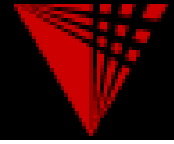


ILLINOIS INSTITUTE OF TECHNOLOGY

CS430 Project

Introduction Algorithms

Student name : GyuHyeon Choi
Student ID : A20316532

1. Problem Description

Consider a straight line L in the plane.
A finite set T of targets are located above the line L ,and a finite set S of wireless sensors are located below the line L .
A sensor s can monitor a target t if and only if the Euclidean distance between s and t is at most one.
Suppose that each sensor $s \in S$ has a positive cost $c(s)$ and each target $t \in T$ can be monitored by at least one sensor in S .
Consider a subset S' of sensors in S . S' Is said to be a cover if each target in T is covered by at least one sensor in S' .
The cost of S' is the total costs of the sensors in S' .
The objective is to compute a cover S' of minimum cost.
Please develop a polynomial time algorithm and write a program to implement it.

2. Algorithm design

I chose to represent the relation between each target and each sensor by using the array $M[i][j]$ because I thought it is more intuitive to determine which sensors cover each target before solving the problem.

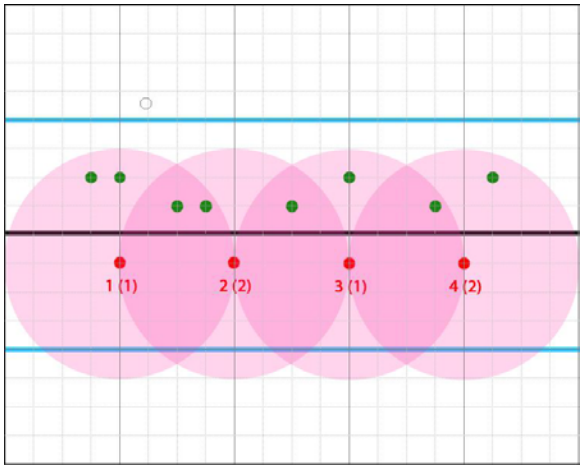
The parameter i and j denote indices of a sensor(column) and a target(row). So an index of a row corresponds to an index of each sensor and an index of a column corresponds to an index of each target. For example, $M[2][8]$ means that the relation between Sensor[2] and Target[8].

The ways to build up $M[i][j]$ is bellow:

- All elements $M[0][j]$ and $M[i][0]$, which means the first row and column, are 0. This is for the boundary condition.

- If Target[j] is covered by Sensor[i], $M[i][j]$ is $M[i][j-1]$ + 1.

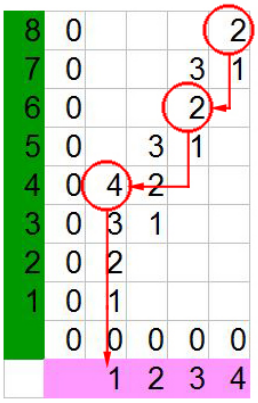
- Otherwise, $M[i][j]$ is 0.

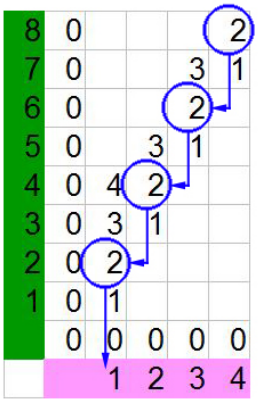


By using rules above, we can build the following array from the previous picture. Before we build up the array, we should align both targets and sensors in increasing order of x-coordinate.

8	0				2
7	0			3	1
6	0			2	
5	0		3	1	
4	0	4	2		
3	0	3	1		
2	0	2			
1	0	1			
	0	0	0	0	0
		1	2	3	4

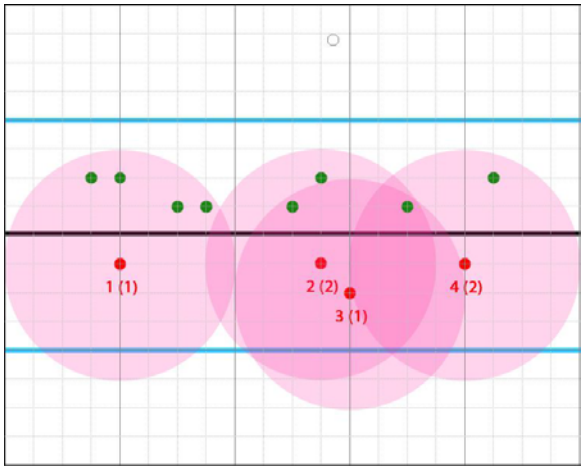
Now, we can trace the subset S'' which covers all targets by exploring the array $M[i][j]$ by following the arrows described in the next two pictures.



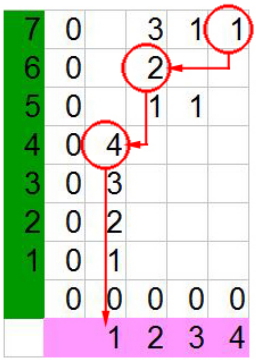


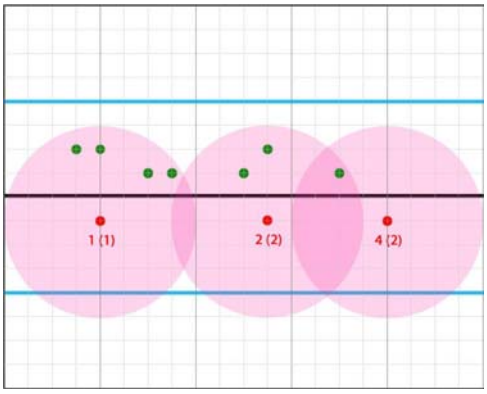
The left picture shows that we have to choose $S''=\{1,3,4\}$ and the picture on the right side show that we have to choose $S''=\{1,2,3,4\}$.

It works. An element 0 means that the sensor cannot cover the target. So once the exploring process encounter the 0 element while it is going down through the column, it finds another sensor which covers the target in another column and the same row.

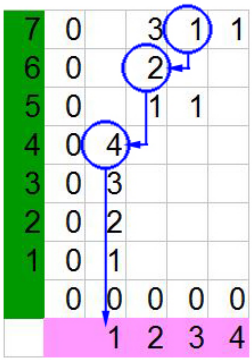


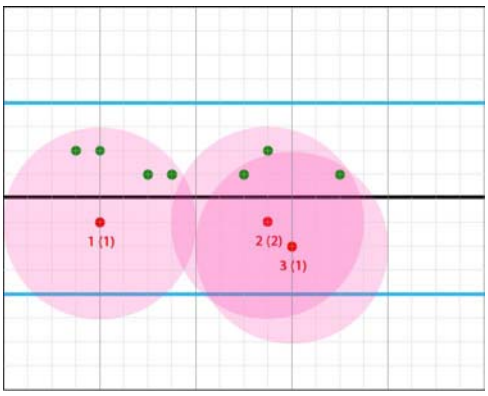
It is slightly more difficult in the previous case because Target[5] is covered by both Sensor[2] and Sensor[3] but Target[6] is only covered by Sensor[2]. However, it still works. The arrays and corresponding cover sets are shown below.



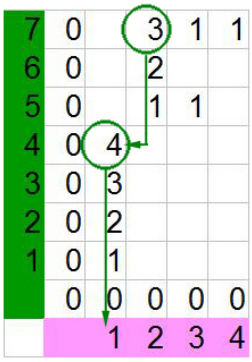


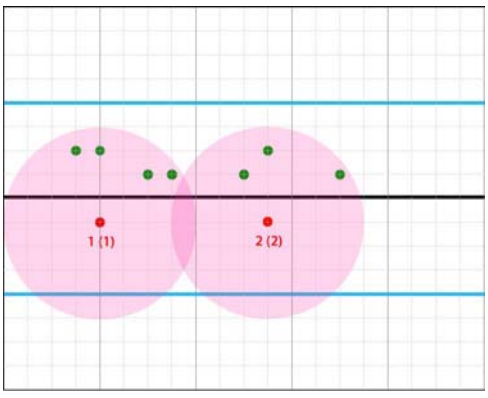
$S'' = \{1, 2, 4\}$





$S'' = \{1, 2, 3\}$





$S'' = \{1, 2\}$

If there is only one element which is not 0 in a row like Row[6], the exploring process coming down from other columns must stop at the row because all the other elements in a row are 0. The exploring process seeks for another sensor which covers the target once it encounters the element 0 because an element 0 means that the sensor cannot cover the target.

The important part is Column[3] whose Row[5] and Row[7] is 1 but Row[6] is 0. Since Target[6] is only covered by Sensor[2], although Sensor[3] can cover two targets, the value of $M[3][6]$ is reset to 0 following the rule, if Target[j] is covered by Sensor[i], $M[i][j]$ is $M[i][j-1]$ + 1. Otherwise, $M[i][j]$ is 0.

Thus, the meaning of $M[i][j]$ is how many consecutive targets Sensor[i] can cover alone from Target[j] to Target[1]. In other words, The number of targets from the last target, which is covered by Sensor[i], the sensor covers. As an example, although a certain sensor Sensor[i] can cover set of targets{j, j-1, j-3, j-4}, $M[i][j]$ is 2 because Sensor[i] cannot cover Target[j-2].

Now, we can derive the important formula which implies the regularity described so far.

$$coverEntry(j) = \{i\} + coverEntry(j - M[i][j])$$

Let braces mean that add the value inside to the list. Then, let's see how it works with the first case we used.

8	0				2
7	0			3	1
6	0			2	
5	0		3	1	
4	0	4	2		
3	0	3	1		
2	0	2			
1	0	1			
	0	0	0	0	0
		1	2	3	4

$coverEntry(8) = \{4\} + coverEntry(6) \rightarrow \{4\}$

$coverEntry(6) = \{3\} + coverEntry(4) \rightarrow \{3, 4\}$ // $coverEntry(4)$ has two subproblems

$coverEntry(4) = \{2\} + coverEntry(2) \rightarrow \{2, 3, 4\}$

$coverEntry(2) = \{1\} + coverEntry(0) \rightarrow \{1, 2, 3, 4\}$

$coverEntry(4) = \{1\} + coverEntry(0) \rightarrow \{1, 3, 4\}$

About the formula again,

$$coverEntry(j) = \{i\} + coverEntry(j - M[i][j])$$

$M[i][j]$ is the number of last targets that are covered by Sensor[i] consecutively. Then $j - M[i][j]$ automatically means the number of the targets remaining except of which are covered by Sensor[i] consecutively from the last target. Therefore, we can recursively reduce the target from the last.

We can finally notice that $coverEntry(j)$ find cover sets which cover all the targets. Once more, we can get all cover sets that cover all targets from Target[1] to Target[j] by using $coverEntry(j)$. If we can have all cover sets which cover all targets, it is quite possible to compute the cost of each cover set. Like:

$$entryCost(j) = c(i) + entryCost(j - M[i][j])$$

To find the cover sets of the minimum cost, we can easily choose the cover set of the minimum cost at each step.

$$OPT'(j) = \min_{1 \le i \le m} (c(i) + OPT'(j - M[i][j]))$$ (m is the number of sensors)

However, it is still a recursive form because the parameter of $OPT'(j)$ is reducing and it goes down to its subproblems. But because it can be solved easily from its subproblems, we now satisfy all conditions for dynamic programming.

Firstly, there are only a polynomial number of subproblems which is $i \times j$ where i is the number of sensors and j is the number of targets.

Secondly, the solution to the original problem can be easily computed from the solutions to the subproblems. The formula right above shows this.

Thirdly, there is a natural ordering on subproblems from smallest to largest, together with an easy-to-compute recurrence that allows us to determine the solution to a subproblem from the solutions to some number of smaller subproblems. There are at most $i-1$ subproblems and we can pick the minimum solution among them.

In conclusion, we can define the algorithm of dynamic programming.

Let $OPT(j-1)$ denote the optimal solution, the minimum cost among cover sets which cover all targets from Target[1] to Target[j-1]. When we deal with Target[j], there exist two distinct cases. It can be already covered by $OPT(j-1)$, or it cannot be covered by $OPT(j-1)$. In the first case, we are already done. In the second case, let a certain Sensor[i] cover Target[j]. But the very important part is that Sensor[i] can possibly cover targets more than just Target[j]. At the moment, Target[j] is the last target, so Sensor[i] can cover $M[i][j]$ number of targets from Target[j] consecutively. That is, $OPT(j - M[i][j])$ plus the cost of Sensor[i] is the new optimal solution for targets {1, … , j}.

If $Target[j] \in OPT(j-1) \rightarrow OPT(j) = OPT(j-1)$, Otherwise,

$$Target[j] \notin OPT(j-1) \rightarrow OPT(j) = c(i) + \min_{1 \le i \le m} OPT(j - M[i][j])$$ (m is the number of sensors)

These can be represented as bellow.

$$OPT(j) = min(OPT(j-1), \min_{1 \le i \le m} (c(i) + OPT(j - M[i][j])))$$ (m is the number of sensors)

However, $OPT(j-1)$ also comes from $\min_{1 \le i \le m} (c(i) + OPT((j-1) - M[i][j]))$. For example, at the very beginning, $OPT(1) = \min_{1 \le i \le m} (c(i) + OPT(0))$. Conclusively, the last formula can be written more simply.

$$OPT(j) = \min_{1 \le i \le m} (c(i) + OPT(j - M[i][j]))$$ (m is the number of sensors)

This is the formula we have been looking for.

3. Pseudo code

There are n targets and m sensors. Once we get the lists of targets and sensors, we first align them in increasing order of x-coordinate. Each target and sensor is indexed in the same order. And c[i] contains the cost of Sensor[i].

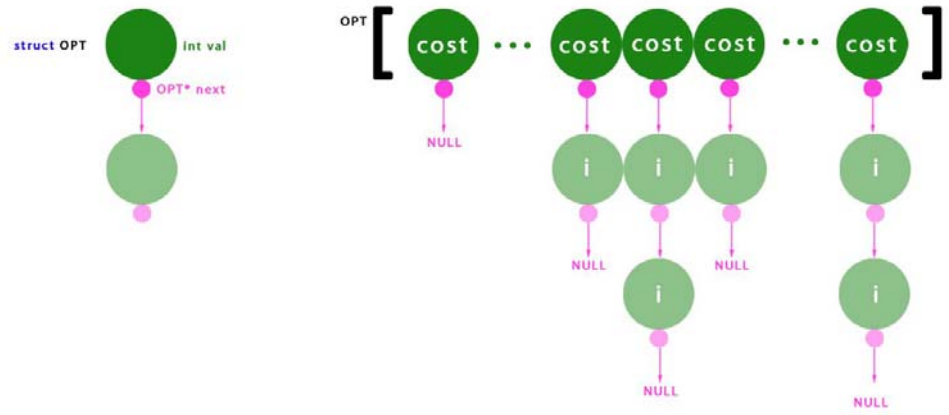
After that, we build up the 2-dimensional array M[n+1][m+1], the elements of the first row and column of M are all 0 as the boundary condition. Then M[i][j] directly denote the relation Sensor[i] and Target[j].

OPT[j] is the optimal solution the minimum sum of total costs of sensors. optSensor[j] stores the index of the optimal sensor corresponding to OPT[j] so that we can trace which sensors. OPT[j] stores the minimum cost but we need an index of the corresponding index. In this regard, we need to keep indices of sensors too.

```
dynamicFindMincov( c[n+1], M[m+1][n+1] )
    Allocate an array OPT[n+1]
    Allocate an array optSensor[n+1]        //only to store the index of the corresponding optimal sensor
    OPT[0] = 0        //the boundary condition
    optSensor[0] = NULL

    For j = 1 … n
        OPT[j] = ∞
    Endfor

    For j = 1 … n
        For i = 1 … m
            If M[i][j] ≠ 0 then          //Target[j] is covered by Sensor[i]
                If ( c[i] + OPT[j – M[i][j]] ) < OPT[j] then        //more optimal solution
                    OPT[j] = (c[i] + OPT[ j – M[i][j]] )
                    optSensor[j] = i
                Endif
            Endif
        Endfor
    Endfor
```

The blue box denotes optSensor[j] which contains an index of the corresponding sensor. We already have j, which is initially the number of targets, so we can refer M[i][j] with i from optSensor[j](optSensor[j] stores an index i of the corresponding optimal sensor). It means that we can print out the list of the cover set of the minimum cost recursively because we can get next index j from by subtracting M[i][j](OPT[j] = c[i] + OPT[j – M[i][j]]).

However, if we use optSensor[j] we can only trace only one optimal solution but there can be more than one optimal solution. Let OPT is a structure which can contain int data type and OPT type pointer OPT*.

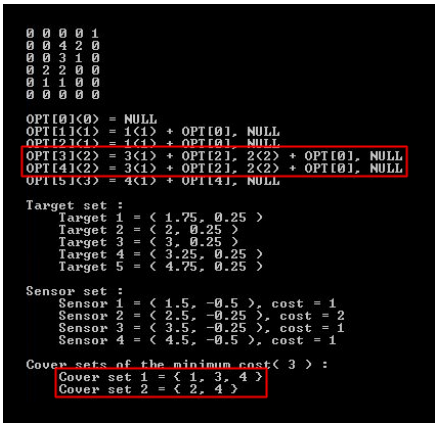


If we use this structure using pointer to itself instead of optSensor[j], not only can we save the time to allocate the array, but also we can possibly contain all optimal solutions and print them out.

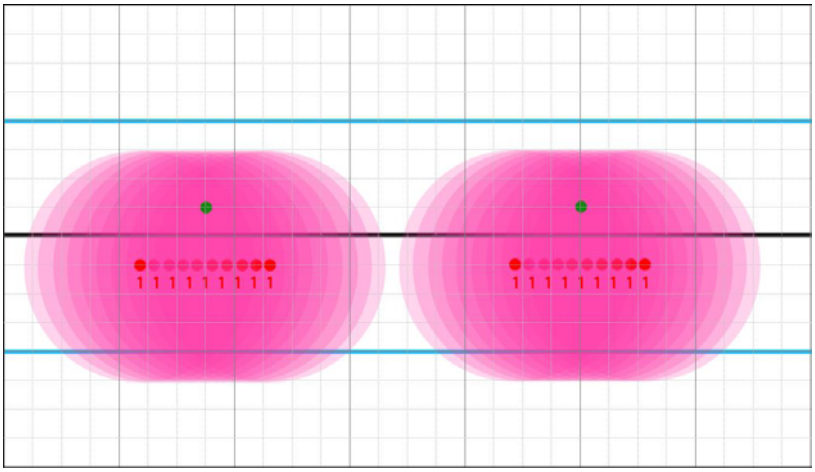
Due to efficiency, we will use a pointer in lieu of the array optSensor[n+1].

```
dynamicFindMincov( c[n+1], M[m+1][n+1] )
    Allocate an array OPT[n+1]
    OPT[0] = 0          //the boundary condition

    For j = 1 … n
        OPT[j] = ∞
    Endfor

    For j = 1 … n
        For i = 1 … m
            If M[i][j] ≠ 0 then             //Target[j] is covered by Sensor[i]
                If ( c[i] + OPT[j – M[i][j]] ) < OPT[j] then          //more optimal solution
                    OPT[j].val = (c[i] + OPT[ j – M[i][j]] )
                    OPT[j]→ next = new OPT with the value i
                Else if ( c[i] + OPT[j – M[i][j]] ) = OPT[j] then
                    Add new OPT with the value i to in front of the sequence OPT[j]
                Endif
            Endif
        Endfor
    Endfor
```

Now we can possibly store all optimal solutions, which means that we can also print out them all.



So far, we have found the optimal solution by using dynamic programming. Now we have to print the optimal cover sets but there is them problem.



The picture is describing the problem. There are 2 targets and 20 sensors and each target is covered by 10 sensors which have same cost. And then the possible combination of optimal cover set is 100. There is possibility of exponentially many optimal solutions. So we just print out only one optimal cover set for now.

Let's assume that the function printMincov(j) can refer OPT[n] and M[m+1][n+1] inside the function.

```
printMincov( j )
    If j = 0 then
        return
    Else
        printMincov( j – M[OPT[j]->next->i][j] )      //optSensor[j] is i, index of the corresponding optimal sensor
    Endif

    print OPT[j]->next->i
```

---

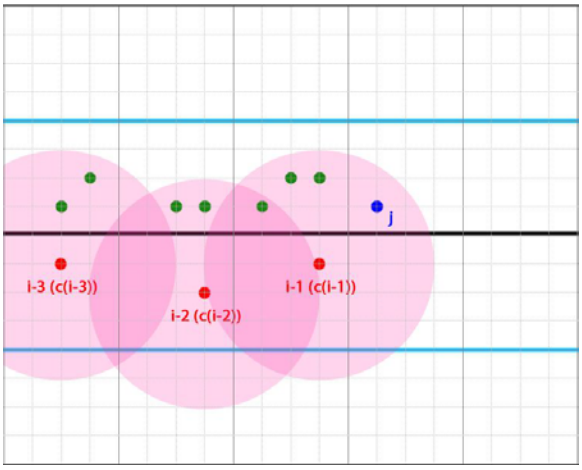
4. Proof of correctness

By induction,

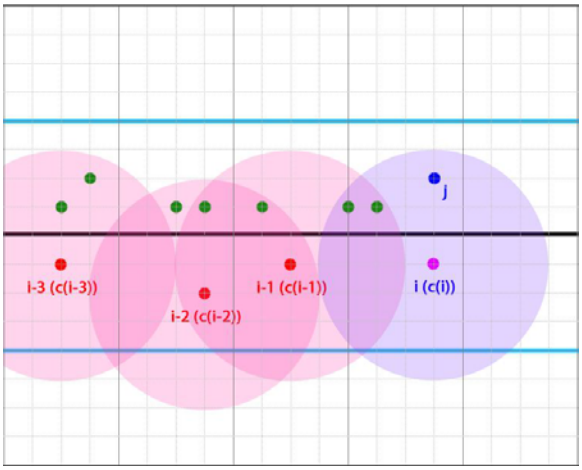
$$OPT(1) = \min_{1 \le i \le m} (c(i) + OPT(0))$$ (m is the number of sensors)

It is quite intuitive because Target[1], the first target, can be covered by at least one sensor(the project description assumes that each target can be monitored by at least one sensor). $OPT(0)$ is 0 as the boundary condition and then $OPT(1)$ is the minimum cost of sensors which cover Target[1].

Let's assume that $OPT(j-1)$ is the optimal solution, the minimum sum of costs of cover set which cover all targets. When Target[j] comes to the problem, there are two cases.



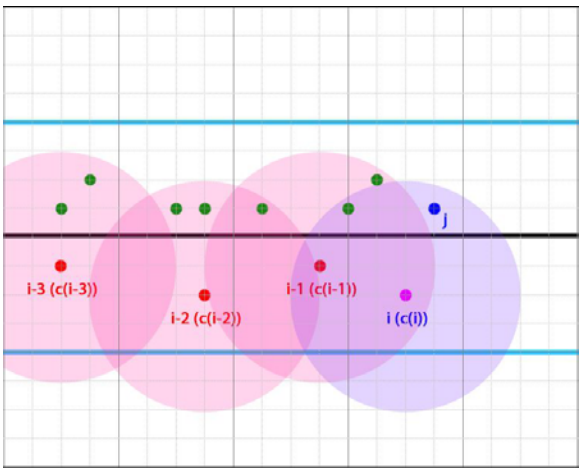
if $Target[j] \in OPT(j-1)$, then $OPT(j) = OPT(j-1)$. This is obvious because $OPT(j-1)$ is already optimal for targets {1, … , j-1}. If we add any sensor to the optimal cover set the total will be increased. If we change any sensor of the optimal cover set, the total cost will be same if there is another optimal solution. Or the total cost will be also increased because it is optimal.



Another case is $Target[j] \notin OPT(j-1)$. In this case, it is inevitable to add Sensor[i] which covers Target[j]. The most important part here is that Sensor[i] can possibly cover more than just Target[j]. In the above example, Sensor[i] covers three targets. Then it seems like below:

$$OPT(j) = c(i) + \min_{1 \le i \le m} OPT(j-3)$$ (m is the number of sensors)

However, it must not be just the number of targets covered by Sensor[i].



The previous picture shows that Sensor[i] covers two targets, Target[j-2] and Target[j]. In this case, following formula dose not work.

$$OPT(j) = c(i) + \min_{1 \le i \le m} OPT(2)$$ (m is the number of sensors)

Which ignores Target[j-1]. Conclusively, we can only remove the consecutive set of targets from the last one covered by Sensor[i]. $M[i][j]$ denotes the number of consecutive targets from the last target which are covered by Sensor[i]. So we can use the following formula.

$$OPT(j) = c(i) + \min_{1 \le i \le m} OPT(j - M[i][j])$$ (m is the number of sensors)

We ignore the case of $M[i][j]$ is 0, which means no relation between senor[i] and target[j]. That is, $M[i][j]$ is bigger than or equal to 1. We assume that $OPT(j-1)$ is optimal. Therefore, we can get $OPT(j)$ by adding a sensor of the minimum cost which can cover Target[j] to $OPT(j-1)$.

5. Analysis of the running time

First of all, we read the list of targets and the list of sensors. There are n targets and m sensors and it takes $O(1)$ to read from the file and write to the memory. So each takes $O(n)$ and $O(m)$ to read and write the lists.

Once we get the lists of targets and sensors, we first have to align them in the increasing order of x-coordinate. By using heap sorting, adding targets and sensors into heap takes each $O(n)$ and $O(m)$. After that, we can align them by using minHeap, the function of heap structure that returns the minimum value. The function minHeap takes the time of logarithm. So it takes total $O(nlogn)$ for sorting targets and $O(mlogm)$ for sorting sensors.

We also have to make the array $M[m+1][n+1]$ which contains relation between each target and sensor. It takes $O(mn)$ to allocate the array. And then we assign value to each element of the array. It takes another $O(mn)$ but it is still bounded by $O(mn)$.

Now we can compute $OPT(n)$. Allocating the array $OPT(n)$ takes $O(n)$.

$$OPT(j) = \min_{1 \le i \le m} (c(i) + OPT(j - M[i][j]))$$ (m is the number of sensors)

There are total n iterations for j($1 \le j \le n$) and each iteration has m iterations for i($1 \le i \le m$). Thus, it also takes $O(mn)$ to compute optimal solution. We are now done.

$$O(n) + O(m) + O(nlogn) + O(mlogn) + O(mn) + O(mn) = O(mn)$$

To find the optimal cost and store indices of the optimal cover sets, it takes total $O(mn)$ time.

There was a problem about the number of the optimal solutions. There can be exponentially many optimal solutions. So we would print out only one optimal solution. This process can be done by exploring through $OPT(n)$. In worst case, each target is covered by each cover, we have to explore all elements of $OPT(n)$. Thus, it takes at most $O(n)$ to print out the optimal solution, the cover set of the minimum cost.

To sum up, $O(mn)$ time for finding optimal solutions and $O(n)$ time to print out one optimal cover set.