CS 130B
DATA STRUCTURES AND ALGORITHMS
Programming Assignment II
INDIVIDUAL WORK ON THIS ASSIGNMENT IS MANDATORY
Due Date: Friday March 6 at 3:00 PM
Remember: Points will be deducted for homework turned in after the Due Date.
Deadline: Monday March 9 at 8:00 AM
Remember: Projects will not be accepted after the Deadline
Preliminary Version

1 Introduction

Remember that you must work on this assignment INDIVIDUALLY. It is very important that you work hard on this assignment because the second programming assignment may build on this one.

For this assignment you will code in C++ several algorithms to provide several different solutions to the problem of finding *energy-efficient broadcasting tree in a static wireless network*. Given a set of n stations (objects) in 2D with a special station called s, the goal is find an energy-efficient schedule to broadcast from s to all the stations allowing re-transmission, i.e., once a station receives a message it may re-transmit it. There are many related problems and applications, but for brevity we do not elaborate on these more complex problems. The purpose of this assignment is to see how greedy methods and divide-and-conquer algorithms can be used to provide solutions to this problem. *The idea is for you to practice C++ and learn more about it, rather than to make you an expert in C++ and to experiment to evaluate the performance of the different algorithms.*

We have as input a set of *objects (or points or stations)* in 2D with a special station being called the *source station* s. The i^{th} object or station (for $i \geq 1$) is called station O_i and it is located at (x_i, y_i), where x_i and y_i are integers. The index of O_i is i. The index of a station is assigned when the program reads in its location and it never changes. It is assumed that no two stations are located at the same position in 2D, i.e., for $i \neq j$, either $x_i \neq x_j$ or $y_i \neq y_j$. The distance between station O_i located at (x_i, y_i) and station O_j located at (x_j, y_j) is defined as $d(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. When station O_i transmits over a wireless channel it needs at least a power level equal to $d(i,j)^2$ in order to reach station O_j. Station O_j will not be reached if the transmission is at a lower power. When station O_i transmits at a power r^2 then the transmission will be received by all the stations located at a distance at most r from O_i.

Let us now consider Example 1. There are 10 stations $\{O_1, O_2, O_3, O_4, O_5, O_6, O_7, O_8, O_9, O_{10}\}$ and the source station is O_{10}, i.e., $s = 10$. The location of the stations is given in Table 1.

Table 1: Station Locations for Example 1.

O	1	2	3	4	5	6	7	8	9	10
x	-4	3	-2	-2	7	5	0	0	0	0
y	0	4	4	6	0	0	-4	-7	-8	0

In Figure 1 we give two possible ways of broadcasting information originating at O_{10} to all stations. We call these figures *transmissions maps*. The transmission map given in Figure 1(b) indicates that O_{10}

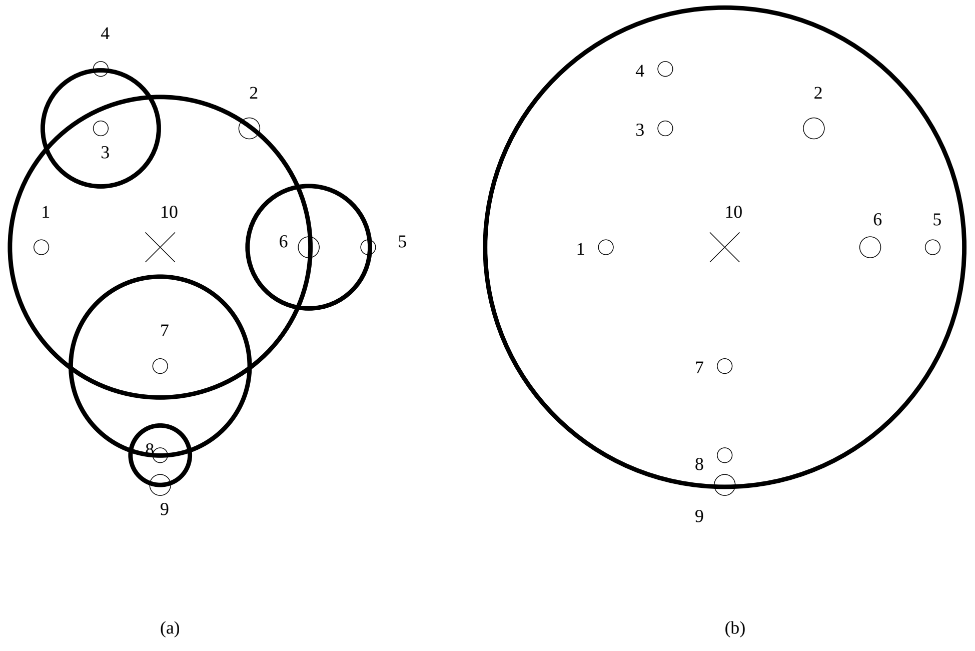

Figure 1: Two Transmissions Maps for Example 1.

transmits to everyone using power equal to $8^2 = 64$. The objective function value (or total power utilized) is 64 in this case. The transmission map given in Figure 1(a) has station O_{10} transmitting with a power equal to $5^2 = 25$ and it reaches stations $\{O_1, O_2, O_3, O_6, O_7\}$. Then the message is re-transmitted from O_3 with power $2^2 = 4$ and it reaches O_4; it is also re-transmitted from O_6 with power $2^2 = 4$ and it reaches O_5; and also re-transmitted from O_7 with power $3^2 = 9$ and it reaches O_8. The information is re-transmitted from O_8 with a power $1^1 = 1$ and reaches O_9. The total power (objective function value) is $25 + 4 + 4 + 9 + 1 = 43$. Therefore, information is transmitted to all stations in Figure 1(a) using less power than in Figure 1(b). Is the transmission map given in Figure 1(a) optimal? I.e., does it use the least amount of power to broadcast from station O_{10} to all the other stations? I think so, but I do not have a proof for this. The following table gives the *transmissions rate* for each station using the transmission maps given in Figure 1(a) and (b). We call the array r the *transmission rate array*. The transmission rate array r just indicates for each station O_i how far its transmissions can reach. The power used by station O_i is r_i^2.

Table 2: Transmission rates for the transmission maps in Figure 1.

r	1	2	3	4	5	6	7	8	9	10
Trans. Map Fig 1(a)	0	0	2	0	0	2	3	1	0	5
Trans. Map Fig 1(b)	0	0	0	0	0	0	0	0	0	8

Let S be a subset of the stations that includes station s. For the moment think as S being all the stations. A transmission rate array is said to be a *transmission schedule for S* if procedure `Tree`(r) given below constructs from the transmission rate array a corresponding *transmission tree* that includes a vertex for each station in S. Procedure `Tree`(r, S) is given below.

```
Procedure Tree ($r$, $S$);
// The marks we use here are different from the marks used by other procedures//
The root of the tree is vertex $s$;
unmark all the stations in $S$;
add vertex $s$ to the empty queue $Q$; // Queue is FIFO//
mark station $O_s$;
while $Q$ is not empty do
    $i \leftarrow$ delete-from-queue $Q$;
    Let $L$ be the set of all unmarked stations in $S$ inside the circle centered at $O_i$ with radius $r_i$;
    while $L$ is not empty do
        $j \leftarrow$ index of the smallest indexed station in $L$;
        delete $j$ from $L$;
        add $j$ as a child of $i$;
        mark station $O_j$;
        add $j$ to queue $Q$;
    endwhile
endwhile
End Procedure
```

Procedure `Tree`(r, S) generates a transmission tree that includes all stations in S for each of the transmission rate arrays given in Table 1 (corresponding to the transmission maps in Figure 1(a) and (b)). The transmission trees for the transmission rate arrays in Table 2 are given in Figure 2. As we said before, a transmission rate array r is said to be a *transmission schedule* if procedure `Tree`(r, S) constructs a transmission tree that includes a vertex for each station in S. The *lowest power* for the transmission schedule r is $\sum t[i]^2$, where $t[i]$ is 0 if vertex i does not have any children in the tree generated by procedure `Tree`(r, S) and it is equal to $max\{d(i,j)|j$ is a child of i in the tree constructed by procedure `Tree`(r, S)$\}$. The array $t[]$ is called the *lowest transmission schedule* for the transmission rate array r for S. Note that $\sum t[i]^2 \leq \sum r[i]^2$, and equality does not always hold.

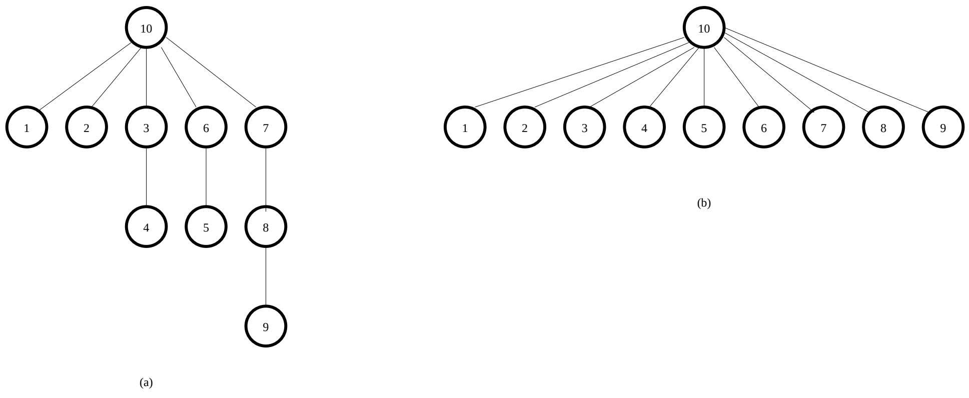

Figure 2: Transmission Trees for the Transmission maps in Figure 1.

Suppose that in the transmission map in Figure 1(a) the transmissions out of O_6 and O_7 had more power and their transmission area overlaps. Furthermore, suppose that there was a station, say O_{11}, inside both of these transmission areas as well as inside the transmission area originating out of O_{10}. Then the corresponding tree would have O_{11} as a child of O_{10}. On the other hand, if O_{11} was inside the transmission area of only O_6 and O_7, then it would be a child of O_6 in the corresponding transmission tree.

Given a set of n stations located at the positions indicated by the arrays x and y with the source station being s, the problem is to find a transmissions schedule r with minimum power. This problem is computationally intractable and currently there is no known efficient algorithm for its solution. In this assignment we will code efficient algorithms that generate suboptimal solutions to this problem, i.e., it generates a transmission schedule whose total power is not necessarily the minimum power.

Given a problem instance your program will select the appropriate algorithm to use depending on an input value and will generate a transmission schedule to broadcast from O_s. Then the algorithm will print out some relevant information. In the next section we explain the different algorithms that you must implement. Then we give the format for the input, the internal representation, explain the required output, talk about coding and experimentation, and point out the turnin procedure for the project.

The following algorithms use the above definitions when set S includes only a subset of the stations, but always including station S.

2 The Methods

In this section we explain the greedy methods and the divide-and-conquer algorithm you will be using to solve our broadcasting problem.

2.1 Three Greedy Methods

There are three greedy methods: Add the closest to O_s first (Add-Closest to Source First); Add the closest to the ones that currently have received the transmission (Add-Closest to Set First); and Add the cheapest one that can be reached first (Add-Cheapest First).

2.1.1 Add-Closest to Source First (ACSF)

```
global $O, n, s, r$
ACSF();
// The marks we use here are different from the marks used in other procedures//
$r[i] \leftarrow 0$ for $1 \leq i \leq n$;
Unmark all the stations;
Mark station $O_s$;
For i=2 to n do // Add a station.
      Let $j$ be the smallest integer such that station $O_j$ is unmarked and
          $d(s,j) \leq d(s,k)$ for every station $O_k$ that is unmarked;
      Mark station $O_j$; // Station $O_j$ is the station to be added.
      $r[j] \leftarrow 0$;
      For each marked station $O_l$ such that $l \neq j$ do
        //If we transmit from from $O_l$ to $O_j$, find the lowest power schedule for $S$,
        //and store the lowest power in $p[l]$.
          $q[k] \leftarrow r[k]$ for $1 \leq k \leq n$;
          $q[l] \leftarrow max\{q[l], d(l,j)\}$;
          Let $T$ be the tree generated by $Tree(q,S)$, where $S$ is the current set of marked stations;
          $p[l] \leftarrow$ lowest power for $T$;
      endfor
        // Select the best station to transmit to $O_j$.
      Let $l$ be the smallest integer such that $l \neq j$, $O_l$ is a marked station
          and $p[l] \leq p[k]$ for every $O_k$ that is marked;
      $r[l] \leftarrow max\{r[l], d(l,j)\}$;
      Let $T$ be the tree generated by procedure $Tree(r,S)$, where $S$ is the current
          set of marked stations
      Let $t[l]$, for every marked station $O_l$, be the lowest transmission rates for
          the tree generated by $Tree(r,S)$
      // Update array $r$ with lowest powered schedule for $S$.
      $r[l] \leftarrow t[l]$ for every marked station $O_l$;
  endfor
end ACSF
```

2.1.2 Add-Closest to Set First (ACSet)

```
global $O, n, s, r$
ACSet();
// The marks we use here are different from the marks used in other procedures//
$r[i] \leftarrow 0$ for $1 \leq i \leq n$;
Unmark all the stations;
Mark station $O_s$;
For i=2 to n do // Add a station.
    Let $j$ be the smallest integer such that station $O_j$ is unmarked, $O_e$ is marked and
        $d(e,j) \leq d(f,k)$ for every station $O_k$ that is unmarked and every station $O_f$ that is marked;
    Mark station $O_j$;// Station $O_j$ is the station to be added.
    $r[j] \leftarrow 0$;
    For each marked station $O_l$ such that $l \neq j$ do
      //If we transmit from from $O_l$ to $O_j$, find the lowest power schedule for $S$,
      //and store the lowest power in $p[l]$.
        $q[k] \leftarrow r[k]$ for $1 \leq k \leq n$;
        $q[l] \leftarrow max\{q[l], d(l,j)\}$;
        Let $T$ be the tree generated by $Tree(q,S)$, where $S$ is the current set of marked stations;
        $p[l] \leftarrow$ power of lowest power for $T$;
    endfor
      // Select the best station to transmit to $O_j$.
    Let $l$ be the smallest integer such that $l \neq j$, $O_l$ is a marked station
        and $p[l] \leq p[k]$ for every $O_k$ that is marked;
    $r[l] \leftarrow max\{r[l], d(l,j)\}$;
    Let $T$ be the tree generated by procedure $Tree(r,S)$, where $S$ is the current
        set of marked stations
    Let $t[l]$, for every marked station $O_l$, be the lowest transmission rates for
        the tree generated by $Tree(r,S)$
    // Update array $r$ with lowest powered schedule for $S$.
    $r[l] \leftarrow t[l]$ for every marked station $O_l$;
  endfor
end ACSF
```

2.1.3 Add-Cheapest First (ACF)

```
global $O, n, s, r$
ACF();
// The marks we use here are different from the marks used in other procedures//
$r[i] \leftarrow 0$ for $1 \leq i \leq n$;
Unmark all the stations;
Mark station $O_s$;
For i=2 to n do // Add a station.
   For each unmarked station $O_j$ do
      Mark station $O_j$; // Consider station $O_j$ as the station to be added.
      $r[j] \leftarrow 0$;
      For each marked station $O_l$ do
    //If we transmit from from $O_l$ to $O_j$, find the lowest power schedule for $S$,
    //and store the lowest power in $p[j, l]$.
         $q[k] \leftarrow r[k]$ for $1 \leq k \leq n$;
         $q[l] \leftarrow max\{q[l], d(l, j)\}$;
         Let $T$ be the tree generated by $Tree(q, S)$ where $S$ is the current set of marked stations;
         $p[j, l] \leftarrow$ lowest power for $T$;
      endfor
      Unmark $O_j$;
   endfor
    // Select station to transmit to $O_j$ and let $O_l$ be the station that will transmit to $O_j$.
   Let $j$ be the smallest integer such that $O_j$ is an unmarked station and there is a marked station
      $O_l$ for which $p[j, l] \leq p[k, e]$ for every $O_k$ that is unmarked and every $O_e$ that is marked;
   Let $l$ be the smallest integer such that $O_l$ is a marked station
      and $p[j, l] \leq p[j, k]$ for every $O_k$ that is marked;
   Mark $O_j$;
   $r[l] \leftarrow max\{r[l], d(l, j)\}$;
   Let $T$ be the tree generated by procedure $Tree(r, S)$, where $S$ is the current
      set of marked stations
   Let $t[l]$, for every marked station $O_l$, be the lowest transmission rates for
      the tree generated by $Tree(r, S)$
   // Update array $r$ with lowest powered schedule for $S$.
   $r[l] \leftarrow t[l]$ for every $l$ that is a marked station;
 endfor
end ACF
```

2.2 Divide-and-Conquer

There is only one divide and conquer method.

2.2.1 Divide-and-Conquer (DC)

```
global $O, n, s, r$
DC();
// The marks we use here are different from the marks used in other procedures//
Let $\pi_1, \pi_2, \ldots, \pi_n\}$ be a permutation of $\{1, 2, \ldots, n\}$ such that $\pi_1 = s$ and
    $d(\pi_1, \pi_i) \leq d(\pi_1, \pi_{i+1})$ for $2 \leq i \leq n-1$ and if $d(\pi_1, \pi_i) = d(\pi_1, \pi_{i+1})$ then $\pi_i < \pi_{i+1}$;
Mark stations $O_{\pi_1}, O_{\pi_2}, \ldots, O_{\pi_{\lfloor n/4 \rfloor}}$;
Apply procedure ACF to the marked stations only;
Mark stations $O_{\pi_{\lfloor n/4 \rfloor + 1}}, \ldots O_{\pi_{2\lfloor n/4 \rfloor}}$;
Now continue applying ACF to augment the previous solution with the newly marked stations;
Mark stations $O_{\pi_{2\lfloor n/4 \rfloor + 1}}, \ldots O_{\pi_{3\lfloor n/4 \rfloor}}$;
Now continue applying ACF to augment the previous solution with the newly marked stations;
Mark stations $O_{\pi_{3\lfloor n/4 \rfloor + 1}}, \ldots O_{\pi_n}$;
Now continue applying ACF to augment the previous solution with the newly marked stations;
end DC
```

2.3 Your Own Method

In here you will implement your own method. The only restriction is that it should normally (not always, but on average) be faster than the sum of the times of the three greedy methods.

2.4 Dynamic Programming

In this section we explain the dynamic programming algorithms for our problem when all the points are located along the x-axis with the leftmost point being the source station. In this case the input value s will be called k and its value will be $1 \leq k \leq n$ and indicated the maximum number of stations that can broadcast.

We will refer to the n stations after being sorted from left to right as $P_1, P_2, \ldots, P_n$. But you need to remember their initial names (i.e., their initial index) for the purpose of printing the output. The reason for this is that the points are not provided in sorted order initially.

Let i and j be integers such that $1 \leq i < j \leq n$. When P_i transmits all the way to P_j (and not beyond P_j) it uses power $(P_j - P_i)^2$. Note that if P_i just transmits to P_i, then the contribution to the modified objective function value for this case is zero. The limitation is that at most k stations can be transmitters.

Our algorithm to find the optimal transmission rate begins with the computation of $VAL(1, n, k)$ as the total power of the best possible broadcast tree which can be constructed for $P_1, P_2, \ldots, P_n$ when P_1 and $k-1$ other stations broadcast to the rest of the stations. We define $VAL(i, j, m)$, for $1 \leq i \leq j \leq n$ and $0 \leq m \leq k$, as the objective function value of the least power needed to broadcast from P_i to $P_{i+1}, \ldots, P_j$ using no more than m transmitters. Note that we use $d(P_i, P_j)$ to mean $d(i, j)$.

$$VAL(i,i,m) = 0 \text{ for } 1 \le i \le n \text{ and } 0 \le m \le k,$$
$$VAL(i,j,0) = \infty \text{ for } 1 \le i < j \le n.$$
$$VAL(i,j,1) = d(P_i,P_j)^2 \text{ for } 1 \le i < j \le n.$$

$VAL(i,j,m)$ is defined as follows for all $1 \le i < j \le n$ and $2 \le m \le k$:

$$VAL(i,j,m) = min_{i<p\le j \textbf{ and } 1\le q<m}\{VAL(i,p,q) + VAL(p,j,m-q)\}$$

To find an optimal modified transmission rate we define $keyP(i,j,m)$ as the smallest value for p that makes $\{VAL(i,p,q) + VAL(p,j,m-q)\}$ a minimum, and define $keyQ(i,j,m)$ as the smallest value of q that makes $\{VAL(i,keyP(i,j,m),q) + VAL(keyP(i,j,m),j,m-q)\}$.

In other words $keyP(i,j,m)$ and $keyQ(i,j,m)$ are such that

$$VAL(i,j,m) = VAL(i,keyP(i,j,m),keyQ(i,j,m)) + VAL(keyP(i,j,m),j,m-keyQ(i,j,m));$$

$$VAL(i,j,m) < \{VAL(i,p,q) + VAL(p,j,m-q) \text{ for all } i \le p < keyP(i,j,m) \text{ and } 1 \le q < m;$$

$$VAL(i,j,m) < \{VAL(i,keyP(i,j,m),q) + VAL(keyP(i,j,m),j,m-q) \text{ for all } 1 \le q < keyQ(i,j,m)$$

Note that $key(i,j,m)$ is a 3-dimensional array, which for this part you are allowed to use, but it needs to be allocated dynamically. In this case DO NOT invoke the `Tree()` procedure. You can recover the tree from $keyP(i,j,m)$ and $keyQ(i,j,m)$. The total power is $VAL(i,j,k)$ and the individual power for the stations from the tree constructed from $keyP(i,j,m)$ and $keyQ(i,j,m)$.
The above computation process (of VAL) should be implemented recursively. First with recomputation of the $VAL(i,j,m)$ values, and then without recomputation by using a 3D array, just as we have done in class. Note that my definitions can be reduced to use only a 2D array. This will be faster and more efficient. Your code should be efficient.

The third implementation is to compute VAL iteratively, and saving each computed result in a table (3-dimensional array, or if you exploit some nice properties you may use a 2-dimensional array).

3 Input

The first input line contains three positive integer values whose meaning is:

- *Method*: integer in (1,2,3,4,5,6,7,8), where

  1: use ACSF

  2: use ACSet

  3: use ACF

  4: use DC

  5: use YourOwnMethod

  6 DP recursive with recomputation

7 DP recursive without recomputation

8 Iterative.

- n: number of stations (assume $n \geq 10$).
- s: Source station (assume $1 \leq s \leq n$). (note that for the DP procedures the source will be the leftmost point and k will be the value of s that the program reads in.

The next n input lines define the stations. The i^{th} line ($1 \leq i \leq n$) defines station O_i. Each line contains two integers, the first is the x-coordinate value and the second is the y-coordinate value of the object. The station has index i and it will not change throughout the execution of the program. Note that for the DP cases you will read the same information, but will only use the x-coordinate values.

~~After this input you will find a line with the positive integer m and then a list of m input lines. The ith line contains the integer $z(i)$, such that $1 \leq z(i) \leq n$. This information will be used for printing.~~

You may assume that the input is correct, but to play it safe you may check that it is a valid input. Note that all your structures must by allocated "DYNAMICALLY" so that the algorithm would work no matter what the input size is provided there is enough space. You may NOT use the type `vector`, it is too slow for this application. Read the input with `cin` from the standard input. A sample input file will be added to the Web page later on.

4 Internal Representation

You may use any of the code in the text which is available in the cs130b directory. The overall requirement is that all your structures must by allocated "DYNAMICALLY" so that the algorithm works no matter what the input size is provided there is enough space. Do not use `vector` **or STL functions**, they are too slow for this application. Read the input with `cin`. Also, use the "standard output" for writing your output and "standard input" for reading. When we test your program we will run your program as "`a.out < input.1 > output.1`", with several different input files we will be creating.

5 Output

~~For methods 1–4 your output should print the nonzero entries in the lowest transmission schedule (r array) as pairs i and $r(i)$ (one per line). Then for $i = 1, 2, ..., m$ your program should print i and $z(i)$ in one line, followed by a line for each child of $z(i)$ in the final transmission tree. The children (index) will be printed in increasing order, one per line.~~ **For ALL METHODS you will print the following information. The first line of output will have the total power used by the final lowest transmission schedule. Then you will have an output line for each nonzero entry in the lowest transmission schedule (r array). Each such line will have three values of the form i, $r(i)^2$, and index of the parent station in the tree final lowest transmission schedule, (one per line in increasing order of the station index). Note that $r(i)^2$ (which is the transmission power of station O_i) should be printed rounded to the closest `long long int`. Also note that the predecessor of the root is -1.**

~~Then you will print the lowest power for your solution, i.e., the objective function value.~~

6 Coding and Experimentation

Once your programs are working correctly, time it with the Linux `time` command (type `man time` for an explanation of the `time` command), and run the program with several INPUT FILES which you can create and YOU CAN SHARE WITH OTHER STUDENTS. When you turnin your program you must also turn in your test files. Try test files with n = 10, 30, 80, 500, 5000 as well as other values. The files may be created at random. Plot the actual running time (user time (the number followed by the letter "u")) for your algorithms for the different input files against the number of stations for the different methods. **Compare Your Own Algorithm with the other ones. How good is it?** Note that there are many different experiments you can design. Just try a set of experiments that will provide some picture that is some sense is a "fair comparison". Try to determine experimentally which of the methods is fastest and compare the solutions generated (with respect to the objective function value, i.e., the total power for your solution) by the different methods. Make some recommendations as to when it would be more appropriate to use these methods. Note that this is a general recommendation based on your experimentation. There are many ways one can make the comparisons. Choose any way and apply it consistently. The program must be in C++, you MUST use `makefiles`, and you MUST turn in electronically your code. Note that your code does not have to be C++ object oriented masterpiece. To grade it, we will save it in a directory and type "`make`", followed by "`a.out < data.1 > output.1`", "`a.out < data.2 > output.2 `", ..., where "data.1", "data.2", ... are files we will be creating, and the answers you program computes is stored in output.1, output.2, etc. After everyone turns in their project we will make these files available to you. Your program MUST work in the CSIL PCs running Linux. Note that your grade depends on how well you program works on the examples we will use to test your program. Some of these examples will have quite a few points.

Store the x and y arrays in `int` arrays. But store the array r as a `long long int`. **(Note that you can also store it as *long double*, but see the section entitled BIG HINT.)** Do the same with the total power as well as other data that needs it. Note that computing the square root takes quite a bit of time. So your implementations can compute the same answers WITHOUT computing any square roots. Note that your implementations MUST compute the same answers as the ones with my algorithms, but your implementations can cut corners and implement operations more efficiently. But as I said before, the answers you print must be identical to the ones my algorithms would print.

Implement your procedures as efficient (fast) as possible, but don't go too far on this. Try to use a small amount of additional space, but do not try to reduce it to to the absolute minimum. `You may NOT use any two dimensional arrays of size` $\Omega(n)$ `by` $\Omega(n)$ `for Methods 1 -5, but for Methods 6 - 8 you may use 3D arrays that must be allocated dynamically.` Note that you do not need store the array $p[][]$ in ACF. You just need to remember the smallest value. The two dimensional array $d(i, j)$ should be recomputed every time. You may use any "constant" (constant independent of n) number of one dimensional arrays of size $O(n)$. Your member functions must perform logical sub-functions. Your code must be readable and there should be appropriate comments all over the code describing its behavior. You may use any of the procedures in the book. Look in the cs130b Web page for the location where you can find the book programs.

Partial credit will be given if not all of the code is working correctly. The partial credit will depend on how well your code works on the test files. You will be deducted points if your program takes too long. YOU MUST WORK ON THIS PROJECT INDIVIDUALLY.

7 BIG HINT

Note that in this project we use distances between points in 2D at integer coordinate values. To compute the distance between two points you need to do a square-root computation. Performing many square roots will slow your program and the result is not exact. For example square root of 2, or 3 or 5, or 10, etc. cannot be represented exactly as a `long double` in C++.

When we transmit with power d^2 we will reach all points at a distance d from the transmitter. So in array r, as specified before, we store how far a transmission will reach. In other words $r[i]$ tells us that we can transmit to all points at a distance r[i] from station O_i. But we said that array r has to be a `long long int`. Now, distances are real numbers so how can this work? One possibility is to store array r as a `long double` and that is fine. You I will allow this, but it will not be efficient because of the computation of square root, which has to be done many times. Also you have to be careful because the result of the computation of square root cannot always be represented exactly in a `long double`, e.g., square root of 2, or 3 or 5, or 10, etc.

I am looking for an efficient implementation. The best thing is to avoid computing so many square roots. So you can work with the square of the distances and store in array r the square of the distances (or power). This way you can define array r as `long long int`, as the square of the distances between points at integer coordinates is an integer. But if you do this you have to modify parts of the algorithm, i.e., instead of "all stations inside the circle centered at O_i with radius r_i", it should be "all stations at a square distance of at most r_i from station O_i". So in array r you will store the square of a distance (or power) instead of a distance. This will always be an integer since all points are stored at integer coordinates.

8 TURNIN

Read the Web page for cs130b for instructions on how to turnin electronically your code.

GOOD LUCK