Programming Assignment 1

Yang Zhang

March 11, 2024

Statement of Integrity: I, Yang Zhang, attempted to answer each question honestly and to the best of my abilities. I cited any and all help that I received completing this assignment.

2(a). Construct an algorithm and return the distances between the m closest pairs of points

i.Pseudocode.

```
findClosestPairs(P, m)
1    distances = []
2    //Calculate distances between all pairs of points
3    for i from 0 to P.length - 1:
4        for j from i+1 to length(P) - 1:
5            //Euclidean Distance: sqrt((xi − xj)^2 + (yi − yj)^2)
6            distance = Euclidean_distance(P[i], P[j])
7            distances.append((distance, P[i], P[j]))
8    //Create a min-heap to maintain the smallest distances
9    min_heap = heap.min_heapify(distances)
10    //Extract the m smallest distances
11   closest_pairs = []
12   for _ form 1 to min(m, distances.length):
13       closest_pairs.append(min_heap.pop(distances))
14   return closest_pairs
```

ii. The worst-case running time

I start by presenting the findClosestPairs procedure with the time “cost” of each statement and the number of times each statement is executed. For each i =1,2,3,…,n where n = P.length, we let t_j denote the number of times the for loop test in line 4 is executed for that value of j . When a for loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body. That’s the

reason that n + 1 in line 3, t_j + 1 in line 4, and m + 1 in line 12. We also assume that comments are not executable statements, and so they take no time.

		cost	time
findClosestPairs(P, m)			
1	distances = []	c_1	1
2	//Calculate distances between all pairs of points	0	1
3	for i from 1 to P.length:	c_3	n + 1
4	for j from i+1 to length(P):	c_4	$\sum_{j=i+1}^{n} (t_j + 1)$
5	//Euclidean Distance: $\sqrt{(xi - xj)^2 + (yi - yj)^2}$	0	$\sum_{j=i+1}^{n} t_j$
6	distance = Euclidean_distance(P[i], P[j])	c_6	$\sum_{j=i+1}^{n} t_j$
7	distances.append((distance, P[i], P[j]))	c_7	$\sum_{j=i+1}^{n} t_j$
8	//Create a min-heap to maintain the smallest distances	0	1
9	min_heap = heap.min_heapify(distances)	c_9	t_9
10	//Extract the m smallest distances	0	1
11	closest_pairs = []	c_{11}	1
12	for _ form 1 to min(m, distances.length):	c_{12}	m + 1
13	closest_pairs.append(min_heap.pop(distances))	c_{13}	$\sum_{j=1}^{m} t_j$
14	return closest_pairs	c_{14}	1

The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute $c_i * n$ to the total running time.To compute T(n), we sum the products of the cost and times columns, obtaining

$$T(n) = c_1 1 + 0*1 + c_3(n+1) + c_4 \sum_{j=i+1}^{n} (t_j + 1) + 0* \sum_{j=i+1}^{n} t_j + c_6 \sum_{j=i+1}^{n} t_j + c_7 \sum_{j=i+1}^{n} t_j + 0*1 + c_9 t_9 +$$

$$0*1 + c_{11} 1 + c_{12}(m + 1) + c_{13} \sum_{j=1}^{m} t_j + c_{14} 1$$

The variable 'distances' points to a specific number of elements, which is $\binom{n}{2}$. In other words, the "distances" contain $\binom{n}{2}$ elements after the nested for loop. In addition, the Python built-in append() method takes the time complexity of O(1).

$$\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{n(n-1)}{2} = \sum_{j=i+1}^{n} t_j$$

Maintaining the heap property during the insertion or removal of an element into a heap of length k requires O(log k) time. In addition, getting the smallest or largest element from a heap is an operation that takes O(1) time.

$$\text{k} = \frac{n(n-1)}{2} \qquad t_9 = \log(\frac{n(n-1)}{2})$$

$$\sum_{j=1}^{m} t_j = \log(\frac{n(n-1)}{2} - 1) + \log(\frac{n(n-1)}{2} - 2) + \ldots + \log(\frac{n(n-1)}{2} - \text{m})$$

$$\sum_{j=i+1}^{n} 1 = \text{n} - 1 \text{ where i} = 1$$

To simplify T(n)

$$\text{T(n)} = c_1 1 + 0*1 + c_3(\text{n}+1) + c_4 \sum_{j=i+1}^{n} (t_j + 1) + 0* \sum_{j=i+1}^{n} t_j + c_6 \sum_{j=i+1}^{n} t_j + c_7 \sum_{j=i+1}^{n} t_j + 0*1 + c_9 t_9 +$$

$$0*1 + c_{11}1 + c_{12}(\text{m} + 1) + c_{13} \sum_{j=1}^{m} t_j + c_{14}1$$

$$\text{T(n)} = c_1 + c_3 + c_{11} + c_{12} + c_{14} - c_4 + \text{n}(c_3 + c_4) + \frac{n(n-1)}{2}(c_4 + c_6 + c_7) + c_9 \log(\frac{n(n-1)}{2}) +$$

$$c_{12}\text{m} + c_{13}(\log(\frac{n(n-1)}{2} - 1) + \log(\frac{n(n-1)}{2} - 2) + \ldots + \log(\frac{n(n-1)}{2} - \text{m}))$$

As we know keeping a min-heap takes O(log(k)) and getting the smallest element from the min-heap takes O(log(1)).

mlog(n^2) =2mlog(n)

Therefore, the time complexity of T(n) is

T(n) = O(n^2 + n + 2mlog(n))

Even for inputs of a given size, an algorithm's running time may depend on which input of that size is given as well as the size of m. For example, in findClosestPairs, the best case occurs if m = 1. Thus, the best-case running time is the dominant factor which is the O(n^2) part.

In findClosestPairs, the worst case occurs when m = $\binom{n}{2}$. Thus, the worst-case running time is the dominant factor which is the O(n^2log(n)).

(b) Algorithm Implementation.

```python
from math import sqrt
from math import comb
import random

import MinHeap

def find_closest_pairs(P, m):
    distances = []

    # Calculate distances between all pairs of points
    for i in range(len(P)):
        for j in range(i + 1, len(P)):
            distance = calculate_distance(P[i], P[j])
            distances.append([distance, P[i], P[j]])

    # Use a min-heap to maintain the m smallest distances
    min_heap = MinHeap.MinHeap()
    min_heap.heapify_arr(distances)

    # Extract the m smallest distances
    closest_pairs = [min_heap.heap_pop(distances) for _ in range(min(m, len(distances)))]

    return closest_pairs


def calculate_distance(p1, p2):
    return sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)
```

```python
class MinHeap:
    def heapify_arr(self, array):
        # Transform an array to a heap with O(len(array)).
        n = len(array)
        for i in reversed(range(n // 2)):
            self.sift_up(array, i)

    def sift_up(self, array, i):
        endpos = len(array)
        startpos = i
        newitem = array[i]
        childpos = 2 * i + 1

        while childpos < endpos:
            rightpos = childpos + 1
            if rightpos < endpos and not array[childpos][0] < array[rightpos][0]
                childpos = rightpos

            array[i] = array[childpos]
            i = childpos
            childpos = 2 * i + 1

        array[i] = newitem
        self.shift_down(array, startpos, i)

    def shift_down(self, array, startpos, i):
        newitem = array[i]

        while i > startpos:
            parentpos = (i - 1) >> 1
            parent = array[parentpos]
            if newitem[0] < parent[0]:
                array[i] = parent
                i = parentpos
                continue
            break
        array[i] = newitem

    def heap_pop(self, heap):
        lastelt = heap.pop()
        if heap:
            returnitem = heap[0]
            heap[0] = lastelt
            self.sift_up(heap, 0)
            return returnitem
        return lastelt
```

```python
if __name__ == "__main__":

    a = input("Please input how many pairs you want to generate: ")

    #Error checks
    while not a.isdigit():

        print("Your input must be a positive integer value, please try again!")
        a = input("Please input how many pairs you want to generate: ")


    a = int(a)

    #Error checks

    while a < 2:
        print("Your input must be greater than 1, please try again!")
        a = input("Please input how many pairs you want to generate: ")
        a = int(a)


    m = input("Please input how many closest pairs you want to get: ")


    #Error checks
    while not m.isdigit():

        print("Your input must be an integer value, please try again!")
        m = input("Please input how many closest pairs you want to get: ")

    m = int(m)

    #Error checks
    while m < 1:
        print("Your input must be greater than 0, please try again!")
        m = input("Please input how many closest pairs you want to get: ")

        m = int(m)


    combination = comb(a, 2)

    #Error checks
    while m > combination:

        print("m must be not greater than " + str(combination) +", please try again!")

        m = input("Please input how many closest pairs you want to get: ")

        m = int(m)

    points = []

    for _ in range(a):
        ran1 = random.randint(1, 100)
        ran2 = random.randint(1, 100)
        points.append([ran1, ran2])
    print("————————————————————————————————————————————————————————————————————————————————")
    print("Your generated pairs are: ", points)

    '''
    #Example usage from the meeting:
    points = [[1, 1], [0, 0], [10, 10], [20, 20], [35, 40],
          [5, 2], [-6, -7], [100, 100], [-3, 42], [5, 10]]
    m = 10
    '''

    #Generate the result
    print("————————————————————————————————————————————————————————————————————————————————")
    closest_pairs = find_closest_pairs(points, m)
    for distance, p1, p2 in closest_pairs:
        print(f"Distance: {distance}, Points: {p1}, {p2}")
```

(c) Trace runs demonstrating the proper functioning of the code.

```
Your generated pairs are:  [[1, 1], [0, 0], [10, 10], [20, 20], [35, 40], [5, 2]
, [-6, -7], [100, 100], [-3, 42], [5, 10]]
m is :  10
Distance: 1.4142135623730951, Points: [1, 1], [0, 0]
Distance: 4.123105625617661, Points: [1, 1], [5, 2]
Distance: 5.0, Points: [10, 10], [5, 10]
Distance: 5.385164807134504, Points: [0, 0], [5, 2]
Distance: 8.0, Points: [5, 2], [5, 10]
Distance: 9.219544457292887, Points: [0, 0], [-6, -7]
Distance: 9.433981132056603, Points: [10, 10], [5, 2]
Distance: 9.848857801796104, Points: [1, 1], [5, 10]
Distance: 10.63014581273465, Points: [1, 1], [-6, -7]
Distance: 11.180339887498949, Points: [0, 0], [5, 10]
```

```
Your generated pairs are:  [[-81, -69], [-37, -5], [97, -73]]
m is :  2
Distance: 77.6659513557904, Points: [-81, -69], [-37, -5]
Distance: 150.2664300500947, Points: [-37, -5], [97, -73]
```

```
Your generated pairs are:  [[-96, 41], [54, -58], [-89, 61], [66, -56], [56, 42]
, [78, 28], [53, 24], [16, -100], [-38, -88], [63, -7], [13, 73], [48, 14], [-69
, -74], [-94, -65], [-44, -45]]
m is :  5
Distance: 11.180339887498949, Points: [53, 24], [48, 14]
Distance: 12.165525060596439, Points: [54, -58], [66, -56]
Distance: 18.24828759089466, Points: [56, 42], [53, 24]
Distance: 21.18962010041709, Points: [-96, 41], [-89, 61]
Distance: 25.317977802344327, Points: [78, 28], [53, 24]
```

```
Your generated pairs are:  [[-74, 72], [-74, 35], [-90, 15], [-28, 16]]
m is :  3
Distance: 25.612496949731394, Points: [-74, 35], [-90, 15]
Distance: 37.0, Points: [-74, 72], [-74, 35]
Distance: 49.76946855251722, Points: [-74, 35], [-28, 16]
```

```
Your generated pairs are:  [[32, 56], [45, 8], [79, 58], [26, 66], [17, 22]]
m is :  3
Distance: 11.661903789690601, Points: [32, 56], [26, 66]
Distance: 31.304951684997057, Points: [45, 8], [17, 22]
Distance: 37.16180835212409, Points: [32, 56], [17, 22]
```

(d) Perform tests to measure the asymptotic behavior of the program

Size of Points (m = 20)	Cost (Increment Comparison)
10	1
20	2.08
50	1.93
100	2.13
200	3.68
500	6.48
1000	16.31
2000	65.38
5000	416.69
10000	3921.20

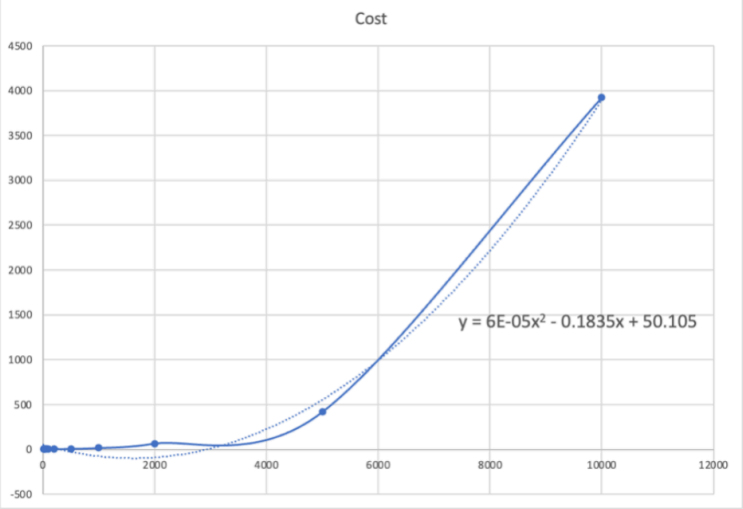


Figure 1

Size of Points $\binom{n}{2}$	Cost (Increment Comparison)
10	1
20	4.98
30	10.76
50	31.48
100	115.72
200	443.99
210	494.65
215	600.63
220	540.07
225	552.21
250	708.15
300	999.40

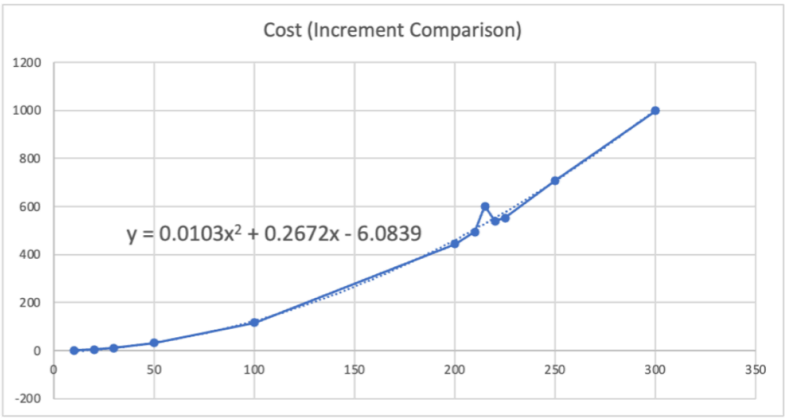


Figure 2

(e) Compare algorithm's worst-case running time to the code's worst-case running time.

Given that m = 20 in Figure 1, the dotted line represents a trendline closely mirroring the real data. The generated equation is polynomial, suggesting a potential time complexity of O(n^2). As demonstrated in the pseudocode earlier, the dominant factor n^2 arises when m is small.

In Figure 2, the number of closest pairs, denoted as m = $\binom{n}{2}$, represents the worst-case scenario. It indicates that the maximum number of closest pairs for every input is always generated. The dotted trendline closely aligns with the real data. In this scenario, m cannot be overlooked, as it would become the dominant factor.

Figure 2 highlights that when m = = $\frac{n(n-1)}{2}$, the dominant factor escalates to 2mlog(n)), showcasing the significance of m when it reaches a considerable size. However, discerning whether the dominant component encompasses log(n) is challenging due to the limitations of the laptop's performance. Utilizing a min-heap in the code, known for its O(log(k)) complexity in maintenance, suggests that the dominant aspect should include log(k).

size of Points (m = 20)	real cost	pseudocode cost
10	1	1
20	2.08	2.46
50	1.93	11.70
100	2.13	44.08
200	3.68	173.07
500	6.48	1075.07
1000	16.31	4295.82
2000	65.38	17178.32
5000	416.69	107354.86
10000	3921.2	429413.29

size of Points (m = $\binom{n}{2}$)	real cost	pseudocode cost
10	1	1
20	4.98	5.04
30	10.76	12.71
50	31.48	40.05
100	115.72	185.52
200	443.99	842.17
210	494.65	936.22
215	600.63	985.23
220	540.07	1035.58
225	552.21	1087.27
250	708.15	1365.91
300	999.4	2025.62

Based on the information from the two tables, it's evident that the real implementation of the algorithm generally outperforms the pseudocode in terms of worst-case running time. Upon comparing the first and second tables, we observe that the pseudocode's cost is significantly closer to the real cost in the latter. This discrepancy can be attributed to our deliberate selection of the largest number of closest pairs for each set of points, thus edging closer to extreme scenarios. Consequently, as both m and n increase, the pseudocode's cost converges toward the real cost.

The real implementation is typically faster due to its ability to leverage specific data structures and algorithms optimized for performance. In contrast, pseudocode operates at a higher level, omitting implementation specifics. Additionally, real code benefits from compiler, hardware, and

language-specific optimizations, enhancing its performance. Pseudocode, being language-agnostic and high-level, lacks these optimizations, thereby trailing behind real implementations in terms of speed and efficiency.

3(a) Now that you have designed, implemented, and tested your algorithm, what aspects of your algorithm and/or code could change and reduce the worst-case running time of your algorithm? Be specific in your response to this question.

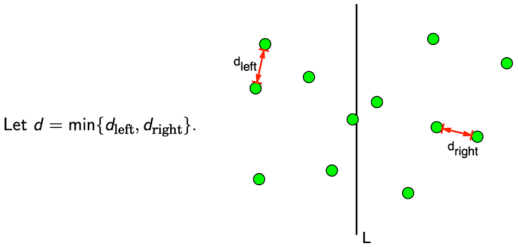
A brute force approach can be employed to tackle this problem, utilizing a nested for loop and a built-in method like sorted(). However, the sorted() method has a time complexity of O(klog(k)) and it is more sensitive to an ordered list in reverse. Consequently, the overall cost becomes O($n^2 log n$). In contrast, utilizing a heap structure incurs a maintenance cost of O(log(k)), with retrieving the smallest element from the heap accomplished in O(1) time. Hence, the overall cost is O(mlog(n)), where m is less than n^2, indicating superior performance.

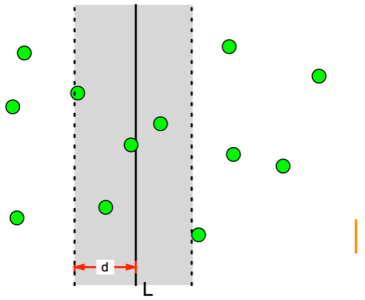
Initially, I aim to circumvent the use of nested for loops. The method outlined below, with a time complexity of O(nlog(n)), offers an alternative to finding the closest pair of points without resorting to nested loops. To achieve this, at each iteration, I ascertain the 1st, 2nd, 3rd, …, m-th closest pair by employing the expression min(max(d1, the last closest distance), max(d2, the last closest distance)). Nevertheless, I encountered difficulty in obtaining the m closest pairs of points. There may exist mathematical approaches to address this challenge, potentially mitigating the worst-case running time.

```
ClosestPair(Px, Py):
   if |Px| == 2: return dist(Px[1],Px[2])    // base

   d1 = ClosestPair(FirstHalf(Px,Py))     // divide
   d2 = ClosestPair(SecondHalf(Px,Py))
   d = min(d1,d2)

   Sy = points in Py within d of L     // merge
   For i = 1,...,|Sy|:
       For j = 1,...,15:
           d = min( dist(Sy[i], Sy[j]), d )
   Return d
```

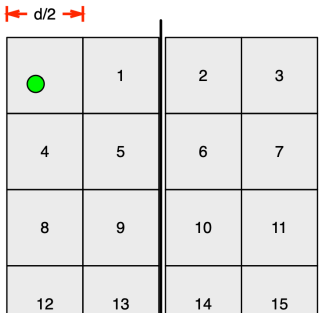




Theorem

Suppose $S_y = p_1, \ldots, p_m$. *If* $dist(p_i, p_j) < d$ *then* $j - i \leq 15$.

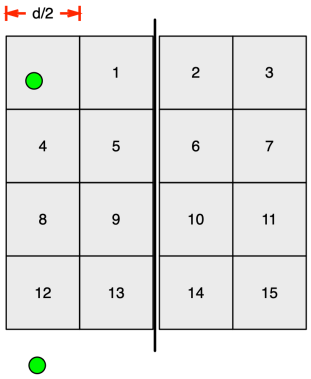
Divide the region up into squares with sides of length $d/2$:



How many points in each box?

At most 1 because each box is completely contained in one half and no two points in a half are closer than d.

Suppose 2 points are separated by > 15 indices.



- Then, at least 3 full rows separate them (the packing shown is the smallest possible).
- But the height of 3 rows is $> 3d/2$, which is $> d$.
- So the two points are father than d apart.

Theorem

Suppose $S_y = p_1, \ldots, p_m$. *If* $dist(p_i, p_j) < d$ *then* $j - i \leq 15$.