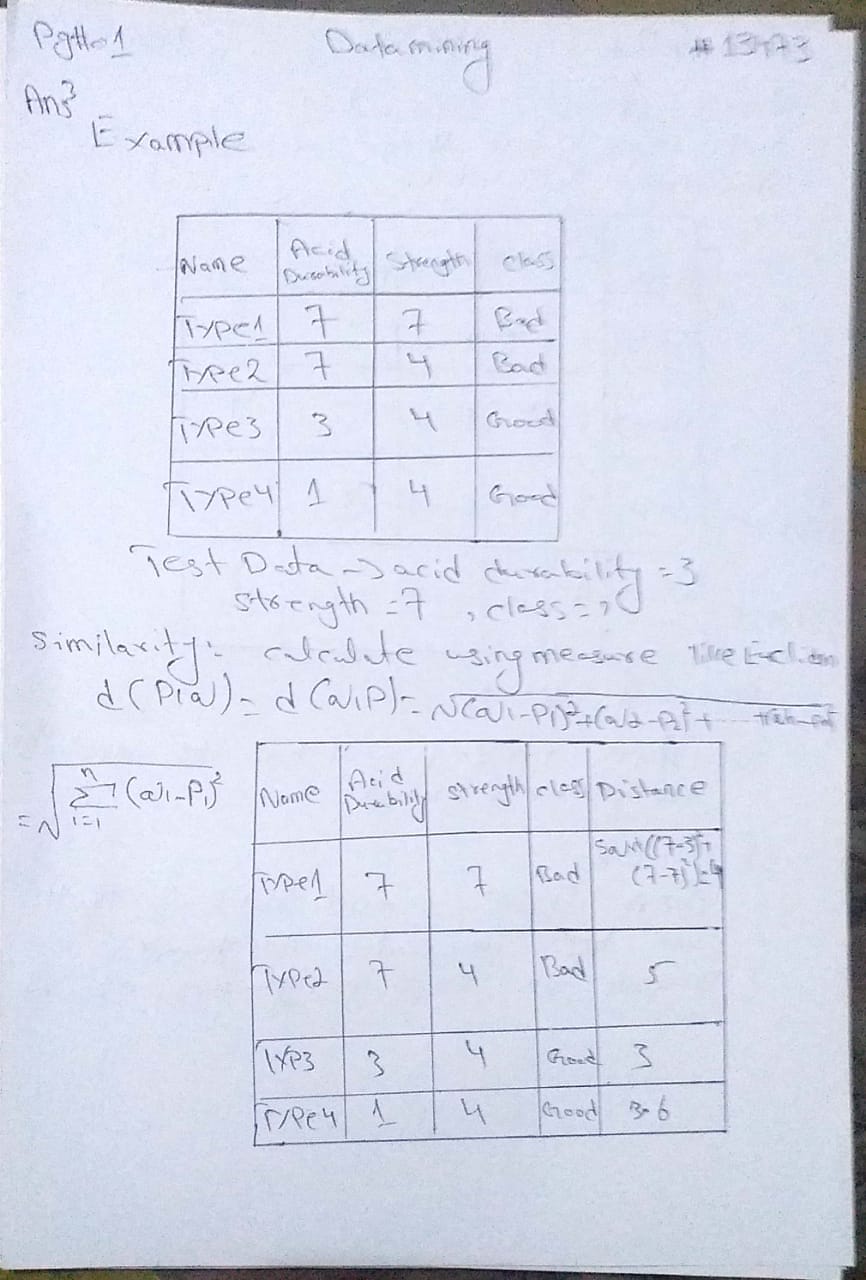
     }

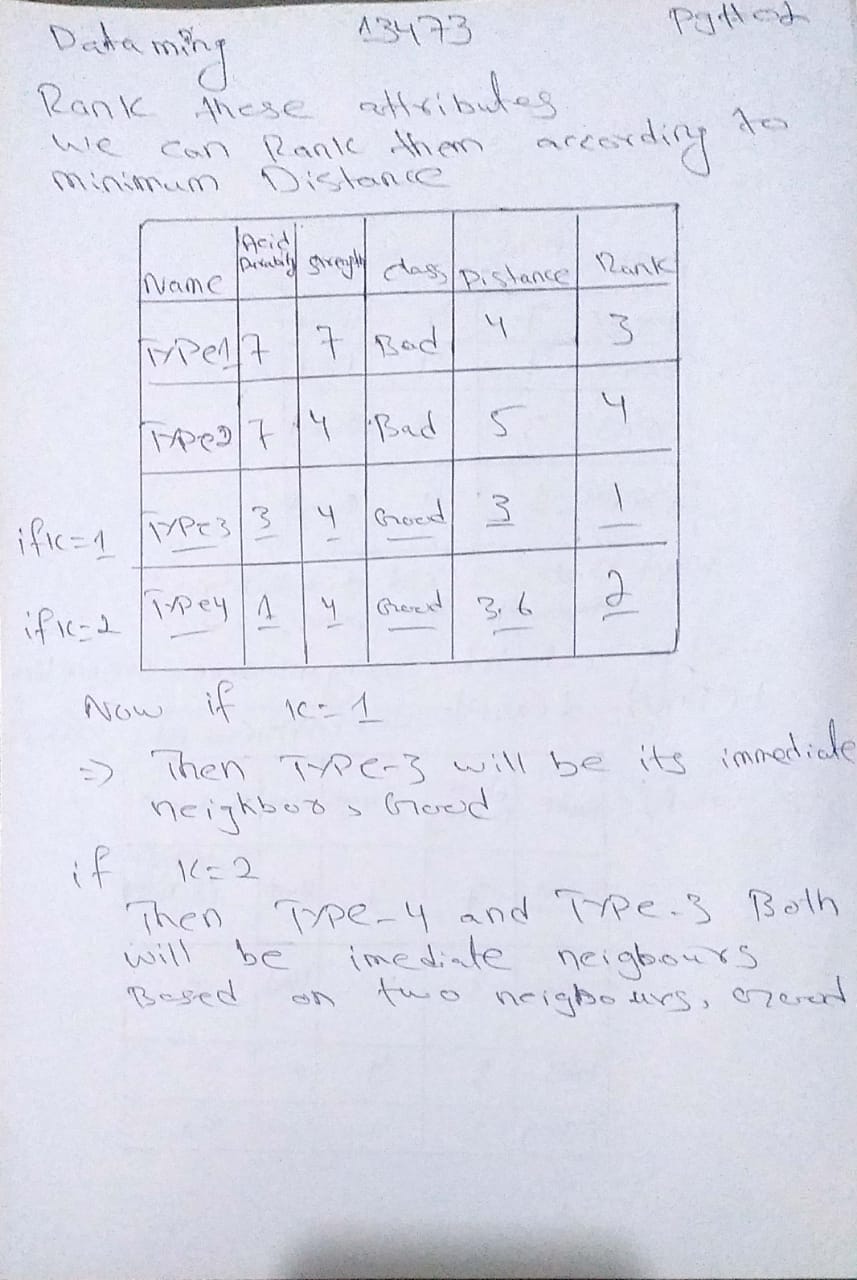
  so->value = sum\_of\_products/sum\_of\_areas;   /\* weighted average \*/

  }}

Q3. Solve this using KNN.



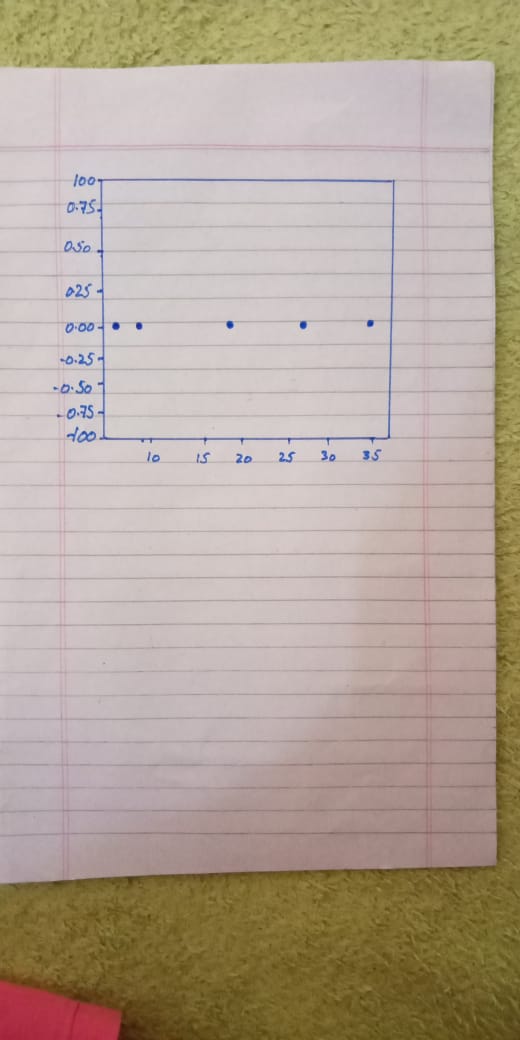




Q4. Give solved example of hierarchical Clustering.

**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.

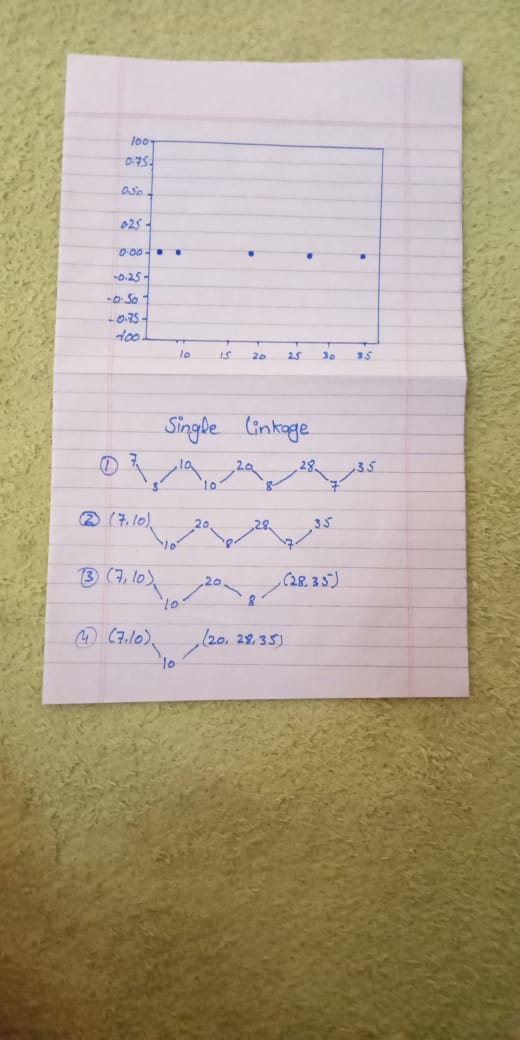


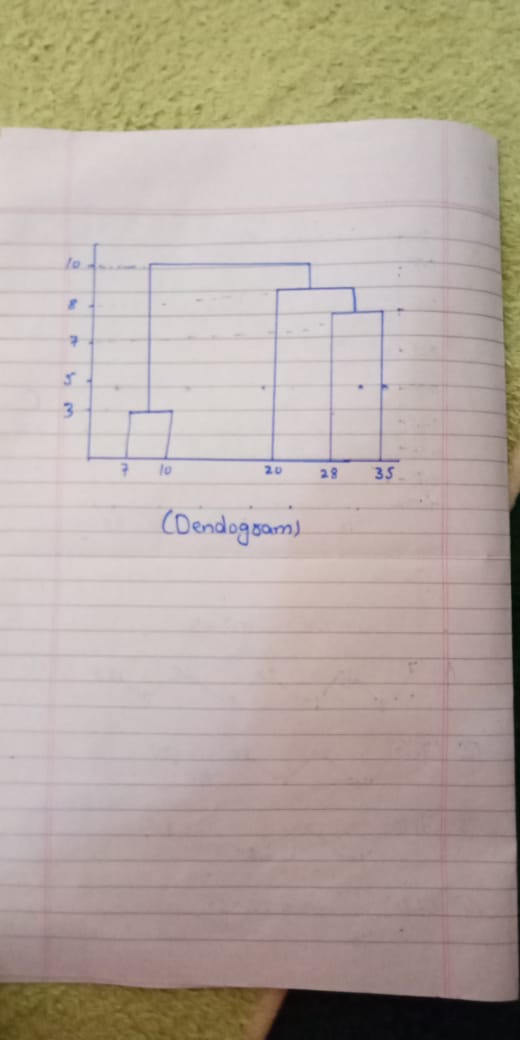
Observing the plot above, we can intuitively conclude that:

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.



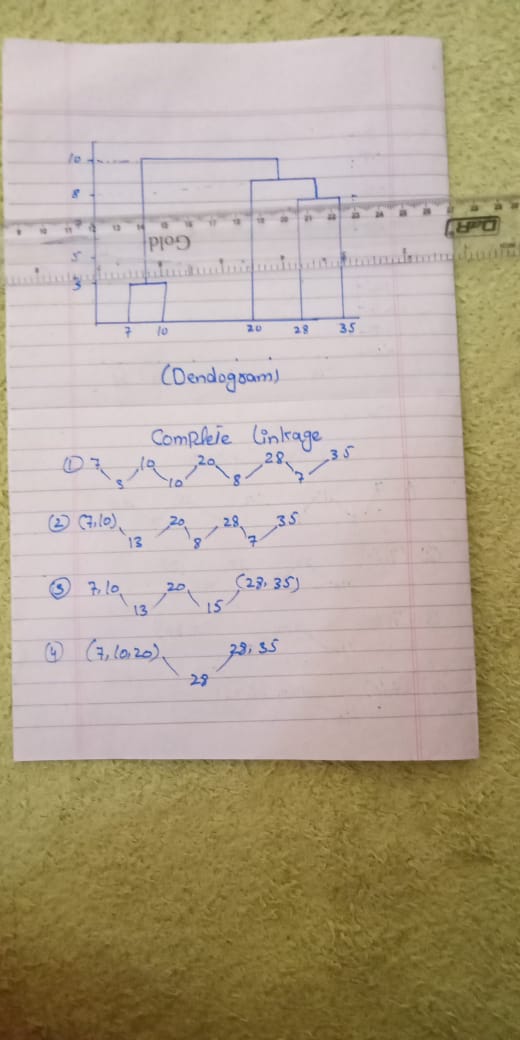


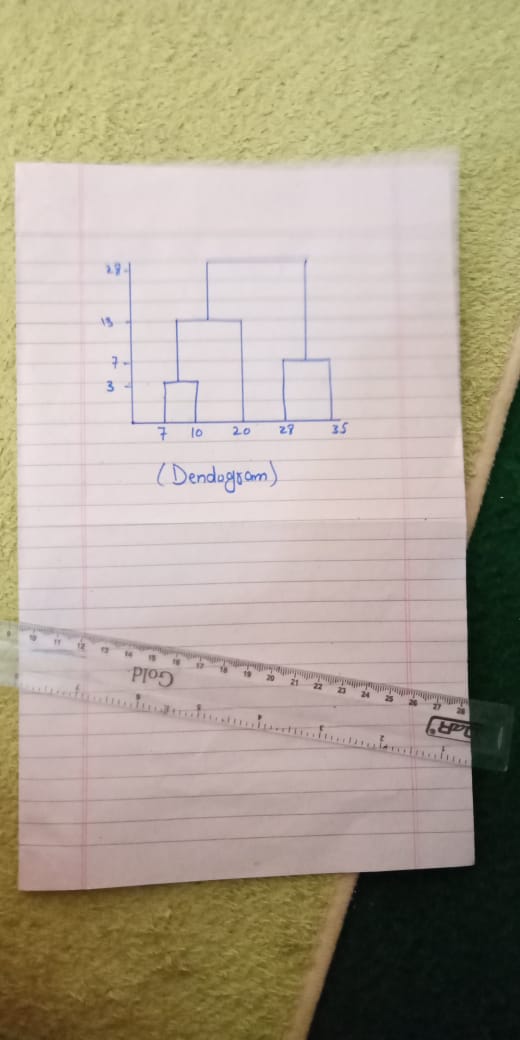
Using single linkage two clusters are formed :

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