**Java program:** Prob13.java

**Input File:** Prob13.in.txt

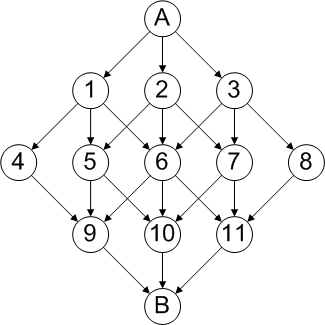
**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

Choices are everywhere. Left or right?Beef or chicken?To be or not to be? These are the questions. But your task involves a different kind of choice – one that will take you from point A to point B.

Choosing a path from point A to point B involves evaluating your choices at each step along the way. During the first half of your journey, the possibilities are many. Specifically, at each point along the way you will have to choose to move in one of three directions.

However, sometimes in life the closer we get to our destination, the fewer choices we have in terms of how to get there. Therefore, for the second half of the journey, your choices will start to narrow. The set of all possible paths will look like a diamond.



Starting at node A, you can choose to go to node 1, 2, or 3. Your choice there determines your next set of choices and so on. Once you hit the middle, your choices narrow if you are on the edge. Being in node 6 means you still have three choices to get to the next level, but being in nodes 5 or 7 reduce your options to 2. Nodes 4 and 8 have zero choice.

Each path will be given a weight representing the difficulty of going down that path. Your job is to determine the path (or paths) of least resistance in getting from point A to point B.

**Program Input**

The file Prob13.in.txt will contain path weights separated by spaces. All paths will have a weight. Path weights will be grouped based on the nodes above them from left to right. For example, the input file will have all three path weights associated with node 1 before the three path weights associated with node 2. Paths that do not exist will be omitted.

Each line of input will contain the connectors for each level separated by spaces. There will always be a multiple of three connectors per line.

**Example Input:**

1 2 3

2 1 1 1 1 1 1 1 1

0 2 1 1 2 1 2 1 2

1 3 1

**Program Output**

Your program should report two things:

1. The lowest possible path difficulty. Use the format “Lowest path difficulty: #”.
2. The number of paths that have the lowest possible path difficulty. Use the format “Number of paths with the lowest difficulty: #”.

**Example Output:**

Lowest path difficulty: 4

Number of paths with the lowest difficulty: 3

**Java program:** Prob14.java

**Input File:** Prob14.in.txt

**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

If you were here last year then you may remember Mabel, our resident bingo lover. Since you coded a solution to check her bingo cards, she has grown bored of the traditional bingo game. In order to challenge herself again, she started a 3D Bingo parlor to kick it up a notch. Your task is to help her check for bingos. The following information will help you.

**Traditional bingo:**

A bingo card utilizes the numbers 1 through 75. The five columns of the card are labeled 'B', 'I', 'N', 'G', and 'O' from left to right. The center space is usually marked "Free" or "Free Space", and is considered automatically filled. The range of printed numbers that can appear on the card is normally restricted by column, with the 'B' column only containing numbers between 1 and 15 inclusive, the 'I' column containing only 16 through 30, 'N' containing 31 through 45, 'G' containing 46 through 60, and 'O' containing 61 through 75. The object of the game is to fill 5 consecutive spaces (a whole row, a whole column, or one of the two diagonals) – which is called a bingo. The game ends when a bingo is found on one or more cards.

**3D bingo:**

3D bingo is very similar to traditional bingo, except that sets of 5 cards can be used to form a “bingo cube”, allowing for a bingo in more ways (straight up and diagonally up). Obviously the free space would cause an automatic bingo, so it has been eliminated in the 3D version.

Mabel really wanted a challenge keeping up with her cards, so she decided that cards could be used in any combination to form a cube. This means that if you buy 4 cards, you have to rely on a single card for a bingo. However, if you buy 5 cards, then you can make a bingo cube. Since the order of the cards does not need to be set before the game starts, there are 120 ways to make the bingo cube out of 5 cards. Buying a sixth card means that you can make 720 different bingo cubes out of your cards. You can see how this game would be much more challenging to keep track of.

Your task is to write a program that will tell Mabel how many possible bingo combinations there are during a 3D bingo game.

**Program Input**

The file Prob14.in.txt will contain two sections:

1. The first section will contain the bingo card information. There will be 5 input lines per card. Each line will contain the numbers for one row of a bingo card separated by spaces. The first 5 rows will make up card number 1, the second 5 rows will make up card number 2, and so on. The number of rows will be a multiple of 5 (meaning there are no extra rows to throw away).
2. The second section will be the numbers called during the game. The word PLAY will appear on a line by itself to separate the card information from the game simulation. Following the word PLAY each line will contain a value to be played on your bingo cards (i.e. B3, I27, or B5). Once a bingo is found, the game stops. Any extra input after a bingo is found should be ignored.

**Example Input:**

10 23 36 49 67

13 26 34 50 70

14 20 42 59 71

6 25 35 47 72

7 27 31 57 61

9 24 36 48 65

11 16 41 47 71

1 19 43 56 64

2 17 44 53 62

3 18 42 51 66

12 20 39 59 69

4 22 41 55 61

6 29 33 53 72

14 17 37 58 65

11 16 44 47 66

7 26 31 57 73

1 21 40 55 61

8 29 43 47 62

11 27 35 46 66

6 16 36 52 67

5 16 33 47 62

7 27 35 52 66

11 22 44 51 70

12 19 36 49 72

13 21 41 59 73

PLAY

I28

G46

I16

N43

B3

O67

O62

O64

G49

G54

B7

N44

O63

B5

G57

B6

I27

N38

G55

B8

B11

O73

**Program Output**

Your program should print the total number of bingos found using the format shown below.

**Example Output:**

Number of bingos: 2

**Java program:** Prob15.java

**Input File:** Prob15.in.txt

**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

XML (Extensible Markup Language) is a powerful tool for moving data around because the user is allowed to define tag names. If written properly, XML can be interpreted easily by both humans and machines. However, if written poorly, XML can be a nightmare for a human to read. Your task is to write an XML pretty printer. We’ll start with a little XML lesson in case you are unfamiliar.

Every XML document starts with a declaration line that looks something like this:

* <?xml version="1.0" encoding="UTF-8" ?>

The declaration section starts with the character sequence <?xml and ends with the character sequence ?>.

After that, the rest of the XML document is made up of XML elements. XML elements are denoted by tags. A tag is a string of characters surrounded by less than (<) and greater than (>) signs. There are three types of tags:

* A start tag starts with a less than sign and ends with a greater than sign marks the beginning of an XML element: <element>
* An end tag starts with a less than sign with a forward slash and ends with a greater than sign marks the end of an XML element: </element>
* A self-closing tag, which combines the start and end tags into a single tag starts with a less than sign and ends with a forward slash with a greater than sign: <element />

XML tags may also contain attributes. Attributes have information that pertains to the XML element they appear in. Attributes appear inside XML tags and are name-value pairs separated by an equal sign, with the value being double quoted. Here is an example of an XML start tag with two attributes:

<myTag att1=”attribute 1 value” attribute2=”att2value”>

The content of an XML element is what comes between the start tag and the end tag. XML elements can contain plain text content, or they can contain other XML elements. If you are familiar with XML, this problem will not deal with escaped special characters or CDATA.

**Program Input**

The file Prob15.in.txt will contain an XML file. It will not be pretty, but it will be valid XML.

**Example Input:**

<?xml version="1.0" encoding="UTF-8" ?><RootElement>

<Element1 AtTrIbUTe1 = " This whitespace should appear as is." > This text

should all be

on one line!

</Element1 >< Element2 name="VALUE" name2="VAlue2"/>< Element3 attr

=

"Element1" />

<Element1 AtTrIbUTe1 = " This whitespace should appear as is." > This text

should all be

on one line

too! Watch out for "quoted" content.

</Element1 >< Element2 name="VALUE" name2="VAlue2"> Nested content< Element3 attr

=

"Element1" />

anyone?

</ Element2>

<operation name="process">

<soap:operationsoapAction="" style="document"/>

<input>

<soap:body use="literal"/>

</input>

<output>

<soap:body use="literal"/>

</output>

</operation>

</ RootElement>

**Program Output**

Your program should print the XML from the input file in a more human readable format. Specifically, your program should:

1. Trim all whitespace from the beginning and end of the XML file.
2. Print the XML declaration as-is by itself on the first line.
3. Start each XML element on its own line. If an element is nested, indentation should be used to visually show how deep the indentation is. Use four periods per level to show the indentation inserted by your program. There is no limit to the level of nesting possible in an XML file.
4. Self-closing tags and elements with no nested elements in their content should be printed on a single line with no content following the end tag.
5. For elements with nested elements in their content, print the end tag at the same level of indentation as the start tag. In this case, both the start tag and the end tag should appear on their own line, with the content indented.
6. Leave no whitespace between the beginning of any XML tag and the XML element name.
7. Leave no whitespace between the greater than sign at the end of a start or end tag and the character preceding it.
8. Make sure there is one space between the forward slash at the end of a self-closing XML tag and the character preceding it.
9. Leave no space on either side of the equal sign of an attribute.
10. Make sure attributes are separated from XML element names and other attributes by a single space.
11. Change all attribute names to lowercase.
12. Remove whitespace between any element’s start and end tags and text content. Also remove whitespace between XML tags.
13. Condense extra whitespace within an element’s text content to a single space.

**Example Output:**

<?xml version="1.0" encoding="UTF-8" ?>

<RootElement>

....<Element1 attribute1=" This whitespace should appear as is.">This text should all be on one line!</Element1>

....<Element2 name="VALUE" name2="VAlue2" />

....<Element3 attr="Element1" />

....<Element1 attribute1=" This whitespace should appear as is.">This text should all be on one line too! Watch out for "quoted" content.</Element1>

....<Element2 name="VALUE" name2="VAlue2">

........Nested content

........<Element3 attr="Element1" />

........anyone?

....</Element2>

....<operation name="process">

........<soap:operationsoapaction="" style="document" />

........<input>

............<soap:body use="literal" />

........</input>

........<output>

............<soap:body use="literal" />

........</output>

....</operation>

</RootElement>

**Java program:** Prob16.java

**Input File:** Prob16.in.txt

**Output:** Your output needs to be directed to stdout (i.e., using System.out.println())

**Introduction**

Treasure hunting is a lost art form because of all this newfangled technology that everyone uses today. Did Captain Hook use Google Maps to find his way around the Neverland seas? No. He read a map. And so will you in this problem. Happy hunting! Oh, and bring a torch because it’s dark in there.

**Program Input**

The file Prob16.in.txt will contain a single treasure map. Here is the key to reading the map and finding the treasure:

* H is the starting position of our treasure hunter (that’s you). Don’t get too lost!
* T is the treasure that you are trying to find. You win by stepping on this space.
* t is the symbol for a torch. You start out with 15 steps worth of torch light, and every torch you find gives you 15 more. The torches are unlit when you collect them, so assume you light them just as the previous torch is going out. So, picking up a torch on your first step would mean that you would have 29 more steps worth of light (15 to start + 15 for picking up a torch -1 for the first step). Each torch can only be collected and used once. You must step on a torch’s space to collect it.
* x is an immovable and impassable barrier, like a cave wall. You can’t step on an x.
* | and - denote the barrier of the cave. You cannot step on these spaces either. Don’t worry about how you got in the cave or how you’ll get out – finding the treasure will distract you from things like how to survive in a cave that apparently has no exit. ☺
* Stepping on a space more than once is permitted (and in some cases required – see the example problems). However, as previously mentioned stepping on a torch space twice will only give you extra light the first time.
* You can move up, down, left, and right only. Moving to a diagonal space takes two moves.

**Example Input:**

----------------------

|xxxxtx xxxxxxxxx|

|xxx x xxx Txxx|

|xx xx x xxxx|

|xx xxxxxxxxxx|

|xxx xxxxxxxxxxxxxxx|

|xxx xxx t xxxxx|

|xxx xxxx x xxx xxxx|

|xxtxxxx xxx|

|x xxxxxxxxxtxxxx|

|xxxxxtxxxxxxxx|

|xxx xx xxx xxxxx|

|xxxxxxxxxxxx|

|H xxxxxxxxxxxxx|

----------------------

**Program Output**

Your program should output a single integer: the least number of steps required to reach the treasure before (or just as) your light runs out. You can win the game by stepping on the treasure with your last step of light.

**Example Output:**

73

**Additional Maps**

While we were solving this problem, we found ourselves making maps to try out different situations. To spare you the difficulty of typing in a bunch of maps on your own, we have provided several of our test maps in the example input zip file. Below is a table containing the map names, the least number of steps, and the time it took our solution to run (just for reference, of course). The judging software does have time and memory constraints, so be careful. If you attempt this problem, we strongly suggest that you run these maps through your solution to make sure you can handle them. The judging map will strain your code more than any of these.

|  |  |  |
| --- | --- | --- |
| **Map Name** | **Least # Steps** | **Execution Time (seconds)** |
| SimpleTH.in.txt | 14 | 0.0 |
| SecondSimpleTH.in.txt | 8 | 0.0 |
| HardTH.in.txt | 67 | 0.46 |
| SuperHardTH.in.txt | 56 | 0.78 |
| LotsOfTorchesTH.in.txt | 46 | 0.312 |
| FourStepsTH.in.txt | 60 | 0.0 |
| UltraHardTH.in.txt | 119 | 0.983 |