Table of Contents

[2-Point Approach 2](#_Toc94380329)

[Solution: 2](#_Toc94380330)

[Pseudocode: 2](#_Toc94380331)

[Theoretical Asymptotic Complexity (Time Complexity) 3](#_Toc94380332)

[Data Structure and Memory: 3](#_Toc94380333)

# 2-Point Approach

## Solution:

* The solution to this problem is to go through all the possible 2 points combinations in any given number of points and then if all the other points exist either inside the chosen circle or on the circumference of it then this circle is a possible solution, but we can’t possibly say it’s the smallest circle until we have checked all the possible sets of points.

## Pseudocode:

listOfPoints-> dynamic array that stores all the given points

smallestCircle -> is circle object that has 2 points and each point has an X and Y coordinate and a get Area function, and the circle object can get Xc and Yc and Rc using built in functions. This object is set to null at first

N -> size of listOfPoints

for int i -> 0 loop while i < N-1, increment i by 1 for each iteration

for int j -> i + 1 loop while j < N, increment j by 1 for each iteration

P1 -> listOfPoints[i]

P2 -> listOfPoints[j]

C -> Circle(P1,P2)

if Circle C has a radius bigger than or equal to smallestCircle then continue j loop

for int k -> 0 loop while k < N, increment k by 1 for each iteration

if k != i and k != j then check

if listOfPoints[k] is not within Circle C or on circumference then continue j loop

inside j loop but outside k loop:

smallestCircle = circle C

outside All loops:

print smallestCircle

## Theoretical Asymptotic Complexity (Time Complexity)

|  |  |  |  |
| --- | --- | --- | --- |
| Statement | S/E | Frequency | Total Steps |
| listOfPoints -> gets all given points | 1 | N | N |
| smallestCircle -> null | 1 | 1 | 1 |
| N -> number of points | 1 | 1 | 1 |
| for int i -> 0 loop while i < N-1, increment i by 1 for each iteration | 1 | N | N |
| for int j -> i + 1 loop while j < N, increment j by 1 for each iteration | 1 | N-1(N+2)/2 | 2N2+2N-4 |
| P1 -> listOfPoints[i] | 1 | N-1(N)/2 | 2N2-2N |
| P2 -> listOfPoints[j] | 1 | N-1(N)/2 | 2N2-2N |
| C -> Circle(P1,P2) | 1 | N-1(N)/2 | 2N2-2N |
| if Circle C has a radius bigger than or equal to smallestCircle then continue j loop | 1 | N-1(N)/2 | 2N2-2N |
| for int k -> 0 loop while k < N, increment k by 1 for each iteration | 1 | (N-1(N)/2)(N+1) | N3-N |
| if k != i and k != j then check | 1 | (N-1(N)/2)(N) | N3-N2 |
| if listOfPoints[k] is not within Circle C or on circumference then continue j loop | 1 | (N-1(N)/2)(N) | N3-N2 |
| smallestCircle = circle C | 1 | N-1(N)/2 | 2N2-2N |
| print smallestCircle | 1 | 1 | 1 |
| Total |  | 3N3+10N2-7N-1 |  |

Time Complexity: O(N3) (Found by taking the highest power and getting rid of the coefficients)

I used Big O notation rather than checking for Omega or Theta, for the Omega checks for best case scenario, and theta checks for average scenario, but a programmer’s worst fear is the worst case scenario their algorithm produces which is what the Big O notation takes care of.

## Data Structure and Memory:

* The algorithm uses a dynamic array as the data structure to store all the given points into them which produces a space complexity of O(N). By using a dynamic array the algorithm doesn’t have to set a maximum number of given points and if the user decides to add more points to the algorithm, it doesn’t have to reinitialize the array like a normal static array would do since their sizes can’t be changed dynamically.