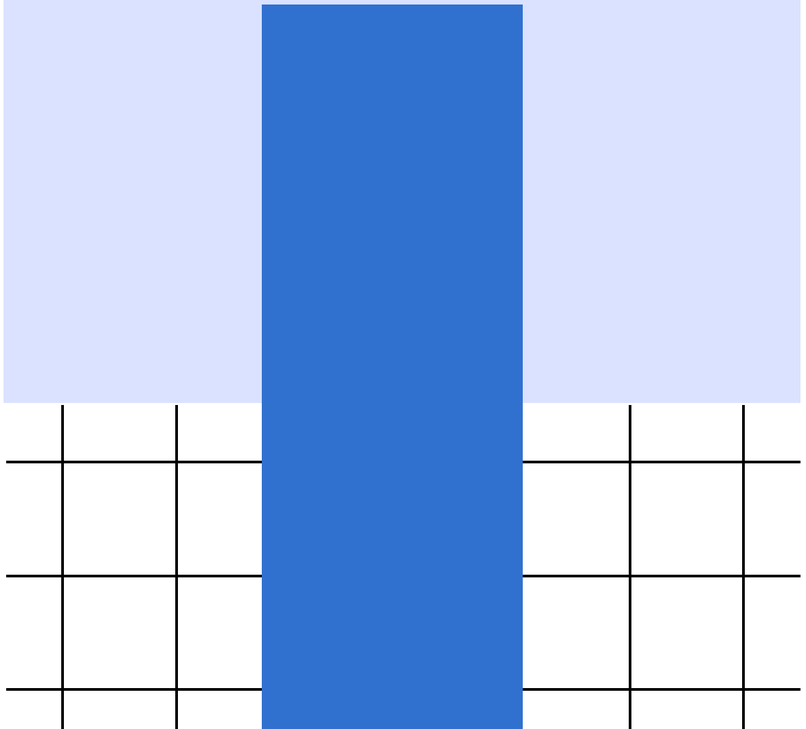
CLUSTERING ALGORITHMS

– Matee Vadrukchid –



DBSCAN

A Density-Based Algorithm for Discovering Clusters
A Density-Based Algorithm for Discovering Clusters
in Large Spatial Databases with Noise

Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu

Institute for Computer Science, University of Munich
Oettingenstr. 67, D-80538 München, Germany
{ester | kriegel | sander | xwxu}@informatik.uni-muenchen.de

Abstract

Clustering algorithms are attractive for the task of class identification in spatial databases. However, the application to large spatial databases rises the following requirements for clustering algorithms: minimal requirements of domain knowledge to determine the input parameters, discovery of clusters with arbitrary shape and good efficiency on large databases. The well-known clustering algorithms offer no solution to the combination of these requirements. In this paper, we present the new clustering algorithm DBSCAN relying on a density-based notion of clusters which is designed to discover clusters of arbitrary shape. DBSCAN requires only one input parameter and supports the user in determining an appropriate value for it. We performed an experimental evaluation of the effectiveness and efficiency of DBSCAN using synthetic data and real data of the SEQUOIA 2000 benchmark. The results of our experiments demonstrate that (1) DBSCAN is significantly more effective in discovering clusters of arbitrary shape than the well-known algorithm CLARANS, and that (2) DBSCAN outperforms CLARANS by a factor of more than 100 in terms of efficiency.

Keywords: Clustering Algorithms, Arbitrary Shape of Clusters, Efficiency on Large Spatial Databases, Handling Nlj4-275oise.

1. Introduction

Numerous applications require the management of *spatial* data, i.e. data related to space. *Spatial Database Systems (SDBS)* (Gueting 1994) are database systems for the management of spatial data. Increasingly large amounts of data are obtained from satellite images, X-ray crystallography or other automatic equipment. Therefore, automated knowledge discovery becomes more and more important in spatial databases.

Several tasks of *knowledge discovery in databases* (KDD) have been defined in the literature (Matheus, Chan & Piatetsky-Shapiro 1993). The task considered in this paper is *class identification*, i.e. the grouping of the objects of a database into meaningful subclasses. In an earth observation database, e.g., we might want to discover classes of houses along some river.

are often not known in advance when dealing with large databases.

(2) Discovery of clusters with arbitrary shape, because the shape of clusters in spatial databases may be spherical, drawn-out, linear, elongated etc.

(3) Good efficiency on large databases, i.e. on databases of significantly more than just a few thousand objects.

The well-known clustering algorithms offer no solution to the combination of these requirements. In this paper, we present the new clustering algorithm DBSCAN. It requires only one input parameter and supports the user in determining an appropriate value for it. It discovers clusters of arbitrary shape. Finally, DBSCAN is efficient even for large spatial databases. The rest of the paper is organized as follows. We discuss clustering algorithms in section 2 evaluating them according to the above requirements. In section 3, we present our notion of clusters which is based on the concept of density in the database. Section 4 introduces the algorithm DBSCAN which discovers such clusters in a spatial database. In section 5, we performed an experimental evaluation of the effectiveness and efficiency of DBSCAN using synthetic data and data of the SEQUOIA 2000 benchmark. Section 6 concludes with a summary and some directions for future research.

2. Clustering Algorithms

There are two basic types of clustering algorithms (Kaufman & Rousseeuw 1990): partitioning and hierarchical algorithms. *Partitioning algorithms* construct a partition of a database D of n objects into a set of k clusters. k is an input parameter for these algorithms, i.e some domain knowledge is required which unfortunately is not available for many applications. The partitioning algorithm typically starts with an initial partition of D and then uses an iterative control strategy to optimize an objective function. Each cluster is represented by the gravity center of the cluster *(k-means algorithms)* or by one of the objects of the cluster located near its center (*k-medoid algorithms*). Consequently, partitioning

An Improved DBSCAN, A Density Based Clustering Algorithm with Parameter Selection for High Dimensional Data Sets

Glory H.Shah

Abstract*— Emergence of modern techniques for scientific data collection has resulted in large scale accumulation of data pertaining to diverse fields. Cluster analysis is one of the major data analysis methods. It is the art of detecting group of similar objects in large data sets without having specified groups by means of explicit features. The problem of detecting clusters is challenging when the clusters are of different size, density and shape. This paper gives a new approach towards density based clustering approach. DBSCAN which is considered a pioneer of density based clustering technique, this paper gives a new move towards detecting clusters that exists within a cluster. Based on various parameters needed for a good clustering the algorithm is evaluated such as number of clusters formed, noise ratio on distance change, time elapsed to form cluster, unclustered instances as well as incorrectly clustered instances.

Index Terms*-- Inter cluster, DBSCAN, Spatial Data, High dimensional.

I.INTRODUCTION

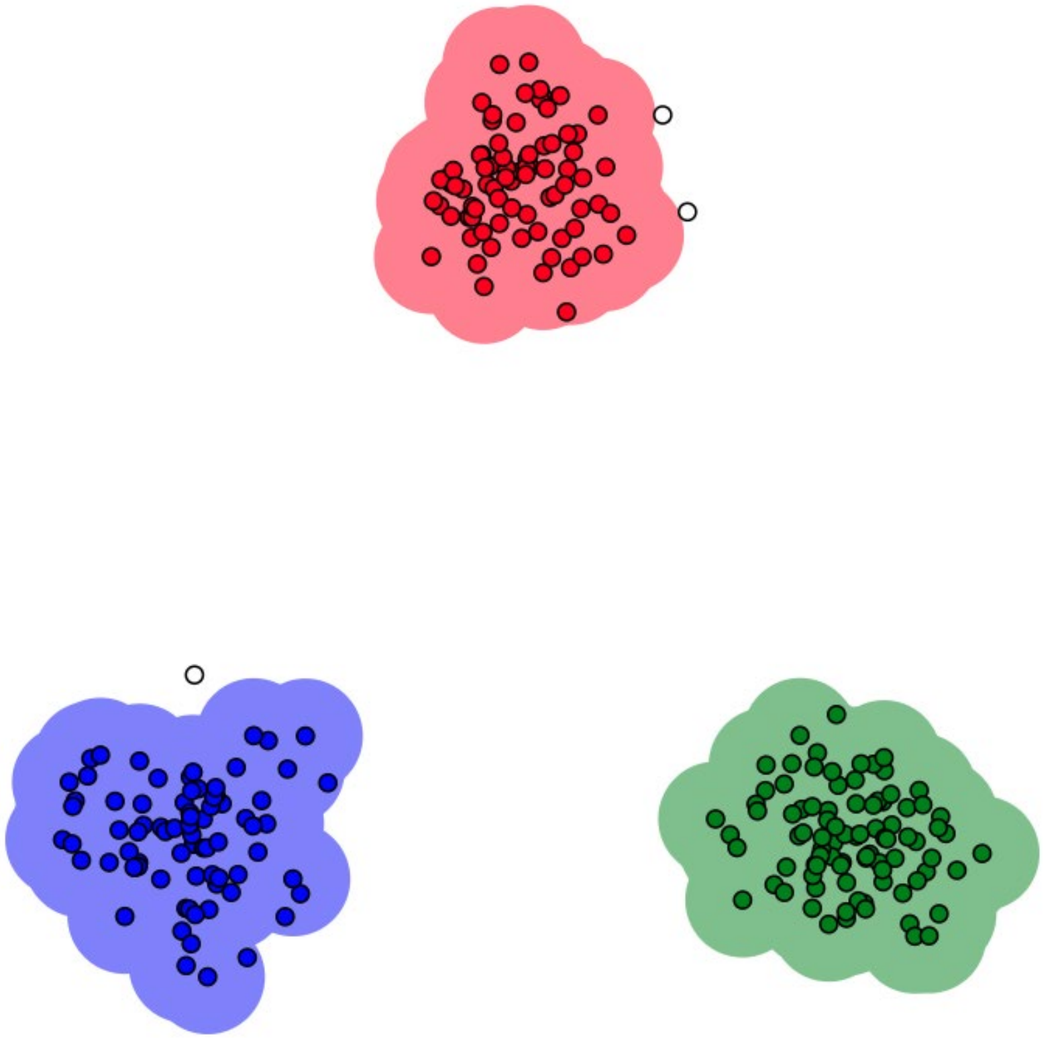
Clustering is an initial and fundamental step in data analysis. It can be considered as an unsupervised classification of patterns into groups. Clustering Techniques are useful in various fields such as mining skin lesion images, pattern analysis, machine learning situations and many other fields.

requirements: [1] each group must contain at least one object and [2] each object must belong to exactly one group.

In *Hierarchical Method*, it creates a hierarchical decomposition of the given set of data objects. It can be either agglomerative or divisive, based on how hierarchical decomposition is formed. The *agglomerative approach*, also called the *bottom-up* approach, starts with each object forming a separate group. It successively merges the objects or groups that are close to one another, until all of the groups are merged into one (the topmost level of the hierarchy), or until a termination condition holds. The *divisive approach*, also called the *top-down* approach, starts with all of the objects in the same cluster. In each successive iteration, a cluster is split up into smaller clusters, until eventually each object is in one cluster, or until a termination condition holds.

In *Density-Based Method*, most partitioning methods cluster objects based on the distance between objects. Such methods can find arbitrary shaped clusters. The general idea here is to continue growing the given cluster as long as the density or say number of objects or data points in the neighborhood exceeds some threshold. Such methods can be used to filter our noise or outliers.

LINK: HTTPS://WWW.NAFTALIHARRIS.COM/BLOG/VISUALIZING - DBSCAN - CLUSTERING//



4.1 The Algorithm

To find a cluster, DBSCAN starts with an arbitrary point p and retrieves all points density-reachable from p wrt. Eps and MinPts. If p is a core point, this procedure yields a cluster wrt. Eps and MinPts (see Lemma 2). If p is a border point, no points are density-reachable from p and DBSCAN visits the next point of the database.

Since we use global values for Eps and MinPts, DBSCAN may merge two clusters according to definition 5 into one cluster, if two clusters of different density are "close" to each other. Let the *distance between two sets of points* S_1 and S_2 be defined as $dist(S_1, S_2) = min\{dist(p,q) \mid p \in S_1, q \in S_2\}$. Then, two sets of points having at least the density of the thinnest cluster will be separated from each other only if the distance between the two sets is larger than Eps. Consequently, a recursive call of DBSCAN may be necessary for the detected clusters with a higher value for MinPts. This is, however, no disadvantage because the recursive application of DBSCAN yields an elegant and very efficient basic algorithm. Furthermore, the recursive clustering of the points of a cluster is only necessary under conditions that can be easily detected.

In the following, we present a basic version of DBSCAN omitting details of data types and generation of additional information about clusters:

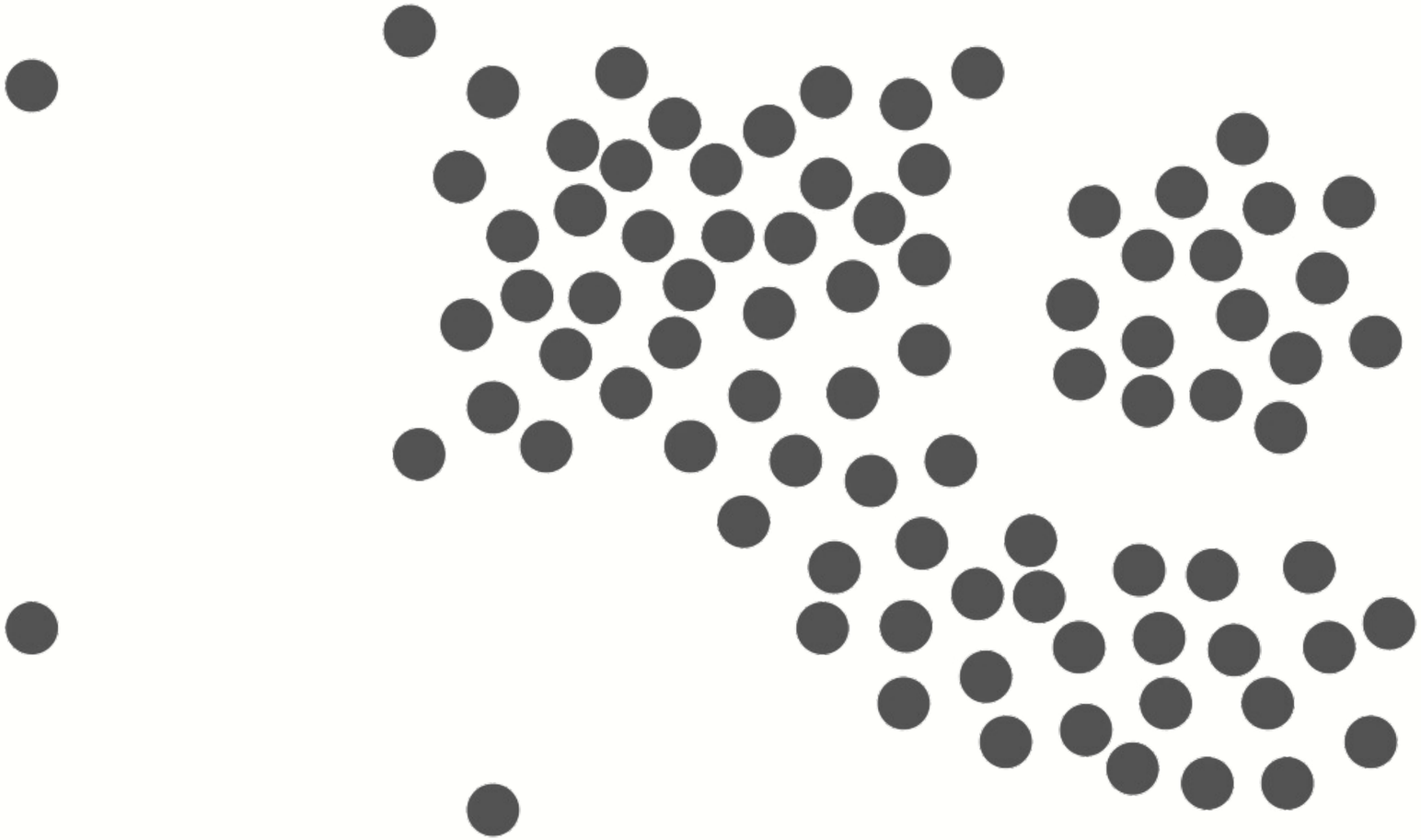
```
DBSCAN (SetOfPoints, Eps, MinPts)

// SetOfPoints is UNCLASSIFIED
  ClusterId := nextId(NOISE);
  FOR i FROM 1 TO SetOfPoints.size DO
    Point := SetOfPoints.get(i);
    IF Point.ClId = UNCLASSIFIED THEN
      IF ExpandCluster(SetOfPoints, Point,
            ClusterId, Eps, MinPts) THEN
        ClusterId := nextId(ClusterId)
      END IF
    END IF
  END FOR
END; // DBSCAN
```

`SetOfPoints` is either the whole database or a discovered cluster from a previous run. `Eps` and `MinPts` are the global density parameters determined either manually or according to the heuristics presented in section 4.2. The function `SetOfPoints.get(i)` returns the i-th element of `SetOfPoints`. The most important function used by DBSCAN is `ExpandCluster` which is presented below:

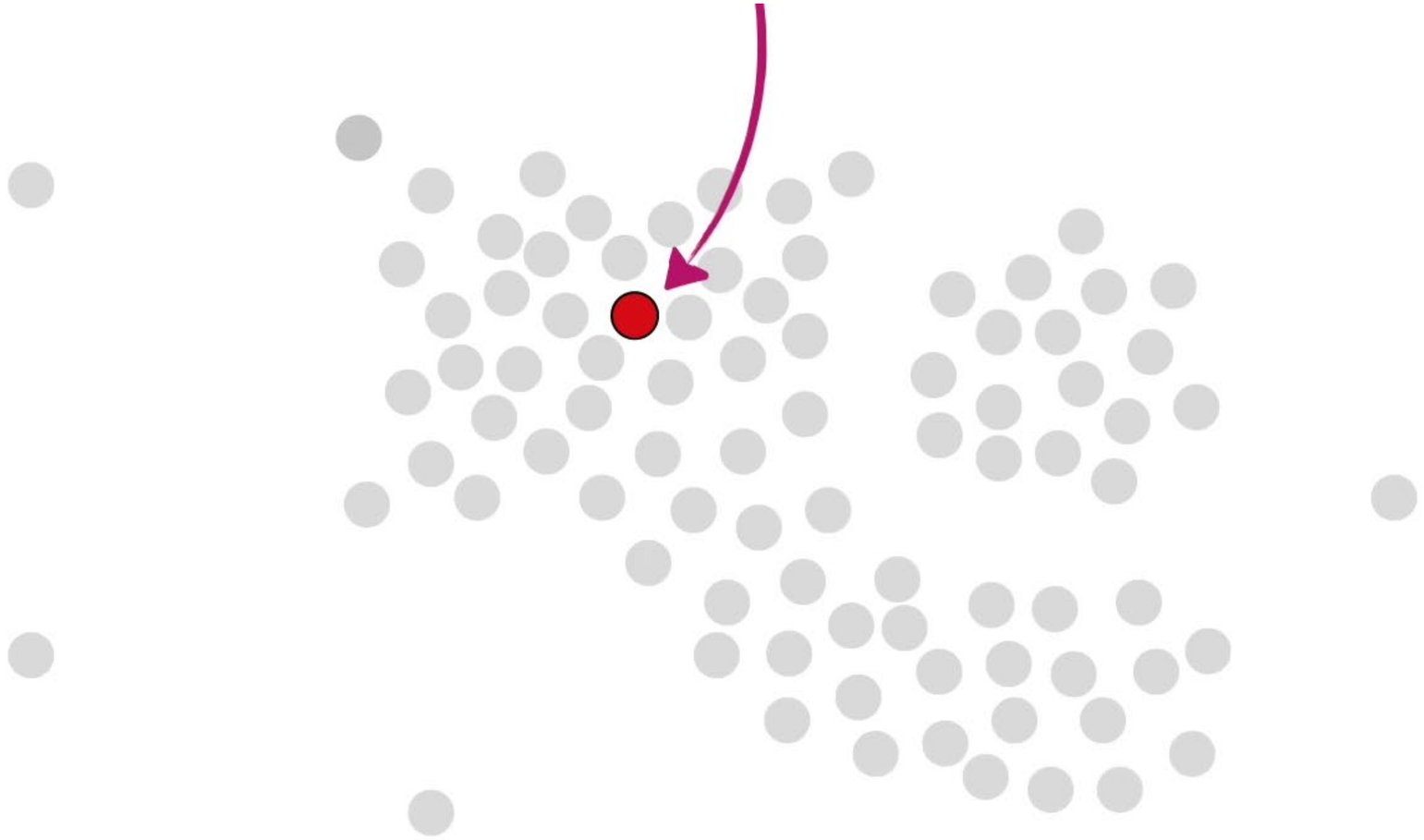
```
ExpandCluster(SetOfPoints, Point, ClId, Eps,
              MinPts) : Boolean;
  seeds:=SetOfPoints.regionQuery(Point,Eps);
  IF seeds.size<MinPts THEN // no core point
    SetOfPoint.changeClId(Point,NOISE);
    RETURN False;
  ELSE   // all points in seeds are density-
         // reachable from Point
    SetOfPoints.changeClIds(seeds,ClId);
    seeds.delete(Point);
    WHILE seeds <> Empty DO
      currentP := seeds.first();
      result := SetOfPoints.regionQuery(currentP,
                                        Eps);
      IF result.size >= MinPts THEN
        FOR i FROM 1 TO result.size DO
          resultP := result.get(i);
          IF resultP.ClId
              IN (UNCLASSIFIED, NOISE) THEN
            IF resultP.ClId = UNCLASSIFIED THEN
              seeds.append(resultP);
            END IF;
            SetOfPoints.changeClId(resultP,ClId);
          END IF; // UNCLASSIFIED or NOISE
        END FOR;
      END IF; // result.size >= MinPts
      seeds.delete(currentP);
    END WHILE; // seeds <> Empty
    RETURN True;
  END IF
END; // ExpandCluster
```

This is unclustered data

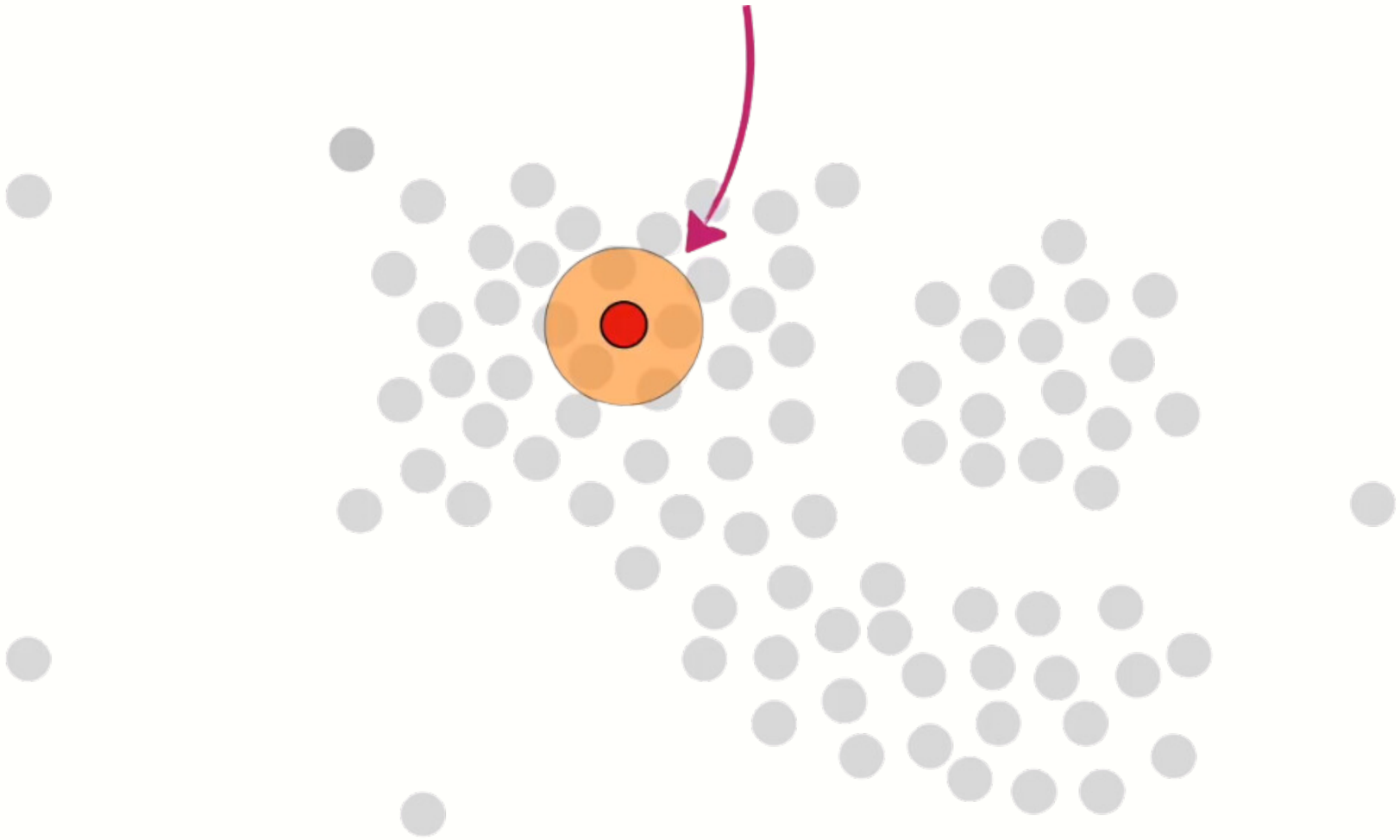


The first thing we can do is count the number of points close to each point

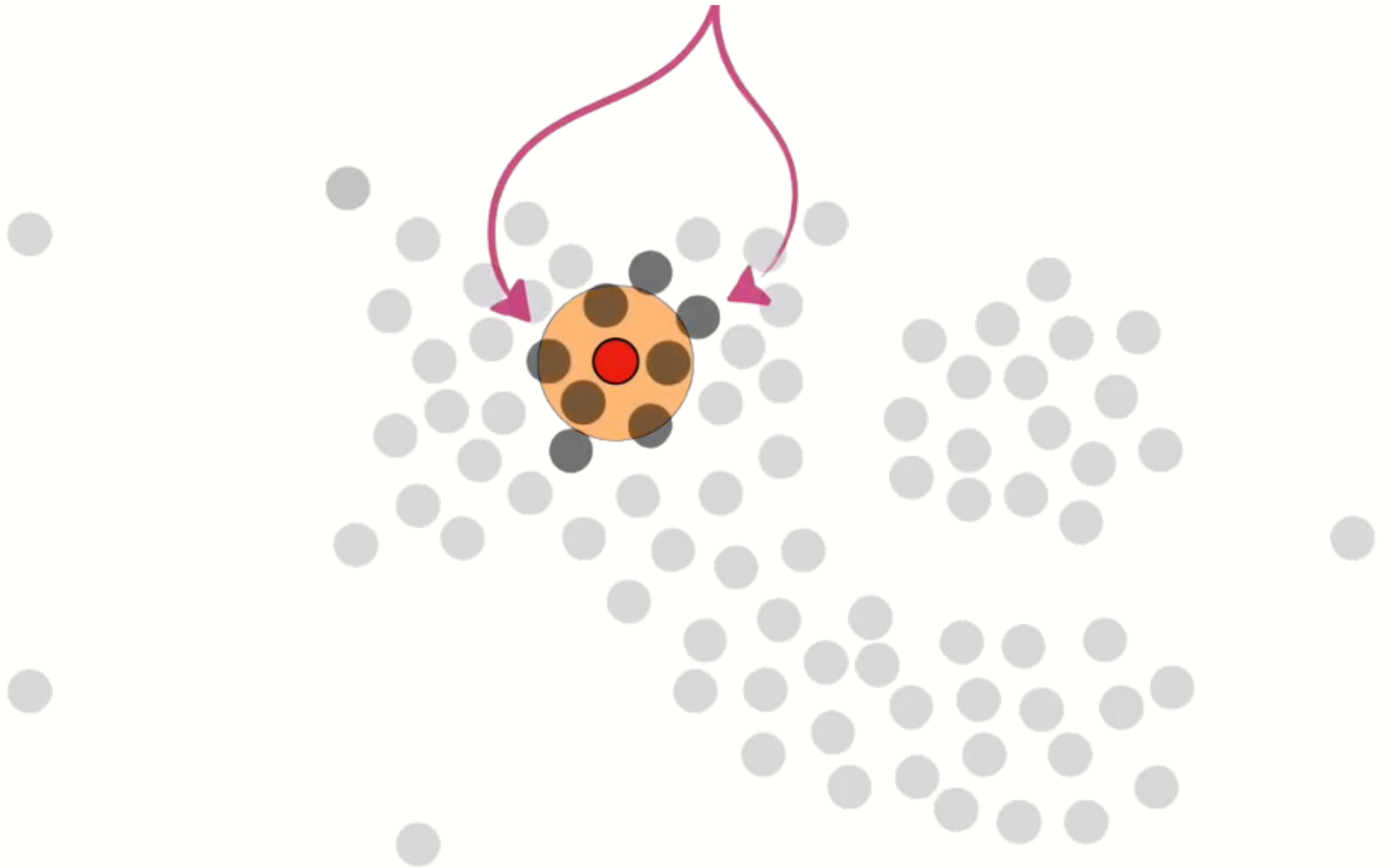
For example, if we start with this **red point**



and we draw an orange circle around it

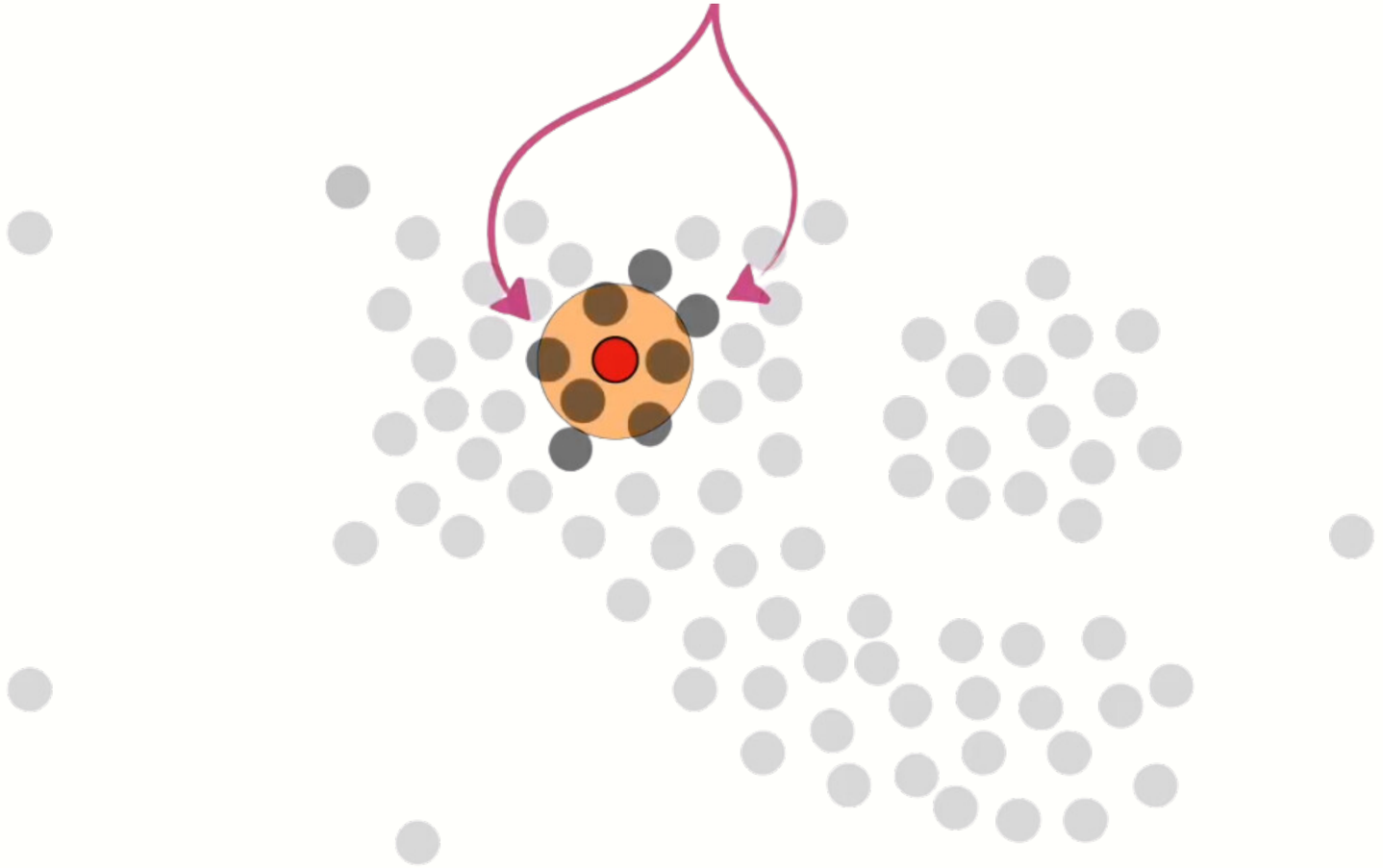


then we see that the orange circle overlaps, at least partially, 8 other points.

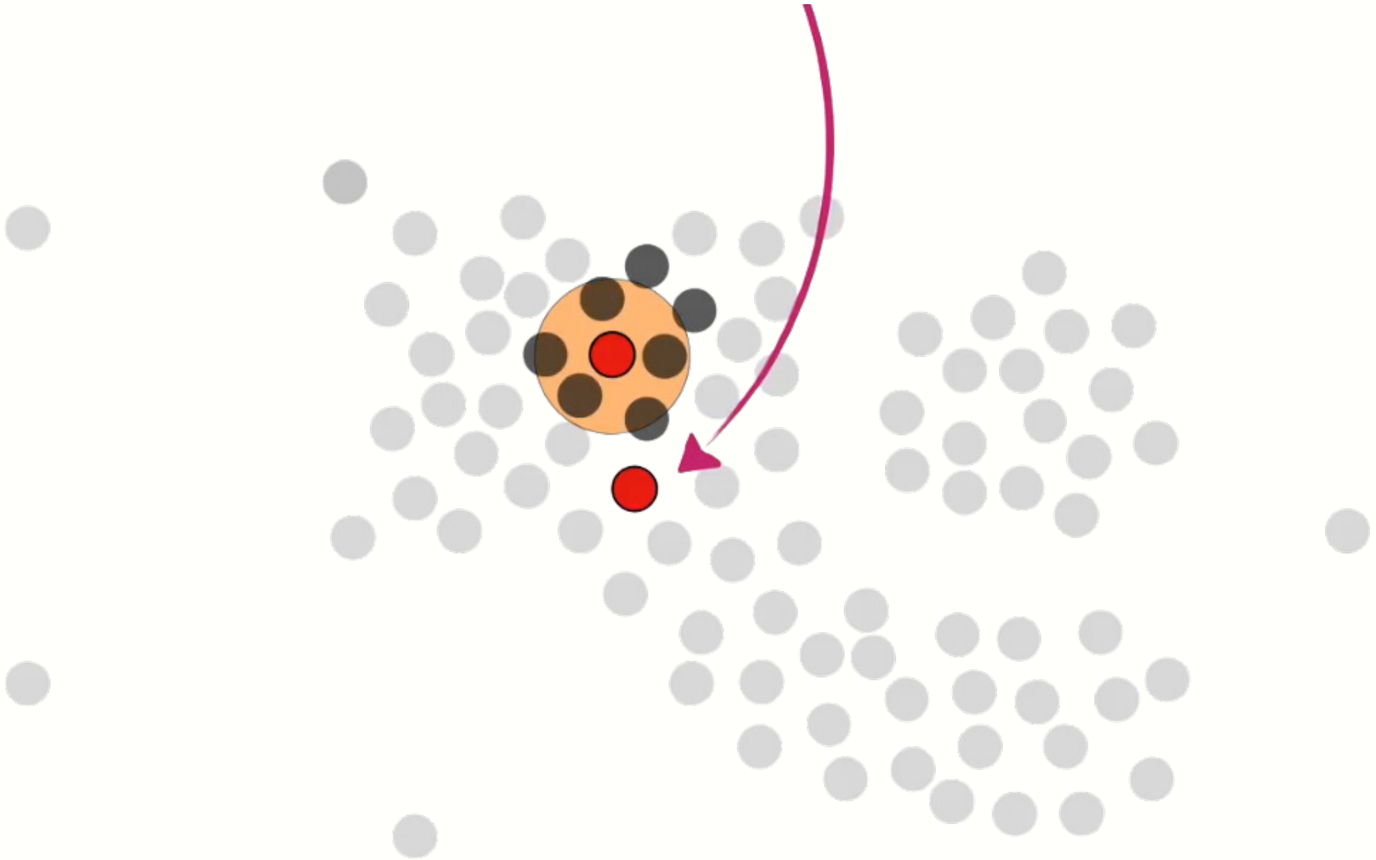


So the red point is close to 8 other points.

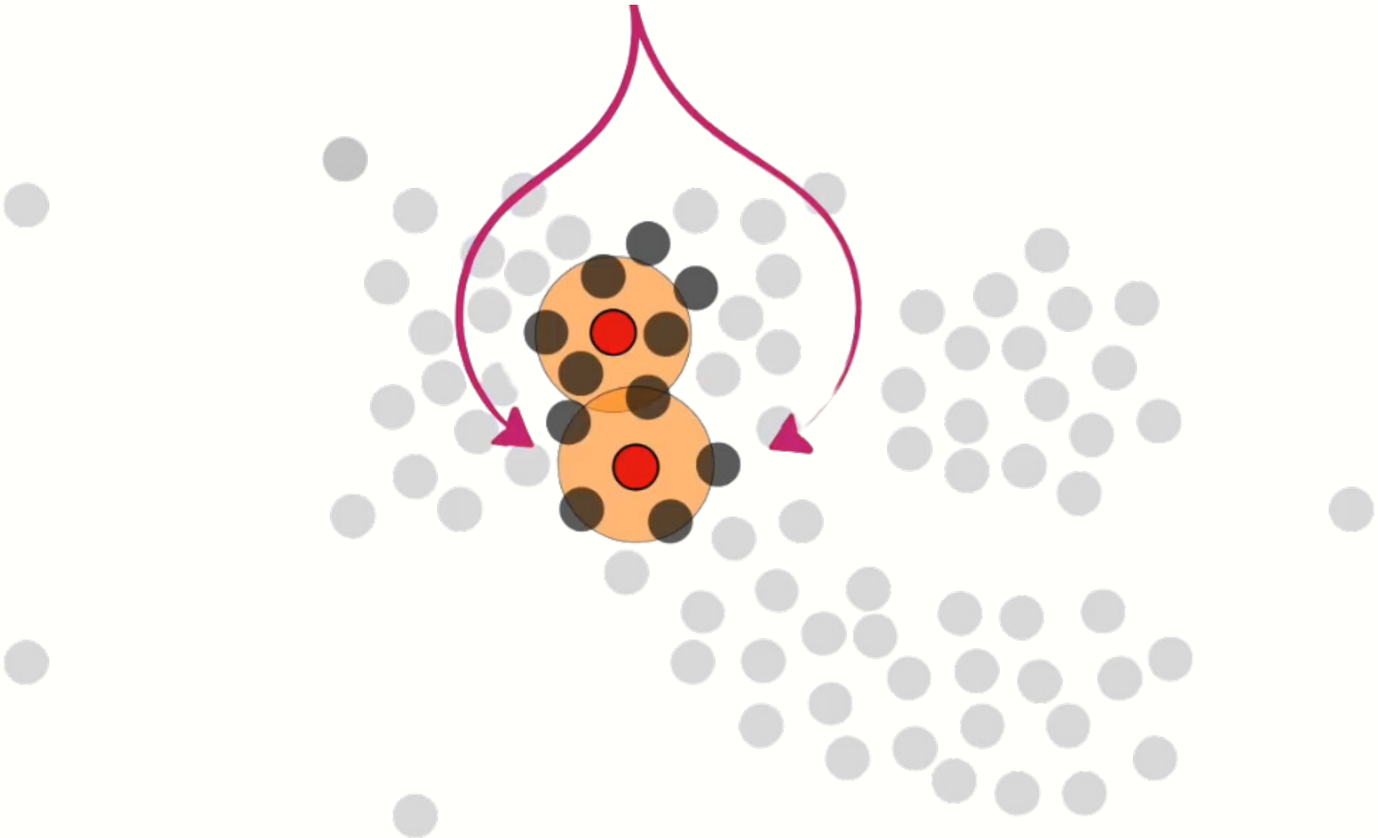
NOTE: The radius of the **orange circle** is user defined, so when using DBSCAN, you may need to fiddle around with this parameter.



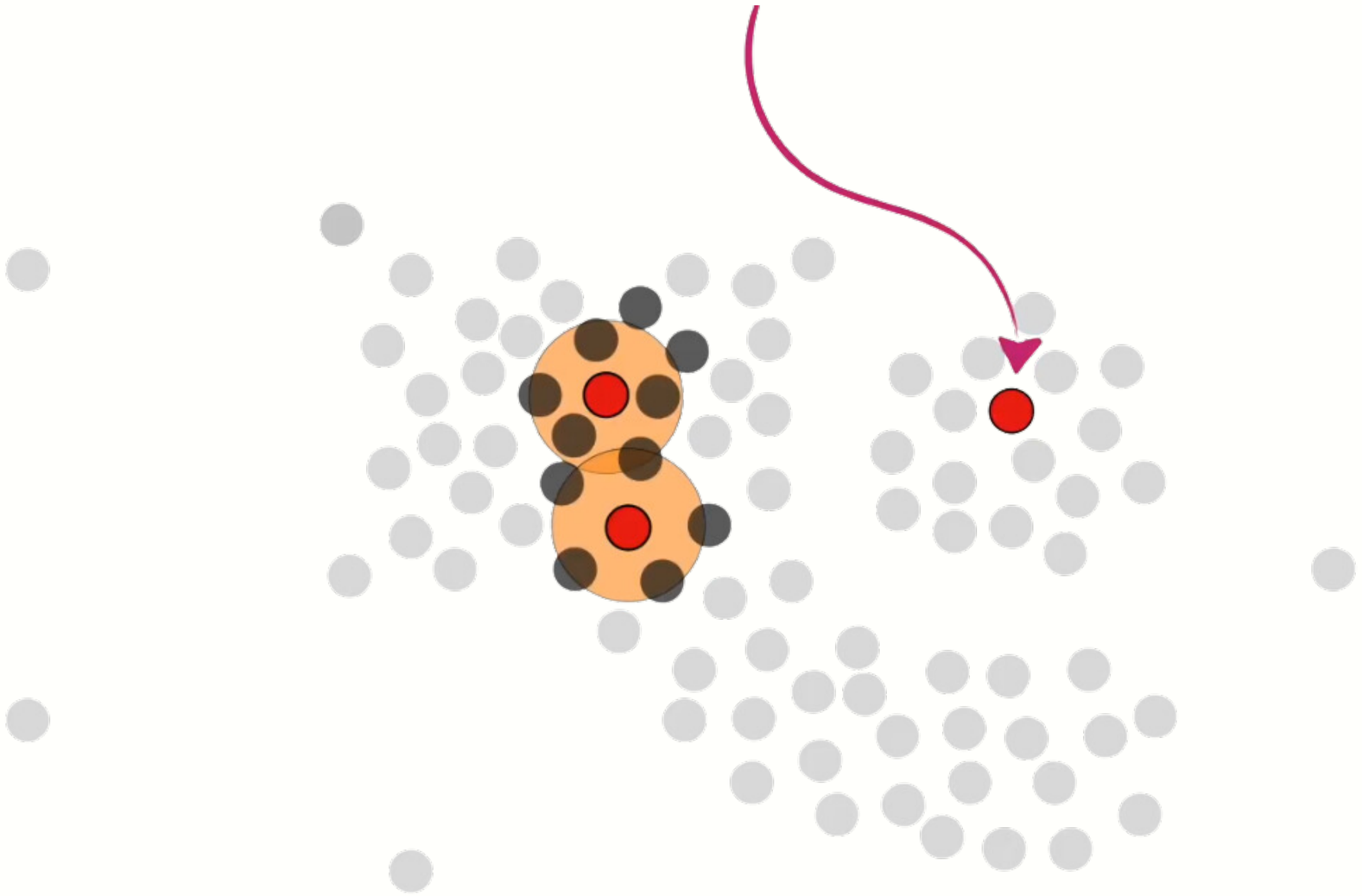
Now, this **red point**



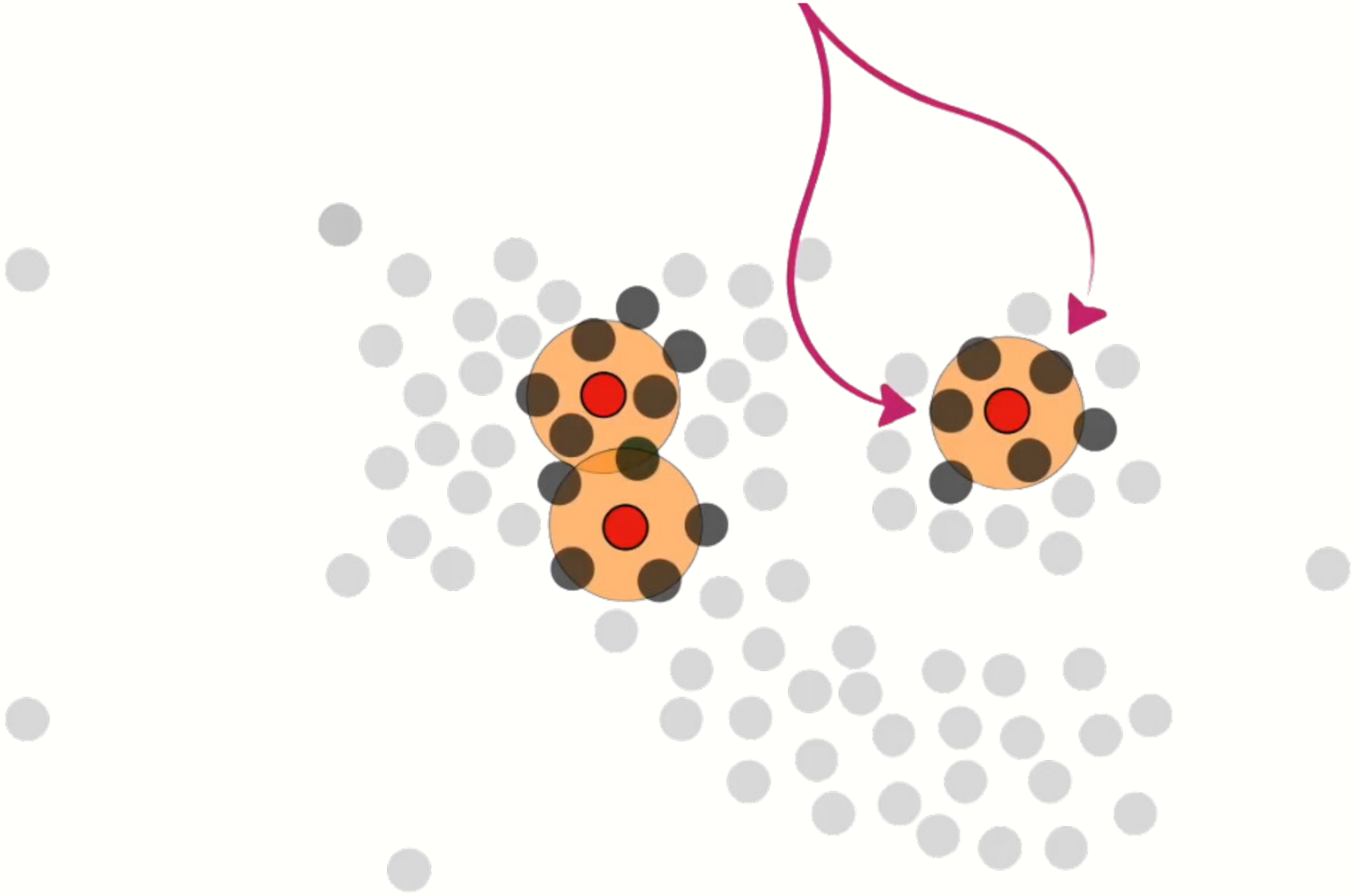
...is close to 5 other points because the **orange circle** overlaps, at least partially, 5 other points.



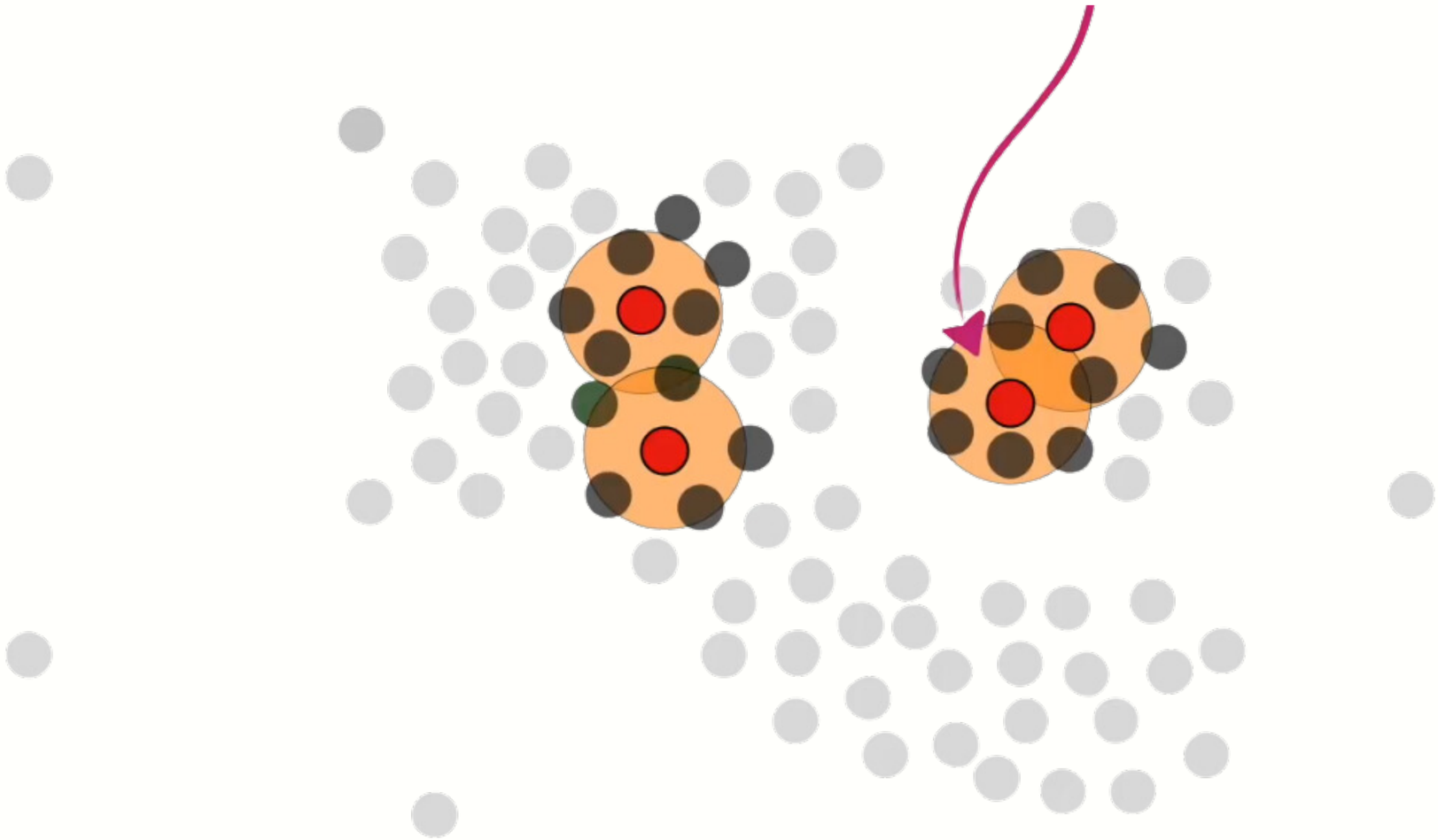
This **red point**



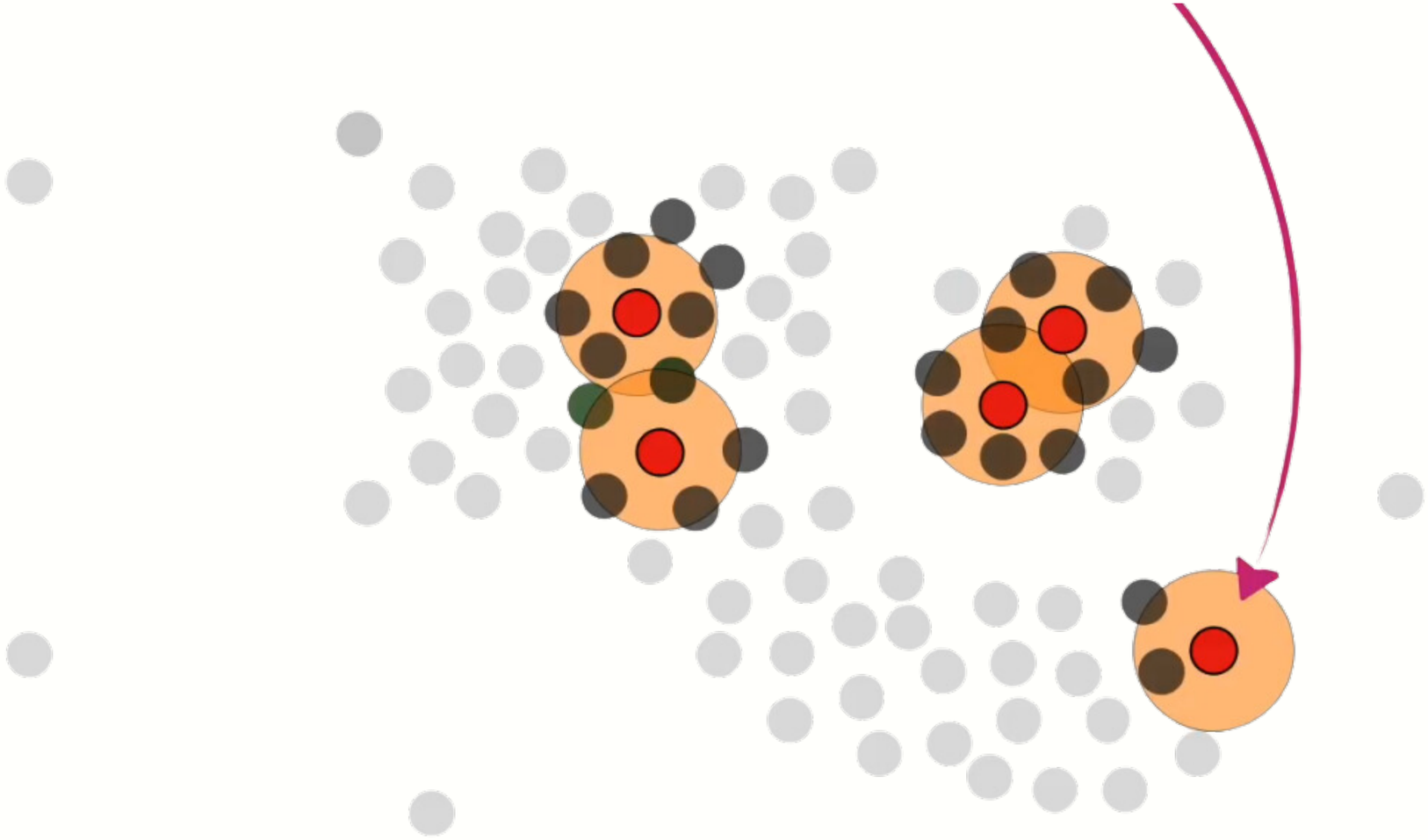
... is close to 6 other points ...



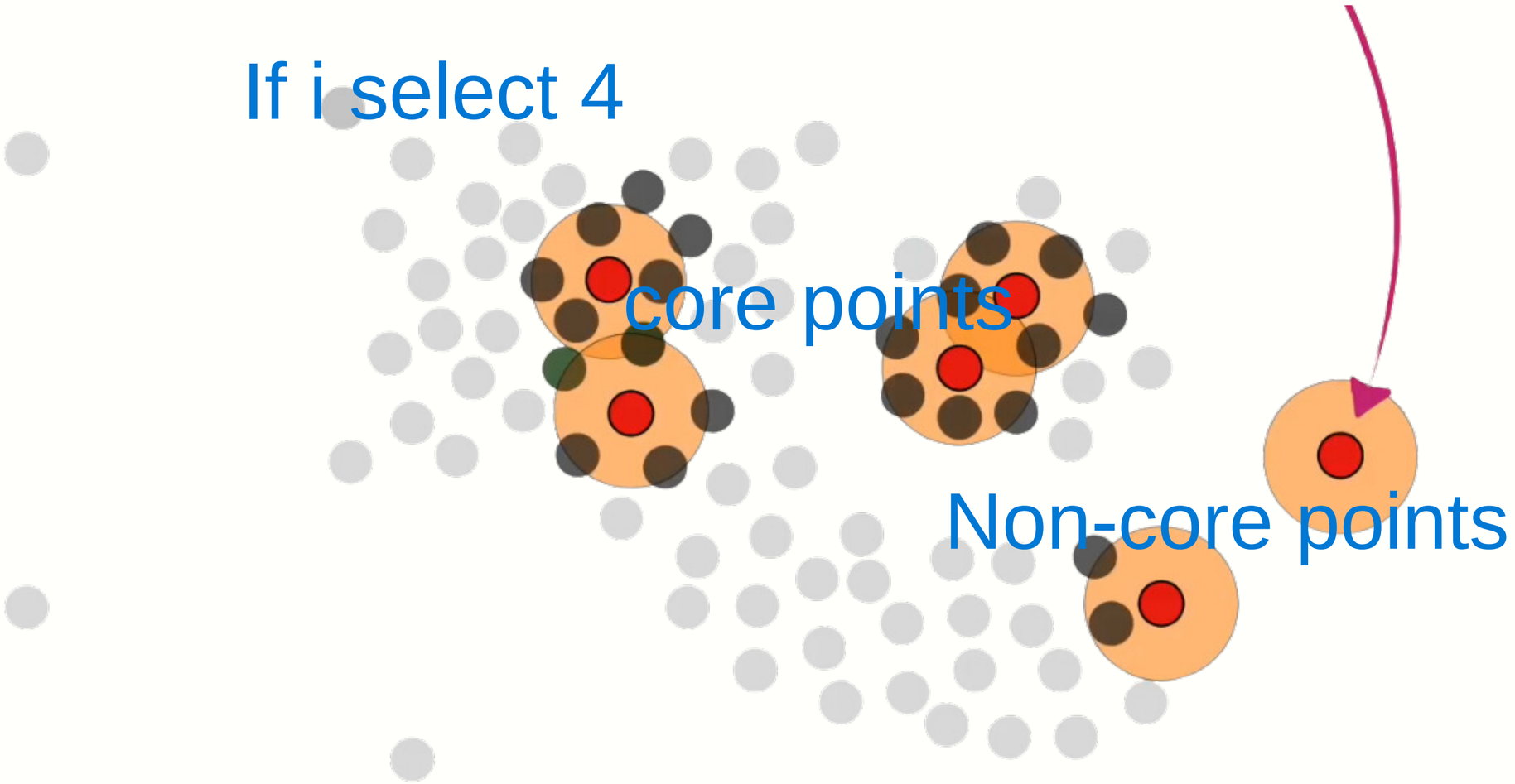
...and this **red point** is close to 7 other points.



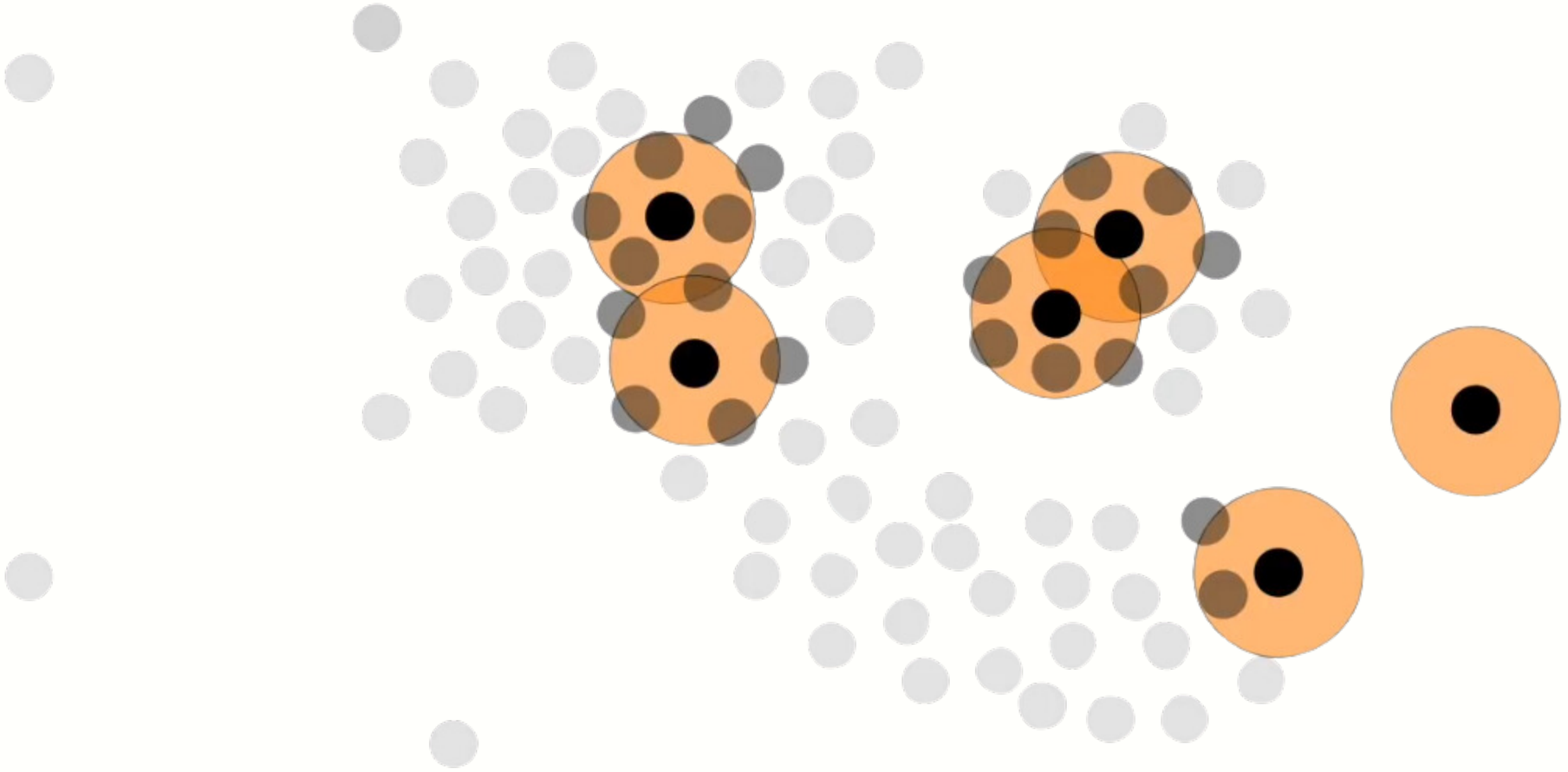
This **red point** is only close to 2 other points...



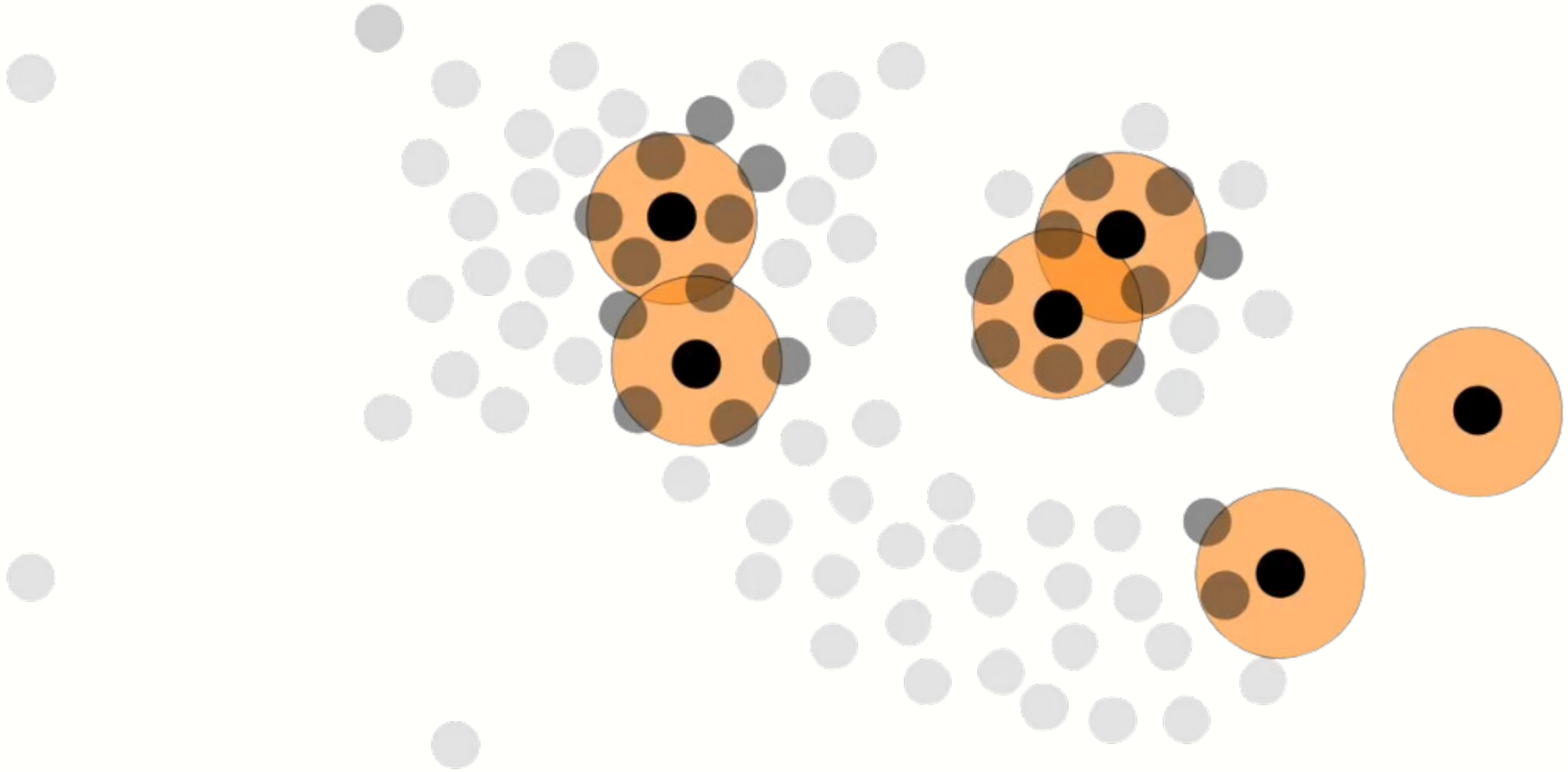
...and this **red point** is not close to any other point because the **orange circle** does not overlap anything else.



Likewise, for all of the remaining points, we count the number of close points.

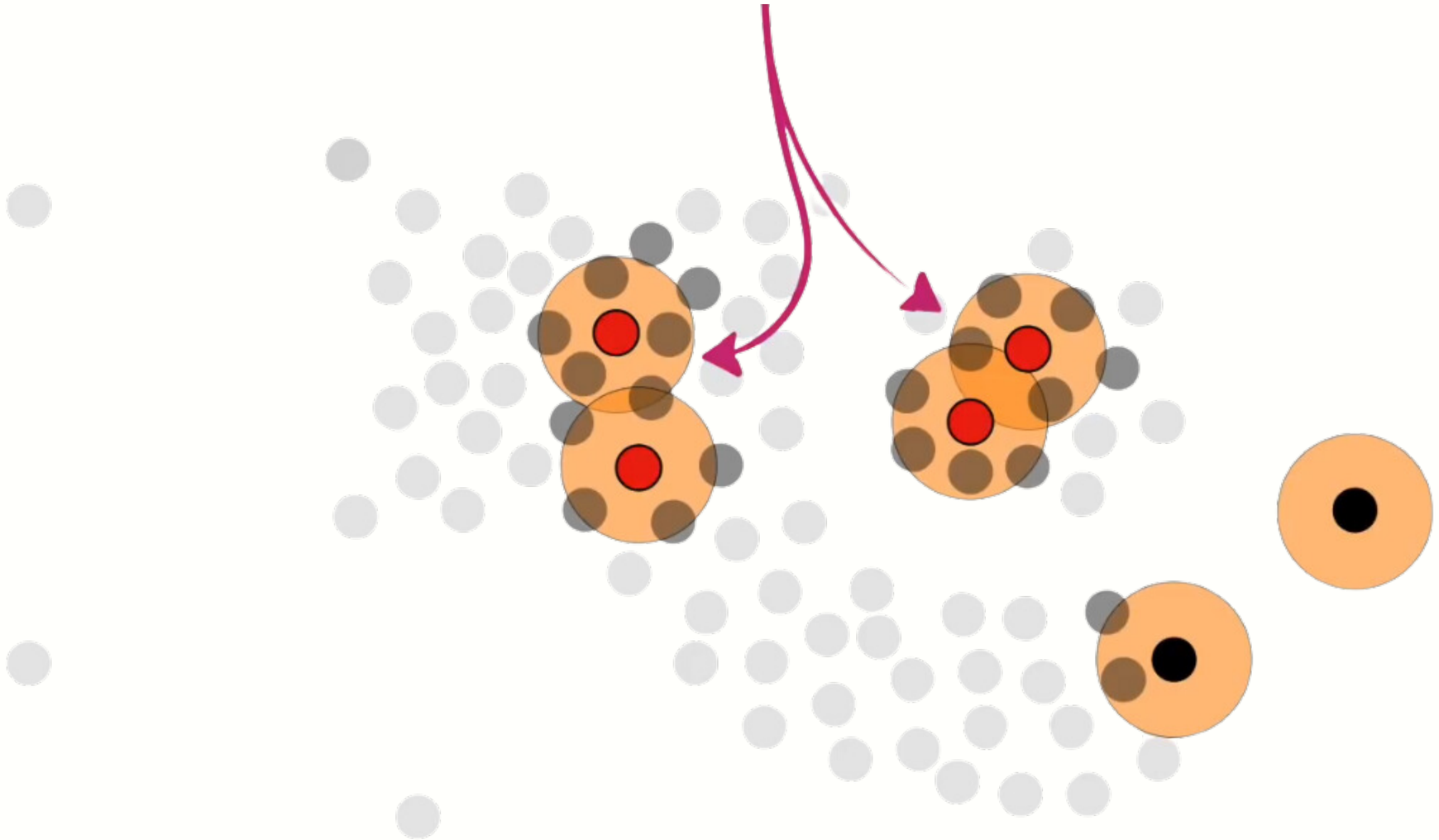


Now, in this example, we will define a Core Point to be one that is close to at least 4 other points.

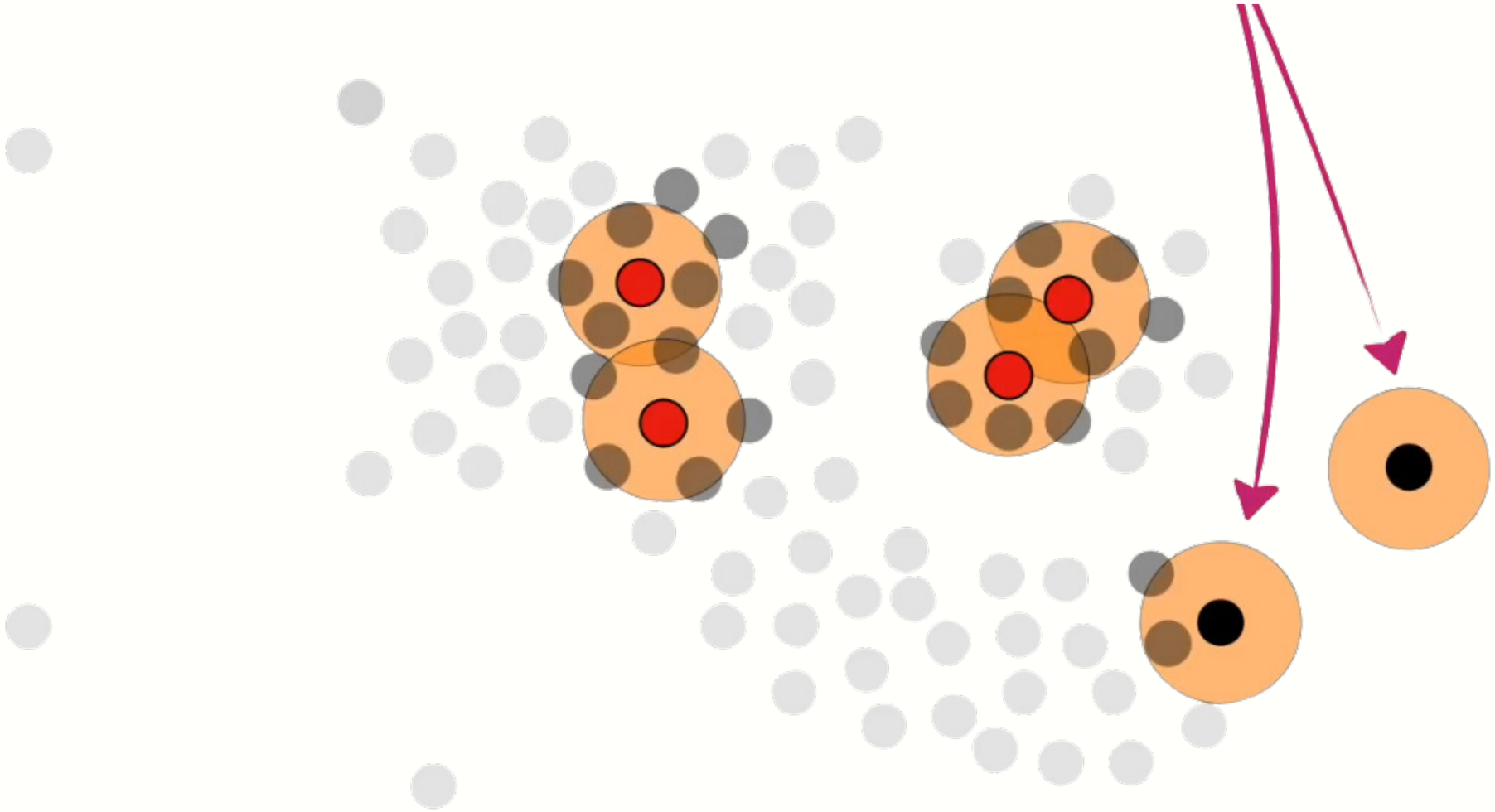


NOTE: The number of close points for a Core Point is user defined, so, when using DBSCAN, you might need to fiddle with this parameter as well.

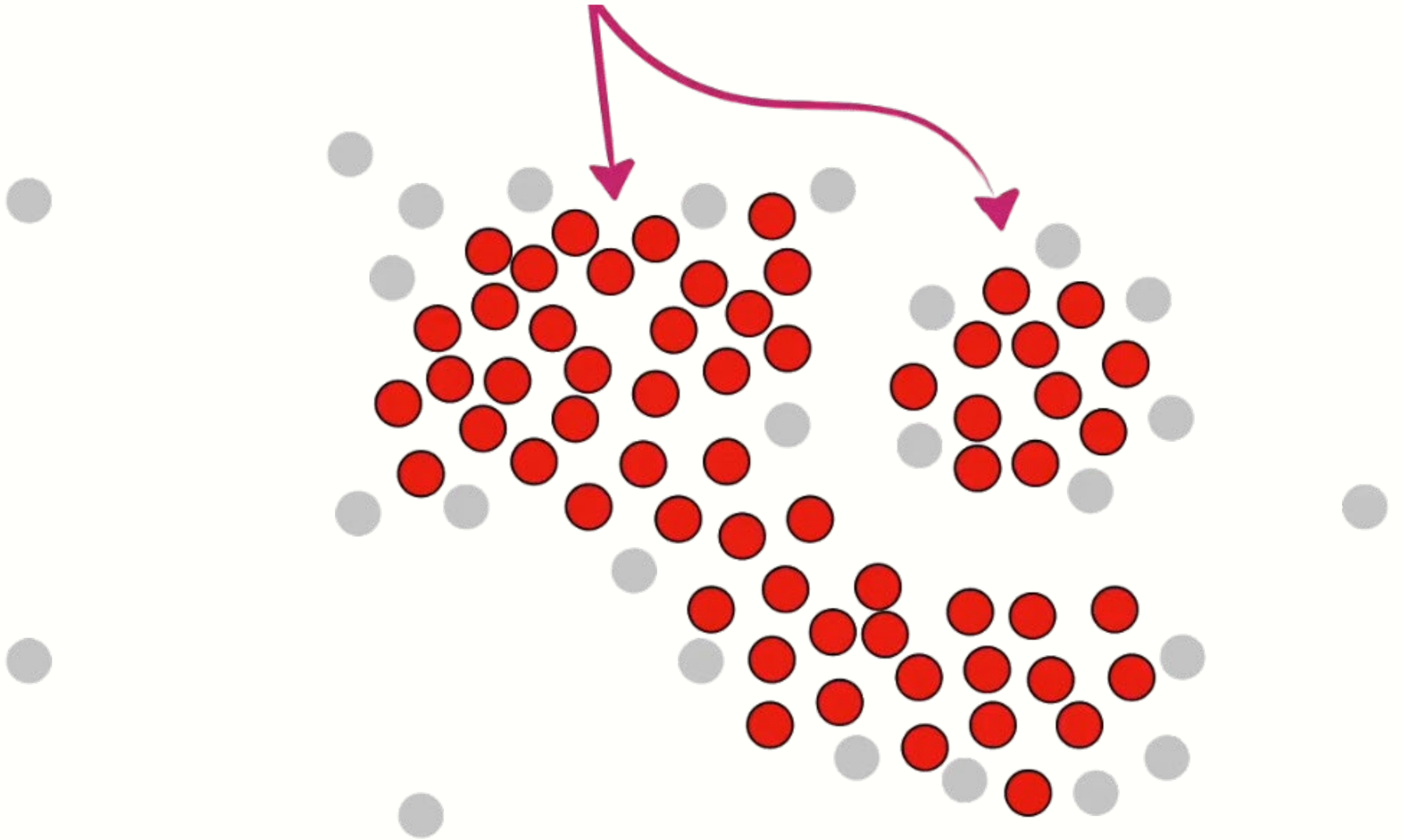
Anyway, these 4 points are some of the Core Points, because their **orange circles** overlap at least 4 other points...



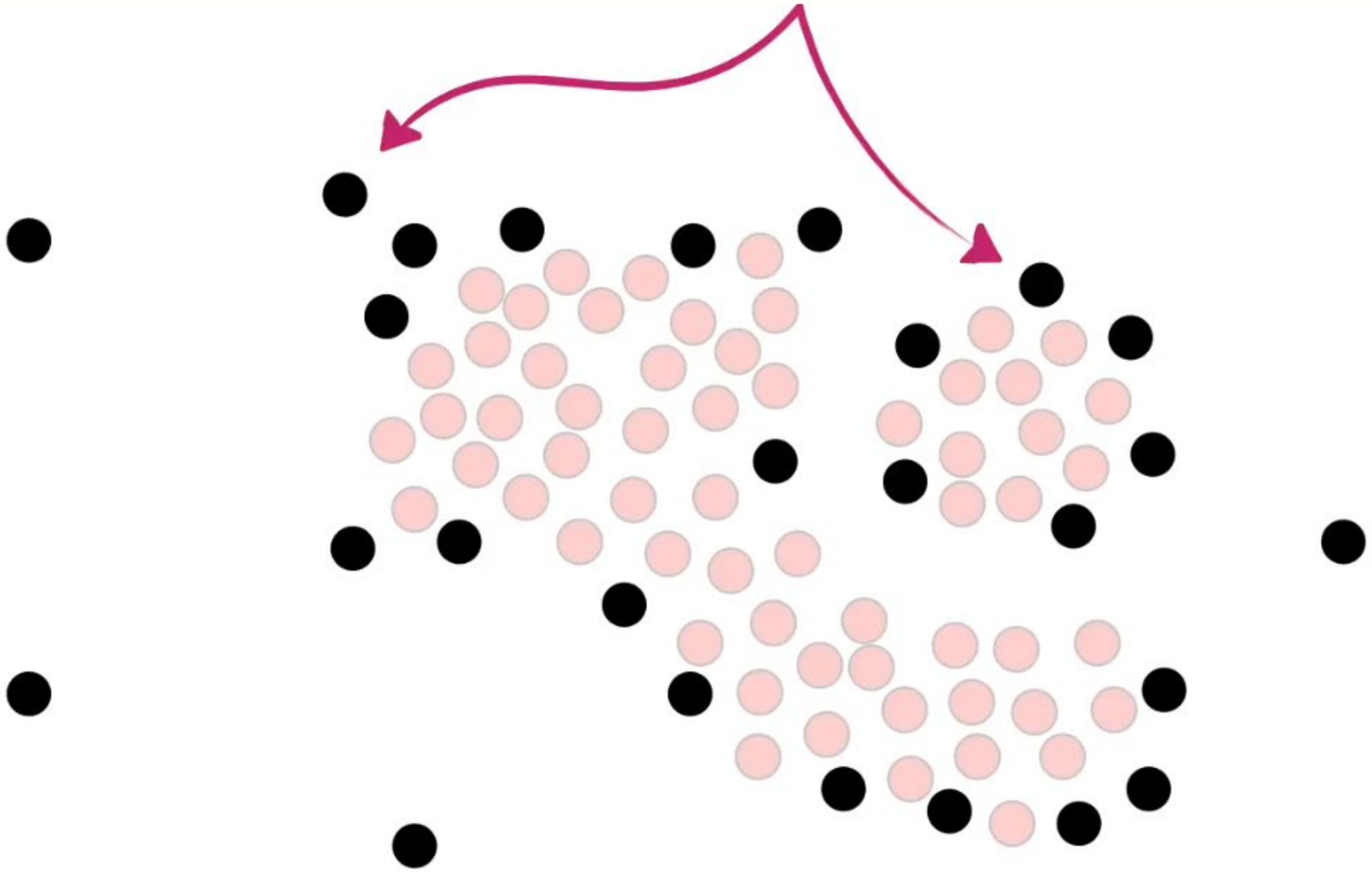
...but neither of these points are Core Points because their orange circles do not overlap 4 or more other points.



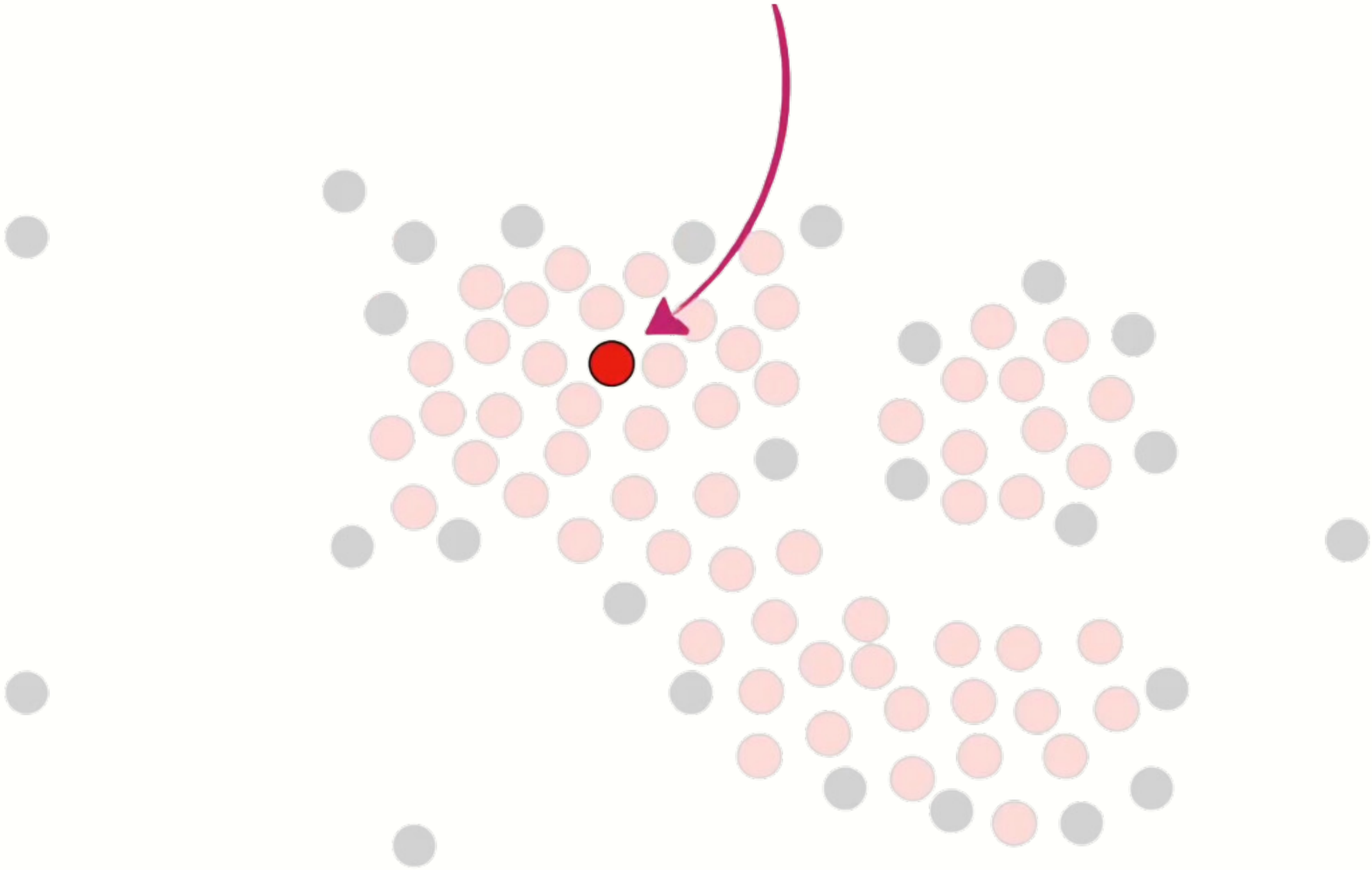
Ultimately, we can call all of these **red points** Core Points because they are all close to 4 or more other points...



...and the remaining points are Non-Core.

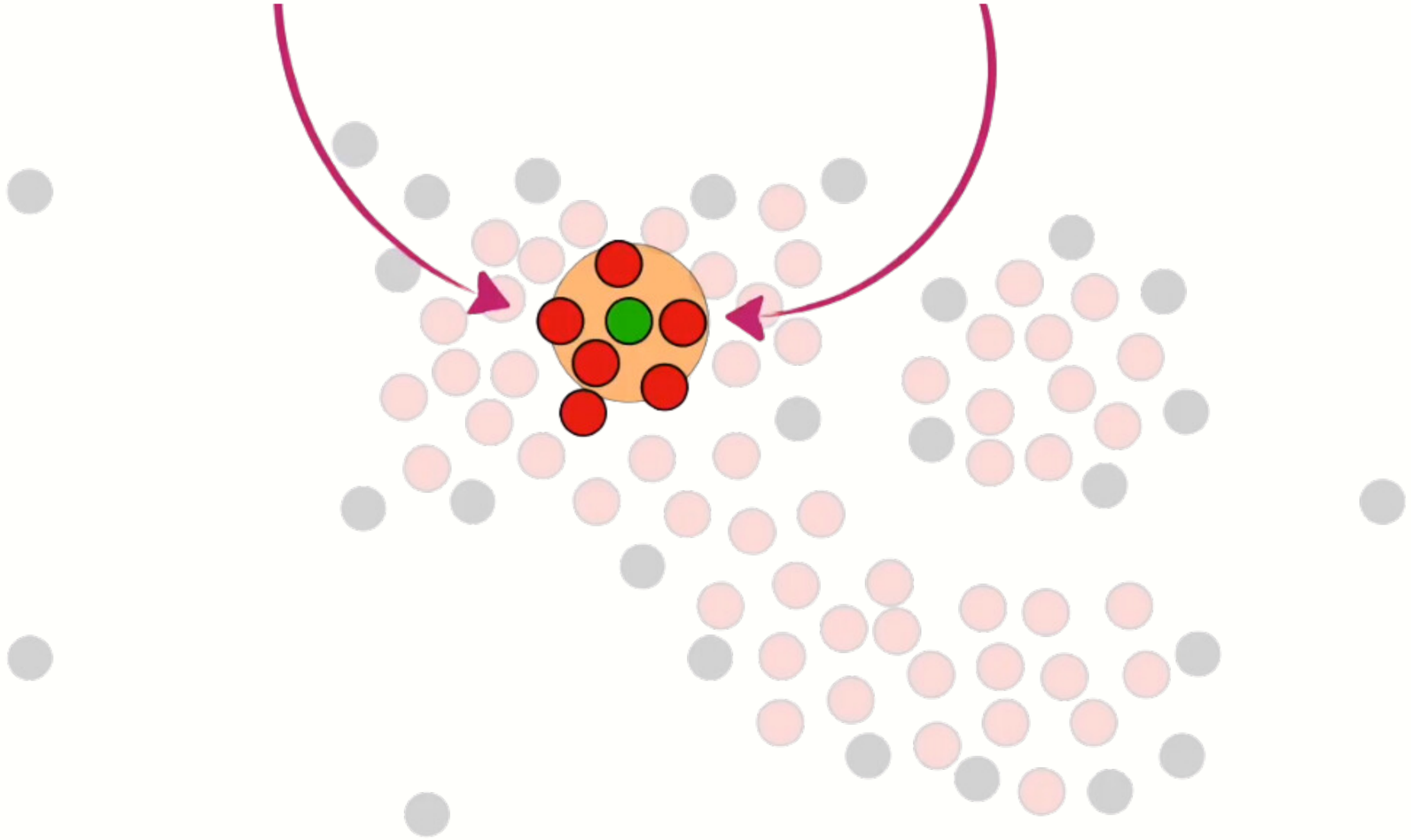


Now we randomly pick a Core Point...

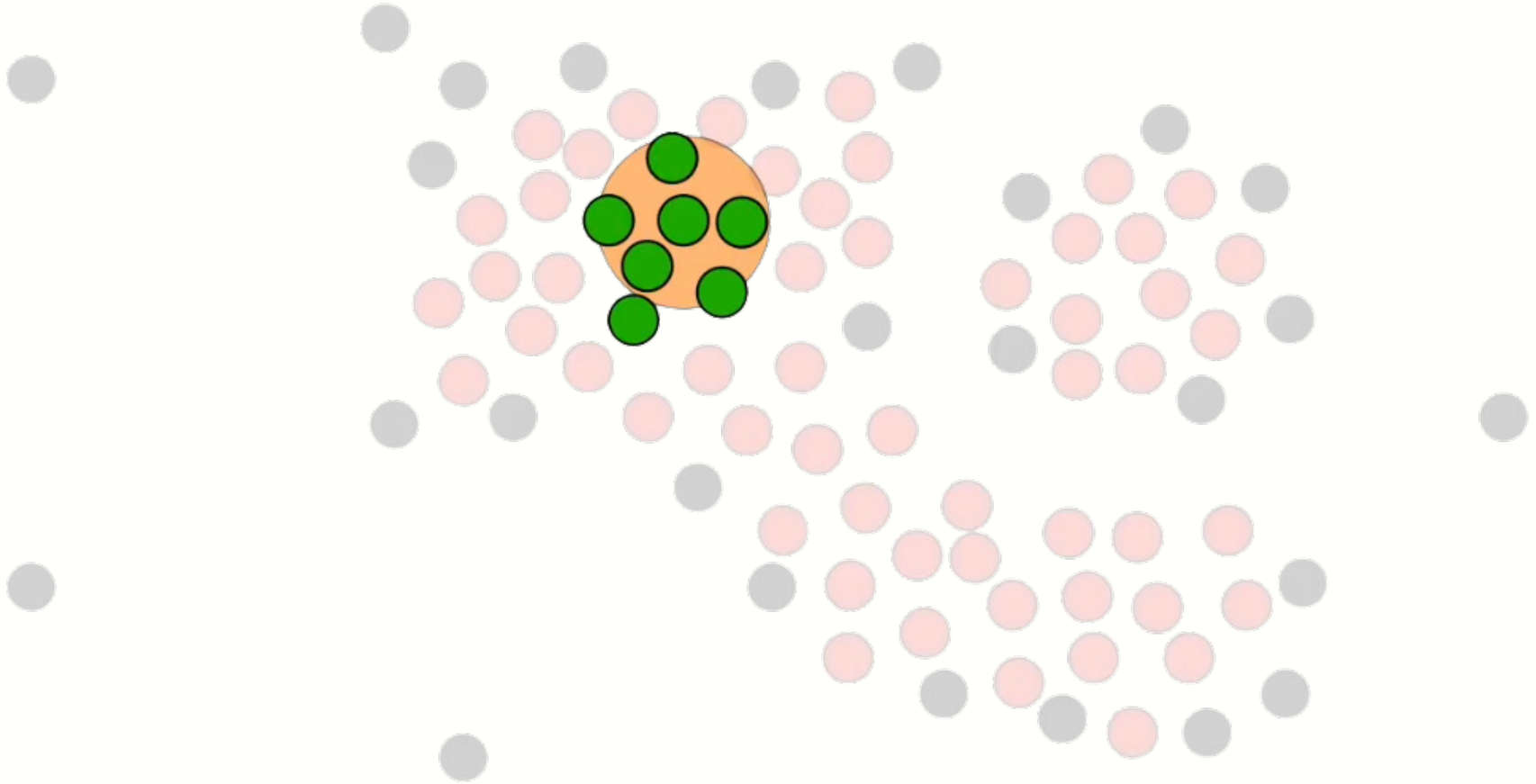


Next, the Core Points that are close to the

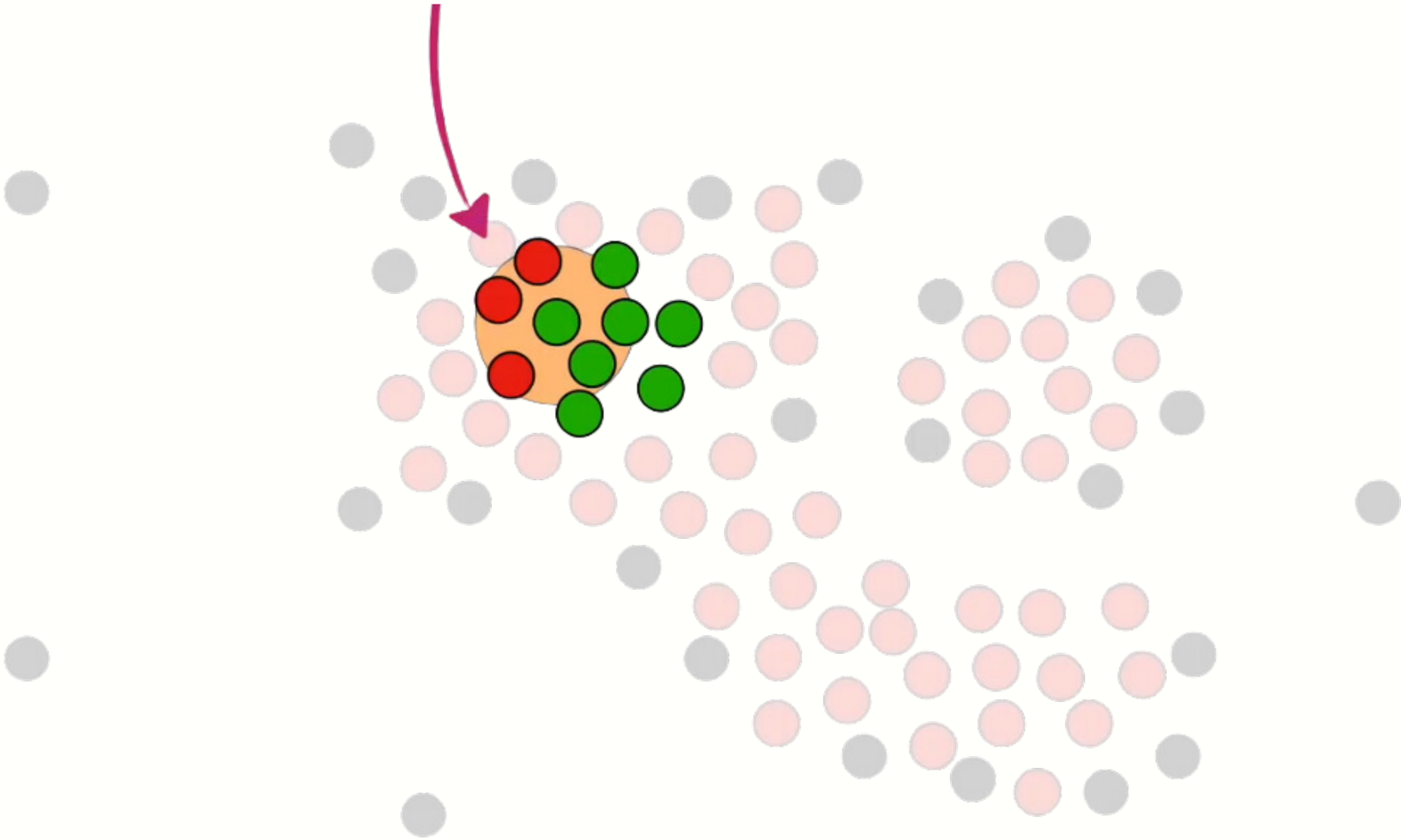
first cluster , meaning they overlap the orange circle ...



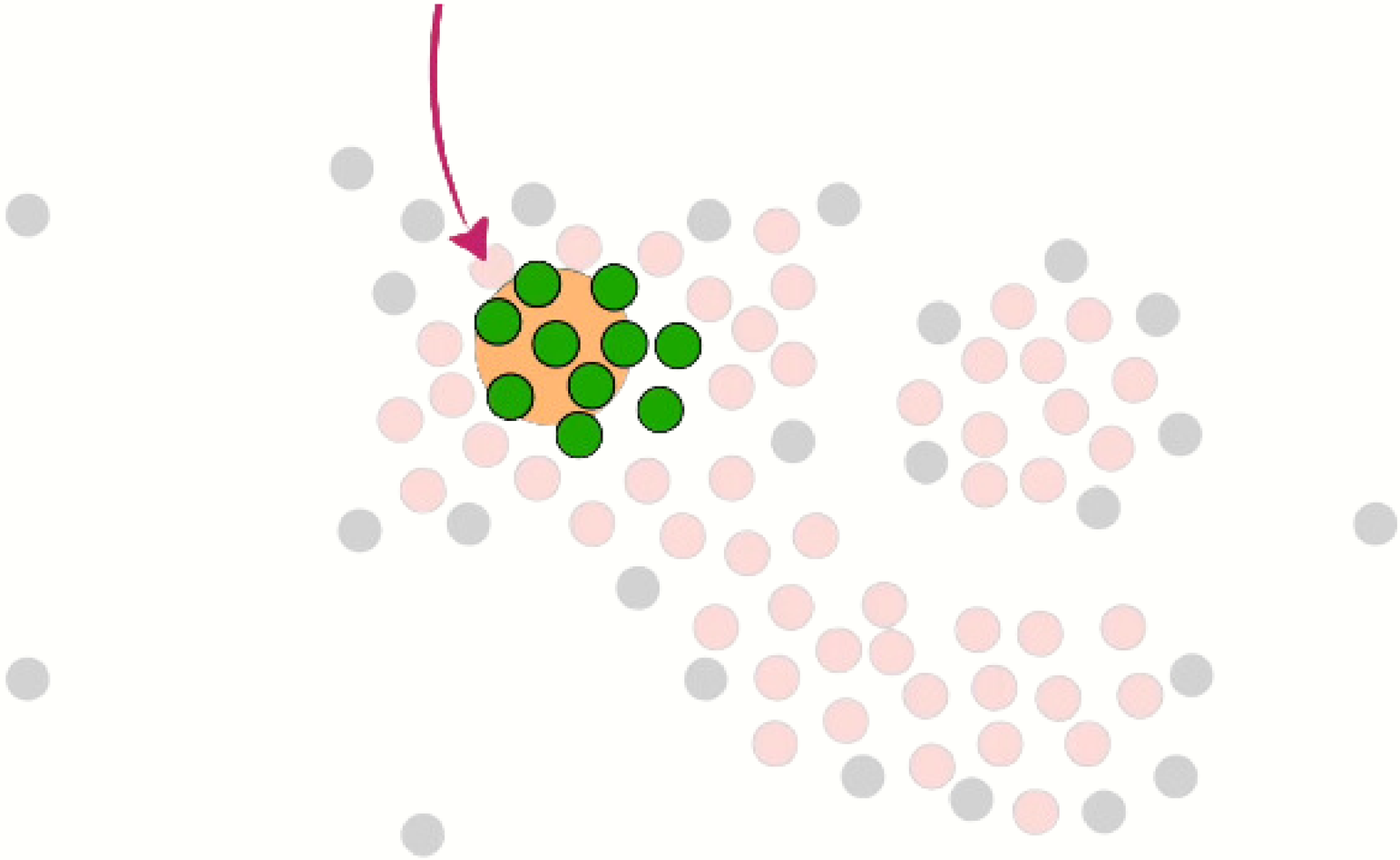
...are all added to the **first cluster** .



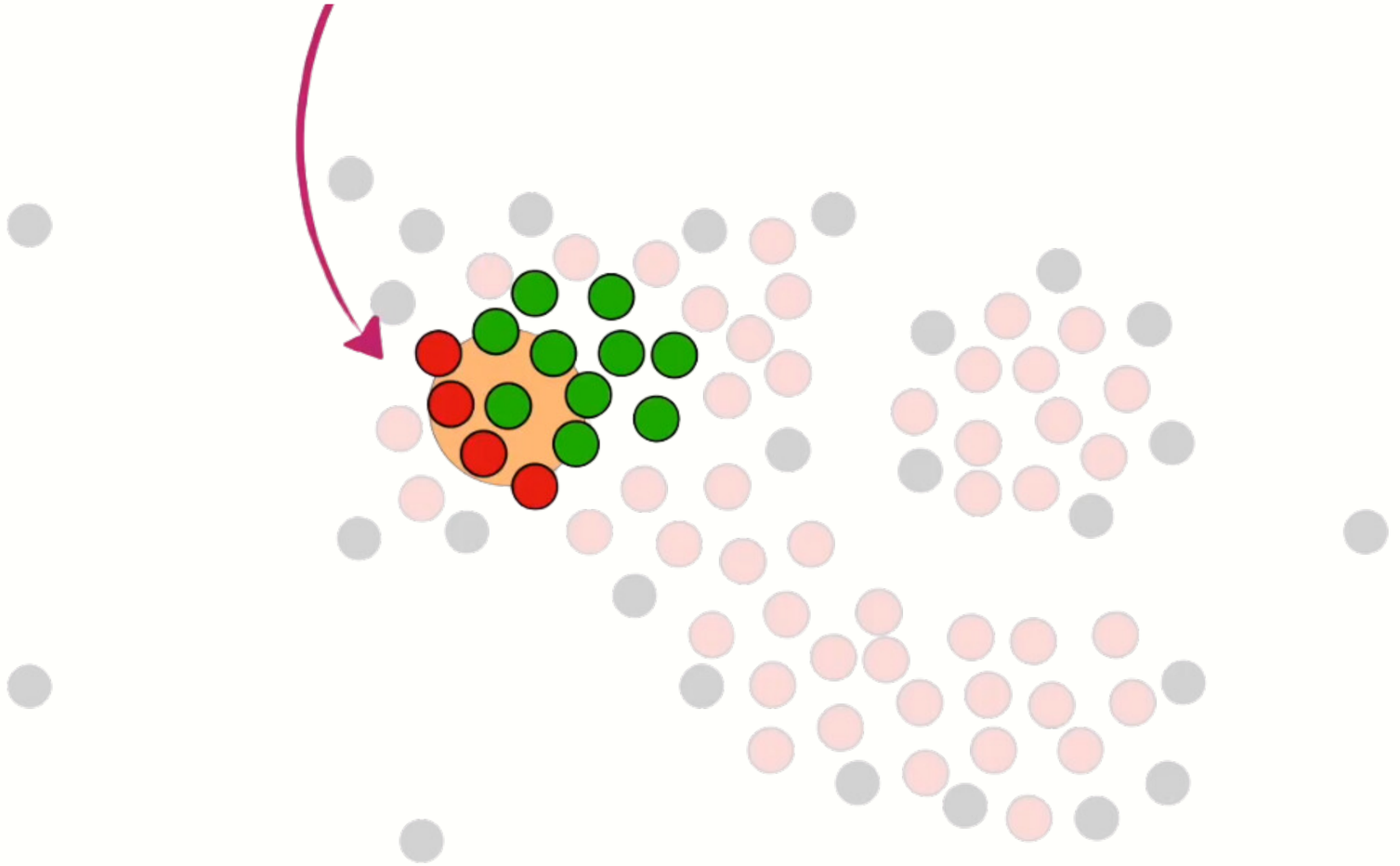
Then the Core Points that are close to the growing **first cluster** join it...



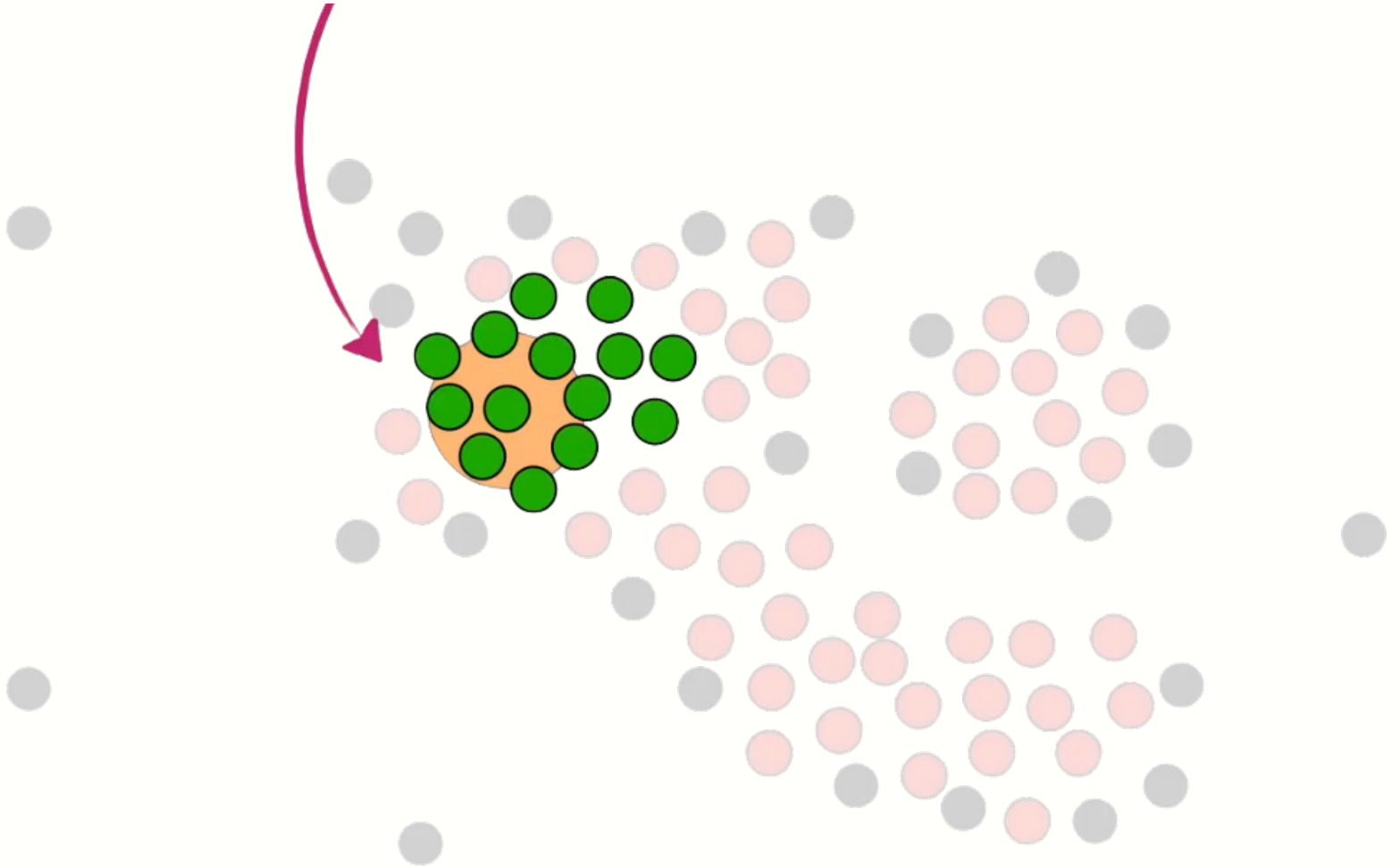
Then the Core Points that are close to the growing **first cluster** join it...



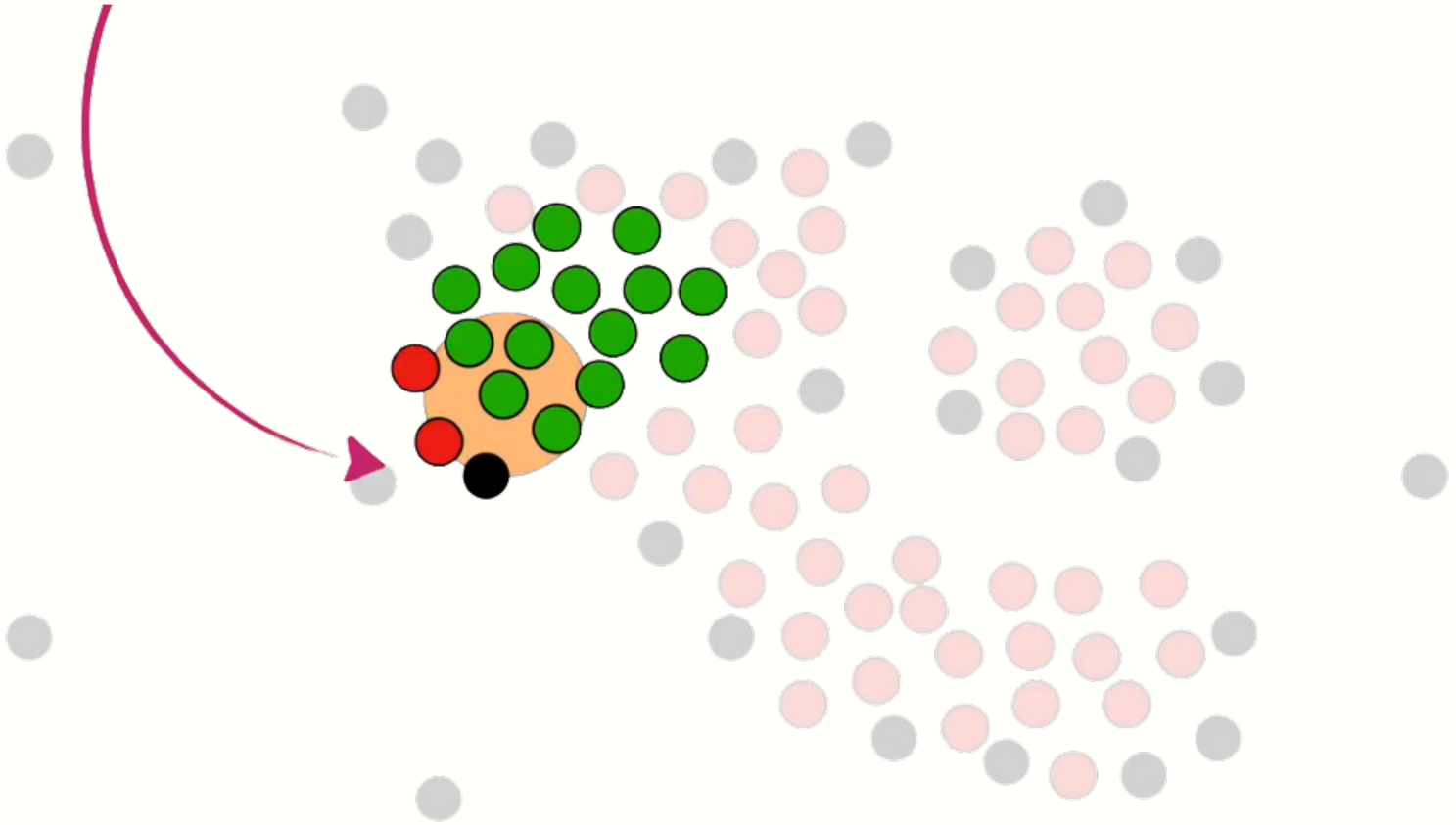
...and extend it to other **Core Points** that are close by.



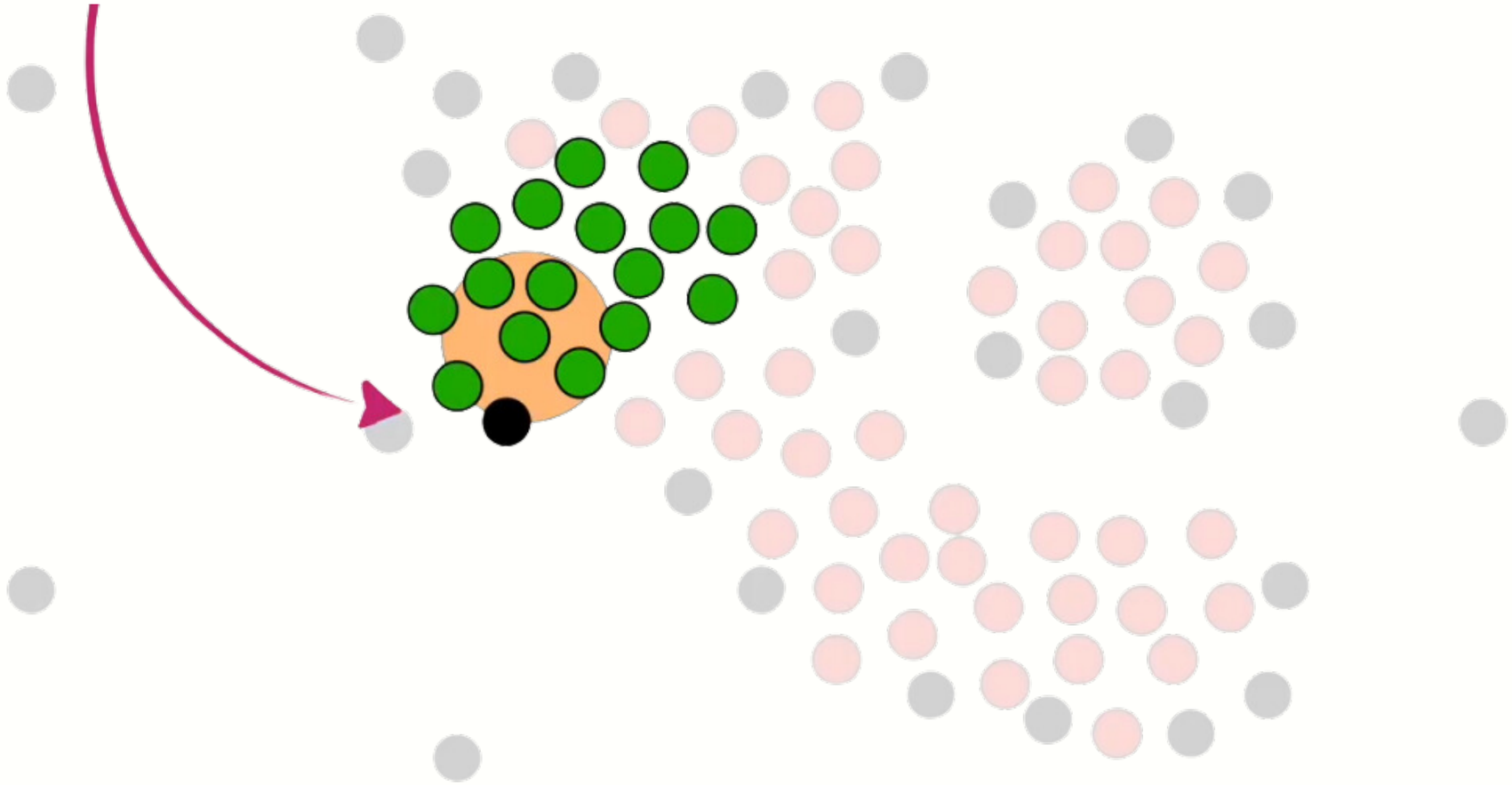
...and extend it to other **Core Points** that are close by.



Here we see 2 Core Points and 1 Non-Core Point that are all close to the growing **first cluster** ...



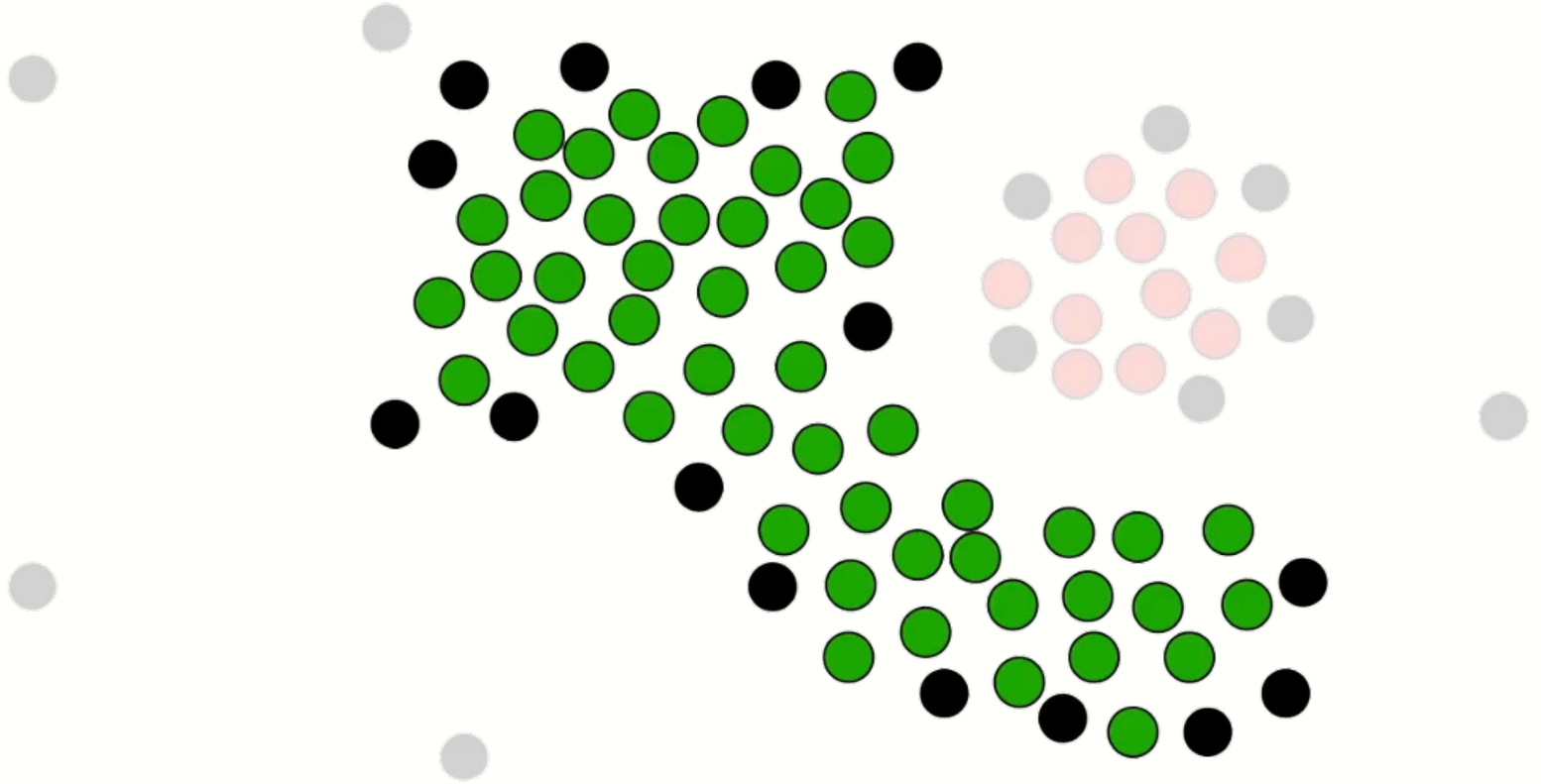
...and at this point, we only add the Core Points to the first cluster.



That said, eventually we will add this Non-Core Point, but right now we are only adding Core Points.

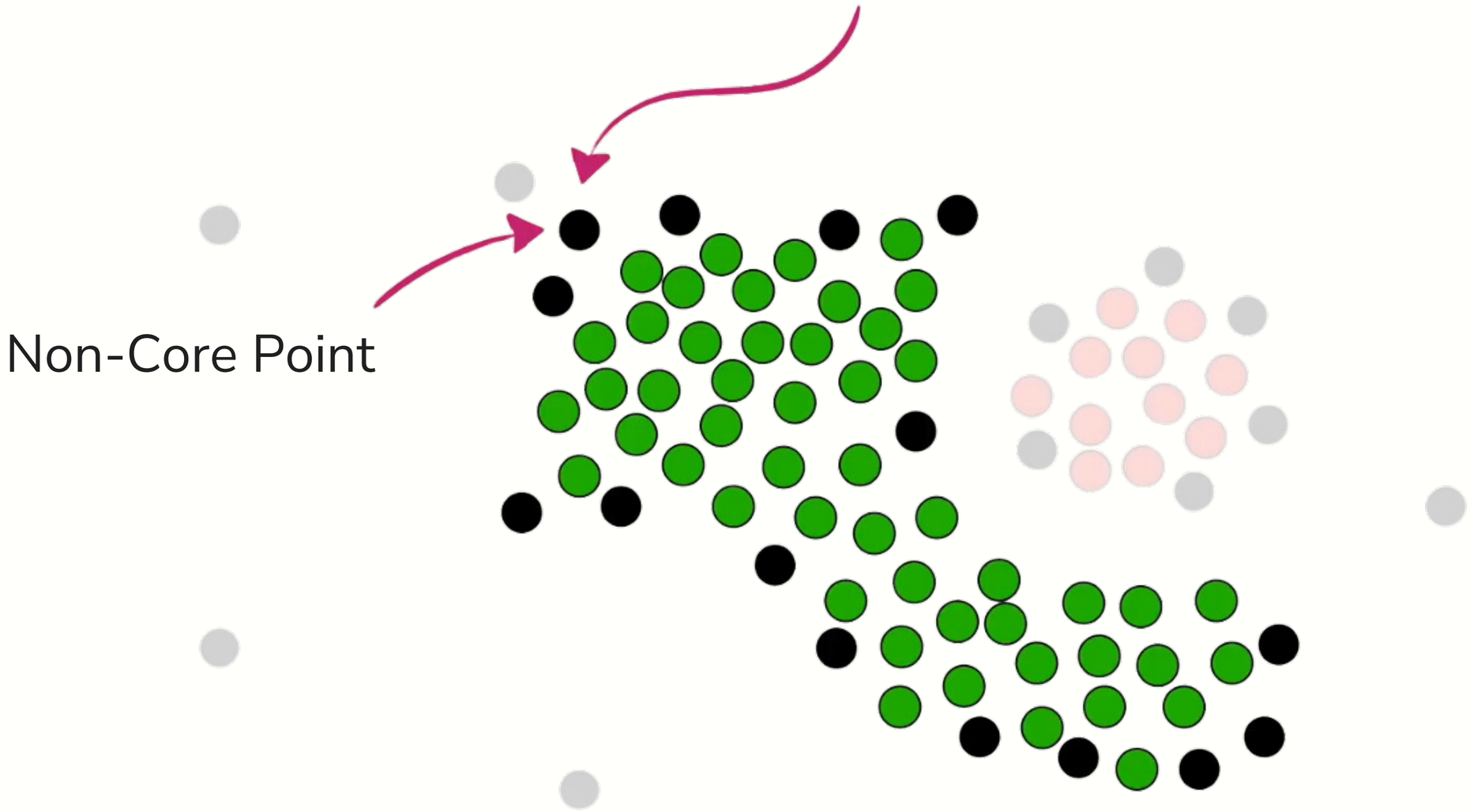
Ultimately, all of the Core Points that are close to the growing first cluster are added to it and then used to extend it further.

NOTE: At this point, every single point in the first cluster is a Core Point and because we can no longer add any more Core Points to the first cluster...

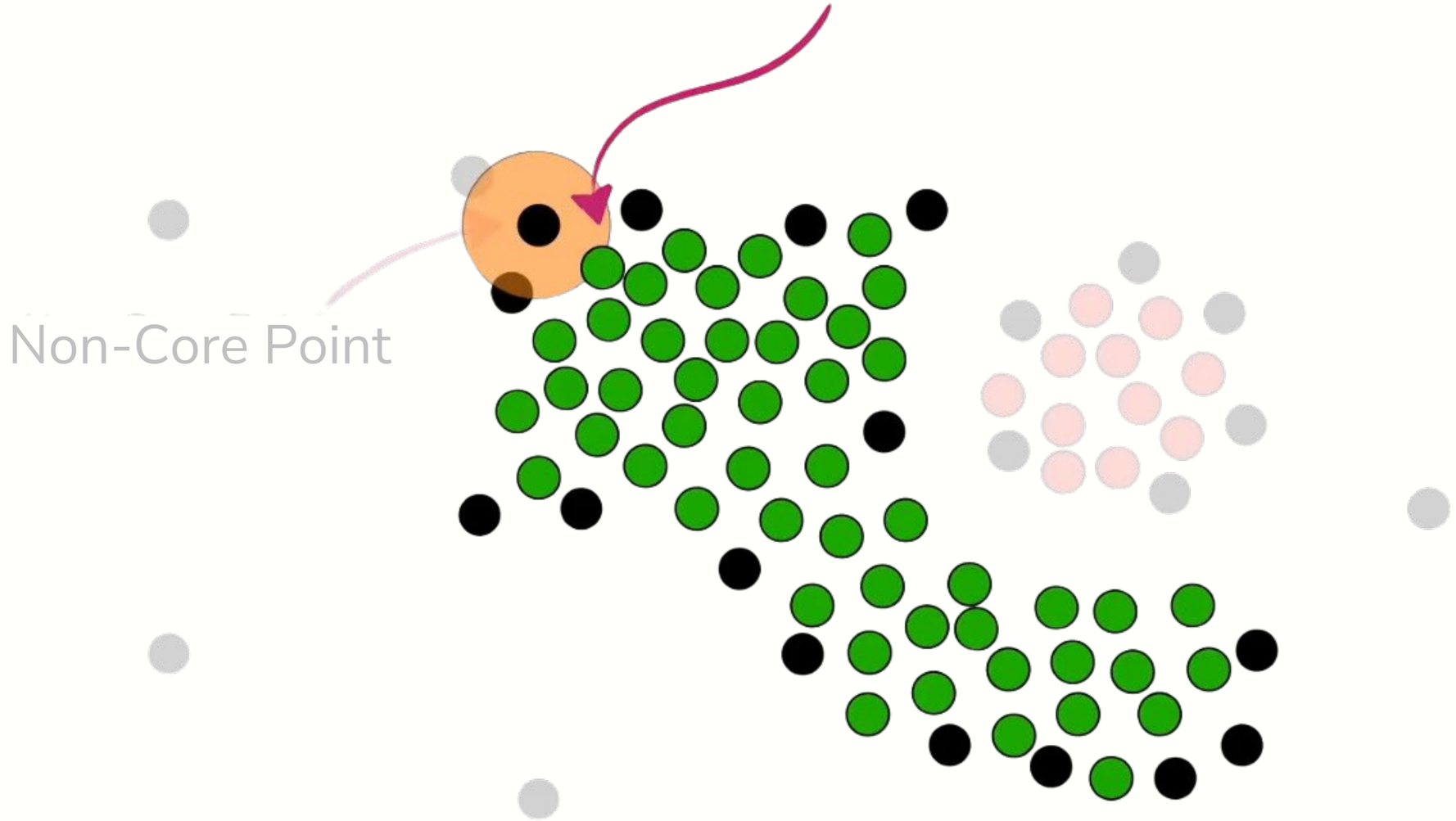


...we add all of the Non-Core Points that are close to Core Points in the first cluster .

For example, this point, which is a Non-Core Point...

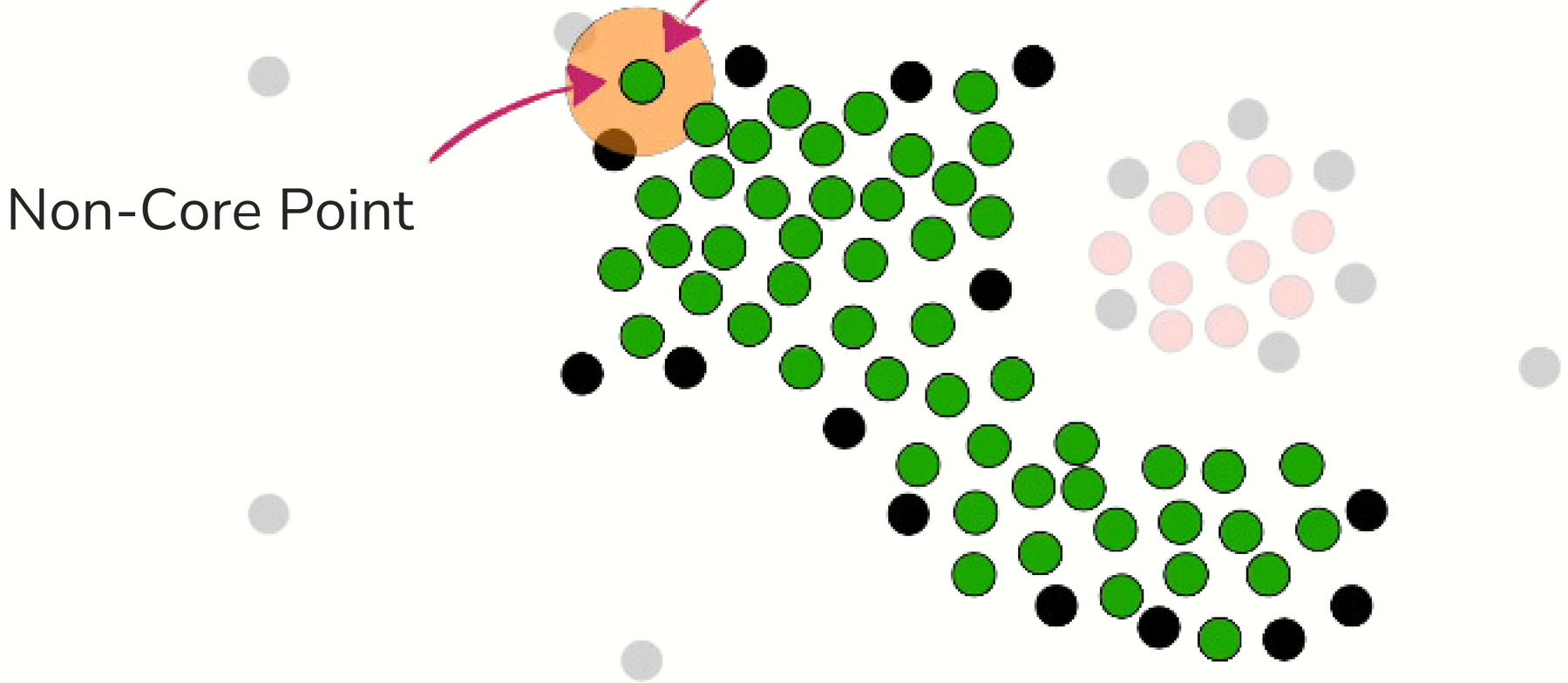


...is close to a Core Point in the **first cluster** ,



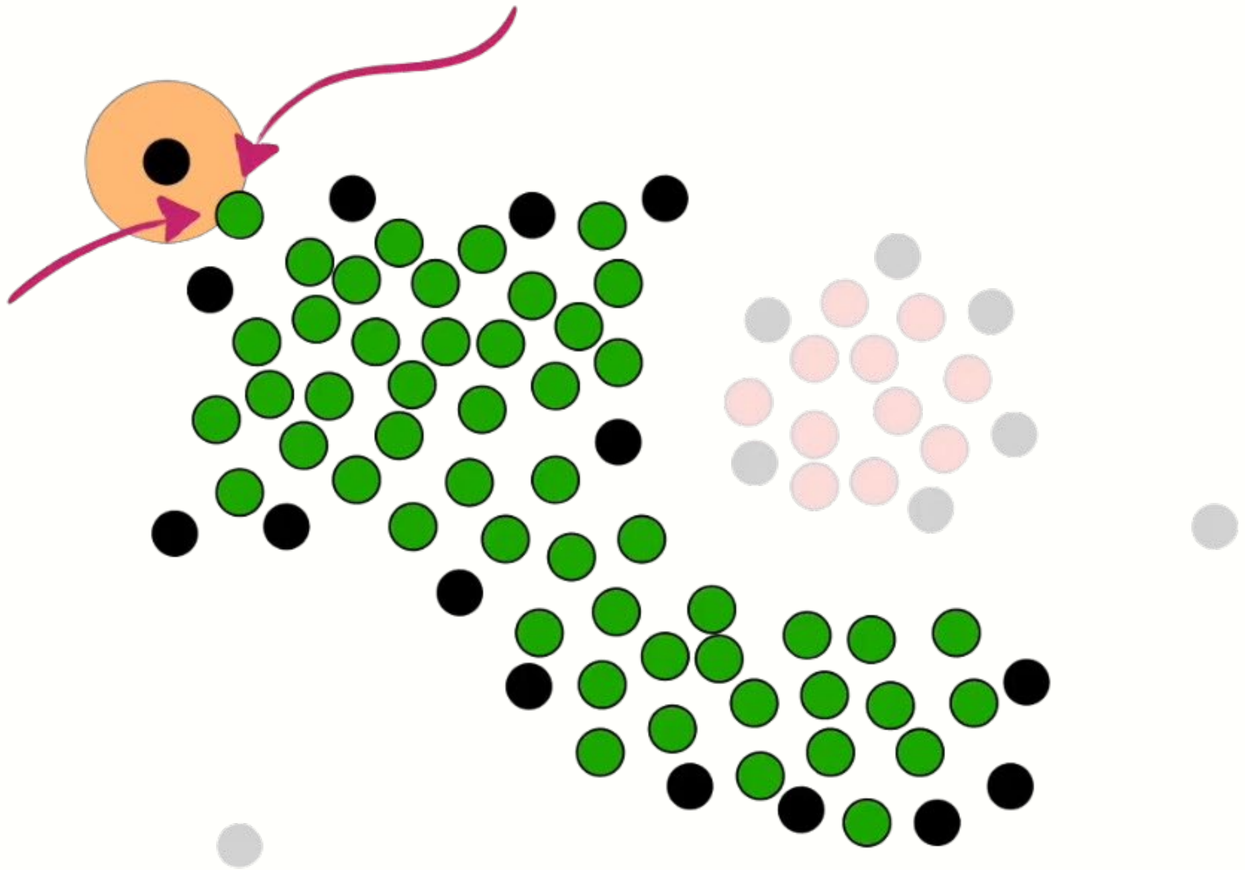
so we add it to **first cluster**

Non-Core Point

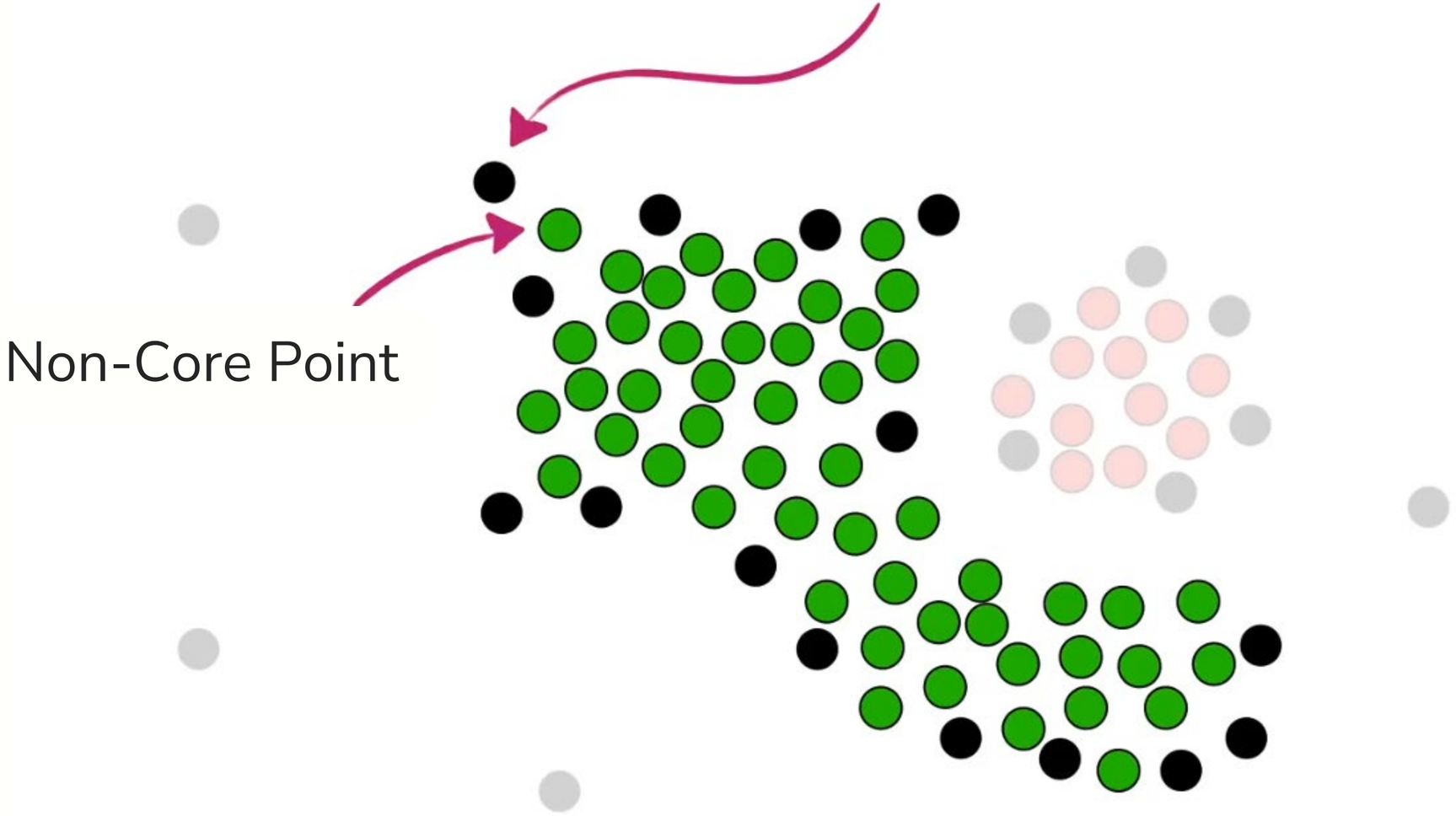


However, because this is not a Core Point, we do not use it to extend the **first cluster** any further.

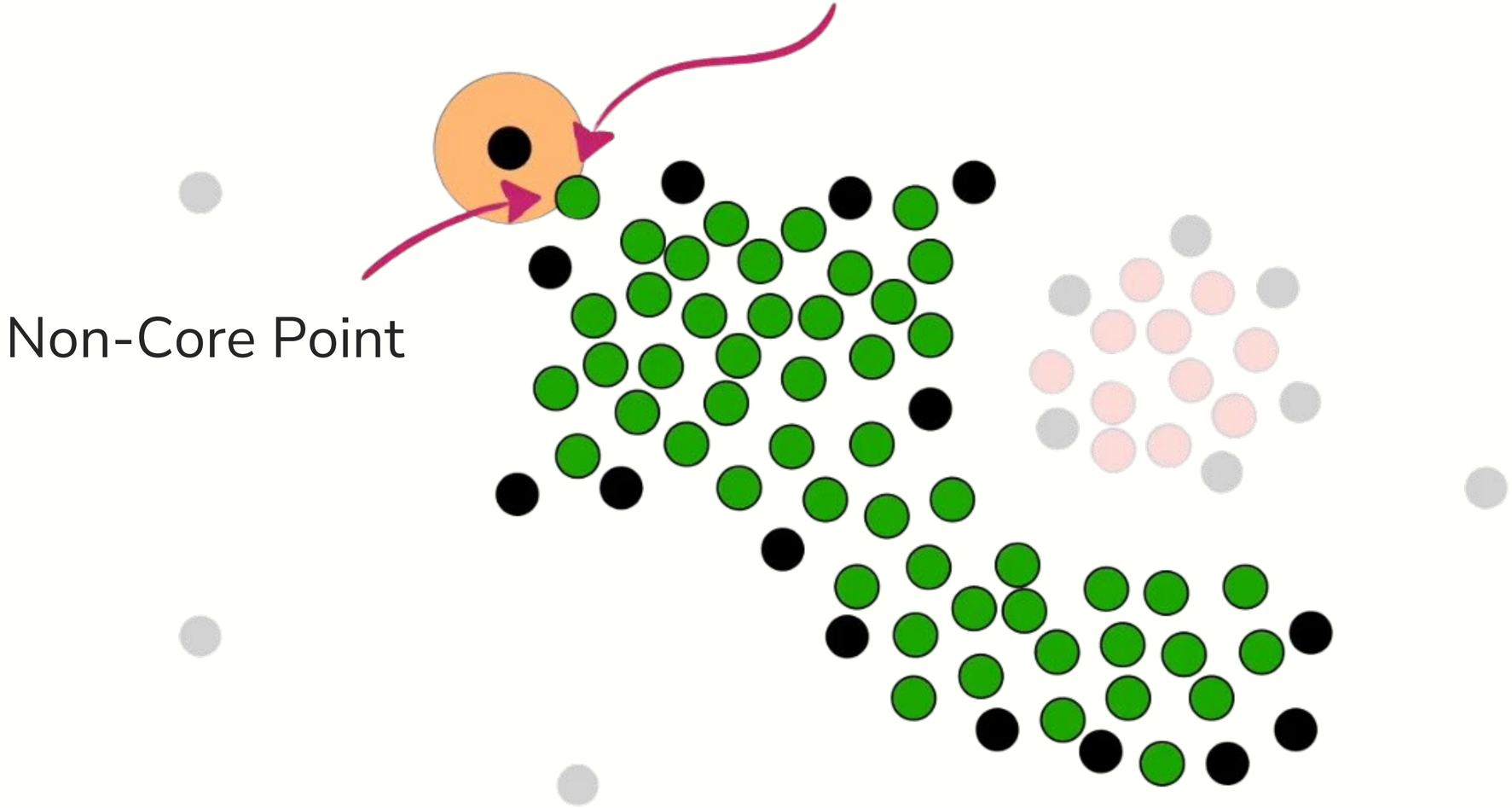
Non-Core Point



That means that this other Non-Core Point...



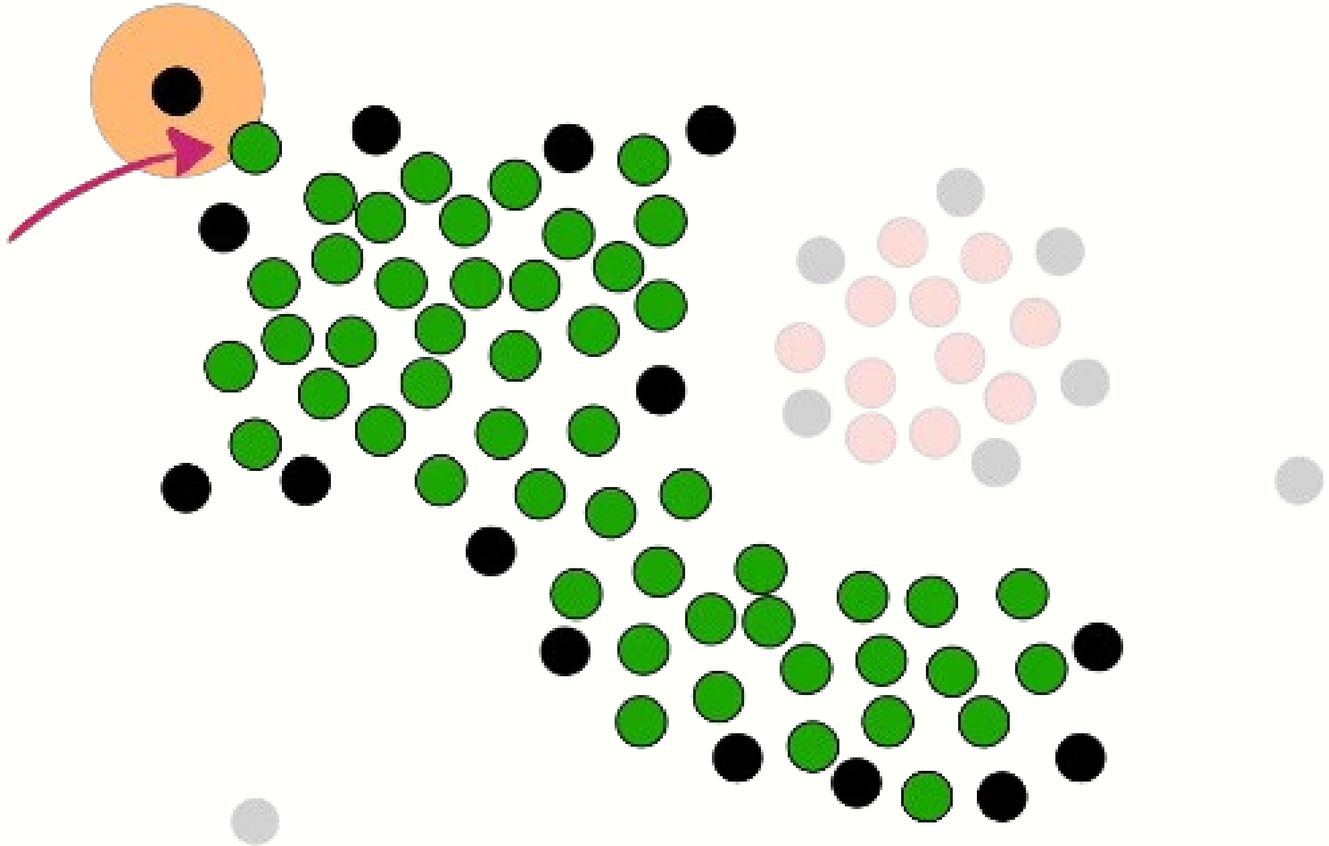
...which is close to the Non-Core Point that was just made part of the **first cluster** ...



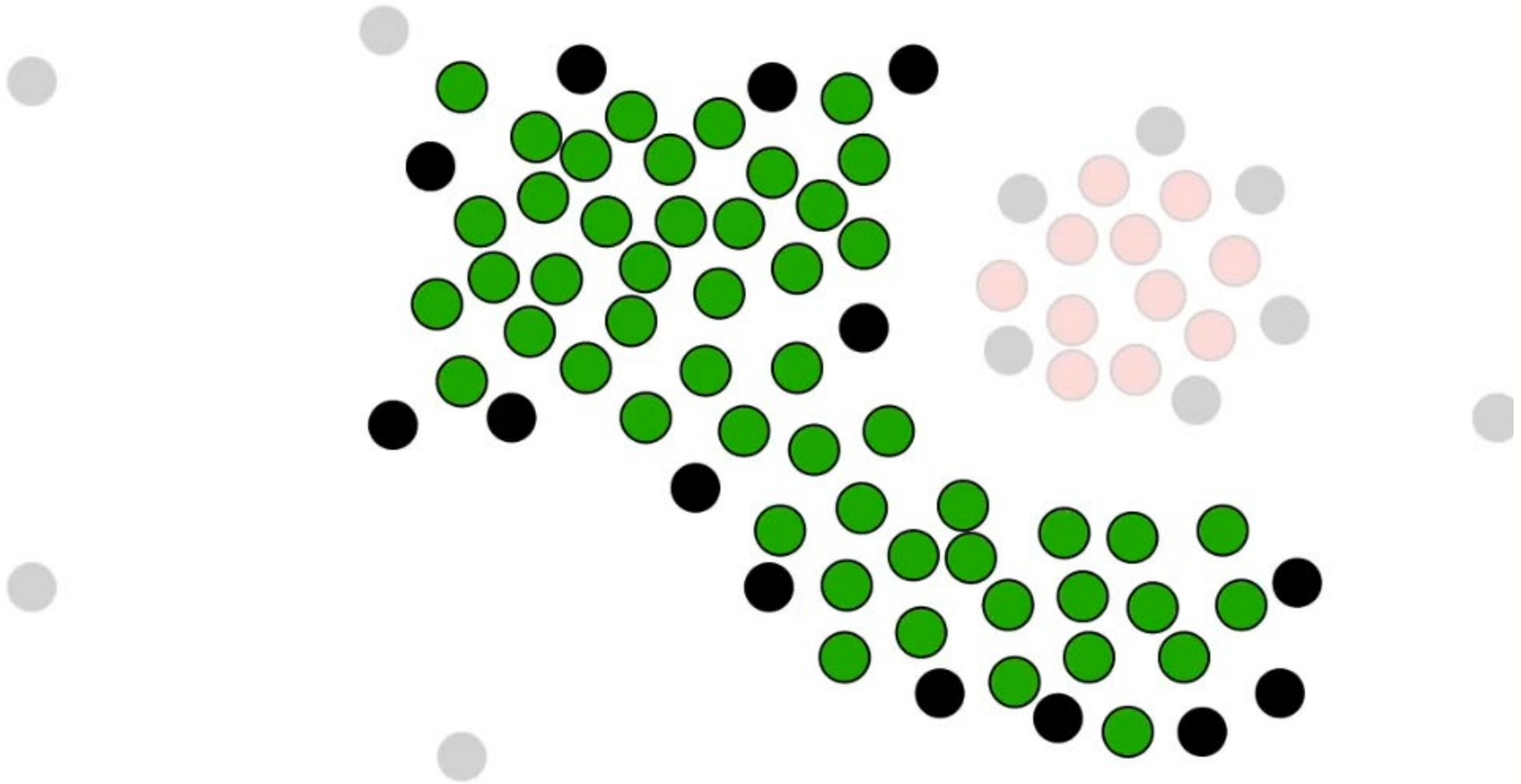
...will not be added to the **first cluster** because it is not close to a Core Point.

So, unlike Core Points, Non-Core Points can only join a cluster.
They can not extend it further.

Non-Core Point

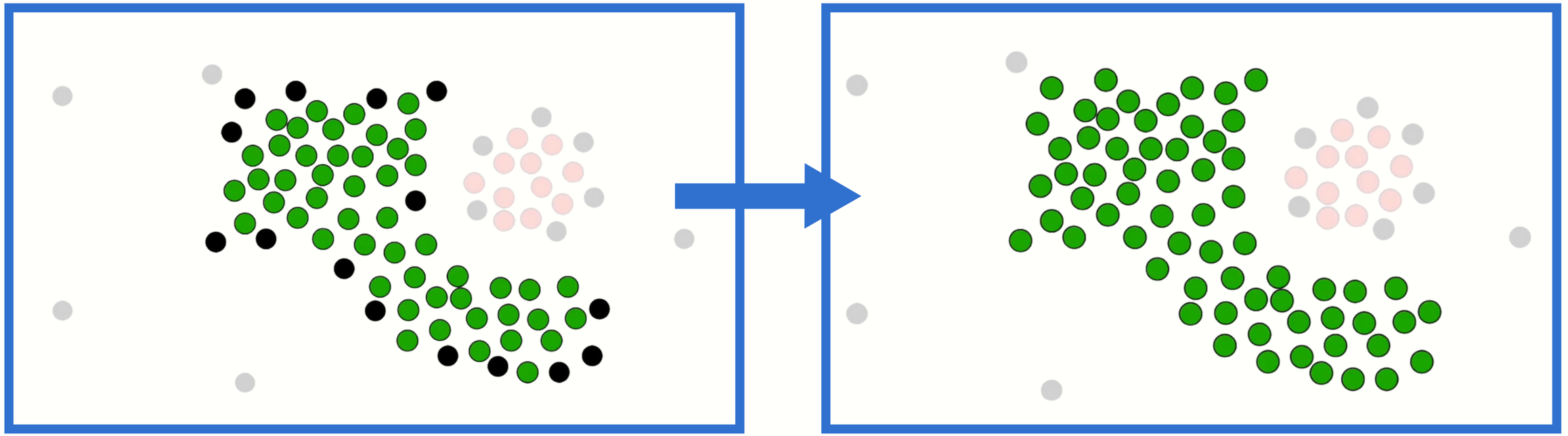


Now we add all of the Non-Core Points that are close Core Points in the **first cluster** to the **first cluster** .

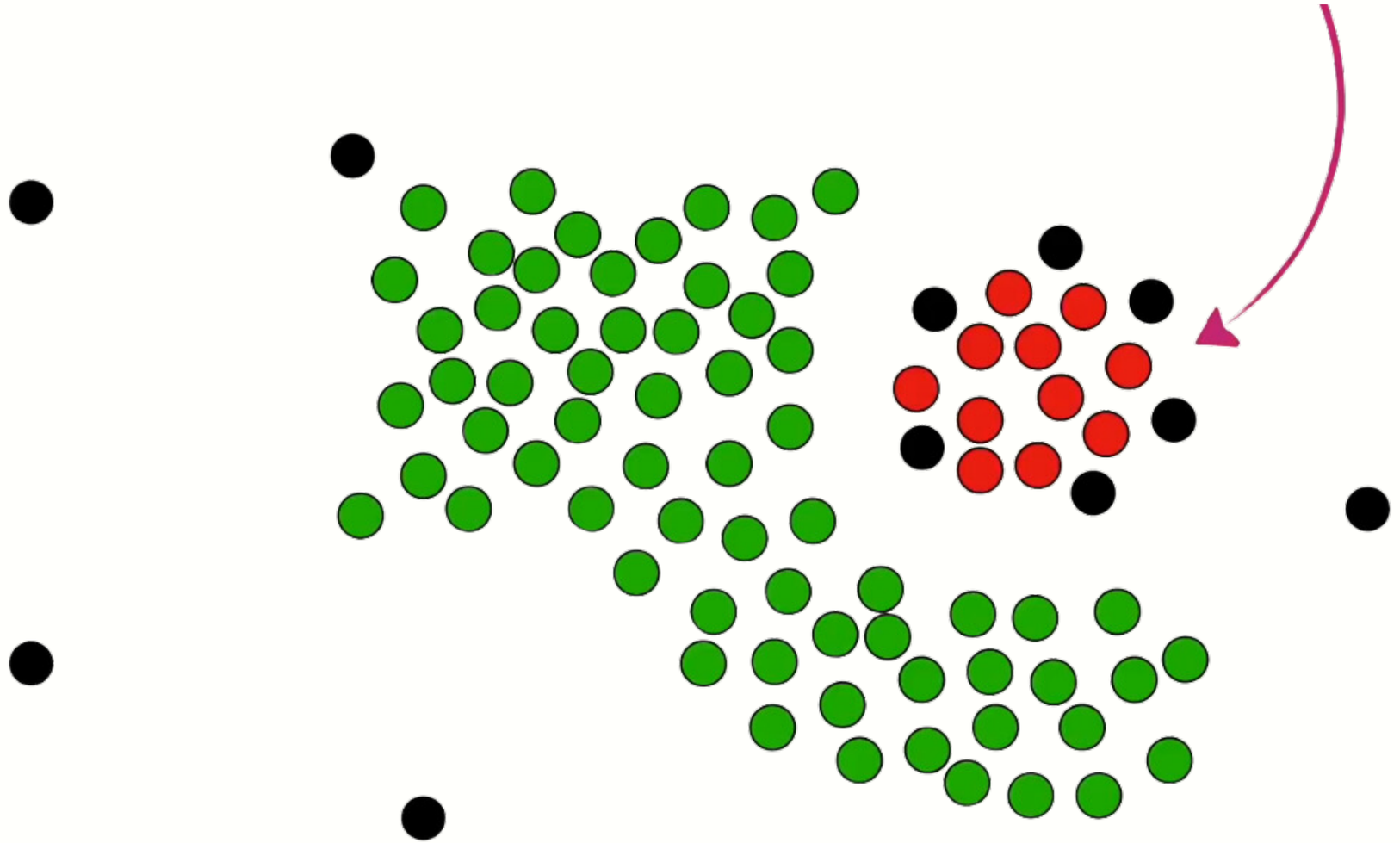


And now we are done creating the **first cluster** .

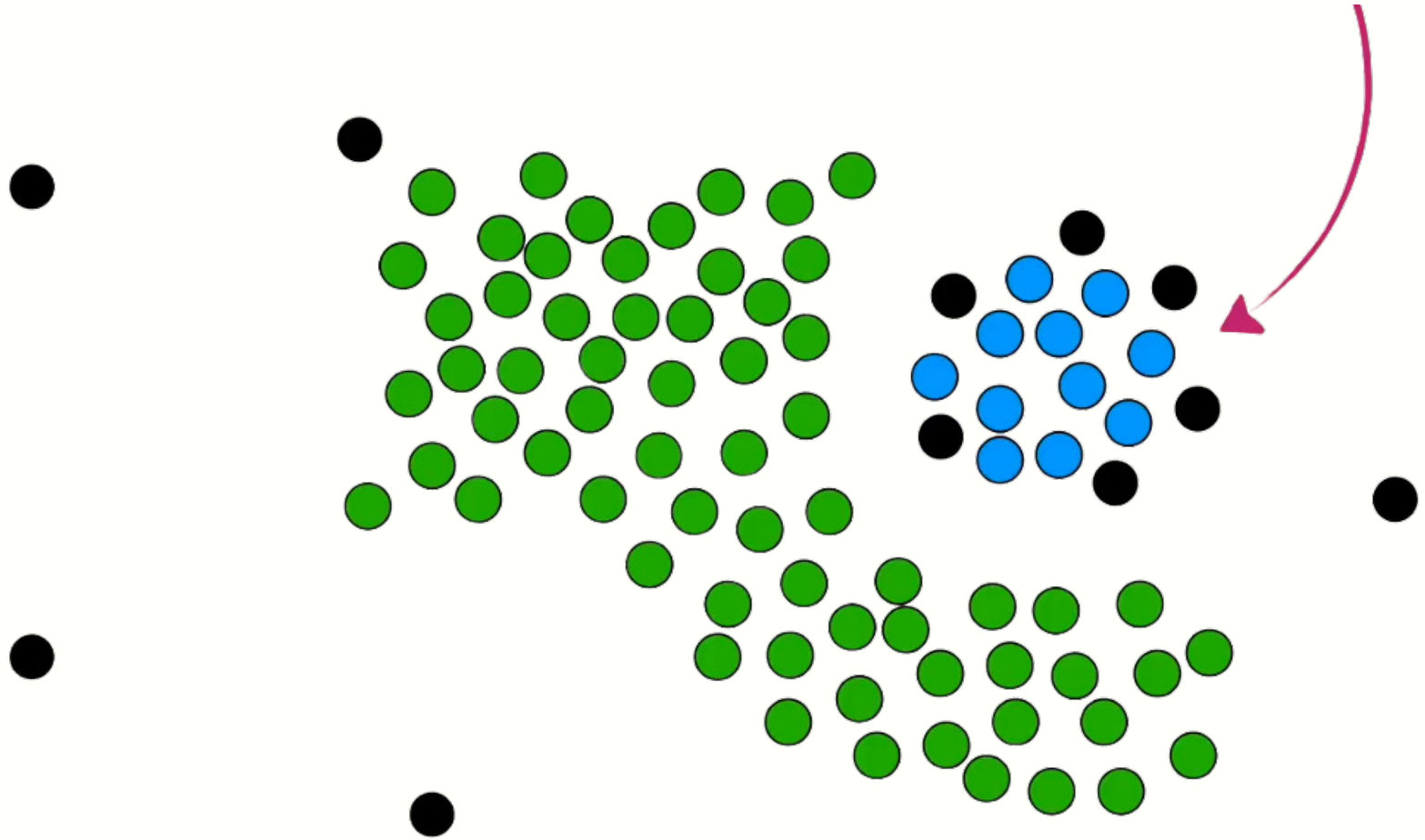
...and Non-Core Points only joined the **first cluster** .



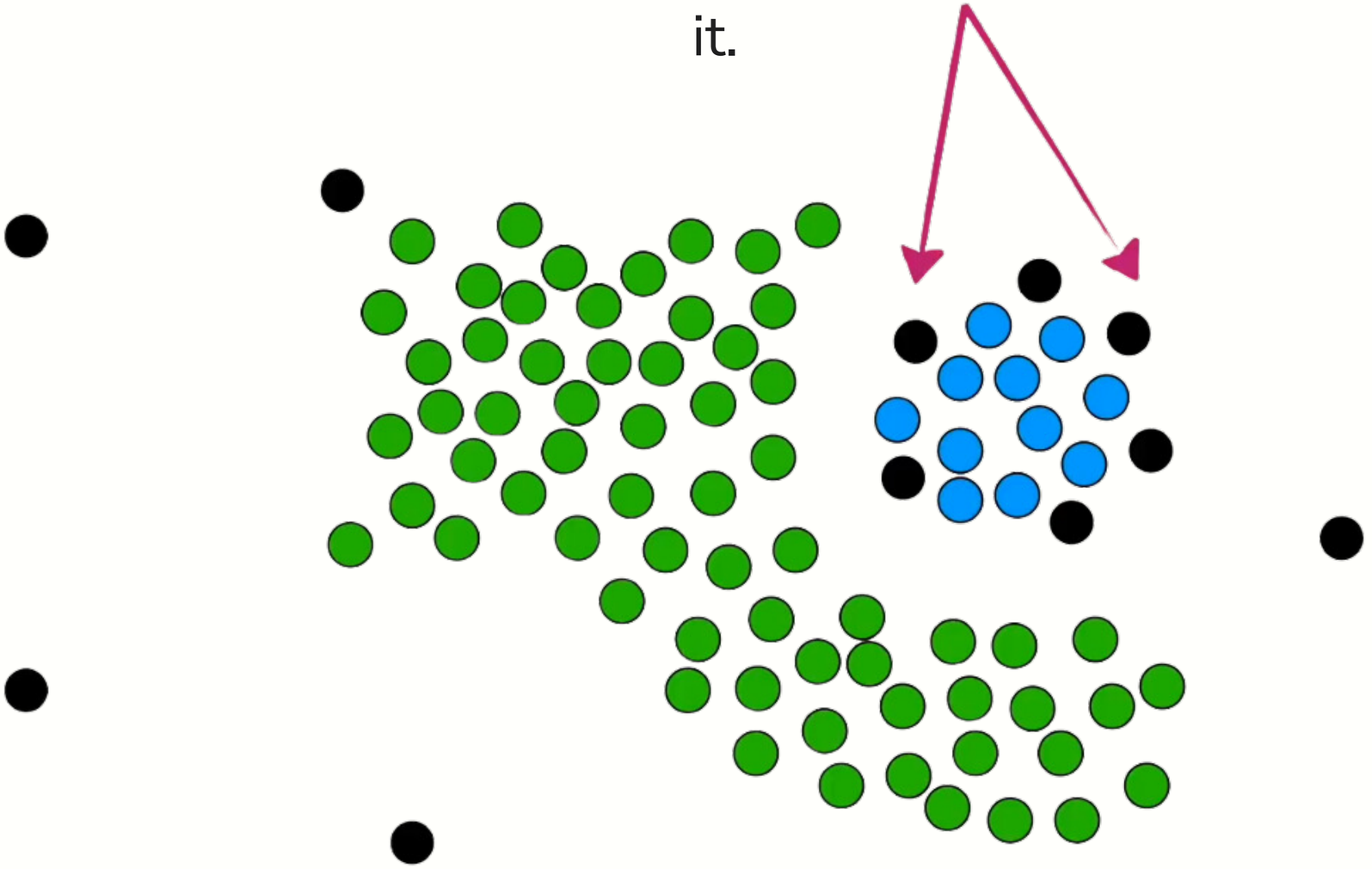
Now, because none of these Core Points are close to the **first cluster** ...



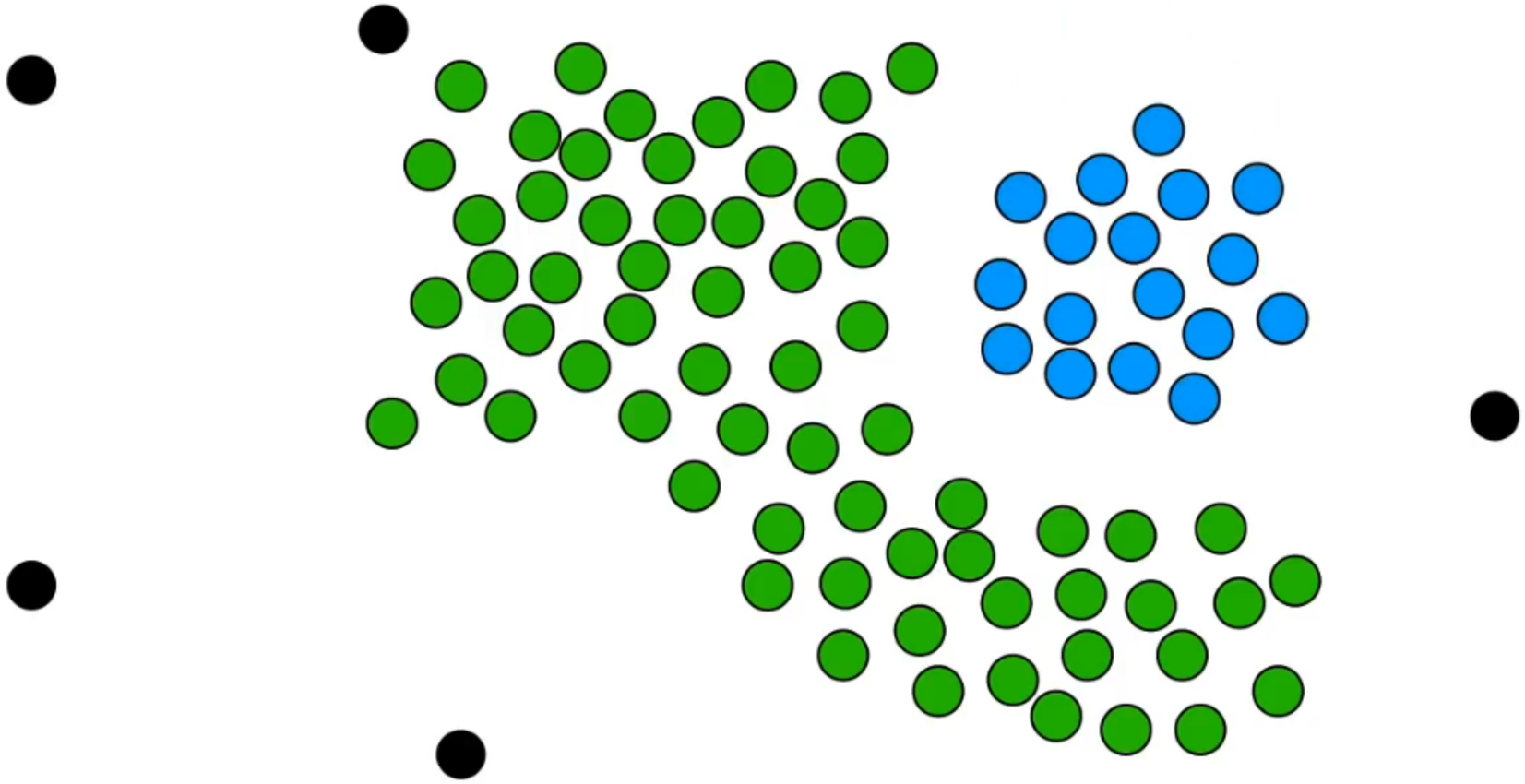
...they form a new, **second cluster** because they are close to each other...



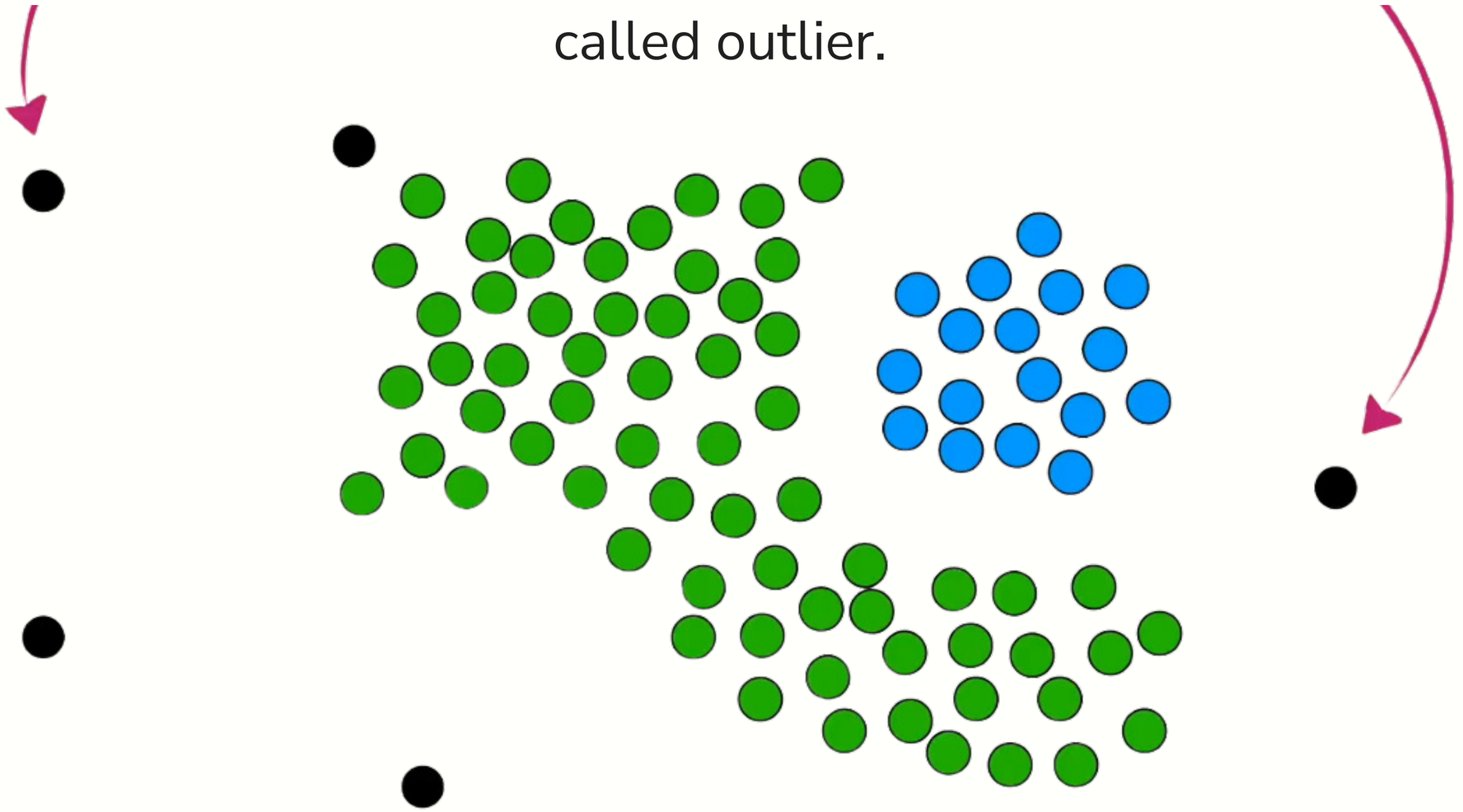
...and the Non-Core Points that are close to the **second cluster** are added to it.



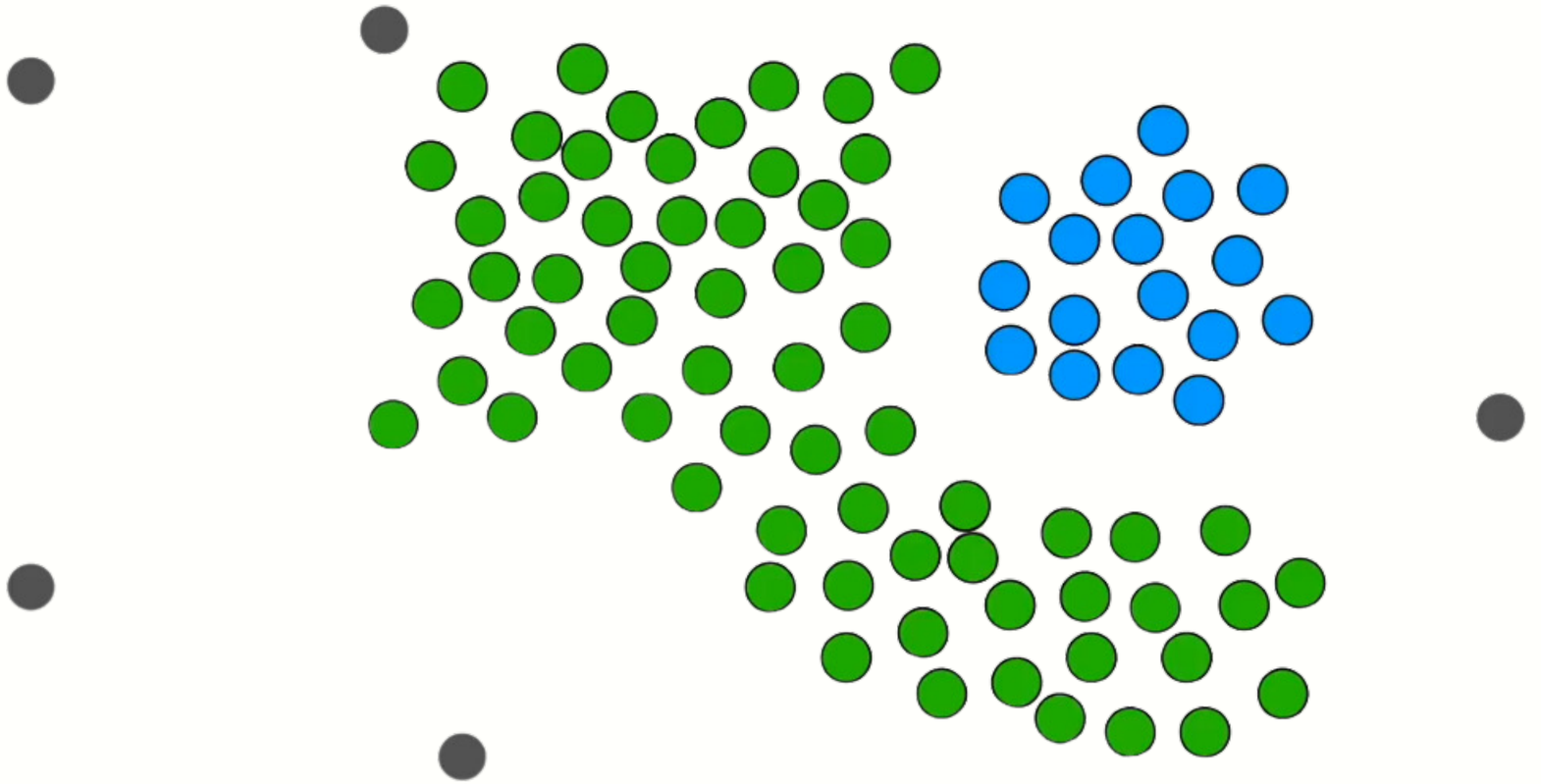
Lastly, because all of Core Points have been assigned to a cluster, we’re done making new clusters...



and any remaining Non-Core Points that are not close to Core Points in either cluster are not added to clusters and called outlier.

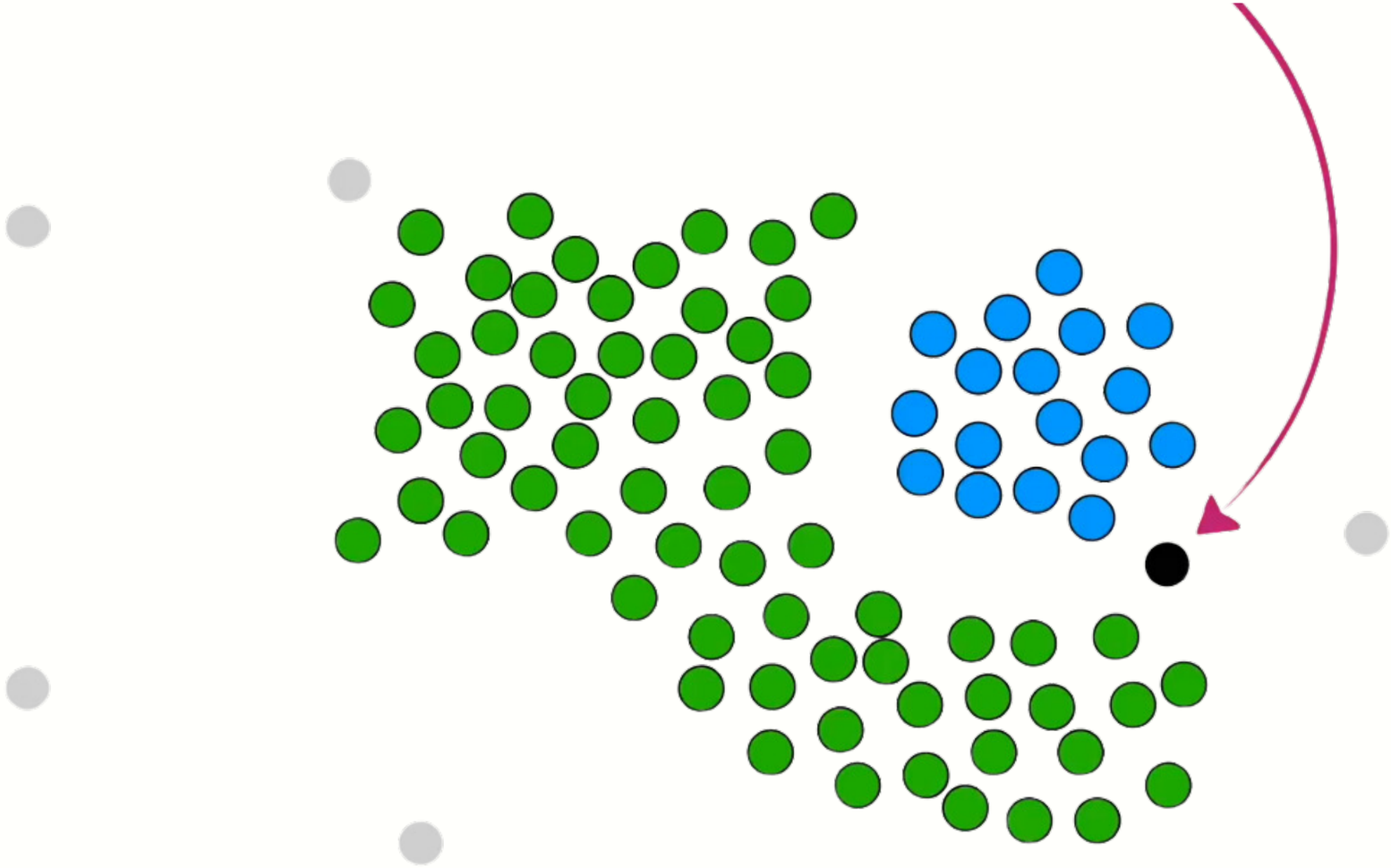


...and that is how the **DBSCAN algorithm** works!

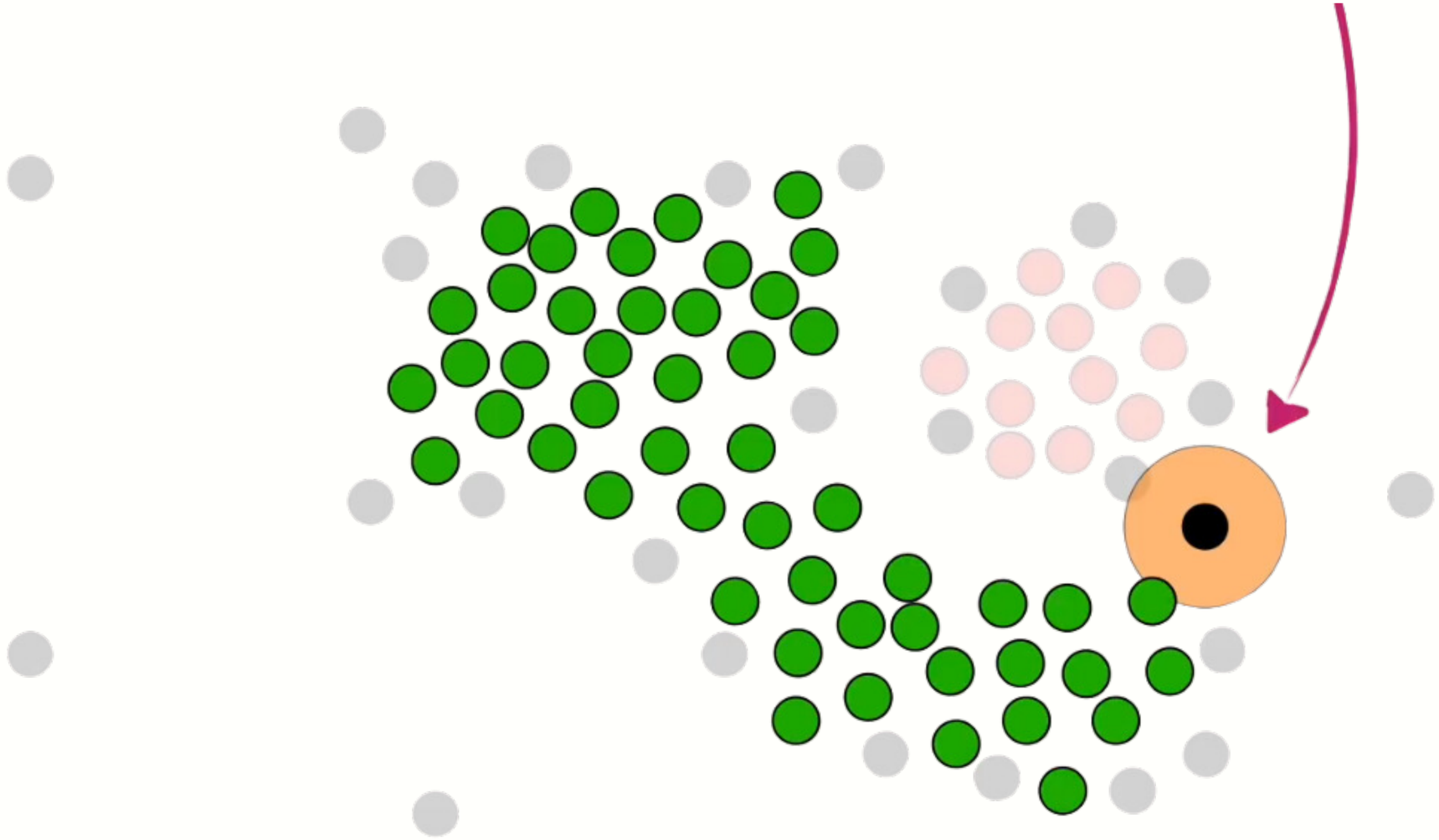


NOTE: As we just saw, clusters are created sequentially.

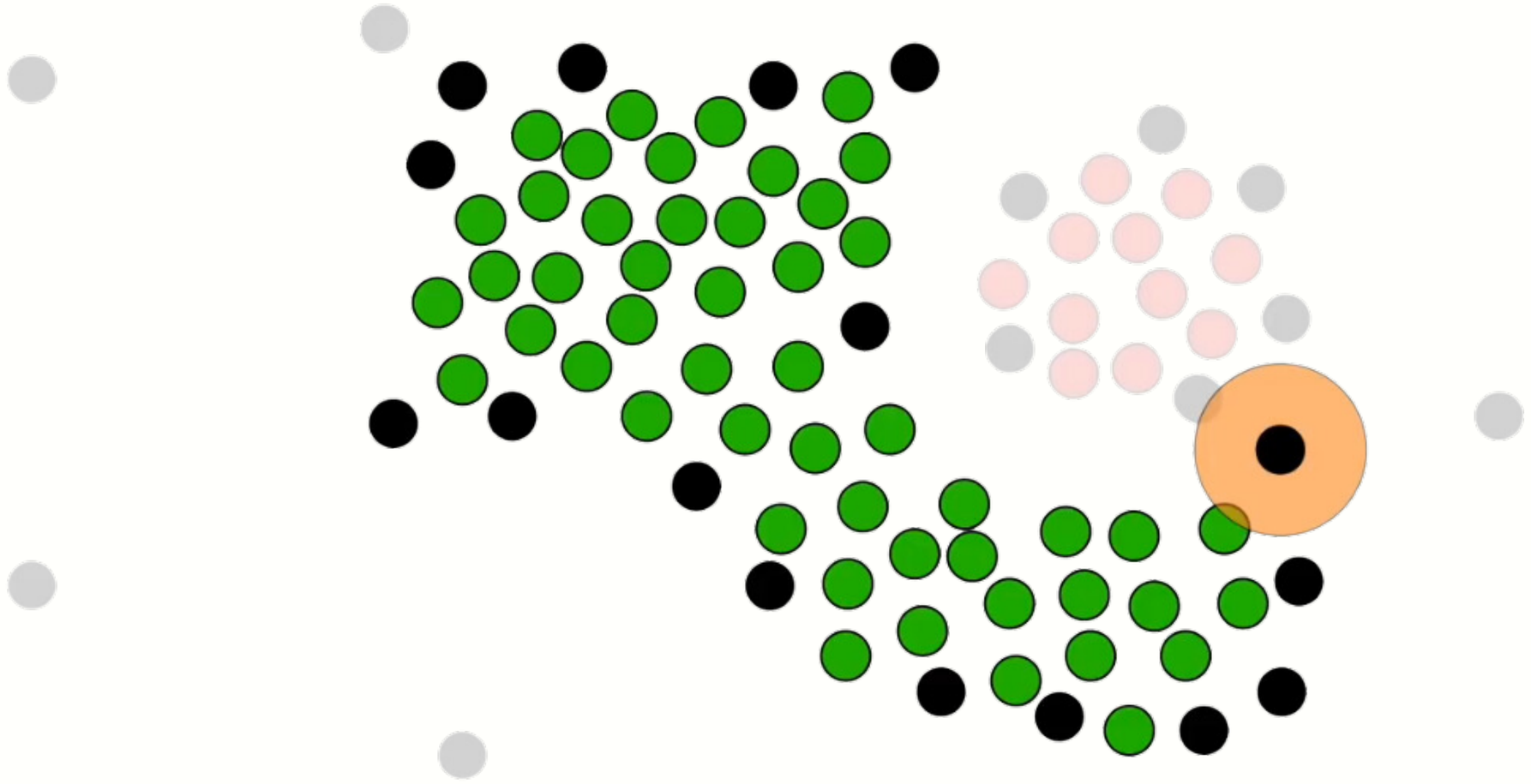
That means, if we had a Non-Core Point close to both clusters...



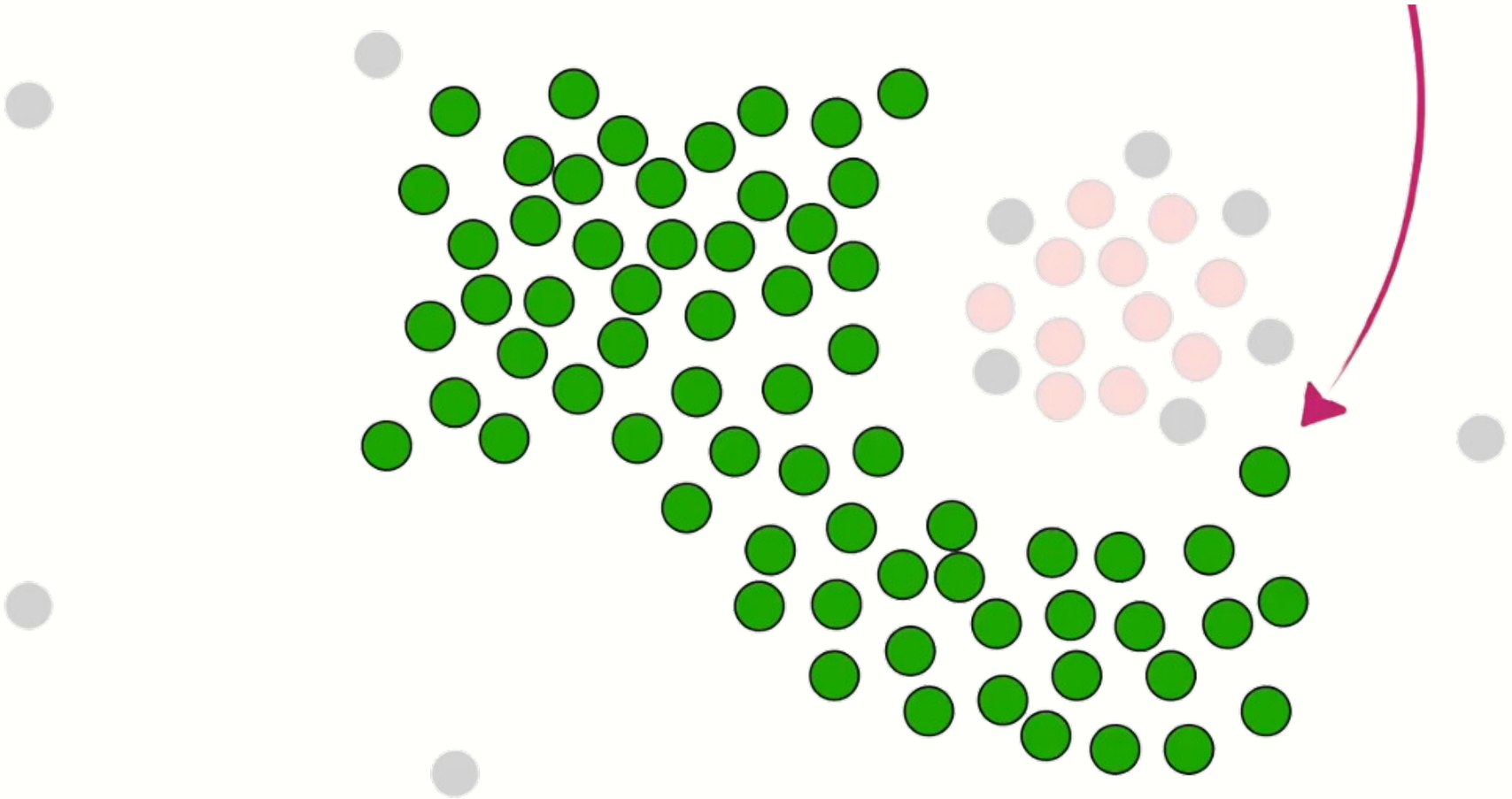
...we would add this Non-Core Point to the **first cluster**
because it is close to a Core Point...



...along with all of the other Non-Core Points that were close.



And now that this point is part of the **first cluster** ,
it is not eligible to be in any other cluster.

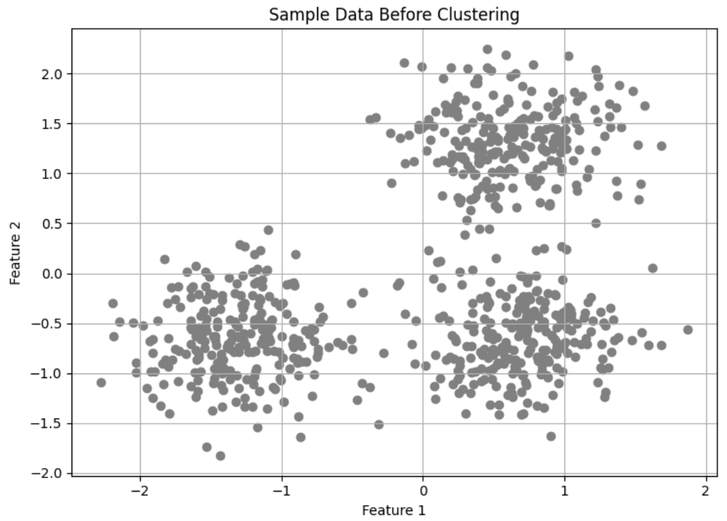


CODING TIME !!!!!!!

```python
# ====================== Initial ================================
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler

# Generate sample data
centers = [[1, 1], [-1, -1], [1, -1]]
X, _ = make_blobs(n_samples=750, centers=centers, cluster_std=0.4, random_state=0)
X = StandardScaler().fit_transform(X)

# Plotting before clustering
plt.figure(figsize=(6, 6))
plt.scatter(X[:, 0], X[:, 1], color='gray', marker='o')
plt.title('Sample Data Before Clustering')
plt.grid(True)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
# ==============================================================
```



```
eps=0.3, min_samples=10
```