Simplified Keyboard

filename: `typing`

(*Difficulty Level:* `Easy`)

Consider a simplified keyboard consisting of the 26 lowercase letters as illustrated below:

a	b	c	d	e	f	g	h	i
j	k	l	m	n	o	p	q	r
s	t	u	v	w	x	y	z	

We define the neighbors of a key (letter) as all the letters adjacent to it. For example, the neighbors of 'a' are {b, k, j}, neighbors of 'b' are {a, c, l, k, j}, neighbors of 'n' are {d, e, f, o, x, w, v, m}, and neighbors of 'z' are {p, q, r, y}.

The Problem:

Given two words consisting of lowercase letters only, you are to determine which of the following three cases applies to them:

1. identical: this is when the two words are of the same length and they match letter-by-letter. For example, "cool" and "cool" are identical, "cool" and "col" are not, and "cool" and "colo" are not.

2. similar: this is when the two words are of the same length, they are not identical words, and each corresponding two letters either match or are neighbors. For example, "aaaaa" and "abkja" are similar, "moon" and "done" are similar, "knq" and "bxz" are similar, but "ab" and "cb" are not (because of 'a' in the first word and the corresponding 'c' in the second word).

3. different: this is when neither of the above two cases applies to the two words, i.e., they are not identical and they are not similar. For example, "ab" and "abc" are different, "ab" and "az" are different, and "az" and "za" are different.

The Input:

The first input line contains a positive integer, *n*, indicating the number of test cases to process. Each of the following *n* input lines represents a test case, consisting of two words separated by one space. Each word consists of lowercase letters only and will be between 1 and 20 letters, inclusive.

The Output:

For each test case, output one line. That line should contain the digit (number) 1, 2, or 3, to indicate which of the above three cases applies to the two input words.

Sample Input:

```
7
a k
a a
a z
cool cool
aaaaa abkja
ab abc
az za
```

Sample Output:

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

Call your program file one of the following:

typing.py typing.c

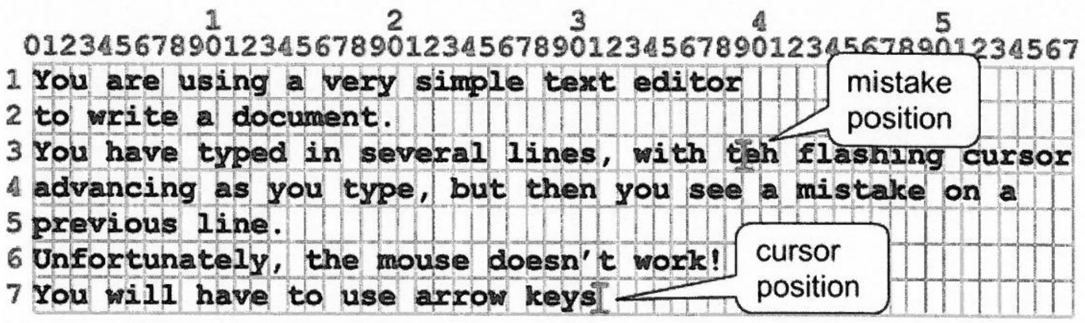
typing.java typing.cpp

Editor Navigation

filename: `editor`

(*Difficulty Level:* `Medium`)

You are using a very simple text editor to create a document. You have typed in several lines, with the flashing cursor advancing as you type, but then you see a mistake on a previous line. Unfortunately, the mouse doesn't work! You will have to press arrow keys to move the cursor back to the position where you can fix the mistake. Of course, you want to get to this position as quickly as possible.



```
            1         2         3         4         5
  0123456789012345678901234567890123456789012345678901234567
1 You are using a very simple text editor
2 to write a document.
3 You have typed in several lines, with teh flashing cursor
4 advancing as you type, but then you see a mistake on a
5 previous line.
6 Unfortunately, the mouse doesn't work!
7 You will have to use arrow keys
```

This simple editor uses a monospace font, so each character is exactly the same width. The cursor can be at the beginning of the line (before the first character), end of the line (after the last character), or at a horizontal position between two characters on a given line. The following keys can be pressed to move the cursor (each keypress is independent of any preceding keypress):

🡐 (Left arrow key)	Moves the cursor one character to the left on the same line, unless the cursor is at the beginning of the line, in which case it moves to the end of the previous line. If the cursor is at the beginning of the line and there is no previous line, the cursor does not move.
🡒 (Right arrow key)	Moves the cursor one character to the right on the same line, unless the cursor is at the end of the line, in which case it moves to the beginning of the next line. If the cursor is at the end of the line and there is no next line, the cursor does not move.
🡑 (Up arrow key)	Moves the cursor up to the previous line; keeps the same horizontal position, unless the previous line is shorter, in which

	case the cursor goes to the end of the previous line. If there is no previous line, the cursor does not move.
↓ (Down arrow key)	Moves the cursor down to the next line; keeps the same horizontal position, unless the next line is shorter, in which case the cursor goes to the end of the next line. If there is no next line, the cursor does not move.

The Problem:

Given the line lengths of a text file that was loaded in this simple editor, along with the current cursor position and a different, desired cursor position (e.g., to fix a mistake), you are to determine the minimum number of keypresses, using arrow keys only, required to move the cursor to the desired position.

The Input:

The first input line contains a positive integer, n, indicating the number of editor navigation scenarios to process. Each scenario will occupy exactly 4 input lines. The first line of each scenario contains an integer f ($1 \leq f \leq 120$), indicating the number of lines of text in the file that is loaded in the editor. The next input line contains f integers, s_1 to s_f, where each value s_i ($0 \leq s_i \leq 80$) indicates the number of characters on line i of the file; the values will be separated by exactly one space. A value $s_i = 0$ means that there are no characters on line i. The newline character (common character indicating end of a line) does not count as a character on the line.

The third input line of each scenario will contain two integers (separated by a space) providing the current cursor position in the file: r_c ($1 \leq r_c \leq f$) and c_c ($0 \leq c_c \leq 80$), where r_c represents the line of the file, counting from 1 (as with i), and c_c represents the horizontal position of the cursor on that line, 0 for the beginning (before the first character). It is guaranteed that the cursor position is valid, so if, for instance, $r_c = 17$ and $s_{17} = 56$, then $0 \leq c_c \leq 56$; the maximum value of c_c is the end of the line. The fourth input line of each scenario is similar to the third and indicates the desired cursor position to begin fixing the mistake, i.e., this line consists of two integers separated by a space, r_m ($1 \leq r_m \leq f$) and c_m ($0 \leq c_m \leq 80$). The constraints on the values for r_c and c_c also apply to r_m and c_m.

The Output:

For each scenario in the input, output a single integer on a line by itself indicating the minimum number of keypresses needed to move the cursor from its current position (r_c, c_c) to the desired position (r_m, c_m) for fixing the mistake.

Sample Input:

```
2
7
39 20 57 54 14 38 31
7 31
3 39
3
15 30 20
1 12
3 3
```

Sample Output:

```
21
8
```

Explanation for Sample Output:

For Case #1, one possible sequence for the minimum number of keypresses to move the cursor from its current position to the desired position is: Up, Up, Right, Up, Left, Up, Left 15 times.

Simple Darts

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

A staple to many game rooms, darts is a fun game that doesn't require a lot of physical space. The standard dartboard (see Figure 1) consists of 6 concentric rings and 20 wedges, dividing the board into a number of regions, each with well-defined scores (note: for a higher picture resolution/clarity, fewer than 20 wedges are depicted in Figure1). The centermost ring is scored as a double bullseye, while the second ring gives the score of a single bullseye. Beyond the second ring, each wedge has a score between 1 and 20 with the spaces between certain rings having double and triple score modifiers. A new, simpler version of the game is being proposed to attract younger players to the game.

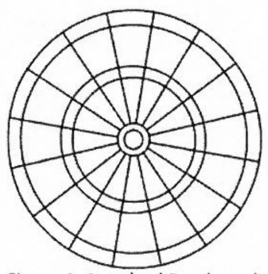


Figure 1, Standard Dartboard

The Problem:

The simpler version of the game is illustrated in Figure 2. More specifically, the board consists of exactly 3 (instead of 6) concentric rings and it may have fewer (instead of exactly 20) wedges. Also, in the simpler board, the wedges are of equal size. The scoring for this simple version is as follows:
Assume the board is centered at the origin ("0,0" in Cartesian plane). Let *w* refer to the number of wedges on the board (8 in Figure 2), and let *b, d, s* refer to (respectively) the radii of the smallest, second smallest, and the largest rings.
The wedge immediately above the positive x axis (Wedge 1 in Figure 2) is scored at 1 point and the score for each wedge is increased by 1 in a counterclockwise fashion, thus resulting in the wedge below the positive x axis (Wedge 8 in Figure 2) having a score of *w*. The area within the circle centered at the origin with radius *b* represents the bullseye and is always scored at 50 points. The area between radii *b* and *d* denotes the double ring and any dart landing within the double ring is scored at twice the value of the surrounding wedge. The area between radii *d* and *s* denotes the single ring and any dart landing within the single ring is scored at the value of the surrounding wedge. Any dart landing outside the dartboard carries a score of zero (0).

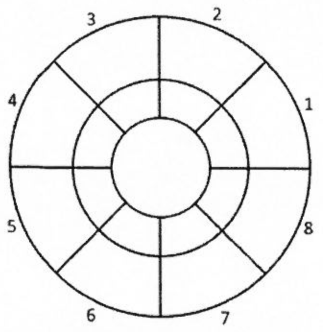


Figure 2, Example Simple Dartboard

Given the description (layout) of such a board centered at the origin and a number of dart throws, you are to calculate the total score.

The Input:

The first input line contains a positive integer, n, indicating the number of test cases to process. Each test case will contain multiple input lines. The first line of each test case will contain four integers separated by a space: w ($2 \leq w \leq 20$), representing the number of equal-sized wedges on the board, followed by b, d, s ($0 < b < d < s < 100$), representing the radii of the bullseye, double ring and single ring regions of the board. The second input line of each test case will contain an integer, t ($1 \leq t \leq 100$), indicating the number of dart throws. Each of the following t input lines contains two floating point numbers (three decimal places), x and y ($-100 \leq x,y \leq 100$), providing the Cartesian coordinates of a dart throw. Assume that no dart will land within 10^{-5} of any boundary (line or arc/curve).

The Output:

For each test case, output a single integer on a line by itself indicating the total score for all dart throws.

Sample Input:

```
3
4 7 13 10
2
4.000 4.000
6.000 -4.000
10 1 6 10
1
20.000 -0.500
8 3 7 50
5
-0.750 1.207
1.180 3.132
27.111 -44.630
-43.912 -22.104
2.000 -6.000
```

Sample Output:

```
58
0
73
```

Videogame Probability

filename: `game`

(*Difficulty Level:* `Medium/Hard`)

You have just joined the illustrious Ewokin guild (group of players) in your favorite video game. This guild is known for attempting the hardest raids the instant that they are released. For these new raids you need the best gear (items) in order to have a viable chance of success. With your skills as a programmer, you have managed to determine the current drop rates for different item types in the game (drop rate refers to the probability of catching an item when the item is dropped). Now you need to know how likely it is that you will gather your gear (items) and bring your guild one step closer to success.

The Problem:

Given the number of different item types in a game, how many of each item type you need, the probability of obtaining (catching) each item type when it drops, and the maximum number of possible attempts you have for catching the items, determine the probability that you will obtain the desired number of each item type.

The Input:

The first input line contains a positive integer, n, indicating the number of test cases to process. Each test case starts with an integer, g ($1 \leq g \leq 50$), indicating the number of different item types in the game. The following g input lines provide the information about the different item types. Each such line contains two values: an integer, c ($0 \leq c \leq 50$), representing how many of this item type is needed, and a floating point number, p ($0.0 \leq p \leq 1.0$), representing the probability of catching this item type when it drops. The last input line for a test case contains an integer, a ($0 \leq a \leq 10000$), representing the maximum number of attempts you have for catching the items. Note that this last input represents the total number of attempts and not the attempts for each item type. An attempt can, of course, be used (applied) to obtain any item type.

The Output:

For each test case, print a floating point number on a line by itself, representing the probability that you will obtain the desired number of each item type. Output the results to 3 decimal places, rounded to the nearest thousandth (e.g., 0.0113 should round to 0.011, 0.0115 should round to 0.012, and 0.0117 should round to 0.012).

Sample Input:

```
4
2
3 0.5
3 0.3
20
4
2 0.75
1 0.01
2 1.0
3 0.8
25
2
1 0.5
1 0.3
2
2
50 .4
50 .3
250
```

Sample Output:

```
0.816
0.153
0.150
0.036
```

Rotating Cards

filename: `cards`

(*Difficulty Level:* `Medium/Hard`)

A magician has a stack of n cards labeled 1 through n, in random order. Her trick involves discarding all of the cards in numerical order (first the card labeled 1, then the card labeled 2, etc.). Unfortunately, she can only discard the card on the top of her stack and the only way she can change the card on the top of her stack is by moving the bottom card on the stack to the top, or moving the top card on the stack to the bottom. The cost of moving any card from the top to the bottom or vice versa is simply the value of the label on the card. There is no cost to discard the top card of the stack. Help the magician calculate the minimum cost for completing her trick.

The Problem:

Given the number of cards in the magician's stack and the order of those cards in the stack, determine the minimum cost for her to discard all of the cards.

The Input:

The first input line contains a positive integer, t, indicating the number of test cases to process. Each test case is on a separate input line by itself and starts with an integer, c ($1 \le c \le 10^5$), indicating the number of cards in the stack, followed by c labels for the cards in the stack (starting from the top going to the bottom). Each of these labels will be in between 1 and c, inclusive, and each label will be unique.

The Output:

For each test case, output a single integer on a line by itself indicating the minimum cost for the magician to complete her magic trick.

Sample Input:

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

Sample Output:

```
15
0
```