COMP1511 Problems and Notes

and some other fun stuff

J. Lam

About this book

Acknowledgements

The questions came from various sources. Some were past exercises or exam questions from UNSW, and some were past competition questions from the Australian Informatics Olympiad. Where possible, the acknowledgement has been made in the question.

About the authors

Contents

1 Mathematical Problems **4**
- 1.1 Snap Dragons II (AIO 2015) . . . 6
- 1.2 Chocolate Shop (AIO 2008) . . . 8
- 1.3 Missing Mango (AIO 2017) . . . 9
- 1.4 Culture (AIO 2003) . . . 10
- 1.5 Halloween (AIO 2002) . . . 12
- 1.6 Posters (AIO 2012) . . . 12
- 1.7 Inverse Modulo . . . 14

2 Pattern Drawing **15**
- 2.1 Hash A . . . 15
- 2.2 Draw X . . . 17
- 2.3 Hollow and Filled Diamonds . . . 18
- 2.4 Spiral . . . 20
- 2.5 Decimal Spiral . . . 22

3 Arrays **23**
- 3.1 Castle Cavalry (AIO 2018) . . . 23
- 3.2 Tag (AIO 2017) . . . 26
- 3.3 Doors Problem . . . 29
- 3.4 Atlantis Rising . . . 29

4 Characters and Strings **32**
- 4.1 Case Conversion . . . 32
- 4.2 Strings . . . 34

5 Linked Lists **35**
- 5.1 Edge Cases . . . 36
- 5.2 List Reversal . . . 36
- 5.3 Josephus Problem . . . 37

6 Grid Problems **39**
- 6.1 Atlantis (AIO 2006) . . . 39
- 6.2 Cartography . . . 44

1 Mathematical Problems

Most of the exercises and problems in this chapter are more mathematical in nature, and do not require as much programming knowledge. In particular, these questions will only require the use of

- Simple mathematical operations (addition, multiplication, modulus)
- If statements
- Basic loops

In addition, these questions (especially the ones from the informatics olympiad) require you to read input files and write output files. Whilst you are expected to know how to do these (covered in chapter XX), solution templates are provided for these questions which perform all the input and output for the student.

Some questions make use of mathematical formulae and other properties which simplifies the question. In particular, the following formula might be useful:

$$1 + 2 + \cdots + n = \frac{n(n-1)}{2}.$$

Mathematical Operations

- Addition and subtraction
- Use `*` for multiplication
- Use `a/b` for $\left\lfloor \frac{a}{b} \right\rfloor$
- Use `a % b` for $a \pmod{b}$

If - Else if - Else

```
if (condition 1) {
    // do something
} else if (condition 2) {
    // do something
} else if (condition 3) {
    // do something
} else {
    // do something
}
```

`<`	less than	first operand is less than second
`<=`	less than or equal to	first operand is less than or equal to the second
`>`	greater than	first operand is greater than second
`>=`	greater than or equal to	first operand is greater than or equal to the second
`==`	equal to	both operands equal to each other
`!=`	not equal to	first operand is not equal to second
`&&`	and	both true
`||`	or	at least one true
`!`	not the operand is false	

De Morgan's Laws

Recall the following laws from set theory:

$$(A \cup B)^c = A^c \cap B^c \text{ and } (A \cap B)^c = A^c \cup B^c.$$

These can be used in C:
`!(a && b)` which is equivalent to `(!a || !b)`
`!(a || b)` which is equivalent to `(!a && !b)`

Powers

Powers of numbers can be obtained by using the `math.h` library and using `pow(a, b)` to calculate a^b. It is recommended the library is used where allowed, as the library includes optimisation tricks to calculate large powers. A simple solution would be to multiply repeatedly:

```
int pow(int a, int b) {
    for (int i = 0; i < b; i++) {
        ans = ans * a;
    }
return ans;
}
```

Powers using bit shifting

For the special case of $b \equiv 0 \pmod 2$, we can apply bit shifting to obtain powers. For example, if we want to find 2^n, then we can use `2 << n`.

Macro Loop Shortcut

The for loop idiom is so commonly used, it is often helpful to use a macro to expand it

```
#define FOR(i,a,b) for (int i = (a); i < (b); i++)
#define TFOR(i,a,b) for (int i = a; i >= b; i--)
```

This is effectively saying 'loop the variable `i` from a to b'.

1.1 Snap Dragons II (AIO 2015)

Input File: `snapin.txt`
Output File: `snapout.txt`
Time Limit: 1 second

Have you ever heard of Melodramia, my friend? It is a land of magic forests and mysterious swamps, of sprinting heroines and dashing heroes. And it is home to two dragons, Rose and Scarlet, who, despite their competitive streak, are the best of friends.

Rose and Scarlet love playing snap tag, a game for two players on an grid. The game goes as follows:

- The two dragons start on different squares.
- It's Rose's turn first. On her turn she must move to an adjacent square (i.e. she must make one step left, right, up, or down).
- It's Scarlet's turn next. On her turn she must move to an adjacent square.
- The two dragons continue alternating turns...
- ...until one dragon lands on the same square as another. When this happens, the dragon who moved last shouts 'Snap tag!' and wins the game.
- Rose and Scarlet are both snap tag experts and always find a winning strategy if one exists. If it is not possible for either player to gain the upper hand, then the game goes on forever.

In this task, you are given the size of the grid and the starting locations of both dragons. You must write a program to determine how the game ends: Does Rose win? Does Scarlet win? Does the game go on forever?

Input
The input file will contain six space separated integers on a single line, in the format:

$$R\ C\ r_{\text{ROSE}}\ c_{\text{ROSE}}\ r_{\text{SCARLET}}\ c_{\text{SCARLET}}$$

where:

- R and C are the number of rows and columns in the grid, respectively;
- r_{ROSE} and c_{ROSE} are the row and column of Rose's starting square (rows are numbered 1 to R from top to bottom; columns are numbered 1 to C from left to right); and
- r_{SCARLET} and c_{SCARLET} are the row and column of Rose's starting square.

Output
Output should consist of a single upper-case word with no punctuation.

If Rose can guarantee herself a win, output ROSE. If Scarlet can guarantee herself a win, output SCARLET. If neither player can guarantee a win, output DRAW. **Sample Input 1**

```
2 3 1 1 2 3
```

Sample Output 1

```
ROSE
```

Explanation
In the first example, Rose can guarantee herself a win if she is clever and cautious.

On her first move, Rose steps to the right. Then, no matter whether Scarlet goes up or left on her turn, Rose can tag her the next turn.

Sample Input 2 5 1 2 1 4 1 Sample Output 2 SCARLET

Explanation

In the second example, Scarlet can guarantee herself a win if she is clever and cautious.

If Rose moves down, then Scarlet will tag her on her very next turn. Otherwise, if Rose moves up, then Scarlet moves up and Rose will be forced to move down next to Scarlet, who will tag her the next turn.

Sample Input 3 15 15 3 5 12 13 Sample Output 3 ROSE

Sample Solution

```
FILE *in = fopen("snapin.txt", "r");
FILE *out = fopen("snapout.txt", "w");
int R, C, xa, ya, xb, yb;
fscanf(in, "%d %d %d %d %d %d", &R, &C, &xa, &xb, &ya, &yb);

if ((xa+xb+ya+yb)%2==0) {
    fprintf(out, "SCARLET");
} else {
    fprintf(out, "ROSE");
}
```

1.2 Chocolate Shop (AIO 2008)

Input File: `chocin.txt`
Output File: `chocout.txt`

Each weekend you work in a local chocolate shop. The chocolates are stored in boxes with ten chocolates in each box. Each time a customer comes in to buy some chocolates you take them from an already-open box. If there are not enough chocolates in the current box, you completely finish the box before opening a new box. You are not allowed to have any more than one box open at a time, and you cannot open a new box unless a customer has purchased some of the chocolates in that box.

The chocolates in this shop are particularly famous, so every person who comes in buys at least one chocolate. However, as they are very expensive, no one ever buys more than nine.

At the start of the day, there are no open boxes. If at the end of the day the currently open box has chocolates left in it, you are allowed to take them home and share them with your friends (or just eat them yourself if you are feeling greedy). Your task is to write a program that reads in the series of purchases for the day and prints how many chocolates you will get at the end.

Input
The first line of the input file will contain a single integer N, the number of purchases that have taken place during the day ($0 \leq N \leq 10000000$). Following this will be N lines listing the day's purchases in the order they took place. Each of these lines will contain a single digit between 1 and 9, inclusive.

Output
Your output file should contain a single integer, giving the number of chocolates you will take home at the end of the day.

Sample Input

```
10
2
9
4
7
8
3
2
5
3
4
```

Sample Output

```
3
```

Sample Solution

```
int main() {
    FILE *in = fopen("chocin.txt", "r");
    FILE *out = fopen("chocout.txt", "w");
    int N; int answer;
    fscanf(in, "%d", &N);
    int i, sum = 0;

    for (i = 0; i < N; i++) {
        int chocs;
        fscanf(in, "%d", &chocs);
        sum += chocs;
    }

    answer = sum % 10;
    if (answer == 0) {
        answer = 10;
    }
    fprintf(out, "%d", 10-answer);
    fclose(in);
    fclose(out);
    return 0;
}
```

This problem is more mathematical than programmy.

1.3 Missing Mango (AIO 2017)

Input File: `manin.txt`
Output File: `manout.txt`
Time Limit: 1 second

Ishraq and Clement have lost their mango and need help finding it again! Ishraq and Clement are standing on a straight line. Their positions are denoted by a single integer specifying how far they are from the left-hand end of the line. The mango is also on the line, but its exact location is unknown. Both Ishraq and Clement know how far away from the mango they are, but they don't know what direction the mango is in. Given the location of Ishraq and Clement, and their respective distances to the mango, determine the location of the mango. It is guaranteed that there will always be a single possible location for the mango.

Input
The first and only line of input will contain the four non-negative integers I_x, C_x, I_d, and C_d, representing Ishraq's location, Clement's location, Ishraq's distance to the mango, and Clement's distance to the mango, respectively.

Output
Output should be a single integer, the location of the mango. You should note that this value might be less than zero.

Sample Input and Output 1

```
2 6 1 3
```

```
3
```

Sample Input and Output 2

```
8 3 7 2
1
```

Sample Input and Output 3

```
23 40 17 0
40
```

Sample Solution

```c
#include <stdio.h>

int main (void) {
    int a,b,c,d, answer;
    FILE *input_file = fopen("manin.txt", "r");
    FILE *output_file = fopen("manout.txt", "w");
    fscanf(input_file, "%d %d %d %d", &a, &b, &c, &d);
    if (a-c == b-d || a-c == b+d) {
        answer = a-c;
    } else {
        answer = a+c;
    }
    fprintf(output_file, "%d", answer);
    return 0;
}
```

1.4 Culture (AIO 2003)

Input File: `cultin.txt` Output File: `cultout.txt` Time Limit: 1 second

You are a biologist working in a large laboratory. For the last month you have been growing a culture of your favourite bacteria, Bacillus Fortranicus. You are particularly interested in the way in which it grows in hostile environments.

Today is the last day of your experimentation. With anticipation you pull your log book from the shelf, but in your excitement you knock a bottle of acid from the bench. You watch in despair as it spills across your log book and your precious notes dissolve before your eyes.

You try desperately to recall some statistics. How many individual bacteria did you begin with? You can't even remember for how many days the experiment has been running. In desperation you call over your lab assistant.

"No, I don't remember how many bacteria we began with either," she says. "But I do remember that it was an odd number. Oh yes, and the number of bacteria doubled each day." She looks down at the bench, sniffs the acid and walks back to her desk with a wrinkled nose.

Although you have lost your notes, you can still count the total number of bacteria that you have now. Combining this total with the assistant's information, you must write a program to answer your two original questions. That is, you must calculate (i) how many bacteria you began with, and (ii) for how many days the experiment has been running.

Input

The input file will consist of one line only. This line will contain a single integer n representing the number of bacteria that you have now. You are guaranteed that $1 \leq n \leq 30,000$.

Output

The output file must consist of the two integers b and d on a single line, where b represents the number of bacteria at the beginning of the experiment and d represents the number of days for which the experiment has been running. These two integers must be separated by a single space.

Sample Input

```
136
```

Sample Output

```
17 3
```

The sample data above can be explained as follows. We are given that the final bacteria count is 136. Observe that $136 = 17 \times 2 \times 2 \times 2$. Since 17 is odd, we see that the initial bacteria count was 17. Furthermore, the bacteria count has doubled three times and so the experiment must have been running for precisely three days.

```
int d = 0;
while (a % (2**d)==0) {
    d++;
}

d--;
int b = a/(2**d); // rounds down since int
printf("%d %d", b, d);
```

1.5 Halloween (AIO 2002)

Input File: `hallin.txt`
Output File: `hallout.txt`
Time Limit: 0.333 second

October 31 is approaching. As Director of Operations for trick-or-treating in your suburb, it is your job to determine precisely which children should visit which houses dressed in scary costumes demanding free food.

Intelligence from your market research team indicates that adults in your neighbourhood tend to behave as follows. When the first child knocks on the door, (s)he gives them one lolly. When the second child knocks on the door, (s)he gives them two lollies. The third child is given three lollies and so on, giving more lollies to each successive child in the hope that the lollies will run out faster so (s)he can bolt the door and go to bed.

Your surveillance team has reported precisely how many lollies each adult has purchased over the last week. Your task is to determine how many children to send to each house.

Input
Each input file will contain a single integer n representing the number of lollies a particular adult has. You are guaranteed that $0 \leq n \leq 10,000$.

Output
The output file must contain a single line representing the first dissatisfied child (the child who doesn't receive as many lollies as they expect).

Example
Say the input file indicates an adult with 12 lollies. In this case the first child receives 1 lolly, the second child receives 2, the third receives 3 and the fourth receives 4. At this point the adult has given away $1 + 2 + 3 + 4 = 10$ lollies.

The fifth child should receive 5 lollies, but the adult has only 2 lollies remaining. Thus the fifth child will not receive as many lollies as they expect.

Sample Input
```
12
```
Sample Output
```
5
```

Sample Solution

```
int n;
fscanf(in, "%d", &n);
int i = 1;
while ((i*(i+1))/2 <= n) {
        i++;
}
fprintf(out, "%d", i);
```

1.6 Posters (AIO 2012)

Input File: `postin.txt`
Output File: `postout.txt`

Time Limit: 1 second

You run a poster advertisement company. Your company is quite small: all it owns is a rectangular wall in the city. Advertisers pay to put up their poster on your wall at some time at some position along the wall. These posters may have different widths, but are all exactly the same height as the wall. When a poster is put up, it may cover some of the posters already on the wall.

You have a log of every single poster put on your wall: their distance from the left end, their width, and the time they were put up.

Since people always walk from the left to the right of your wall, and we all know advertisement posters are only effective when they are completely uncovered (e.g. a burger picture will only be mouthwatering if you see the whole burger), you would like to determine the leftmost fully visible poster.

Input

The first line will contain a single integer N, the number of posters that have been put up. The next N lines will contain descriptions of the posters in chronological order (from earliest to most recent). The ith of these lines will contain two integers x_i and w_i, which indicate the distance from the left end and the width of poster i respectively. (All units are in metres.)

Output

Output should consist of a single integer: the index i of the leftmost, completely visible poster (where the index of the first poster is 1).

Sample Input 1

```
4
1 5
3 7
2 6
8 9
```

Sample Input 2

```
7
40 50
30 45
1 10
20 30
1 30
5 15
10 40
```

Sample Output 1: `3`
Sample Output 2: `7`

Constraints

All of the test cases will adhere to the following bounds:

- $1 \leq x_i \leq 1,000,000$
- $1 \leq w_i \leq 10,000$
- $1 \leq N \leq 100,000$

As some of the test cases will be quite large, you may need to think about how well your solution scales for larger input values.

Sample Solution

```c
#include <stdio.h>

int main() {
    FILE *in = fopen("postin.txt", "r");
    FILE *out = fopen("postout.txt", "w");
    int N, i;
    fscanf(in, "%d", &N);
    int currentX, currentY, answer=1;
    fscanf(in, "%d %d", &currentX, &currentY);
    for (i = 2; i <= N; i++) {
        int X, Y;
        fscanf(in, "%d %d", &X, &Y);
        if (X < currentX + currentY) {
            currentX = X;
            currentY = Y;
            answer = i;
        }
    }
    fprintf(out, "%d", answer);
    fclose(in);
    fclose(out);
    return 0;
}
```

1.7 Inverse Modulo

Write a program which calculates the multiplicative inverse of a number mod b. The multiplicative inverse of a number a mod b is the integer x $(0 < x < m)$ such that $ax \equiv 1 \pmod{m}$.

Input is of the form `a b`.

Output is to be a single integer. If no inverse exists, the output should be `NONE`.

```c
#include <stdio.h>
int main (void) {
    int a, b;
    scanf("%d %d", &a, &b);
    int ans = -1;
    for (int x = 1; x < b; x++) {
        if (a*x % b == 1) {
            ans = x; break;
        }
    }
    if (ans == -1) {
        printf("NONE\n");
    } else {
        printf("%d\n", ans);
    }
    return 0;
}
```

2 Pattern Drawing

This section is more fun

- Split into cases (odd, even) or separate by rows
- Exploit symmetry (reverse the loop)
- Find a function of a coordinate
- Use recursion/dynamic programming

2.1 Hash A

File name: `hash_a.c`
Input: Standard input
Output: Standard output

Write a program that prints out the letter A of size $n \times n$ using hash characters (#).

Sample Input and Output

```
./hash_a
4
####
#  #
####
#  #

./hash_a
5
#####
#   #
#####
#   #
#   #
```

```
./hash_a
6
######
#    #
#    #
######
#    #
#    #
```

Constraints For all test cases, $3 < n \leq 100$.

Discussion
These pattern drawing style questions are often very simple in terms of the programming knowledge required - often a simple loop and if statement is all you'll need. What makes some of these challenging is determining mathematically what sort of conditions are required.

Fortunately, it looks like these A-shaped patterns only contain two possible patterns of lines: either the entire row is filled with hashes, or only the first and last characters are hashes. Since there are `n` rows, the code begins to look like this:

```
for (int i = 0; i < n; i++) {
    if ( first line || middle line ) {
        // print full row
```

```
    } else {
        printf("#");
        for (int j = 0; j < n-2; j++) {printf(" ");}
        printf("#");
    }
    printf("\n");
}
```

So now, how do we print a full row of hashes? This is easily done, as we know there are n hashes to print. A for loop will work:

```
for (int j = 0; j < n; j++) {
    printf("#");
}
```

But how about

This solution provides a very natural approach for this problem. Another approach is to write a function `printA(x,y)` which takes coordinates (or indices) as input and calculates whether a hash or space is to be displayed. The main body might look like this:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        printA(i, j);
    }
    printf("\n");
}
```

So what might the function `printA` look like? We can start with the two rows completely filled with hashes.

```
void printA (int x, int y) {
    if (y == 0 || y == ceil) {
        printf("#");
    } else if (x == 0 || x == n) {
        printf("#");
    } else {
        printf(" ");
    }
}
```

2.2 Draw X

Write a program called x.c that reads an integer n from standard input. and prints an $n \times n$ pattern of asterisks and dashes in the shape of an "X".

You can assume n is odd and $n \geq 5$.

Sample Input and Output

```
Enter size: 5
*---*
-*-*-
--*--
-*-*-
*---*

Enter size: 9
*-------*
-*-----*-
--*---*--
---*-*---
----*----
---*-*---
--*---*--
-*-----*-
*-------*
```

```
Enter size: 15
*-------------*
-*-----------*-
--*---------*--
---*-------*---
----*-----*----
-----*---*-----
------*-*------
-------*-------
------*-*------
-----*---*-----
----*-----*----
---*-------*---
--*---------*--
-*-----------*-
*-------------*
```

Sample Solution 1

```
int main(void) {
    int size;
    printf("Enter size: ");
    scanf("%d", &size);
    int row=1;
    while (row <= size) {
        int column=1;
        while (column <= size) {
            if ((column==row) || (column == size-row+1)) {
                printf("*");
            } else {
                printf("-");
            }
            column++;
        }
        printf("\n");
        row++;
    }
    return 0;
}
```

A dynamic programming approach

Dynamic programming involves using recursion to break the diagram into something easier. We start off by using the same `main` function in the previous question, but writing our own `printX` function.

2.3 Hollow and Filled Diamonds

Write two programs that prints out a diamond with length n. One program should draw a hollow diamond, and the other should draw a filled diamond. You may assume $2 \leq n \leq 100$.

Sample Input and Output

(Note: Your program only needs to take a single integer as input, and print the pattern only. The table layout used in this book is to save space when printed on paper)

```
n=3 (hollow)        (filled)
    *                  *
   * *                 ***
  *   *               *****
   * *                 ***
    *                  *

n=5 (hollow)        (filled)
    *                  *
   * *                 ***
  *   *               *****
 *     *             *******
*       *           *********
 *     *             *******
  *   *               *****
   * *                 ***
    *                  *
```

Sample Solution - Hollow Diamond

```
int main (int argc, char *argv[]) {
    int n; scanf("%d", &n);
                // diamond will be of size n*2 - 1
    n = n*2 - 1; // (bc the length of a side of the diamond is n)

    // Idea is similar to asterisks, need nested loop
    // We need to figure out the formula for each quadrant of the diamond
    int row = 0;
    while (row < n) {
        int col = 0;
        while (col < n) {
            // top two quadrants (left and right, respectively)
            if (row == n/2-col || row == col - n/2) {
                printf("*");
            }
            // bottom two quadrants (left and right, respectively)
            else if (row == n/2 + col || row == 3*n/2 - col- 1) {
                printf("*");
            }
            else printf(" ");
            col ++;
        }
        printf("\n"); row ++;
    } return 0;
```

```
}
```

Sample Solution - Filled Diamond

The solution is mostly the same as for hollow diamond, except for the if-statement conditions. We need to change our conditions to be inequalities, rather than equalities. We will also need to bound each quadrant (otherwise, when we apply the inequalities, we will just get a box of stars rather than a diamond).

```
if ((col <= n/2 && row <= n/2 && row >= n/2-col)
|| (col >= n/2 && row <= n/2 && row >= col - n/2)) {
    printf("*");
} else if ((col<=n/2 && row >= n/2 && row <= n/2 + col)
|| (col >= n/2 && row >= n/2 && row <= 3*n/2 - col- 1)) {
    printf("*");
} else printf(" ");
```

2.4 Spiral

This is doable using loops, although we will also explore alternative techniques.

Sample Solution 1

```
#include <stdio.h>

int main(void) {
    int size;
    printf("Enter size: ");
    scanf("%d", &size);
    // first half
    for (int row = 1; row <= (size+1)/2; row++) {
        // For each line (top half of the box)
        for (int col = 1; col <= size; col++){
            //For each char in line
            if (row % 2 == 1) {
            // if odd row
                if (col%2==0 && !(col>row-2 && col<= size-row)) {
                    printf("-");
                }
                else {
                    printf("*");
                }
            } else {
            // even rows
                if (col%2==1 && !(col>row-3 && col< size-row+1)) {
                    printf("*");
                } else {
                    printf("-");
                }
            }
        }
        printf("\n");
    }

    for (int row = (size+1)/2-1; row >= 1; row--) {
        // For each line (top half of the box)
        for (int col = 1; col <= size; col++){
            //For each char in line
            if (row % 2 == 1) {
            // if odd row
                if (col%2==0 && !(col>row-1 && col<= size-row)) {
                    printf("-");
                }
                else {
                    printf("*");
                }
            } else {
            // even rows
                if (col%2==1 && !(col>row-1 && col< size-row+1)) {
```

```
                    printf("*");
                } else {
                    printf("-");
                }
            }

        }
        printf("\n");
    }
    return 0;
}
```

Sample Solution 2

```
#include <stdio.h>

void draw (int x, int y, int size) {
    if (y == 0 || y == size-1) {
        printf("*");
    } else if (y == 1) {
        if (x == size-1) {
            printf("*");
        } else {
            printf("-");
        }
    } else if (y == size-2) {
        if (x == 0 || x == size-1) {
            printf("*");
        } else {
            printf("-");
        }
    } else if (x == 0 || x == size-1) {
        printf("*");
    } else if (x == size-2) {
        printf("-");
    } else if (x == 1) {
        if (y == 2) {
            printf("*");
        } else {
            printf("-");
        }
    } else {
        draw (x-2, y-2, size-4);
    }
}

int main(void) {
    int size;
    printf("Enter size: ");
    scanf("%d", &size);
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
```

```
            draw(j, i, size);
        }
        printf("\n");
    }
    return 0;
}
```

This approach uses only a nested for loop in the main function, and pushes the responsibility of doing the drawing onto `draw` function. It takes a coordinate, as well as the size of the box. The draw function also only actually draws the outer two edges. The rest of the box is drawn using recursion.

2.5 Decimal Spiral

Exploit symmetry! When we do this, we halve the amount of work we have to do, because all we need to do is take our previous efforts, and cleverly change our for loop to reverse it. This trick is especially useful for patterns which have some sort of symmetry in them.

3 Arrays

3.1 Castle Cavalry (AIO 2018)

Input File: `cavalryin.txt`
Output File: `cavalryout.txt`
Time Limit: 1 second

As the Queen's chief technologist, you have been tasked with organising the army's newest cutting edge1 division: the cavalry.

Naturally, the Queen is sceptical, so to prove it's worth it you are going to conduct a quick field test. Firstly, you will need to group your knights into squads.

Unfortunately, the N knights in your division are very inexperienced, having only been training for two weeks! The ith knight (counting from 1) has told you that they would only be comfortable in a squad containing exactly a_i knights.

You can make as many or as few squads as you like of any size, so long as every knight is comfortable.

After feeding your whining, whinnying horses their pheasant-based supper, you return to your lonely lodge to determine if it is possible to divide up your cavalry.

Input
The first line of input will contain N, the number of knights in your division.
Then, N lines will follow. The ith line (counting from 1) will contain a_i, the size of the squad the ith knight wants to join.

Output
Print YES on a single line if it is possible to put the knights into squads such that they are all comfortable. If it is not possible, then print NO instead.

Sample Input and Output 1

```
5
2
3
2
3
3
YES
```

Sample Input and Output 2

```
3
2
2
2
NO
```

Sample Input and Output 3

```
6
2
2
2
2
2
2
YES
```

Explanation

In the first case, you can put knights 1 and 3 in one squad, and knights 2, 4 and 5 into a second one. This makes them all comfortable, so the answer is YES.

In the second case, you can put knights 1 and 2 together in the same squad, but then knight 3 cannot form a squad by themselves (since knight 3 wants to be in a squad of size 2). No matter what you do, one of the knights is going to get left out, so the answer is NO.

Constraints For all test cases, $1 \leq N \leq 100,000$ and $1 \leq a_i \leq N$.

Sample Solution

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    FILE *in = fopen("cavalryin.txt", "r");
    FILE *out = fopen("cavalryout.txt", "w");

    int N;
    fscanf(in, "%d", &N);

    int *sizes = malloc((N+1)*sizeof (int));

    for (int i = 0; i <= N; i++) {
        sizes[i] = 0;
    }

    for (int i = 0; i < N; i++) {
        int a;
        fscanf(in, "%d", &a);
        sizes[a] = sizes[a]+1;
    }

    int check = 1;
    for (int i = 1; i <= N; i++) {
        if (sizes[i] % i != 0 && sizes[i] != 0) {
            check = 0;
        }
    }

    if (check == 1) {
        fprintf(out, "YES\n");
    } else {
        fprintf(out, "NO\n");
    }
    return 0;
}
```

Discussion

Again, we should form some sort of strategy that provides us our solution before implementing it.

3.2 Tag (AIO 2017)

Input File: `tagin.txt`
Output File: `tagout.txt`
Time Limit: 1 second

Thank you for being the scorer for our game of tag. The rules are pretty simple:

- There are N people playing today, and everyone has a different number between 1 and N inclusive.
- At the start of the game, player 1 is on the red team and player 2 is on the blue team. All other players start on no team.
- Over the course of the game, people on a team will tag people who are not on a team. The person who has been tagged then joins the team of the person who tagged them.
- Note that it is possible some of the players will not have been tagged by the end of the game.

As scorer, we need you to figure out how many people are on each team at the end of the game.

Input
The first line of input will contain two integers N and M: the number of players in the game and the number of tags that occur during the game, respectively. M lines will follow, describing the tags that happen in chronological order. Each line will contain two integers a and b. This indicates that player a tags player b, so now player b joins player a's team. It is guaranteed that, prior to this tag, player a is already on a team and player b is not.

Output
Your program must output two integers on one line. The number of players on the red team at the end of the match, and the number of players on the blue team at the end of the match.

Sample Input

```
8 5
2 7
1 8
8 4
7 5
8 6
```

Sample Output

```
4 3
```

Explanation

- Initially, player 1 is on the red team and player 2 is on the blue team.
- Player 2 tags player 7, now player 7 is also on the blue team. The blue team now has two players.
- Player 1 tags player 8, now player 8 is on the red team with player 1. The red team now has two players.
- Player 8 tags player 4, now player 4 is on the red team with players 1 and 8. The red team now has three players.
- Player 7 tags player 5, now player 5 is on the blue team with players 2 and 7. The blue team now has three players.

- Player 8 tags player 6, now player 6 is on the red team with players 1, 8 and 4. The red team now has four players.

At the end of this, the red team consists of four players 1, 8, 6, and 4. The blue team consists of three players 2, 7, and 5. Note that player 3 was never tagged so is on neither team at the end of the match.

Sample Solution

```
int main() {
    FILE *in = fopen("tagin.txt", "r");
    FILE *out = fopen("tagout.txt", "w");
    int N, M;
    fscanf(in, "%d %d", &N, &M);

    int team[100003] = {0};
    team[1] = 1; // RED
    team[2] = 2; // BLUE

    for (int i = 0; i < M; i++) {
        int a,b;
        fscanf(in, "%d %d", &a, &b);
        team[b] = team[a];
    }

    // Do the counting
    int countRed = 0;
    int countBlue = 0;
    for (int i = 1; i <= N; i++) {
        if (team[i] == 1) {
            countRed ++;
        } else if (team[i] == 2) {
            countBlue ++;
        }
    }

    fprintf(out, "%d %d", countRed, countBlue);
    return 0;
}
```

The algorithm uses an array to keep track of everyone's team allocation. Each person starts off with an unassigned team (array value of 0) with the exception of players 1 and 2 who are on the red and blue team respectively. These are encoded in the array using the values of 1 and 2. We then use a for loop to scan through the list of players being tagged. The loop takes the value of the 'tagger' and assigns that to the person being tagged. Finally, another loop is used to count the number of red team members (amount of '1's that are in the array) and the number of blue team members (the amount of '2's that are in the array).

3.3 Doors Problem

In a monastery in a faraway country there is a chamber with 64 doors numbered 1 to 64. Every year the 64 monks that live in the monastery participate in the following ritual.

All doors in the chamber are initially closed. Each monk is assigned a unique integer between 1 and 64. The monks then enter the chamber one at a time, in numeric order (though that doesn't affect the outcome). If a monk is assigned the integer k, he changes the state of every kth door. That is, if the door is closed he opens it and if it's open he closes it.

So the first monk changes every door (from closed to open), the second changes doors #2, #4, #6 etc, the third changes #3, #6, #9 etc, and so on until the last monk, who just changes #64.

At the end of the ritual the doors that are currently open lead to (spiritual) treasures, so their numbers are significant.

You are a maths-god so you already know the monks want the doors belonging to perfect squares for the secret maths prayers. However, your code must *simulate* the opening and closing of doors (through, perhaps, an array?) rather than just printing out the square numbers.

(a) Write a program that **simulates** the ritual and displays the numbers of the doors that are open at the end, on one line.

(b) Why is it that only square numbered doors remain open?

Solution
It is not hard to see that the door i is only opened or closed on iteration k if k divides i. Door 1 is toggled 64 times. Door 2 is toggled 32 times. Door 3 is toggled .. times

3.4 Atlantis Rising

3.4.1 Problem Statement

Life in the under-sea city of Atlantis is getting rather dull. Sure, there are talking seahorses, wish-granting clams and all kinds of exotic sea life, but ultimately nothing has really happened for the past few millennia. Rumour has it that the weather is getting nice and warm up above and now King Triton has issued a royal decree–it is time for Atlantis to resurface.

There is one slight problem however–the talking seahorses, wish-granting clams and exotic sea life who live in Atlantis are not happy at the prospect of migrating to dry land. Despite their protests, the King refuses to change his mind. As a result, the citizens scramble to migrate to areas that will remain underwater when the city resurfaces.

Atlantis is a series of flat plains with varying heights, each one unit wide as shown in the example above. When the city majestically emerges from the waves, some water may become trapped between plains, resulting in lakes where the citizens can still habitate and frolic in the cool water. Any water that can flow to the edge of Atlantis will drain away into the sea. A plain at the same height as a lake is not considered habitable by its citizens.

In the above example, once the water has settled, three units of land remain covered in seawater at positions 3, 4 and 6 from the left, and thus remain habitable. Note that although there are two units of water above position 4, this only counts as one unit of habitable land. Your task is to write a program that calculates how many habitable units of land will remain, given the shape of Atlantis.

Input
The first line of the input file will contain a single integer N, the number of units of land in all of Atlantis ($1 \leq N \leq 1000000$). The following N lines describe the shape of the city from left to right. Each line will contain a single positive integer, denoting the final height of each unit of land above sea level. You are guaranteed that all heights are integers between 1 and 2000000000, inclusive.

Output
Your output file should consist of a single integer H, the number of habitable units of land after the rising of Atlantis.

Sample Input

```
9
1
3
2
1
4
3
5
1
1
```

Sample Output

```
3
```

3.4.2 Discussion

Our first thought might be to try a direct approach. We note that each island is habitable if and only if there exists an island of greater height on both the left and the right of that island. Our algorithm might look like this:

For each island:

- Check there exists an island of greater height to the left of me
- Check there exists an island of greater height to the right of me
- If both conditions are true, lands++

This algorithm is in $O(n^2)$ time, and is extremely inefficient. Indeed, using such an algorithm in the contest will not score you full marks. We should reconsider our approach.

Instead of the direct approach of counting the habitable islands, we can consider an indirect approach of counting the uninhabitable islands, and then subtracting that from the original number of islands. It can be seen, through some careful thought, that the number of inhabitable islands is equivalent to the number of islands that are (non-strictly) increasing.

Thus our new algorithm is as follows:

- Find the (index of) island of max height
- Count the length of non-strictly increasing sequence
- Count the length of non-strictly decreasing sequence
  - We can use symmetry, and reverse the loop. We start from the outside island and work our way inside. This then becomes finding the same increasing sequence.

- Subtract these lengths from N (the total number of islands)

3.4.3 Sample Solution

```
#include <cstdio>
int main() {
    FILE *in = fopen("atlanin.txt", "r");
    FILE *out = fopen("atlanout.txt", "w");
    int N, max = 0, index_max;
    fscanf(in, "%d", &N);
    int lands = N;
    int heights[N];
    for (int i = 0; i < N; i++) {
        // read input, and then find (index of) max height
        fscanf(in, "%d", &heights[i]);
        if (heights[i] > max) {
            max = heights[i];
            index_max = i;
        }
    }

    // the left
    int currMax = 0;
    for (int i = 0; i < index_max; i++) {
        if (heights[i] >= currMax) {
            lands--;
            currMax = heights[i];
        }
    }

    // the right, working backwards
    currMax = 0;
    for (int i = N-1; i >= index_max; i--) {
        if (heights[i] >= currMax) {
            lands--;
            currMax = heights[i];
        }
    }
    fprintf(out, "%d\n", lands);
    fclose(in);
    fclose(out);
    return 0;
}
```

4 Characters and Strings

4.1 Case Conversion

This section outlines some concepts related to case conversion of characters. The idea revolves around the fact that the difference between a lower case character and its corresponding upper case character is a constant. We begin with an exercise:

Swap Case

File name: `swap_case.c`
Input: Standard input
Output: Standard output

Write a program that takes in a string (each line will contain less than 60,000 characters) and outputs the same character according to the following rules.

- If the character is upper case, convert it to lower case
- If the character is lower case, convert it to upper case
- Non-alphabetical characters remain as is.

Sample Input and Output

```
./swap_case
hey Hi HEY!
HEY hI hey!
Generic Asian Name is the coolest tutor!
gENERIC aSIAN nAME IS THE COOLEST TUTOR!
Ctrl-D
```

Sample Solution

```
int main (void) {
    char c;
    while (ch != EOF) {
        // if it is lowercase, turn to upper
        if (c >= 'a' && c <= 'z') {
            c = c - 'a' + 'A';
        }
        // if it is uppercase, turn to lower
        else if (c >= 'A' && c <= 'Z') {
            c = c - 'A' + 'a';
        }
        putchar(c);
        ch = getchar();
    }
```

```
    return 0;
}
```

Discussion

This makes use of a classic C-idiom. An idiom in programming is code for performing some simple task that is so common that it has become an underlying pattern for all similar styles of questions. Here, the idiom is for reading characters until end of input. The idiom is:

```
char ch = getchar();
while (ch != EOF) {
    // do something
    ch = getchar();
}
```

Alternatively, you can also use:

```
char c;
while ((c=getchar()) != EOF) {
    // do something
}
```

Some more common idioms associated with operations involving cases of characters involve checking for upper and lower case:

```
if (c >= 'a' && c <= 'z')
```

and converting cases: `c - 'a' + 'A'`. At no time should you ever need to use the ASCII table to look up values. Instead, use single quotes (and not double quotes).

In fact, the original implementation of the `toupper` and `tolower` functions used macros:

```
#define toupper(c) ((c) + 'A' - 'a')
#define tolower(c) ((c) + 'a' - 'A')
```

instead of functions. This is because the subroutine call overhead is much longer than the actual calculations. This implementation relies on the assumption that the difference between an uppercase character and its corresponding lowercase character is constant for all letters. This is a reasonable assumption to make and is perfectly valid for ASCII characters which is used in this course. The other assumption these 'macro functions' rely on is that they must be given a character of the appropriate case. That is, the `toupper` function must take in a lower case character is input and vice versa. This could be rewritten as a function:

```
int toupper(char c) {
    if (c >= 'a' && c <= 'z') { // is lower case character,
        return c + 'A' - 'a'; // then return upper case character
    }
    return c; // return same character, unchanged, if not lower case
}
```

This could also be implemented as a macro:

```
#define toupper(c) ((c)>='a' && (c)<='z' ? (c)+'A'-'a' : (c))
```

which is fine when the input is a constant character, but what if you had an expression like `toupper(*p++)`? This would be bad because `c` is evaluated between 1 and 3 times for each call, causing the character to change. This would not be a problem when using the function.

4.2 Strings

The term *string* refers to a variable-length array of characters. Strings are defined by a starting point and by a string-termination character marking the end.

The below implements some simply operations that we commonly perform on strings. They all involve processing the strings by scanning through them from beginning to end. Many of these functions are available in the `string` library.

- String length `strlen(a)`

```
for (i = 0; a[i] != 0; i++) ; // do nothing
return i;
```

- Copy `strcopy(a, b)`
- Concatenate `strcat(a, b)` `strcpy(a+strlen(a), b)`

5 Linked Lists

In particular, we use pointers for links (to the next node) and a struct for node itself:

```
struct node {
    Item item;
    struct node *next;
};
```

Once a node is created, how do we refer to the information it comprises - its item and its next node? We have already seen the basic operations that we need for this task. We simply dereference the pointer using `*curr` and then use the structure member names to access the item. For example, we can use `(*curr).item` or `(*curr).next`. These operations are so heavily used, however, that C provides the shorthand `curr -> item` and `curr -> next` which are equivalent forms.

Deleting nodes

To delete the node following node `curr`, we use the statements

```
struct node *temp = curr -> next;
curr -> next = temp -> next;
```

or more concisely

```
curr -> next = curr -> next -> next;
```

Inserting nodes

To insert the node `new` into a list at the position after node `curr`, we use the statements

```
new -> next = curr -> next;
curr -> next = new;
```

One of the most common operations that we perform on linked lists is to *traverse* them. That is, we start from the head and scan through the items on the list sequentially, performing some sort of operation on each. For example,

```
for (struct node *curr = head; curr != NULL; curr = curr -> next) {
    // do something
    visit(curr -> value);
}
```

or its equivalent `while` form,

```
struct node *curr = head;
while (curr != NULL) {
    // do something
    visit(curr -> value);
    curr = curr -> next
}
```

is as ubiquitous in list-processing programs as is the corresponding `for (int i = 0; i < N; i++)` in array-processing programs.

5.1 Edge Cases

Here are some common edge cases to consider when doing linked list operations.

- What happens if you are given a NULL list?
- What happens if no item is found while searching?
- What happens if you need to insert before the first term?
- What happens if the first term is what you search
- What happens if last term?

5.2 List Reversal

This function reverses the order of a linked list, returning a pointer to the final node, which the points to the second-last node, and so on. To do this, we need to set up a temporary node.

```
struct node *reverse(struct node *head) {
    struct node *temp;
    struct node *curr = head;
    struct node *prev = NULL;
    while (curr != NULL) {
        temp = curr -> next;
        curr -> next = prev;
        prev = curr;
        curr = temp; // move to next along original list
    }
    return prev;
}
```

5.3 Josephus Problem

Imagine that N people have decided to elect a leader by arranging themselves in a circle and eliminating every Mth person around the circle, with the circle becoming smaller as each person drops out. The task requires you to find out which person will be the last one remaining.

(a) For the case of $M = 2$, we have N people standing in a circle and every second person dies. For example, person 1 shoots 2, then passes the gun to person 3, who then shoots 4. Person 4 passes the gun to person 5 who then shoots 6. And so on. Show that for $M = 2, N = 100$, we have the survivor being person 73.

(i) Solve this by simulating people standing in a circle using a circular linked list.

(ii) Solve this by using recursion.

(b) Write a program for the general case, taking from `stdin` two integers `M N` and output to `stdout` a single integer, representing the person who survives.

(c) Derive a formula for the case of $M = 2$, finding the survivor as a function of N.

The Josephus problem naturally uses linked lists well. takes advantages of linked list (compared to arrays). Using an array here would be costly, because of the algorithm's efficiency depends on its ability to delete items quickly.

Sample Solution - Linked Lists

```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int value;
    struct node *next;
};

int main() {
    struct node *head = malloc(sizeof *head);
    head -> value = 1;
    head -> next = head;
    struct node *curr = head;

    // create linked list and set initial values
    for (int i = 2; i<=100; i++) {
        curr = (curr -> next = malloc(sizeof *curr));
        curr -> value = i;
        curr -> next = head;
    }

    // kill people
    while (curr != curr -> next) {
        curr = curr -> next;
        curr -> next = curr -> next -> next;
    }
    printf("%d\n", curr -> value);
}
```

Sample Solution - Recursion

```
int josephus(int n) {
  if (n == 1)
    return 1;
  else
    return (josephus(n - 1) + 1) % n + 1;
}

// In the main function, we would call josephus(100) to get a value of 73
```

The general case - Linked Lists The setup of the program remains mostly the same. However, instead of 100 people initially, there are now N people. The loop that sets up the linked list now loops until N instead of 100. Also, instead of killing every second person (deleting the next node), we are now deleting every Mth node. This is carried out using a loop to skip $M - 1$ people, and then killing the next person. Compare the two killing operations:

```
// Before:
curr = curr -> next; // skip one person
curr -> next = curr -> next -> next; // delete next person


// The general case:
for (i = 1; i < M; i++) curr = curr -> next; // skip M-1 people
curr -> next = curr -> next -> next; // delete next person
```

6 Grid Problems

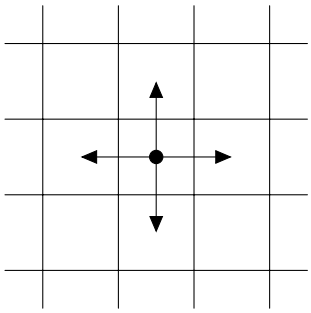
A common strategy for these problems is to create two arrays with the differences in x and y coordinates. We borrow the notation from calculus:

For example, we can declare

```
int dx[] = {-1, 1, 0, 0};
int dy[] = {0, 0, -1, 1};
```

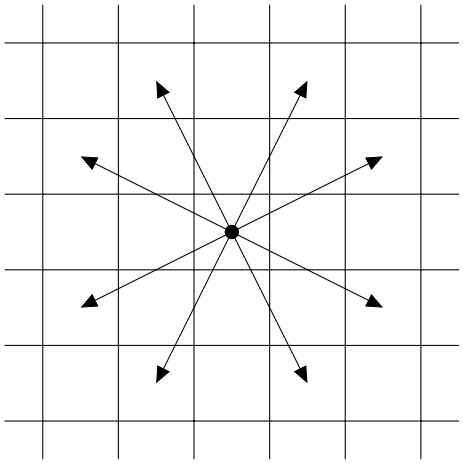
and then we can access the four neighbours using a loop:

```
for (int i = 0; i < 4; i++) {
    visit(x + dx[i], y + dy[i]);
}
```



Knights:

```
int dx[] = {-1, 1, -2, 2, -1, 1, -2, 2};
int dy[] = {2, 2, 1, 1, -2, -2, -1, -1};
```



6.1 Atlantis (AIO 2006)

6.1.1 Problem Statement

Input File: `atlanin.txt`
Output File: `atlanout.txt`

Time Limit: 1 second

You are a geological surveyor on the peaceful island of Atlantis. It is many years yet before your island becomes famous for sinking beneath the sea; instead you are facing water problems of a less troublesome nature.

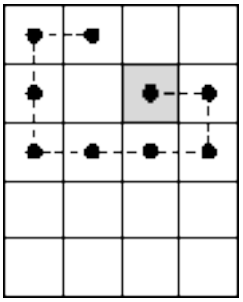
In the middle of your island is a gushing spring, which spills water down across the land. Your job is to map out the paths that this water might take.

You have with you a map of Atlantis, which divides the island into a grid of squares. For each square you have measured the height above sea level. An example of such a map is illustrated below, where the spring is located in the shaded square of height 80.

40	35	35	20
45	50	80	75
55	60	65	70
55	60	65	65
55	60	65	65

As every Atlantean knows, water can only flow downhill. In particular, water can flow from any square to any adjacent square of lower height, where adjacent squares may be to the north, south, east or west (water cannot flow across diagonals). Water cannot flow to an adjacent square of the same height; it must flow strictly downwards. If it cannot flow any further down, it forms a pool and all the local cats, llamas and unicorns come to drink and frolic. You may assume that water will never flow outside the island.

Your task is to find the longest path that the water can possibly take from the spring to a pool. For the example map given earlier, the longest path is illustrated by the dotted line below.



Input

The first line of input will contain the two integers r and c separated by a single space, where r is the number of rows in the map and c is the number of columns in the map. The rows of the map are numbered $1, 2, \ldots, r$ and the columns are numbered $1, 2, \ldots, c$. It is guaranteed that $1 \leq r, c \leq 50$.

The second line of input will contain the integers a and b separated by a single space, describing the location of the spring. Specifically, the spring is located in row a, column b. It is guaranteed that $1 \leq a \leq r$ and $1 \leq b \leq c$.

Following this will be r lines, each representing a single row of the map. Each of these lines will contain c integers separated by spaces, representing the heights above sea level of the c squares in the corresponding row of the map. Each height will be an integer between 0 and 100,000 inclusive.

Output

Your output must consist of a single line containing a single integer, which is the length of the longest

possible path that the water can take. You must count all squares that the water flows through, including the spring at the beginning and the pool at the end (so, for example, the path illustrated earlier has a total length of 9).

Sample Input

```
5 4
2 3
40 35 35 20
45 50 80 75
55 60 65 70
55 60 65 65
55 60 65 65
```

Sample Output

```
9
```

6.1.2 Approaches

This is yet another grid traversal problem, and thus we should immediately think of some common strategies:

- Store the values of the grid in a 2D array
- Use dynamic programming
- Start at the value
- Visit the four neighbours (up, down, left, right). In particular, for each neighbour:
  - Check if it is out of bounds
  - Check if it is of the appropriate height (strictly less than)
  - Otherwise, its max length is given by 1 + solve

  For the answer, choose the max out of the above. If there are no appropriate

Implementation We use a few helper functions.

- `max` - this finds the max of an array, but not the usual way. First of all, the max of the array might not even necessarily be the maximum. Here, we code `-1` to mean an illegal move. If all of the neighbours are illegal moves, then the true max of the array will be -1, however, the function will return zero, meaning there is no possible move.
- `outOfBounds` - takes a coordinate as input and checks if it is out of bounds. We define a point to be out of bounds if its x or y value is less than 0, or greater than the respective maximum number of rows/columns.

Memoization
An initial run of the code will result in correct code, although it is not very time efficient. This is especially an issue in informatics because marks are rewarded for time as well! The reason the code timeouts is because the program repeatedly evaluates the same function calls using different inputs. For example, consider our example above.

A classic example of memoization is for calculating fibbonacci numbers. Without storing the values (in an array, or otherwise), the number of function calls will become exponential. However, this reduces to linear time for unseen n, and $O(1)$ time for previously found n.

Memoization is typically implemented by saving values in an array (here, we use the array `saved`) and looking up the value from the array to check if it is already there.

The function `solve` from the below sample solution first checks if a value has previously been calculated. If it has, the function simply returns the value from the array. If no value is found, the program calculates that value, and saves it in the array for possible future use.

Note that we initialise all the `saved` array values as -1. Thus, any array value of negative 1 can be treated as unvisited, and thus we must calculate its value ourselves.

6.1.3 Sample Solution

```
#include <stdio.h>

int r, c, a, b;
int heights[51][51];
int saved[51][51];
int outOfBounds (int x, int y);
int solve(int x, int y);
int max(int arr[]);

int main(void) {
    FILE *in = fopen("atlanin.txt", "r");
    FILE *out = fopen("atlanout.txt", "w");

    fscanf(in, "%d %d", &r, &c);
    fscanf(in, "%d %d", &a, &b);
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            fscanf(in, "%d", &heights[i][j]);
        }
    }
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            saved[i][j] = -1;
        }
    }
    int answer = solve(a-1, b-1);
    fprintf(out, "%d\n", answer);
}

int solve(int x, int y) {
    int dx[] = {-1, 1, 0, 0};
    int dy[] = {0, 0, -1, 1};
    int next[4] = {0};

    if (saved[x][y] >= 0) {
        return saved[x][y];
    }

    for (int i = 0; i < 4; i++) {
        int newx = x + dx[i];
```

```
            int newy = y + dy[i];
            if (outOfBounds(newx, newy)) {
                next[i] = -1;
            } else if (heights[newx][newy] >= heights[x][y]) {
                next[i] = -1;
            } else {
                next[i] = solve(newx, newy);
            }
        }
        int answer = 1 + max(next);
        saved[x][y] = answer;
        return answer;
}

int max(int arr[]) {
        int ans = 0; // returns a max of 0, even if max is actually -1 (invalid
        for (int i = 0; i < 4; i++) {// guaranteed length is 4
            if (arr[i] > ans) {
                ans = arr[i];
            }
        }
        return ans;
}

int outOfBounds (int x, int y) {
        if (x < 0 ||  x >= r || y < 0 || y >= c) {
            return 1;
        } else {
            return 0;
        }
}
```

6.1.4 Improvements

The above solution has been written to make it as easy as possible to understand the solution and method. However, in an exam environment where code style and readability does not matter (only correctness and speed), what could you do as a contestant to work faster? First of all, the helper functions do not need to be there. They can be directly implemented in the code. Perhaps this section should have been renamed 'unimprovements' as we're actually unimproving the code from here.

The code fragment

```
if (outOfBounds(newx, newy)) {
    next[i] = -1;
} else if (heights[newx][newy] >= heights[x][y]) {
    next[i] = -1;
```

could be replaced with

```
if (x < 0 ||  x >= r || y < 0 || y >= c
          || heights[newx][newy] >= heights[x][y]) {
    next[i] = -1;
}
```

6.2 Cartography

6.2.1 Sample Solution

```c
#include <stdio.h>
int visit(int i, int j);
int w, h;
char map[102][102];
int visit(int i, int j);

int main(void) {
    FILE *in = fopen("cartin.txt", "r");
    FILE *out = fopen("cartout.txt", "w");
    fscanf(in, "%d %d", &w, &h);
    for (int i = 0; i < h; i++) {
        for (int j = 0; j <= w; j++) {
            fscanf(in, "%c", &map[i][j]);
        }
    }
    fscanf(in, "*d");
    int count = 0;
    for (int i = 0; i < h; i++) {
        for (int j = 1; j <= w; j++) {
            if (map[i][j] == '#') {
                count += visit(i, j);
            }
        }
    }
    fprintf(out, "%d\n", count);
    return 0;
}

int visit(int r, int c) {
    int count = 0;
    int dx[] = {-1, 1, 0, 0};
    int dy[] = {0, 0, -1, 1};
    for (int i = 0; i < 4; i++) {
        int newY = r + dy[i];
        int newX = c + dx[i];
        if (map[newY][newX] != '#') {
            count++;
        }
    }
    return count;
}
```