Planning Document – Arup Guha Program 3: Contact Tracing

Planning Phase

The assignment specified storing "your location" globally as two separate integer variables. I will store mine as follows.

```
int myx;
int myy;
```

Since critical information about a point is the distance from the point to "your location", I will store a Cartesian point as a struct, with three components:

```
int x;
int y;
int distSq;
```

In main, there will be an array of pointers to point structs, so the type of the main array in main will be as follows:

```
pt** array;
```

where array will be allocated as an array of pointers, and each pointer will point to a single point struct.

The most critical function will be the compareTo function which will have the following signature:

```
int compareTo(pt* ptA, pt* ptB);
```

This function will work like strcmp for strings in returning a negative integer if the pt pointed to by ptA "comes before" the pt pointed to by ptrB, a positive integer if the pt pointed to by ptA "comes after" the pt pointed to by ptB, and 0 if the two points are the same.

The function for the insertion sort will have the following signature:

```
void insertionSort(pt** array, int sI, int eI);
```

This function will sort the sub-array starting at index sI and ending at index eI, inclusive.

The rest of the functions for sorting will be as follows:

```
void sort(pt** array, int length, int threshold);
void mergeSort(pt** array, int sI, int eI, int threshold);
void merge(pt** array, int s1, int s2, int e2);
```

The first function is the wrapper function for the sort. It sorts the whole array of length length utilizing threshold as specified in the assignment. The second is a regular merge sort of the sub-array starting at index sI and ending at index eI, with the specified threshold value. The last function is the standard prototype for the merge function which merges array[s1..s2-1] and array[s2..e2] and then copies the merged array back into array[s1..e2].

Finally, to handle queries, there will be a binary search function. Since we want to practice recursion, this binary search will be recursive and come with a wrapper function. The two prototypes are as follows:

```
int binarySearch(pt** array, int length, pt* searchPt);
int binarySearchRec(pt** array, int sI, int eI, pt* searchPt);
```

The first returns the index that the point pointed to by searchPt is located in the array, which is of length length. If the point isn't in the array, -1 is returned. The second is the recursive function which performs the task for the subarray from sI to eI.

main will read in all the data, store it, and farm out the appropriate work, and print the results as needed.

Assistance Received

I did not receive any assistance, I worked on the program on my own from start to finish.

Debugging Phase

I decided to code up everything to the sort and then test on the sample. When I did that, I got incorrect output. Instead of analyzing the output, I decided to quickly check to see if the insertion sort was working. To do this, I hard coded the following line in my mergeSort function:

```
if (eI-sI+1 <= 100) {
    insertionSort(array, sI, eI);
    return;
}
```

This way, since the sample array was size less than size 100, my merge sort code never runs and I could see if my insertion sort was correct. Sure enough, the insertion sort was correct (this change fixed my output. So, this meant my error had to be in my merge sort.

I changed the code back to putting threshold instead of hard-coding 100, and focused on reading my merge sort code. Almost immediately, as I traced through the code, I noticed that I had merged my two arrays into an array tmp, ***but I never copied the values BACK into array***! So, all I did was add the following code to my merge function:

```
for (int i=s1; i<=e2; i++)
    array[i] = tmp[i-s1];
```

This copied the temporary merged values back into array, which is what I had forgotten to do!

When I put in this change the sample worked.

Then, I moved onto implementing the binary search, after I knew the sort worked.

After typing up the two binary search functions, and then adding the code in main to read the queries. When compiling, I found out that array was an undefined variable in my call to the binary search. In glancing at my code, I realized that the ***formal*** parameter name was array, but in main, I called my array people. I changed this and the code ran, and luckily produced the correct answer on the sample.

Testing Phase

In my testing phase I designed four large extra test cases beyond the sample. I also wrote an alternate Java solution (I don't expect any students to do this), because Java has a built in sorting function that I know works. For each of the four test cases I generated, I made sure that both my Java solution and C solutions matched. The design of my test cases were as follows:

1. One test case of me being at (0, 0) and all infected people being a distance of exactly 10,000 from me. There are 36 such points. (I just wrote a brute force algorithm that tried all possible x values and did math to figure out the corresponding y values, if they existed.) Thus, since all of the points were the same distance away from me, this case ONLY test sorting by x and then y, which is fairly easy to check by hand. The point (-10000, 0) should come first, followed by (-9600, -2800) and then (-9600, 2800). Note, the last two points are part of the primitive Pythagorean Triple 7-24-25. This matched.

2. The next test case was a maximum random case, where I generated 100,000 unique random points and a random point for me as well. The queries were 40% from the set and 60% random.

3. The third case was all concentric circles centered around (0, 0) that were an integer distance away from the center. I queried all of the points in the input once and make the rest of the queries random.

4. For the last case, I took the third case and changed my location to a different random location, but kept everything else the same.

In addition to these large cases, I developed several hand-made cases:

1. One input point and two queries - one for the point and one for a point one off. This just test the minimum sized test case, making sure my code doesn't break for an array of size 1.

2. Me at (10000, 10000) and nine points all by (-10000, -10000) all far away. Query each point in the sample and a couple outside of it. This tests the maximum value of the distance squared function and all possible query points in the set.

3. A small sample with me at (0, 0), and a few points otherwise, with query points between every gap. Make sure to include two extremal points, one closest to me and one farthest from me. Other points to include are those with the same distance from me as a valid point, but not a valid point.

4. A test case with me at (5, 6) and all the points a distance of 5 away from me except for two. Make the queries so that all 10 pts in the same are found and the two not in the sample are not.

Once all of these cases matched on both solutions, I finished my program development.

My four hand made cases are included for convenience. Note: Output isn't included since this is easy enough to work out by hand.

mytest1.in

```
3 5 1 2 1
4 3
3 4
4 3
```

mytest2.in

```
10000 10000 9 16 5
-10000 -10000
-10000 -9999
-10000 -9998
-9999 -10000
-9999 -9999
-9999 -9998
-9998 -10000
-9998 -9999
-9998 -9998
-9997 -9997
-9997 -9998
-9997 -9999
-9997 -10000
-9998 -9997
-9998 -9998
-9998 -9999
-9998 -10000
-9999 -9997
-9999 -9998
-9999 -9999
-9999 -10000
-10000 -9997
-10000 -9998
-10000 -9999
-10000 -10000
```

mytest3.in

```
0 0 3 12 10
1 2
-7 10
13 -19
-10000 -10000
13 -19
13 19
6 22
-7 10
-7 -10
-9 -8
1 2
1 -2
-1 2
-1 -2
0 1
```

mytest4.in

```
5 6 10 12 3
3 4
3 -4
-3 4
-3 -4
4 3
4 -3
-4 3
5 0
-5 0
0 -5
3 4
3 -4
-3 4
0 5
-3 -4
4 3
-4 -3
4 -3
-4 3
5 0
-5 0
0 -5
```