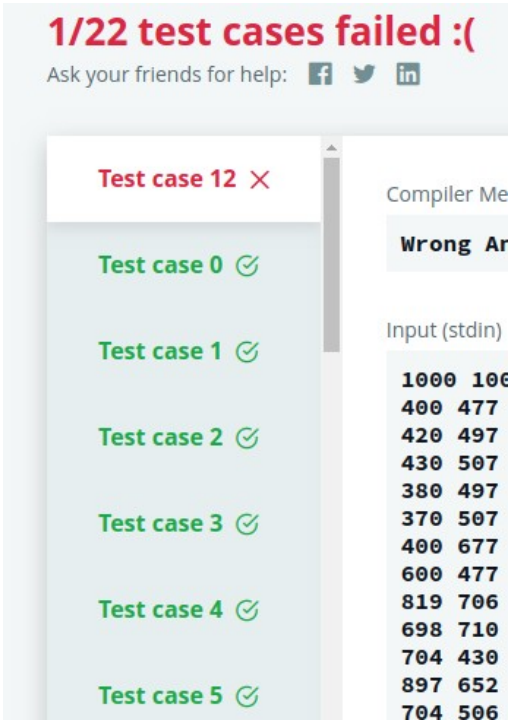
Editorial: Queen’s attack-2

problem link: https://www.hackerrank.com/challenges/queens-attack-2/problem

My solution passes 21/22 tests and got 28.17/30 points

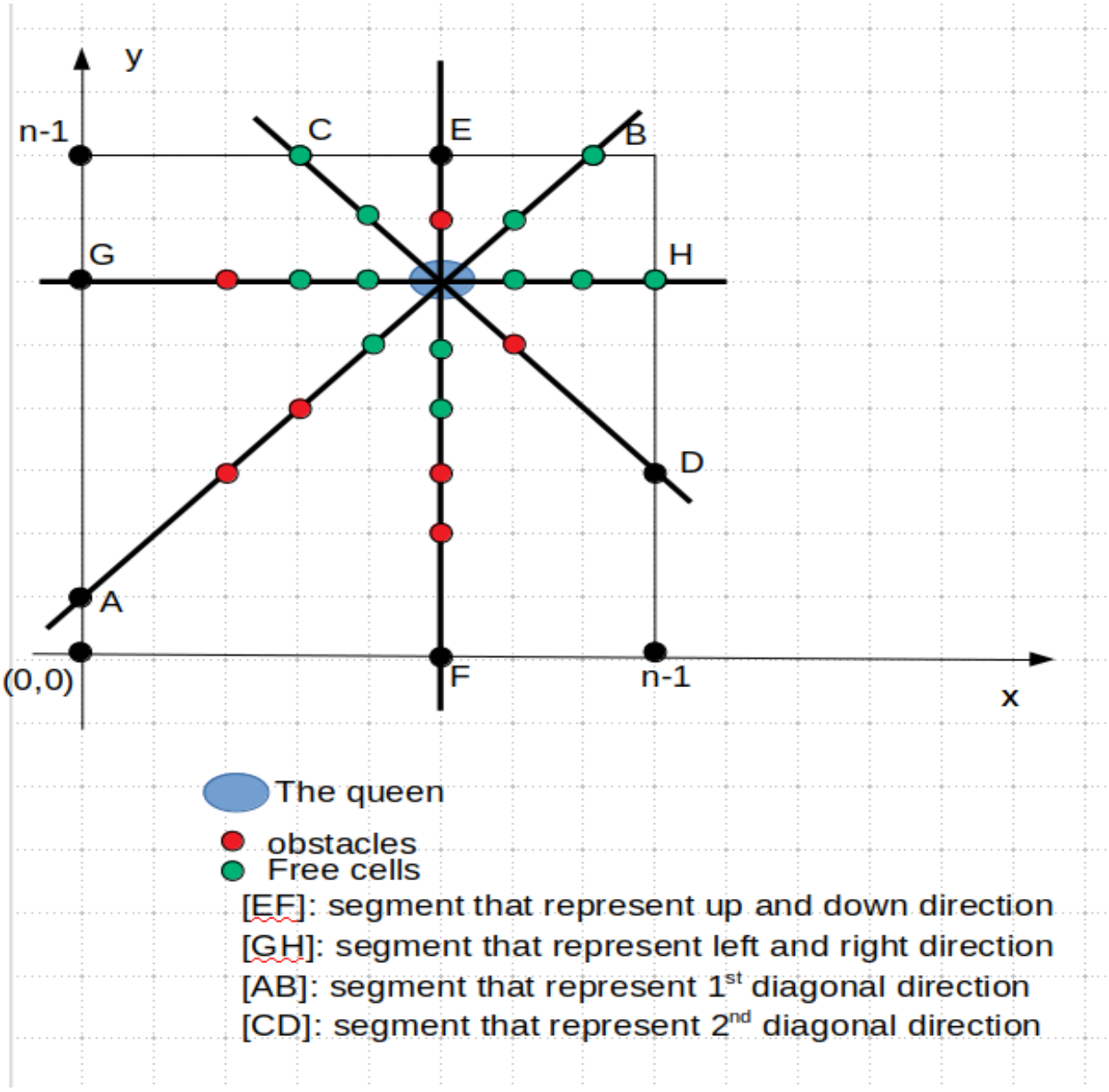


Problem statement

In the problem there is a chessboard of dimension $n x n$. In one cell there is a queen, also there are k obstacles that block the queen's path. You have to determine the number of **free cells** the queen can attack.

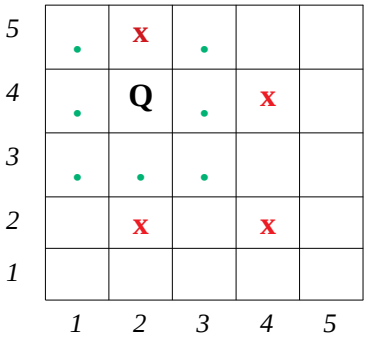
O(k) approach

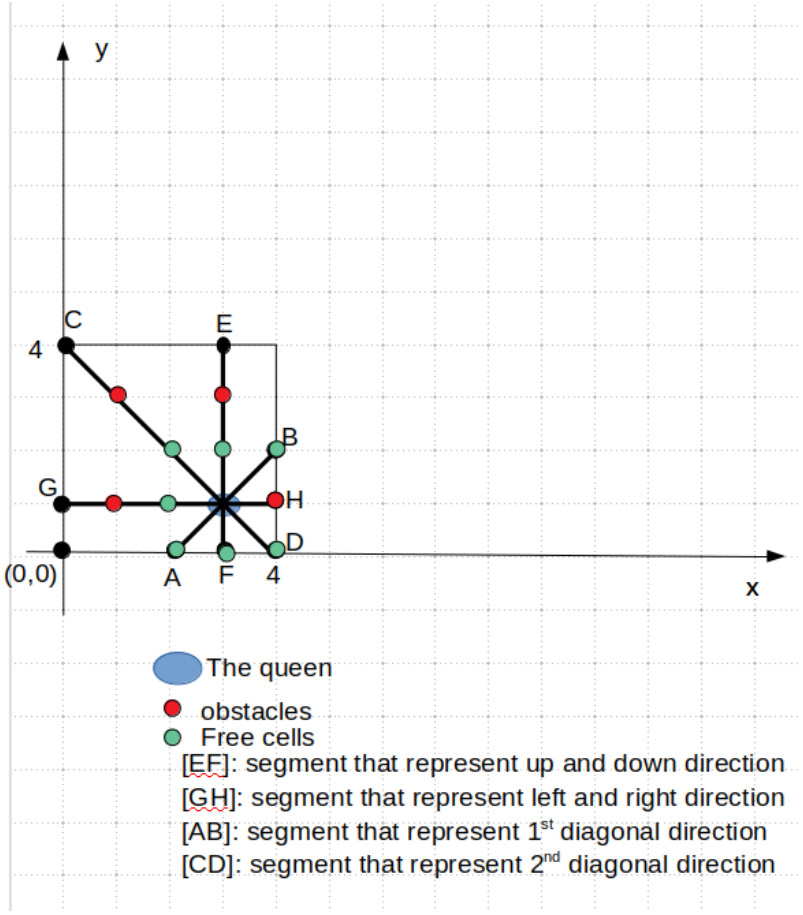
The idea is consider the chessboard as a 2d Cartesian plan:



Example:

5 4
4 2
5 2
2 2
2 4
4 4





Why computing lines segments?

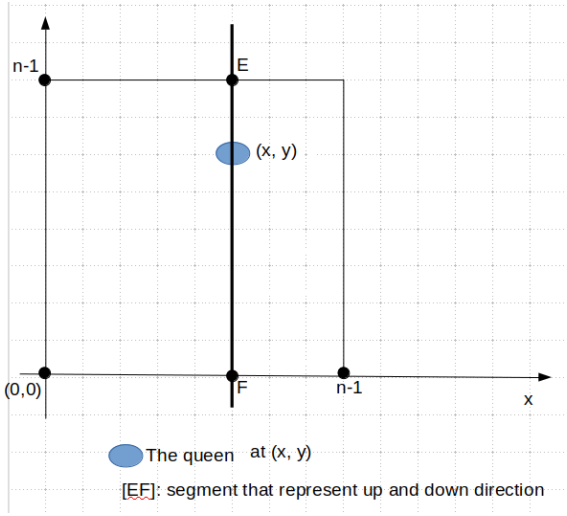
The purpose of computing all these lines segments, is to eliminate all obstacles that are on these line segments.

Computing all the points (the coordinates of the four lines segments)

All the computing is O(1)

Line segment [EF]

Points E and F

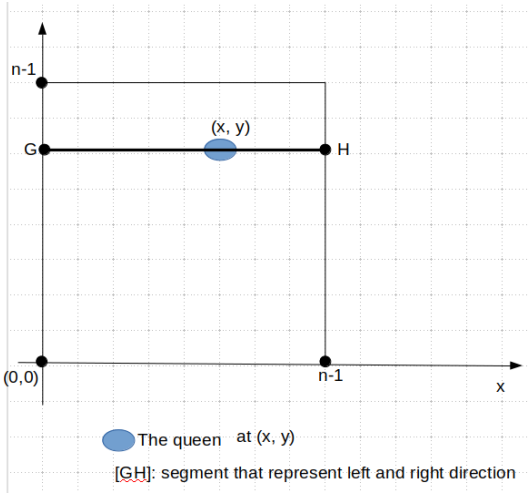


If the queen is at position (x, y) , then:

- $x_e = x, y_e = n-1$
- $x_f = x, y_f = 0$

Line segment [GH]

Points G and H



If the queen is at position (x, y) , then:

- $x_g = 0, y_g = y$
- $x_h = n-1, y_f = y$

Line segment [AB]

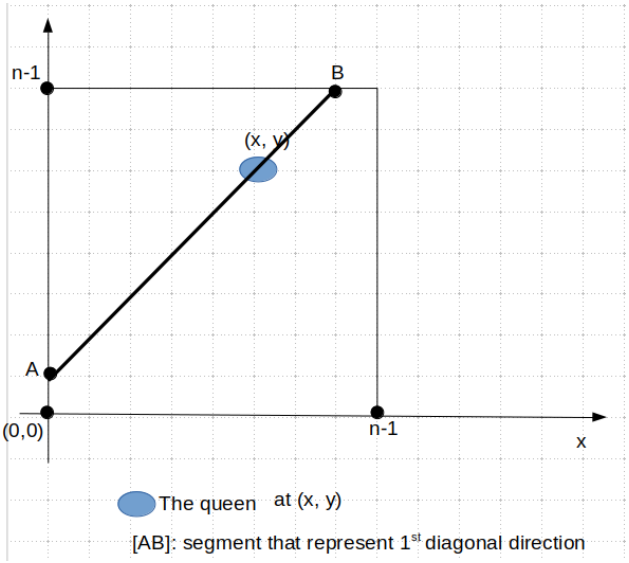


Figure #AB-1

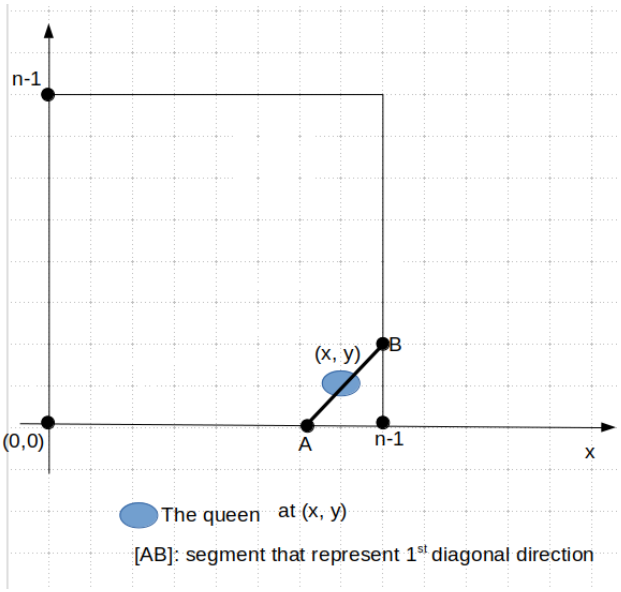


Figure #AB-2

As you see, in *figure #AB-1* and *figure #AB-2*, we (me and you :)) can't know the exact positions of A and B . but we know that are always in the edge of the square that represent the chessboard.
That's why, we have to determine the affine function of the $[AB]$ line segment.
We know that: $queen \in [AB]$ so, $y = ax + b$
another point is on $[AB]$, with coordinates $(x-1, y-1)$, so $y-1 = a(x-1) + b$

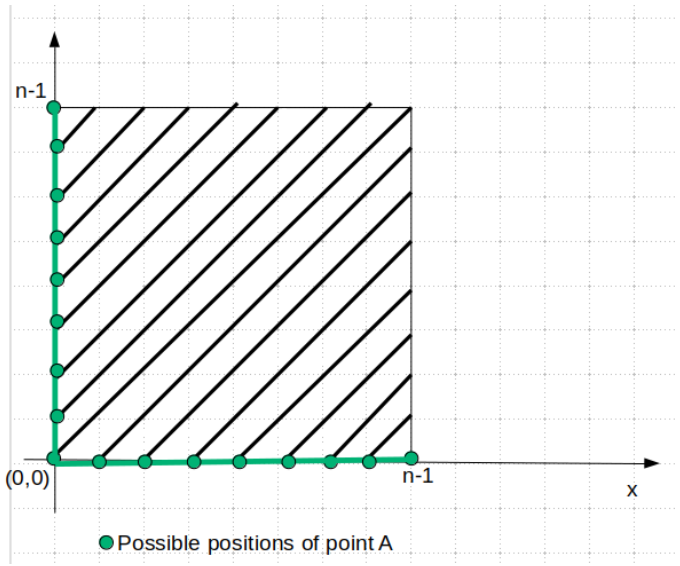
with this system of two equation, we can find the two parameters a and b, in function of x and y :

$$\begin{cases} y = ax + b \\ y - 1 = a(x-1) + b \end{cases} <=> \begin{cases} b = y - ax \\ y - 1 = ax - a + b \end{cases} <=> \begin{cases} b = y - ax \\ y - 1 = ax - a + y - ax \end{cases} <=> \begin{cases} b = y - x \\ a = 1 \end{cases}$$

C++ code:

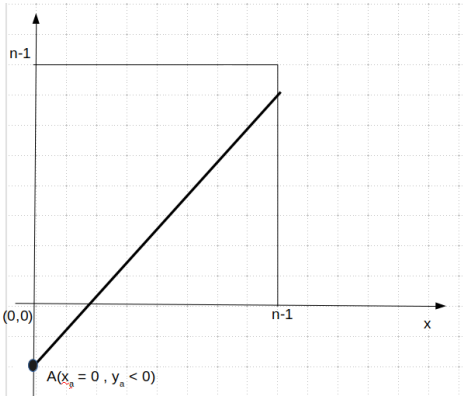
```
// Compute equation for AB ( y = ax + b )
int a_ab = 1;
int b_ab = y - x;
```

Point A



recall: (x, y) *are the given coordinates of the queen.*

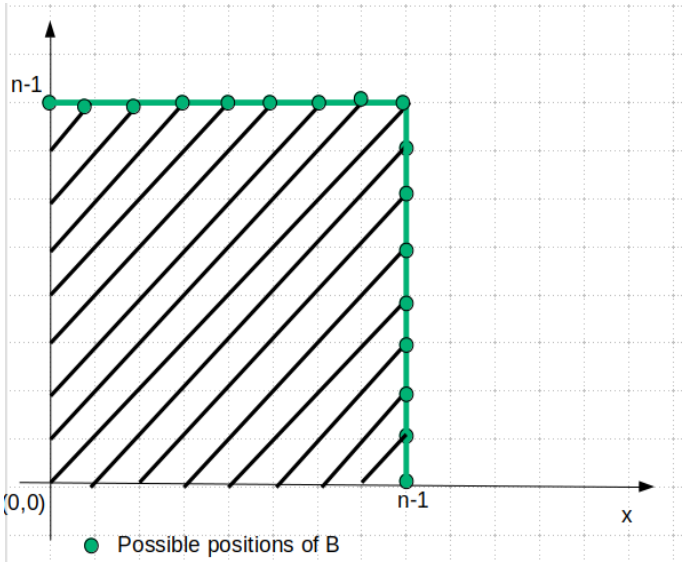
As you see in the illustration in the left
$x_a = 0$ or $y_a = 0$
if $x_a = 0$, then $y_a = ax_a + b = b = y - x$
but, be careful y_a could be less than 0.



So, we must recompute (x_a, y_a), to get A on the edge of the plan that represent the chessboard.

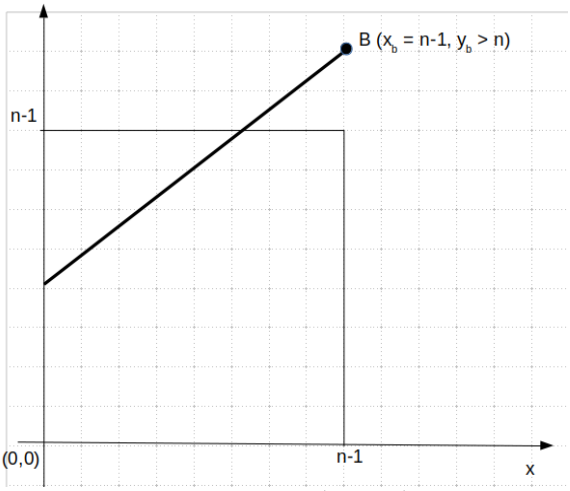
if $y_a < 0$, then $y_a = 0$, and recompute x_a by the formula of the affine function:
$x_a = -\frac{b}{a}$

Point B



recall: (x, y) *are the given coordinates of the queen.*

As you see in the illustration in the left , if $x_b = n-1$ then $y_b = ax_b + b$ but, be careful y_b could be greater than $n-1$.

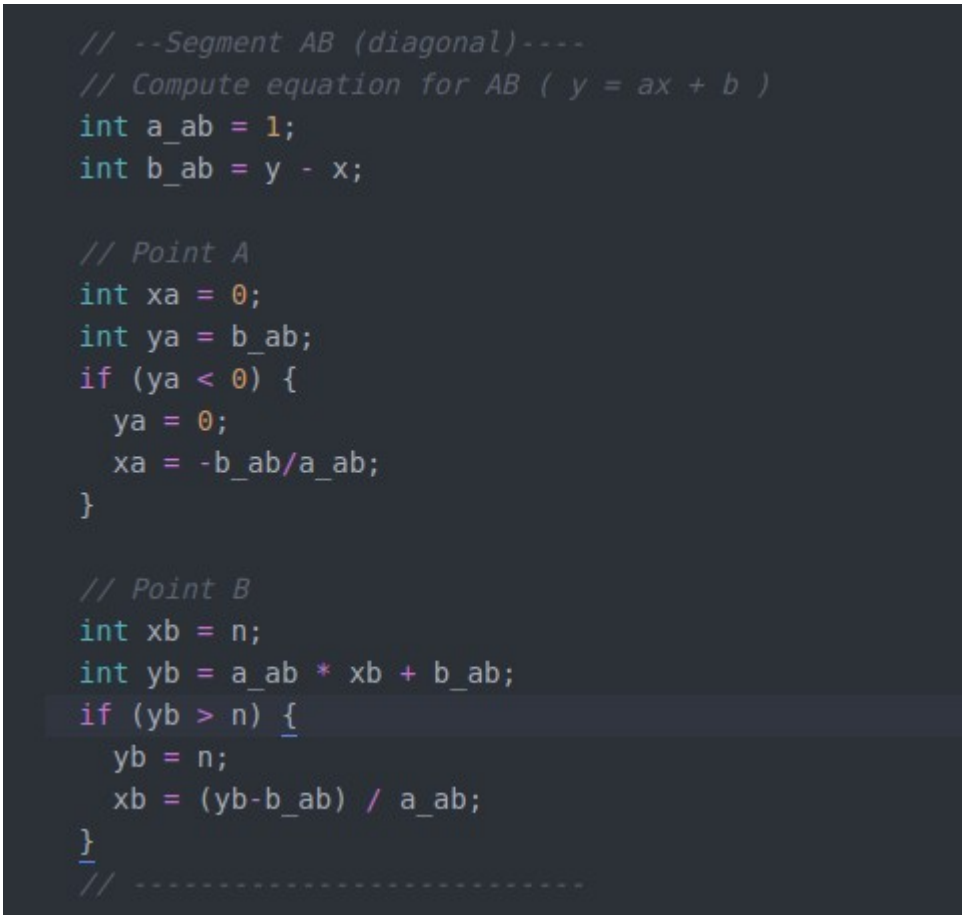


So, we must recompute (x_b, y_b) , to get B on the edge of the plan that represent the chessboard.

if $y_b > n-1$, then $y_b = n$, and recompute x_b by the formula of the affine function:

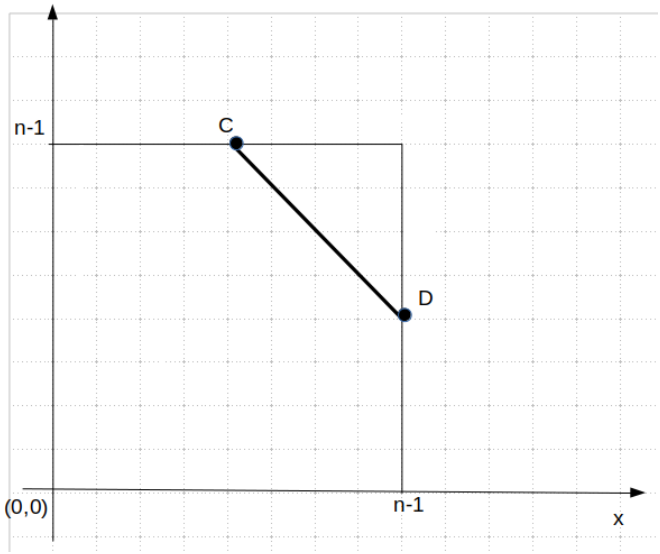
$$x_b = \frac{y_b - b}{a}$$

C++ code



```
// --Segment AB (diagonal)----
// Compute equation for AB ( y = ax + b )
int a_ab = 1;
int b_ab = y - x;

// Point A
int xa = 0;
int ya = b_ab;
if (ya < 0) {
  ya = 0;
  xa = -b_ab/a_ab;
}

// Point B
int xb = n;
int yb = a_ab * xb + b_ab;
if (yb > n) {
  yb = n;
  xb = (yb-b_ab) / a_ab;
}
// ---------------------------
```

Line segment [CD]



By applying the same approach of computing line segment [AB]:
computing the affine function:

$$\begin{cases} b = y + x \\ a = -1 \end{cases}$$

computing C point

by assuming that $y_c = n-1$, we can compute $x_c = \frac{y_c - b}{a}$

But, here x_c could be greater than $n-1$, if so, $x_c = n-1$ and $y_c = ax_c + b$
also x_c could be less than 0 , so, $x_c = 0$ and $y_c = b$

computing D point

assume that $y_d = 0$, → $x_d = b$
x_d could be greater than $n-1$, if so, $x_d = n-1$ and $y_d = ax_d + b$

C++ code

```
// --------Segment CD (diagonal)----
// Compute equation for CD ( y = ax + b )
int a_cd = -1;
int b_cd = y + x;

// Point C
int yc = n;
int xc = (yc-b_cd) / a_cd;
if (xc < 0) {
  xc = 0;
  yc = b_cd;
}
if (xc > n) {
  xc = n;
  yc = a_cd * xc + b_cd;
}

// Point D
int yd = 0;
int xd = b_cd;
if (xd > n) {
  xd = n;
  yd = a_cd * xd + b_cd;
}
```

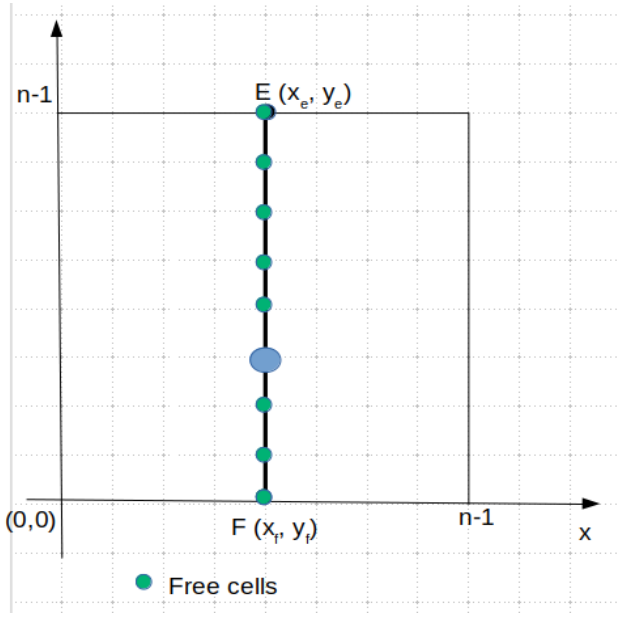
Bottom-up process

To compute the number of free cells that the queen could access:

1. we compute all the free cells for k=0 (no obstacles),
2. if an obstacle exists, reduce the number of cells blocked by it.

Compute all the free cells for k=0

On [EF]



It's the same the number of elements in an ordered list = $final\_index - first\_index + 1$
But, we don't need +1 here, because we don't wanna touch the edge of the chessboard

$$x_e = x_f \quad , \quad \#\,free\,cells = |y_e - y_f| = y_e$$

On [GH]

$$y_g = y_h \quad , \quad \#\,free\,cells = |x_g - x_h| = x_h$$

on [AB]

$$\#\,free\,cells = |x_a - x_b| = |y_a - y_b|$$

on [CD]

$$\#\,free\,cells = |x_c - x_d| = |y_c - y_d|$$

C++ code

```
int ans = abs(ye-yf) + abs(xg-xh) + abs(xc-xd) + abs(xa-xb);
```

Reducing the number of cells blocked by obstacles

For each obstacle (x_o, y_o):

- Determine in which line segment is it?,
- Determine in which part of the line segment is it?,
- Determine if it's the nearest one to the queen? (because we find more than on obstacles by line).

How to know if the obstacle is on [EF]?

It enough to check if $x_o = x_e$ (or x_f)
Now, to know if that obstacle is near E or F , just check if y_o is greater or less y (y of the queen)

How to know if the obstacle is on [GH]?

It enough to check if $y_o = y_g$ (or y_h)
Now, to know if that obstacle is near G or H , just check if x_o is greater or less x (x of the queen)

How to know if the obstacle is on [AB]/[CD]?

For both lines segments $[AB]$ and $[GH]$, it's a little bit different.

We have the coordinates of the obstacle (x_o, y_o)

1st, compute the value of given by the affine function of $[AB]/[CD]$ for x_o . And 2nd compare the result with y_o , if it's the same, then this obstacle is on $[AB]/[CD]$.

C++ code to know on each part of the segment an obstacle is it

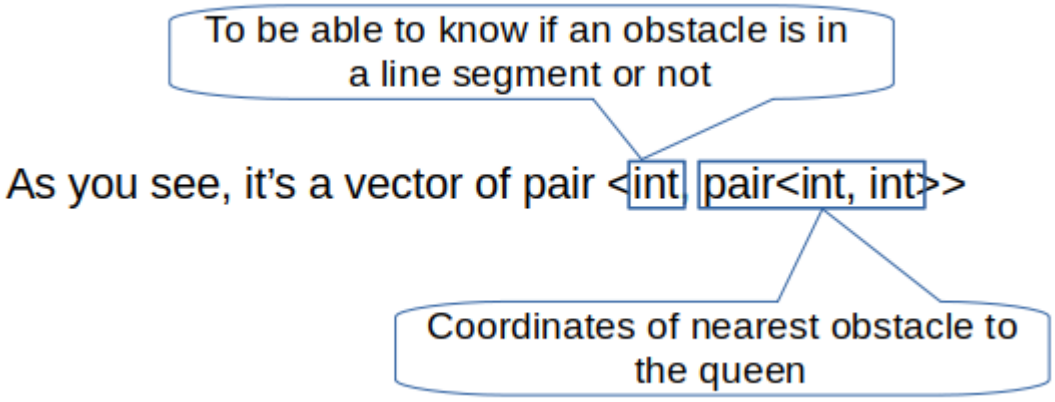
```
for (int i = 0 ; i < k ; ++i){
    int xo = obstacles[i].first;
    int yo = obstacles[i].second;

    // obstacle on EF (same x)
    if (xo == xe) {
      // E side
      if (yo > y)

      // F side
      else

    }

    // obstacles on GH (same y)
    if (yo == yg) {
      // H side
      if (xo > x)

      // G side
      else

    }

    // obstacles on AB
    int x_tmp = (yo - b_ab) / a_ab;
    if (x_tmp == xo) {
      // B side
      if (yo > y)

      // A side
      else

    }

    // obstacles on CD
    x_tmp = (yo - b_cd) / a_cd;
    if (x_tmp == xo) {
      // C side
      if (yo > y)

        // D side
      else

    }
  }
```

Computing the nearest obstacles to the queen

To achieve this task, we must store the computed nearest obstacle while progressing. We can use an array of size 8 (because we have 8 directions = 8 points).

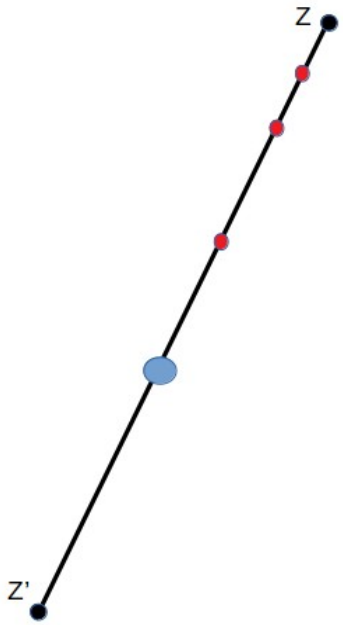
In my code I define, index 0 for point E, index 1 for point F, index 2 for point G, index 3 (point H), index 4 (point A), index 5 (point B), index 6 (point C) and index 7 (point D).

```
vector<pair<bool, pair<int, int>>> nearest_obstacle(8, make_pair(false, make_pair(1E+9, 1E+9)));
```



All initialized to 1E+9, to be able to compute the minimum.
Now, to compute the nearest obstacle, we must compare the distance between each obstacle with the queen, and store the minimum each time. Let's illustrate this:

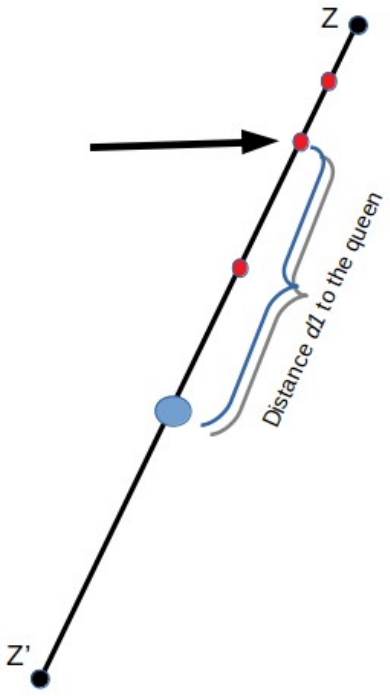
let’s take this line segment [ZZ’], and let’s compute the nearest obstacle to the queen:



In the beginning, the array status is: (let’s assume that index i refers to point Z)

nearest_obstacles:	{false, {1E+9,1E+9}}	{false, {1E+9,1E+9}}	{false, {1E+9,1E+9}}
	0	*i*	*7*

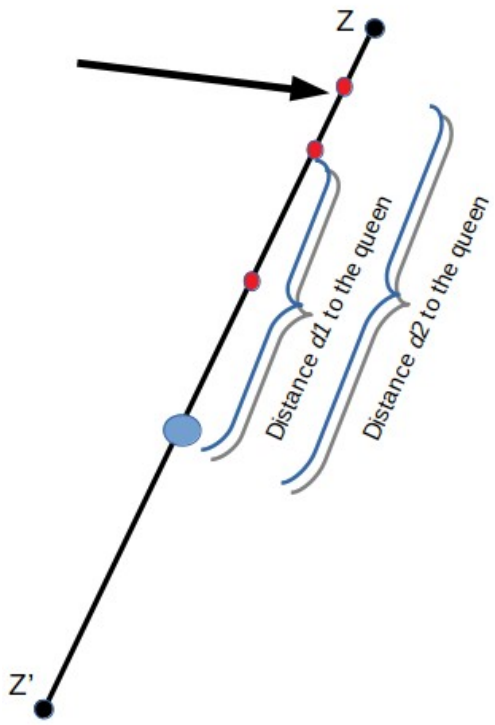
For the 1st obstacle:



change the array, because d1 < distance from point (1E+9, 1E+9) to the queen:

nearest_obstacles:	{false, {1E+9,1E+9}}	{true, {x_o1, y_o1}}	{false, {1E+9,1E+9}}
	0	i	7

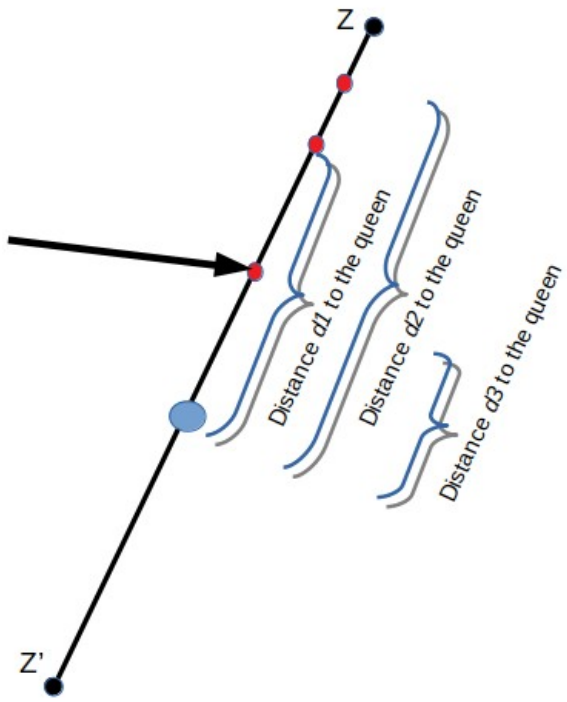
For the 2nd obstacle:



unchanged, because d2 > d2

nearest_obstacles:	{false, {1E+9,1E+9}}	{true, {x_o1, y_o1}}	{false, {1E+9,1E+9}}
	0	i	7

For 3rd obstacle:



changed, because d3 < d2

nearest_obstacles:	{false, {1E+9,1E+9}}	{true, {x_o3, y_o3}}	{false, {1E+9,1E+9}}
	0	*i*	*7*

C++ function that compute the nearest obstacle of each point of all lines segments

```
double compute_distance(int x1, int y1, int x2, int y2) {
  return sqrt(pow((x1-x2), 2) + pow((y1-y2), 2));
}

void manage_obstacles(int x, int y, int xo, int yo, int i, vector<pair<bool, pair<int, int>>> & nearest_obstacle){

  double d_qo = compute_distance(x, y, xo, yo);
  double d_q_no = compute_distance(nearest_obstacle[i].second.first, nearest_obstacle[i].second.second, xo, yo);
  if (d_qo < d_q_no) {
    nearest_obstacle[i].first = true;
    nearest_obstacle[i].second = {xo, yo};
  }
}
```

C++ code to know on each part of the segment an obstacle is it

```
  vector<pair<bool, pair<int, int>>> nearest_obstacle(8,
make_pair(false, make_pair(1E+9, 1E+9)));
  for (int i = 0 ; i < k ; ++i){
    int xo = obstacles[i].first;
    int yo = obstacles[i].second;

    // obstacle on EF (same x)
    if (xo == xe) {
      // E side
      if (yo > y)
        manage_obstacles(x, y, xo, yo, 0, nearest_obstacle);
      // F side
      else
        manage_obstacles(x, y, xo, yo, 1, nearest_obstacle);
    }

    // obstacles on GH (same y)
    if (yo == yg) {
      // H side
      if (xo > x)
        manage_obstacles(x, y, xo, yo, 3, nearest_obstacle);
      // G side
      else
        manage_obstacles(x, y, xo, yo, 2, nearest_obstacle);
    }

    // obstacles on AB
    int x_tmp = (yo - b_ab) / a_ab;
    if (x_tmp == xo) {
      // B side
      if (yo > y)
        manage_obstacles(x, y, xo, yo, 5, nearest_obstacle);
      // A side
      else
        manage_obstacles(x, y, xo, yo, 4, nearest_obstacle);
    }

    // obstacles on CD
    x_tmp = (yo - b_cd) / a_cd;
    if (x_tmp == xo) {
      // C side
      if (yo > y)
        manage_obstacles(x, y, xo, yo, 6, nearest_obstacle);
        // D side
      else
        manage_obstacles(x, y, xo, yo, 7, nearest_obstacle);
    }
  }
```

Reducing the number of free cells that are blocked by an obstacle

For each point (E, F, G, H, A, B, C and D) reduce the all free cells from the nearest obstacle to that point

C++ code:

```
// Reduce from the number of squares the number of non reachable squares
if (nearest_obstacle[0].first) ans -= (abs(ye-nearest_obstacle[0].second.second) + 1);
if (nearest_obstacle[1].first) ans -= (abs(yf-nearest_obstacle[1].second.second) + 1);
if (nearest_obstacle[2].first) ans -= (abs(xg-nearest_obstacle[2].second.first) + 1);
if (nearest_obstacle[3].first) ans -= (abs(xh-nearest_obstacle[3].second.first) + 1);
if (nearest_obstacle[4].first) ans -= (abs(xa-nearest_obstacle[4].second.first) + 1);
if (nearest_obstacle[5].first) ans -= (abs(xb-nearest_obstacle[5].second.first) + 1);
if (nearest_obstacle[6].first) ans -= (abs(xc-nearest_obstacle[6].second.first) + 1);
if (nearest_obstacle[7].first) ans -= (abs(xd-nearest_obstacle[7].second.first) + 1);
```

Why +1? to reduce the obstacle it self :)

Whole C++ code:

```
#include <bits/stdc++.h>

using namespace std;
```

```
double compute_distance(int x1, int y1, int x2, int y2) {
  return sqrt(pow((x1-x2), 2) + pow((y1-y2), 2));
}
```

```
void manage_obstacles(int x, int y, int xo, int yo, int i, vector<pair<bool, pair<int,
int>>> & nearest_obstacle){

  double d_qo = compute_distance(x, y, xo, yo);
  double d_q_no = compute_distance(nearest_obstacle[i].second.first,
nearest_obstacle[i].second.second, xo, yo);
  if (d_qo < d_q_no) {
    nearest_obstacle[i].first = true;
    nearest_obstacle[i].second = {xo, yo};
  }
}
```

```
int queens_attack(int n, int k, int x, int y, vector<pair<int, int>> obstacles) {
  // ************************1- Compute the points' coordinates of all segemnts***

  // --Segment EF---------------
  // Point E
  int xe = x;
  int ye = n;

  // Point F
  int xf = x;
  int yf = 0;
  // ----------------------------

  //--Segment GH---------------
  // Point G
  int xg = 0;
  int yg = y;

  // Point H
  int xh = n;
  int yh = y;
  // ----------------------------

  // --Segment AB (diagonal)----
  // Compute equation for AB ( y = ax + b )
  int a_ab = 1;
  int b_ab = y - x;

  // Point A
  int xa = 0;
  int ya = b_ab;
  if (ya < 0) {
    ya = 0;
    xa = -b_ab/a_ab;
  }

  // Point B
  int xb = n;
  int yb = a_ab * xb + b_ab;
  if (yb > n) {
    yb = n;
    xb = (yb-b_ab) / a_ab;
  }
  // ----------------------------

  // --------Segment CD (diagonal)----
  // Compute equation for CD ( y = ax + b )
  int a_cd = -1;
  int b_cd = y + x;

  // Point C
  int yc = n;
  int xc = (yc-b_cd) / a_cd;
  if (xc < 0) {
    xc = 0;
    yc = b_cd;
  }
  if (xc > n) {
    xc = n;
    yc = a_cd * xc + b_cd;
  }

  // Point D
  int yd = 0;
  int xd = b_cd;
  if (xd > n) {
    xd = n;
```

```
  yd = a_cd * xd + b_cd;
}

// ----------------------------
// ********************************************************************************
```

```
  // 2- For k = 0, Compute number of squares that the queen can attack from
  // position (rq, cq)*****************************************************
  int ans = abs(ye-yf) + abs(xg-xh) + abs(xc-xd) + abs(xa-xb);
  // ***********************************************************************


  // ************************3- For each obstacle: ************************
  // ********************** - Determine in which segment is it*************
  // ********************** - Determne in which part of the segment is it**
  // ********************** - In each segment, compute the nearest obstacle to the queen***

  vector<pair<bool, pair<int, int>>> nearest_obstacle(8, make_pair(false, make_pair(1E+9,
1E+9)));
  for (int i = 0 ; i < k ; ++i){
    int xo = obstacles[i].first;
    int yo = obstacles[i].second;

    // obstacle on EF (same x)
    if (xo == xe) {
      // E side
      if (yo > y)
        manage_obstacles(x, y, xo, yo, 0, nearest_obstacle);
      // F side
      else
        manage_obstacles(x, y, xo, yo, 1, nearest_obstacle);
    }

    // obstacles on GH (same y)
    if (yo == yg) {
      // H side
      if (xo > x)
        manage_obstacles(x, y, xo, yo, 3, nearest_obstacle);
      // G side
      else
        manage_obstacles(x, y, xo, yo, 2, nearest_obstacle);
    }

    // obstacles on AB
    int x_tmp = (yo - b_ab) / a_ab;
    if (x_tmp == xo) {
      // B side
      if (yo > y)
        manage_obstacles(x, y, xo, yo, 5, nearest_obstacle);
      // A side
      else
        manage_obstacles(x, y, xo, yo, 4, nearest_obstacle);
    }

    // obstacles on CD
    x_tmp = (yo - b_cd) / a_cd;
    if (x_tmp == xo) {
      // C side
      if (yo > y)
        manage_obstacles(x, y, xo, yo, 6, nearest_obstacle);
        // D side
      else
        manage_obstacles(x, y, xo, yo, 7, nearest_obstacle);
    }
  }
//*************************************************************************
```

```
  // Reduce from the number of squares the number of non reachable squares
  if (nearest_obstacle[0].first) ans -= (abs(ye-nearest_obstacle[0].second.second) + 1);
  if (nearest_obstacle[1].first) ans -= (abs(yf-nearest_obstacle[1].second.second) + 1);
  if (nearest_obstacle[2].first) ans -= (abs(xg-nearest_obstacle[2].second.first) + 1);
  if (nearest_obstacle[3].first) ans -= (abs(xh-nearest_obstacle[3].second.first) + 1);
  if (nearest_obstacle[4].first) ans -= (abs(xa-nearest_obstacle[4].second.first) + 1);
  if (nearest_obstacle[5].first) ans -= (abs(xb-nearest_obstacle[5].second.first) + 1);
  if (nearest_obstacle[6].first) ans -= (abs(xc-nearest_obstacle[6].second.first) + 1);
  if (nearest_obstacle[7].first) ans -= (abs(xd-nearest_obstacle[7].second.first) + 1);

  return ans;
}
```

```
int main(){
  int n, k;
  cin >> n >> k;

  int rq, cq;
  cin >> rq >> cq;

  int x = --rq;
  int y = --cq;
  n--;


  vector<pair<int, int>> obstacles;
  for (int i = 0; i < k; i++) {
    int ro, co;
    cin >> ro >> co;
    ro--;
    co--;
    obstacles.push_back({ro, co});
  }

  int result = queens_attack(n, k, x, y, obstacles);

  cout << result << "\n";

  return 0;
}
```