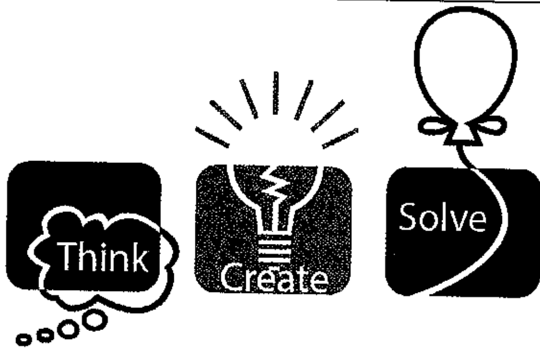
University of Central Florida



2015 (Fall) "Practice" Local Programming Contest

Problems

Problem#	Difficulty Level	Filename	Problem Name
1	Easy	symm	Symmetric Diagonals
2	Easy	snail	Snail in Matrix
3	Easy	gasprice	Gas Price Is Going Up/Down
4	Medium	space	Broken Space Bar
5	Medium	parenlineup	Let's Line Up Parentheses
6	Medium	pagealloc	The Clock Algorithm
7	Medium	fildir	File/Directory Manipulation
8	Hard	goldknight	Knight Moves – Gold Edition
9	Hard	mazetwins	Twins in Maze
10	Hard	magicbeans	Magic Beans
11	Hard	sneetches	Star-Belly Sneetches

Call your program file: filename.c, filename.cpp, or filename.java

For example, if you are solving The Clock Algorithm:

Call your program file: pagealloc.c, pagealloc.cpp, or pagealloc.java

UCF "Practice" Local Contest — Aug 29, 2015

Symmetric Diagonals

Filename: `symm`
(*Difficulty Level:* `Easy`)

Consider the following matrix:

```
A B D F H
C A B D F
E C A B D
G E C A B
I G E C A
```

The elements labeled A are on the First *Symmetric Diagonal* of this matrix. The elements labeled B are on one of the Second Symmetric Diagonals, and the elements labeled C are on the other Second Symmetric Diagonal. Likewise, D's and E's are on the Third Symmetric Diagonals, and so on.

You are to write a program which, given a square matrix of characters, prints some requested symmetric diagonals.

The Input:

There will be several sets of input. Each input set will begin with an integer n $(1 \leq n \leq 15)$, indicating the size of the matrix. On each of the next n input lines, there will be n capital letters (with exactly a single space between letters) representing a row of the matrix. On the next input line will be a positive integer k, indicating the number of symmetric diagonals being requested from this matrix. On each of the next k input lines, there will be an integer representing a request. The integers representing requests are valid, i.e., they are guaranteed to be between 1 and n inclusive.
End of data is indicated by a value of zero for the matrix size (i.e., $n = 0$).

The Output:

Print a heading for each input matrix. Then, echo print the matrix on consecutive output lines with a single space between letters. Then, for each request, print the message:

Symmetric diagonals r:

where r is the number of the symmetric diagonals requested. Print the symmetric diagonals on subsequent output lines. Print the upper diagonal before the lower, print the values as they appear from left to right in the matrix, and print a single space between letters. Leave a blank line after the output for each data set. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
2
T B
E O
1
1
4
F A S X
F R L O
L U E I
Q A N D
2
2
3
0
```

Sample Output:

```
Input matrix #1:
T B
E O
Symmetric diagonals 1:
T O

Input matrix #2:
F A S X
F R L O
L U E I
Q A N D
Symmetric diagonals 2:
A L I
F U N
Symmetric diagonals 3:
S O
L A
```

3

UCF "Practice" Local Contest — Aug 29, 2015

Snail in Matrix

Filename: `snail`
(*Difficulty Level:* `Easy`)

Given a two-dimensional matrix, your program is to print the matrix elements in a "snail-like" fashion as follows:

- print elements in the first row from left to right on an output line
- print elements in the last column from top to bottom on an output line
- print elements in the last row from right to left on an output line
- print elements in the first column from bottom to top on an output line
- print elements in the second row from left to right on an output line

  ...

  (Note that, as illustrated in Sample Input/Output, each matrix element is printed only once.)

The Input:

The input is divided into sets. Each input set starts with two integer values for M and N, dimensions of a matrix (both values between 1 and 15, inclusive, and the two values are separated by at least one space). The next M input lines contain values for matrix rows, one row per input line. Assume that each matrix element is a single character, that input values for matrix elements start in column 1, and that there is exactly one space between matrix elements in the input.
End of data is indicated by values 0 and 0 for M and N.

The Output:

For each matrix, print the original version with one space between the elements. Then, print the matrix elements in a "snail-like" fashion with no space between the elements on an output line. Leave a blank line after the output for each matrix. Follow the format illustrated in Sample Output.

Sample Input:

```
3 4
A B C D
```

```
J K L E
I H G F
5 6
A B C D E F
R S T U V G
Q Z Z Z W H
P Z Z Y X I
O N M L K J
0 0
```

Sample Output:

```
Matrix #1:
Original:
A B C D
J K L E
I H G F
Snail:
ABCD
EF
GHI
J
KL

Matrix #2:
Original:
A B C D E F
R S T U V G
Q Z Z Z W H
P Z Z Y X I
O N M L K J
Snail:
ABCDEF
GHIJ
KLMNO
PQR
STUV
WX
YZZ
Z
ZZ
```

UCF "Practice" Local Contest — Aug 29, 2015

Gas Price Is Going Up/Down

filename: `gasprice`
(*Difficulty Level:* `Easy`)

The gas stations have billboards showing the gas prices (we'll assume only three categories of gas: Regular, Plus, and Super). These billboards display the prices of gas, but the problem is that sometimes some digits are missing from the prices on the billboards!

The Problem:

Given prices for the three categories of gas with at most one digit missing from a given price, you are to determine the exact value of each price. We will assume that Regular is cheaper than Plus which is cheaper than Super. Also assume that Regular is at least $2.00 and Super is at most $5.00.

The Input:

There will be multiple billboards (test cases) in the input. The first input line contains a positive integer *n*, indicating the number of billboards to be processed. The billboards will be on the following *n* input lines, each on a separate line. Each billboard contains three prices, the first showing Regular, the second representing Plus, and the last showing Super. The first price starts in column one, each price uses three columns (decimal points are not in the input), and there is exactly one space separating prices. The characters used in a price are only digits 0 through 9 and hyphen to indicate a missing digit (there is at most one hyphen per price). Since gas is at least $2.00, the digits 0 or 1 will not appear as the first character for a price in the input. Similarly, the maximum gas price ($5.00) dictates possible valid values for the first character.

The Output:

At the beginning of each test case, output "`Gas Station #`*g*`:`", where *g* is the test case number (starting from 1). For each gas station, print the input values and then the output values (each on a separate line and indented three columns). If there are multiple possible (valid) answers, use the lowest valid value for Regular. Then, with the lowest valid value for Regular, use the lowest valid value for Plus. Then, with the lowest valid value for Plus, use the lowest valid value for Super. Assume that input values will always result into at least one possible valid answer.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output. Be sure to line up the output with spaces (not tabs) exactly as given in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
2
2-9 285 -99
-50 -99 -99
```

Sample Output:

```
Gas Station #1:
   Input:  2-9 285 -99
   Output: 209 285 299

Gas Station #2:
   Input:  -50 -99 -99
   Output: 250 299 399
```

UCF "Practice" Local Contest — Aug 29, 2015

Broken Space Bar

Filename: `space`
(*Difficulty Level:* `Medium`)

When people type very fast, they sometimes don't hit the space key hard enough and miss the blank, e.g., instead of typing "sample word list", they get "samplewordlist". You are to write a program to first check whether a word appears in a dictionary. If it doesn't, then you check to see if it appears as a concatenation of two or more words in the dictionary.

The Input:

There will be multiple data sets. The first input line for a data set will be an integer n ($1 \le n \le 50$), indicating the number of words in a dictionary. Each of the following n input lines contains a word. Assume that each word contains only lower-case letters, is at least two letters and at most 20 letters, and starts in column 1. The next input line for the data set will be an integer m ($m \ge 1$), indicating the number of words to be spell-checked. Each of the following m input lines contains a word. Assume that each word contains only lower-case letters, is at least two letters and at most 50 letters, and starts in column 1. End of data is indicated by a data set with a value of zero for n, i.e., a data set with a dictionary size of zero.

The Output:

Print a heading for each data set. Then, print each word to be spell-checked and whether it is in the dictionary. If it is not in the dictionary, print whether it is a concatenation of two or more words in the dictionary and, if it is, print those words. If there is more than one combination of the words in the dictionary that will generate the word to be spell-checked, you only need to print one such combination and not all of them. If the word is not a combination of dictionary words either, print an error message. Leave a blank line after the output for each data set. Follow the format illustrated in Sample Output. Be sure to line up the output with spaces (not tabs) exactly as given in Sample Output.

Sample Input:

```
4
coaches
are
really
nice
5
are
```

```
reallyare
wrong
reallynicecoaches
wrongword
6
how
about
ali
is
nice
isnice
5
funfunfun
aboutali
ali
aliali
isnice
0
```

Sample Output:

```
Data set #1:
     are --- the word is in dictionary
     reallyare --- the word is concatenation of
          really
          are
     wrong --- misspelled word
     reallynicecoaches --- the word is concatenation of
          really
          nice
          coaches
     wrongword --- misspelled word

Data set #2:
     funfunfun --- misspelled word
     aboutali --- the word is concatenation of
          about
          ali
     ali --- the word is in dictionary
     aliali --- the word is concatenation of
          ali
          ali
     isnice --- the word is in dictionary
```

UCF "Practice" Local Contest — Aug 29, 2015

Let's Line Up Parentheses

Filename: `parenlineup`
(*Difficulty Level:* `Medium`)

Given a string of parentheses, you are to use the vertical bar ('|') and underscore ('_') characters to connect the matching parentheses.

The Input:

The first input line will be a positive integer n, indicating the number of strings to be processed. Each of the following n input lines contains a string. Each string will be at most 30 characters and will contain only the open and close parentheses and spaces, i.e., no other characters. There will be at least one parenthesis in each input string.

The Output:

Print a heading for each input string. Then, print the string with exactly one space between parentheses. If there is any mismatched parentheses in the string, print the message "Parentheses do not match!" Otherwise, use the vertical bar and underscore characters to draw lines between matching parentheses. Each line must be at least one vertical bar in depth but no line should be deeper than it needs to be. Leave a blank line after the output for each data set. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
3
( ()   ( ( )() )    )    ( )
   (     ()
())
```

Sample Output:

```
Data set #1:
( ( ) ( ( ) ( ) ) ) ( )
| |_| | |_| |_| | | |_|
|     |_________| |
|_________________|

Data set #2:
( ( )
Parentheses do not match!

Data set #3:
( ) )
Parentheses do not match!
```

UCF "Practice" Local Contest — Aug 29, 2015

The Clock Algorithm

filename: `pagealloc`
(*Difficulty Level:* `Medium`)

In operating systems design, there is a technique called *virtual memory*, which enables the computer to run a program whose required memory (or logical memory) is larger than the available physical memory of the computer. The memory (for computer and program) are divided into pages, each consisting of a block of fixed size. When a program requests to access a page that is not yet loaded into memory, a *page fault* occurs. *Thus, a page fault will occur the first time each page is accessed.* In addition, if the program needs to access more pages than the available memory can provide, one of the pages that were loaded previously will have to be swapped out (written onto disk) in order to provide free memory space for the requested page. The algorithm that decides which page to swap out is called the page replacement strategy. One of the widely-known replacement algorithm is called the least-recently-used (LRU) strategy.

The Problem:

In this problem, we'll explore a variation of LRU known as the clock algorithm. This algorithm works as follows. Initially, all n page cells in memory are free; you can think of them as an array with indices $1 \ldots n$. Each element of the array contains the amount of memory for one page, and a "*flag*". The clock algorithm also keeps a "*hand pointer*", which initially points to cell 1.

When the program needs to access a page, first it checks to see if the page is currently loaded in memory; if so, it simply accesses that page (no page fault) and sets the page's flag to *new*.

If the page it needs is not currently loaded in memory, then it loads the page in the next available free cell (cell with the lowest index) if there is one available and sets the page's flag to *new*. If there is no free cell, then it checks the cell pointed by *hand pointer*. If the page pointed by hand is marked as *old*, then it will be swapped out for the new request (i.e., the page is loaded and marked as *new*) and the pointer advances one position. If the page pointed by hand is marked as *new* (i.e., the page is not *old*), the algorithm then marks that cell/page as *old* and the pointer advances one position. Advancing the pointer simply means the pointer will point to the next memory cell (with 1 higher index), and wraps back to the first cell if advancing from the nth cell, similar to the hands of a clock. Eventually, some page pointed by the hand will be seen as *old* and is swapped out, giving a free cell for the requested page.

Note that when a page is referenced or loaded, its flag is set to *new*.

This strategy gives each cell a second chance, being set to *old* before getting swapped out. If a page is referenced while the cell/page is *old*, it will be marked as *new* again, indicating that it is recently used. Thus a page will only be swapped out if the hand pointer sees it twice (first time marks it as *old*, and second time swaps it out) without the page ever being used during that period.

Your task is to implement the clock algorithm.

The Input:

There will be multiple test cases. The first line of each test case contains two integers, n ($1 \leq n \leq 50$), the number of page cells in available memory, and r ($1 \leq r \leq 50$), the number of page requests made by the program. The next line contains r integers r_i ($1 \leq r_i \leq 50$), separated by spaces. These are pages that are accessed by the program (the pages are listed in the order of being accessed by the program).

The last test case will be followed by a line which contains "`0 0`", i.e., end-of-data is indicated by $n = 0$ and $r = 0$.

The Output:

At the beginning of each test case, output "`Program` p", where p is the test case number (starting from 1). For each request, output: "`Page` r_i `loaded into cell` c." if the request r_i is not in memory, and is loaded into cell c. If the request r_i is already in memory, output: "`Access page` r_i `in cell` c." where c is the cell that page r_i is located in memory. At the end of each test case, output: "`There are a total of` k `page faults.`" where k is the total number of page faults (loads) experienced during the execution of the program.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Notes:

- The size of each page is not important, as the input is already given in units of pages.
- The logical memory requirement for a program is not specified, but you may assume that all programs have a size of 50 pages (as bounded by page request, r_i).

Sample Input:

```
3 12
3 2 1 5 3 2 4 3 2 1 5 4
5 16
1 3 5 7 9 9 7 5 1 8 3 5 2 3 12 18
8 1
1
0 0
```

(Sample Output on the next page)

Sample Output:

```
Program 1
Page 3 loaded into cell 1.
Page 2 loaded into cell 2.
Page 1 loaded into cell 3.
Page 5 loaded into cell 1.
Page 3 loaded into cell 2.
Page 2 loaded into cell 3.
Page 4 loaded into cell 1.
Access page 3 in cell 2.
Access page 2 in cell 3.
Page 1 loaded into cell 2.
Page 5 loaded into cell 3.
Access page 4 in cell 1.
There are a total of 9 page faults.

Program 2
Page 1 loaded into cell 1.
Page 3 loaded into cell 2.
Page 5 loaded into cell 3.
Page 7 loaded into cell 4.
Page 9 loaded into cell 5.
Access page 9 in cell 5.
Access page 7 in cell 4.
Access page 5 in cell 3.
Access page 1 in cell 1.
Page 8 loaded into cell 1.
Access page 3 in cell 2.
Access page 5 in cell 3.
Page 2 loaded into cell 4.
Access page 3 in cell 2.
Page 12 loaded into cell 5.
Page 18 loaded into cell 3.
There are a total of 9 page faults.

Program 3
Page 1 loaded into cell 1.
There are a total of 1 page faults.
```

UCF "Practice" Local Contest — Aug 29, 2015

File/Directory Manipulation

Filename: `fildir`
(*Difficulty Level:* `Medium`)

Operating systems typically allow a user to organize his/her files in a hierarchical (tree structure) fashion. This provides better organization and it makes it easy to locate a file. Your program is to process some basic commands for file and directory manipulation in such operating systems. The commands are as follows:

init

This command appears at the beginning of the input to indicate initialization, i.e., we are starting an empty file system. An empty file system contains only the 'root' directory, which is always designated as '/home'. This will also become the current directory. The init command is used later on in the input when we wish to start with an empty file system again, i.e., all files and directories are deleted (go away).

end

This command indicates end of the input (your program terminates).

mkfile filename

This command creates a file in the current directory. The filename will be at least one and at most 10 characters long, and will consist of only letters (upper case and lower case) and digits.

mkdir dirname

This command creates a directory (subdirectory) in the current directory. The dirname will be at least one and at most 10 characters long, and will consist of only letters (upper case and lower case) and digits. The mkdir command will not be used to create more than five levels of subdirectories in the file system hierarchy (tree), i.e., the file system hierarchy (tree) will have at most five levels (precluding the root directory).

cd path

This command is used to change to a different directory. The path can be '..' which indicates go up to the parent directory, a dirname which goes down to the specified subdirectory, or any combination of '..' and dirnames, separated by '/', to go up and/or down the tree multiple times in one command. There will be at most five '/' in path, and the result will always be a valid directory.

ls

This command is used to list the file/directory names in the current directory (i.e., where we are). List the names in increasing order (by using a string comparison function such as strcmp for sorting). Assume that there will be at most 20 names (i.e., at most 20 file/directory names in any directory), that these names are distinct (i.e.,

no duplicate names in any directory), and that there will be at most 5000 user-created names in the entire file system.
If the current directory is empty (i.e., contains no files or directories), print a message indicating so.

The Input:

Each input command starts in column one. If an input line consists of two parts, there will be exactly one space separating the two parts. Assume that all the commands are syntactically and semantically correct, i.e., no error checking.

The Output:

Only the commands init and ls produce output. The init command generates only a heading. The ls command generates a heading and the information requested. Note that, for ls, the heading includes the current directory. At the top level, this will be /home. At the lower levels, it will include the rest of the path as well. Assume that every file system will have one or more ls commands, i.e., there will be one or more ls after each init. The program output (including the headings and spacing) are illustrated in Sample Output; follow the given format.

(Sample Input/Output on the next two pages)

Sample Input:

```
init
mkfile temp
mkdir COURSES
mkfile junk
mkdir CONTESTS
ls
cd COURSES
mkdir COP4020
cd COP4020
mkfile exam1
mkfile exam3
mkfile exam2
mkfile EXAM1
ls
cd ..
mkdir COT4810
cd COT4810
cd ../../CONTESTS
mkdir REAL
mkfile scoring
mkdir PRACTICE
ls
cd REAL
mkfile temp
mkdir junk
cd ../../COURSES/COT4810
ls
cd ..
ls
init
mkfile aaa
mkfile bbb
mkdir Level1
cd Level1
mkdir Level2a
mkdir Level2b
cd Level2a
mkdir Level3
cd Level3
mkdir Level4
cd ../../..
ls
end
```

Sample Output:

```
File system #1:

Listing for /home:
CONTESTS
COURSES
junk
temp

Listing for /home/COURSES/COP4020:
EXAM1
exam1
exam2
exam3

Listing for /home/CONTESTS:
PRACTICE
REAL
scoring

Listing for /home/COURSES/COT4810:
There are no files/directories.

Listing for /home/COURSES:
COP4020
COT4810

File system #2:

Listing for /home:
Level1
aaa
bbb
```

UCF "Practice" Local Contest — Aug 29, 2015

Knight Moves – Gold Edition

filename: `goldknight`
(*Difficulty Level:* `Hard`)

You have a chessboard of size N x N. The rows and columns are numbered from 1 to N. In a cell located at row R_1 and Column C_1, a knight is starting his journey. The knight wants to go to the cell located at row R_2 and Column C_2. Move the knight from the starting cell to this destination cell with minimum number of moves.
As a reminder, a knight's jump moves him 2 cells along one of the axes, and 1 cell along the other one. In other words, if a knight is at (A,B), it may move to (A-2,B -1), (A-2, B+1), (A+2, B-1), (A+2, B+1), (A-1, B-2), (A+1,B-2), (A-1, B+2) or (A+1, B+2). Of course, the knight cannot leave the board.

The Problem:

Given N, R_1, C_1, R_2 and C_2, determine the minimum number of steps necessary to move the knight from (R_1, C_1) to (R_2, C_2).

The Input:

The first input line contains a positive integer, *T*, indicating the number of test cases. Each case consists of a line containing five integers N ($3 \le N \le 20$), R_1, C_1, R_2 and C_2 ($1 \le R_1, C_1, R_2, C_2 \le N$).

The Output:

For each test case, first output "`Case #i:`" where *i* is the test case number, starting with 1. Then, output the minimum number of steps needed to move the knight from (R_1, C_1) to (R_2, C_2). Assume that there will always be a solution, i.e., it's possible to move the knight from its starting cell to its destination cell. Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
2
5 1 1 2 3
5 1 1 2 2
```

Sample Output:

```
Case #1: 1

Case #2: 4
```

UCF "Practice" Local Contest — Aug 29, 2015

Twins in Maze

Filename: `mazetwins`
(*Difficulty Leval:* `Hard`)

Dr. Orooji's twins (Mack and Zack) are 14-years old and, at times, they drive Dr. O crazy. The situation is really bad when Mrs. O is not home and Dr. O has to watch the twins by himself. To make sure he keeps his sanity, Dr. O has devised a square maze and, when he needs a break, he puts the twins in two different cells in this maze. While Mack and Zack are running around, trying to find each other, Dr. O has some peace! But, here you come, trying to write a program to help the twins and make Dr. O mad!

Let's define cell A to be a "neighbor" of an adjacent cell B if A is to the north, south, east, or west of cell B (i.e., diagonal cells are not considered neighbors). For mazes with more than two rows, cells in the first row are not considered neighbors of cells in the last row (i.e., no wrap around). Similarly, for mazes with more than two columns, cells in the first column are not considered neighbors of cells in the last column (i.e., no wrap around).

The twins can travel only one cell at a time. From a cell, they can only go to a neighboring cell. The twins are originally placed, by Dr. O, in two cells that are not neighbors. The twins find each other by making some moves such that they end up in neighboring cells.

The Input:

The first input line will be a positive integer d, indicating the number of data sets (mazes) to be processed. The data sets are on the following input lines.

Each data set starts with an integer n ($3 \le n \le 15$), representing a maze size. Each of the following n input lines contains n characters (starting in column 1 and ending in column n), representing a row of the maze. These characters are as follows:

M shows the original position for Mack (exactly one M in each input maze).

Z shows the original position for Zack (exactly one Z in each input maze).

0 a cell where neither twin can move to.

1 a cell where only Mack (twin #1) can move to.

2 a cell where only Zack (twin #2) can move to.

3 a cell where either twin (i.e., both) can move to.

Assume that each input maze is such that the twins will find each other (Dr. O needs to rest sometimes but he is not a mean dad).

The Output:

Print a heading for each data set. Then, print all the moves Mack has to make (in order), followed by all the moves Zack has to make (in order). Also print the total number of moves made by Mack and Zack. Leave a blank line after the output for each data set. Follow the format illustrated in Sample Output.

Your program must find the solution with the smallest number of total moves. If there is more than one solution with the smallest total, you may print any one of them.

Sample Input:

```
2
6
000000
00M130
000010
000020
000030
00Z220
4
000Z
1022
3010
31M0
```

Sample Output:

```
Maze #1:
   Mack move east
   Mack move east
   Mack move south
   Zack move east
   Zack move east
   Zack move north
   Zack move north
   Total number of moves: 7

Maze #2:
   Mack move north
   Zack move south
   Zack move west
   Total number of moves: 3
```

UCF "Practice" Local Contest — Aug 29, 2015

Magic Beans

filename: magicbeans
(*Difficulty Level:* Hard)

Peter loves magic beans. It turns out that the beans make it very easy to travel from place to place in his country, the Cartesian Coordinate System. To travel, Peter merely eats a bean and ends up in a new place!

Each bean teleports Peter in a predictable way. Say Peter is at position (x, y) on the plane. If he eats a bean it will teleport him to cell (x+a, y+b) where ***a*** and ***b*** are numbers written on the bean. Your task is to find the number of ways Peter can reach his destination location by eating beans. Peter's starting location is (0, 0). Two ways are considered different if it causes Peter to eat a different number of beans or the ith bean eaten differs. (Note: Peter may arrive at his destination prematurely as long as his final destination is reached when he completes eating beans. Also note that if two different beans contain the same values ***a*** and ***b***, those beans are considered different from one another.)

The Input:

The first line of input contains a single positive integer ***t*** ($t \le 50$), the number of journeys to process. For each journey, the first line contains three integers ***n***, ***dx***, ***dy*** ($1 \le n \le 32$, $-10^9 \le dx, dy \le 10^9$), representing the number of beans available to Peter, the x coordinate of his destination and the y coordinate of his destination location, respectively. Peter's destination will never be his starting location.

The next ***n*** lines contain two integers a_i and b_i, ($-10^9 \le a_i, b_i \le 10^9$), representing the numbers written on each bean.

The Output:

For each journey, output a single integer on a line by itself representing the number of ways Peter can eat the beans. As this number can get large, output the result modulo 1,000,000,007.

Sample Input:	Sample Output:
3 3 2 2 3 3 -1 -1 2 2 6 5 3 2 1 2 2 1 0 -2 -2 3 3 2 0 4 2 2 1 1 1 1 -1 2 1 -2	3 32 26

In the last sample case, we count all 4! orderings of eating all four beans which take Peter to (2,2), treating bean 1 and bean 2 as different beans, even though they have the same values ***a*** and ***b*** written on them. Note that some of these 4! orderings included visiting the destination (2, 2) in the middle of the journey. The last two ways we can get to (2,2) is by just eating bean 1 followed by bean 2, or eating bean 2 followed by bean 1.

UCF "Practice" Local Contest — Aug 29, 2015

Star-Belly Sneetches

filename: `sneetches`
(*Difficulty Level:* `Hard`)

Now, the Star-Belly Sneetches had bellies with stars.
The Plain-Belly Sneetches had none upon thars.
Those stars weren't so big. They were really so small.
You might think such a thing wouldn't matter at all.

"My friends", he announced in a voice clear and clean,
"My name is Sylvester McMonkey McBean.
And I've heard of Your troubles. I've heard you're unhappy.
But I can fix that, I'm the Fix-It-Up Chappie.

Then, quickly, Sylvester McMonkey McBean
Put together a very peculiar machine.
And he said, "You want stars like a Star-Belly Sneetch?
My friends, you can have them for three dollars each!"

Then, of course, those with stars got all frightfully mad.
To be wearing a star was frightfully bad.
Then, of course, old Sylvester McMonkey McBean
invited THEM into his Star-Off Machine.

Then, of course from THEN on, as you probably guess,
Things really got into a horrible mess.

Now Sylvester McMonkey McBean is using his Star-Swapping machine to add and remove stars from Sneetches that want them or don't want them. The Sneetches keep wanting to either add a star to their belly or remove the star from their belly. Sylvester now wants to keep track of which sneetches on the beaches are most likely to want their star swapped next.

He does this by observing that the sneetches are lined up on the beaches in order from left to right. He can label the sneetches using the numbers 1 to n, where n is the number of sneetches. Sylvester knows that each Sneetch is more likely to swap stars if those around him are the same type of sneetch. The Sneetches want to be different! That is why he wants to know the longest contiguous block of Star-Bellied Sneetches and the longest contiguous block of Plain-Bellied Sneetches. As the swapping happens things may change. That is why he needs to know these values when some range of consecutive sneetches choose to swap their stars.

The Input:

The first line contains a positive integer ***t*** (***t*** ≤ 50), representing the number of beaches with Sneetches to consider.

For each town, the first line contains two positive integers ***n*** and ***s***, (1 ≤ ***n***, ***s*** ≤ 10^5) representing the number of Sneetches on the beaches and the number of star swapping operations to perform.

The following line is a string of ***n*** characters. Each character is either 'S' or 'P', representing if the ith sneetch on the beach is a Star-Bellied Sneetch or a Plain-Bellied Sneetch.

The next line ***s*** lines contain two integers each ***a*** and ***b***, (1 ≤ ***a*** ≤ ***b*** ≤ ***n***) representing the interval of sneetches on which to perform the star swapping. The range [***a***, ***b***] is inclusive.

The Output:

For each star swapping, output two integers on a line by themselves. The first represents the largest consecutive group of star-bellied sneetches and the second represents the largest consecutive group of plain-bellied sneetches.

Sample Input:	Sample Output:
3 4 2 SSSS 2 3 3 4 4 1 SPPP 2 4 6 5 PPPPPP 1 3 2 4 3 5 4 4 6 6	1 2 1 1 4 0 3 3 1 2 1 1 3 1 4 1

The annotations below show how the sneetches change after each swap operation.
Case 1: SSSS -> SPPS -> SPSP
Case 2: SPPP -> SSSS
Case 3: PPPPPP -> SSSPPP -> SPPSPP -> SPSPSP -> SPSSSP -> SPSSSS