In[1]:=
```
(* function to make an array image from the list *)
gridBin[dx_, dy_, data_] := Module[{maxVal, tempGrid, nx, ny, i},
   maxVal = Max[Flatten[data]];
   nx = maxVal / dx + 1;
   ny = maxVal / dy + 1;
   tempGrid = Table[0, {i, 1, nx}, {j, 1, ny}];
   For[i = 1, i <= Length[data], ++i,
    tempGrid[[Floor[data[[i, 1]] / dx] + 1, Floor[data[[i, 2]] / dy] + 1] ] =
     tempGrid[[Floor[data[[i, 1]] / dx] + 1, Floor[data[[i, 2]] / dy] + 1] ] + 10];
   tempGrid
  ];
```

In[2]:=
```
(* make a kernal for convolution *)
makeKernal[kernSize_] := Module[{kern},
   kern = Table[Exp[-(i^2 + j^2)], {i, -kernSize, kernSize}, {j, -kernSize, kernSize}] ;
   kern /= Total[Flatten[kern]];
   kern
  ];
```

In[3]:=
```
(* function to find peaks in the images made *)
findPeak[data_] := Module[{peakList, i, j, dims},
   peakList = Position[data, Max[data]];
   dims = Dimensions[data];
   For[j = 1, j < dims[[2]], ++j,
    For[i = 1, i < dims[[1]], ++i,

      If[{i, j} == peakList[[1]], Continue[]];

      If[data[[i, j]] > data[[ Mod[i + 1 + dims[[1]], dims[[1]], 1 ] , j]] &&
        data[[i, j]] > data[[ Mod[i - 1 + dims[[1]], dims[[1]], 1 ] , j]] &&
        data[[i, j]] > data[[i, Mod[j - 1 + dims[[2]], dims[[2]], 1 ]]] &&
        data[[i, j]] > data[[i, Mod[j + 1 + dims[[2]], dims[[2]], 1 ]]] &&
        data[[i, j]] >
         data[[Mod[i + 1 + dims[[1]], dims[[1]], 1 ], Mod[j + 1 + dims[[2]], dims[[2]], 1 ]]] &&
        data[[i, j]] > data[[Mod[i + 1 + dims[[1]], dims[[1]], 1 ],
          Mod[j - 1 + dims[[2]], dims[[2]], 1 ]]] &&
        data[[i, j]] > data[[Mod[i - 1 + dims[[1]], dims[[1]], 1 ],
          Mod[j + 1 + dims[[2]], dims[[2]], 1 ]]] &&
        data[[i, j]] > data[[Mod[i - 1 + dims[[1]], dims[[1]], 1 ],
          Mod[j - 1 + dims[[2]], dims[[2]], 1 ]]] &&
        data[[i, j]] > 1,
       peakList = Append[peakList, {i, j}]];
     ];
    ];
   DeleteDuplicates[peakList];
   peakList
  ];
```

In[4]:=
```
(* function to convert image indexes back to positions in real space *)
indexToLoc[index_, dx_, dy_] := {(index[[1]] - 0.5) * dx, (index[[2]] - 0.5) * dy};
```

In[5]:=
```
(* calculate the mahalanobis distances *)
mahalDist[locPeak_, locPoint_, cov_] =
  Sqrt[(locPeak - locPoint).Inverse[cov].(locPeak - locPoint)];
```

In[6]:=
```
(* partitions clusters based on mahalanobis distance *)
partitionClosest[data_, peakList_, covList_] :=
  Module[{parTable, tempTable, minDist, minIndex, dist, dataIndex, peakIndex},
   (* do partitioning *)
   tempTable = Table[{}, {i, 1, Length[peakList]}];
   For[dataIndex = 1, dataIndex ≤ Length[data], ++dataIndex,
    minDist = 1 000 000 000;
    For[peakIndex = 1, peakIndex ≤ Length[peakList], ++peakIndex,
     dist = mahalDist[peakList[[peakIndex]], data[[dataIndex]], covList[[peakIndex]]];
     If[dist < minDist,
      minDist = dist;
      minIndex = peakIndex;
     ];
    ];
    tempTable[[minIndex]] = Append[tempTable[[minIndex]], data[[dataIndex]]];
   ];

   (* remove empty lists *)
   For[peakIndex = 1, peakIndex ≤ Length[tempTable], ++peakIndex,
    If[0 == Length[tempTable[[peakIndex]]],
      tempTable = Drop[tempTable, {peakIndex}];
      --peakIndex
     ];
   ];

   tempTable
  ];
```

In[7]:=
```
(* constructs a covariance matrix through the origin to the peak *)
makeCovarMatrix[peakLoc_] := RotationMatrix[ArcTan[peakLoc[[1]], peakLoc[[2]]]].
   {{0.1, 0}, {0, 0.05}}.Transpose[RotationMatrix[ArcTan[peakLoc[[1]], peakLoc[[2]]]]];
```

In[8]:=
```
(* if list is greater than 1 return covar from function,
otherwise return a default matrix *)
safeCovar[set_] := If[Length[set] > 2, Covariance[set], makeCovarMatrix[set[[1]]]];
```

In[9]:=
```
(* calculates new means and std. dev. for clusters, merges close clusters *)
calcAndMerge[partList_, dist_] :=
  Module[{cenList, covList, i, j, newPartList, merged, localPartList},
   localPartList = partList;
   newPartList = {{}};

   (* calculate center and standard dev *)
   cenList = Table[Mean[partList[[i]]], {i, 1, Length[partList]}];
   covList = Table[safeCovar[partList[[i]]], {i, 1, Length[partList]}];

   (* see if we can merge any of the partitions *)
   For[i = 1, i <= Length[localPartList], ++i,
    merged = 0;
    For[j = i + 1, j ≤ Length[localPartList], ++j,
     If[mahalDist[cenList[[i]], cenList[[j]], covList[[i]] + covList[[j]]] ≤ dist,
       merged = 1;
       newPartList = Append[newPartList, Join[localPartList[[i]], localPartList[[j]]]];
       localPartList = Drop[localPartList, {j}];
       Break[];
      ];
    ];
    If[0 == merged , newPartList = Append[newPartList, localPartList[[i]]]];
   ];
   newPartList = Drop[newPartList, 1];

   (* if did merge any partitions call this function again recursively *)
   If[Length[partList] ≠ Length[newPartList],
    calcAndMerge[newPartList, dist], {cenList, covList}]
  ];
```

In[10]:=
```
(* actual clustering algorithm *)
cluster[data_, graphs_] :=
  Module[{dataImage, peakList, peakLocs, peakCovars,
    g1, g2, dx, dy, parts, prevParts, kernal, mahalBound, dims, i},

   (* convert data into an image *)
   dx = 0.1;
   dy = 0.1;
   dataImage = gridBin[dx, dy, data];

   (* convolve with kernal *)
   kernal = makeKernal[2];
   dataImage = ListConvolve[kernal, dataImage, {3, 3}];
```

```
(* find peaks in that image *)
peakList = findPeak[dataImage];
peakLocs = Table[indexToLoc[peakList[[i]], dx, dy], {i, 1, Length[peakList]}];
peakCovars = Table[makeCovarMatrix[peakLocs[[i]]], {i, 1, Length[peakLocs]}];

(* partition data once using peaks found in image *)
parts = partitionClosest[data, peakLocs, peakCovars];
prevParts = {{}};

If[1 == graphs,
 g1 = ListPlot[parts];
 g2 = ListPlot[peakLocs, PlotStyle -> Directive[Black, PointSize[Large]]];
 Print[Show[g1, g2]];
];

(* begin loop of partitioning and re-calculating *)
While[parts ≠ prevParts || Length[parts] > 3,

 (* save old version *)
 prevParts = parts;

 (* see if we need to modify dist *)
 mahalBound = If[Length[parts] ≤ 3 || parts ≠ prevParts ,
    3,
    Min[Table[ If[i ≠ j,
       mahalDist[Mean[parts[[i]]],
        Mean[parts[[j]]], safeCovar[parts[[i]]] + safeCovar[parts[[j]]]],
       10 000 000
      ], {i, 1, Length[parts]}, {j, 1, Length[parts]}]]
  ];

 (* do the calculation *)
 {peakLocs, peakCovars} = calcAndMerge[parts, mahalBound];
 parts = partitionClosest[data, peakLocs, peakCovars];

 (* graph it *)
 If[1 == graphs,
  g1 = ListPlot[parts];
  g2 = ListPlot[peakLocs, PlotStyle -> Directive[Black, PointSize[Large]]];
  Print[Show[g1, g2]];
 ];
];
```

```
    (* return the final partitioning *)
    parts
   ];
```

In[11]:=
```
(* sort partitions based on the ratio of y/x of the center of the cluster *)
sortPartitions[parts_] := Module[{cenList, ratioList, sortedParts, i, j, center},
   (* get the centers of the partitions *)
   cenList = Table[Mean[parts[[i]]], {i, 1, Length[parts]}];

   (* calculate ratios of x and y *)
   ratioList = Sort[Table[cenList[[i, 2]] / cenList[[i, 1]], {i, 1, Length[cenList]}]];

   (* use sorted ratios to re-order partitions *)
   sortedParts = parts;
   For[j = 1, j ≤ Length[cenList], ++j,
    For[i = 1, i ≤ Length[parts], ++i,
      center = Mean[parts[[i]]] ;
      If[center[[2]] / center[[1]] == ratioList[[j]], sortedParts[[j]] = parts[[i]]];
     ];
   ];
   sortedParts
  ];
```

In[12]:=
```
(* assign calls and SNP ID's to the data points in the partition *)
convertToIDs[parts_, syms_, lookup_] :=
  Module[{tempTab, index, index2, index3, results, thisPart, found},
   results = parts;
   For[index = 1, index ≤ Length[parts], ++index,
    thisPart = parts[[index]];
    For[index2 = 1, index2 ≤ Length[thisPart], ++index2,
     For[index3 = 1, index3 ≤ Length[lookup] , ++index3,
       If[thisPart[[index2, 1]] == lookup[[index3, 1]] &&
          thisPart[[index2, 2]] == lookup[[index3, 2]],
         thisPart[[index2]] = {lookup[[index3, 3]], syms[[index]]};
         Break[];
        ];
      ];
    ];
    results[[index]] = thisPart;
   ];
   results
  ];
```

In[13]:=
```
(* make a list of possible calls sorted by ratio *)
makeSymList[parts_] := Module[{numParts, localSyms, cenList, ratioList, i},
   localSyms = {"A", "H", "B"};
   cenList = Table[Mean[parts[[i]]], {i, 1, Length[parts]}];
   ratioList = Sort[Table[cenList[[i, 2]] / cenList[[i, 1]], {i, 1, Length[cenList]}]];

   (* case of 2 groups *)
   If[2 == Length[parts] ,
    If[ Abs[ratioList[[1]] - 1.0] < Abs[ratioList[[2]] - 1.0],
      localSyms = {"H", "B"},
      localSyms = {"A", "H"}
     ];
   ];

   (* case of 1 group *)
   If[1 == Length[parts],
    If[ Abs[ratioList[[1]] - 0.7] < Abs[ratioList[[1]] - 1] &&
      Abs[ratioList[[1]] - 0.7] < Abs[ratioList[[1]] - 1.3], localSyms = {"A"};];
    If[ Abs[ratioList[[1]] - 1] < Abs[ratioList[[1]] - 1.3] &&
      Abs[ratioList[[1]] - 1] < Abs[ratioList[[1]] - 0.7], localSyms = {"H"};];
    If[ Abs[ratioList[[1]] - 1.3] < Abs[ratioList[[1]] - 1] &&
      Abs[ratioList[[1]] - 1.3] < Abs[ratioList[[1]] - 0.7], localSyms = {"B"};];
   ];
   localSyms
  ];
```

In[14]:=
```
(* function to compare my calls with Affy calls *)
compareToTruth[parts_, lookup_, truth_] :=
  Module[{ids, localSyms, sortedParts, callList, i, a, percentCorrect, g1, g2, aa},
   (* sort the partitions *)
   sortedParts = sortPartitions[parts];

   (* make a list of possible IDs *)
   localSyms = makeSymList[sortedParts];

   (* associate calls with IDs *)
   ids = Sort[Flatten[convertToIDs[sortedParts, localSyms, lookup], 1]];

   (* strip off the IDs after sorting *)
   callList = Table[ids[[i, 2]], {i, 1, Length[ids]}];

   (* print if tables are equal length *)
   Print[Length[callList] == Length[truth]];

   (* print the table of truth if not same as call *)
   a = Table[ If[callList[[i]] == truth[[i]], "T", truth[[i]]], {i, 1, Length[truth]}];
   Print[a];

   (* print the table of calls if not same as truth *)
   Print[
    Table[ If[callList[[i]] == truth[[i]], "T", callList[[i]]], {i, 1, Length[truth]}]];

   (* print % correct *)
   percentCorrect = Count[a, "T"] / (Length[a] - Count[a, "NN"]) ;
   Print[percentCorrect // N];

   (* print graph of mistakes *)
   aa = Table[ If[callList[[i]] != truth[[i]] && "NN" ≠ truth[[i]],
      {lookup[[i, 1]], lookup[[i, 2]]} , {0, 0}], {i, 1, Length[truth]}];
   g1 = ListPlot[sortedParts, PlotStyle → {Red, Green, Blue}];
   g2 = ListPlot[aa, PlotStyle -> Directive[Black, PointSize[Large]]];
   Print[Show[g2, g1]];

   percentCorrect
  ];
```

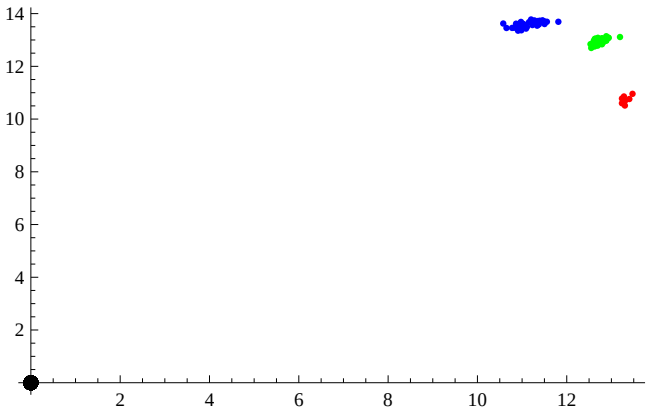
In[15]:=
```
fileList = ReadList["G:/cs224/NickData/chopped/fileListingAll.txt", Word];
names = ReadList["G:/cs224/NickData/rowOrder.names", Word];
results = Table[-1, {i, 1, 10}];
For[index = 1, index ≤ 10, ++index,
  probeA =
   (ReadList["G:/cs224/nickData/chopped/" <> fileList[[index]] <> "_sA.txt", Number] +
      ReadList["G:/cs224/nickData/chopped/" <>
        fileList[[index]] <> "_asA.txt", Number]) / 2;
  probeB = (ReadList["G:/cs224/nickData/chopped/" <> fileList[[index]] <> "_sB.txt",
       Number] + ReadList["G:/cs224/nickData/chopped/" <>
        fileList[[index]] <> "_asB.txt", Number]) / 2;

  answers = ReadList["G:/cs224/nickData/chopped/" <> fileList[[index]] <> "_ans.txt", Word];

  (* make tables for later *)
  data = Table[{probeA[[i]], probeB[[i]]}, {i, 1, 90}];
  lookup = Table[{probeA[[i]], probeB[[i]], names[[i]]}, {i, 1, 90}];

  (* cluster and compare *)
  res = cluster[data, 0];
  results[[index]] = compareToTruth[res, lookup, answers];
 ];
```

```
True
```

```
{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}
```

```
{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}
```

```
1.
```

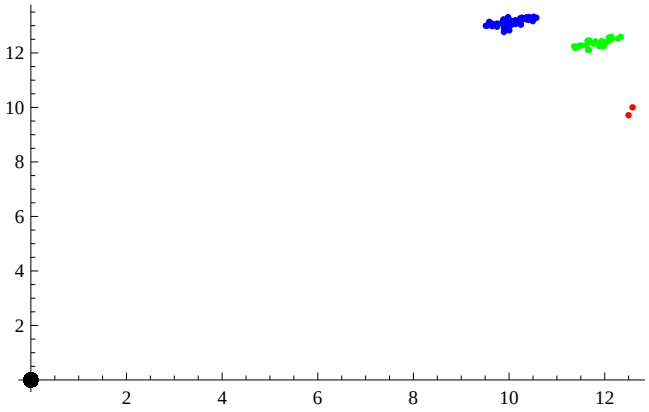



```
True
```

```
{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}
```

```
{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}
```

```
1.
```

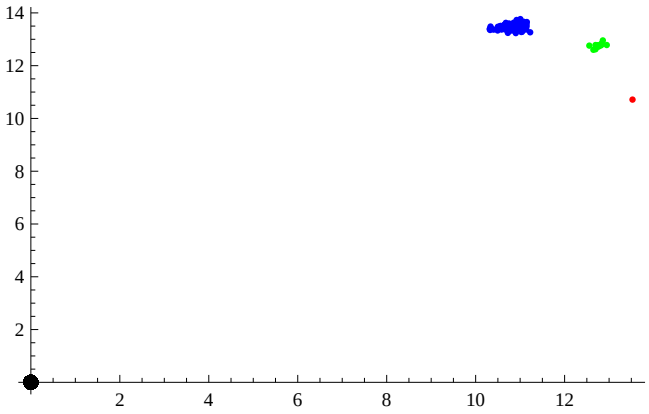



```
True
```

```
{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}
```

```
{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}
```

```
1.
```

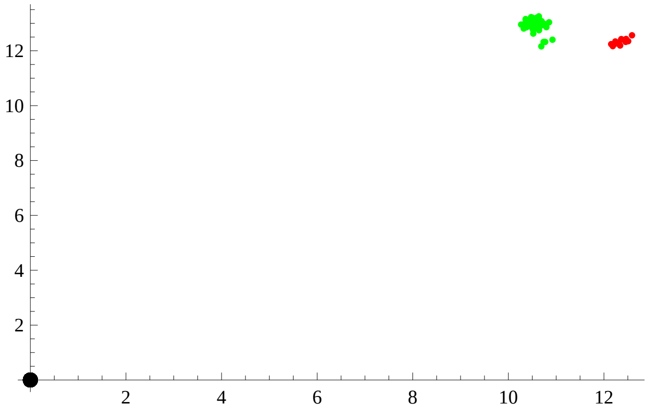



```
True
```

```
{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}
```

```
{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}
```

```
1.
```

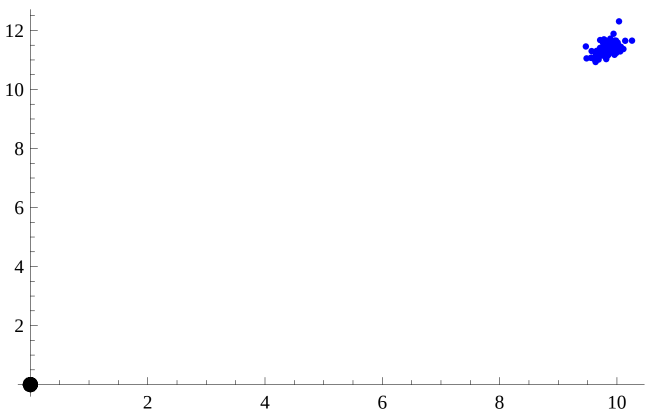


True

{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}

{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}

1.

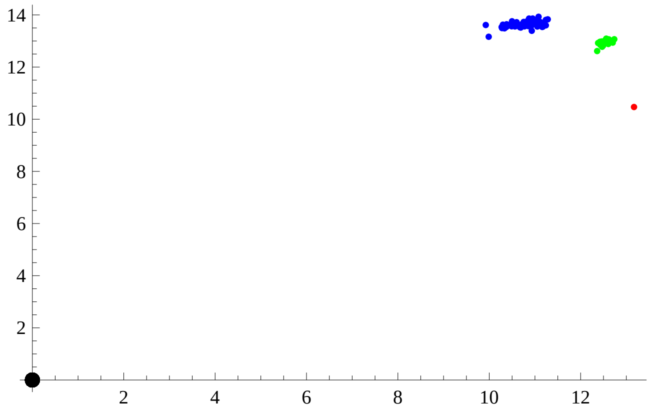


True

{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}

{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}

1.

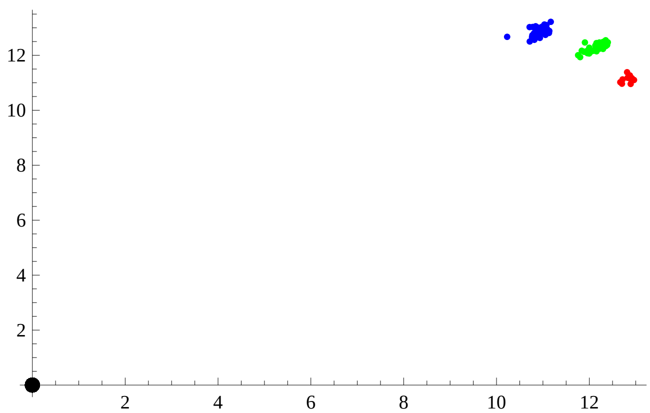


True

{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}

{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}

1.

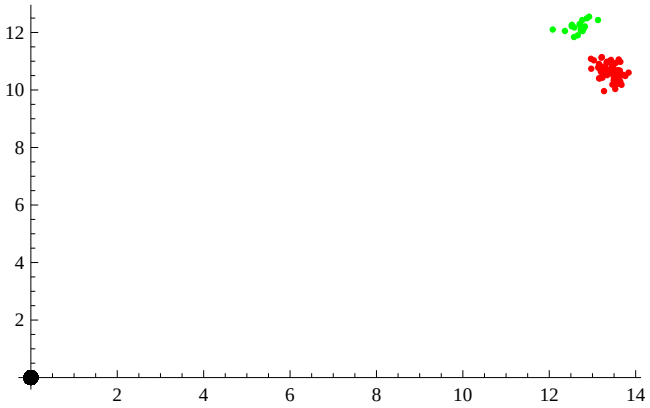


True

{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}

{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}

1.

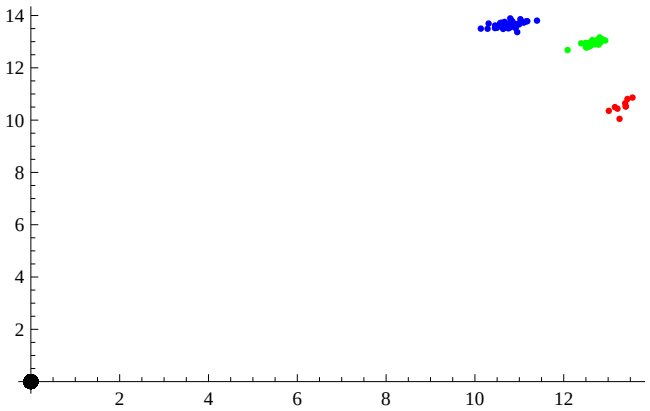



```
True
```

```
{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}
```

```
{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}
```

```
1.
```

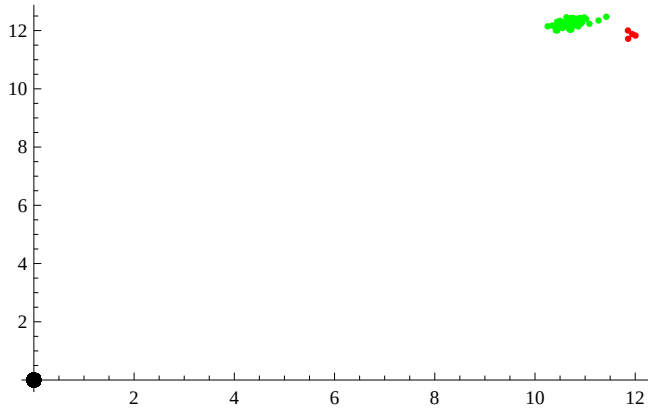



```
True
```

```
{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}
```

```
{T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T,
 T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T, T}
```

```
1.
```


12
10
8
6
4
2
2
4
6
8
10
12