Hallgatók tehetséggondozása és felkészítése egyetemi szintű, szakmai tanulmányi versenyekre az algoritmuselmélet és programozástechnika témakörében a 2011/2012. tavaszi félévben

„Tehetségtámogató programok az 5 kiemelt kutatási területen”

Szakmai dokumentáció, Engedy Balázs, 2011/1438.

Összefoglalás

A Számítástudományi és Információelméleti Tanszék immáron tíz éve szervez „ACM szakkör” néven olyan kiscsoportos foglalkozásokat, amelyek célja a legtehetségesebb hallgatóink algoritmuselmélet- és programozástechnika-tudásának, problémamegoldó készségének fejlesztése, illetve a hallgatók felkészítése napjaink legrangosabb programozási versenyeire. A szakkör hetente egy alkalommal, fakultatív jelleggel, órarenden kívüli foglalkozásként kerül megtartásra, nincsen – a szokásos értelemben vett – számonkérés vagy (formális) előkövetelmény. Ennek ellenére az érdeklődő hallgatók még a félév végéhez közeledve is szép számban látogatják a foglalkozásokat, illetve több más egyetemről is érkeznek résztvevők.

Részletes ismertetés

A szakkör célja, felépítése

A szakkör általános célja, hogy fejlessze a hallgatók készségét programozási versenyfeladatok önálló és csapatban történő megoldására. Egy versenyfeladat megoldása – egy általános mérnöki feladathoz hasonlóan – alapvetően három fázisra bontható:

1. a feladat értelmezése, a feladatban vázolt problémára egy absztrakt, matematikai modell felállítása,
2. a matematikai modellben egy hatékony algoritmus kidolgozása a probléma megoldására,
3. az algoritmus implementálása egy adott programnyelven.

Az első fázishoz jó modellezőképesség kell, amihez nélkülözhetetlen a diszkrét matematika egyes területeinek (pl. a gráfelméletnek) a mélyreható ismerete. A második fázishoz tipikusan gráfelméleti, algoritmuselméleti, számelméleti és csoportelméleti ismeretekre és matematikai gondolkodási képességre van szükség, a harmadik fázishoz pedig természetesen arra, hogy a hallgató precízen, gyorsan és hibamentesen tudjon programozni.

A foglalkozások során tipikusan a feladatmegoldás mindhárom fázisára hangsúlyt fektetünk. Ezzel összhangban a szakkör feladatmegoldó szeminárium jelleggel zajlik, azaz elsősorban a gyakorlatra fókuszál, és elméleti ismereteket, versenyekhez szükséges elméleti alaptudást a gyakorlati feladatokhoz kapcsolódóan ad át. Az óra elején három-négy gondosan összeválogatott, különböző nehézségű versenyfeladatot tartalmazó feladatsort osztunk ki a hallgatóknak. Ezután önálló munka következik, ennek során a szakkörvezető egyénileg konzultál a hallgatókkal, szükség esetén pótolja az elméleti hiányosságokat, apróbb ötleteket ad a feladat megoldásához. A foglalkozás végül a feladatok közös megbeszélésével az egyes megoldások összehasonlításával zárul. Amennyiben az idő engedi, egyes feladatokhoz az implementációt egy-egy on-line kiértékelőrendszerbe is feltöltjük ellenőrzésre.

A 2011/2012. tavaszi félév során összesen 10 alkalommal tartottunk 90-120 perc terjedelemben foglalkozást. A függelékben csatolva megtalálható az ebben a félévben megoldott minden feladat szövege.

Tematika

A szakkör során megbeszélt feladatok többsége napjaink három legrangosabb programozási versenyének archívumából származik, melyek tipikusan évente kerülnek megrendezésre, és több éves, évtizedes múltra tekintenek vissza. Név szerint:

1. „The ACM-ICPC Internat. Collegiate Programming Contest" – az IBM szervezi 1977 óta,
2. „Google Code Jam" – a Google 2008 óta rendezi,
3. „Topcoder Open, Topcoder SRM" – a Topcoder szervezi, előbbit 2003 óta évente, utóbbit pedig kéthetente (egyfordulós versenyek).

A versenyekhez szükséges elméleti alapismeretek – amely témákhoz a versenyfeladatok kapcsolódnak – az alábbiak:

- gráfreprezentációk és alapvető gráfalgoritmusok, többek között:
  - legrövidebbút-keresés, párosítások, folyamok, többszörös, illetve erősen összefüggőség, színezések, BFS, DFS, DAG-ok,
- dinamikus programozás, backtracking módszerek,
- mintaillesztés, szövegkeresés problémák,
- geometriai problémák,
- kombinatorika, csoportelmélet, számelmélettel kapcsolatos algoritmikus problémák.

Ezen kívül fontos még, hogy a versenyzőkben kialakuljon az a készség, hogy felismerje, egy adott feladathoz milyen tanult eszközt, ismert algoritmust használjon fel. Ennek érdekében egy adott foglalkozás keretében általában a legváltozatosabb problémákat állítjuk össze, és a résztvevőknek azt is feladatuk kitalálni, hogy a feladat szövegezésében milyen ismert probléma bújik meg.

Elért eredmények

A szakkörre az utóbbi félévekben járó hallgatók a 2011/2012. tavaszi félévben az alábbi eredményeket érték el:

- **1. hely + 2 különdíj – Sapientia-ECN nemzetközi programozási verseny**
  Csapattagok: Peregi Tamás, Berghammer Tamás, Mártonka Zoltán

Továbbfejlesztési lehetőségek

A külföldi tapasztalatokat is figyelembe véve úgy látjuk, hogy hallgatóink az egyes rangos nemzetközi versenyeken a jelenleginél is jobb eredményeket érhetnének el, a tehetséggondozásuk hatékonyabban lenne megvalósító, ennek elsődleges gátja, hogy a sok egyéb tanulmányi terhelés mellett elsősorban hallgatói oldalról nem áll rendelkezésre több idő a heti egyszeri (2 órás) foglalkozásnál.

Ez például oly módon lenne orvosolható, hogy a szakkörre járó hallgatók kreditpontokban részesülnének, ezáltal egyéb tanulmányi terhelésük csökkenne. Több más európai egyetemen ilyen rendszerben akár heti 6-8 órát is készülnek a versenyzők, és ennek köszönhetően előkelő eredményeket érnek el.

Függelék

A félév során megoldott feladatok

Problem Statement (BinaryCards)

Karel is a robot. He has a set of 64 cards: for each x between 0 and 63, he has a card that is blank on one side and has 2^x dots on the other side.

Karel's cards are placed on a table. At any moment, the cards show some integer between 0 and (2^64)-1, inclusive. To read the number, you just count all the dots you see.

Karel is using the cards to count from **A** to **B**. That is, he is flipping some of the cards in such a way that the numbers **A**, **A**+1, ..., **B** appear in this order.

Of course, Karel is using the shortest possible sequence of flips. Additionally, he always flips the cards one at a time. Sometimes, changing the number from some Z to Z+1 requires Karel to flip more than one card. In that case, he flips the necessary cards ordered by the number of dots they have, starting with the one with the most dots.

For example, if **A**=6 and **B**=8, the following will happen:

- In the beginning, the card with 4 dots and the card with 2 dots are showing the dots, all other cards are blank side up. This shows the number 6.
- Karel flips the card with 1 dot. Now the number 7 is shown.
- Karel flips the card with 8 dots.
- Karel flips the card with 4 dots.
- Karel flips the card with 2 dots.
- Karel flips the card with 1 dot. Now the number 8 is shown and Karel is done.

Given are longs **A** and **B**. Your method must return the largest number that will be shown at any moment during Karel's counting.

Definition

Class:	BinaryCards
Method:	largestNumber
Parameters:	long, long
Returns:	long
Method signature:	long largestNumber(long A, long B)

(be sure your method is public)

Constraints

- **A** will be between 1 and 10^18, inclusive.
- **B** will be between **A** and 10^18, inclusive.

Examples

0)
```
6

6
```
```
Returns: 6
```

1)
```
6

7
```
```
Returns: 7
```

2)
```
6

8
```
```
Returns: 15
```
This is the example from the problem statement. When flipping cards to create the number 8 from the number 7, Karel starts by flipping the card with 8 dots. At this moment, the number shown on the cards is 15.

3)
```
1

11
```
```
Returns: 15
```

4)
```
35

38
```
```
Returns: 39
```

5)
```
1125899906842630

1125899906842632
```
```
Returns: 1125899906842639
```

Problem B
Binary Search Tree

A binary search tree is a binary tree that satisfies the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

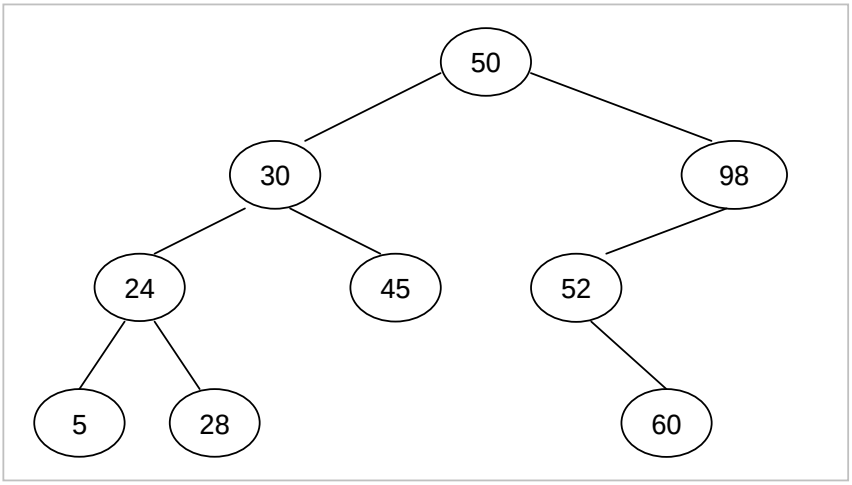


Figure 1. Example binary search tree

Pre-order traversal (Root-Left-Right) prints out the node's key by visiting the root node then traversing the left subtree and then traversing the right subtree. *Post-order traversal* (Left – Right-Root) prints out the left subtree first and then right subtree and finally the root node. For example, the results of pre-order traversal and post-order traversal of the binary tree shown in Figure 1 are as follows:

Pre-order: `50 30 24 5 28 45 98 52 60`
Post-order: `5 28 24 45 30 60 52 98 50`

Given the pre-order traversal of a binary search tree, you task is to find the post-order traversal of this tree.

Input
The keys of all nodes of the input binary search tree are given according to pre-order traversal. Each node has a key value which is a positive integer less than 10^6. All values are given in separate lines (one integer per line). You can assume that a binary search tree does not contain more than 10,000 nodes and there are no duplicate nodes.

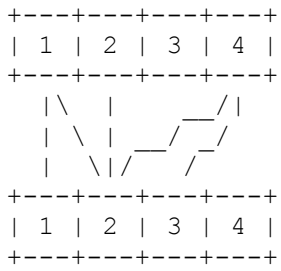
Output
The output contains the result of post-order traversal of the input binary tree. Print out one key per line.

Sample input	Sample output
50 30 24 5 28 45 98 52 60	5 28 24 45 30 60 52 98 50

Problem Statement (CircuitDesign)

A given circuit part is made of two rows of cells with a space in between them. Some of the cells of the top row need to be connected with some other cells of the bottom row with wires. A connection can be described as a pair of integers (x,y), which means that cell x of the top row needs to be connected with cell y of the bottom row.

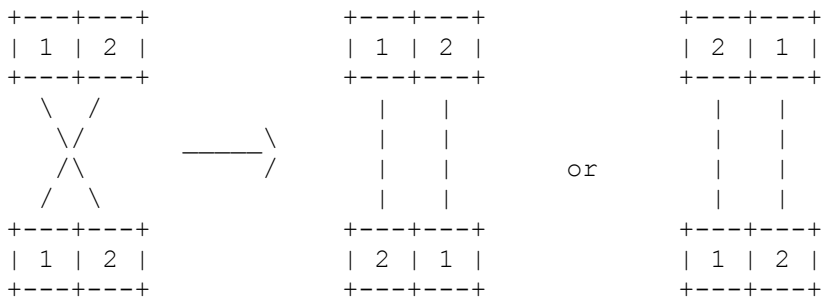
The following picture illustrates an arrangement with the connections (1,1) (1,2) (2,2) (4,2) (4,3).



Wires have to be printed in a 2-dimensional space and cannot cross each other, so if cell x1 of the top row needs to be connected to cell y1 of the bottom row and cell x2 of the top row needs to be connected with cell y2 of the bottom row, the following must hold:

- (x1<=x2 and y1<=y2) or (x1>=x2 and y1>=y2).

Fortunately, the scientists noted that they may exchange cells placed on the same row. For instance, the following list of connections would not be valid according to the above rule: (1,2) (2,1). But, if they place the top row in the regular order [1,2] and the bottom row in reverse order [2,1], then no wires cross and the arrangement is valid. Moreover, they could also achieve that by placing the top row in reverse order and the bottom row in regular order. The following picture illustrates the situation:



You will be given an int **n** that represents the number of cells in each row, and the list of connections as two int[]s **top** and **bottom**. For each i, the pair (**top**[i],**bottom**[i]) is a required connection. Return the number of possible arrangements of the cells in each row that result in no wires crossing, modulo 1000000007. Two arrangements are different if any cell in any row has a different placement. See examples for further clarification.

Definition

Method signature:	int countPerms(int n, int[] top, int[] bottom)

Constraints

- **n** will be between 1 and 50, inclusive.
- **top** will contain between 1 and 50 elements, inclusive.
- t**op** and **bottom** will contain the same number of elements.
- Each element of **top** and **bottom** will be between 1 and **n**, inclusive.
- Each pair (**top**[i] , **bottom**[i]) will be different.

Examples

0)
```
4

{1,1,2,4,4}

{1,2,2,2,3}

Returns: 32
```
This is the first example drawn in the problem statement.

1)
```
2

{1,2}

{2,1}

Returns: 2
```
The second example drawn in the problem statement.

2)
```
3

{1,1,1,2,2,2,3,3,3}

{1,2,3,1,2,3,1,2,3}

Returns: 0
```
There are too many wires to avoid crossing.

3)
```
30

{5,5,5,5,5,5,5,5,5, 5,11,12,13,14,15,16,
17,18,19,20,21,22,23,24,25,26,27,28,29,30}

{1,2,3,4,5,6,7,8,9,10,15,15,15,15,15,15,
Ű15,15,15,15,25,25,25,25,25,25,25,25,25,25}

Returns: 628573100
```

4)
```
5

{1,2,3,4,5,1,2,3,4}

{1,2,3,4,5,5,1,4,2}

Returns: 2
```

Problem Statement (ColorfulCookie)

Little Fox Jiro received a box from his sister Fox Ciel as a present. The box contained many cookies of various colors. Jiro wants to eat as many of these cookies as possible. The cookies are locked in a strange box. Jiro cannot take cookies from the box directly. On the box there is a dial and a button. These can be used to obtain cookies in the following way:

1. Jiro uses the dial to choose any pair of distinct colors C1 and C2.
2. Jiro pushes the button. If there are less than **P1** cookies of color C1, nothing happens. Also, if there are less than **P2** cookies of color C2, nothing happens. Otherwise, exactly **P1** cookies of color C1 and exactly **P2** cookies of color C2 drop out of the box and Jiro eats all of them.

You are given a int[] **cookies** describing the initial contents of the box: for each i, the box contains **cookies**[i] cookies of color i. You are also given the ints **P1** and **P2**. Your method must return the maximum number of cookies Jiro can eat by repeating above procedure optimally.

Definition

Class:	ColorfulCookie
Method:	getMaximum
Parameters:	int[], int, int
Returns:	int

Method signature: int getMaximum(int[] cookies, int P1, int P2)

(be sure your method is public)

Constraints

- **cookies** will contain between 1 and 50 elements, inclusive.
- Each element of **cookies** will be between 1 and 2,000, inclusive.
- **P1** and **P2** will each be between 50 and 2,000, inclusive.

Examples

0)

```
{100, 100}
50
50
Returns: 200
```

The optimal solution is to select colors 0 and 1 and to push the button twice: each time obtaining 50 cookies of each color.

1)

```
{50, 250, 50}
50
100
```

```
Returns: 300
```

An optimal solution:

1. Pick colors 0 and 1 (note that order matters) and push the button to obtain 50 cookies of color 0 and 100 cookies of color 1.
2. Pick colors 2 and 1 (again, note the order) and push the button to obtain 50 cookies of color 2 and 100 cookies of color 1.

This gives Jiro a total of 300 cookies. Note that 50 cookies of color 1 remained in the box, but there is no way to get them out.

2)

```
{2000}
100
200
Returns: 0
```

In this case all cookies have the same color. It is impossible to obtain any of them.

3)

```
{123, 456, 789, 555}
58
158
Returns: 1728
```

Problem Statement (DecodeDigits)

A simple way to encode a word into a string of digits is to replace each letter by its order in the alphabet. That is, "a" will change to "1", "b" to "2", ..., and "z" to "26". For example, encode("cow")="31523" and encode("cat")="3120".

Sadly, this encoding cannot always be uniquely decoded, because two different words can yield the same string of digits when encoded. For example, encode("beard")=encode("yard")="251184".

String A is a subsequence of string B if it is possible to erase some letters of B (possibly none, possibly all of them) to obtain A. For example, "cage" is a subsequence of "cabbages".

You are given a String **D** containing a string of digits. If there is no string Y such that encode(Y)=**D**, return the String "NONE". Otherwise, find and return the longest string X with the following property: whenever encode(Y)=**D**, X is a subsequence of Y. If there are multiple such strings, return the lexicographically smallest one.

Definition

Class:	DecodeDigits
Method:	solve
Parameters:	String
Returns:	String
Method signature:	String solve(String D)

(be sure your method is public)

Constraints

-**D** will contain between 1 and 50 characters, inclusive.

-Each character in **D** will be a digit ('0'-'9').

Examples

0)
```
"38956"
```

```
Returns: "chief"
```
There is only one way to decode this string of digits.

1)
```
"13919156"
```

```
Returns: "if"
```
This string of digits can be decoded in 8 different ways. Each of them contains an "i" followed by an "f".

2)
```
"1122"
```

```
Returns: ""
```
We have encode("kbb")=encode("aav")="1122". The strings "kbb" and "aav" have no common subsequence other than the empty one.

3)
"3120"

Returns: "cat"
The only valid decoding of this string is "cat".

4)
"0"

Returns: "NONE"

Problem Statement (EllysFractions)

A positive common fraction is a fraction of the form A/B, where A and B are positive integers. An irreducible fraction is a positive common fraction such that A and B are relatively prime. (In other words, the only positive integer that divides both A and B is 1.)

Elly recently invented the *factor fractions*: A factor fraction is an irreducible fraction such that the product A*B is a factorial number (see Notes for definition). We will only be interested in factor fractions that lie strictly between 0 and 1. (That is, A must be strictly smaller than B.)

Examples:

- 2/3, 4/6, 4/7, 7/7, 6/1, 42/49 are six positive common fractions
- Out of those six, the following three are irreducible: 2/3, 4/7, 6/1.
- The fraction 2/3 is a factor fraction, because 2*3 = 6 and that is a factorial number.
- The fraction 4/7 is not a factor fraction, because 4*7 = 28 and that is not a factorial number.
- The fraction 6/1 is a factor fraction, because 6*1 = 6 and that is a factorial number. However, as 6/1 does not lie strictly between 0 and 1, we are not interested in this fraction.
- Note that 4/6 is not a factor fraction. (We do have 4*6=24, but a factor fraction has to be irreducible.)

You are given an int **N**. Compute and return the number of factor fractions A/B such that 0 < A/B < 1 and A*B is one of the factorial numbers from 1! to **N**! (inclusive).

Definition

Class:	EllysFractions
Method:	getCount
Parameters:	int
Returns:	long
Method signature:	long getCount(int N)
(be sure your method is public)	

Notes

- The factorial of X, denoted X!, is the product of the first X positive integers. For example, 6! is 1*2*3*4*5*6 = 720. The factorial numbers are the numbers of the form X! for positive integer X. The smallest few: 1, 2, 6, 24, 120, 720, …
- The answer will always fit in a 64-bit signed integer.

Constraints

- **N** will be between 1 and 250, inclusive.

Examples

0)
```
1
```

```
Returns: 0
```
We are interested in factor fractions such that A*B = 1. The only positive common fraction with this property is the fraction 1/1. However, this fraction is not strictly between 0 and 1.

1)
```
2
```

```
Returns: 1
```
Here the only valid factor fraction is 1/2.

2)
```
3
```

```
Returns: 3
```
The three fractions are 1/2, 1/6, and 2/3. (We have 1*2 = 2! and 1*6 = 2*3 = 3!.)

3)
```
5
```

```
Returns: 9
```

4)
```
100
```

```
Returns: 177431885
```
Plenty of fractions here.

Problem Statement (EllysJuice)

Elly's memories of last night's sleepover with her friends are rather blurry. She recalls that they played a game involving drinking orange and apple juice. In the beginning there were two identical bottles. One contained a gallon of orange juice, the other a gallon of apple juice. The game was played in turns, starting with turn zero. In even-numbered turns (turn 0, 2, 4, ...) the current player drank half of the remaining orange juice. In odd-numbered turns (turn 1, 3, 5, ...) the current player drank half of the remaining apple juice. The winner of the game is the player who drank the greatest total amount of liquid during the game. If multiple players are tied for the greatest amount, there is no winner.

For example, consider a game consisting of five turns. Suppose that the players to drink were, in order, Elly, Kriss, Stancho, Elly, and Stancho. (That is, Elly and Stancho both drank twice: Elly in turns 0 and 3, Stancho in turns 2 and 4.) In this game, Elly drank 0.75 gallons of liquid (0.5 gallons of orange juice and 0.25 of apple juice), Kriss drank 0.5 gallons of apple juice, and Stancho drank 0.375 gallons of orange juice (0.25 gallons in turn 2 and another 0.125 gallons in turn 4). In this game Elly would be the winner.

Elly knows all the people who played the game yesterday, and for each of them the number of times they drank. However, she remembers nothing about the order in which the players drank.

You are given a String[] **players**, containing the names of all players. If a player's name is contained X times in **players** that means that he or she drank exactly X times during the game. Determine all people who could have possibly won the game (for some particular order of turns). Return their names as a lexicographically sorted String[]. The return value should only contain each of the names once.

Definition

Class:	EllysJuice
Method:	getWinners
Parameters:	String[]
Returns:	String[]
Method signature:	String[] getWinners(String[] players)

(be sure your method is public)

Notes

A sequence of distinct Strings is sorted if each element is lexicographically smaller than all the ones following it.-String A is lexicographically smaller than String B if A contains smaller character at the first index where they differ. If there is no index where they differ, the shorter one is lexicographically smaller. For example "ariadne" is smaller than "elly", "abc" is smaller than "abd", "aaa" is smaller than "aaaaa", and "czzzzzzz" is smaller than "kaaaaaaa".

Constraints

- **players** will contain between 1 and 8 elements, inclusive.
- Each element of **players** will contain between 1 and 20 characters, inclusive.
- Each element of **players** will consist only of lowercase English letters ('a'-'z').

Examples

0)
```
{ "elly", "kriss", "stancho", "elly", "stancho" }
```

```
Returns: {"elly", "stancho" }
```
Elly would win for the order given in the problem statement. If Stancho and Elly swapped all their turns, Stancho would win the game. Kriss cannot win.

1)
```
{"torb", "elly", "stancho", "kriss"}
```

```
Returns: { }
```
In any valid game two of the four players would be tied for the lead (with half a gallon each).

2)
```
{"elly", "elly", "elly", "elly", "elly"}
```

```
Returns: {"elly" }
```
Since she was the only player, apparently she won.

3)
```
{ "ariadne", "elly", "ariadne", "stancho", "stancho", "kriss", "stancho", "elly" }
```

```
Returns: {"ariadne", "elly", "stancho" }
```
Although Stancho has drunk three times, there are orderings in which someone else wins.

Problem Statement (EllysLights)

Elly's house has a number of light bulbs. Some of them are turned on, some of them are not. She wants to switch all of them off for the upcoming Earth Hour.

Unfortunately, her house has a strange system of light switches. Each switch changes the state of some of the bulbs (possibly none or all of them). This means that if she uses the switch, all bulbs that are connected to it go off if they were on, and go on if they were off. She also knows that each bulb is connected to at most two switches. Help Elly determine what is the minimal number of switches she must use in order to turn off all the lights.

You are given a String **initial** and a String[] **switches**. Each character in **initial** represents one light bulb: if the i-th light bulb (0-based index) is initially on, the i-th character of **initial** is 'Y', otherwise it is 'N'. Each element of **switches** represents one switch: if the i-th switch toggles the j-th light bulb, **switches**[i][j] is 'Y', otherwise it is 'N'.

Find and return the smallest nonnegative integer X such that there is a sequence of X switches such that if Elly toggles those switches in the given order, all light bulbs will be turned off at the end. If no such X exists, return -1.

Definition

Class:	EllysLights
Method:	getMinimum
Parameters:	String, String[]
Returns:	int
Method signature:	int getMinimum(String initial, String[] switches)

(be sure your method is public)

Notes

-It is possible that none or all of the bulbs are turned on initially.-Elly is allowed to toggle the same switch more than once.

Constraints

- **initial** will contain between 1 and 50 characters, inclusive.
- Each character of **initial** will be either 'Y' or 'N'.
- **switches** will contain between 1 and 50 elements, inclusive.
- Each element of **switches** will contain the same number of characters as **initial**.
- Each character of each element of **switches** will be either 'Y' or 'N'.
- For each index of **initial** there will be at most two elements of **switches** that will have 'Y' at that index.

Examples

0)
```
"YNYNNN"

{"YNNYNY", "NYYYNN", "YNYNYN", "NNNNYN", "NYNNNY"}
```
Returns: 2
There are multiple ways how to turn off all the lights. For example, she can first use switch 0, then switch 4, and finally switch 1. The shortest solutions only use two switches. For example, it is enough to flip the switch 2 followed by the switch 3.

1)
```
"YNYNYN"

{"NNNNNN", "YYYYYY", "NYNNNN", "NNNYNN", "NNNNNY"}
```
Returns: 4
Some of the switches might change the state of none or all of the bulbs. Some of the bulbs might be influenced by less than 2 switches.

2)
```
"YYN"

{"YNY", "NYN"}
```
Returns: -1
Sometimes there might be no way to switch off all the bulbs. In this case return -1.

3)
```
"NNYNYNYYYNNYYYYN"

{"NYNYNYNYNYNYNYNY",
 "YNYNYNYNYNYNYNYN",
 "NNNNNNNNNNNNNNNN",
 "YNNNNNNNNNNNNNNN",
 "NYNNNNNNNNNNNNNN",
 "NNYNNNNNNNNNNNNN",
 "NNNYNNNNNNNNNNNN",
 "NNNNYNNNNNNNNNNN",
 "NNNNNYNNNNNNNNNN",
 "NNNNNNYNNNNNNNNN",
 "NNNNNNNYNNNNNNNN",
 "NNNNNNNNYNNNNNNN",
 "NNNNNNNNNYNNNNNN",
 "NNNNNNNNNNYNNNNN",
 "NNNNNNNNNNNYNNNN",
 "NNNNNNNNNNNNYNNN",
 "NNNNNNNNNNNNNYNN",
 "NNNNNNNNNNNNNNYN",
 "NNNNNNNNNNNNNNNY"}
```
Returns: 6

There are 9 lit bulbs initially. We can turn them off one by one, but it will be faster to use one of the complex switches and then to toggle the ones that are still lit.

Problem Statement

Cucumberman built a maze. The maze consists of rooms and one-directional corridors. Each corridor leads from one room to another. Rooms are numbered 0 through N-1, where N is the number of rooms. The maze has a special property: for each room X, once you leave room X (by using one of the corridors), you will never be able to get back to room X.

The layout of the maze is described by a String[] **maze**. The j-th character of the i-th element of **maze** will be 'Y' if there is a one-directional corridor from room i to room j, and it will be 'N' otherwise. You are also given a String **rooms**. Some rooms may contain an obstacle. A room with an obstacle cannot be entered. If the i-th character of **rooms** is '?', room i may contain an obstacle. If it is '-', room i doesn't contain an obstacle.

Cucumberman is interested in the number of paths from room 0 to room 1 (see Notes for a formal definition). If the number of different paths is even, he calls the maze *nice*. There are 2^K possible states of the maze, where K is the number of '?'s in **rooms**. Out of these 2^K mazes, some are *nice*. Find and return their count.

Definition

Class:	EvenPaths
Method:	theCount
Parameters:	String[], String
Returns:	long
Method signature:	long theCount(String[] maze, String rooms)
(be sure your method is public)	

Notes

A path from room 0 to room 1 is a sequence of rooms that satisfies the following conditions:

1) The first element of the sequence is room 0 and the last element of the sequence is room 1.

2) For each element of the sequence (except for the last element), there is a one-directional corridor to the next element.

3) No room in the sequence contains an obstacle.

Constraints

- **maze** will contain between 2 and 50 elements, inclusive.
- Each element of **maze** will contain N characters, where N is the number of elements of **maze**.
- Each character in **maze** will be 'Y' or 'N'.
- **maze** will satisfy the property from the problem statement. I.e., there is no sequence of corridors that leads from some room X back to the same room X.
- **maze** will contain at most 500 'Y'.

- **rooms** will contain N characters.
- Each character in **rooms** will be '-' or '?'.
- The number of '?' in **rooms** will be between 0 and 32, inclusive.
- The first two characters of **rooms** will be '-'.

Examples

0)
```
{"NYY", "NNN", "NYN"}

"--?"

Returns: 1
```
If room 2 contains an obstacle, there is one path from room 0 to room 1: 0 -> 1. This is not a *nice* maze.

If room 2 doesn't contain an obstacle, there are two paths from room 0 to room 1: 0 -> 1 and 0 -> 2 -> 1. This is a *nice* maze.

1)
```
{"NYYNN", "NNNNY", "NYNNN", "YNNNN", "NNNNN"}

"--???"

Returns: 4
```
The maze is nice if and only if room 2 doesn't contain an obstacle.

2)
```
{"NNNNN", "NNYYN", "YNNNY", "NNNNN", "NNNNN"}

"--???"

Returns: 8
```
There is no path from room 0 to room 1 regardless of obstacles.

3)
```
{"NNYNYNNY", "NNNNNNNN", "NNNNNYYY", "NYNNNNNN", "NNNNNNNN", "NYNYNNNN", "NYNNYNNN", "NNNYYN

"----???-"

Returns: 4
```

Problem Statement (FoxAndPhotography)

Fox Ciel is attending a party. During the party, people decided to take a commemorative photograph. Ciel picked up a camera to take a photograph. In the party, there are 2*N people (not including Ciel).

For taking the photograph, people formed two rows, each containing N people. In each row, Ciel numbered the people from 0 to N-1 from left to right. The people were aligned so that for each i, person i in the back row was standing exactly behind person i in the front row.

Initially, the people lined up arbitrarily. Therefore it is possible that some of the people in the back row are obscured by people in the front row. You are given two int[]s: **heightsFront** and **heightsBack**. For each i, **heightsFront**[i] is the height of person i in the front row and **heightsBack**[i] is the height of person i in the back row. The person in the back row is obscured by the person in the front row if and only if **heightsFront**[i] >= **heightsBack**[i].

If there are some obscured people, Ciel wants to fix it. The only operation she may do is to select two adjacent people in the same row and swap them. Compute and return the smallest number of such operations necessary to rearrange the people so that nobody in the back row is obscured. If no such sequence of operations exists, return -1.

Definition

Class:	FoxAndPhotography
Method:	getMinimumSwaps
Parameters:	int[], int[]
Returns:	int
Method signature:	int getMinimumSwaps(int[] heightsFront, int[] heightsBack)

(be sure your method is public)

Constraints

- **heightsFront** will contain between 2 and 16 elements, inclusive.
- **heightsBack** will contain the same number of elements with **heightForward**.
- Each element of **heightsFront** and **heightsBack** will be between 140 and 190, inclusive.

Examples

0)
```
{140, 150}

{160, 150}

Returns: 1
```
Initially, person 1 (0-based index) in the back row is obscured. One optimal solution is to swap people 0 and 1 in the front row. Another optimal solution is to swap people 0 and 1 in the back row.

1)
```
{140, 140, 140, 140}

{190, 190, 190, 190}

Returns: 0
```
Nobody is obscured, so no swaps are needed.

2)
```
{170, 170, 170}

{160, 170, 180}

Returns: -1
```
Person 0 in the back row is too short. Regardless of how we swap the people, this person will always be obscured.

3)
```
{140, 141, 142, 143}

{144, 143, 142, 141}

Returns: 6
```

4)
```
{140, 170, 140, 170, 140, 170, 140, 170, 140, 170}

{180, 180, 180, 180, 180, 150, 150, 150, 150, 150}

Returns: 15
```

Problem Statement (ImageTraders)

There is a community of art lovers who meet from time to time and trade images with each other. Each image transaction must obey the following rules:

1. The price at which the image is sold must be greater than or equal to the price at which it was bought.
2. No trader is allowed to buy the same image more than once.

A new image has just appeared on the market. Trader 0 bought it from an unknown artist for the price of '0', and he is now ready to trade it among his fellow art lovers. You are given a String[] **price**, where the *j*-th character of the *i*-th element is a digit representing the price at which trader *j* will buy the image from trader *i*. '0' is the lowest price, and '9' is the highest. Assuming all transactions obey the rules mentioned above, determine the longest possible sequence of transactions involving the new image. After all the transactions are done, return the number of people who have owned the image at some point in time, including both the final owner and trader 0.

Definition

Class:	ImageTraders
Method:	maxOwners
Parameters:	String[]
Returns:	int
Method signature:	int maxOwners(String[] price)
(be sure your method is public)	

Constraints

- **price** will contain exactly N elements, where N is between 2 and 15, inclusive.
- Each element of **price** will contain exactly N characters.
- Each element of **price** will contain only digits ('0'-'9').

Examples

0)

```
{
"01",
"10"
}
```

```
Returns: 2
```

Trader 0 can sell the image to trader 1 for the price of '1'. Both traders were owners of the image, so the answer is 2.

1)

```
{
"022",
"101",
"110"
}
```

```
Returns: 2
```

We have 3 traders here. Trader 0 can sell the image to either trader 1 or trader 2 for the price of '2'. In both cases, the buyer would have to sell the image for a price of at least '2', which is impossible. Therefore, the maximal number of owners is 2.

2)

```
{
"01231",
"00231",
"00031",
"00002",
"00000"
}
```

```
Returns: 4
```

Here the image can have 4 owners:

- Trader 0 sells the image to trader 1 for the price of '1'.
- Trader 1 sells the image to trader 2 for the price of '2'.
- Trader 2 sells the image to trader 3 for the price of '3'.

3)

```
{
"15555",
"11111",
"15111",
"11111",
"11111"
}
```

```
Returns: 3
```

Trader 0 can sell the image to trader 2 for the price of '5'. Then, trader 2 can sell it to trader 1.

4)

```
{
"0100000000",
"0020000000",
"0003000000",
"0000400000",
"0000050000",
"0000006000",
"0000000700",
"0000000080",
"0000000009",
"1111111111"
}
```

```
Returns: 10
```

The image can be traded among all the traders in the following sequence: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Problem Statement (ImportantSequence)

Little Rudolph had an important sequence of positive integers. The sequence consisted of N positive integers a_0, a_1, .., a_{N-1}.

Rudolph wrote the sequence onto the blackboard in the classroom. While Rudolph had gone out, little Arthur came into the classroom and saw the sequence. Arthur likes to play with numbers as much as he likes to give his friends puzzles. So he did the following:

- First, he wrote a '+' or a '-' between each pair of consecutive numbers (possibly using different signs for different pairs of numbers).
- Next, for each sign he computed the result of the corresponding operation and wrote it under the sign. I.e., if he used the '+' sign between a_i and a_{i+1}, he would write the sum a_i+a_{i+1} under this '+' sign. Similarly, if he used the '-' sign between a_i and a_{i+1}, he would write the difference a_i-a_{i+1}. In this way he obtained a new sequence of N-1 numbers b_0, b_1, .., b_{N-2}.
- Finally, he erased the original sequence. Now there was only the operator sequence o_0, o_1, .., o_{N-2} and the resulting number sequence b_0, b_2, .., b_{N-2} left on the blackboard.

For example, if the original sequence was {1, 2, 3, 4}, and Arthur wrote operators {+, -, +}, then the content of the blackboard changed like this:

```
1   2   3   4    ->    1 + 2 - 3 + 4    ->    1 + 2 - 3 + 4    ->     +  -  +
                                                 3  -1   7             3 -1  7
```

When Rudolph returned, he was shocked as his important sequence had disappeared. Arthur quickly told him what operations he had performed and that Rudolph has to simply reconstruct the orginal sequence.

Unfortunately, little Arthur did not realize that it is not necessarily possible to determine the original sequence uniquely. For example, both original sequences {1, 2, 3, 4} and {2, 1, 2, 5} lead to the same sequence {3, -1, 7} when operator sequence is {+, -, +}.

The only thing Rudolph remembers about his original sequence is that all the integers were positive. Rudolph now wants to count all sequences of positive integers that match the blackboard. You are given int[] **B** and String **operators** both containing N-1 elements. The i-th element of **B** is the number b_i and i-th element of **operators** will be '+' or '-', meaning that the i-th operator is + or -, respectively.
Return the number of different positive integer sequences A that lead to sequence B when operators **operators** are used in the way described. If there are infinitely many such sequences, return -1. Note that there may be test cases where no valid sequence A exists. For such test cases the correct return value is 0.

Notes

It is guaranteed that the correct answer will always fit into the 32-bit signed integer type.

The integer 0 (zero) is not positive. It may not occur in Rudolph's original sequence.

Constraints

- **B** will contain between 1 and 50 elements, inclusive.
- **operators** will contain the same number of characters as the number of elements in **B**.
- Each element of **B** will be between -1000000000 (-10^9) and 1000000000 (10^9), inclusive.
- Each character in **operators** will be either '+' or '-' (quotes for clarity).

Examples

0)

```
{3, -1, 7}

"+-+"

Returns: 2
```

From the problem statement.

1)

```
{1}

"-"

Returns: -1
```

There are infinitely many pairs of positive integers that differ by one.

2)

```
{1}

"+"

Returns: 0
```

Note that all numbers a_i have to be positive integers.

3)

```
{10}

"+"

Returns: 9
```

4)

```
{540, 2012, 540, 2012, 540, 2012, 540}

"-+-+-+-"

Returns: 1471
```

Problem Statement (MagicBoard)

Magic Board is a lock that has the shape of a rectangular board, divided into a grid of n × m unit squares. Some of the unit squares contain a Magic Diamond. You are given the contents of the Magic Board as a String[] **board**. The number of elements in **board** is n, and the length of each element is m. Empty squares are denoted by periods ('.'), squares with a Magic Diamond are denoted by '#'.

Let S = (S[0], S[1], ...) be a sequence of cells on the board. We will consider the cells in S one after another, picking up a Magic Diamond from each of them. Doing this will unlock the Magic Board if and only if S has the following properties:

- S contains each cell with a Magic Diamond exactly once.
- S does not contain any other cells.
- For each even i, S[i] and S[i+1] are in the same row of **board**.
- For each odd i, S[i] and S[i+1] are in the same column of **board**.

You are given the String[] **board**. Determine whether this Magic Board can be unlocked. Return "YES" (quotes for clarity) if it can be unlocked and "NO" otherwise.

Definition

Class:	MagicBoard
Method:	ableToUnlock
Parameters:	String[]
Returns:	String
Method signature:	String ableToUnlock(String[] board)

(be sure your method is public)

Constraints

-**board** will contain between 1 and 50 elements, inclusive.

-**board**[0] will contain between 1 and 50 characters, inclusive.

-All elements of **board** will contain the same number of characters.

-Each character in each element of **board** will be either '.' or '#'.

-**board** will contain at least one '#'.

Examples

0)

```
{"##",
 ".#"}
```

```
Returns: "YES"
```

The only solution: Start in the upper left corner, then move to the right, and finally move down.

1)
```
{"#.",
 ".#"}
```

```
Returns: "NO"
```
In this case there is no solution. Regardless of which Magic Diamond we start with, we will not be allowed to take the other one, as it is not in the same row.

2)
```
{"##",
 "##",
 "##"}
```

```
Returns: "YES"
```

3)
```
{"###",
 "###",
 "###"}
```

```
Returns: "NO"
```

4)
```
{"###",
 "..#",
 "###",
 "#..",
 "###"}
```

```
Returns: "NO"
```

5)
```
{".................",
 ".######..######.",
 ".######..######.",
 ".##......##..##.",
 ".##......##..##.",
 ".######..######.",
 ".######..######.",
 ".....##..##..##.",
 ".....##..##..##.",
 ".######..######.",
 ".######..######.",
 "................."}
```

```
Returns: "YES"
```

6)
```
{"#"}
```

```
Returns: "YES"
```

Problem Statement (MismatchedStrings)

Well-parenthesized strings are defined using the following rules:

- The empty string is well-parenthesized.
- If `S` is a well-parenthesized string, then `(S)` is a well-parenthesized string.
- If `S` and `T` are well-parenthesized strings, then `ST` is a well-parenthesized string.
- Every well-parenthesized string can be created using the previous rules only.

In this problem we will deal with the complement of this set of strings: The strings that consist only of the characters '`(`' and '`)`', but are not well-parenthesized. We will call these strings *mismatched*.

You are given an int **N** and a long **K**. All mismatched strings of length **N** can be ordered lexicographically, and numbered sequentially, starting with zero. Return the string that will get the number **K** in this order. If there is no such string, return the empty string instead.

Definition

Class:	MismatchedStrings
Method:	getString
Parameters:	int, long
Returns:	String
Method signature:	String getString(int N, long K)

(be sure your method is public)

Notes

- Given two different strings S and T of equal length, S is lexicographically smaller than T if it has a smaller character at the first position where they differ.
- The ASCII value of '(' is less than the ASCII value of ')'.

Constraints

- **N** will be between 1 and 50, inclusive.-**K** will be between 0 and 2^**N** - 1, inclusive.

Examples

0)

```
4

0
```

```
Returns: "((((" 
```

For any length, the lexicographically smallest mismatched string consists of '(' characters only.

1)

```
4

4
```

```
Returns: "())("
```

There are 14 mismatched strings of length 4. The first five are "`(((`", "`(((`)", "`((`)`(`", "`(`)`((`", and "`)))(`". Note that we use a 0-based index, hence **K**=4 corresponds to the fifth mismatched string.

2)

```
6

63
```

```
Returns: ""
```

There are less than 64 mismatched strings of length 6.

3)

```
7

13
```

```
Returns: "((()))()"
```

Problem Statement (MuddyRoad)

Fox Ciel is going to walk through an unpaved road to meet her friend. The road is one-dimensional. It is separated into N parts numbered from 0 to N-1, where part 0 is Ciel's current part and part N-1 is her destination part.

Ciel will perform her trip tomorrow. Unfortunately today it is raining, so tomorrow some parts of the road will be muddy. You are given a int[] **road** containing N elements. The probability that the i-th part of the road will be muddy tomorrow is **road**[i]/100.

Ciel can walk along the road using any combination of strides with lengths 1 and 2. If she is at part i, a stride of length 1 will move her to part i+1 and a stride of length 2 will move her to part i+2 (skipping part i+1). If there are many ways to reach part N-1 from part 0, Ciel will choose the one among them where the number of visited muddy parts is minimal.

Return the expected number of muddy parts that she will visit tomorrow.

Definition

Class:	MuddyRoad
Method:	getExpectedValue
Parameters:	int[]
Returns:	double
Method signature:	double getExpectedValue(int[] road)

(be sure your method is public)

Notes

- Assume that events "i-th part of the road will be muddy tomorrow" are totally independent.
- Ciel has very good sight, so when starting her trip at part 0, she is already able to see for each part whether it is muddy or not.

Constraints

- **road** will contain between 2 and 50 elements, inclusive.
- Each element of **road** will be between 0 and 100, inclusive.
- The first element and the last element of **road** will be 0.

Examples

0)

```
{0, 60, 60, 0}
```

```
Returns: 0.36
```

There can be four different states of the road tomorrow:

```
.... with probability = 0.16, 0 steps to muddy parts
.M.. with probability = 0.24, 0 steps to muddy parts
..M. with probability = 0.24, 0 steps to muddy parts
.MM. with probability = 0.36, 1 step to muddy parts
```

(Here, '.' represents a non-muddy part and 'M' represents a muddy part.)

Thus, the expected number of steps is 0*0.16+0*0.24+0*0.24+1*0.36=0.36.

1)

```
{0, 50, 50, 50, 50, 0}
```

```
Returns: 0.5625
```

2)

```
{0, 0, 100, 100, 100, 100, 100, 100, 0, 0, 100, 0}
```

```
Returns: 3.0
```

3)

```
{0, 12, 34, 56, 78, 91, 23, 45, 67, 89, 0}
```

```
Returns: 1.7352539420031923
```

4)

```
{0, 50, 50, 100, 50, 100, 50, 50, 100, 66, 0}
```

```
Returns: 2.288125
```

G
Non-negative Partial Sums

You are given a sequence of n numbers $a_0, \ldots, a_{n-1}$. A cyclic shift by k positions ($0 \leq k \leq n-1$) results in the following sequence: $a_k, a_{k+1}, \ldots, a_{n-1}, a_0, a_1, \ldots, a_{k-1}$. How many of the n cyclic shifts satisfy the condition that the sum of the first i numbers is greater than or equal to zero for all i with $1 \leq i \leq n$?

Input

Each test case consists of two lines. The first contains the number n ($1 \leq n \leq 10^6$), the number of integers in the sequence. The second contains n integers $a_0, \ldots, a_{n-1}$ ($-1000 \leq a_i \leq 1000$) representing the sequence of numbers. The input will finish with a line containing 0.

Output

For each test case, print one line with the number of cyclic shifts of the given sequence which satisfy the condition stated above.

Sample Input

```
3
2 2 1
3
-1 1 1
1
-1
0
```

Sample Output

```
3
2
0
```

Problem Statement (NumberGrid)

A *number grid* is a rectangular matrix where each cell is a digit between 0 and 9, inclusive.

Given a number grid G, an *encrypted grid* E with the same dimensions can be generated using the following procedure (using zero-based indexing):

```
for(i=0;i<rows;++i){
  for(j=0;j<cols;++j){
    E_{i,j} = 0
    for(r=0;r<rows;++r){
      for(c=0;c<cols;++c){
        if( (r equals i) or (c equals j))
        {E_{i,j} = E_{i,j} + G_{r,c}}
      }
    }
  }
}
```

Note that this means each cell in E is the sum of all the cells in G that positionally share the same row or column. (The cell in G with both the same row and same column is only counted once.)

An example number grid G is given below:

```
-------------
| 0 | 1 | 3 |
-------------
| 2 | 5 | 7 |
-------------
```

Performing the above procedure yields the encrypted grid below:

```
-----------------
| 6  | 9  | 11 |
-----------------
| 14 | 15 | 17 |
-----------------
```

The cell in row 0, column 1 of E, for example, is calculated by taking the sum of all cells in G with either row 0 or column 1 (0 + 1 + 3 + 5 = 9). Note that the '1' cell is counted only once.

Given an encrypted grid E, reconstruct the original number grid G if possible.

The encrypted grid is given in three separate String[]s. **hundreds** gives the digit in the hundreds place of each cell in the encrypted grid, **tens** gives the digit in the tens place, and **ones** gives the digit in the ones place. In each of these String[]s, the j-th character of the i-th element represents the cell in row i, column j.

For example, if the encrypted grid appeared as follows:

```
------------------
| 106 | 72  | 5  |
------------------
| 14  | 135 | 17 |
------------------
```

The input would appear like this:

hundreds	**tens**	**ones**
{"100", "010"}	{"070", "131"}	{"625", "457"}

If there is exactly one valid number grid G that yields the given encrypted grid, return a String[] representing G, where the j-th character of the i-th element is the digit in row i, column j. If there are no possible number grids, return a String[] containing exactly one element: "NO SOLUTIONS" (quotes for clarity only). If there are multiple distinct solutions, return a String[] containing exactly one element, formatted as "*n* SOLUTIONS" (quotes for clarity only), where *n* is the number of solutions modulo 1000000007 (10^9 + 7), with no extra leading zeroes.

Definition

Class:	NumberGrid
Method:	decrypt
Parameters:	String[], String[], String[]
Returns:	String[]
Method signature:	String[] decrypt(String[] hundreds, String[] tens, String[] ones)

(be sure your method is public)

Notes

Two number grids G_1 and G_2 are considered distinct if at least one entry in G_1 is different from the corresponding entry in G_2.

Constraints

-**hundreds** will contain between 1 and 50 elements, inclusive.-Each element of **hundreds** will contain between 1 and 50 characters, inclusive.-Each element of **hundreds** will have the same number of characters.-Each element of **hundreds** will contain only digits ('0' - '9').-**tens** and **ones** will have the same number of elements as **hundreds**.-Each element of **tens** and **ones** will have the same number of characters as each element of **hundreds**.-Each element of **tens** and **ones** will contain only digits ('0' - '9').

Examples

0)
```
{"000",
 "000"}

{"001",
 "111"}

{"691",
 "457"}

Returns: {"013", "257" }
```
The example given in the problem statement.

1)
```
{"0"}

{"0"}

{"3"}

Returns: {"3" }
```
A single cell means that the solution is equal to the input...

2)
```
{"0"}

{"4"}

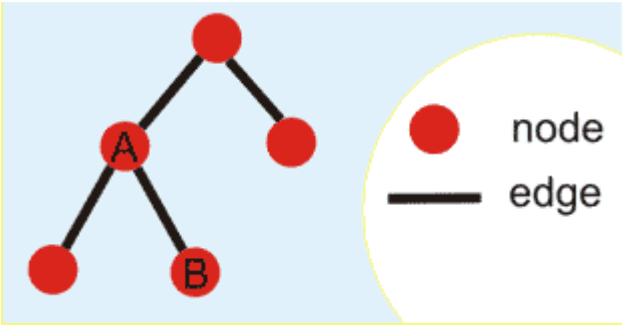
{"2"}

Returns: {"NO SOLUTIONS" }
```

...unless the input would not be a valid number grid.

Problem Statement (P8XGraphBuilder)

You want to build a graph consisting of N nodes and N-1 edges. The graph must be connected.



The degree of a node in the graph is equal to the number of edges adjacent to the node. For example, the degree of node A in the picture above is 3, while the degree of node B is 1. Note that in your graph the degree of each node will be between 1 and N-1, inclusive.

You are given a int[] **scores** with N-1 elements. The score of a node with degree d is **scores**[d-1]. The score of a graph is the sum of the scores of its nodes.

Your method should compute and return the maximum possible score for a graph you can construct.

Definition

Class:	P8XGraphBuilder
Method:	solve
Parameters:	int[]
Returns:	int
Method signature:	int solve(int[] scores)
(be sure your method is public)	

Notes

-In your solution, the number of nodes N in your graph can be determined as one plus the length of **scores**.

-In your graph, there must be at most one edge connecting any pair of nodes, and an edge cannot connect a node with itself.

Constraints

-**scores** will contain between 1 and 50 elements, inclusive.

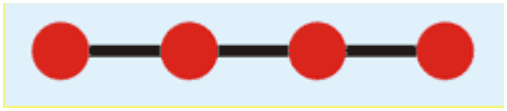
-Each element in **scores** will be between 0 and 10,000, inclusive.

Examples

0)
```
{1, 3, 0}
```

```
Returns: 8
```
As **scores** contains 3 elements, we are building a graph with N=4 nodes. Nodes of degree 1 have score 1, nodes of degree 2 have score 3, and nodes of degree 3 have score 0.

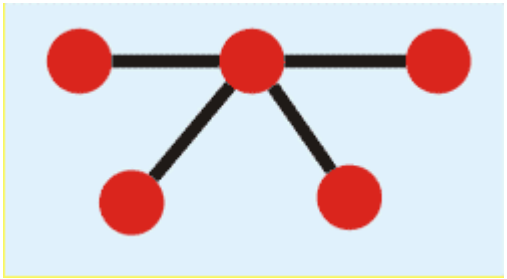
One possible graph with the highest score looks as follows:



In this graph the degrees of the nodes are 1, 2, 2, 1, respectively. The sum of their scores is 1+3+3+1 = 8.

1)
```
{0, 0, 0, 10}
```

```
Returns: 10
```
One possible solution for this test case is:



2)
```
{1, 2, 3, 4, 5, 6}
```

```
Returns: 12
```

3)
```
{5, 0, 0}
```

```
Returns: 15
```

4)
```
{1, 3, 2, 5, 3, 7, 5}
```

```
Returns: 20
```

Problem Statement (PerfectlyFairGame)

Dick and Jane are playing a simple game with darts and a board displaying a grid of squares, represented by the String[] **board**. Each square contains a number from 0 to 9. They each take turns (starting with Dick) tossing a dart at the board. They earn as many points as the number they hit, or 0 points if they miss the board. After each has thrown **darts** times, the player with the most points wins. If the points end up tied, a fair coin is flipped to determine the winner.

Dick and Jane are equally skilled players, but they are not perfect. If they are aiming at a particular grid square, they actually have a uniform chance of hitting any number in the 3x3 block surrounding it. (They will never aim at a grid square on the edge of the board, since missing is counterproductive.)

Both players play to maximize their odds of winning. Calculate the chance that Jane will win. Since the game is fair and the players are equally skilled, this shouldn't be too hard... right?

Definition

Class:	PerfectlyFairGame
Method:	winChance
Parameters:	String[], int
Returns:	double
Method signature:	double winChance(String[] board, int darts)
(be sure your method is public)	

Notes

The returned value must be accurate to within a relative or absolute value of 1E-9.

Constraints

- **board** will contain between 3 and 20 elements, inclusive.
- Each character in **board** will be a digit between '0' and '9', inclusive.
- Each element of **board** will contain between 3 and 20 digits, inclusive.
- Each element of **board** will contain the same number of digits.
- **darts** will be between 1 and 20, inclusive.

Examples

0)
```
{"123","456","789"}

10
```
Returns: 0.5
The odds are even, of course.

1)
```
{"55555","55555","55555","55555","55555"}

20
```
Returns: 0.5
Everyone scores 5 points each throw. Still even. Duh.

2)
```
{"0909","9090","0909","9090"}

20
```
Returns: 0.5
Yawn. Still even. Have you moved on to the Medium yet?

3)
```
{"888","808","888","000","000","999","999"}

1
```
Returns: 0.537037037037037
Still ev- ... wait a second. That can't be right, can it?

4)
```
{"29368","65609","67539","57982","71709"}

5
```
Returns: 0.5440401329247544
admins: Hi, could you please fix the 250's example output?

5)
```
{"4225271513","8352579454","1582795371","3365182453","7374843700","3262631490","526177101
```

```
20
```

```
Returns: 0.5055321764007881
```

Sigh.

Problem Statement (PerfectMemory)

You might have played the game called Memoria. In this game, there is a board consisting of **N** rows containing **M** cells each. Each of the cells has a symbol on its back. Each symbol occurs on exactly two cells on the board.

A move means turning a pair of cells *one by one* to see the symbols behind them. When the symbols differ, both of the cells are turned on their faces, thus hiding the symbols again. The player should remember the symbols. If the symbols on the backs of the turned cells coincide, the cells stay that way, i.e., don't turn back again. As soon as after some move all the cells on the board are turned (such that all the symbols are simultaneously visible), the game ends.

Manao has a perfect memory, so when he sees the symbol behind some cell, he remembers it forever. Manao is trying to finish the game in the least expected number of moves. Find the expected number of moves he will need to accomplish this.

Definition

Class:	PerfectMemory
Method:	getExpectation
Parameters:	int, int
Returns:	double
Method signature:	double getExpectation(int N, int M)

(be sure your method is public)

Notes

- The board Manao plays on is generated as follows. The same set of (**N** * **M**) / 2 symbols is used for each generation.
- The board contents are chosen uniformly among all valid **N** x **M** boards.
- The returned value must have an absolute or relative error less than 1e-9.

Constraints

- **N** will be between 1 and 50, inclusive.
- **M** will be between 1 and 50, inclusive.
- **N** * **M** will be even.

Examples

0)
```
1

2
```
```
Returns: 1.0
```
There are only two cells on the board, so the game always ends in one move.

1)
```
2

2
```
```
Returns: 2.6666666666666665
```
There are four cells. The game may flow in two possible scenarios:

1) In the first move, Manao turns two cells with equal symbols. The game ends in two moves then and the probability of such a first move is 1/3.

2) In the first move, Manao turns two cells with different symbols. Then he finishes the game in three moves and the probability of such a first move is 2/3.

The overall expected number of moves is 1/3 * 2 + 2/3 * 3 = 8/3.

2)
```
2

3
```
```
Returns: 4.333333333333334
```

3)
```
4

4
```
```
Returns: 12.392984792984793
```

Problem Statement (ProductQuery)

Mrs. Jeipouju took part in a programming contest. One of the problems was as follows:

*You are given an array A containing **N** integers. Each element of A is an integer between 0 and 9, inclusive. You are also given two arrays **Qfrom** and **Qto**, containing Q elements each. For each i, **Qfrom**[i] and **Qto**[i] describe the i-th query: you must take all elements of A with indices between **Qfrom**[i] and **Qto**[i], inclusive, and calculate their product modulo 10. Output exactly Q integers. The i-th number of the output must be the answer to the i-th query.*

Mrs. Jeipouju made an input for this task by herself and generated the output using her program. However, she erased the array A by mistake - now she only knows int **N**, int[] **Qfrom**, int[] **Qto** and int[] **output** (**output**[i] is the i-th integer generated by her program). Return the number of possible arrays A, modulo 1,000,000,007. If there are no possible arrays A (i.e. her program is definitely wrong), return 0.

Constraints

- **N** will be between 1 and 100, inclusive.
- **Qfrom**, **Qto** and **output** will contain the same number of elements.
- **Qfrom**, **Qto** and **output** will contain between 1 and 50 elements, inclusive.
- Each element of **Qfrom** and **Qto** will be between 0 and **N-1**, inclusive.
- For each i, **Qto**[i] will be greater than or equal to **Qfrom**[i].-Each element of **output** will be between 0 and 9, inclusive.

Definition

Class:	ProductQuery
Method:	theInput
Parameters:	int, int[], int[], int[]
Returns:	int
Method signature:	int theInput(int N, int[] Qfrom, int[] Qto, int[] output)

(be sure your method is public)

Examples

0)
```
3

{0}

{0}

{5}

Returns: 100
```
A[0] must be 5. A[1] and A[2] are arbitrary integers between 0 and 9.

1)
```
5

{0, 2, 0}

{1, 4, 4}

{3, 4, 6}

Returns: 0
```
Mrs. Jeipouju's program is definitely wrong.

2)
```
3

{0, 1}

{1, 2}

{5, 8}

Returns: 4
```
There are 4 possible inputs:

- A[0] = 5, A[1] = 1, A[2] = 8
- A[0] = 5, A[1] = 3, A[2] = 6
- A[0] = 5, A[1] = 7, A[2] = 4
- A[0] = 5, A[1] = 9, A[2] = 2

3)
```
8

{2, 0, 3, 6}

{3, 7, 4, 6}

{2, 0, 6, 7}

Returns: 118080
```

4)
```
20

{9, 6, 2, 3, 1, 3, 12, 3, 4, 6, 10, 18, 10, 11, 15,

{12, 12, 4, 13, 7, 14, 17, 10, 5, 13, 11, 19, 17, 1

{7, 5, 7, 5, 0, 5, 8, 5, 3, 5, 3, 9, 4, 1, 6, 5}

Returns: 276398080
```

5)
```
58

{5, 5}

{8, 8}

{1, 2}

Returns: 0
```

Problem Statement (SellingGold)

Jack has a store where he is selling gold. He has several boxes of gold and the i-th box contains **gold[i]** grams of gold and costs **price[i]** dollars.

Two customers entered the shop and want to buy all the gold. Jack knows that the total number of boxes is even, so he decided to give each customer half of the boxes. He knows that after buying gold, each customer will calculate the average price that he has paid for 1 gram of gold and announce it. Jack wants the sum of the two announced numbers to be as small as possible.

Given int[]'s **gold** and **price**, return the minimal possible sum of two described averages.

Definition

Class:	SellingGold
Method:	minimalSum
Parameters:	int[], int[]
Returns:	double
Method signature:	double minimalSum(int[] gold, int[] price)
(be sure your method is public)	

Notes

- Your return value must have an absolute or relative error less than 1e-9.

Constraints

- **gold** will contain between 2 and 50 elements, inclusive.
- **gold** will contain an even number of elements.
- **gold** and **price** will contain the same number of elements.
- Each element of **gold** will be between 1 and 500, inclusive.
- Each element of **price** will be between 1 and 500, inclusive.

Examples

0)
```
{2,4}

{3,4}

Returns: 2.5
```
The average price for one of the customers will be 3/2=1.5, and for the other one it will be 4/4=1. The answer is 1.5+1=2.5.

1)
```
{1,1,2,2}

{1,1,1,1}

Returns: 1.3333333333333333
```
There are two different ways to divide the boxes among the customers. In the first case one of the customers gets the first two boxes and the other customer gets two remaining boxes. In this case sum of the averages is 2/2+2/4=1.5. In the second case each customer is sold 3 grams for 2 dollars. Therefore we get 2/3+2/3=1.33(3) which is optimal.

2)
```
{2,1,1,1}

{300,300,300,300}

Returns: 500.0
```
Remember that each customer must get exactly half of the boxes.

3)
```
{500,500,500,500,500,500}

{1,2,4,4,2,2}

Returns: 0.01
```
It doesn't matter how the boxes are divided.

4)
```
{1,2,3,4,5,6,7,8}

{2,3,4,5,6,7,8,10}

Returns: 2.498452012383901
```

Problem Statement (STable)

You are given two Strings **s** and **t**. All characters of **s** and **t** are distinct. No character of **s** is present in **t** and no character of **t** is present in **s**.

Let N be the length of **s**, and M be the length of **t**. Define a 2-dimensional string array "table" as follows:

- table[i][0] = **s**[i-1] (1<=i<=N)
- table[0][j] = **t**[j-1] (1<=j<=M)
- table[i][j] = min(table[i-1][j], table[i][j-1]) + max(table[i-1][j], table[i][j-1]) (1<=i<=N, 1<=j<=M)

Note that min and max are defined by the lexicographical order of strings (see Notes for a more formal definition), and A+B means the concatenation of strings A and B.

Your task is to find a substring of table[N][M]. Let L be the length of table[N][M]. Return the substring of table[N][M] whose start position (0-indexed) is **pos** and length is min(50, L-**pos**).

Definition

Class:	STable
Method:	getString
Parameters:	String, String, long
Returns:	String
Method signature:	String getString(String s, String t, long pos)

(be sure your method is public)

Notes

- A string X is defined as smaller than a string Y if and only if X is a prefix of Y or X has a smaller character than Y at the first position where they differ.
- The order of characters is defined by their ASCII codes: '0'<...<'9'<'A'<...<'Z'<'a'<...<'z'.

Constraints

- **s** and **t** will each contain between 1 and 30 characters, inclusive.
- All characters of **s** and **t** will be distinct.-No character of **s** will be present in **t**.
- No character of **t** will be present in **s**.-**s** and **t** will contain only letters ('A'-'Z', 'a'-'z') and digits ('0'-'9').
- **pos** will be between 0 and L-1, inclusive, where L is the length of table[N][M] as defined in the statement.

Examples

0)

```
"ad"

"cb"

0
```

```
Returns: "acbacd"
```

In this case, the array "table" is as follows.

```
-----------------
| |  c   |  b   |
-----------------
|a|  ac  | acb  |
-----------------
|d| acd  |acbacd|
-----------------
```

1)

```
"fox"

"cat"

0
```

```
Returns: "acfcfoacftacfcfocfox"
```

2)

```
"Ra6b1t"

"W0lf"

66
```

```
Returns: "RWab0RWRWa0RWl0RWRWa6RWa0RWRWa6RWa6RWab0RWRWa6RWa6"
```

In this case, return 50 characters.

Problem Statement (TheCards)

John is playing a solitaire computer game. There are n cards numbered from 1 to n and **m** slots numbered form 1 to **m** from the left to the right. He has to place each card into some slot using the following rules.

Basic rules:

- The first card (card number 1) may be placed in any slot.
- The second card (card number 2) must be placed exactly **dist**[0] slots to the left or exactly **dist**[0] slots to the right of the first card.
- Similarly, each card number *i* (*i* > 1) must be placed exactly **dist**[*i*-2] slots to the left or exactly **dist**[*i*-2] slots to the right of the previous card (card number *i*-1).
- Finally, if the last card (card number n) is exactly **dist**[n-1] slots to the left or exactly **dist**[n-1] slots to the right of the first card (card number 1) then John wins the game.

Special cases:

- If an element of **dist** is equal to 0, then the corresponding card should be placed into the same slot as previous card.
- If an element of **dist** is equal to -1, then the distance between the corresponding card and the previous card can be arbitrary.

You are given int[] **dist** that contains exactly n elements and the int **m**. Return the number (modulo 1234567891) of ways to place the cards into the slots so that John wins.

Definition

Class:	TheCards
Method:	find
Parameters:	int[], int
Returns:	int
Method signature:	int find(int[] dist, int m)
(be sure your method is public)	

Notes

- More than one card can be placed into a single slot.
- Slots do not wrap. There are no slots to the left of slot number 1 and no slots to the right of slot number **m**.

Constraints

- **dist** will contain between 2 and 36 elements, inclusive.
- Each element of **dist** will be between -1 and 10,000, inclusive.
- **m** will be between 1 and 1,000,000, inclusive.

Examples

0)
```
{3, 1, 2, 1, 1}

5

Returns: 8
```
Several possible solutions are shown below (in order from card 1 to card 5):

- slot 1, slot 4, slot 5, slot 3, slot 2;
- slot 1, slot 4, slot 3, slot 1, slot 2;
- slot 2, slot 5, slot 4, slot 2, slot 1;
- ...

1)
```
{4, 7}

47

Returns: 0
```
There is no way to win for John in this case.

2)
```
{1, 1, 1, 1}

7

Returns: 32
```

3)
```
{5, -1, -1, 4, 0, 2, -1, 8, 1, 2}

11

Returns: 4224
```

Problem Statement (TurtleSpy)

The Very Heterogeneous State of Feudalia's army is designing a new spy robot. The robot is disguised as a turtle and takes four different commands:

- "right X", where X is a integer between 1 and 359, inclusive. Will add X degrees to the turtle's direction angle.
- "left X", where X is a integer between 1 and 359, inclusive. Will subtract X degrees from the turtle's direction angle.
- "forward X", where X is a integer. Will move X units in the direction given by the turtle's direction angle.
- "backward X", where X is a integer. Will move X units in the opposite direction of the turtle's direction angle (The angle minus 180 degrees).

The army created a program that made the robot infiltrate deeply into foreign land by making the robot terminate as far as possible from the original location. Unfortunately, Feudalia is world famous for the ineptitude of its public officials. The program got mixed up and there is no way to tell the original order of the commands. You are given a String[] **commands**. Each element of **commands** represents a command in the described format. Find the order for all of the commands in the input that will maximize the Euclidean distance between the point where the robot was started and the point it reached after all commands were executed. Return the maximum distance.

Definition

Class:	TurtleSpy
Method:	maxDistance
Parameters:	String[]
Returns:	double
Method signature:	double maxDistance(String[] commands)

(be sure your method is public)

Notes

-The Euclidean distance between the points (x1,y1) and (x2,y2) is sqrt((x2-x1)^2 + (y2-y1)^2).-Your return value must be within 1e-9 relative or absolute error of the correct value

Constraints

-**commands** will contain between 1 and 50 elements, inclusive.-Each element of **commands** will be a string in the form "COMM X", where COMM is one of "left", "right", "forward" or "backward" (quotes for clarity) and X is a integer between 1 and 1000, inclusive.-For each element of **commands**, if the command is "left" or "right" then X will not exceed 359.

Examples

0)

```
{"forward 100", "backward 100", "left 90" }
```

```
Returns: 141.4213562373095
```

That distance is possible if we make sure to run the *"left 90"* command inbetween the other two commands.

1)

```
{"left 45", "forward 100", "right 45", "forward 100"}
```

```
Returns: 200.0
```

We can, for example, run the commands in the following order: "forward 100", "left 45", "right 45" and "forward 100".

2)

```
{"left 10", "forward 40",  "right 30", "left 10", "backward 4", "forward 4" }
```

```
Returns: 40.58520737741619
```

Problem Statement (YetAnotherBingoProblem)

You might have heard of a game called Bingo. In this problem we will consider a variation of this game. Our version of Bingo is a game for two players. Each player in the game holds one to five cards. Each of the cards has 25 numbers printed on it. The numbers are integers from 1 to 75 and are pairwise distinct within each particular card.

The game consists of at most 75 turns. In each turn, a number from 1 to 75 is drawn. Numbers drawn in different turns are distinct. Both players search for the drawn number in their cards and cross out each of its occurrences. As soon as one of the players crosses out all numbers on at least one of his cards, the game ends and this player is declared the winner. However, if such a situation occurs for both players in the same turn, neither of them is declared the winner and the game ends in a draw instead.

Manao is playing Bingo against his friend. Manao sees the friend's cards and he has also hacked the number drawing machine and can make it draw the numbers in any order he needs. Surely, Manao wants to win, but he doesn't want his friend to suspect him in cheating. So he wants the numbers to be drawn in such an order that he wins and the game lasts the maximum possible number of turns.

You are given String[]s **cards1** and **cards2** representing Manao's and his friend's cards, respectively. Concatenate the elements of **cards1** to obtain a comma-separated list of space-separated lists of integers. Each of the space-separated lists represents a single card and contains the 25 numbers printed on it. **cards2** comprises the friend's cards in the same fashion. If Manao can win the game, return the maximum possible number of turns in which it can happen. Otherwise, return -1.

Definition

Class:	YetAnotherBingoProblem
Method:	longestWinningSequence
Parameters:	String[], String[]
Returns:	int
Method signature:	int longestWinningSequence(String[] cards1, String[] cards2)

(be sure your method is public)

Constraints

- **cards1|2** will contain between 1 and 50 elements, inclusive.
- Each element of **cards1|2** will contain between 1 and 50 characters, inclusive.
- The concatenation of elements in **cards1|2** will represent a comma-separated list of at most five space-separated lists of integers. Each of these lists will contain 25 distinct numbers without leading zeros. Each of the numbers will be between 1 and 75, inclusive.

Examples

0)

```
{"60 28 ","8 25 54"," 31 13 26 71 20 66 55 72 67 18 51 52 48 63 53 ","42 16 ","1 30 56,",
"27 43 70 48 54 24 73 37 41 51 58 55"," 71 12 60 49 64 47 35 4 ","16 69 14 40 50"}
{"5 48 26 64 12 47 39 29 63 50 15 51 33 5","8 49 73 13 61 69 16 53 41 60 ","59 55,",
"16 55 57 43 20 3 74 12 4","2 ","35 19 65 51 17 1 15 69 56"," 31 13 9 24 58"," ","38 75"}
```

Returns: 74

Both Manao and his friend hold two cards. One of the longest possible winning sequences is as follows: 13, 14, 15, ..., 75, 11, 10, 9, ..., 1.

1)

```
{"34 ","56"," 7 22 21 43 57 38 ","12 5 32"," 39 15 13"," 75 20 ","55 69 70 2","3 65 48 33 9 44"}
```

```
{"46 18 3 33 34 58 24 36 30 50 ","45 11 56 25 57 60 54 74 42 8 70 40 20 2 1,26 61 20",
" 43 52 ","2 22 33 53 44 60 27 57 12 7 15 29 73 50 ","2","8 56 34 75 67 37,47 61 36 75 45 22 17 49 33 59",
" 48 39 69 27 3 70 12 46 54 1","1 67 ","43 73 5","2 68"}
```

Returns: 73

Manao has a single card and his friend holds three. If all the numbers from 1 to 75 are drawn in order, omitting 26 and 46, Manao wins after the 73rd turn.

2)

```
{"70 29 47 73 17 49 13 64 42 71 63 28 9 54 ","74 20 69 16 14 66 12 25 4","5 41 68"}
{"4","7 16 68 74 71 42 49 66 45 25 13 9 12"," 17 ","20"," 5","4 70 28 64 69 73 63 14 41"," 29"}
```

Returns: -1

The set of numbers on Manao's card is the same as on his friend's card. No matter how the numbers are drawn, the game ends in a draw.

3)

```
{"2 21 15 48 50 ","75 67 33 60 52 1","4 72 64 25 8 23 39 12 11 74 65 63 56 44 13,",
"13 3","3 52 72 39 63 44 14 21 48 3","8 65 12 56"," 74 60 23 8 15 50 75",
" 67 25 11 2,67 38 14 48 56 52 72 50 23 21 15 13 ","64 39 2 11 25 8 33 44 12 60",
" 65 75 74,38 15 50 11 13 12 44 60 74 21 25"," 48 67 39 72 14 8 63 64 56 75 33 ","52 31 23"}
{"7","4 21 12 60 25 38 15 11 49 ","54 8 63 33 65 56 52 13 14 72 75 23 48 50 67 3",
"9,52 14 25 39 64 50 21 33 2 15 67 13 48 ","56 12 7","5 63 60 11 65 44 74 ",
"23 72 29,25 12 74 2 64 31 39 33 ","14"," 21 13 72 15 23 8 60 5","6 50 38 75 ","44 46 67 1","1 48"}
```

Returns: 73