Question 2

A simple algorithm of this approach is to use brutal force search which calculates the distance between every pair of points.
Here is an algorithm:

```
# the function that returns the Manhattan distance.
Dist( (a,b), (x,y)) {
Return  abs(a-x)  + abs(b-y)
}


Closest_pair(array){
# Initialize the minimum distance
Min_dis = ∞

For i in range(len(p)){
          For j = i+1 to len(p)-1){
  # calculates the Manhattan distance
            d = Dist(p[i],p[j])
            If d < min_dis:{
            Min_dis = d
            Pair = (p[i], p[j])
}
}
}
Return Pair
}
```

This algorithm compares every pair and only update the closest distance pair. This algorithm uses two for loop to go over the array. As a result, the time complexity for this algorithm is O(n^2). The space complexity is obviously O(n) because we only have to store the original array,

To optimize this algorithm, we can use divide and conquer as well as recursion:

CloestPair(input):
1. If the len(input) <=3:
    Just use the brutal force algorithm I just introduced to compute.

2. We first sort points according to x coordinates. (using merge_sort)
   Then we sort the points again to y coordinates.
   First we construct merge sort:

```
merge_sort(array){
# Divide the array into halves: array1 and array2
# Recursively call merge sort on both sides :
merge_sort(array1)
merge_sort(array2)
# finally merge array1 and array2 together  where merge is a function that combine two arrays in sorted order.
merge(array1,array2)

}
```

The recurrence equation for this algorithm is:
$T(n) = 2(T(n/2)) + n$
As a result the time complexity for merge sort is O(nlogn).

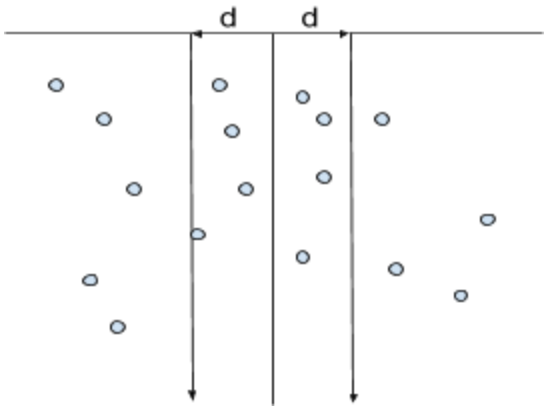
Let the points sorted on x coordinate store in array X_sort.
Let the points sorted on y coordinate store in array Y_sort

3. Divide the sorted array into two halves based X_sort. (a vertical line in the middle)
L = [0…… n/2]
R = [n/2 + 1 …..n-1]
Here,Y_sort is also divided based on the vertical line.
4. Recursively find the smallest distance pairs and distance in the subarrays. Then we find smallest of the two.
This step will yield:
Left Smallest distance : SL, (xl,yl) where xl and yl are the closest distance pairs on the left
Right Smallest distance: SR,(xr,yr) where xl and yl are the closest distance pairs on the right.
We then return d =Min (SL,SR) and record the min pairs.

5. Now, we have the smallest distance pair and their distance from both arrays recorded. We then have to find the smallest distance pair and their distance between SL and SR. This step can be done in O(n) instead of O(n^2).

- The first step is to ignore every points that has a distance >= d from the the division line between left and right subsets on two sides. This is because a point of Manhattan distance >= d from the division line will definitely have a greater distance than d from other points >= from the division line in the other set.

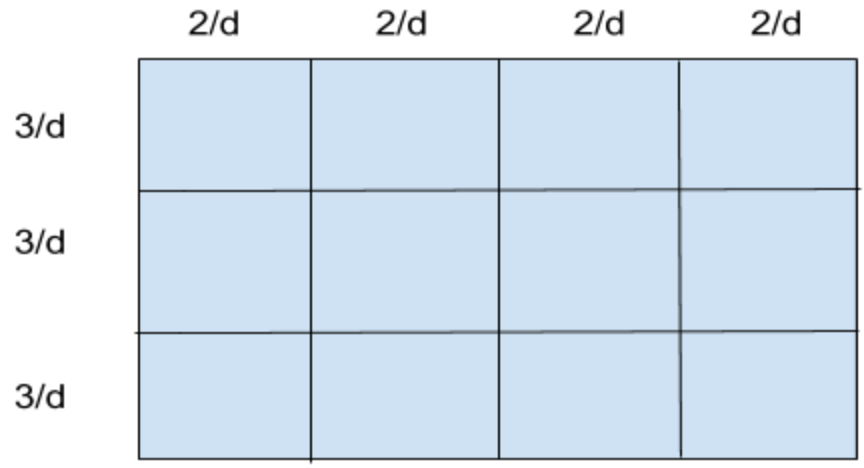
Below is the vision of the graph:



As a result, let array **d_range** be the points within in the d range that is sorted in increasing order of y-coordinates. (we don't have to use sort algorithms here since we already sorted in terms of y axis in the beginning)

As in the graph, we only consider the points within d range to the mid vertical division line.

To improve the running time, we observe that in the following graph:



For any point in **d_range**, we only have to check the area in the above rectangular. The reason is that any points outside of this rectangular box will have a value greater than d and hence can be ignored. One further observation is that no two points on the left side can share the the same square and no two points on the right side can share the same square. The reason is that the manhattan distance of these points is <= d and the maximum manhattan distance within a square is 2/d + 3/d which is less than d. As a result, we only have to check at most 12 points forward for each point in d_range.

- ,As a result, we only have to check the next 12 points for every point. This can be done in O(12n) time which is linear O(n)

6. Finally, let the minimum manhattan distances across two sets and let it be corss_left_right distance and the pairs be (c1,c2). Then we find the min((SL, SR, corss_left_right distance) and return the closest manhattan distance pair according to the distance.

Then we return the points according to the distance.

Running time analysis:

1. O(nlogn) for merge sort based on x coordinates and based on y coordinates
2. Divide: O(logn) : this is because the input can be divided (logn) times.
3. O(n) for find the points within d_range. Another O(n) for dividing Y_sort
4. Conquer: O(n) for find the closest point in the sort y coordinates between two sets.

This improvement can yield a O(nlogn) time complexity because for each logn levels of the tree, only O(n) time is required to find the minimum distance. However, in terms of space of complexity, this approach will require O(2n) instead of O(n), the reason is that we also have to store the array with sorted y coordinates.