SLOTHFUL SLINGSHOT

GROUP 6 :

TUBAGUS IRKHAM I.A. (05111740000012)

CELIA CHINTARA YUWINE (05111740000058)

HAYU AJENG RADRIYANTAMI (05111740000151)

DESCRIPTION

SLOTHFUL SLINGSHOT

SLOHTFUL SLINGSHOT is a program to find the smallest cost that must be spent to depart from the starting point to the finish point through several catapults along the start point to the finish point.

To make it easier to calculate the distance between starting point, finish point, and catapults, we use the coordinate plane. The location of these points can be represented by coordinate points (x, y).

The distance between two points is calculated as a cost.

Each catapult has a different throwing power.

To move from one catapult to another catapult, it can be done in two ways, by moving manually or using the help of its power. Then, the smallest cost will be chosen to continue the calculation process. If the catapult has a throwing power that is greater than the distance to the other catapults, then the cost is the difference from the throwing power minus the distance between those two catapults.

The distance from the start to the other catapult is directly calculated as a cost, because at the start point there is no throwing power.

The smallest amount of the sum of all costs that must be taken from the start point to the finish point will be printed as the output of this program.

OBSERVATION

- First, we have to input the number of catapult, start point, finish point, catapult's coordinat and it's throwing power.
- To determine the minimum cost from start point to finish point we can use BFS (Breadth-First Search) and use the priority queue. However, this priority queue has been designed with sorting from small to large. The coordinat point of catapults and its throwing power and the cost of the distance we store them in a struct, then we store the structs into vectors to make them more dynamic.
- With this method, we store the starting point into the vector and named it catapult[0], then we store the catapults into the vector and named it as catapult[i] which the value of i is 1 to n, and we store the finish point into the vector and named it as catapult[n + 1]. Then we initialize the cost for all points with a max number that is 10 ^ 9 * sqrt (2), except the cost initialization start point is equal to 0.
- Calculate the cost from the start point to other points, then update the value of the cost on the struct, compare it with the previous cost, if it is smaller then push it into the priority queue.
- This step will be repeated according to the order of the contents of the struct in the priority queue before, until the check point is the same as the finish point, then the smallest cost result will be printed which has been stored in the top priority priority queue.
- The program will stop running if we input -1 5times.

SIMULATION

- INPUT

n = 5

x-start = 10

y-start = 10

x-finish = 1

y-finish = 1

catapult-1 cp = (3,3) , F = 7

catapult-2 cp = (8,3) , F = 7

catapult-3 cp = (8,8) , F = 7

catapult-4 cp = (3,8) , F = 7

catapult-5 cp = (5,5) , F = 5

Where

n = number of catapult

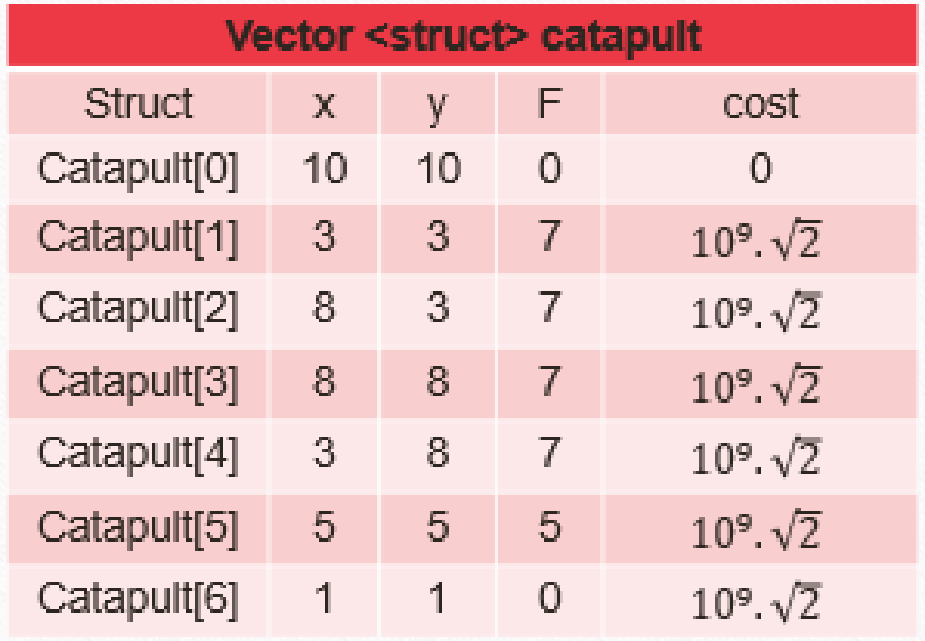
x and y -start = start point

x and y -finish = finish point

Cp = coordinate point of the catapult

F = throwing power of the catapult.

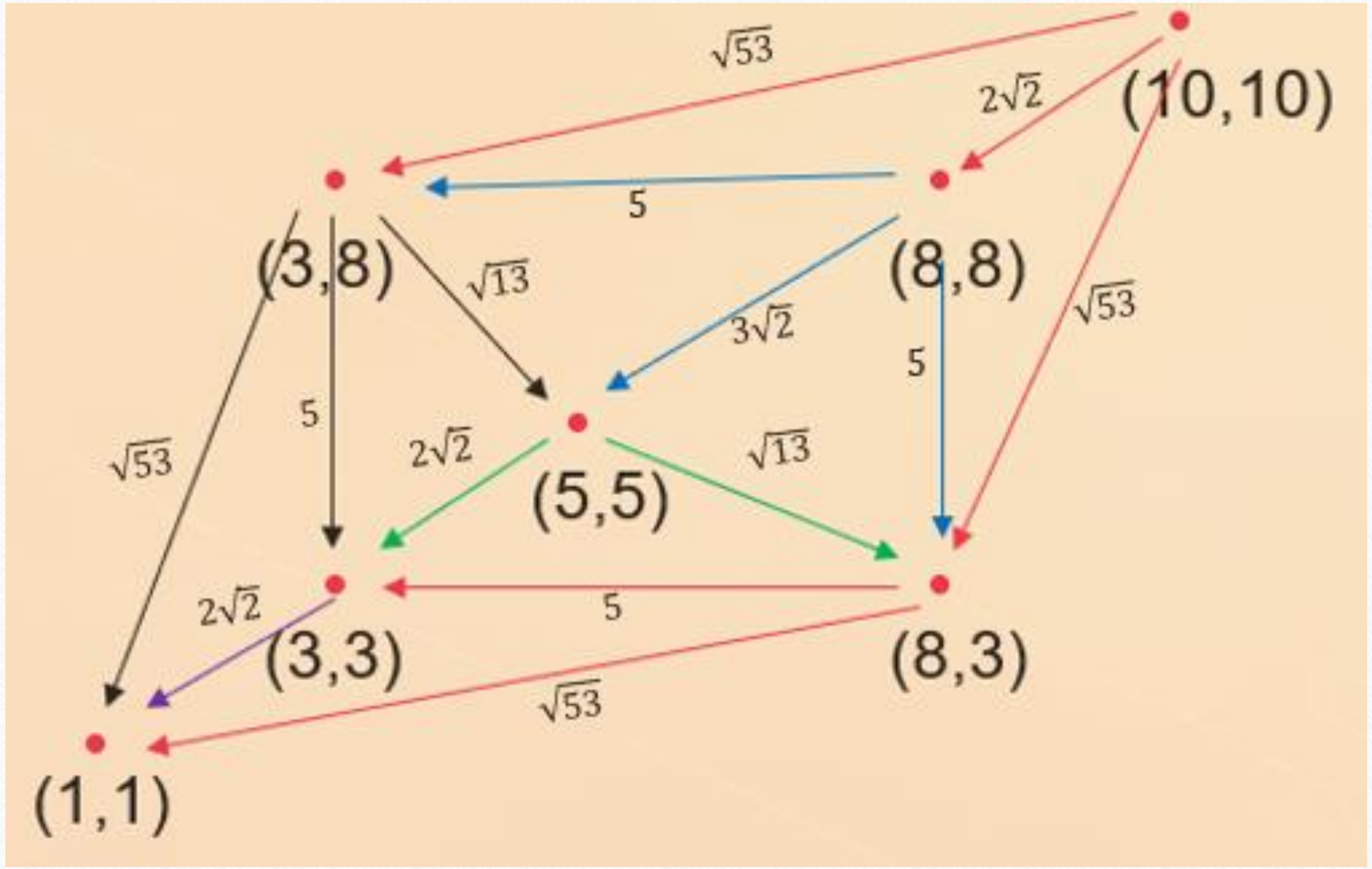
- Store them into the vector



Vector <struct> catapult

Struct	x	y	F	cost
Catapult[0]	10	10	0	0
Catapult[1]	3	3	7	$10^9.\sqrt{2}$
Catapult[2]	8	3	7	$10^9.\sqrt{2}$
Catapult[3]	8	8	7	$10^9.\sqrt{2}$
Catapult[4]	3	8	7	$10^9.\sqrt{2}$
Catapult[5]	5	5	5	$10^9.\sqrt{2}$
Catapult[6]	1	1	0	$10^9.\sqrt{2}$

- Find the distance from start point to others.



- Store catapult[0] into a variable named temp.

Then, find the distance(cost) from temp to other points and update the value of cost in the vector if there is a smaller cost than the previous cost.

Vector <struct> catapult				
Struct	x	y	F	cost
Catapult[0]	10	10	0	0
Catapult[1]	3	3	7	**9.90**
Catapult[2]	8	3	7	**7.28**
Catapult[3]	8	8	7	**2.83**
Catapult[4]	3	8	7	**7.28**
Catapult[5]	5	5	5	**7.07**
Catapult[6]	1	1	0	**12.73**

- Then, we push the struct into a priority queue which has been sorted by the smallest cost as it's top.

Priority queue<struct>		
	Catapult [index]	cost
Top	[3]	2.83
	[5]	7.07
	[4]	7.28
	[2]	7.28
	[1]	9.90
	[6]	12.73

- Check wether the top of priority queue is the same as the finish point or not.
- If it's not, do these steps :

Change the value of variable temp with the top of priority queue, then pop it.

Find the distance(cost) from temp to other points. Then do the calcutlation below :

Temp = catapult[3]

- Cost from temp -> catapult[1]

temp.cost = 7.07

If we use it's throwing power -> abs(7.07 – 7) = 0.07

So, temp.cost = 0.07 + catapult[3].cost = 2.90

Compare it with catapult[1].cost

2.90 < 9.90 , then replace it. So catapult[1].cost = 2.90

Push to priority queue.

Priority queue<struct>		
	Catapult [index]	cost
Top	[3]	2.83
	[5]	7.07
	[4]	7.28
	[2]	7.28
	[1]	9.90
	[6]	12.73

Priority queue before being updated

- Cost from temp -> catapult[2]

temp.cost = 5

If we use it's throwing power -> abs(5 – 7) = 2

So, temp.cost = 2 + catapult[3].cost = 4.83

Compare it with catapult[2].cost

4.83 < 7.28 , then replace it. So catapult[2].cost = 4.83

Push to priority queue.

- Cost from temp -> catapult[3] is skipped because the result will be the same as the previous value.

- Cost from temp -> catapult[4]

temp.cost = 5

If we use it's throwing power -> abs(5 – 7) = 2

So, temp.cost = 2 + catapult[3].cost = 4.83

Compare it with catapult[4].cost

4.83 < 7.28 , then replace it. So catapult[4].cost = 4.83

Push to priority queue.

Priority queue<struct>		
	Catapult [index]	cost
Top	[3]	2.83
	[5]	7.07
	[4]	7.28
	[2]	7.28
	[1]	9.90
	[6]	12.73

Priority queue before being updated

- Cost from temp -> catapult[5]

temp.cost = 4.24

If we use it's throwing power -> abs(4.24 – 7) = 2.76

So, temp.cost = 2.76 + catapult[3].cost = 5.59

Compare it with catapult[5].cost

5.59 < 7.07 , then replace it. So catapult[5].cost = 5.59

Push to priority queue.

- Cost from temp -> catapult[6]

temp.cost = 9.90

If we use it's throwing power -> abs(9.90 – 7) = 2.90

So, temp.cost = 2.90 + catapult[3].cost = 5.73

Compare it with catapult[6].cost

5.73 < 12.73 , then replace it. So catapult[6].cost = 5.73

Push to priority queue.

Priority queue<struct>

Priority queue before being updated

	Catapult [index]	cost
Top	[3]	2.83
	[5]	7.07
	[4]	7.28
	[2]	7.28
	[1]	9.90
	[6]	12.73

Priority queue<struct>

Priority queue after updated

	Catapult[index]	cost
Top	[1]	2.90
	[4]	4,83
	[2]	4,83
	[5]	5,59
	[6]	5,73

- Check wether the top of priority queue is the same as the finish point or not.
- If it’s not, do these steps :

Change the value of variable temp with the top of priority queue, then pop it.

Find the distance(cost) from temp to other points. Then do the calcutlation below :

Temp = catapult[1]

- Cost from temp -> catapult[1] is skipped because the result will be the same as the previous value.
- Cost from temp -> catapult[2]

temp.cost = 5

If we use it’s throwing power -> abs(5 – 7) = 2

So, temp.cost = 2 + catapult[1].cost = 4.90

Compare it with catapult[2].cost

4.90 > 4.83 , then catapult[2].cost is not being replaced.

Priority queue<struct>

	Catapult[index]	cost
Top	[1]	2.90
	[4]	4,83
	[2]	4,83
	[5]	5,59
	[6]	5,73

Priority queue before being updated

- Cost from temp -> catapult[3] is skipped because the result must be greater than the previous value.
- Cost from temp -> catapult[4]

temp.cost = 5

If we use it's throwing power -> abs(5 – 7) = 2

So, temp.cost = 2 + catapult[1].cost = 4.90

Compare it with catapult[4].cost

4.90 > 4.83 , then catapult[4].cost is not being replaced.

- Cost from temp -> catapult[5]

temp.cost = 2.83

If we use it's throwing power -> abs(2.83 – 7) = 4.17 (it's greater than temp.cost so we're not gonna use the throwing power)

So, temp.cost = 2.83 + catapult[1].cost = 5.73

Compare it with catapult[5].cost

5.73 > 5.59 , then catapult[5].cost is not being replaced.

Priority queue<struct>		
	Catapult[index]	cost
Top	[1]	2.90
	[4]	4,83
	[2]	4,83
	[5]	5,59
	[6]	5,73

Priority queue before being updated

- Cost from temp -> catapult[6]

temp.cost = 2.83

If we use it's throwing power -> abs(2.83 – 7) = 4.17 (it's greater than temp.cost so we're not gonna use the throwing power)

So, temp.cost = 2.83 + catapult[1].cost = 5.73

Compare it with catapult[6].cost

5.73 = 5.73 , then catapult[6].cost is not being replaced.

Priority queue<struct>		
	Catapult[index]	cost
Top	[1]	2.90
	[4]	4,83
	[2]	4,83
	[5]	5,59
	[6]	5,73

Priority queue before being updated

Priority queue after updated

Priority queue<struct>		
	Catapult[index]	cost
Top	[4]	4,83
	[2]	4,83
	[5]	5,59
	[6]	5,73

- Check wether the top of priority queue is the same as the finish point or not.
- If it's not, do these steps :

Change the value of variable temp with the top of priority queue, then pop it.

Find the distance(cost) from temp to other points. Then do the calcutlation below :

Temp = catapult[4]

- Cost from temp -> catapult[1] is skipped because the result must be greater than the previous value.
- Cost from temp -> catapult[2]

temp.cost = 7.07

If we use it's throwing power -> abs(7.07 – 7) = 0.07

So, temp.cost = 0.07 + catapult[4].cost = 4.90

Compare it with catapult[2].cost

4.90 > 4.83 , then catapult[2].cost is not being replaced.

- Cost from temp -> catapult[3] is skipped because the result must be greater than the previous value.
- Cost from temp -> catapult[4] is skipped because the result will be the same as the previous value.

Priority queue<struct>

	Catapult[index]	cost
Top	[4]	4,83
	[2]	4,83
	[5]	5,59
	[6]	5,73

Priority queue before being updated

- Cost from temp -> catapult[5]

temp.cost = 3.61

If we use it's throwing power -> abs(3.61 – 7) = 3.39

So, temp.cost = 3.39 + catapult[4].cost = 8.22

Compare it with catapult[5].cost

8.22 > 5.59 , then catapult[5].cost is not being replaced.

- Cost from temp -> catapult[6]

temp.cost = 7.28

If we use it's throwing power -> abs(7.28 – 7) = 0.28

So, temp.cost = 0.28 + catapult[4].cost = 5.11

Compare it with catapult[6].cost

5.11 < 5.73 , then replace it. So catapult[6].cost = 5.11

Push to priority queue.

Priority queue<struct>

	Catapult[index]	cost
Top	[4]	4,83
	[2]	4,83
	[5]	5,59
	[6]	5,73

Priority queue before being updated

Priority queue after updated

Priority queue<struct>

	Catapult[index]	cost
Top	[2]	4,83
	[6]	5,11
	[5]	5,59

- Check wether the top of priority queue is the same as the finish point or not.
- If it's not, do these steps :

Change the value of variable temp with the top of priority queue, then pop it.

Find the distance(cost) from temp to other points. Then do the calcutlation below :

Temp = catapult[2]

- Cost from temp -> catapult[1] is skipped because the result must be greater than the previous value.
- Cost from temp -> catapult[2] is skipped because the result will be the same as the previous value.
- Cost from temp -> catapult[3] is skipped because the result must be greater than the previous value.
- Cost from temp -> catapult[4] is skipped because the result must be greater than the previous value.

Priority queue<struct>		
	Catapult[index]	cost
Top	[2]	4,83
	[6]	5,11
	[5]	5,59

Priority queue before being updated

- Cost from temp -> catapult[5]

temp.cost = 3.61

If we use it's throwing power -> abs(3.61 – 7) = 3.39

So, temp.cost = 3.39 + catapult[2].cost = 8.22

Compare it with catapult[5].cost

8.22 > 5.59 , then catapult[5].cost is not being replaced.

- Cost from temp -> catapult[6]

temp.cost = 7.28

If we use it's throwing power -> abs(7.28 – 7) = 0.28

So, temp.cost = 0.28 + catapult[2].cost = 5.11

Compare it with catapult[6].cost

5.11 = 5.11 , then catapult[6].cost is not being replaced

Priority queue<struct>		
	Catapult[index]	cost
Top	[2]	4,83
	[6]	5,11
	[5]	5,59

Priority queue before being updated

Priority queue<struct>		
	Catapult[indeks]	cost
Top	[6]	5,11
	[5]	5,59

Priority queue after updated

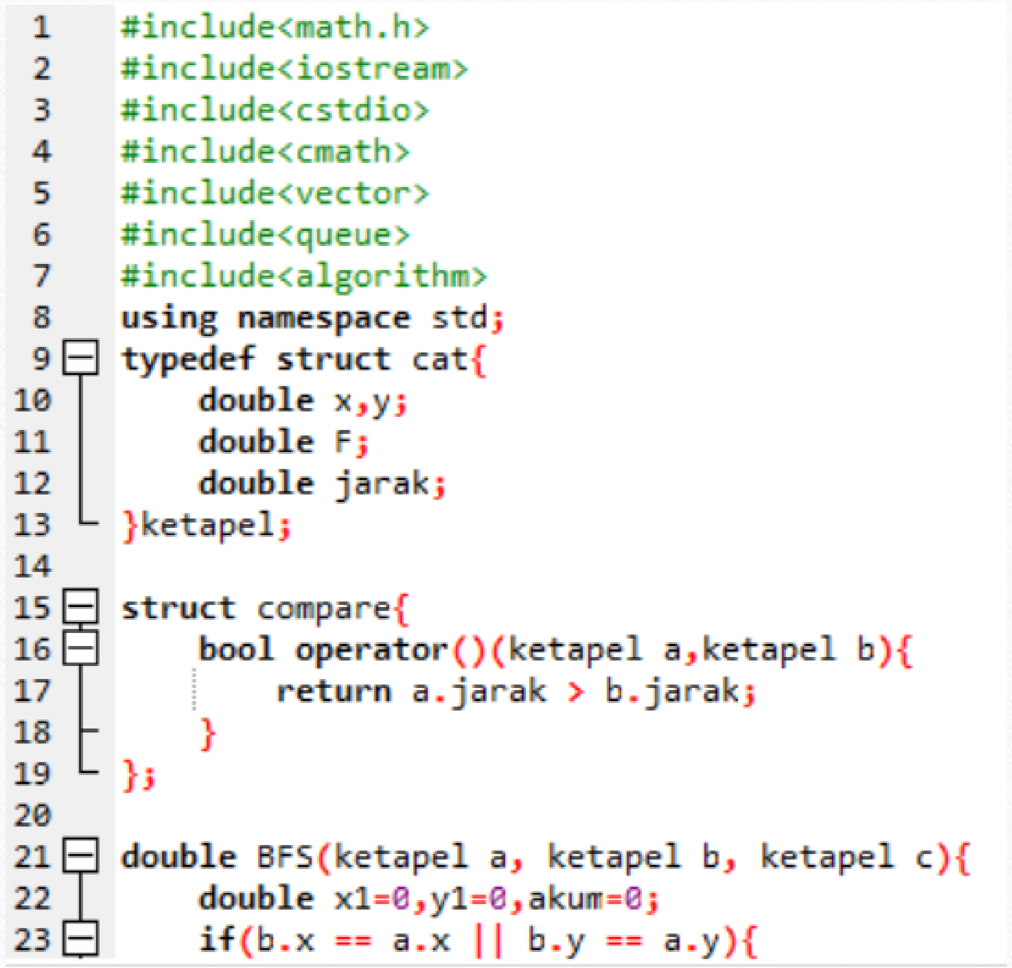
Priority queue<struct>		
	Catapult[indeks]	cost
Top	**[6]**	**5,11**
	[5]	5,59

Priority queue after updated

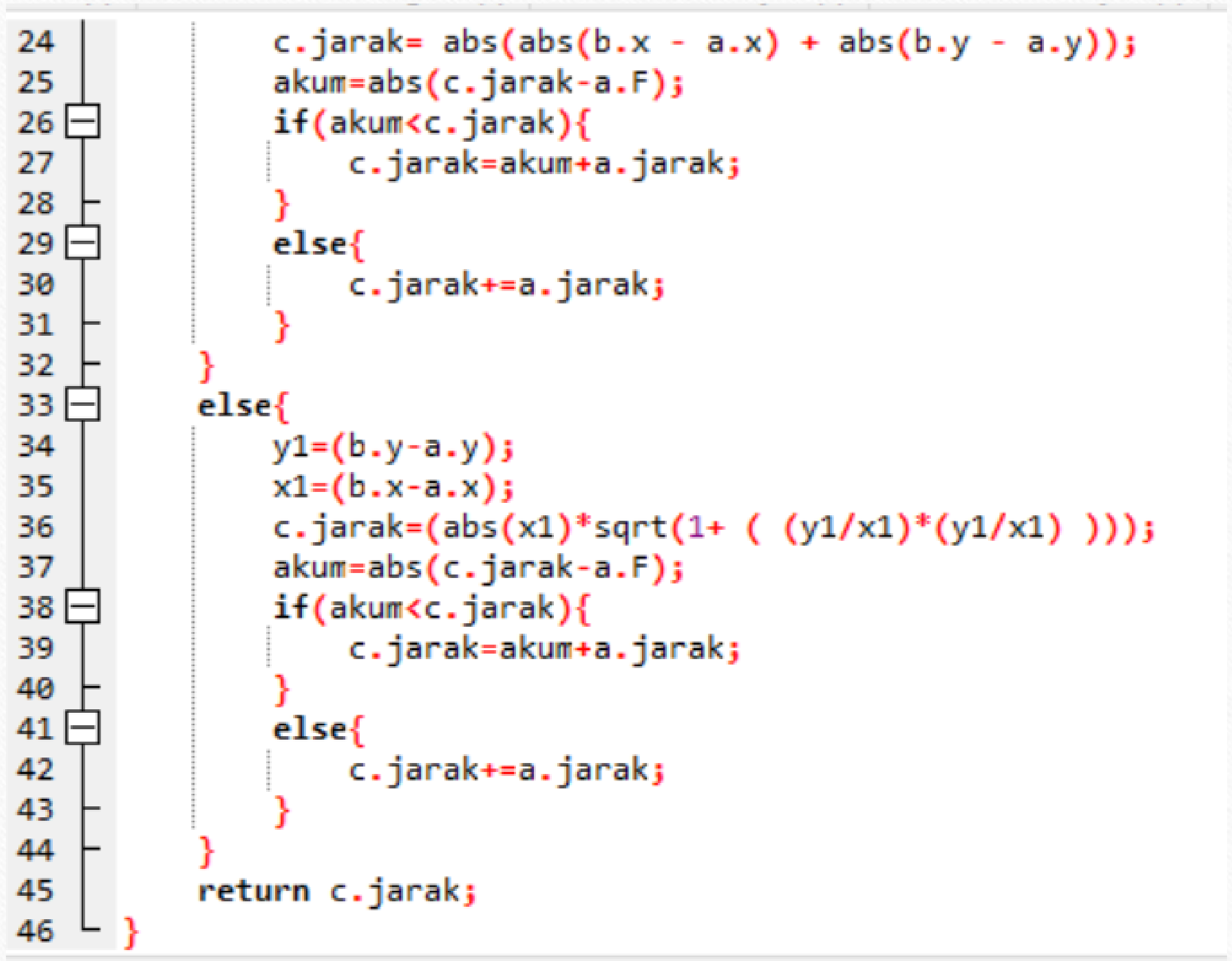
- Check wether the top of priority queue is the same as the finish point or not.
- If it's the same, stop the calculation.

- Then, the output is catapult[6].cost
- So, minimum cost to get to the finish point is **5.11**.

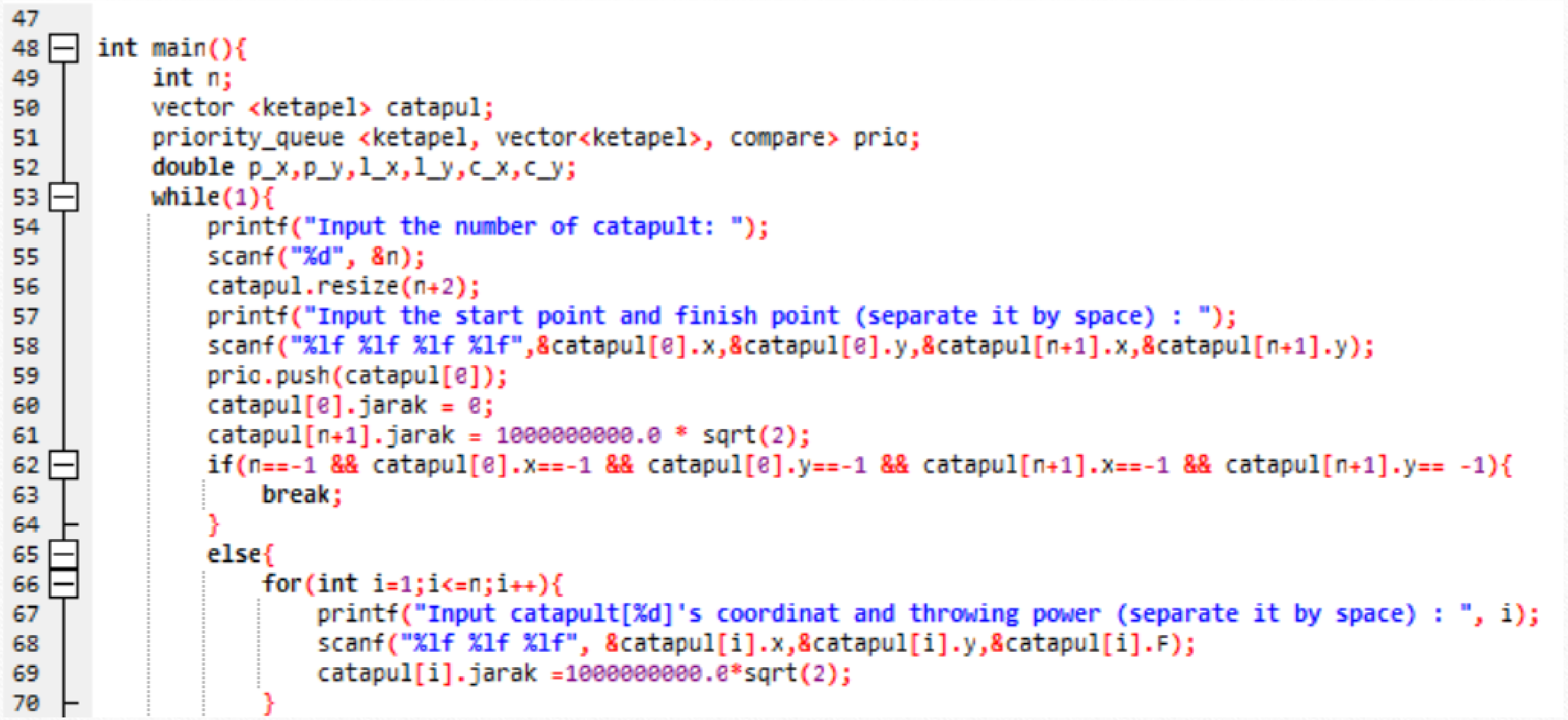
SOURCE CODE



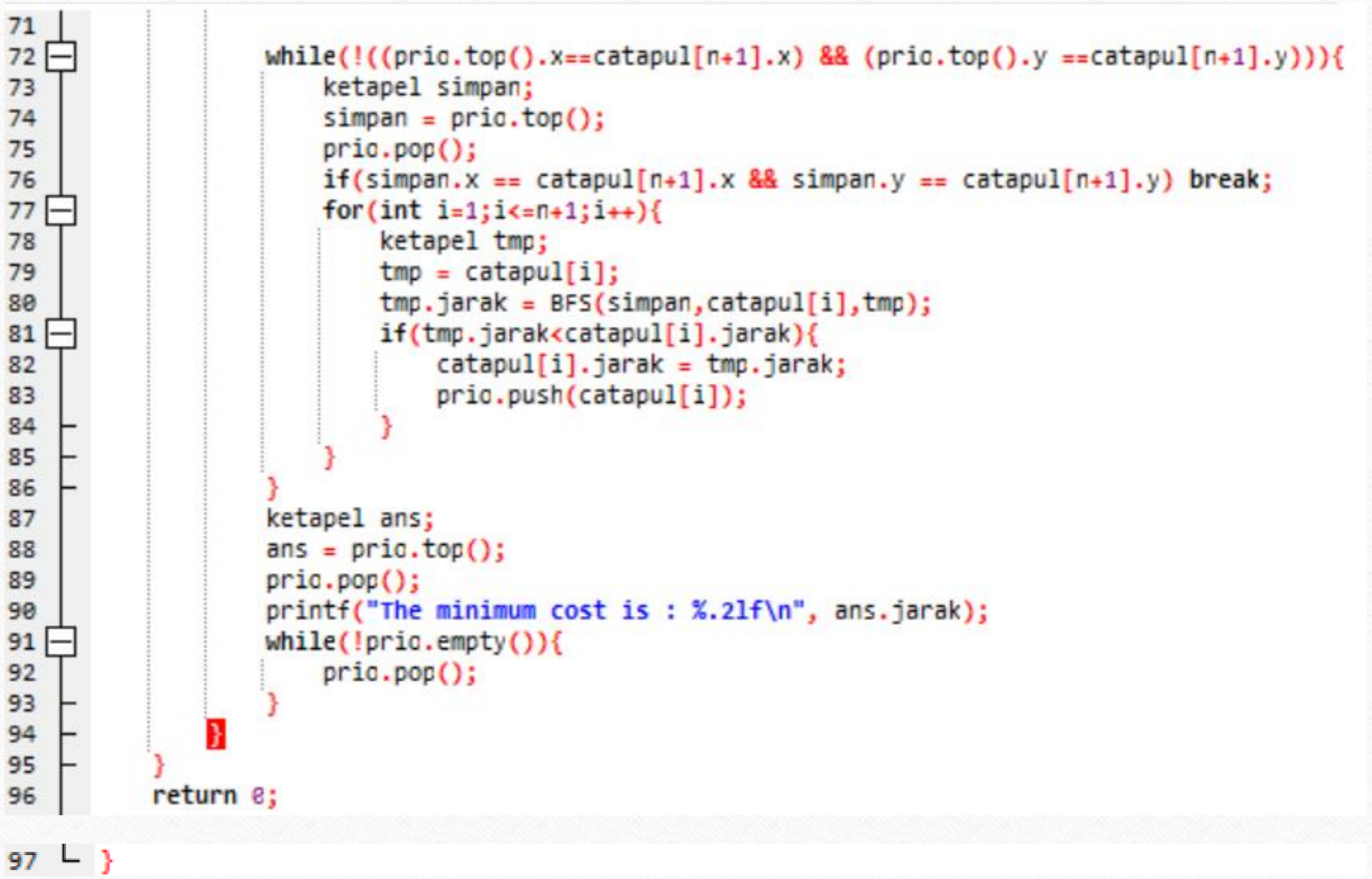
```cpp
#include<math.h>
#include<iostream>
#include<cstdio>
#include<cmath>
#include<vector>
#include<queue>
#include<algorithm>
using namespace std;
typedef struct cat{
    double x,y;
    double F;
    double jarak;
}ketapel;

struct compare{
    bool operator()(ketapel a,ketapel b){
        return a.jarak > b.jarak;
    }
};

double BFS(ketapel a, ketapel b, ketapel c){
    double x1=0,y1=0,akum=0;
    if(b.x == a.x || b.y == a.y){
```



```cpp
        c.jarak= abs(abs(b.x - a.x) + abs(b.y - a.y));
        akum=abs(c.jarak-a.F);
        if(akum<c.jarak){
            c.jarak=akum+a.jarak;
        }
        else{
            c.jarak+=a.jarak;
        }
    }
    else{
        y1=(b.y-a.y);
        x1=(b.x-a.x);
        c.jarak=(abs(x1)*sqrt(1+ ( (y1/x1)*(y1/x1) )));
        akum=abs(c.jarak-a.F);
        if(akum<c.jarak){
            c.jarak=akum+a.jarak;
        }
        else{
            c.jarak+=a.jarak;
        }
    }
    return c.jarak;
}
```

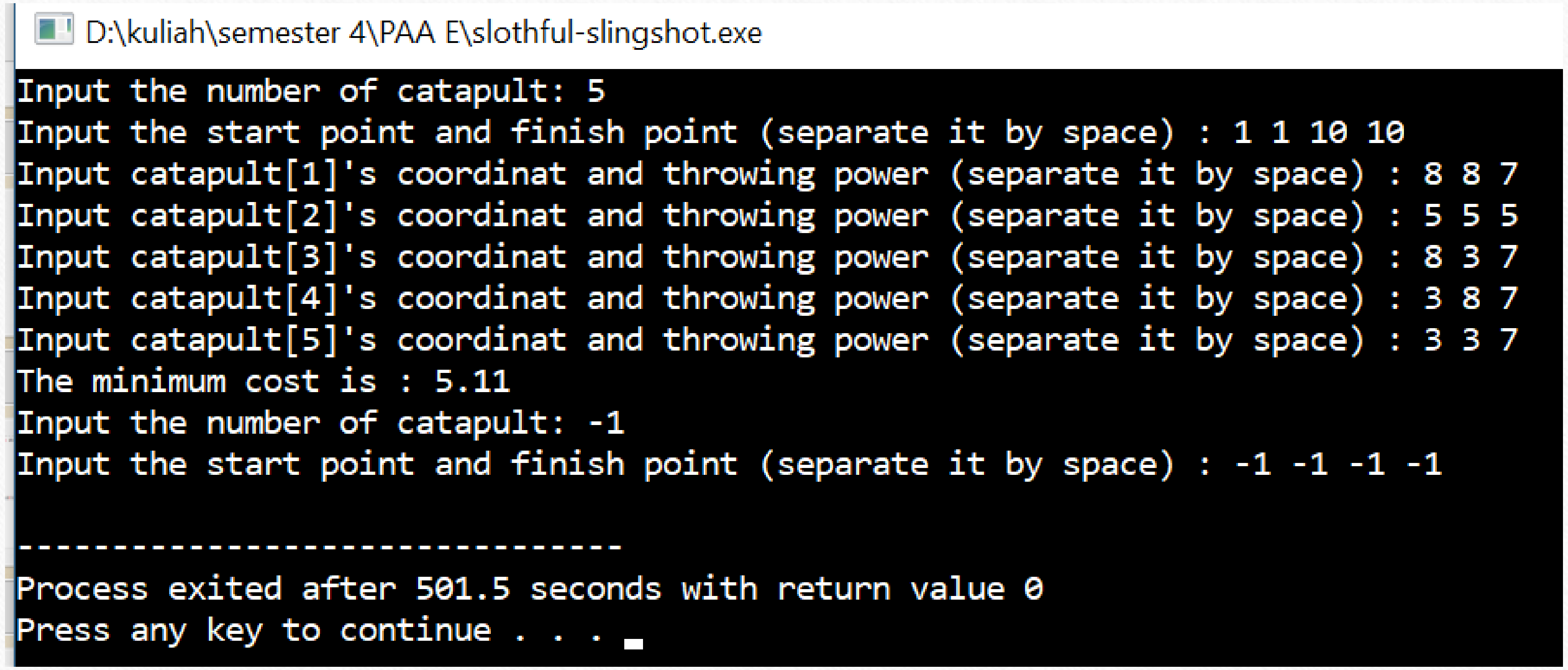


```

int main(){
    int n;
    vector <ketapel> catapul;
    priority_queue <ketapel, vector<ketapel>, compare> prio;
    double p_x,p_y,l_x,l_y,c_x,c_y;
    while(1){
        printf("Input the number of catapult: ");
        scanf("%d", &n);
        catapul.resize(n+2);
        printf("Input the start point and finish point (separate it by space) : ");
        scanf("%lf %lf %lf %lf",&catapul[0].x,&catapul[0].y,&catapul[n+1].x,&catapul[n+1].y);
        prio.push(catapul[0]);
        catapul[0].jarak = 0;
        catapul[n+1].jarak = 1000000000.0 * sqrt(2);
        if(n==-1 && catapul[0].x==-1 && catapul[0].y==-1 && catapul[n+1].x==-1 && catapul[n+1].y== -1){
            break;
        }
        else{
            for(int i=1;i<=n;i++){
                printf("Input catapult[%d]'s coordinat and throwing power (separate it by space) : ", i);
                scanf("%lf %lf %lf", &catapul[i].x,&catapul[i].y,&catapul[i].F);
                catapul[i].jarak =1000000000.0*sqrt(2);
            }
```



```

            while(!((prio.top().x==catapul[n+1].x) && (prio.top().y ==catapul[n+1].y))){
                ketapel simpan;
                simpan = prio.top();
                prio.pop();
                if(simpan.x == catapul[n+1].x && simpan.y == catapul[n+1].y) break;
                for(int i=1;i<=n+1;i++){
                    ketapel tmp;
                    tmp = catapul[i];
                    tmp.jarak = BFS(simpan,catapul[i],tmp);
                    if(tmp.jarak<catapul[i].jarak){
                        catapul[i].jarak = tmp.jarak;
                        prio.push(catapul[i]);
                    }
                }
            }
            ketapel ans;
            ans = prio.top();
            prio.pop();
            printf("The minimum cost is : %.2lf\n", ans.jarak);
            while(!prio.empty()){
                prio.pop();
            }
        }
    }
    return 0;
}
```

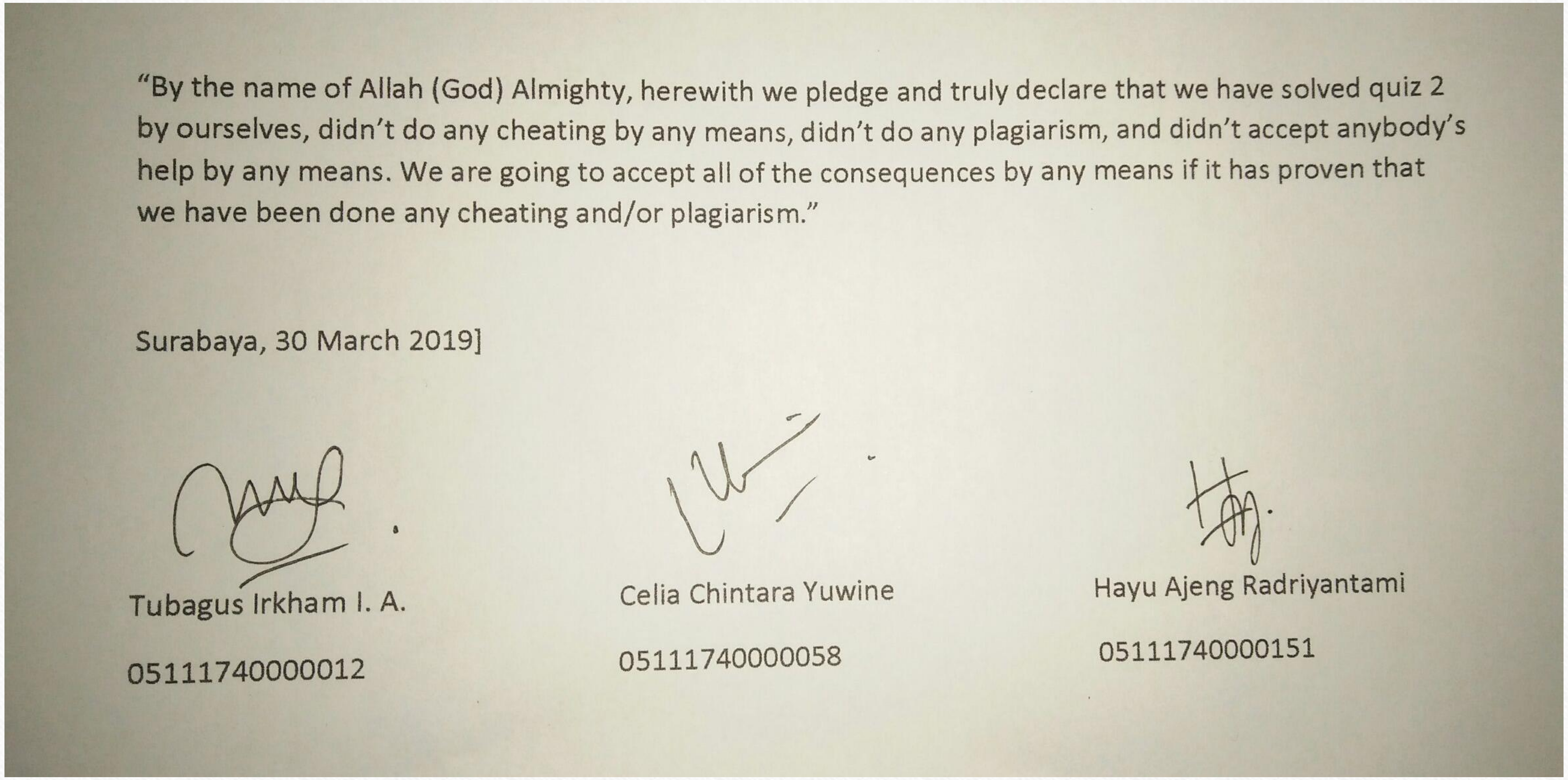
EXECUTION



D:\kuliah\semester 4\PAA E\slothful-slingshot.exe

```
Input the number of catapult: 5
Input the start point and finish point (separate it by space) : 1 1 10 10
Input catapult[1]'s coordinat and throwing power (separate it by space) : 8 8 7
Input catapult[2]'s coordinat and throwing power (separate it by space) : 5 5 5
Input catapult[3]'s coordinat and throwing power (separate it by space) : 8 3 7
Input catapult[4]'s coordinat and throwing power (separate it by space) : 3 8 7
Input catapult[5]'s coordinat and throwing power (separate it by space) : 3 3 7
The minimum cost is : 5.11
Input the number of catapult: -1
Input the start point and finish point (separate it by space) : -1 -1 -1 -1

--------------------------------
Process exited after 501.5 seconds with return value 0
Press any key to continue . . .
```

DECLARATION PAPER



"By the name of Allah (God) Almighty, herewith we pledge and truly declare that we have solved quiz 2 by ourselves, didn't do any cheating by any means, didn't do any plagiarism, and didn't accept anybody's help by any means. We are going to accept all of the consequences by any means if it has proven that we have been done any cheating and/or plagiarism."

Surabaya, 30 March 2019]

[illegible]	[illegible]	[illegible]
Tubagus Irkham I. A.	Celia Chintara Yuwine	Hayu Ajeng Radriyantami
05111740000012	05111740000058	05111740000151