Department of Computer Science and Engineering

Knowledge and Skill Assignment	:	**Complex Problem Solving (CPS)**
Number of Problem Statements	:	**150**
Course Code	:	**ACSD13**
Course Name	:	**Design and Analysis of Algorithms**
Semester	:	**B.Tech IV Semester**
Academic Year	:	**2024 - 25**
Notional time to be spent	:	**08 hours**
AAT Marks	:	**05**

Guidelines for Students

1. This assignment should be undertaken by the students individually (at least, 2 problem statements by each student. More is appreciated).
2. Assessment of this assignment will be done through the rubrics handed over to the students (link: https://www.iare.ac.in/sites/default/files/downloads/Complex_Problem_Solving_Evaluation.pdf)
3. Make a video minimum of 10 minutes.
4. Prepare the self-assessment form (https://iare.ac.in/sites/default/files/downloads/Complex_Problem_Solving_Self_Assessment_Form.pdf)
5. Upload both, self-assessment form and video in IARE - Samvidha portal for evaluation by faculty (https://samvidha.iare.ac.in/).

Note:

1. Late submission up to 48 hours reduces 50% of the marks.
2. Beyond 48 hours 0 marks.

Target Engineering Competencies:
Please tick (✓) relevant engineering competency present in the problem

EC Number	Attributes	Profiles	Present in the problem
EC1	Depth of knowledge required (CP)	Ensures that all aspects of an engineering activity are soundly based on fundamental principles - by diagnosing, and taking appropriate action with data, calculations, results, proposals, processes, practices, and documented information that may be ill-founded, illogical, erroneous, unreliable or unrealistic requirements applicable to the engineering discipline	✓
EC2	Depth of analysis required (CP)	Have no obvious solution and require abstract thinking, originality in analysis to formulate suitable models.	✓
EC3	Design and development of	Support sustainable development solutions by ensuring functional requirements, minimize environmental	✓

	solutions (CA)	impact and optimize resource utilization throughout the life cycle, while balancing performance and cost effectiveness.	
EC4	Range of conflicting requirements (CP)	Competently addresses complex engineering problems which involve uncertainty, ambiguity, imprecise information and wide-ranging or conflicting technical, engineering and other issues.	
EC5	Infrequently encountered issues (CP)	Conceptualises alternative engineering approaches and evaluates potential outcomes against appropriate criteria to justify an optimal solution choice.	
EC6	Protection of society (CA)	Identifies, quantifies, mitigates and manages technical, health, environmental, safety, economic and other contextual risks associated to seek achievable sustainable outcomes with engineering application in the designated engineering discipline.	
EC7	Range of resources (CA)	Involve the coordination of diverse resources (and for this purpose, resources include people, money, equipment, materials, information and technologies) in the timely delivery of outcomes	✓
EC8	Extent of stakeholder involvement (CP)	Design and develop solution to complex engineering problem considering a very perspective and taking account of stakeholder views with widely varying needs.	
EC9	Extent of applicable codes, legal and regulatory (CP)	Meet all level, legal, regulatory, relevant standards and codes of practice, protect public health and safety in the course of all engineering activities.	
EC10	Interdependence (CP)	High level problems including many component parts or sub-problems, partitions problems, processes or systems into manageable elements for the purposes of analysis, modelling or design and then re-combines to form a whole, with the integrity and performance of the overall system as the top consideration.	
EC11	Continuing professional development (CPD) and lifelong learning (CA)	Undertake CPD activities to maintain and extend competences and enhance the ability to adapt to emerging technologies and the ever-changing nature of work.	
EC12	Judgement (CA)	Recognize complexity and assess alternatives in light of competing requirements and incomplete knowledge. Require judgement in decision making in the course of all complex engineering activities.	

ALCP Algorithm Design Manual

Algorithms are fundamental to computer science and software engineering. The real-world performance of any software system depends on the algorithms chosen and the suitability of the various layers of implementation. Good algorithm design is therefore crucial for the performance of all software systems. Moreover, the study of algorithms provides insight into the intrinsic nature of the problem as well as possible solution techniques independent of programming language, programming paradigm, computer hardware, or any other implementation aspect. An important part of computing is the ability to select algorithms appropriate to particular purposes and to apply them, recognizing the possibility that no suitable algorithm may exist. Algorithms are essential in all advanced areas of computer science: artificial intelligence, databases, distributed computing, graphics, networking, operating systems, programming languages, security, and so on.

Note:

Algorithms serve as the backbone of numerous real-world applications across diverse industries. Here are some areas where algorithms play a crucial role:

Applications of Algorithms

1. Web Search Engines

Search engines like Google, Bing, and Yahoo utilize complex algorithms to index and rank web pages, providing users with relevant search results. These algorithms analyze factors such as content relevance, user intent, and website authority to deliver accurate search outcomes.

2. Data Analysis and Machine Learning

Algorithms are extensively used in data analysis and machine learning to extract valuable insights from large datasets. Techniques like clustering algorithms, decision trees, and neural networks enable organizations to uncover patterns, make predictions, and make data-driven decisions.

3. Optimization and Planning

Algorithms are integral to optimization problems, such as route planning for logistics companies, resource allocation in manufacturing, and financial portfolio optimization. These algorithms facilitate efficient decision-making by finding the best combination of resources and minimizing costs.

4. Computer Graphics and Gaming

Algorithms are employed in computer graphics rendering to create realistic visuals in video games and movies. From rendering 3D images to simulating intricate physics-based movements, algorithms enable captivating visual experiences in the gaming and entertainment industry.

5. Financial Modeling and Trading

Financial institutions rely on algorithms to model market behaviour, predict stock trends, and execute high-frequency trades. Algorithmic trading algorithms analyze large volumes of financial data and make rapid decisions to capitalize on market opportunities in milliseconds.

6. Networking and Routing

Network routing algorithms determine the most efficient paths for data packets to traverse through complex networks. These algorithms optimize data flow, minimize latency, and ensure reliable communication across interconnected systems, such as the internet and telecommunications networks.

7. Artificial Intelligence and Robotics

Algorithms power artificial intelligence (AI) systems, enabling machines to perceive, reason, and learn. From natural language processing to computer vision and autonomous vehicles, AI algorithms emulate human intelligence and drive technological advancements in various fields.

Table of Contents

ALCP 59	Common Permutation
ALCP 60	Crypt Kicker II
ALCP 61	Automated Judge Script
ALCP 62	File Fragmentation
ALCP 63	Doublets
ALCP 64	Fmt
ALCP 65	Light, More Light
ALCP 66	Carmichael Numbers
ALCP 67	Euclid Problem
ALCP 68	Factovisors
ALCP 69	Summation of Four Primes
ALCP 70	Smith Numbers
ALCP 71	Marbles
ALCP 72	Repackaging
ALCP 73	Ant on a Chessboard
ALCP 74	The Monocycle
ALCP 75	Star
ALCP 76	Bee Maja
ALCP 77	Robbery
ALCP 78	(2/3/4) – D Sqr/Rects/Cubes/Boxes?
ALCP 79	Dermuba Triangle
ALCP 80	Airlines
ALCP 81	Maximum Students Taking Exam
ALCP 82	Cracking the Safe
ALCP 83	Reconstruct Itinerary
ALCP 84	Shortest Path Visiting All Nodes
ALCP 85	The Stable Marriage Problem
ALCP 86	College Admission Problem
ALCP 87	Compatible intervals
ALCP 88	Bridge Crossing Revisited

ALCP 119	Max-cost Spanning Tree
ALCP 120	Dynamic Investment Strategy for Maximizing Returns
ALCP 121	Tower of Hanoi and the End of the Universe
ALCP 122	Winning a War
ALCP 123	Decrypting Permuted Text
ALCP 124	Maximum Skill Utilization
ALCP 125	Efficient Longest Monotonic Sequence
ALCP 126	Matrix chain product
ALCP 127	Optimal BST
ALCP 128	Maximizing Taxi Driver's Profit problem
ALCP 129	Optimizing File Replication Across Servers
ALCP 130	Finding the Longest Bitonic Subsequence
ALCP 131	Dart Game and Analysis
ALCP 132	Deferred data structure
ALCP 133	Latin Square
ALCP 134	Valid Sudoku Completion
ALCP 135	Completing a Partially Filled N-Queens Board
ALCP 136	Magic Square Transformation
ALCP 137	Smart Warehouse Item Retrieval
ALCP 138	Merging Two Upper Hulls
ALCP 139	Dynamic Maintenance of Convex Hull
ALCP 140	Finding All Intersecting Pairs
ALCP 141	Escape the Maze
ALCP 142	Weighted Word Chain
ALCP 143	Land Division Optimization
ALCP 144	Robo-Arena
ALCP 145	The Matrix Illumination
ALCP 146	Friendship Graph
ALCP 147	Circuit Board Placement
ALCP 148	Finding All Intersecting Pairs

ALCP 01 — The 3n + 1 Problem

Consider the following algorithm to generate a sequence of numbers. Start with an integer *n*. If *n* is even, divide by 2. If *n* is odd, multiply by 3 and add 1. Repeat this process with the new value of *n*, terminating when *n=1*. For example, the following sequence of numbers will be generated for *n* = 22:

22 11 34 17 52 26 13 40 20 10 5 16 84 21

It is conjectured (but not yet proven) that this algorithm will terminate at *n=1* for every integer *n*. Still, the conjecture holds for all integers up to at least 1,000,000. For an input *n*, the cycle-length of *n* is the number of numbers generated up to and including the1. In the example above, the cycle length of 22 is 16. Given any two numbers *i* and *j*, you are to determine the maximum cycle length overall numbers between *i* and *j,* including both end points.

Input

The input will consist of a series of pairs of integers *i* and *j*, one pair of integers per line. All integers will be less than 1,000,000 and greater than 0.

Output

For each pair of input integers *i* and *j*, output *i*, *j* in the same order in which they appeared in the input and then the maximum cycle length for integers between and including *i* and *j*. These three numbers should be separated by one space, with all three numbers on one line and with one line of output for each line of input.

Sample Input	**Sample Output**
1 10	1 10 20
100 200	100 200 125
201 210	201 210 89
900 1000	900 1000 174

ALCP 02 Minesweeper

Have you ever played Minesweeper? This cute little game comes with a certain operating system whose name we can’t remember. The goal of the game is to find where all the mines are located within a *M x N* field.

The game shows a number in a square which tells you how many mines there are adjacent to that square. Each square has at most eight adjacent squares. The 4x4 field on the left contains two mines, each represented by *a* “*” character. If we represent the same field by the hint numbers described above, we end up with the field on the right:

```
*...        *100
....        2210
.*..        1*10
....        1110
```

Input

The input will consist of an arbitrary number of fields. The first line of each field contains two integers *n* and *m* (0 < *n, m* <=100) which stand for the number of lines and columns of the field, respectively. Each of the next n lines contains exactly m characters, representing the field.

Safe squares are denoted by “.” and mine squares by “*,” both without the quotes. The first field line where *n* = *m* = 0 represents the end of input and should not be processed.

Output

For each field, print the message Field #x: on a line alone, where *x* stands for the number of the field starting from 1. The next n lines should contain the field with the “.” characters replaced by the number of mines adjacent to that square. There must be an empty line between field outputs.

Sample Input	**Sample Output**
4 4	Field #1:
*...	*100
....	2210
.*..	1*10
....	1110
3 5	Field #2:
**...	**100
.....	33200
.*...	1*100
0 0	

ALCP 03 The Trip

A group of students are members of a club that travels annually to different locations. Their destinations in the past have included Indianapolis, Phoenix, Nashville, Philadelphia, San Jose, and Atlanta. This spring they are planning a trip to Eindhoven. The group agrees in advance to share expenses equally, but it is not practical to share every expense as it occurs. Thus, individuals in the group pay for particular things, such as meals, hotels, taxi rides, and plane tickets. After the trip, each student's expenses are tallied and money is exchanged so that the net cost to each is the same, to within one cent. In the past, this money exchange has been tedious and time consuming. Your job is to compute, from a list of expenses, the minimum amount of money that must change hands in order to equalize (within one cent) all the students' costs.

Input

Standard input will contain the information for several trips. Each trip consists of a line containing a positive integer *n* denoting the number of students on the trip. This is followed by *n* lines of input, each containing the amount spent by a student in dollars and cents. There are no more than 1000 students and no student spent more than $10,000.00. A single line containing 0 follows the information for the last trip.

Output

For each trip, output a line stating the total amount of money, in dollars and cents that must be exchanged to equalize the students' costs.

Sample Input

```
3
10.00
20.00
30.00
4
15.00
15.01
3.00
3.01
0
```

Sample Output

```
$10.00
$11.99
```

ALCP 04 — LCD Display

A friend of yours has just bought a new computer. Before this, the most powerful machine he ever used was a pocket calculator. He is a little disappointed because he liked the LCD display of his calculator more than the screen on his new computer! To make him happy, write a program that prints numbers in LCD display style.

Input

The input file contains several lines, one for each number to be displayed. Each line contains integers *s* and *n,* where *n* is the number to be displayed (0 <= *n* <= 99, 999, 999) and s is the size in which it shall be displayed (1 <= *s* <= 10). The input will be terminated by a line containing two zeros, which should not be processed.

Output

Print the numbers specified in the input file in an LCD display-style using *s* "-" signs for the horizontal segments and *s* "|" signs for the vertical ones. Each digit occupies exactly *s* + 2 columns and 2*s* + 3 rows. Be sure to fill all the white space occupied by the digits with blanks, including the last digit. There must be exactly one column of blanks between two digits.

Output a blank line after each number. You will find an example of each digit in the sample output below.

Sample Input

```
2  12345
3  67890
0  0
```

Sample Output

```
      --   --        --   
   |    |    | |  | |      
   |    |    | |  | |      
      --   --   --   --   
   | |       |    |    |  
   | |       |    |    |  
      --   --        --   

 ---   ---   ---   ---   --- 
|         | |   | |   | |   |
|         | |   | |   | |   |
|         | |   | |   | |   |
 ---         ---   ---       
|   |     | |   |     | |   |
|   |     | |   |     | |   |
|   |     | |   |     | |   |
 ---         ---   ---   --- 
```

ALCP 05 Graphical Editor

Graphical editors such as Photoshop allow us to alter bit-mapped images in the same way that text editors allow us to modify documents. Images are represented as an *M x N* array of pixels, where each pixel has a given color. Your task is to write a program which simulates a simple interactive graphical editor.

Input

The input consists of a sequence of editor commands, one per line. Each command is represented by one capital letter placed as the first character of the line. If the command needs parameters, they will be given on the same line separated by spaces. Pixel coordinates are represented by two integers, a column number between 1 ...*M* and a row number between 1 ...*N*, where <= *M*, *N* <= 250. The origin sits in the upper-left corner of the table. Colors are specified by capital letters.

The editor accepts the following commands:

I M N	Create a new *M* × *N* image with all pixels initially colored white (O).
C	Clear the table by setting all pixels white (O). The size remains unchanged.
L X Y C	Colors the pixel (*X, Y*) in color (C).
V X Y1 Y2 C	Draw a vertical segment of color (C) in column *X*, between the rows *Y* 1 and *Y* 2 inclusive.
H X1 X2 Y C	Draw a horizontal segment of color (C) in the row *Y*, between the columns *X*1 and *X*2 inclusive.
K X1 Y1 X2 Y2 C	Draw a filled rectangle of color C, where (*X*1, *Y* 1) is the upper-left and (*X*2, *Y*2) the lower right corner.
F X Y C	Fill the region *R* with the color C, where *R* is defined as follows. Pixel (*X, Y*) belongs to *R*. Any other pixel which is the same color as pixel (*X, Y*) and shares a common side with any pixel in *R* also belongs to this region.
S Name	Write the file name in MSDOS 8.3 format followed by the contents of the current image.
X	Terminate the session.

Output

On every command S NAME, print the filename NAME and contents of the current image. Each row is represented by the color contents of each pixel. See the sample output.

Ignore the entire line of any command defined by a character other than I, C, L, V, H, K, F, S, or X, and pass on to the next command. In case of other errors, the program behaviour is unpredictable.

Sample Input

```
I 5 6
L 2 3 A
S one.bmp
G 2 3 J
```

Sample Output

```
one.bmp
OOOOO
OOOOO
OAOOO
```

F 3 3 J	OOOOO
V 2 3 4 W	OOOOO
H 3 4 2 Z	OOOOO
S two.bmp	two.bmp
X	JJJJJ
	JJZZJ
	JWJJJ
	JWJJJ
	JJJJJ
	JJJJJ

ALCP 06 | Interpreter

A certain computer has ten registers and 1,000 words of RAM. Each register or RAM location holds a three-digit integer between 0 and 999. Instructions are encoded as three-digit integers and stored in RAM. The encodings are as follows:

100	means halt
2dn	means set register *d* to *n* (between 0 and 9)
3dn	means add *n* to register *d*
4dn	means multiply register *d* by *n*
5ds	means set register *d* to the value of register *s*
6ds	means add the value of register *s* to register *d*
7ds	means multiply register *d* by the value of register *s*
8da	means set register *d* to the value in RAM whose address is in register *a*
9sa	means set the value in RAM whose address is in register *a* to that of register *s*
0ds	Means goto the location in register *d* unless *s* contains 0

All registers initially contain 000. The initial content of the RAM is read from standard input. The first instruction to be executed is at RAM address 0. All results are reduced modulo 1,000.

Input

The input begins with a single positive integer on a line by itself indicating the number of cases, each described as below. This is followed by a blank line, and there will be a blank line between each two consecutive inputs. Each input case consists of up to 1,000 three-digit unsigned integers, representing the contents of consecutive RAM locations starting at 0. Unspecified RAM locations are initialized to 000.

Output

The output of each test case is a single integer: the number of instructions executed up to and including the halt instruction. You may assume that the program does halt. Separate the output of two consecutive cases by a blank line.

Sample Input

```
1
299
492
495
399
492
495
399
283
279
689
078
100
000
```

Sample Output

```
16
```

000

000

ALCP 07 | Check the Check

Your task is to write a program that reads a chessboard configuration and identifies whether a king is under attack (in check). A king is in check if it is on square which can be taken by the opponent on his next move.

White pieces will be represented by uppercase letters, and black pieces by lowercase letters. The white side will always be on the bottom of the board, with the black side always on the top.

For those unfamiliar with chess, here are the movements of each piece:

Pawn (p or P): can only move straight ahead, one square at a time. However, it takes pieces diagonally, and that is what concerns you in this problem.

Knight (n or N): has an L-shaped movement shown below. It is the only piece that can jump over other pieces.

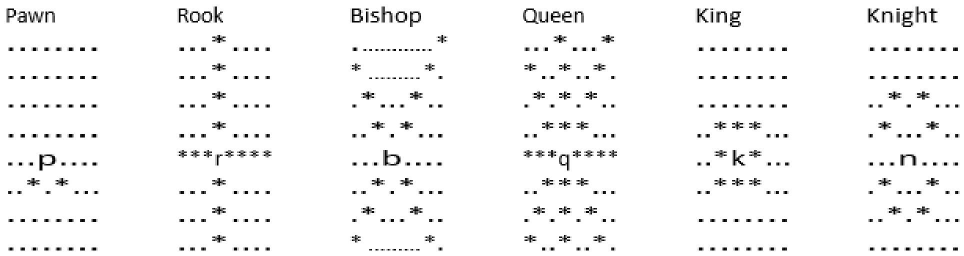
Bishop (b or B): can move any number of squares diagonally, either forward or backward.

Rook (r or R): can move any number of squares vertically or horizontally, either forward or backward.

Queen (q or Q): can move any number of squares in any direction (diagonally, horizontally, or vertically) either forward or backward.

King (k or K): can move one square at a time in any direction (diagonally, horizontally, or vertically) either forward or backward.

Movement examples are shown below, where "*" indicates the positions where the piece can capture another piece:



```
Pawn        Rook        Bishop      Queen       King        Knight
........    ...*....    .......*    ...*...*    ........    ........
........    ...*....    *.....*.    *..*..*.    ........    ........
........    ...*....    .*...*..    .*.*.*..    ........    ..*.*...
........    ...*....    ..*.*...    ..***...    ..***...    .*...*..
...p....    ***r****    ...b....    ***q****    ..*k*...    ...n....
..*.*...    ...*....    ..*.*...    ..***...    ..***...    .*...*..
........    ...*....    .*...*..    .*.*.*..    ........    ..*.*...
........    ...*....    *.....*.    *..*..*.    ........    ........
```

Remember that the knight is the only piece that can jump over other pieces. The pawn movement will depend on its side. If it is a black pawn, it can only move one square diagonally down the board. If it is a white pawn, it can only move one square diagonally up the board. The example above is a black pawn, described by a lowercase "p". We use "move" to indicate the squares where the pawn can capture another piece.

Input

There will be an arbitrary number of board configurations in the input, each consisting of eight lines of eight characters each. A "." denotes an empty square, while upper- and lowercase letters represent the pieces as defined above. There will be no invalid characters and no configurations where both kings are in check. You must read until you find an empty board consisting only of "." characters, which should not be processed. There will be an empty line between each pair of board configurations. All boards, except for the empty one, will contain exactly one white king and one black king.

Output

For each board configuration read you must output one of the following answers:

Game #d: white king is in check.

Game #d: black king is in check.

Game #d: no king is in check.

Where d stands for the game number starting from 1.

Sample Input

```
..k.....
ppp.pppp
........
.R...B..
........
........
PPPPPPPP
K.......

rnbqk.nr
ppp..ppp
....p...
...p....
.bPP....
.....N..
PP..PPPP
RNBQKB.R

........
........
........
........
........
........
........
........
```

Sample Output

```
Game #1: black king is in check.
Game #2: white king is in check.
```

ALCP 08 — Australian Voting

Australian ballots require that voters rank all the candidates in order of choice. Initially only the first choices are counted, and if one candidate receives more than 50% of the vote then that candidate is elected. However, if no candidate receives more than 50%, all candidates tied for the lowest number of votes are eliminated. Ballots ranking these candidates first are recounted in favour of their highest-ranked non-eliminated candidate. This process of eliminating the weakest candidates and counting their ballots in favour of the preferred non-eliminated candidate continues until one candidate receives more than 50% of the vote, or until all remaining candidates are tied.

Input

The input begins with a single positive integer on a line by itself indicating the number of cases following, each as described below. This line is followed by a blank line. There is also a blank line between two consecutive inputs.

The first line of each case is an integer *n* <=20 indicating the number of candidates. The next n lines consist of the names of the candidates in order, each up to 80 characters in length and containing any printable characters. Up to 1,000 lines follow, each containing the contents of a ballot. Each ballot contains the numbers from 1 to n in some order. The first number indicates the candidate of first choice; the second number indicates candidate of second choice, and so on.

Output

The output of each test case consists of either a single line containing the name of the winner or several lines containing the names of all candidates who are tied. The output of each two consecutive cases are separated by a blank line.

Sample Input

```
1
3
John Doe
Jane Smith
Jane Austen
1 2 3
2 1 3
2 3 1
1 2 3
3 1 2
```

Sample Output

```
John Doe
```

ALCP 09 — Jolly Jumpers

A sequence of *n* > 0 integers is called a jolly jumper if the absolute values of the differences between successive elements take on all possible values 1 through *n*-1. For instance,

1 4 2 3

is a jolly jumper, because the absolute differences are 3, 2, and 1, respectively. The definition implies that any sequence of a single integer is a jolly jumper. Write a program to determine whether each of a number of sequences is a jolly jumper.

Input

Each line of input contains an integer n < 3, 000 followed by n integers representing the sequence.

Output

For each line of input generate a line of output saying "Jolly" or "Not jolly".

Sample Input	Sample Output
4 1 4 2 3	Jolly
5 1 4 2 -1 6	Not jolly

ALCP 10 Poker Hands

A poker deck contains 52 cards. Each card has a suit of either clubs, diamonds, hearts, or spades (denoted *C, D, H, S* in the input data). Each card also has a value of either 2 through 10, jack, queen, king, or ace (denoted 2, 3, 4, 5, 6, 7, 8, 9, *T, J, Q, K, A*). For scoring purposes card values are ordered as above, with 2 having the lowest and ace the highest value. The suit has no impact on value. A poker hand consists of five cards dealt from the deck. Poker hands are ranked by the following partial order from lowest to highest.

High Card. Hands which do not fit any higher category are ranked by the value of their highest card. If the highest cards have the same value, the hands are ranked by the next highest, and so on.

Pair. Two of the five cards in the hand have the same value. Hands which both contain a pair are ranked by the value of the cards forming the pair. If these values are the same, the hands are ranked by the values of the cards not forming the pair, in decreasing order.

Two Pairs. The hand contains two different pairs. Hands which both contain two pairs are ranked by the value of their highest pair. Hands with the same highest pair are ranked by the value of their other pair. If these values are the same the hands are ranked by the value of the remaining card.

Three of a Kind. Three of the cards in the hand have the same value. Hands which both contain three of a kind are ranked by the value of the three cards.

Straight. Hand contains five cards with consecutive values. Hands which both contain a straight are ranked by their highest card.

Flush. Hand contains five cards of the same suit. Hands which are both flushes are ranked using the rules for High Card.

Full House. Three cards of the same value, with the remaining two cards forming a pair. Ranked by the value of the three cards.

Four of a Kind. Four cards with the same value. Ranked by the value of the four cards.

Straight Flush. Five cards of the same suit with consecutive values. Ranked by the highest card in the hand.

Your job is to compare several pairs of poker hands and to indicate which, if either, has a higher rank.

Input

The input file contains several lines, each containing the designation of ten cards: the first five cards are the hand for the player named "Black" and the next five cards are the hand for the player named "White".

Output

Black wins.

White wins.

Tie.

Sample Input	Sample Output
2H 3D 5S 9C KD 2C 3H 4S 8C AH	White wins.
2H 4S 4C 2D 4H 2S 8S AS QS 3S	Black wins.
2H 3D 5S 9C KD 2C 3H 4S 8C KH	Black wins.
2H 3D 5S 9C KD 2D 3H 5C 9S KH	Tie.

ALCP 11 Hartals

Political parties in Bangladesh show their muscle by calling for regular hartals (strikes), which cause considerable economic damage. For our purposes, each party may be characterized by a positive integer *h* called the hartal parameter that denotes the average number of days between two successive strikes called by the given party.

Consider three political parties. Assume h_1 = 3, h_2 = 4, and h_3 = 8, where h_i is the hartal parameter for party i. We can simulate the behaviour of these three parties for *N* = 14 days. We always start the simulation on a Sunday. There are no hartals on either Fridays or Saturdays.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Days	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
Party 1			x			x			x			x		
Party 2				x				x				x		
Party 3								x						
Hartals			1	2				3	4			5		

There will be exactly five hartals (on days 3, 4, 8, 9, and 12) over the 14 days. There is no hartal on day 6 since it falls on Friday. Hence, we lose five working days in two weeks.

Given the hartal parameters for several political parties and the value of *N*, determine the number of working days lost in those *N* days.

Input

The first line of the input consists of a single integer *T* giving the number of test cases to follow. The first line of each test case contains an integer *N* (7 <=*N* <=3, 650), giving the number of days over which the simulation must be run. The next line contains another integer *P* (1<= *P* <=100) representing the number of political parties. The i^{th} of the next *P* lines contains a positive integer hi (which will never be a multiple of 7) giving the hartal parameter for party *i* (1 ≤ *i* ≤ *P*).

Output

For each test case, output the number of working days lost on a separate line.

Sample Input

```
2
14
3
3
4
8
100
4
12
15
25
40
```

Sample Output

```
5
15
```

ALCP 12 Crypt Kicker

A common but insecure method of encrypting text is to permute the letters of the alphabet. In other words, each letter of the alphabet is consistently replaced in the text by some other letter. To ensure that the encryption is reversible, no two letters are replaced by the same letter. Your task is to decrypt several encoded lines of text, assuming that each line uses a different set of replacements, and that all words in the decrypted text are from a dictionary of known words.

Input

The input consists of a line containing an integer *n*, followed by n lowercase words, one per line, in alphabetical order. These *n* words compose the dictionary of words which may appear in the decrypted text. Following the dictionary are several lines of input. Each line is encrypted as described above.

There are no more than 1,000 words in the dictionary. No word exceeds 16 letters. The encrypted lines contain only lower-case letters and spaces and do not exceed 80 characters in length.

Output

Decrypt each line and print it to standard output. If there are multiple solutions, any one will do. If there is no solution, replace every letter of the alphabet by an asterisk.

Sample Input

```
6
and
dick
jane
puff
spot
yertle
bjvg xsb hxsn xsb qymm xsb rqat xsb pnetfn
xxxx yyy zzzz www yyyy aaa bbbb ccc dddddd
```

Sample Output

```
dick and jane and puff and spot and yertle
**** *** **** *** **** *** **** *** ******
```

ALCP 13 Stack ’em Up

The Big City has many casinos. In one of them, the dealer cheats. She has perfected several shuffles; each shuffle rearranges the cards in exactly the same way whenever it is used. A simple example is the “bottom card” shuffle, which removes the bottom card and places it at the top. By using various combinations of these known shuffles, the crooked dealer can arrange to stack the cards in just about any particular order. You have been retained by the security manager to track this dealer. You are given a list of all the shuffles performed by the dealer, along with visual cues that allow you to determine which shuffle she uses at any particular time. Your job is to predict the order of the cards after a sequence of shuffles. A standard playing card deck contains 52 cards, with 13 values in each of four suits. The values are named 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace. The suits are named Clubs, Diamonds, Hearts, Spades. A particular card in the deck can be uniquely identified by its value and suit, typically denoted < value > of < suit >. For example, “9 of Hearts” or “King of Spades.” Traditionally a new deck is ordered first alphabetically by suit, then by value in the order given above.

Input

The input begins with a single positive integer on a line by itself indicating the number of test cases, followed by a blank line. There is also a blank line between two consecutive inputs.

Each case consists of an integer n 100, the number of shuffles that the dealer knows. Then follow *n* sets of 52 integers, each comprising all the integers from 1 to 52 in some order. Within each set of 52 integers, *i* in position *j* means that the shuffle moves the i^{th} card in the deck to position *j*. Several lines follow, each containing an integer *k* between 1 and *n*. These indicate that you have observed the dealer applying the kth shuffle given in the input.

Output

For each test case, assume the dealer starts with a new deck ordered as described above. After all the shuffles had been performed, give the names of the cards in the deck, in the new order. The output of two consecutive cases will be separated by a blank line.

Sample Input

```
1

2

2 1 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26

27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 52 51

52 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26

27 28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50 51 1

1

2
```

Sample Output

```
King of Spades

2 of Clubs

4 of Clubs

5 of Clubs

6 of Clubs

7 of Clubs

8 of Clubs

9 of Clubs

10 of Clubs

Jack of Clubs

Queen of Clubs

King of Clubs

Ace of Clubs

2 of Diamonds

3 of Diamonds

4 of Diamonds
```

```
5 of Diamonds
6 of Diamonds
7 of Diamonds
8 of Diamonds
9 of Diamonds
10 of Diamonds
Jack of Diamonds
Queen of Diamonds
King of Diamonds
Ace of Diamonds
2 of Hearts
3 of Hearts
4 of Hearts
5 of Hearts
6 of Hearts
7 of Hearts
8 of Hearts
9 of Hearts
10 of Hearts
Jack of Hearts
Queen of Hearts
King of Hearts
Ace of Hearts
2 of Spades
3 of Spades
4 of Spades
5 of Spades
6 of Spades
7 of Spades
8 of Spades
9 of Spades
10 of Spades
Jack of Spades
Queen of Spades
Ace of Spades
3 of clubs
```

ALCP 14 | Erdo¨s Numbers

The Hungarian Paul Erdo¨s (1913–1996) was one of the most famous mathematicians of the 20th century. Every mathematician having the honor of being a co-author of Erd¨os is well respected. Unfortunately, not everybody got the chance to write a paper with Erdo¨s, so the best they could do was publish a paper with somebody who had published a scientific paper with Erdo¨s. This gave rise to the so-called Erdo¨s numbers. An author who has jointly published with Erdo¨s had Erd¨os number 1. An author who had not published with Erd¨os but with somebody with Erdo¨s number 1 obtained Erdo¨s number 2, and so on. Your task is to write a program which computes Erd¨os numbers for a given set of papers and scientists.

Input

The first line of the input contains the number of scenarios. Each scenario consists of a paper database and a list of names. It begins with the line *P N*, where P and *N* are natural numbers. Following this line is the paper database, with P lines each containing the description of one paper specified in the following way:

Smith, M.N., Martin, G., Erdos, P.: Newtonian forms of prime factors

Note that umlauts, like "o¨," are simply written as "o". After the *P* papers follow *N* lines with names. Such a name line has the following format:

Martin, *G*.

Output

For every scenario you are to print a line containing a string "Scenario i" (where i is the number of the scenario) and the author names together with their Erdo¨s number of all authors in the list of names. The authors should appear in the same order as they appear in the list of names. The Erdo¨s number is based on the papers in the paper database of this scenario. Authors which do not have any relation to Erd¨os via the papers in the database have Erd¨os number "infinity."

Sample Input

```
1
4 3
Smith, M.N., Martin, G., Erdos, P.: Newtonian forms of prime factors
Erdos, P., Reisig, W.: Stuttering in petri nets
Smith, M.N., Chen, X.: First order derivates in structured programming
Jablonski, T., Hsueh, Z.: Selfstabilizing data structures


Smith, M.N.
Hsueh, Z.
Chen, X.
```

Sample Output

```
Scenario 1
Smith, M.N. 1
Hsueh, Z. infinity
Chen, X. 2
```

ALCP 15 | Contest Scoreboard

Want to compete in the ACM ICPC? Then you had better know how to keep score! Contestants are ranked first by the number of problems solved (the more the better), then by decreasing amounts of penalty time. If two or more contestants are tied in both problems solved and penalty time, they are displayed in order of increasing team numbers.

A problem is considered solved by a contestant if any of the submissions for that problem was judged correct. Penalty time is computed as the number of minutes it took until the first correct submission for a problem was received, plus 20 minutes for each incorrect submission prior to the correct solution. Unsolved problems incur no time penalties.

Input

The input begins with a single positive integer on a line by itself indicating the number of cases, each described as below. This line is followed by a blank line. There is also a blank line between two consecutive inputs.

The input consists of a snapshot of the judging queue, containing entries from some or all of contestants 1 through 100 solving problems 1 through 9. Each line of input consists of three numbers and a letter in the format contestant problem time *L*, where *L* can be *C, I, R, U*, or *E*. These stand for Correct, Incorrect, clarification Request, Unjudged, and Erroneous submission. The last three cases do not affect scoring.

The lines of input appear in the order in which the submissions were received.

Output

The output for each test case will consist of a scoreboard, sorted by the criteria described above. Each line of output will contain a contestant number, the number of problems solved by the contestant and the total time penalty accumulated by the contestant. Since not all contestants are actually participating, only display those contestants who have made a submission.

The output of two consecutive cases will be separated by a blank line.

Sample Input

```
1
1    2    10    I
3    1    11    C
1    2    19    R
1    2    21    C
1    1    25    C
```

Sample Output

```
1  2  66
3  1  11
```

ALCP 16 — Yahtzee

The game of Yahtzee involves five dice, which are thrown in 13 rounds. A score card contains 13 categories. Each round may be scored in a category of the player's choosing, but each category may be scored only once in the game. The 13 categories are scored as follows:

- ones - sum of all ones thrown
- twos - sum of all twos thrown
- threes - sum of all threes thrown
- fours - sum of all fours thrown
- fives - sum of all fives thrown
- sixes - sum of all sixes thrown
- chance - sum of all dice
- three of a kind - sum of all dice, provided at least three have same value
- four of a kind - sum of all dice, provided at least four have same value
- five of a kind - 50 points, provided all five dice have same value
- short straight - 25 points, provided four of the dice form a sequence (that is, 1,2,3,4 or 2,3,4,5 or 3,4,5,6)
- long straight - 35 points, provided all dice form a sequence (1,2,3,4,5 or 2,3,4,5,6)
- full house - 40 points, provided three of the dice are equal and the other two dice are also equal. (for example, 2,2,5,5,5)

Each of the last six categories may be scored as 0 if the criteria are not met.

The score for the game is the sum of all 13 categories plus a bonus of 35 points if the sum of the first six categories is 63 or greater.

Your job is to compute the best possible score for a sequence of rounds.

Input

Each line of input contains five integers between 1 and 6, indicating the values of the five dice thrown in each round. There are 13 such lines for each game, and there may be any number of games in the input data.

Output

Your output should consist of a single line for each game containing 15 numbers: the score in each category (in the order given), the bonus score (0 or 35), and the total score. If there is more than categorization that yields the same total score, any one will do.

Sample Input

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

Sample Output

```
1  2  3  4  5  0  15  0  0  0  25  35  0  0  90
3  6  9  12  15  30  21  20  26  50  25  35  40  35  327
```

```
1    2    3    4    5
1    2    3    4    5
1    1    1    1    1
6    6    6    6    6
6    6    6    1    1
1    1    1    2    2
1    1    1    2    3
1    2    3    4    5
1    2    3    4    6
6    1    2    6    6
1    4    5    5    5
5    5    5    5    6
4    4    4    5    6
3    1    3    6    3
2    2    2    4    6
```

ALCP 17 — Vito's Family

The famous gangster Vito Dead stone is moving to New York. He has a very big family there, all of them living on Lamafia Avenue. Since he will visit all his relatives very often, he wants to find a house close to them. Indeed, Vito wants to minimize the total distance to all of his relatives and has black mailed you to write a program that solves his problem.

Input

The input consists of several test cases. The first line contains the number of test cases. For each test case you will be given the integer number of relatives *r* ($0<r<500$) and the street numbers (also integers) $s_1, s_2, \ldots, s_i, \ldots, s_r$ where they live ($0<s_i<30{,}000$).

Note that several relatives might live at the same street number.

Output

For each test case, your program must write the minimal sum of distances from the optimal Vito's house to each one of his relatives. The distance between two street numbers s_i and s_j *is* $d_{ij}=|si-s_j|$.

Sample Input

```
2
2 2 4
3 2 4 6
```

Sample Output

```
2
4
```

ALCP 18 — Stacks of Flapjacks

Cooking the perfect stack of pancakes on a grill is a tricky business, because no matter how hard you try all pancakes in any stack have different diameters. For neatness's sake, however, you can sort the stack by size such that each pancake is smaller than all the pancakes below it. The size of a pancake is given by its diameter. Sorting a stack is done by a sequence of pancake "flips." A flip consists of inserting a spatula between two pancakes in a stack and flipping (reversing) all the pancakes on the spatula (reversing the sub-stack). A flip is specified by giving the position of the pancake on the bottom of the sub-stack to be flipped relative to the entire stack. The bottom pancake has position1, while the top pancake on a stack of n pancakes has position *n*.

A stack is specified by giving the diameter of each pancake in the stack in the order in which the pancakes appear. For example, consider the three stacks of pancakes below in which pancake 8 is the top-most pancake of the left stack:

8	7	2
4	6	5
6	4	8
7	8	4
5	5	6
2	2	7

The stack on the left can be transformed to the stack in the middle via flip(3).The middle stack can be transformed into the right stack via the command flip(1).

Input

The input begins with a single positive integer on a line by itself indicating the number of test cases, followed by a blank line. There is also a blank line between each two consecutive inputs. The first line of each case contains n, followed by n lines giving the crossing times for each of the people. There are not more than 1,000 people and nobody takes more than100 seconds to cross the bridge.

Output

For each test case, the first line of output must report the total number of seconds required for all n people to cross the bridge. Subsequent lines give a strategy for achieving this time. Each line contains either one or two integers, indicating which person or people form the next group to cross. Each person is indicated by the crossing time specified in the input. Although many people may have the same crossing time, this ambiguity is of no consequence.

Note that the crossings alternate directions, as it is necessary to return the flash light so that more may cross. If more than one strategy yields the minimal time, anyone will do. The output of two consecutive cases must be separated by a blank line.

Sample Input	Sample Output
1 2 3 4 5	1 2 3 4 5
5 4 3 2 1	0
5 1 2 3 4	5 4 3 2 1
	10
	5 1 2 3 4
	120

ALCP 19 Bridge

A group of n people wish to cross a bridge at night. At most two people may cross at any time, and each group must have a flash light. Only one flash light is available among the n people, so some sort of shuttle arrangement must be arranged in order to return the flash light so that more people may cross. Each person has a different crossing speed; the speed of a group is determined by the speed of the slower member. Your job is to determine a strategy that gets all *n* people across the bridge in the minimum time.

Input

The input begins with a single positive integer on a line by itself indicating the number of test cases, followed by a blank line. There is also a blank line between each two consecutive inputs. The first line of each case contains *n*, followed by n lines giving the crossing times for each of the people. There are not more than 1,000 people and nobody takes more than100 seconds to cross the bridge.

Output

For each test case, the first line of output must report the total number of seconds required for all *n* people to cross the bridge. Subsequent lines give a strategy for achieving this time. Each line contains either one or two integers, indicating which person or people form the next group to cross. Each person is indicated by the crossing time specified in the input. Although many people may have the same crossing time, this ambiguity is of no consequence.

Note that the crossing alternate directions, as it is necessary to return the flash light so that more may cross. If more than one strategy yields the minimal time, anyone will do. The output of two consecutive cases must be separated by a blank line.

Sample Input

```
1
 4
 1
 2
 5
 10
```

Sample Output

```
17
 12
1
 510
 2
 12
```

ALCP 20 | Longest Nap

Professors lead very busy lives with full schedules of work and appointments. Professor P likes to nap during the day, but his schedule is so busy that he doesn't have many chances to do so. He really wants to take one nap every day, however. Naturally, he wants to take the longest nap possible given his schedule. Write a program to help him with the task.

Input

The input consists of an arbitrary number of test cases, where each test case represents one day. The first line of each case contains a positive integer *s* ≤ 100, representing the number of scheduled appointments for that day. The next s lines contain the appointments in the format time1 time2 appointment, where time1 represents the time which the appointment starts and time2 the time it ends. All times will be in the *hh:mm* format; the ending time will always be strictly after the starting time, and separated by a single space.

All times will be greater than or equal to 10:00 and less than or equal to 18:00. Thus your response must be in this interval as well; i.e., no nap can start before 10:00 and last after 18:00. The appointment can be any sequence of characters, but will always be on the same line. You can assume that no line is be longer than 255 characters, that 10 ≤ *hh* ≤ 18 and that 0 ≤ *mm* < 60. You cannot assume, however, that the input will be in any specific order, and must read the input until you reach the end of file.

Output

For each test case, you must print the following line:

Day #*d*: the longest nap starts at *hh : mm* and will last for [*H* hours and] *M* minutes, where d stands for the number of the test case (starting from 1) and hh : mm is the time when the nap can start. To display the length of the nap, follow these rules:

1. If the total time *X* is less than 60 minutes, just print "*X* minutes."

2. If the total duration X is at least 60 minutes, print "*H* hours and *M* minutes," where

H =*X*÷60 (integer division, of course) and *M* = *X* mod 60.

You don't have to worry about correct pluralization; i.e., you must print "1 minutes" or "1 hours" if that is the case. The duration of the nap is calculated by the difference between the ending time and the beginning time. That is, if an appointment ends at 14:00 and the next one starts at 14:47, then you have 14:47– 14:00 = 47 minutes of possible napping.

If there is more than one longest nap with the same duration, print the earliest one. You can assume the professor won't be busy all day, so there is always time for at least one possible nap.

Sample Input

```
4
10:00 12:00 Lectures
12:00 13:00 Lunch, like always.
13:00 15:00 Boring lectures...
15:30 17:45 Reading
4
10:00 12:00 Lectures
12:00 13:00 Lunch, just lunch.
13:00 15:00 Lectures, lectures... oh, no!
```

Sample Output

```
Day #1: the longest nap starts at 15:00 and will last for 30 minutes.
Day #2: the longest nap starts at 15:00 and will last for 1 hours and 45 minutes.
Day #3: the longest nap starts at 17:15 and will last for 45 minutes.
Day #4: the longest nap starts at 13:00 and will last for 5 hours and 0 minutes.
```

```
16:45 17:45 Reading (to be or not to be?)
4
10:00 12:00 Lectures, as every day.
12:00 13:00 Lunch, again!!!
13:00 15:00 Lectures, more lectures!
15:30 17:15 Reading (I love reading, but should I schedule it?)
1
12:00 13:00 I love lunch! Have you ever noticed it? :)
```

ALCP 21 Shoemaker's Problem

A shoe maker has N orders from customers which he must satisfy. The shoe maker can work on only one job in each day, and jobs usually take several days. For the i^{th} job, the integer T_i ($1 \le T_i \le 1{,}000$) denotes the number of days it takes the shoe maker to finish the job. But popularity has its price. For each day of delay before starting to work on the i^{th} job, the shoe maker has agreed to pay a fine of S_i ($1 \le S_i \le 10{,}000$) cents per day. Help the shoe maker by writing a program to find the sequence of jobs with minimum total fine.

Input

The input begins with a single positive integer on a line by itself indicating the number of the test cases, followed by a blank line. There is also a blank line between two consecutive cases. The first line of each case contains an integer reporting the number of jobs N, where $1 \le N \le 1{,}000$.The i^{th} subsequent line contains the completion time T_i and daily penalty S_i for the i^{th} job.

Output

For each test case, your program should print the sequence of jobs with minimal fine. Each job should be represented by its position in the input. All integers should be placed on only one output line and each pair separated by one space. If multiple solutions are possible, print the first one in lexicographic order. The output of two consecutive cases must be separated by a blank line.

Sample Input

```
1
4
3 4
1 1000
2 2
5 5
```

Sample Output

```
2 1 3 4
```

ALCP 22 — CDVII

Roman roads are famous for their sound engineering. Unfortunately, sound engineering does not come cheap, and some modern neo-Caesars have decided to recover the costs through automated tolling. A particular toll highway, the CDVII, has a fare structure that works as follows: travel on the road costs a certain amount per km travelled, depending on the time of day when the travel begins. Cameras at every entrance and every exit capture the license numbers of all cars entering and leaving. Every calendar month, a bill is sent to the registered owner for each km travelled (at a rate determined by the time of day), plus one dollar per trip, plus a two dollar account charge. Your job is to prepare the bill for one month, given a set of license plate photos.

Input

The input begins with an integer on a line by itself indicating the number of test cases, followed by a blank line. There will also be a blank line between each two consecutive test cases. Each test case has two parts: the fare structure and the license photos. The fare structure consists of a line with 24 non-negative integers denoting the toll (cents/*km*) from 00:00 to 00:59, the toll from 01:00 to 01:59, and so on for each hour in the day. Each photo record consists of the license number of the vehicle (up to 20 alphanumeric characters), the time and date (*mm: dd: hh: mm*), the word enter or exit, and the location of the entrance or exit (in km from one end of the highway). All dates will be within a single month. Each "enter" record is paired with the chronologically next record for the same vehicle, provided it is an "exit" record. Unpaired "enter" and "exit" records are ignored. You may assume that no two records for the same vehicle have the same time. Times are recorded using a 24-hour clock. There are not more than 1,000 photo records.

Output

For each test case, print a line for each vehicle indicating the license number and the total bill in alphabetical order by license number. The output of two consecutive cases must be separated by a blank line.

Sample Input

```
1
10 10 10 10 10 10 20 20 20 15 15 15 15 15 15 15
20 30 20 15 15 10 10 10
ABCD123 01:01:06:01 enter 17
765DEF 01:01:07:00 exit 95
ABCD123 01:01:08:03 exit 95
765DEF 01:01:05:59 enter 17
```

Sample Output

```
765DEF $10.80
ABCD123 $18.60
```

ALCP 23 Shell Sort

King Yertle wishes to rearrange his turtle throne to place his highest-ranking nobles and closest advisors nearer to the top. A single operation is available to change the order of the turtles in the stack: a turtle can crawl out of its position in the stack and climb up over the other turtles to sit on the top. Given an original ordering of a turtle stack and a required ordering for the same turtle stack, your job is to find a minimal sequence of operations that rearranges the original stack into the required stack.

Input

The first line of the input consists of a single integer *K* giving the number of test cases. Each test case consists of an integer n giving the number of turtles in the stack. The next n lines describe the original ordering of the turtle stack. Each of the lines contains the name of a turtle, starting with the turtle on the top of the stack and working down to the turtle at the bottom of the stack. Turtles have unique names, each of which is a string of no more than eighty characters drawn from a character set consisting of the alphanumeric characters, the space character and the period ("."). The next n lines in the input give the desired ordering of the stack, once again by naming turtles from top to bottom. Each test case consists of exactly $2n+1$ lines in total. The number of turtles (n) will be less than or equal to 200.

Output

For each test case, the output consists of a sequence of turtle names, one per line, indicating the order in which turtles are to leave their positions in the stack and crawl to the top. This sequence of operations should transform the original stack into the required stack and should be as short as possible. If more than one solution of shortest length is possible, any of the solutions may be reported.

Print a blank line after each test case.

Sample Input

```
2
3
Yertle
Duke of Earl
Sir Lancelot
Duke of Earl
Yertle
Sir Lancelot
9
Yertle
Duke of Earl
Sir Lancelot
Elizabeth Windsor
Michael Eisner
Richard M. Nixon
Mr. Rogers
```

Sample Output

```
Duke of Earl
Sir Lancelot
Richard M. Nixon
Yertle
```

```
Ford Perfect
Mack
Yertle
Richard M. Nixon
Sir Lancelot
Duke of Earl
Elizabeth Windsor
Michael Eisner
Mr. Rogers
Ford Perfect
Mack
```

ALCP 24 | Football (aka Soccer)

Football is the most popular sport in the world, even though Americans insist on calling it "soccer." A country such as five-time World Cup-winning Brazil has so many national and regional tournaments that is it very difficult to keep track. Your task is to write a program that receives the tournament name, team names, and games played and outputs the tournament standings so far. A team wins a game if it scores more goals than its opponent, and loses if it scores fewer goals. Both teams tie if they score the same number of goals. A team earns 3 points for each win, 1 point for each tie, and 0 points for each loss. Teams are ranked according to these rules (in this order):

1. Most points earned.
2. Most wins.
3. Most goal difference (i.e., goals scored– goals against)
4. Most goals scored.
5. Fewest games played.
6. Case-insensitive lexicographic order.

Input

The first line of input will be an integer *N* in a line alone (0 <*N*<1,000). Then follow N tournament descriptions, each beginning with a tournament name. These names can be any combination of at most 100 letters, digits, spaces, etc., on a single line. The next line will contain a number *T* (1 <*T*≤ 30), which stands for the number of teams participating on this tournament. Then follow *T* lines, each containing one team name. Team names consist of at most 30 characters, and may contain any character with ASCII code greater than or equal to 32 (space), except for "#" and "@" characters. Following the team names, there will be a non-negative integer G on a single line which stands for the number of games already played on this tournament. *G* will be no greater than 1,000. *G* lines then follow with the results of games played in the format:

team name 1#goals1@goals2#team name 2

For instance, Team *A*#3@1#Team *B* means that in a game between Team *A* and Team *B*, Team *A* scored 3 goals and Team *B* scored 1. All goals will be non-negative integers less than 20. You may assume that all team names mentioned in game results will have appeared in the team names section, and that no team will play against itself.

Output

For each tournament, you must output the tournament name in a single line. In the next *T* lines you must output the standings, according to the rules above. Should lexicographic order be needed as a tie-breaker, it must be done in a case-insensitive manner. The output format for each line is shown below: [*a*]) Team name [*b*]*p*, [*c*]*g*([*d*]-[*e*]-[f]), [*g*]*gd* ([*h*]-[*i*])

where [a] is team rank, [b] is the total points earned, [c] is the number of games played, [d] is wins, [*e*] is ties, [*f*] is losses, [*g*] is goal difference, [*h*] is goals scored, and [*i*] is goals against.

There must be a single blank space between fields and a single blank line between output sets. See the sample output for examples.

Sample Input	**Sample Output**
2	World Cup 1998- Group A
World Cup 1998- Group A	1) Brazil 6p, 3g (2-0-1), 3gd (6-3)
4	2) Norway 5p, 3g (1-2-0), 1gd (5-4)
Brazil	3) Morocco 4p, 3g (1-1-1), 0gd (5-5)

Norway Morocco Scotland 6 Brazil#2@1#Scotland Norway#2@2#Morocco Scotland#1@1#Norway Brazil#3@0#Morocco Morocco#3@0#Scotland Brazil#1@2#Norway Some strange tournament 5 Team A Team B Team C Team D Team E 5 Team A#1@1#Team B Team A#2@2#Team C Team A#0@0#Team D Team E#2@1#Team C Team E#1@2#Team D	4) Scotland 1p, 3g (0-1-2),-4gd (2-6) Some strange tournament 1) Team D 4p, 2g (1-1-0), 1gd (2-1) 2) Team E 3p, 2g (1-0-1), 0gd (3-3) 3) Team A 3p, 3g (0-3-0), 0gd (3-3) 4) Team B 1p, 1g (0-1-0), 0gd (1-1) 5) Team C 1p, 2g (0-1-1),-1gd (3-4)

ALCP 25 Bicoloring

The four-color theorem states that every planar map can be colored using only four colors in such a way that no region is colored using the same color as a neighbour. After being open for over 100 years, the theorem was proven in 1976 with the assistance of a computer. Here you are asked to solve a simpler problem. Decide whether a given connected graph can be bicolored, i.e., can the vertices be painted red and black such that no two adjacent vertices have the same color. To simplify the problem, you can assume the graph will be connected, undirected, and not contain self-loops (i.e., edges from a vertex to itself).

Input

The input consists of several test cases. Each test case starts with a line containing the number of vertices *n*, where 1 <*n*< 200. Each vertex is labelled by a number from 0 to *n* − 1. The second line contains the number of edges *l*. After this, *l* lines follow, each containing two vertex numbers specifying an edge. An input with *n* = 0 marks the end of the input and is not to be processed.

Output

Decide whether the input graph can be bicolored, and print the result as shown below.

Sample Input

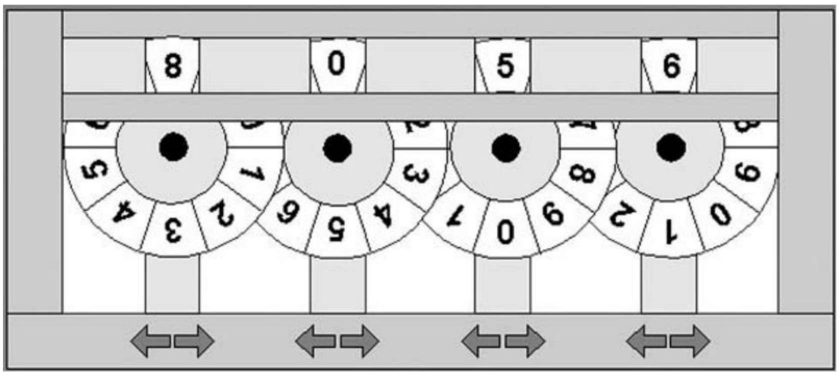
```
3
3
0 1
1 2
2 0
9
8
0 1
0 2
0 3
0 4
0 5
0 6
0 7
0 8
0
```

Sample Output

```
NOT BICOLORABLE.
BICOLORABLE
```

ALCP 26 Playing With Wheels

Consider the following mathematical machine. Digits ranging from 0 to 9 are printed consecutively (clockwise) on the periphery of each wheel. The topmost digits of the wheels form a four-digit integer. For example, in the following figure the wheels form the integer 8,056. Each wheel has two buttons associated with it. Pressing the button marked with a left arrow rotates the wheel one digit in the clockwise direction and pressing the one marked with the right arrow rotates it by one digit in the opposite direction.



We start with an initial configuration of the wheels, with the topmost digits forming the integer $S_1S_2S_3S_4$. You will be given a set of n forbidden configurations F_{i1} F_{i2} F_{i3} F_{i4} $(1 \le i \le n)$ and a target configuration $T_1T_2T_3T_4$. Your job is to write a program to calculate the minimum number of button presses required to transform the initial configuration to the target configuration without passing through a forbidden one.

Input

The first line of the input contains an integer *N* giving the number of test cases. A blank line then follows. The first line of each test case contains the initial configuration of the wheels, specified by four digits. Two consecutive digits are separated by a space. The next line contains the target configuration. The third line contains an integer *n* giving the number of forbidden configurations. Each of the following n lines contains a forbidden configuration. There is a blank line between two consecutive input sets.

Output

For each test case in the input print a line containing the minimum number of button presses required. If the target configuration is not reachable print "-1".

Sample Input

```
2
8 0 5 6
6 5 0 8
5
8 0 5 7
8 0 4 7
5 5 0 8
7 5 0 8
6 4 0 8
0 0 0 0
```

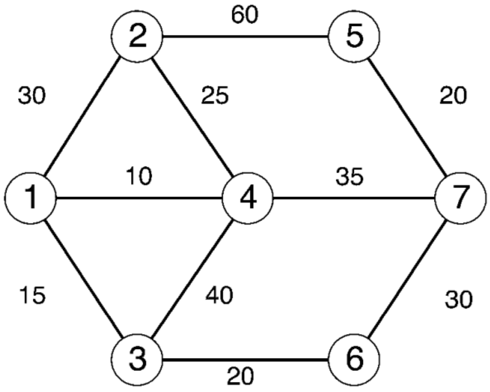
Sample Output

```
14
-1
```

```
5 3 1 7
8
0 0 0 1
0 0 0 9
0 0 1 0
0 0 9 0
0 1 0 0
0 9 0 0
1 0 0 0
9 0 0 0
```

ALCP 27 | The Tourist Guide

Mr. G. works as a tourist guide in Bangladesh. His current assignment is to show a group of tourists a distant city. As in all countries, certain pairs of cities are connected by two-way roads. Each pair of neighbouring cities has a bus service that runs only between those two cities and uses the road that directly connects them. Each bus service has a particular limit on the maximum number of passengers it can carry. Mr. G. has a map showing the cities and the roads connecting them, as well as the service limit for each bus service. It is not always possible for him to take all the tourists to the destination city in a single trip. For example, consider the following road map of seven cities, where the edges represent roads and the number written on each edge indicates the passenger limit of the associated bus service.



It will take at least five trips for Mr. G. to take 99 tourists from city 1 to city 7, since he has to ride the bus with each group. The best route to take is 1 - 2 - 4 - 7. Help Mr. G. find the route to take all his tourists to the destination city in the minimum number of trips.

Input

The input will contain one or more test cases. The first line of each test case will contain two integers: *N* (*N* ≤ 100) and *R*, representing the number of cities and the number of road segments, respectively. Each of the next R lines will contain three integers (C_1, C_2, and *P*) where C_1 and C_2 are the city numbers and *P* (*P* > 1) is the maximum number of passengers that can be carried by the bus service between the two cities. City numbers are positive integers ranging from 1 to *N*. The $(R + 1)^{th}$ line will contain three integers (*S*, *D*, and *T*) representing, respectively, the starting city, the destination city, and the number of tourists to be guided. The input will end with two zeros for *N* and *R*.

Output

For each test case in the input, first output the scenario number and then the minimum number of trips required for this case on a separate line. Print a blank line after the output for each test case.

Sample Input

```
7 10
1 2 30
1 3 15
1 4 10
2 4 25
2 5 60
3 4 40
3 6 20
```

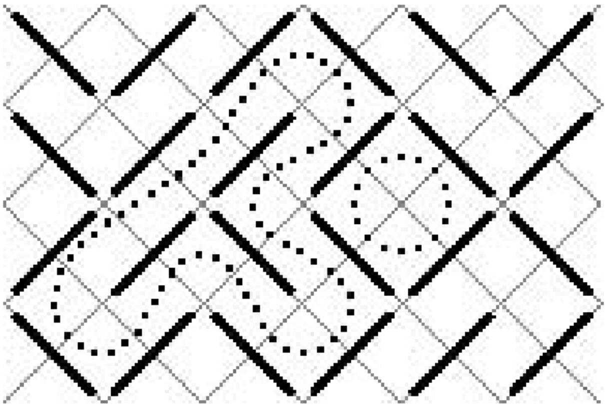
Sample Output

```
Scenario #1
Minimum Number of Trips = 5
```

```
4 7 35
5 7 20
6 7 30
1 7 99
0 0
```

ALCP 28 — Slash Maze

By filling a rectangle with slashes (/) and backslashes (\), you can generate nice little mazes. Here is an example:



As you can see, paths in the maze cannot branch, so the whole maze contains only (1) cyclic paths and (2) paths entering somewhere and leaving somewhere else. We are only interested in the cycles. There are exactly two of them in our example. Your task is to write a program that counts the cycles and finds the length of the longest one. The length is defined as the number of small squares the cycle consists of (the ones bordered by grey lines in the picture). In this example, the long cycle has length 16 and the short one length 4.

Input

The input contains several maze descriptions. Each description begins with one line containing two integers w and h (1 ≤ *w*, *h* ≤ 75), representing the width and the height of the maze. The next *h* lines describe the maze itself and contain w characters each; all of these characters will be either "/" or "\". The input is terminated by a test case beginning with *w* = *h* = 0. This case should not be processed.

Output

For each maze, first output the line "Maze #*n*:", where n is the number of the maze. Then, output the line "*k* Cycles; the longest has length *l*.", where k is the number of cycles in the maze and l the length of the longest of the cycles. If the maze is acyclic, output "There are no cycles." Output a blank line after each test case.

Sample Input

```
6 4
\//\\/
\///\/
//\\/\
\/\///
3 3
///
\//
\\\
0 0
```

Sample Output

```
Maze #1:
2 Cycles; the longest has length 16.
Maze #2:
There are no cycles.
```

ALCP 29 | Edit Step Ladders

An edit step is a transformation from one word *x* to another word *y* such that *x* and *y* are words in the dictionary, and *x* can be transformed to *y* by adding, deleting, or changing one letter. The transformations from dig to dog and from dog to do are both edit steps. An edit step ladder is a lexicographically ordered sequence of words w_1, w_2,...,w_n such that the transformation from w_i *to* w_{i+1} is an edit step for all *i* from 1 to *n* − 1.

For a given dictionary, you are to compute the length of the longest edit step ladder.

Input

The input to your program consists of the dictionary: a set of lowercase words in lexicographic order at one word per line. No word exceeds 16 letters and there are at most 25,000 words in the dictionary.

Output

The output consists of a single integer, the number of words in the longest edit step ladder.

Sample Input	Sample Output
cat dig dog fig fin fine fog log wine	5

ALCP 30 | Tower of Cubes

You are given *N* colorful cubes, each having a distinct weight. Cubes are not monochromatic – indeed, every face of a cube is colored with a different color. Your job is to build the tallest possible tower of cubes subject to the restrictions that (1) we never put a heavier cube on a lighter one, and (2) the bottom face of every cube (except the bottom one) must have the same color as the top face of the cube below it.

Input

The input may contain several test cases. The first line of each test case contains an integer *N* ($1 \le N \le 500$) indicating the number of cubes you are given. The i^{th} of the next *N* lines contains the description of the i^{th} cube. A cube is described by giving the colors of its faces in the following order: front, back, left, right, top, and bottom face.

For your convenience colors are identified by integers in the range 1 to 100. You may assume that cubes are given in increasing order of their weights; that is, cube 1 is the lightest and cube *N* is the heaviest. The input terminates with a value 0 for *N*.

Output

For each case, start by printing the test case number on its own line as shown in the sample output. On the next line, print the number of cubes in the tallest possible tower. The next line describes the cubes in your tower from top to bottom with one description per line. Each description gives the serial number of this cube in the input, followed by a single whitespace character and then the identification string (front, back, left, right, top, or bottom of the top face of the cube in the tower. There may be multiple solutions, but any one of them is acceptable.

Print a blank line between two successive test cases.

Sample Input

```
3
1 2 2 2 1 2
3 3 3 3 3 3
3 2 1 1 1 1
10
1 5 10 3 6 5
2 6 7 3 6 9
5 7 3 2 1 9
1 3 3 5 8 10
6 6 2 2 4 4
1 2 3 4 5 6
10 9 8 7 6 5
6 1 2 3 4 7
1 2 3 3 2 1
3 2 1 1 2 3
0
```

Sample Output

```
Case #1
2
2 front
3 front
Case #2
8
1 bottom
2 back
3 right
4 left
6 top
8 front
9 front
10 top
```

ALCP 31 — From Dusk Till Dawn

Vladimir has white skin, very long teeth and is 600 years old, but this is no problem because Vladimir is a vampire. Vladimir has never had any problems with being a vampire. In fact, he is a successful doctor who always takes the night shift and so has made many friends among his colleagues. He has an impressive trick which he loves to show at dinner parties: he can tell blood group by taste. Vladimir loves to travel, but being a vampire, he must overcome three problems.

1. He can only travel by train, because he must take his coffin with him. Fortunately, he can always travel first class because he has made a lot of money through long term investments.

2. He can only travel from dusk till dawn, namely, from 6 P.M. to 6 A.M. During the day he has must stay inside a train station.

3. He has to take something to eat with him. He needs one litre of blood per day, which he drinks at noon (12:00) inside his coffin.

Help Vladimir to find the shortest route between two given cities, so that he can travel with the minimum amount of blood. If he takes too much with him, people ask him funny questions like, "What are you doing with all that blood?"

Input

The first line of the input will contain a single number telling you the number of test cases. Each test case specification begins with a single number telling you how many route specifications follow. Each route specification consists of the names of two cities, the departure time from city one, and the total traveling time, with all times in hours. Remember, Vladimir cannot use routes departing earlier than 18:00 or arriving later than 6:00.

There will be at most 100 cities and less than 1,000 connections. No route takes less than 1 hour or more than 24 hours, but Vladimir can use only routes within the 12 hours travel time from dusk till dawn. All city names are at most 32 characters long. The last line contains two city names. The first is Vladimir's start city; the second is Vladimir's destination.

Output

For each test case you should output the number of the test case followed by "Vladimir needs # litre(s) of blood." or "There is no route Vladimir can take.".

Sample Input

```
2
3
Ulm Muenchen 17 2
Ulm Muenchen 19 12
Ulm Muenchen 5 2
Ulm Muenchen
10
Lugoj Sibiu 12 6
Lugoj Sibiu 18 6
Lugoj Sibiu 24 5
Lugoj Medias 22 8
Lugoj Medias 18 8
```

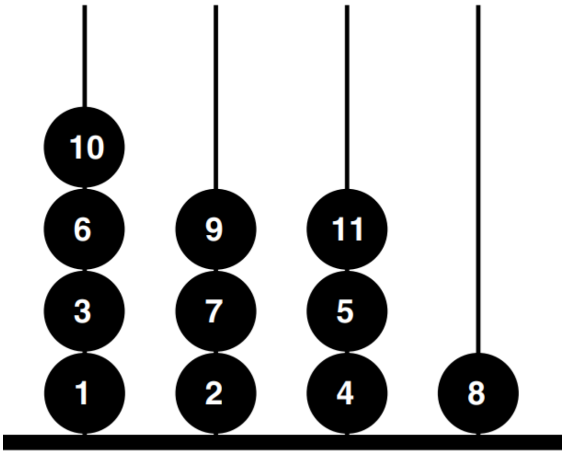
Sample Output

```
Test Case 1.
There is no route Vladimir can take.
Test Case 2.
Vladimir needs 2 litre(s) of blood.
```

```
Lugoj Reghin 17 4
Sibiu Reghin 19 9
Sibiu Medias 20 3
Reghin Medias 20 4
Reghin Bacau 24 6
Lugoj Bacau
```

ALCP 32	Hanoi Tower Troubles Again!

There are many interesting variations on the Tower of Hanoi problem. This version consists of *N* pegs and one ball containing each number from 1, 2, 3,...,∞. Whenever the sum of the numbers on two balls is not a perfect square (i.e., c^2 for some integer *c*), they will repel each other with such force that they can never touch each other.



The player must place balls on the pegs one by one, in order of increasing ball number (i.e., first ball 1, then ball 2, then ball 3. . .). The game ends where there is no non-repelling move. The goal is to place as many balls on the pegs as possible. The figure above gives a best possible result for 4 pegs.

Input

The first line of the input contains a single integer T indicating the number of test cases ($1 \le T \le 50$). Each test case contains a single integer *N* ($1 \le N \le 50$) indicating the number of pegs available.

Output

For each test case, print a line containing an integer indicating the maximum number of balls that can be placed. Print "-1" if an infinite number of balls can be placed.

Sample Input

```
2
4
25
```

Sample Output

```
11
337
```

ALCP 33 Freckles

In an episode of the Dick Van Dyke show, little Richie connects the freckles on his Dad's back to form a picture of the Liberty Bell. Alas, one of the freckles turns out to be a scar, so his Ripley's engagement falls through. Consider Dick's back to be a plane with freckles at various (x, y) locations. Your job is to tell Richie how to connect the dots so as to minimize the amount of ink used. Richie connects the dots by drawing straight lines between pairs, possibly lifting the pen between lines. When Richie is done there must be a sequence of connected lines from any freckle to any other freckle.

Input

The input begins with a single positive integer on a line by itself indicating the number of test cases, followed by a blank line. The first line of each test case contains $0 < n \le 100$, giving the number of freckles on Dick's back. For each freckle, a line follows; each following line contains two real numbers indicating the (x, y) coordinates of the freckle. There is a blank line between each two consecutive test cases.

Output

For each test case, your program must print a single real number to two decimal places: the minimum total length of ink lines that can connect all the freckles. The output of each two consecutive cases must be separated by a blank line.

Sample Input

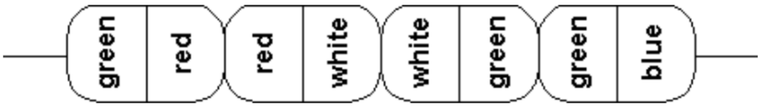
```
1
3
1.0  1.0
2.0  2.0
2.0  4.0
```

Sample Output

```
3.41
```

ALCP 34 | The Necklace

My little sister had a beautiful necklace made of colorful beads. Each two successive beads in the necklace shared a common color at their meeting point, as shown below:



But, alas! One day, the necklace tore and the beads were scattered all over the floor. My sister did her best to pick up all the beads, but she is not sure whether she found them all. Now she has come to me for help. She wants to know whether it is possible to make a necklace using all the beads she has in the same way that her original necklace was made. If so, how can the beads be so arranged? Write a program to solve the problem.

Input

The first line of the input contains the integer *T*, giving the number of test cases. The first line of each test case contains an integer *N* (5 ≤ *N* ≤ 1,000) giving the number of beads my sister found. Each of the next *N* lines contains two integers describing the colors of a bead. Colors are represented by integers ranging from 1 to 50.

Output

For each test case, print the test case number as shown in the sample output. If reconstruction is impossible, print the sentence "some beads may be lost" on a line by itself. Otherwise, print *N* lines, each with a single bead description such that for 1 ≤ i ≤ *N* − 1, the second integer on line *i* must be the same as the first integer on line *i* + 1. Additionally, the second integer on line N must be equal to the first integer on line 1. There may be many solutions, any one of which is acceptable.

Print a blank line between two successive test cases.

Sample Input

```
2
5
1 2
2 3
3 4
4 5
5 6
5
2 1
2 2
3 4
3 1
2 4
```

Sample Output

```
Case #1
some beads may be lost
Case #2
2 1
1 3
3 4
4 2
2 2
```

ALCP 35

Fire Station

A city is served by a number of fire stations. Residents have complained that the distance between certain houses and the nearest station is too far, so a new station is to be built. You are to choose the location of the new station so as to reduce the distance to the nearest station from the houses of the poorest-served residents. The city has up to 500 intersections, connected by road segments of various lengths. No more than 20 road segments intersect at a given intersection. The locations of houses and fire stations alike are considered to be at intersections. Furthermore, we assume that there is at least one house associated with every intersection. There may be more than one fire station per intersection.

Input

The input begins with a single line indicating the number of test cases, followed by a blank line. There will also be a blank line between each two consecutive inputs. The first line of input contains two positive integers: the number of existing fire stations f ($f \le 100$) and the number of intersections i ($i \le 500$). Intersections are numbered from 1 to i consecutively. Then f lines follow, each containing the intersection number at which an existing fire station is found. A number of lines follow, each containing three positive integers: the number of an intersection, the number of a different intersection, and the length of the road segment connecting the intersections. All road segments are two-way (at least as far as fire engines are concerned), and there will exist a route between any pair of intersections.

Output

For each test case, output the lowest intersection number at which a new fire station can be built so as to minimize the maximum distance from any intersection to its nearest fire station. Separate the output of each two consecutive cases by a blank line.

Sample Input

```
1
1 6
2
1 2 10
2 3 10
3 4 10
4 5 10
5 6 10
6 1 10
```

Sample Output

```
5
```

ALCP 36	Railroads

Tomorrow morning Jill must travel from Hamburg to Darmstadt to compete in the regional programming contest. Since she is afraid of arriving late and being excluded from the contest, she is looking for the train which gets her to Darmstadt as early as possible. However, she dislikes getting to the station too early, so if there are several schedules with the same arrival time then she will choose the one with the latest departure time.

Jill asks you to help her with her problem. You are given a set of railroad schedules from which you must compute the train with the earliest arrival time and the fastest connection from one location to another. Fortunately, Jill is very experienced in changing trains and can do this instantaneously, i.e., in zero time!

Input

The very first line of the input gives the number of scenarios. Each scenario consists of three parts. The first part lists the names of all cities connected by the railroads. It starts with a number $1 < C \leq 100$, followed by C lines containing city names. All names consist only of letters.

The second part describes all the trains running during a day. It starts with a number $T \leq 1,000$ followed by T train descriptions. Each of them consists of one line with a number $t_i \leq 100$ and then t_i more lines, each with a time and a city name, meaning that passengers can get on or off the train at that time at that city.

The final part consists of three lines: the first containing the earliest possible starting time, the second the name of the city where she starts, and the third with the destination city. The start and destination cities are always different.

Output

For each scenario print a line containing "Scenario *i*", where i is the scenario number starting from 1. If a connection exists, print the two lines containing zero padded timestamps and locations as shown in the example. Use blanks to achieve the indentation. If no connection exists on the same day (i.e., arrival before midnight), print a line containing "No connection".

Print a blank line after each scenario.

Sample Input

```
2
3
Hamburg
Frankfurt
Darmstadt
3
2
0949 Hamburg
1006 Frankfurt
2
1325 Hamburg
1550 Darmstadt
2
```

Sample Output

```
Scenario 1
Departure 0949 Hamburg
Arrival 1411 Darmstadt
Scenario 2
No connection
```

```
1205 Frankfurt
1411 Darmstadt
0800
Hamburg
Darmstadt
2
Paris
Tokyo
1
2
0100 Paris
2300 Tokyo
0800
Paris
Tokyo
```

ALCP 37 | War

A war is being fought between two countries, *A* and *B*. As a loyal citizen of *C*, you decide to help your country by secretly attending the peace talks between *A* and *B*. There are *n* other people at the talks, but you do not know which person belongs to which country. You can see people talking to each other, and by observing their behaviour during occasional one-to-one conversations you can guess if they are friends or enemies.

Your country needs to know whether certain pairs of people are from the same country, or whether they are enemies. You can expect to receive such questions from your government during the peace talks, and will have to give replies on the basis of your observations so far.

Now, more formally, consider a black box with the following operations:

setFriends(*x, y*) shows that *x* and *y* are from the same country

setEnemies(*x, y*) shows that x and y are from different countries

areFriends(*x, y*) returns true if you are sure that *x* and *y* are friends

areEnemies(*x, y*) returns true if you are sure that *x* and y are enemies

The first two operations should signal an error if they contradict your former knowledge. The two relations "friends" (denoted by $\sim$) and "enemies" (denoted by $*$) have the following properties:

$\sim$ is an equivalence relation: i.e.,

1. If $x \sim y$ and $y \sim z$, then $x \sim z$ (The friends of my friends are my friends as well.)
2. If $x \sim y$, then $y \sim x$ (Friendship is mutual.)
3. $x \sim x$ (Everyone is a friend of himself.)

$*$ is symmetric and irreflexive:

1. If $x * y$ then $y * x$ (Hatred is mutual.)
2. Not $x * x$ (Nobody is an enemy of himself.)
3. If $x * y$ and $y * z$ then $x \sim z$ (A common enemy makes two people friends.)
4. If $x \sim y$ and y $* z$ then $x * z$ (An enemy of a friend is an enemy.)

Operations setFriends(x, y) and setEnemies(x, y) must preserve these properties.

Input

The first line contains a single integer, *n*, the number of people. Each subsequent line contains a triple of integers, *cxy*, where *c* is the code of the operation,

c = 1, setFriends

c = 2, setEnemies

c = 3, areFriends

c = 4, areEnemies

and *x* and *y* are its parameters, integers in the range [0, *n*) identifying two different people. The last line contains 0 0 0. All integers in the input file are separated by at least one space or line break. There are at most 10,000 people, but the number of operations is unconstrained.

Output

For every are Friends and are Enemies operation write "0" (meaning no) or "1" (meaning yes) to the output. For every setFriends or setEnemies operation which conflicts with previous knowledge, output a "-1" to the output; such an operation should produce no other effect and execution should continue. A successful setFriends or setEnemies gives no output.

All integers in the output file must be separated by one line break.

Sample Input

```
10
1 0 1
1 1 2
2 0 5
3 0 2
3 8 9
4 1 5
4 1 2
4 8 9
1 8 9
1 5 2
3 5 2
0 0 0
```

Sample Output

```
1
0
1
0
0
-1
0
```

ALCP 38 — Tourist Guide

Rio de Janeiro is a very beautiful city, but there are so many places to visit that sometimes you feel overwhelmed, fortunately, your friend Bruno has promised to be your tour guide. Unfortunately, Bruno is terrible driver. He has a lot of traffic fines to pay and is eager to avoid paying more. Therefore, he wants to know where all the police cameras are located so he can drive more carefully when passing by them. These cameras are strategically distributed over the city, in locations that a driver must pass through in order to travel from one zone of the city to another. A location *C* will have a camera if and only if there are two city locations *A* and *B* such that all paths from *A* to *B* pass through a location *C*. For instance, suppose that we have six locations (*A, B, C, D, E,* and *F*) with seven bidirectional routes *B* − *C*, *A* − *B*, *C* − *A*, *D* − *C*, *D* − *E*, *E* − *F*, and *F* − *C*. There must be a camera on *C* because to go from *A* to *E* you must pass through *C*. In this configuration, *C* is the only camera location.

Given a map of the city, help Bruno avoid further fines during your tour by writing a program to identify where all the cameras are.

Input

The input will consist of an arbitrary number of city maps, where each map begins with an integer *N* (2 < *N* ≤ 100) denoting the total number of locations in the city. Then follow N different place names at one per line, where each place name will consist of least one and at most 30 lowercase letters. A non-negative integer *R* then follows, denoting the total routes of the city. The next *R* lines each describe a bidirectional route represented by the two places that the route connects.

Location names in route descriptions will always be valid, and there will be no route from one place to itself. You must read until *N* = 0, which should not be processed.

Output

For each city map you must print the following line:

City map #*d*: *c* camera(s) found

where d stands for the city map number (starting from 1) and c stands for the total number of cameras. Then should follow c lines with the location names of each camera in alphabetical order. Print a blank line between output sets.

Sample Input

```
6
sugarloaf
maracana
copacabana
ipanema
corcovado
lapa
7
ipanema copacabana
copacabana sugarloaf
ipanema sugarloaf
maracana lapa
sugarloaf maracana
```

Sample Output

```
City map #1: 1 camera(s) found
sugarloaf
City map #2: 1 camera(s) found
sambodromo
```

```
corcovado sugarloaf
lapa corcovado
5
guanabarabay
downtown
botanicgarden
colombo
sambodromo
4
guanabarabay sambodromo
downtown sambodromo
sambodromo botanicgarden
colombo sambodromo
0
```

ALCP 39	The Grand Dinner

Each team participating in this year's ACM World Finals is expected to attend the grand banquet arranged for after the award ceremony. To maximize the amount of interaction between members of different teams, no two members of the same team will be allowed to sit at the same table.

Given the number of members on each team (including contestants, coaches, reserves, and guests) and the seating capacity of each table, determine whether it is possible for the teams to sit as described. If such an arrangement is possible, output one such seating assignment. If there are multiple possible arrangements, any one is acceptable.

Input

The input file may contain multiple test cases. The first line of each test case contains two integers, $1 \le M \le 70$ and $1 \le N \le 50$, denoting the number of teams and tables, respectively. The second line of each test case contains *M* integers, where the i^{th} integer mi indicates the number of members of team i. There are at most 100 members of any team. The third line contains *N* integers, where the j^{th} integer n_j , $2 \le nj \le 100$, indicates the seating capacity of table *j*. A test case containing two zeros for *M* and *N* terminates the input.

Output

For each test case, print a line containing either 1 or 0, denoting whether there exists a valid seating arrangement of the team members. In case of a successful arrangement, print *M* additional lines where the i^{th} line contains a table number (from 1 to *N*) for each of the members of team i.

Sample Input	**Sample Output**
4 5	1
4 5 3 5	1 2 4 5
3 5 2 6 4	1 2 3 4 5
4 5	2 4 5
4 5 3 5	1 2 3 4 5
3 5 2 6 3	0
0 0	

ALCP 40 — The Problem with the Problem Setter

So many students are interested in participating in this year's regional programming contest that we have decided to arrange a screening test to identify the most promising candidates. This test may include as many as 100 problems drawn from as many as 20 categories. I have been assigned the job of setting problems for this test. At first the job seemed to be very easy, since I was told that I would be given a pool of about 1,000 problems divided into appropriate categories. After getting the problems, however, I discovered that the original authors often wrote down multiple category names in the category fields. Since no problem can used in more than one category and the number of problems needed for each category is fixed, assigning problems for the test is not so easy.

Input

The input file may contain multiple test cases, each of which begins with a line containing two integers, n_k and n_p, where n_k is the number of categories and n_p is the number of problems in the pool. There will be between 2 and 20 categories and at most 1,000 problems in the pool.

The second line contains n_k positive integers, where the i^{th} integer specifies the number of problems to be included in category i ($1 \le i \le n_k$) of the test. You may assume that the sum of these n_k integers will never exceed 100. The j^{th} ($1 \le j \le n_p$) of the next np lines contains the category information of the j^{th} problem in the pool.

Each such problem category specification starts with a positive integer specifying the number of categories in which this problem can be included, followed by the actual category numbers. A test case containing two zeros for n_k and n_p terminates the input.

Output

For each test case, print a line reporting whether problems can be successfully selected from the pool under the given restrictions, with 1 for success and 0 for failure. In case of successful selection, print n_k additional lines where the i^{th} line contains the problem numbers that can be included in category i. Problem numbers are positive integers not greater then np and each two problem numbers must be separated by a single space. Any successful selection will be accepted.

Sample Input

```
3 15
3 3 4
2 1 2
1 3
1 3
1 3
1 3
3 1 2 3
2 2 3
2 1 3
1 2
1 2
2 1 2
2 1 3
2 1 2
```

Sample Output

```
1
8 11 12
1 6 7
2 3 4 5
0
```

```
1 1
3 1 2 3
3 15
7 3 4
2 1 2
1 1
1 2
1 2
1 3
3 1 2 3 2 2 3
2 2 3
1 2
1 2
2 2 3
2 2 3
2 1 2
1 1
3 1 2 3
0 0
```

ALCP 41 — Is Bigger Smarter?

Some people think that the bigger an elephant is, the smarter it is. To disprove this, you want to analyze a collection of elephants and place as large a subset of elephants as possible into a sequence whose weights are increasing but IQ's are decreasing.

Input

The input will consist of data for a bunch of elephants, at one elephant per line terminated by the end-of-file. The data for each particular elephant will consist of a pair of integers: the first representing its size in kilograms and the second representing its IQ in hundredths of IQ points. Both integers are between 1 and 10,000. The data contains information on at most 1,000 elephants. Two elephants may have the same weight, the same IQ, or even the same weight and IQ.

Output

The first output line should contain an integer *n*, the length of elephant sequence found. The remaining n lines should each contain a single positive integer representing an elephant. Denote the numbers on the i^{th} data line as *W[i]* and *S[i]*. If these sequence of n elephants are *a*[1], *a*[2],..., *a*[*n*] then it must be the case that W[a[1]] < *W*[*a*[2]] < ... < W[a[n]] and S[a[1]] > S[a[2]] > ... > S[a[n]]i

In order for the answer to be correct, n must be as large as possible. All inequalities are strict: weights must be strictly increasing, and IQs must be strictly decreasing.

Your program can report any correct answer for a given input.

Sample Input

```
6008  1300
6000  2100
500  2000
1000  4000
1100  3000
6000  2000
8000  1400
6000  1200
2000  1900
```

Sample Output

```
4
4
5
9
7
```

ALCP 42 Distinct Subsequences

A subsequence of a given sequence S consists of S with zero or more elements deleted. Formally, a sequence $Z = z_1 z_2 \ldots z_k$ is a subsequence of $X = x_1 x_2 \ldots x_m$ if there exists a strictly increasing sequence < $i_1, i_2,\ldots,i_k$ > of indices of X such that for all j = 1, 2,...,k, we have $x_{ij} = z_j$. For example, Z = *bcdb* is a subsequence of X = *abcbdab* with corresponding index sequence < 2, 3, 5, 7 >.

Your job is to write a program that counts the number of occurrences of Z in X as a subsequence such that each has a distinct index sequence.

Input

The first line of the input contains an integer N indicating the number of test cases to follow. The first line of each test case contains a string X, composed entirely of lowercase alphabetic characters and having length no greater than 10,000. The second line contains another string Z having length no greater than 100 and also composed of only lowercase alphabetic characters. Be assured that neither Z nor any prefix or suffix of Z will have more than 10^{100} distinct occurrences in X as a subsequence.

Output

For each test case, output the number of distinct occurrences of Z in X as a subsequence.

Output for each input set must be on a separate line.

Sample Input	**Sample Output**
2	5
babgbag	3
bag	
rabbbit	
rabbit	

ALCP 43 Weights and Measures

A turtle named Mack, to avoid being cracked, has enlisted your advice as to the order in which turtles should be stacked to form Yertle the Turtle's throne. Each of the 5,607 turtles ordered by Yertle has a different weight and strength. Your task is to build the largest stack of turtles possible.

Input

Standard input consists of several lines, each containing a pair of integers separated by one or more space characters, specifying the weight and strength of a turtle. The weight of the turtle is in grams. The strength, also in grams, is the turtle's overall carrying capacity, including its own weight. That is, a turtle weighing 300 g with a strength of 1,000 g can carry 700 g of turtles on its back. There are at most 5,607 turtles.

Output

Your output is a single integer indicating the maximum number of turtles that can be stacked without exceeding the strength of any one.

Sample Input

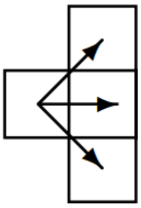
```
300 1000
1000 1200
200 600
100 101
```

Sample Output

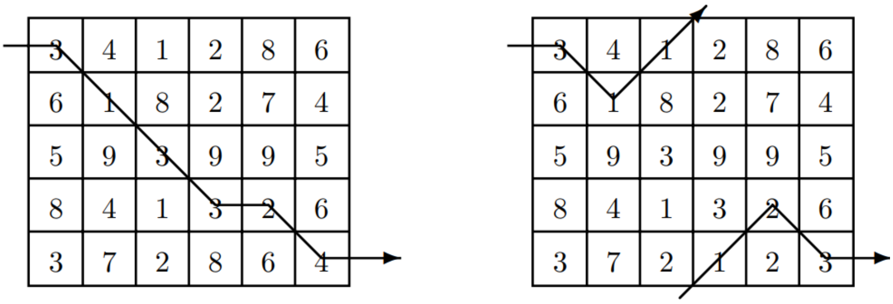
```
3
```

ALCP 44	Unidirectional TSP

Given an *m*×*n* matrix of integers, you are to write a program that computes a path of minimal weight from left to right across the matrix. A path starts anywhere in column 1 and consists of a sequence of steps terminating in column n. Each step consists of traveling from column *i* to column *i* + 1 in an adjacent (horizontal or diagonal) row. The first and last rows (rows 1 and m) of a matrix are considered adjacent; i.e., the matrix "wraps" so that it represents a horizontal cylinder. Legal steps are illustrated below.



The weight of a path is the sum of the integers in each of the n cells of the matrix that are visited. The minimum paths through two slightly different 5 × 6 matrices are shown below. The matrix values differ only in the bottom row. The path for the matrix on the right takes advantage of the adjacency between the first and last rows.



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

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

Input

The input consists of a sequence of matrix specifications. Each matrix consists of the row and column dimensions on a line, denoted *m* and *n*, respectively. This is followed by m · n integers, appearing in row major order; i.e., the first n integers constitute the first row of the matrix, the second n integers constitute the second row, and so on. The integers on a line will be separated from other integers by one or more spaces.

Note: integers are not restricted to being positive. There will be one or more matrix specifications in an input file. Input is terminated by end-of-file. For each specification the number of rows will be between 1 and 10 inclusive; the number of columns will be between 1 and 100 inclusive. No path's weight will exceed integer values representable using 30 bits.

Output

Two lines should be output for each matrix specification. The first line represents a minimal-weight path, and the second line is the cost of this minimal path. The path consists of a sequence of n integers (separated by one or more spaces) representing the rows that constitute the minimal path. If there is more than one path of minimal weight, the lexicographically smallest path should be output.

Sample Input	**Sample Output**
5 6	1 2 3 4 4 5
3 4 1 2 8 6	16
6 1 8 2 7 4	1 2 1 5 4 5
5 9 3 9 9 5	11

8 4 1 3 2 6	1 1
3 7 2 8 6 4	19
5 6	
3 4 1 2 8 6	
6 1 8 2 7 4	
5 9 3 9 9 5	
8 4 1 3 2 6	
3 7 2 1 2 3	
2 2	
9 10 9 10	

ALCP 45 — Cutting Sticks

You have to cut a wood stick into several pieces. The most affordable company, Analog Cutting Machinery (ACM), charges money according to the length of the stick being cut. Their cutting saw allows them to make only one cut at a time.

It is easy to see that different cutting orders can lead to different prices. For example, consider a stick of length 10 m that has to be cut at 2, 4, and 7 m from one end. There are several choices. One can cut first at 2, then at 4, then at 7. This leads to a price of 10 + 8 + 6 = 24 because the first stick was of 10 m, the resulting stick of 8 m, and the last one of 6 m. Another choice could cut at 4, then at 2, then at 7. This would lead to a price of 10 + 4 + 6 = 20, which is better for us.

Your boss demands that you write a program to find the minimum possible cutting cost for any given stick.

Input

The input will consist of several input cases. The first line of each test case will contain a positive number l that represents the length of the stick to be cut. You can assume *l* < 1,000. The next line will contain the number n (n < 50) of cuts to be made. The next line consists of n positive numbers ci (0 < ci < l) representing the places where the cuts must be made, given in strictly increasing order.

An input case with l = 0 represents the end of input.

Output

Print the cost of the minimum cost solution to cut each stick in the format shown below.

Sample Input

```
100
3
25 50 75
10
4
4 5 7 8
0
```

Sample Output

```
The minimum cutting is 200.
The minimum cutting is 22.
```

ALCP 46 Ferry Loading

Ferries are used to transport cars across rivers and other bodies of water. Typically, ferries are wide enough to support two lanes of cars throughout their length. The two lanes of cars drive onto the ferry from one end, the ferry crosses the water, and the cars exit from the other end of the ferry.

The cars waiting to board the ferry form a single queue, and the operator directs each car in turn to drive onto the port (left) or starboard (right) lane of the ferry so as to balance the load. Each car in the queue has a different length, which the operator estimates by inspecting the queue. Based on this inspection, the operator decides which side of the ferry each car should board, and boards as many cars as possible from the queue, subject to the length limit of the ferry. Write a program that will tell the operator which car to load on which side so as to maximize the number of cars loaded.

Input

The input begins with a single positive integer on a line by itself indicating the number of test cases, followed by a blank line.

The first line of each test case contains a single integer between 1 and 100: the length of the ferry (in meters). For each car in the queue there is an additional line of input specifying the length of the car in cm, an integer between 100 and 3,000 inclusive. A final line of input contains the integer 0. The cars must be loaded in order, subject to the constraint that the total length of cars on either side does not exceed the length of the ferry. As many cars should be loaded as possible, starting with the first car in the queue and loading cars in order until the next car cannot be loaded.

There is a blank line between each two consecutive inputs.

Output

For each test case, the first line of output should give the number of cars that can be loaded onto the ferry. For each car that can be loaded onto the ferry, in the order the cars appear in the input, output a line containing "port" if the car is to be directed to the port side and "starboard" if the car is to be directed to the starboard side. If several arrangements of cars meet the criteria above, any one will do.

The output of two consecutive cases will be separated by a blank line.

Sample Input

```
1
50
2500
3000
1000
1000
1500
700
800
0
```

Sample Output

```
6
port
starboard
starboard
starboard
port
port
```

ALCP 47 Chopsticks

In China, people use pairs of chopsticks to eat food, but Mr. L is a bit different. He uses a set of three chopsticks, one pair plus an extra; a long chopstick to get large items by stabbing the food. The length of the two shorter, standard chopsticks should be as close as possible, but the length of the extra one is not important so long as it is the longest. For a set of chopsticks with lengths A, B, C ($A \le B \le C$), the function $(A - B)^2$ defines the "badness" of the set.

Mr. L has invited K people to his birthday party, and he is eager to introduce his way of using chopsticks. He must prepare $K + 8$ sets of chopsticks (for himself, his wife, his little son, little daughter, his mother, father, mother-in-law, father-in-law, and K other guests). But Mr. L's chopsticks are of many different lengths! Given these lengths, he must find a way of composing the $K + 8$ sets such that the total badness of the sets is minimized.

Input

The first line in the input contains a single integer T indicating the number of test cases (1 ≤ T ≤ 20). Each test case begins with two integers K and N (0 ≤ K ≤ 1,000, 3K + 24 ≤ N ≤ 5,000) giving the number of guests and the number of chopsticks. Then follow N positive integers Li, in non–decreasing order, indicating the lengths of the chopsticks (1 ≤ Li ≤ 32,000).

Output

For each test case in the input, print a line containing the minimal total badness of all the sets.

Sample Input

```
1

1 40

1 8 10 16 19 22 27 33 36 40 47 52 56 61
63 71 72 75 81 81 84 88 96 98

103 110 113 118 124 128 129 134 134 139
148 157 157 160 162 164
```

Sample Output

```
23
```

Note: A possible collection of the nine chopstick sets for this sample input is

(8, 10, 16), (19, 22, 27), (61, 63, 75), (71, 72, 88), (81, 81, 84), (96, 98, 103), (128, 129, 148),

(134, 134, 139), and (157, 157, 160).

ALCP 48 Adventures in Moving

You are considering renting a moving truck to help you move from Waterloo to the big city. Gas prices being so high these days, you want to know how much the gas for this beast will set you back.

The truck consumes a full liter of gas for each kilometre it travels. It has a 200-liter gas tank. When you rent the truck in Waterloo, the tank is half-full. When you return it in the big city, the tank must be at least half-full, or you'll get gouged even more for gas by the rental company. You would like to spend as little as possible on gas, but you don't want to run out along the way.

Input

The input begins with a single positive integer on a line by itself indicating the number of test cases, followed by a blank line. Each test case is composed only of integers. The first integer is the distance in kilometres from Waterloo to the big city, at most 10,000. Next comes a set of up to 100 gas station specifications, describing all the gas stations along your route, in non-decreasing order by distance. Each specification consists of the distance in kilometres of the gas station from Waterloo, and the price of a litre of gas at the gas station, in tenths of a cent, at most 2,000. There is a blank line between each two consecutive inputs.

Output

For each test case, output the minimum amount of money that you can spend on gas to get from Waterloo to the big city. If it is not possible to get from Waterloo to the big city subject to the constraints above, print "Impossible". The output of each two consecutive cases will be separated by a blank line.

Sample Input

```
1
500
100  999
150  888
200  777
300  999
400  1009
450  1019
500  1399
```

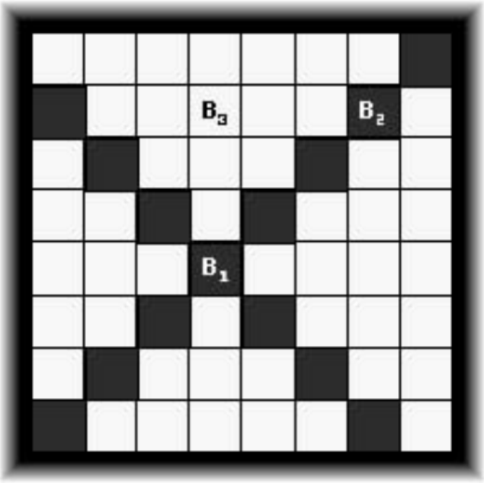
Sample Output

```
450550
```

ALCP 49

Little Bishops

A bishop is a piece used in the game of chess which can only move diagonally from its current position. Two bishops attack each other if one is on the path of the other. In the figure below, the dark squares represent the reachable locations for bishop B1 from its current position. Bishops *B*1 and *B*2 are in attacking position, while *B*1 and *B*3 are not. Bishops *B*2 and *B*3 are also in non-attacking position.



Given two numbers n and k, determine the number of ways one can put k bishops on an n × n chessboard so that no two of them are in attacking positions.

Input

The input file may contain multiple test cases. Each test case occupies a single line in the input file and contains two integers n(1 ≤ n ≤ 8) and k(0 ≤ k ≤ n2).

A test case containing two zeros terminates the input.

Output

For each test case, print a line containing the total number of ways one can put the given number of bishops on a chessboard of the given size so that no two of them lie in attacking positions. You may safely assume that this number will be less than 1015.

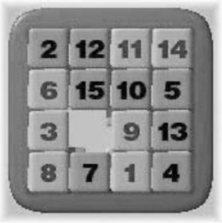
Sample Input	Sample Output
8 6	5599888
4 4	260
0 0	

ALCP 50 | 15-Puzzle Problem

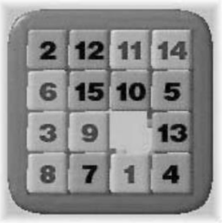
The 15-puzzle is a very popular game: you have certainly seen it even if you don't know it by that name. It is constructed with 15 sliding tiles, each with a different number from 1 to 15, with all tiles packed into a 4 by 4 frame with one tile missing. The object of the puzzle is to arrange the tiles so that they are ordered as below:

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15

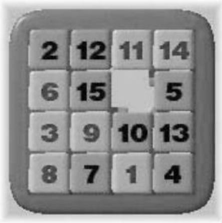
The only legal operation is to exchange the missing tile with one of the 2, 3, or 4 tiles it shares an edge with. Consider the following sequence of moves:



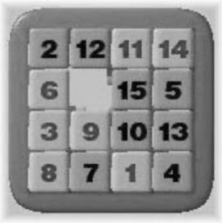
A random puzzle position



The missing tile moves right (R)



The missing tile moves upwards (U)



The missing tile moves left (L)

We denote moves by the neighbor of the missing tile is swapped with it. Legal values are "R," "L," "U," and "D" for right, left, up, and down, based on the movements of the hole. Given an initial configuration of a 15-puzzle you must determine a sequence of steps that take you to the final state. Each solvable 15-puzzle input requires at most 45 steps to be solved with our judge solution; you are limited to using at most 50 steps to solve the puzzle.

Input

The first line of the input contains an integer n indicating the number of puzzle set inputs. The next 4n lines contain n puzzles at four lines per puzzle. Zero denotes the missing tile.

Output

For each input set you must produce one line of output. If the given initial configuration is not solvable, print the line "This puzzle is not solvable." If the puzzle is solvable, then print the move sequence as described above to solve the puzzle.

Sample Input	Sample Output
2	LLLDRDRDR
2 3 4 0	This puzzle is not solvable.
1 5 7 8	

```
9 6 10 12

13 14 11 15

13 1 2 4

5 0 3 7

9 6 10 12

15 8 11 14
```

ALCP 51 Queue

Consider a queue with *N* people, each of a different height. A person can see out to the left of the queue if he or she is taller than all the people to the left; otherwise the view is blocked. Similarly, a person can see to the right if he or she is taller than all the people to the right.

A crime has been committed, where a person to the left of the queue has killed a person to the right of the queue using a boomerang. Exactly P members of the queue had unblocked vision to the left and and exactly *R* members had unblocked vision to the right, thus serving as potential witnesses.

The defense has retained you to determine how many permutations of *N* people have this property for a given *P* and *R*.

Input

The input consists of *T* test cases, with *T* (1 ≤ *T* ≤ 10,000) given on the first line of the input file. Each test case consists of a line containing three integers. The first integer *N* indicates the number of people in a queue (1 ≤ *N* ≤ 13). The second integer corresponds to the number of people who have unblocked vision to their left (*P*). The third integer corresponds to the number of people who have unblocked vision to their right (*R*).

Output

For each test case, print the number of permutations of N people where P people can see out to the left and R people can see out to the right.

Sample Input

```
3
10  4  4
11  3  1
3   1   2
```

Sample Output

```
90720
1026576
1
```

ALCP 52 | Servicing Stations

A company offers personal computers for sale in *N* towns (3 ≤ *N* ≤ 35), denoted by 1, 2,...,*N*. There are direct routes connecting *M* pairs among these towns. The company decides to build servicing stations to ensure that for any town *X*, there will be a station located either in *X* or in some immediately neighboring town of *X*. Write a program to find the minimum number of stations the company has to build.

Input

The input consists of multiple problem descriptions. Every description starts with number of towns N and number of town-pairs M, separated by a space. Each of the next M lines contains a pair of integers representing connected towns, at one pair per line with each pair separated by a space. The input ends with N = 0 and M = 0.

Output

For each input case, print a line reporting the minimum number of servicing stations needed.

Sample Input	Sample Output

8 12
1 2
1 6
1 8
2 3
2 6
3 4
3 5
4 5
4 7
5 6
6 7
6 8
0 0
	2

ALCP 53 — Tug of War

Tug of war is a contest of brute strength, where two teams of people pull in opposite directions on a rope. The team that succeeds in pulling the rope in their direction is declared the winner. A tug of war is being arranged for the office picnic. The picnickers must be fairly divided into two teams. Every person must be on one team or the other, the number of people on the two teams must not differ by more than one, and the total weight of the people on each team should be as nearly equal as possible.

Input

The input begins with a single positive integer on a line by itself indicating the number of test cases following, each described below and followed by a blank line. The first line of each case contains n, the number of people at the picnic. Each of the next n lines gives the weight of a person at the picnic, where each weight is an integer between 1 and 450. There are at most 100 people at the picnic. Finally, there is a blank line between each two consecutive inputs.

Output

For each test case, your output will consist of a single line containing two numbers: the total weight of the people on one team, and the total weight of the people on the other team. If these numbers differ, give the smaller number first. The output of each two consecutive cases will be separated by a blank line.

Sample Input

```
1
3
100
90
200
```

Sample Output

```
190  200
```

ALCP 54 | Garden of Eden

Cellular automata are mathematical idealizations of physical systems in which both space and time are discrete, and the physical quantities take on a finite set of discrete values. A cellular automaton consists of a lattice (or array) of discrete-valued variables.

The state of such automaton is completely specified by the values of the variables at each position in the lattice. Cellular automata evolve in discrete time steps, with the value at each position (cell) being affected by the values of variables at sites in its neighbourhood on the previous time step. For each automaton there is a set of rules that define its evolution.

For most cellular automata there are configurations (states) that are unreachable: no state will produce them by the application of the evolution rules. These states are called Gardens of Eden, because they can only appear as initial states. As an example, consider a trivial set of rules that evolve every cell into 0. For this automaton, any state with non-zero cells is a Garden of Eden.

In general, finding the ancestor of a given state (or the non-existence of such an ancestor) is a very hard, computing-intensive, problem. For the sake of simplicity we will restrict the problem to one-dimensional binary finite cellular automata. In other words, the number of cells is a finite number, the cells are arranged in a linear fashion, and their state will be either "0" or "1." To simplify the problem further, each cell state will depend only on its previous state and that of its immediate left and right neighbours. The actual arrangement of the cells will be along a circle, so that the last cell is a neighbor of the first cell.

Problem definition

Given a circular binary cellular automaton, you must determine whether a given state is a Garden of Eden or a reachable state. The cellular automaton will be described in terms of its evolution rules. For example, the table below shows the evolution rules for the automaton: Cell = XOR(Left, Right).

Left	Cell	Right	New		
$[i-1]$	$[i]$	$[i+1]$	State		
0	0	0	0	$0 * 2^0$	
0	0	1	1	$1 * 2^1$	
0	1	0	0	$0 * 2^2$	
0	1	1	1	$1 * 2^3$	
1	0	0	1	$1 * 2^4$	
1	0	1	0	$0 * 2^5$	
1	1	0	1	$1 * 2^6$	
1	1	1	0	$0 * 2^7$	
				90	= Automaton Identifier

With the restrictions imposed on this problem, there are only 256 different automata. An identifier for each automaton can be generated by taking the new state vector and interpreting it as a binary number, as shown in the table. The example automaton has identifier 90, while the identity automaton (where every state evolves to itself) has identifier 204.

Input

The input will consist of several test cases. Each input case describes a cellular automaton and a state on a single line. The first item on the line will be the identifier of the cellular automaton you must work with. The second item in the line will be a positive integer N (4 ≤ N ≤ 32) indicating the number of cells for this test case. Finally, the third item in the line will be a state represented by a string of exactly N zeros and ones. Your program must keep reading lines until the end of file.

Output

If an input case describes a Garden of Eden, output the string GARDEN OF EDEN. If the input does not describe a Garden of Eden (it is a reachable state) you must output the string REACHABLE. The output for each test case must be on a different line.

Sample Input	Sample Output
0 4 1111	GARDEN OF EDEN
204 5 10101	REACHABLE
255 6 000000	GARDEN OF EDEN
154 16 1000000000000000	GARDEN OF EDEN

ALCP 55	Color Hash

This puzzle consists of two wheels. Both wheels can rotate clockwise and counter clockwise. They contain 21 colored pieces, 10 of which are rounded triangles and 11 of which are separators. The left panel in Figure 8.2 shows the final puzzle position. Note that to perform a one-step rotation you must turn the wheel until you have advanced a triangle and a separator.

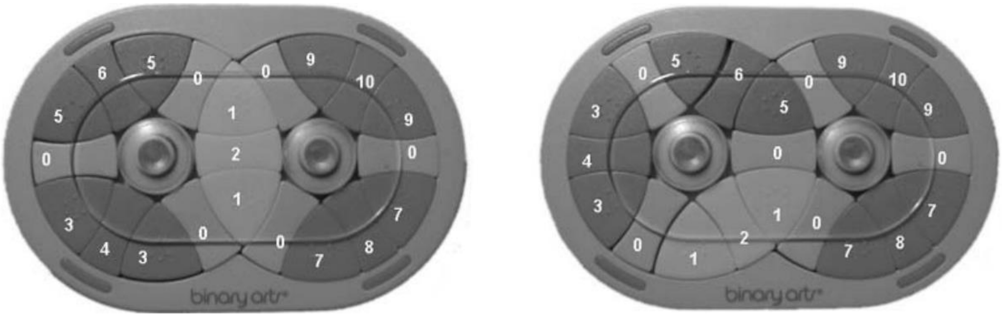


Figure 8.2. Final puzzle configuration (*l*), with the puzzle after rotating the left wheel on step clockwise from the final configuration (*r*).

Your job is to write a program that reads the puzzle configuration and prints the minimum sequence of movements required to reach the final position. We will use the following integer values to encode each type of piece:

0	gray separator
1	yellow triangle
2	yellow separator
3	cyan triangle
4	cyan separator
5	violet triangle
6	violet separator
7	green triangle
8	green separator
9	red triangle
10	red separator

A puzzle configuration will be described using 24 integers; the first 12 describe the left wheel configuration; the last 12, the right wheel. The first integer represents the bottom right separator of the left wheel and the next 11 integers describe the left wheel clockwise. The 13th integer represents the bottom left separator of the right wheel and the next 11 integers describe the right wheel counterclockwise. The final position is therefore encoded

0 3 4 3 0 5 6 5 0 1 2 1 0 7 8 7 0 9 10 9 0 1 2 1

If we rotate the left wheel clockwise one position from the final configuration (as shown in the right-hand figure) the puzzle configuration would be encoded

2 1 0 3 4 3 0 5 6 5 0 1 0 7 8 7 0 9 10 9 0 5 0 1

Input

Input for your program consists of several puzzles. The first line of the input will contain an integer n specifying the number of puzzles. There will then be n lines, each containing 24 integers separated with one white space, describing the initial puzzle configuration as explained above.

Output

For each configuration, your program should output one line with just one number representing the solution. Each movement is encoded using one digit from 1 to 4 in the following way:

1 Left Wheel Clockwise rotation

2 Right Wheel Clockwise rotation

3 Left Wheel Counter clockwise rotation

4 Right Wheel Counter clockwise rotation

No space should be printed between each digit. Since multiple solutions could be found, you should print the solution that is encoded as the smallest number. The solution will never require more than 16 movements.

If no solution is found you should print, “NO SOLUTION WAS FOUND IN 16 STEPS”.

If you are given the final position you should print, “PUZZLE ALREADY SOLVED”.

Sample Input

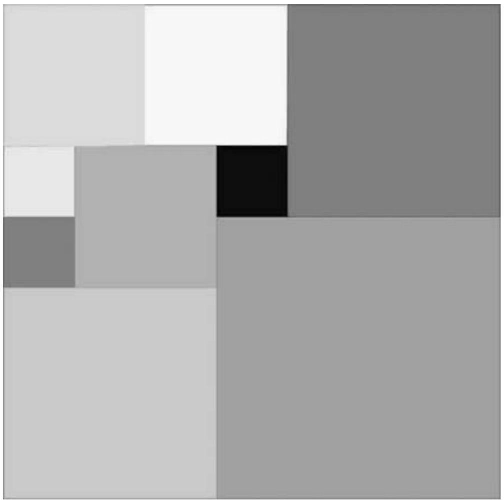
```
3
0 3 4 3 0 5 6 5 0 1 2 1 0 7 8 7 0 9 10 9 0 1 2 1
0 3 4 5 0 3 6 5 0 1 2 1 0 7 8 7 0 9 10 9 0 1 2 1
0 9 4 3 0 5 6 5 0 1 2 1 0 7 8 7 0 9 10 3 0 1 2 1
```

Sample Output

```
PUZZLE ALREADY SOLVED
1434332334332323
NO SOLUTION WAS FOUND IN 16 STEPS
```

ALCP 56 — Bigger Square Please

Tomy has many paper squares. The side length (size) of them ranges from 1 to *N* −1, and he has an unlimited number of squares of each kind. But he really wants to have a bigger one – a square of size *N*. He can make such a square by building it up from the squares he already has. For example, a square of size 7 can be built from nine smaller squares as shown below:



There should be no empty space in the square, no extra paper outside the square, and the small squares should not overlap. Further, Tomy wants to make his square using the minimal number of possible squares. Can you help?

Input

The first line of the input contains a single integer T indicating the number of test cases. Each test case consists of a single integer N, where 2 ≤ N ≤ 50.

Output

For each test case, print a line containing a single integer K indicating the minimal number of squares needed to build the target square. On the following K lines, print three integers x, y, l indicating the coordinates of top-left corner (1 ≤ x, y ≤ N) and the side length of the corresponding square.

Sample Input

```
3
4
3
7
```

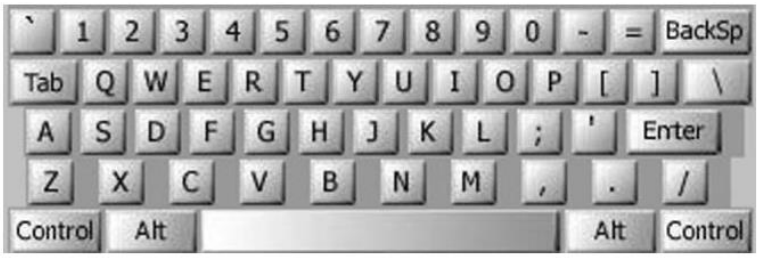
Sample Output

```
4
1 1 2
1 3 2
3 1 2
3 3 2
6
1 1 2
1 3 1
2 3 1
3 1 1
3 2 1
```

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

ALCP 57 WERTYU

A common typing error is to place your hands on the keyboard one row to the right of the correct position. Then “*Q*” is typed as “*W*” and “J” is typed as “*K*” and so on. Your task is to decode a message typed in this manner.


` 1 2 3 4 5 6 7 8 9 0 - = BackSp
Tab Q W E R T Y U I O P [ ] \
A S D F G H J K L ; ' Enter
Z X C V B N M , . /
Control Alt Alt Control

Input

Input consists of several lines of text. Each line may contain digits, spaces, uppercase letters (except ”Q”, “A”, ”Z”), or punctuation shown above [except back-quote (‘)]. Keys labeled with words [Tab, BackSp, Control, etc.] are not represented in the input.

Output

You are to replace each letter or punctuation symbol by the one immediately to its left on the QWERTY keyboard shown above. Spaces in the input should be echoed in the output.

Sample Input	Sample Output
O S, GOMR YPFSU/	I AM FINE TODAY.

ALCP 58	Where's Waldorf?

Given an m by n grid of letters and a list of words, find the location in the grid at which the word can be found. A word matches a straight, uninterrupted line of letters in the grid. A word can match the letters in the grid regardless of case (i.e., upper- and lowercase letters are to be treated as the same). The matching can be done in any of the eight horizontal, vertical, or diagonal directions through the grid.

Input

The input begins with a single positive integer on a line by itself indicating the number of cases, followed by a blank line. There is also a blank line between each two consecutive cases. Each case begins with a pair of integers m followed by n on a single line, where 1<= m, n <=50 in decimal notation. The next m lines contain n letters each, representing the grid of letters where the words must be found. The letters in the grid may be in upper- or lowercase. Following the grid of letters, another integer k appears on a line by itself (1 <=k <=20). The next k lines of input contain the list of words to search for, one word per line. These words may contain upper- and lowercase letters only – no spaces, hyphens, or other non-alphabetic characters.

Output

For each word in each test case, output a pair of integers representing its location in the corresponding grid. The integers must be separated by a single space. The first integer is the line in the grid where the first letter of the given word can be found (1 represents the topmost line in the grid, and m represents the bottommost line). The second integer is the column in the grid where the first letter of the given word can be found (1 represents the leftmost column in the grid, and n represents the rightmost column in the grid). If a word can be found more than once in the grid, then output the location of the uppermost occurrence of the word. If two or more words are uppermost, output the leftmost of these occurrences. All words can be found at least once in the grid. The output of two consecutive cases must be separated by a blank line.

Sample Input

```
1
8 11
abcDEFGhigg
hEbkWalDork
FtyAwaldORm
FtsimrLqsrc
byoArBeDeyv
Klcbqwikomk
strEBGadhrb
yUiqlxcnBjf
4
Waldorf
Bambi
Betty
Dagbert
```

Sample Output

```
2    5
2    3
1    2
7    8
```

ALCP 59 Common Permutation

Given two strings *a* and *b*, print the longest string *x* of letters such that there is a permutation of *x* that is a subsequence of *a* and there is a permutation of *x* that is a subsequence of *b*.

Input

The input file contains several cases, each case consisting of two consecutive lines. This means that lines 1 and 2 are a test case, lines 3 and 4 are another test case, and so on. Each line contains one string of lowercase characters, with first line of a pair denoting a and the second denoting b. Each string consists of at most 1,000 characters.

Output

For each set of input, output a line containing x. If several x satisfy the criteria above, choose the first one in alphabetical order.

Sample Input

```
pretty
women
walking
down
the
street
```

Sample Output

```
e
nw
et
```

ALCP 60 — Crypt Kicker II

A popular but insecure method of encrypting text is to permute the letters of the alphabet. That is, in the text, each letter of the alphabet is consistently replaced by some other letter. To ensure that the encryption is reversible, no two letters are replaced by the same letter.

A powerful method of cryptanalysis is the known plain text attack. In a known plain text attack, the cryptanalyst manages to have a known phrase or sentence encrypted by the enemy, and by observing the encrypted text then deduces the method of encoding. Your task is to decrypt several encrypted lines of text, assuming that each line uses the same set of replacements, and that one of the lines of input is the encrypted form of the plain text the quick brown fox jumps over the lazy dog.

Input

The input begins with a single positive integer on a line by itself indicating the number of test cases, followed by a blank line. There will also be a blank line between each two consecutive cases.

Each case consists of several lines of input, encrypted as described above. The en- crypted lines contain only lowercase letters and spaces and do not exceed 80 characters in length. There are at most 100 input lines.

Output

For each test case, decrypt each line and print it to standard output. If there is more than one possible decryption, any one will do. If decryption is impossible, output

No solution.

The output of each two consecutive cases must be separated by a blank line.

Sample Input

```
1

vtz ud xnm xugm itr pyy jttk gmv xt otgm xt xnm puk ti xnm fprxq
xnm ceuob lrtzv ita hegfd tsmr xnm ypwq ktj
frtjrpgguvj otvxmdxd prm iev prmvx xnmq
```

Sample Output

```
now is the time for all good men to come to the aid of the party
the quick brown fox jumps over the lazy dog
programming contests are fun arent they
```

ALCP 61 Automated Judge Script

Human programming contest judges are known to be very picky. To eliminate the need for them, write an automated judge script to judge submitted solution runs. Your program should take a file containing the correct output as well as the output of submitted program and answer either Accepted, Presentation Error, or Wrong Answer, defined as follows:

Accepted: You are to report "Accepted" if the team's output matches the standard solution exactly. All characters must match and must occur in the same order.

Presentation Error: Give a "Presentation Error" if all numeric characters match in the same order, but there is at least one non-matching non-numeric character. For example, "15 0" and "150" would receive "Presentation Error", whereas "15 0" and "1 0" would receive "Wrong Answer," described below.

Wrong Answer: If the team output cannot be classified as above, then you have no alternative but to score the program a 'Wrong Answer'.

Input

The input will consist of an arbitrary number of input sets. Each input set begins with a line containing a positive integer n < 100, which describes the number of lines of the correct solution. The next n lines contain the correct solution. Then comes a positive integer m < 100, alone on its line, which describes the number of lines of the team's submitted output. The next m lines contain this output. The input is terminated by a value of n = 0, which should not be processed.

No line will have more than 100 characters.

Output

For each set, output one of the following:

Run #x: Accepted

Run #x: Presentation Error Run #x: Wrong Answer

where x stands for the number of the input set (starting from 1).

Sample Input

```
2
The answer is: 10 The answer is: 5 2
The answer is: 10
The answer      is:     5
2
The answer      is:     10
The answer      is:     5
2
The answer      is:     10
The answer      is:     15
2
The answer      is:     10
The answer      is:     5
2
```

Sample Output

```
Run #1: Accepted
Run #2: Wrong Answer
Run #3: Presentation Error Run #4: Wrong Answer
Run #5: Presentation Error Run #6: Presentation Error
```

```
The answer      is:     10
The answer      is:     5
3
Input Set       #1:     YES
Input Set       #2:     NO
Input Set       #3:     NO
3
Input Set       #0:     YES
Input Set       #1:     NO
Input Set       #2:     NO
1
1 0 1 0
1
1010
1
The judges are mean! 1
The judges are good! 0
```

ALCP 62 | File Fragmentation

Your friend, a biochemistry major, tripped while carrying a tray of computer files through the lab. All of the files fell to the ground and broke. Your friend picked up all the file fragments and called you to ask for help putting them back together again.

Fortunately, all of the files on the tray were identical, all of them broke into exactly two fragments, and all of the file fragments were found. Unfortunately, the files didn't all break in the same place, and the fragments were completely mixed up by their fall to the floor.

The original binary fragments have been translated into strings of ASCII 1's and 0's. Your job is to write a program that determines the bit pattern the files contained.

Input

The input begins with a single positive integer on its own line indicating the number of test cases, followed by a blank line. There will also be a blank line between each two consecutive cases.

Each case will consist of a sequence of "file fragments," one per line, terminated by the end-of-file marker or a blank line. Each fragment consists of a string of ASCII 1's and 0's.

Output

For each test case, the output is a single line of ASCII 1's and 0's giving the bit pattern of the original files. If there are 2N fragments in the input, it should be possible to concatenate these fragments together in pairs to make N copies of the output string. If there is no unique solution, any of the possible solutions may be output.

Your friend is certain that there were no more than 144 files on the tray, and that the files were all less than 256 bytes in size.

The output from two consecutive test cases will be separated by a blank line.

Sample Input

```
1
011
0111
01110
111
0111
10111
```

Sample Output

```
01110111
```

ALCP 63 Doublets

A doublet is a pair of words that differ in exactly one letter; for example, “booster” and “rooster” or “rooster” and “roaster” or “roaster” and “roasted”.

You are given a dictionary of up to 25,143 lowercase words, not exceeding 16 letters each. You are then given a number of pairs of words. For each pair of words, find the shortest sequence of words that begins with the first word and ends with the second, such that each pair of adjacent words is a doublet. For example, if you were given the input pair “booster” and “roasted”, a possible solution would be (“booster,” “rooster,” “roaster,” “roasted”), provided that these words are all in the dictionary.

Input

The input file contains the dictionary followed by a number of word pairs. The dictionary consists of a number of words, one per line, and is terminated by an empty line. The pairs of input words follow; each pair of words occurs on a line separated by a space.

Output

For each input pair, print a set of lines starting with the first word and ending with the last. Each pair of adjacent lines must be a doublet.

If there are several minimal solutions, any one will do. If there is no solution, print a line: “No solution.” Leave a blank line between cases.

Sample Input

```
booster
rooster
roaster
coasted
roasted
coastal
postal
booster roasted
coastal postal
```

Sample Output

```
booster
rooster
roaster
roasted
No solution.
```

ALCP 64 — Fmt

The UNIX program fmt reads lines of text, combining and breaking them so as to create an output file with lines as close to 72 characters long as possible without exceeding this limit. The rules for combining and breaking lines are as follows:

- A new line may be started anywhere there is a space in the input. When a new line is started, blanks at the end of the previous line and at the beginning of the new line are eliminated.
- A line break in the input may be eliminated in the output unless (1) it is at the end of a blank or empty line, or (2) it is followed by a space or another line break. When a line break is eliminated, it is replaced by a space.
- Spaces must be removed from the end of each output line.
- Any input word containing more than 72 characters must appear on an output line by itself.

You may assume that the input text does not contain any tabbing characters.

Input

```
Unix fmt

The unix fmt program reads lines of text, combining and breaking lines so as to create an output file
with lines as close to without exceeding 72 characters long as possible. The rules for combining and
breaking lines are as follows.

   1. A new line may be started anywhere there is a space in the input. If a new line is started, there
      will be no trailing blanks at the end of the previous line or at the beginning of the new line.
   2. A line break in the input may be eliminated in the output, provided it is not followed by a
      space or another line break. If a line break is eliminated, it is replaced by a space.
```

Output

```
Unix fmt

The unix fmt program reads lines of text, combining and breaking lines so as to create an output file
with lines as close to without exceeding 72 characters long as possible. The rules for combining and
breaking lines are as follows.

   1. A new line may be started anywhere there is a space in the input. If a new line is started, there
      will be no trailing blanks at the end of the previous line or at the beginning of the new line.
   2. A line break in the input may be eliminated in the output, provided it is not followed by a
      space or another line break. If a line break is eliminated, it is replaced by a space.
```

ALCP 65 — Light, More Light

There is man named Mabu who switches on-off the lights along a corridor at our university. Every bulb has its own toggle switch that changes the state of the light. If the light is off, pressing the switch turns it on. Pressing it again will turn it off. Initially each bulb is off.

He does a peculiar thing. If there are n bulbs in a corridor, he walks along the corridor back and forth *n* times. On the i^{th} walk, he toggles only the switches whose position is divisible by *i*. He does not press any switch when coming back to his initial position. The i^{th} walk is defined as going down the corridor (doing his peculiar thing) and coming back again. Determine the final state of the last bulb. Is it on or off?

Input

The input will be an integer indicating the nth bulb in a corridor, which is less than or equal to 232 – 1. A zero indicates the end of input and should not be processed.

Output

Output “yes” or “no” to indicate if the light is on, with each case appearing on its own line.

Sample Input	**Sample Output**
3	no
6241	yes
8191	no
0	

ALCP 66 Carmichael Numbers

Certain cryptographic algorithms make use of big prime numbers. However, checking whether a big number is prime is not so easy.

Randomized primality tests exist that offer a high degree of confidence of accurate determination at low cost, such as the Fermat test. Let a be a random number between 2 and n 1, where n is the number whose primality we are testing. Then, n is probably prime if the following equation holds:

$$a^n \bmod n = a$$

If a number passes the Fermat test several times, then it is prime with a high probability. Unfortunately, there is bad news. Certain composite numbers (non-primes) still pass the Fermat test with every number smaller than themselves. These numbers are called Carmichael numbers. Write a program to test whether a given integer is a Carmichael number.

Input

The input will consist of a series of lines, each containing a small positive number n (2 < n < 65, 000). A number n = 0 will mark the end of the input, and must not be processed.

Output

For each number in the input, print whether it is a Carmichael number or not as shown in the sample output.

Sample Input

```
1729
17
561
1109
431
0
```

Sample Output

```
The number 1729 is a Carmichael number.
17 is normal.
The number 561 is a Carmichael number.
1109 is normal.
431 is normal.
```

ALCP 67 Euclid Problem

From Euclid, it is known that for any positive integers A and B there exist such integers X and Y that $AX + BY = D$, where D is the greatest common divisor of A and B. The problem is to find the corresponding X, Y, and D for a given A and B.

Input

The input will consist of a set of lines with the integer numbers A and B, separated with space ($A, B <$ 1, 000, 000, 001).

Output

For each input line the output line should consist of three integers X, Y , and D, separated with space. If there are several such X and Y, you should output that pair for which $X \leq Y$ and $|X| + |Y|$ is minimal.

Sample Input	**Sample Output**
4 6	-1 1 2
17 17	0 1 17

ALCP 68 Factovisors

The factorial function, n! is defined as follows for all non-negative integers *n*:

$$0! = 1$$

$$n! = n \times (n - 1)! \qquad (n > 0)$$

We say that a divides b if there exists an integer *k* such that

$$k \times a = b$$

Input

The input to your program consists of several lines, each containing two non-negative integers, n and m, both less than 2^{31}.

Output

For each input line, output a line stating whether or not m divides n!, in the format shown below.

Sample Input

```
6  9
6  27
20  10000
20  100000
1000  1009
```

Sample Output

```
9 divides 6!
27 does not divide 6!
10000 divides 20!
100000 does not divide 20!
1009 does not divide 1000!
```

ALCP 69 Summation of Four Primes

Waring's prime number conjecture states that every odd integer is either prime or the sum of three primes. Goldbach's conjecture is that every even integer is the sum of two primes. Both problems have been open for over 200 years.

In this problem you have a slightly less demanding task. Find a way to express a given integer as the sum of exactly four primes.

Input

Each input case consists of one integer n (n <=10000000) on its own line. Input is terminated by end of file.

Output

For each input case n, print one line of output containing four prime numbers which sum up to n. If the number cannot be expressed as a summation of four prime numbers print the line "Impossible." in a single line. There can be multiple solutions. Any good solution will be accepted.

Sample Input	Sample Output
24	3 11 3 7
36	3 7 13 13
46	11 11 17 7

ALCP 70 — Smith Numbers

While skimming his phone directory in 1982, mathematician Albert Wilansky noticed that the telephone number of his brother-in-law H. Smith had the following peculiar property: The sum of the digits of that number was equal to the sum of the digits of the prime factors of that number. Got it? Smith's telephone number was 493-7775. This number can be written as the product of its prime factors in the following way:

$$4937775 = 3 \cdot 5 \cdot 5 \cdot 65837$$

The sum of all digits of the telephone number is 4 + 9 + 3 + 7 + 7 + 7 + 5 = 42, and the sum of the digits of its prime factors is equally 3 + 5 + 5 + 6 + 5 + 8 + 3 + 7 = 42. Wilansky named this type of number after his brother-in-law: the Smith numbers.

As this property is true for every prime number, Wilansky excluded them from the definition. Other Smith numbers include 6,036 and 9,985. Wilansky was not able to find a Smith number which was larger than the telephone number of his brother-in-law. Can you help him out?

Input

The input consists of several test cases, the number of which you are given in the first line of the input. Each test case consists of one line containing a single positive integer smaller than 10^9.

Output

For every input value n, compute the smallest Smith number which is larger than n and print it on a single line. You can assume that such a number exists.

Sample Input	Sample Output
1 4937774	4937775

ALCP 71 | Marbles

I collect marbles (colorful small glass balls) and want to buy boxes to store them. The boxes come in two types:

Type 1: each such box costs c_1 dollars and can hold exactly n_1 marbles

Type 2: each such box costs c_2 dollars and can hold exactly n_2 marbles

I want each box to be filled to its capacity, and also to minimize the total cost of buying them. Help me find the best way to distribute my marbles among the boxes.

Input

The input file may contain multiple test cases. Each test case begins with a line con- taining the integer n (1 <=n <=2,000,000,000). The second line contains c_1 and n_1, and the third line contains c_2 and n_2. Here, c_1, c_2, n_1, and n_2 are all positive integers having values smaller than 2,000,000,000.

A test case containing a zero for the number of marbles terminates the input.

Output

For each test case in the input print a line containing the minimum cost solution (two nonnegative integers m1 and m2, where mi = number of type i boxes required if one exists. Otherwise print "failed". If a solution exists, you may assume that it is unique.

Sample Input

```
43
1  3
2  4
40
5  9
5  12
0
```

Sample Output

```
13  1
failed
```

ALCP 72 | Repackaging

Coffee cups of three different sizes (size 1, size 2, and size 3) are manufactured by the Association of Cup Makers (ACM) and are sold in various packages. Each type of package is identified by three positive integers (S_1, S_2, S_3), where S_i ($1 <= i <= 3$) denotes the number of size *i* cups included in the package. Unfortunately, there is no package such that $S_1 = S_2 = S_3$.

Market research has discovered there is great demand for packages containing equal numbers of cups of all three sizes. To exploit this opportunity, ACM has decided to unpack the cups from some of the packages in its unlimited stock of unsold products and repack them as packages having equal number of cups of all three sizes. For example, suppose ACM has the following packages in its stock: (1, 2, 3), (1, 11, 5), (9, 4, 3), and (2, 3, 2). Then we can unpack three (1, 2, 3) packages, one (9, 4, 3) package, and two (2, 3, 2) packages and repack the cups to produce sixteen (1, 1, 1) packages. One can even produce eight (2, 2, 2) packages or four (4, 4, 4) packages or two (8, 8, 8) packages or one (16, 16, 16) package, etc. Note that all the unpacked cups must be used to produce the new packages; i.e., no unpacked cup is wasted.

ACM has hired you to write a program to decide whether it is possible to produce packages containing an equal number of all three types of cups using all the cups that can be found by unpacking any combination of existing packages in stock.

Input

The input may contain multiple test cases. Each test case begins with a line containing an integer N (3<= N<=1,000) indicating the number of different types of packages that can be found in the stock. Each of the next N lines contains three positive integers denoting, respectively, the number of size 1, size 2, and size 3 cups in a package. No two packages in a test case will have the same specification. A test case containing a zero for N in the first line terminates the input.

Output

For each test case print a line containing "Yes" if packages can be produced as desired. Print "No" if they cannot be produced.

Sample Input

```
4
1 2 3
1 11 5
9 4 3
2 3 2
4
1 3  3
1 11 5
9 4  3
2 3  2
0
```

Sample Output

```
Yes
No
```

ALCP 73 — Ant on a Chessboard

One day, an ant named Alice came upon an *MxM* chessboard. She wanted to explore all the cells of the board. So she began to walk along the board by peeling off a corner of the board. Alice started at square (1, 1). First, she went up for a step, then a step to the right, and a step downward. After that, she went a step to the right, then two steps upward, and then two grids to the left. In each round, she added one new row and one new column to the corner she had explored.

For example, her first 25 steps went like this, where the numbers in each square denote on which step she visited it.

25	24	23	22	21
10	11	12	13	20
9	8	7	14	19
2	3	6	15	18
1	4	5	16	17

Her 8th step put her on square (2, 3), while her 20th step put her on square (5, 4). Your task is to decide where she was at a given time, assuming the chessboard is large enough to accept all movements.

Input

The input file will contain several lines, each with an integer N denoting the step number where 1 <=N<= 2<=10^9. The file will terminate with a line that contains the number 0.

Output

For each input situation, print a line with two numbers (x, y) denoting the column and the row number, respectively. There must be a single space between them.

Sample Input

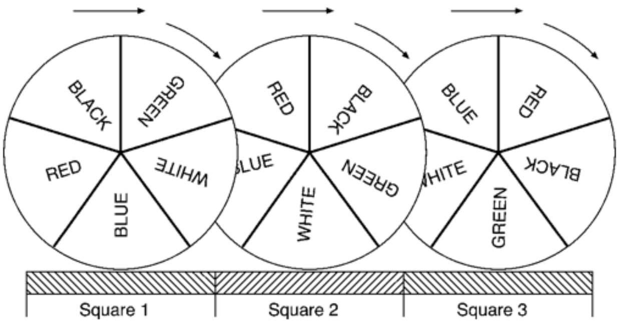
```
8
20
25
0
```

Sample Output

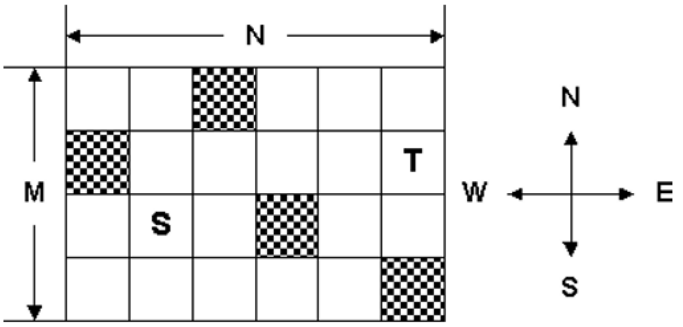
```
2  3
5  4
1  5
```

ALCP 74 The Monocycle

A monocycle is a cycle that runs on one wheel. We will be considering a special one which has a solid wheel colored with five different colors as shown in the figure:



The colored segments make equal angles (72°) at the center. A monocyclist rides this cycle on an *MxN* grid of square tiles. The tiles are of a size such that moving forward from the center of one tile to that of the next one makes the wheel rotate exactly 72° around its center. The effect is shown in the above figure. When the wheel is at the center of square 1, the midpoint of its blue segment is in touch with the ground. But when the wheel moves forward to the center of the next square (square 2) the midpoint of its white segment touches the ground.



Some of the squares of the grid are blocked and hence the cyclist cannot move to them. The cyclist starts from some square and tries to move to a target square in minimum amount of time. From any square he either moves forward to the next square or he remains in the same square but turns 90o left or right. Each of these actions requires exactly 1 second to execute. He always starts his ride facing north and with the midpoint of the green segment of his wheel touching the ground. In the target square, too, the green segment must touch the ground but he does not care which direction he will be facing.

Please help the monocyclist check whether the destination is reachable and if so the minimum amount of time he will require to reach it.

Input

The input may contain multiple test cases.

The first line of each test case contains two integers M and N (1<=M, N<=25) giving the dimensions of the grid. Then follows the description of the grid in M lines of N characters each. The character "#" will indicate a blocked square, but all other squares are free. The starting location of the cyclist is marked by "S" and the target is marked by "T".

The input terminates with two zeros for M and N.

Output

For each test case first print the test case number on a separate line, as shown in the sample output. If the target location can be reached by the cyclist, print the minimum amount of time (in seconds) required to reach it in the format shown below. Otherwise print “destination not reachable”.

Print a blank line between two successive test cases.

Sample Input

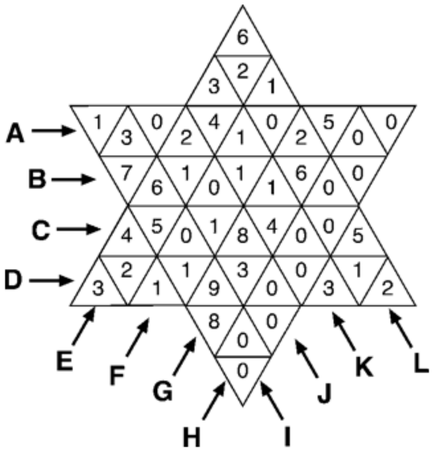
```
1   3
S#T
10   10
#S…….#
#..#.##.##
#.##.##.##
.#.        ##.#
##.##..#.#
#..#.##…
#……….   ##.
..##.##…
#.###…#.
#……….   ###T
0   0
```

Sample Output

```
Case #1
destination not reachable
Case #2
minimum time = 49 sec
```

ALCP 75 Star

A board contains 48 triangular cells. In each cell is written a digit in a range from 0 through 9. Every cell belongs to two or three lines. These lines are marked by letters from *A* through *L*. See the figure below, where the cell containing digit 9 belongs to lines *D, G*, and I and the cell containing digit 7 belongs to lines *B* and *I*.



For each line, we can measure the largest digit on the line. Here the largest digit for line *A* is 5, B is 7, *E* is 6, *H* is 0, and *J* is 8.

Write a program that reads the largest digit for all 12 of the depicted lines and computes the smallest and the largest possible sums of all digits on the board.

Input

Every line in the input contains 12 digits, each separated by a space. The first of these digits describes the largest digit in line A, the second in line B, and so on, until the last digit denotes the largest one in line L.

Output

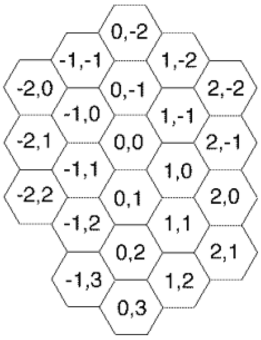
For each input line, print the value of the smallest and largest sums of digits possible for the given board. These two values should appear on the same line and be separated by exactly one space. If there does not exists a solution, your program must output “NO SOLUTION”.

Sample Input	Sample Output
5 7 8 9 6 1 9 0 9 8 4 6	40 172

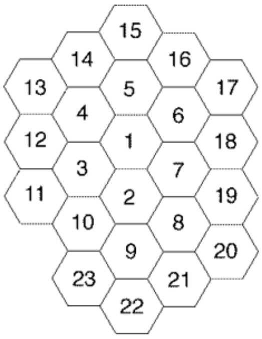
ALCP 76 | Bee Maja

Maja is a bee. She lives in a hive of hexagonal honeycombs with thousands of other bees. But Maja has a problem. Her friend Willi told her where she can meet him, but Willi (a male drone) and Maja (a female worker) have different coordinate systems:

- Maja's Coordinate System — Maja (on left) flies directly to a special honeycomb using an advanced two-dimensional grid over the whole hive.
- Willi's Coordinate System — Willi (on right) is less intelligent, and just walks around cells in clockwise order starting from 1 in the middle of the hive.



Maja's system



Willi's system

Help Maja to convert Willi's system to hers. Write a program which for a given honeycomb number returns the coordinates in Maja's system.

Input

The input file contains one or more integers each standing on its own line. All honeycomb numbers are less than 100,000.

Output

Output the corresponding Maja coordinates for Willi's numbers, with each coordinate pair on a separate line.

Sample Input	Sample Output
1	0 0
2	0 1
3	-1 1
4	-1 0
5	0 -1

ALCP 77 | Robbery

Inspector Robostop is very angry. Last night, a bank was robbed and the robber escaped. As quickly as possible, all roads leading out of the city were blocked, making it impossible for the robber to escape. The inspector then asked everybody in the city to watch out for the robber, but the only messages he got were "We don't see him."

Robostop is determined to discover exactly how the robber escaped. He asks you to write a program which analyzes all the inspector's information to find out where the robber was at any given time. The city in which the bank was robbed has a rectangular shape. All roads leaving the city were blocked for a certain period of time *t*, during which several observations of the form "The robber isn't in the rectangle R_i at time t_i" were reported. Assuming that the robber can move at most one unit per time step, try to find the exact position of the robber at each time step.

Input

The input file describes several robberies. The first line of each description consists of three numbers W , H, and t (1<=W, H, t<=100), where W is the width, H the height of the city, and t is the length of time during which the city is locked.

The next line contains a single integer n (0<=n<=100), where n is the number of messages the inspector received. The next n lines each consist of five integers t_i, L_i, T_i, R_i, B_i, where t_i is the time at which the observation has been made (1<=t_i<=t), and L_i, T_i, R_i, B_i are the left, top, right, and bottom, respectively, of the rectangular area which has been observed. The point (1, 1) is the upper-left-hand corner, and (W, H) is the lower-right-hand corner of the city. The messages mean that the robber was not in the given rectangle at time t_i. The input is terminated by a test case starting with W = H = t = 0. This case should not be processed.

Output

For each robbery, output the line "Robbery #k:", where k is the number of the robbery. Then, there are three possibilities:

If it is impossible that the robber is still in the city, output "The robber has escaped." In all other cases, assume that the robber is still in the city. Output one line of the form "Time step τ: The robber has been at x, y." for each time step in which the exact location can be deduced, and x and y are the column and row, respectively, of the robber in time step τ . Output these lines ordered by time τ. If nothing can be deduced, output the line "Nothing known." and hope that the inspector does not get even angrier.Print a blank line after each processed case.

Sample Input

```
4 4 5
4
1 1 1 4 3
1 1 1 3 4
4 1 1 3 4
4 4 2 4 4
10 10 3
1
2 1 1 10 10
0 0 0
```

Sample Output

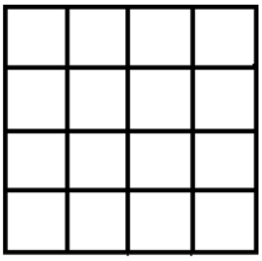
```
Robbery #1:
Time step 1: The robber has been at 4, 4.
Time step 2: The robber has been at 4, 3.
Time step 3: The robber has been at 4, 2.
Time step 4: The robber has been at 4, 1.
Robbery #2:
The robber has escaped.
```

ALCP 78 — (2/3/4)-D Sqr / Rects / Cubes / Boxes?

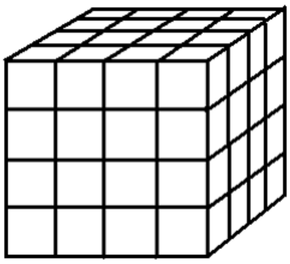
How many squares and rectangles are hidden in the 4x4 grid below? Maybe you can count it by hand for such a small grid, but what about for a 100x100 grid or even larger?

What about higher dimensions? Can you count how many cubes or boxes of different size there are in a 10x10x10 cube, or how many hyper-cubes and hyper-boxes there are in a four-dimensional 5x5x5x5 hypercube?

Your program needs to be efficient, so be clever. You should assume that squares are not rectangles, cubes are not boxes, and hyper-cubes are not hyper-boxes.



A 4 × 4 Grid



A 4 × 4 × 4 Cube

Input

The input contains one integer N (0 <=N<=100) in each line, which is the length of one side of the grid, cube, or hypercube. In the example above N = 4. There may be as many as 100 lines of input.

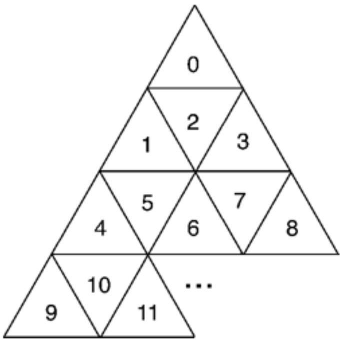
Output

For each line of input, output six integers S_2, R_2, S_3, R_3, S_4, R_4 on a single line, where S_2 denotes the number of squares and R2 the number of rectangles occurring in a two-dimensional (NxN) grid. The integers S_3, R_3, S_4, R_4 denote similar quantities in higher dimensions.

Sample Input	Sample Output
1	1 0 1 0 1 0
2	5 4 9 18 17 64
3	14 22 36 180 98 1198

ALCP 79	Dermuba Triangle

Dermuba Triangle is the universally-famous flat and triangular region in the L-PAX planet in the Geometria galaxy. The people of Dermuba live in equilateral-triangular fields with sides exactly equal to 1 km. Houses are always built at the circumcentres of the triangular fields. Their houses are numbered as shown in the figure below.



When Dermubian people visit each other, they follow the shortest path from their house to the destination house. This shortest path is obviously the straight-line distance that connects these two houses. Now comes your task. You have to write a program which computes the length of the shortest path between two houses given their house numbers.

Input

The input consists of several lines with two non-negative integer values n and m which specify the start and destination house numbers, where 0 ≤ n, m ≤ 2,147,483,647.

Output

For each line in the input, print the shortest distance between the given houses in kilometers rounded off to three decimal places.

Sample Input	Sample Output
0 7	1.528
2 8	1.528
9 10	0.577
10 11	0.577

ALCP 80 | Airlines

A leading airline has hired you to write a program that answers the following query: given lists of city locations and direct flights, what is the minimum distance a passenger needs to fly to get from one given city to another? The city locations are specified by latitude and longitude. To get from a city to another a passenger may take a direct flight if one exists; otherwise he must take a sequence of connecting flights.

Assume that if a passenger takes a direct flight from *X* to *Y* he never flies more than the geographical distance between *X* and *Y*. The geographical distance between two locations *X* and *Y* is the length of the geodetic line segment connecting *X* and *Y*. The geodetic line segment between two points on a sphere is the shortest connecting curve lying entirely on the surface of the sphere. Assume that the Earth is a perfect sphere of radius exactly 6,378 km, and that the value of π is approximately 3.141592653589793. Round the geographical distance between every pair of cities to the nearest integer.

Input

The input may contain multiple test cases. The first line of each test case contains three integers N <=100, M<=300, and Q<=10,000, where *N* indicates the number of cities, *M* represents the number of direct flights, and *Q* is the number of queries.

The next *N* lines contain the list of cities. The i^{th} of these lines contains a string c_i followed by two real numbers lt_i and ln_i, representing the city name, latitude, and longitude, respectively. The city name will be at most 20 characters and will not contain white-space characters. The latitude will be between 90o (South Pole) and +90o (North Pole). The longitude will be between 180o and +180o, where negative (positive) numbers denote locations west (east) of the meridian passing through Greenwich, England.

The next *M* lines contain the direct flight list. The i^{th} of these lines contains two city names a_i and b_i, indicating that there exists a direct flight from city a_i to city b_i. Both city names will occur in the city list.

The next *Q* lines contain the query list. The i^{th} of these lines will contain city names a_i and b_i asking for the minimum distance a passenger needs to fly to get from a_i to b_i. Be assured that a_i and b_i are not equal and both city names will occur in the city list.

The input will terminate with three zeros for *N*, *M*, and *Q*.

Output

For each test case, first output the test case number (starting from 1) as shown in the sample output. Then for each input query, print a line giving the shortest distance (in km) a passenger needs to fly to get from the first city (a_i) to the second one (b_i). If there exists no route form a_i to b_i, just print the line "no route exists".

Print a blank line between two consecutive test cases.

Sample Input

```
3 4 2
Dhaka  23.8500  90.4000
Chittagong  22.2500  91.8333
Calcutta  22.5333  88.3667
Dhaka Calcutta
Calcutta Dhaka
```

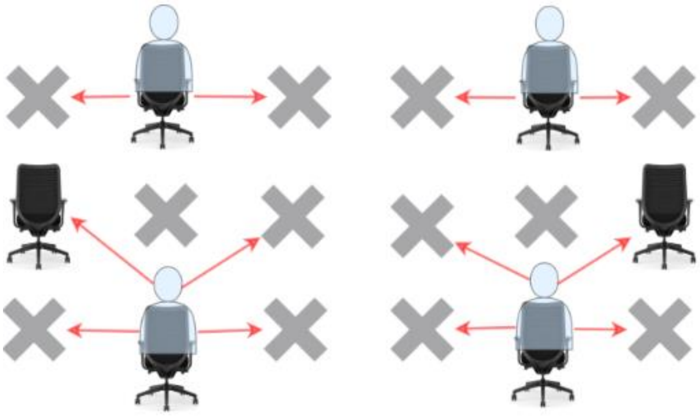
Sample Output

```
Case #1
485 km
231 km
Case #2
19654 km
no route exists 12023 km
```

```
Dhaka Chittagong
Chittagong Dhaka
Chittagong Calcutta
Dhaka Chittagong
5 6 3
Baghdad  33.2333  44.3667
Dhaka  23.8500  90.4000
Frankfurt  50.0330  8.5670
Hong_Kong  21.7500  115.0000
Tokyo  35.6833  139.7333
Baghdad Dhaka
Dhaka Frankfurt
Tokyo Hong_Kong
Hong_Kong Dhaka
Baghdad Tokyo
Frankfurt Tokyo
Dhaka Hong_Kong
Frankfurt Baghdad
Baghdad Frankfurt
0  0  0
```

ALCP 81 | Maximum Students Taking Exam

Given a *m* * *n* matrix seats that represent seats distributions in a classroom. If a seat is broken, it is denoted by '#' character otherwise it is denoted by a '.' character. Students can see the answers of those sitting next to the left, right, upper left and upper right, but he cannot see the answers of the student sitting directly in front or behind him. Return the maximum number of students that can take the exam together without any cheating being possible. Students must be placed in seats in good condition.



Input

A 2D list seats representing the seating arrangement in a classroom. Each seat can either be broken (#) or available for sitting (.).

```
seats = [
    [".",".",".","#",".",".",".","#"],
    [".","#",".",".","#",".",".","."],
    [".",".",".","#",".",".",".","#"]
]
```

Output

The maximum number of students that can sit together such that no student can cheat.

4

Sample Input	**Sample Output**
seats = [["#",".","#","#",".","#"], [".","#","#","#","#","."], ["#",".","#","#",".","#"]]	4
seats = [[".","#"], ["#","#"], ["#","."], ["#","#"], [".","#"]]	3
seats = [["#",".",".",".","#"], [".","#",".","#","."], [".",".","#",".","."], [".","#",".","#","."], ["#",".",".",".","#"]]	10

ALCP 82 | Cracking the Safe

There is a safe protected by a password. The password is a sequence of n digits where each digit can be in the range [0, *k* - 1]. The safe has a peculiar way of checking the password. When you enter in a sequence, it checks the most recent n digits that were entered each time you type a digit.

For example, the correct password is "345" and you enter in "012345":

After typing 0, the most recent 3 digits is "0", which is incorrect.

After typing 1, the most recent 3 digits is "01", which is incorrect.

After typing 2, the most recent 3 digits is "012", which is incorrect.

After typing 3, the most recent 3 digits is "123", which is incorrect.

After typing 4, the most recent 3 digits is "234", which is incorrect.

After typing 5, the most recent 3 digits is "345", which is correct and the safe unlocks.

Return any string of minimum length that will unlock the safe at some point of entering it.

Input

The input consists of two integers:

n: Length of the password ($1 \leqslant n \leqslant 4$).

k: Range of digits ($1 \leqslant k \leqslant 10$).

Output

Print a single string representing the shortest sequence that will guarantee unlocking the safe.

The output should be a valid De Bruijn sequence, covering all possible n-digit combinations from {0, 1, ..., k-1} at least once.

Sample Input	**Sample Output**

n = 1
k = 2
	“01”

n = 2
k = 2
	"00110"

n = 2
k = 3
	"002112010"

ALCP 83 Reconstruct Itinerary

You are given a list of airline tickets where tickets[*i*] = [$from_i$, to_i] represent the departure and the arrival airports of one flight. Reconstruct the itinerary in order and return it.

All of the tickets belong to a man who departs from "*JFK*", thus, the itinerary must begin with "*JFK*". If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string.

For example, the itinerary ["*JFK*", "*LGA*"] has a smaller lexical order than ["*JFK*", "*LGB*"]. You may assume all tickets form at least one valid itinerary. You must use all the tickets once and only once.

Input

The input consists of a list of airline tickets, where each ticket is represented as a pair [from, to]:

n

$from_1$ to_1

$from_2$ to_2

...

$from_n$ to_n

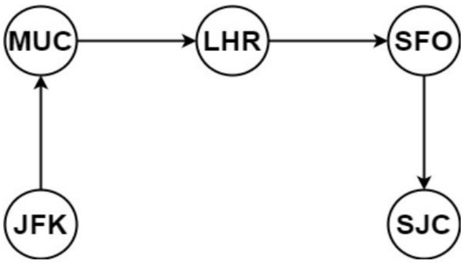
- n ($1 \leqslant n \leqslant 300$) is the number of tickets.

Each airport code is a three-letter uppercase string (e.g., "JFK", "ATL").

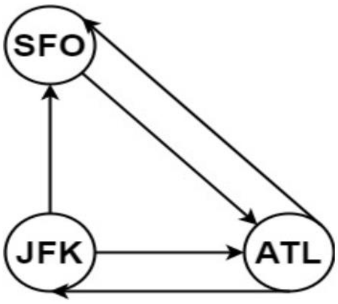
Output

Print the reconstructed itinerary as a space-separated sequence of airport codes.

Sample Input



tickets =
[["MUC","LHR"],["JFK","MUC"],["SFO","SJC"],["LHR","SFO"]]



tickets =
[["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]

Sample Output

["JFK","MUC","LHR","SFO","SJC"]

["JFK","ATL","JFK","SFO","ATL","SFO"]

Another possible reconstruction is ["JFK","SFO","ATL","JFK","ATL","SFO"] but it is larger in lexical order.

ALCP 84 — Shortest Path Visiting All Nodes

You have an undirected, connected graph of *n* nodes labeled from 0 to *n* - 1. You are given an array graph where graph[i] is a list of all the nodes connected with node *i* by an edge.

Return the length of the shortest path that visits every node. You may start and stop at any node, you may revisit nodes multiple times, and you may reuse edges.

Input

The input consists of an integer n (1 ⩽ n ⩽ 12), representing the number of nodes in the graph, followed by n lines.

Each line contains space-separated integers representing the nodes connected to node i.

n

adj_list_0

adj_list_1

...

adj_list_n-1

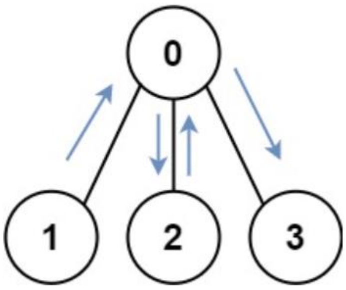
- The graph is undirected and connected.

Each node i (0 ⩽ i < n) is connected to one or more nodes.

Output

A single integer representing the length of the shortest path that visits every node at least once.

Sample Input

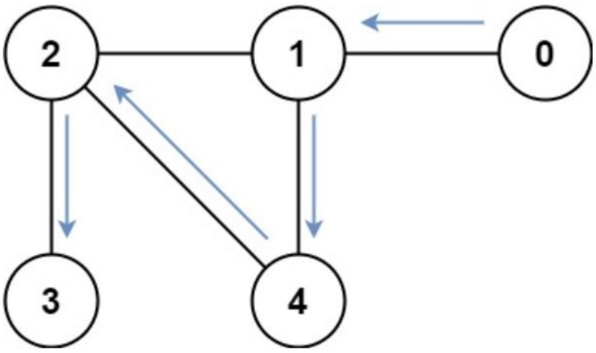


graph = [[1, 2, 3], [0], [0], [0]]

Sample Output

4

One possible path is [1, 0, 2, 0, 3]



graph = [[1], [0, 2, 4], [1, 3, 4], [2], [1, 2]]

4

One possible path is [0, 1, 4, 2, 3]

ALCP 85

The Stable Marriage Problem

Consider a set Y = {m_1, m_2,...,m_n} of n men and a set X ={w_1,w_2,...,w_n} of n women. Each man has a preference list ordering the women as potential marriage partners with no ties allowed. Similarly, each woman has a preference list of the men, also with no ties. Examples of these two sets of lists are given in Figures 10.11a and 10.11b. The same information can also be presented by an $n \times n$ ranking matrix. The rows and columns of the matrix represent the men and women of the two sets, respectively. A cell in row m and column *w* contains two rankings: the first is the position (ranking) of *w* in the m's preference list; the second is the position (ranking) of m in the *w*'s preference list. For example, the pair 3, 1 in Jim's row and Ann's column in the matrix in Figure indicates that Ann is Jim's third choice while Jim is Ann's first. Which of these two ways to represent such information is better depends on the task at hand. For example, it is easier to specify a match of the sets' elements by using the ranking matrix, whereas the preference lists might be a more efficient data structure for implementing a matching algorithm.

A marriage matching *M* is a set of *n*(*m, w*) pairs whose members are selected from disjoint n-element sets *Y* and *X* in a one-one fashion, i.e., each man *m* from *Y* is paired with exactly one woman *w* from *X* and vice versa. (If we represent *Y* and *X* as vertices of a complete bipartite graph with edges connecting possible marriage partners, then a marriage matching is a perfect matching in such a graph.)

men's preferences

	1st	2nd	3rd
Bob:	Lea	Ann	Sue
Jim:	Lea	Sue	Ann
Tom:	Sue	Lea	Ann

women's preferences

	1st	2nd	3rd
Ann:	Jim	Tom	Bob
Lea:	Tom	Bob	Jim
Sue:	Jim	Tom	Bob

ranking matrix

	Ann	Lea	Sue
Bob	2.3	1.2	3.3
Jim	3.1	1.3	2.1
Tom	3.2	2.1	1.2

A pair (*m, w*), where *m* ∈ *Y*, *w* ∈ *X*, is said to be a blocking pair for a marriage matching M if man m and woman *w* are not matched in *M* but they prefer each other to their mates in *M*. For example, (Bob, Lea) is a blocking pair for the marriage matching *M* = {(Bob, Ann), (Jim, Lea), (Tom, Sue)} because they are not matched in *M* while Bob prefers Lea to Ann and Lea prefers Bob to Jim. A marriage matching M is called stable if there is no blocking pair for it; otherwise, *M* is called unstable. According to this definition, the marriage matching is unstable because Bob and Lea can drop their designated mates to join in a union they both prefer. The stable marriage problem is to find a stable marriage matching for men's and women's given preferences. Surprisingly, this problem always has a solution.

Consider an instance of the stable marriage problem given by the following ranking matrix:

	A	B	C
α	1,3	2,2	3,1
β	3,1	1,3	2,2
γ	2,2	3,1	1,3

For each of its marriage matchings, indicate whether it is stable or not. For the unstable matchings, specify a blocking pair. For the stable matchings, indicate whether they are man-optimal, woman-optimal, or neither. (Assume that the Greek and Roman letters denote the men and women, respectively.)

Design a simple algorithm for checking whether a given marriage matching is stable and determine its time efficiency class.

Input

An integer n (1 ⩽ n ⩽ 100) representing the number of men and women.

Two n × n matrices:

The first matrix represents men's preference lists, where row i contains the rankings of all n women for man i.

The second matrix represents women's preference lists, where row j contains the rankings of all n men for woman j.

A list of n pairs (m, w), representing the current matching of men and women.

n

men_pref_matrix (n x n)

women_pref_matrix (n x n)

matching (n pairs)

Output

Print "Stable" if the given matching is stable.

If unstable, print "Unstable" followed by a blocking pair (m, w).

If stable, indicate whether it is "Man-optimal", "Woman-optimal", or "Neither".

Sample Input	**Sample Output**
3 1 3 2 3 1 2 2 2 1 3 1 2 1 3 2 2 1 3 (A, α) (B, β) (C, γ)	Stable Man-optimal
3 1 3 2 3 1 2 2 2 1 3 1 2 1 3 2 2 1 3 (A, β) (B, α) (C, γ)	Unstable Blocking pair: (A, α)

ALCP 86	College Admission Problem

The College Admission Problem (CAP), also referred to as the Residents-Hospitals Assignment Problem, is a generalization of the Stable Marriage Problem (SMP). In CAP, entities on one side of the matching process (colleges or hospitals) can accept multiple participants from the other side (students or residents), unlike SMP, where each participant pairs exclusively with one counterpart. This report explores the problem's definition, mathematical formulation, key properties, algorithmic solutions, and real-world applications.

The College Admission Problem arises in scenarios where one group (e.g., students) must be matched to another group (e.g., colleges) with capacity constraints. Each participant has preferences over whom they wish to be matched with, and the goal is to find a stable matching, where no participant or pair has an incentive to deviate from the proposed allocation. CAP generalizes SMP by accommodating institutions that can accept multiple participants, making it relevant for various allocation problems in education, healthcare, and job markets.

CAP consists of two sets of participants:

Applicants (A): Individuals seeking acceptance (e.g., students or medical residents).

Institutions (I): Entities offering limited slots (e.g., colleges or hospitals).

Each applicant has a strict preference ordering over the institutions they are interested in, and each institution has a strict preference ordering over the applicants. Additionally, institutions have a capacity indicating the maximum number of applicants they can accept.

Stability Condition

A matching is considered stable if:

- There is no applicant and institution who mutually prefer each other over their current matches (blocking pair).
- No applicant or institution prefers being unmatched over their assigned match.

Mathematical Formulation

Given:

- A set of applicants,
- A set of institutions,
- Preference lists for each and for each,
- Capacities for each,

The goal is to find a matching such that:

- Each is matched to at most applicants.
- Each is matched to at most one institution.

How can the College Admission Problem (CAP), which generalizes the Stable Marriage Problem (SMP) by allowing institutions to accept multiple applicants, be defined, analysed, and solved in terms of stability, mathematical formulation, algorithmic approaches, and real-world applications?

Input

n ($1 \leqslant n \leqslant 100$): The number of applicants.

m ($1 \leqslant m \leqslant 100$): The number of institutions (colleges or hospitals).

preference lists:

An array of size n, where each element is a list of m integers representing the preferences of each applicant. The list contains the institution IDs sorted in descending order of preference (first element is the most preferred institution).

An array of size m, where each element is a list of n integers representing the preferences of each institution. The list contains the applicant IDs sorted in descending order of preference (first element is the most preferred applicant).

capacities: A list of size m representing the number of applicants each institution can admit.

matching (optional input, depending on the requirement of the task):

A set of n pairs where each pair (i, j) means applicant i is matched with institution j (if already given). Otherwise, the matching needs to be calculated.

n: Number of applicants

m: Number of institutions

preferences_of_applicants = [list_of_institutions_for_applicant_1, list_of_institutions_for_applicant_2, ..., list_of_institutions_for_applicant_n]

preferences_of_institutions = [list_of_applicants_for_institution_1, list_of_applicants_for_institution_2, ..., list_of_applicants_for_institution_m]

capacities = [capacity_of_institution_1, capacity_of_institution_2, ..., capacity_of_institution_m]

Output

Matching Result: Output the matching between applicants and institutions, ensuring that each institution matches the number of applicants it can admit based on its capacity.

The output should be a list of size m, where each index represents an institution, and the list at that index contains the matched applicants.

Stability Check: Indicate whether the matching is stable:

If stable, print "Stable".

If unstable, print "Unstable" followed by any blocking pair that exists. A blocking pair is an applicant-institution pair where both prefer each other over their current matches.

Matching Result:

Institution 1: [applicant_1, applicant_2, ...]

Institution 2: [applicant_3, applicant_4, ...]

Stability:

Stable

Sample Input

```
n = 3
m = 2
preferences_of_applicants = [
  [0, 1],
  [1, 0],
  [1, 0]
]
preferences_of_institutions = [
  [0, 1, 2],
  [2, 1, 0]
]
```

Sample Output

```
Matching Result:
Institution 0: [applicant_0, applicant_1]
Institution 1: [applicant_2]

Stability:
Stable
```

```
capacities = [2, 1]
```

```
n = 3
m = 2
preferences_of_applicants = [
  [0, 1],
  [1, 0],
  [1, 0]
]
preferences_of_institutions = [
  [0, 1, 2],
  [2, 1, 0]
]
capacities = [1, 2]
```

```
Matching Result:
Institution 0: [applicant_0]
Institution 1: [applicant_2, applicant_1]

Stability:
Stable
```

ALCP 87 Compatible intervals

Given *n* open intervals $(a_1, b_1), (a_2, b_2),\ldots,(a_n, b_n)$ on the real line, each representing start and end times of some activity requiring the same resource, the task is to find the largest number of these intervals so that no two of them overlap. Investigate the three greedy algorithms based on

- Earliest start first.
- Shortest duration first.
- Earliest finish first.

For each of the three algorithms, either prove that the algorithm always yields an optimal solution or give a counterexample showing this not to be the case.

Input

n (1 ⩽ n ⩽ 10^5): The number of intervals.

Intervals: A list of n intervals, where each interval is represented as a tuple (a_i, b_i), indicating the start and end times of the activity.

Each a_i and b_i are integers such that a_i < b_i (i.e., the start time is strictly less than the end time).

The intervals may not be sorted, and the goal is to choose the maximum number of non-overlapping intervals from this list.

n = 5

intervals = [(1, 3), (2, 4), (3, 5), (1, 2), (5, 6)]

Output

Maximum Number of Compatible Intervals: The maximum number of non-overlapping intervals that can be selected from the list.

List of Selected Intervals: The intervals that are chosen based on the selected greedy algorithm.

Algorithm Evaluation: For each algorithm (Earliest Start First, Shortest Duration First, Earliest Finish First), determine if the selected intervals are optimal or provide a counterexample where the algorithm does not work optimally.

Sample Input

n = 5

intervals = [(1, 3), (2, 4), (3, 5), (1, 2), (5, 6)]

Sample Output

Earliest Start First Algorithm

Maximum Compatible Intervals: 3

Selected Intervals: [(1, 2), (3, 5), (5, 6)]

Evaluation:

Not Optimal

Counterexample: The interval (2, 4) could have been chosen, resulting in 4 intervals, not 3.

Shortest Duration First Algorithm

Maximum Compatible Intervals: 3

Selected Intervals: [(1, 2), (2, 4), (5, 6)]

Evaluation:

Not Optimal

	Counterexample: Choosing (1, 2), (3, 5), and (5, 6) would result in a larger number of compatible intervals. **Earliest Finish First Algorithm** Maximum Compatible Intervals: 3 Selected Intervals: [(1, 2), (3, 5), (5, 6)] Evaluation: Optimal The algorithm always yields an optimal solution because by selecting the interval that finishes the earliest, we maximize the available space for future intervals.

ALCP 88 — Bridge Crossing Revisited

Consider the generalization of the bridge crossing puzzle in which we have $n>1$ people whose bridge crossing times are $t_1, t_2,..., t_n$. All the other conditions of the problem remain the same: at most two people at a time can cross the bridge (and they move with the speed of the slower of the two) and they must carry with them the only flashlight the group has.

Design a greedy algorithm for this problem and find how long it will take to cross the bridge by using this algorithm. Does your algorithm yield a minimum crossing time for every instance of the problem? If it does—prove it; if it does not—find an instance with the smallest number of people for which this happens.

Input

n ($2 \leqslant n \leqslant 10^5$): The number of people.

t (t1, t2, ..., tn) ($1 \leqslant t_i \leqslant 10^5$): A list of n integers representing the times it takes each person to cross the bridge.

Each t_i represents the time it takes for the i^{th} person to cross the bridge.

The flashlight can only be carried by one or two people at a time.

Output

Minimum Time: The minimum total time required for all people to cross the bridge using the given greedy algorithm.

Detailed Explanation: If possible, provide a step-by-step description of how the crossing happens, i.e., which people cross together and how the flashlight is passed.

Sample Input	Sample Output

n = 4
t = [1, 2, 5, 10]
	Minimum Time: 17

n = 3
t = [2, 3, 4]
	Minimum Time: 10

ALCP 89 | Huffman Code

Construct a Huffman code for the following data:

symbol	A	B	C	D	-
frequency	0.4	0.1	0.2	0.15	0.15

- Encode ABACABAD using the above code.
- Decode 100010111001010 using the above code.

For data transmission purposes, it is often desirable to have a code with a minimum variance of the code word lengths (among codes of the same average length). Compute the average and variance of the code word length in two Huffman codes that result from a different tie breaking during a Huffman code construction for the following data:

symbol	A	B	C	D	E
probability	0.1	0.1	0.2	0.2	0.4

Indicate whether each of the following properties is true for every Huffman code.

- The code words of the two least frequent symbols have the same length.
- The code word’s length of a more frequent symbol is always smaller than or equal to the code word’s length of a less frequent one.

Input

A string S (the data) consisting of uppercase English letters (e.g., "ABACABAD").

The string may contain multiple occurrences of characters, representing the frequency of each character's occurrence.

Output

Huffman Code: The final Huffman code constructed for the string.

Encoded String: The encoded version of the string using the generated Huffman code.

Decoded String: The decoded version of a given binary string using the Huffman code.

Average Code Length: The average length of the code words used in the Huffman code.

Variance of Code Lengths: The variance in the code word lengths used in the Huffman code.

Properties Check: Check whether the following properties hold:

"The code words of the two least frequent symbols have the same length."

"The code word’s length of a more frequent symbol is always smaller than or equal to the code word’s length of a less frequent one."

Sample Input

```
S = "ABACABAD"
```

Sample Output

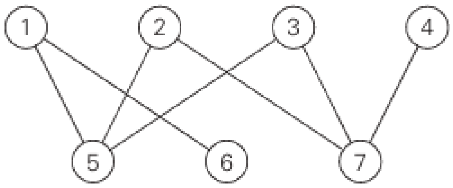
```
Huffman Code:
A -> 0
B -> 11
C -> 101
D -> 100

Encoded String: 010110010111100
```

	Decoded String: ABACABAD Average Code Length: 2.25 Variance of Code Lengths: 0.1875 Properties Check: 1. The code words of the two least frequent symbols have the same length: False 2. The code word’s length of a more frequent symbol is always smaller than or equal to the code word’s length of a less frequent one: True
S = "AABBBCCCC"	Huffman Code: A -> 0 B -> 11 C -> 10 Encoded String: 001111100000 Decoded String: AABBBCCCC Average Code Length: 2.0 Variance of Code Lengths: 0.0 Properties Check: 1. The code words of the two least frequent symbols have the same length: True 2. The code word’s length of a more frequent symbol is always smaller than or equal to the code word’s length of a less frequent one: True

ALCP 90	Hall’s Marriage Theorem

Hall’s Marriage Theorem asserts that a bipartite graph *G = (V, U, E)* has a matching that matches all vertices of the set *V* if and only if for each subset *S* ⊆*V*, |*R*(*S*)|≥|*S*| where *R*(*S)* is the set of all vertices adjacent to a vertex in *S*. Check this property for the following graph with (i) V ={1, 2, 3, 4}and (ii) *V* ={5, 6, 7}.



You have to devise an algorithm that returns yes if there is a matching in a bipartite graph G = (*V*, *U*, *E*) that matches all vertices in *V* and returns no otherwise. Would you base your algorithm on checking the condition of Hall’s Marriage Theorem?

Input

V (Set of vertices on one side of the bipartite graph) - A list of integers representing the vertices in set V (e.g., V = {1, 2, 3, 4}).

U (Set of vertices on the other side of the bipartite graph) - A list of integers representing the vertices in set U (e.g., U = {A, B, C, D}).

E (Set of edges) - A list of pairs of integers, where each pair represents an edge between a vertex in V and a vertex in U (e.g., E = [(1, A), (2, B), (3, C), (4, D)]).

The problem asks to check whether there exists a matching that includes all vertices in set V based on Hall's Marriage Theorem.

Output

Result: Output "YES" if there is a matching that matches all vertices in V in the bipartite graph G. Output "NO" if no such matching exists.

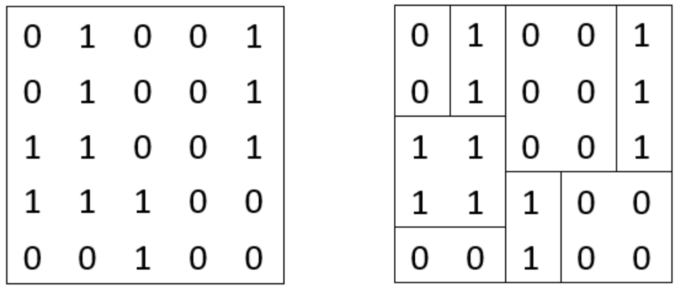
Sample Input	**Sample Output**

V = {1, 2, 3, 4}
U = {A, B, C, D}
E = [(1, A), (2, B), (3, C), (4, D)]
	YES

V = {1, 2, 3}
U = {A, B, C, D}
E = [(1, A), (2, B), (3, D)]
	NO

ALCP 91 | Guillotine Decompositions

You are given an $n \times n$ matrix A whose entries are either 0 or 1. We want to decompose this matrix into rectangular submatrices such that the entries in any submatrix are either all 0 or all 1. We limit the decompositions to a special type called guillotine decompositions. A guillotine decomposition of a subarray $A[i_1..i_2, j_1..j_2]$ is defined recursively, as follows. If $i_1 = i_2$ and $j_1 = j_2$, then no further decomposition is possible. If $i_1 < i_2$ then select any i, $i_1 \leq i < i_2$ and split the array horizontally just after row i, creating two new subarrays $A[i_1..i, j_1..j_2]$ and $A[(i + 1)..i_2, j_1..j_2]$. If $j_1 < j_2$ then select any j, $j_1 \leq j < j_2$ and split the array vertically just after column j, creating two new subarrays $A[i_1..i_2, j_1..j]$ and $A[i1..i_2, (j + 1)..j_2]$. If both $i_1 < i_2$ and $j_1 < j_2$ then you are free to make either type of cut. Once the two submatrices are formed, they may be further decomposed by a guillotine decomposition. An example of a guillotine decomposition into submatrices of one value is shown below.



$$\begin{array}{ccccc} 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array} \qquad \begin{array}{c|c|cc|c} 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ \hline 1 & 1 & 0 & 0 & 1 \\ \cline{3-5} 1 & 1 & 1 & 0 & 0 \\ \cline{1-2} 0 & 0 & 1 & 0 & 0 \end{array}$$

Write a dynamic programming algorithm to determine the minimum number of guillotine cuts needed to decompose the matrix A into sub- matrices of one value. Justify the correctness of your algorithm, and derive its running time. (Note: You do not have to output the actual cuts themselves, just the number of cuts.)

Input

n: An integer representing the size of the matrix. This will be an $n \times n$ matrix.

A: An $n \times n$ matrix where each element $[i][j]$ is either 0 or 1. This matrix represents the grid of values that we need to decompose into submatrices.

Output

Minimum Cuts: An integer representing the minimum number of guillotine cuts needed to decompose the matrix into submatrices, where each submatrix contains either all 0's or all 1's.

Sample Input	Sample Output

n = 3
A = [[0, 1, 0],
[1, 1, 0],
[0, 1, 0]]
	2

n = 4 A = [[1, 0, 1, 0], [0, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 1]]	3

ALCP 92 Printing a Paragraph

Consider the problem of neatly printing a paragraph on a printer. The input text is a sequence of n words of lengths l_1, l_2,... , l_n, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of *M* characters each. Our criterion for "neatness" is as follows. There MUST be at least one space between any two words on the same line. Except for the last line, if a given line contains words *i* through *j*, word *i* starts in the leftmost position of the line, and word *j* ends in the rightmost position of the line, and space is inserted between the words as evenly as possible. The amount of extra space between any two words on this line is

$$\frac{M - \sum_{k=i}^{j} l_k}{j - i} - 1,$$

and the total cost of the line is the cube of this value multiplied times (*j* − *i*). For the last line, we simply put one space between each word, starting at the leftmost position of the line. We wish to minimize the sum of costs of all lines (except the last). Give a dynamic programming algorithm to determine the neatest way to print the words. It suffices to determine the minimum cost, not the actual printing method. You may assume that every word has length less than *M*/2, so that there are at least two words on each line. Analyze the running time and space of your algorithm.

Input

n: An integer representing the number of words in the paragraph.

l[]: A list of integers where each integer l_i represents the length of the i^{th} word in the paragraph.

M: An integer representing the maximum number of characters that a line can hold.

Output

Minimum Cost: An integer representing the minimum cost to print the paragraph neatly, according to the given criteria. The cost is the sum of the cubic extra space for all lines (except the last).

Sample Input	Sample Output

n = 4
l = [3, 2, 2, 5]
M = 6
	4

n = 5
l = [3, 2, 2, 5, 1]
M = 6
	5

ALCP 93

Treasure Trove

While walking on the beach one day you find a treasure trove. Inside there are n treasures with weights $w_1,\ldots, w_n$ and values $v_1,\ldots, v_n$. Unfortunately, you have a knapsack that only holds a total weight M. Fortunately there is a knife handy so that you can cut treasures if necessary; a cut treasure retains its fractional value (so, for example, a third of treasure i has weight $w_i/3$ and value $v_i/3$).

- Describe a $\Theta(n \lg n)$ time greedy algorithm to solve this problem.
- Prove that your algorithm works correctly.
- Improve the running time of your algorithm to $\Theta(n)$.

Input

n: An integer representing the number of treasures.

w[]: A list of integers, where w[i]] is the weight of the ith treasure.

v[]: A list of integers, where v[i]] is the value of the ith treasure.

M: An integer representing the maximum weight capacity of the knapsack.

Output

Maximized Value: A floating-point number representing the maximum value achievable by fitting treasures (and possibly fractional parts of them) into the knapsack of weight M.

Sample Input	**Sample Output**

4
10 20 30 40
60 100 120 80
50
	240.0

3
5 10 15
50 80 90
10
	130.0

5
10 20 30 40 50
100 200 300 400 500
60
	900.0

ALCP 94 — Recompute the MST

Suppose that your boss gives you a connected undirected graph *G*, and you compute a minimum cost spanning tree *T*. Now he tells you that he wants to change the cost of an edge. Your job is to recompute the new minimum spanning tree (say *T*'). A good algorithm should recompute the MST by considering only those edges that might change.

For each of the following situations, indicate whether 1) the MST cannot change 2) the MST can possibly change. If case 2) occurs then describe what changes might occur in the MST. State which edges might possibly be deleted from the MST, and WHICH edges might possibly be inserted into the MST, and how you would select the edges to be inserted/deleted. (For extra credit: give a proof for your scheme.)

- He decreases the cost of an edge e that belongs to *T*.
- He increases the cost of an edge e that belongs to *T*.
- He decreases the cost of an edge f that does not belong to *T*.
- He increases the cost of an edge f that does not belong to *T*.

You do not need to design an algorithm, but to simply state the edges that need to be considered.

Input

The first line contains an integer VVV, the number of vertices in the graph.

The second line contains an integer EEE, the number of edges in the graph.

The following EEE lines each contain three integers u, v, w where:

- u and v are the vertices connected by the edge,
- w is the weight (cost) of the edge.

After the edges, we have a list of changes:

- The number of changes.
- For each change, a line will contain either:
  - An edge that belongs to the current MST and its new weight (either increased or decreased),

Or an edge that does not belong to the current MST and its new weight (either increased or decreased).

Output

For each change:

Indicate if the MST can possibly change by either:

The MST cannot change.

The MST can possibly change. In this case:

Indicate which edges might possibly be deleted from the MST.

Indicate which edges might possibly be inserted into the MST.

If necessary, describe how to select the edges to be inserted/deleted.

Sample Input

```
4
5
1 2 3
```

Sample Output

```
The MST can possibly change.
- Edges to consider:
  - Deleted edges: (1, 3)
```

2 3 1 3 4 2 1 3 4 2 4 5 3 1 3 5 (Increase cost of edge (1, 3)) 2 4 2 (Decrease cost of edge (2, 4)) 1 4 1 (Decrease cost of edge (1, 4))	- Inserted edges: (1, 4) (since its new cost is now less than (1, 3)) The MST can possibly change. - Edges to consider: - Inserted edges: (2, 4) - Deleted edges: (3, 4) The MST cannot change.

ALCP 95 — Graph Coloring Problem

A coloring of an undirected graph is assignment of colors to the nodes so that no two neighboring nodes have the same color. In the graph coloring problem, we desire to find the minimum number of colors needed to color an undirected graph. One possible greedy algorithm for this problem is to start with an arbitrary node and color it "1". Then to consider every other node one at a time and color it "1" if it does not neighbor a node already colored "1". Now do the same for all of the remaining nodes using the color "2". Continue in this fashion until all of the nodes in the graph are colored.

- How would you implement this algorithm efficiently? Be brief but clear. What is your running time as a function of the number of vertices $|V|$, edges $|E|$, and colors $|C|$.
- Prove that for every graph, the algorithm might get lucky and color it with as few colors as possible.
- Prove that the algorithm can be arbitrarily bad. In other words, for every constant $\alpha > 0$ the ratio between the number of colors used by the greedy algorithm and by the optimal algorithm could be at least α.

Input

The first line contains an integer V, the number of vertices in the graph.

The second line contains an integer E, the number of edges in the graph.

The next E lines each contain two integers u and v, representing an undirected edge between vertices u and v.

Output

Output a single integer: the minimum number of colors needed to color the graph using the greedy algorithm described.

Sample Input	**Sample Output**

4
4
1 2
2 3
3 4
4 1
	2

5
6
1 2
1 3
2 4
2 5
3 5
4 5
	3

6	3

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

3                                   3
3
1 2
2 3
1 3

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

ALCP 96 — The (k, n)-Egg Problem

Let k, $n \in N$. The (k, n)-egg problem is as follows: There is a building with n floors. You are told that there is a floor f such that if an egg is dropped off the f^{th} floor then it will break, but if it is dropped off the $(f - 1)^{st}$ floor it will not. (It is possible that $f = 1$ so the egg always breaks, or $f = n + 1$ in which case the egg never breaks.) If an egg breaks when dropped from some floor then it also breaks if dropped from a higher floor. If an egg does not break when dropped from some floor then it also does not break if dropped from a lower floor. Your goal is, given k eggs, to find f. The only operation you can perform is to drop an egg off some floor and see what happens. If an egg breaks then it cannot be reused. You would like to drop eggs as few times as possible. Let $E(k, n)$ be the minimal number of egg-droppings that will always suffice.

- Show that $E(1, n) = n$.
- Show that $E(k, n) = \Theta(n^{1/k})$.
- Find a recurrence for $E(k, n)$. Write a dynamic program to find
- $E(k, n)$. How fast does it run?
- Write a dynamic program that can be used to actually yield the optimal strategy for finding the floor f. How fast does it run?

Input

The first line contains two integers k and n, where:

k represents the number of eggs available.

n represents the number of floors in the building.

Output

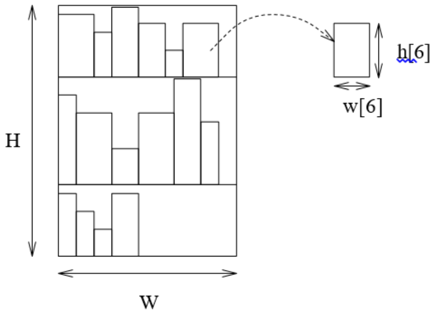
A single integer representing $E(k, n)$, the minimum number of egg drops required to always find the critical floor f in the worst case.

Sample Input	Sample Output
1 10	10
2 10	4
3 100	9
2 36	8
3 500	14
2 1	1
1 50	50
4 1000	19

ALCP 97 The Book-Shelver's Problem

We are given a collection of n books, which must be placed in a library book case. Let *h*[*1..n*] and *w*[1..*n*] be arrays, where *h*[*i*] denotes the height of the *i*th book and *w*[*i*] denotes the width of the ith book. Assume that these arrays are given by the books' catalogue numbers (and the books MUST be shelved in this order), and that books are stacked in the usual upright manner (i.e., you CANNOT lay a book on its side). All book cases have width W, but you may have any height you like. The books are placed on shelves in the book case. You may use as many shelves as you like, placed wherever you like (up to the height of the book case), and you may put as many books on each shelf as you like (up to the width of the book case). Assume for simplicity that shelves take up no vertical height.

The book-shelver's problem is, given *h*[1..*n*], *w*[1..*n*], and *W* , what is the minimum height book case that can shelve all of these books. Below shows an example of a shelving. The height and width of the 6th book are shown to the right. (Notice that there is some unnecessarily wasted height on the third shelf.)



- Write down a recurrence for book shelver's problem.
- Use your recurrence to develop an efficient dynamic programming for solving the book shelver's problem.
- Analyze the running time of your algorithm.

Input

The first line contains two integers n and W, where:

- n is the number of books.
- W is the total width of the bookcase (shelf width).

The next n lines each contain two integers h[i] and w[i], representing the height and width of the ith book.

Output

A single integer representing the minimum height required to shelve all books optimally.

Sample Input	Sample Output

5 10
3 5
2 4
4 6
1 3
5 2
	12

6 8	12

4 2	
2 3	
5 4	
3 2	
6 6	
1 1	
1 5	2
2 4	
4 15	5
3 4	
2 3	
5 5	
1 2	
3 3	6
2 3	
3 3	
1 3	

ALCP 98 — Finding the Longest Monotonically Increasing Subsequence in an Array

Given an array $a[1,\ldots, n]$ of integers, consider the problem of finding the longest monotonically increasing subsequence of not necessarily contiguous entries in your array. For example, if the entries are 10, 3, 9, 5, 8, 13, 11, 14, then a longest monotonically increasing subsequence is 3, 5, 8, 11, 14. The goal of this problem is to find an efficient dynamic programming algorithm for solving this problem.

HINT: Let $L[i]$ be the length of the longest monotonically increasing sequence that ends at and uses element $a[i]$.

- Write down a recurrence for $L[i]$.
- Based on your recurrence, give a recursive program for finding the length of the longest monotonically increasing subsequence. What can you say about its running time?
- Based on your recurrence, give a $\Theta(n^2)$ dynamic programming algorithm for finding the length of the longest monotonically in- creasing subsequence.
- Modify your algorithm to actually find the longest increasing sub- sequence (not just its length).
- Improve your algorithm so that it finds the length of the longest monotonically increasing subsequence in $\Theta(n \lg n)$ time.

Input

The first line contains an integer n (the length of the array).

The second line contains n space-separated integers representing the elements of the array.

Output

The first line contains an integer L (the length of the longest increasing subsequence).

The second line contains the longest increasing subsequence itself.

Sample Input	**Sample Output**
8	5
10 22 9 33 21 50 41 60	10 22 33 50 60
6	3
5 8 3 7 9 1	3 7 9
5	5
1 2 3 4 5	1 2 3 4 5
5	1
5 4 3 2 1	5
6	1
7 7 7 7 7 7	7
10	6
3 10 2 1 20 4 5 6 7 8	3 4 5 6 7 8

ALCP 99

Championship Retention Probability in Chess Matches

The traditional world chess championship is a match of 24 games. The current champion retains the title in case the match is a tie. Not only are there wins and losses, but some games end in a draw (tie). Wins count as 1, losses as 0, and draws as 1/2. The players take turns playing white and black. White has an advantage, because he moves first. Assume the champion is white in the first game, has probabilities w_w, w_d, and w_l of winning, drawing, and losing playing white, and has probabilities b_w, b_d, and b_l of winning, drawing, and losing playing black.

- Write down a recurrence for the probability that the champion retains the title. Assume that there are g games left to play in the match and that the champion needs to win i games (which may end in a 1/2).
- Based on your recurrence, give a dynamic programming to calculate the champion's probability of retaining the title.
- Analyse its running time assuming that the match has n games.

Input

The first line contains an integer n (total games in the match).

The second line contains three floating-point numbers: ww, wd, wl (probabilities of winning, drawing, and losing when the champion plays white).

The third line contains three floating-point numbers: bw, bd, bl (probabilities of winning, drawing, and losing when the champion plays black).

Output

A single floating-point number representing the probability that the champion retains the title.

Sample Input	Sample Output

24
0.4 0.3 0.3
0.35 0.4 0.25
	0.68

24
0.5 0.3 0.2
0.4 0.4 0.2
	0.75

24
0.9 0.05 0.05
0.85 0.1 0.05
	0.99

24
0.2 0.3 0.5
0.15 0.4 0.45
	0.30

ALCP 100 — Heap vs. Sorted Array: Analyzing Efficiency for Key Operations

You wish to store a set of n numbers in either a heap or a sorted array. (Heap is MAX-heap, has MAX element on top, etc.) For each of the following desired properties state whether you are better off using a heap or an ordered array, or that it does not matter. Justify your answers.

- Want to find the MAX quickly.
- Want to be able to delete an element quickly.
- Want to be able to form the structure quickly.
- Want to find the MIN quickly.

Input

A single integer n, representing the number of elements.

Output

Four lines, each containing either "Heap", "Sorted Array", or "Does not matter", based on the best data structure for each operation.

Sample Input	Sample Output
10	Heap Heap Heap Sorted Array
100000	Heap Heap Heap Sorted Array

ALCP 101	Cheapest Game

A game-board has *n* columns, each consisting of a top number, the cost of visiting the column, and a bottom number, the maximum number of columns you are allowed to jump to the right. The top number can be any positive integer, while the bottom number is either 1, 2, or 3. The objective is to travel from the first column off the board, to the right of the nth column. The cost of a game is the sum of the costs of the visited columns.

Assuming the board is represented in a two dimensional array, *B*[2,*n*], the following recursive procedure computes the cost of the cheapest game:

```
int CHEAPEST(int i)
  if i>n then return 0 endif;
  x=B[1,i] + CHEAPEST(i+1);
  y=B[1,i] + CHEAPEST(i+2);
  z=B[1,i] + CHEAPEST(i+3);
  case B[2,i] = 1: return x;
       B[2,i] = 2: return min{x,y};
       B[2,i] = 3: return min{x,y,z};
  endcase
```

- Analyze the asymptotic running time of the procedure.
- Describe and analyze a more efficient algorithm for finding the cheapest game.

Input

An integer n representing the number of columns in the game board.

A 2D array $B[2, n]$, where:

B $[0][i]$ (top number) is the cost of visiting column i.

B $[1][i]$ (bottom number) is the maximum jump allowed from column i.

Output

A single integer representing the minimum cost required to exit the board.

Sample Input	Sample Output

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

4
5 6 2 3
2 1 3 1
	7

ALCP 102 Key-Cost Collection Operations

Consider a collection of items, each consisting of a key and a cost. The keys come from a totally ordered universe and the costs are real numbers. Show how to maintain a collection of items under the following operations:

- ADD(k,c): assuming no item in the collection has key k yet, add an item with key k and cost c to the collection;
- REMOVE(k): remove the item with key k from the collection;
- MAX(k_1,k_2): assuming $k1 \leq k2$, report the maximum cost among all items with keys $k \in [k1, k2]$.
- COUNT($c1$,$c2$): assuming $c1 \leq c2$, report the number of items with cost $c \in [c1,c2]$;

Each operation should take at most O (logn) time in the worst case, where n is the number of items in the collection when the operation is performed.

Input

The first line contains an integer Q, the number of queries.

Each of the next Q lines represents an operation:

ADD k c: Adds an item with key k and cost c.

REMOVE k: Removes the item with key k.

MAX k1 k2: Returns the maximum cost among items with keys in the range [$k1$, $k2$].

COUNT c1 c2: Returns the number of items with cost in the range [$c1$, $c2$].

Output

For MAX k1 k2, print a single number: the maximum cost in the range [$k1$, $k2$].

For COUNT c1 c2, print a single number: the count of items in the range [$c1$, $c2$].

Sample Input

```
7
ADD 10 5.5
ADD 20 3.2
ADD 15 7.8
MAX 10 20
COUNT 3.0 6.0
REMOVE 15
MAX 10 20
```

Sample Output

```
7.8
2
5.5
```

ALCP 103

Efficient Search and Insert with Multi-Sorted Arrays

To search in a sorted array takes time logarithmic in the size of the array, but to insert a new items takes linear time. We can improve the running time for insertions by storing the items in several instead of just one sorted arrays. Let n be the number of items, let $k = [\log_2(n+1)]$, and write $n = n_{k-1}n_{k-2}\ldots n_0$ in binary notation. We use k sorted arrays A_i (some possibly empty), where A_i stores $n_i 2^i$ items. Each item is stored exactly once, and the total size of the arrays is indeed $\sum_{i=0}^{k} n_i 2^i = n$ .Although each individual array is sorted, there is no particular relationship between the items in different arrays.

- Explain how to search in this data structure and analyse your algorithm.
- Explain how to insert a new item into the data structure and analyse your algorithm, both in worst-case and in amortized time.

Input

The first line contains an integer Q, the number of queries.

Each of the next Q lines represents an operation:

- INSERT x: Inserts the element xxx into the data structure.
- SEARCH x: Searches for xxx and returns whether it exists (YES or NO).

PRINT: Displays the current state of all sorted arrays A0,A1,...,Ak.

Output

For SEARCH x, print YES if x exists, otherwise print NO.

For PRINT, print the sorted arrays in order.

Sample Input

```
7
INSERT 5
INSERT 3
INSERT 7
SEARCH 3
SEARCH 10
INSERT 10
PRINT
```

Sample Output

```
YES
NO
A0: [10]
A1: [5, 7]
A2: [3]
```

ALCP 104 | The offline minimum problem

The offline minimum problem is about maintaining a subset of $[n] = \{1,2,...,n\}$ under the operations INSERT and EXTRACTMIN. Given an interleaved sequence of n insertions and m min-extractions, the goal is to determine which key is returned by which min-extraction. We assume that each element $i \in [n]$ is inserted exactly once. Specifically, we wish to fill in an array $E[1..m]$ such that $E[i]$ is the key returned by the i^{th} min-extraction. Note that the problem is offline, in the sense that we are allowed to process the entire sequence of operations before determining any of the returned keys.

- Describe how to use a union-find data structure to solve the problem efficiently.
- Give a tight bound on the worst-case running time of your algorithm.

Input

A sequence of INSERT and EXTRACTMIN operations, where each element i in $[n]$ is inserted exactly once.

INSERT 4

INSERT 8

EXTRACTMIN

INSERT 5

EXTRACTMIN

INSERT 3

INSERT 6

EXTRACTMIN

EXTRACTMIN

Output

An array $[1...m]$ such that $E[i]$ stores the key returned by the i^{th} min-extraction.

E = [4, 5, 3, 6]

Each number corresponds to the minimum element extracted at each EXTRACTMIN operation.

Sample Input	**Sample Output**

INSERT 4
INSERT 8
EXTRACTMIN
INSERT 5
EXTRACTMIN
INSERT 3
INSERT 6
EXTRACTMIN
EXTRACTMIN
	E = [4, 5, 3, 6]

ALCP 105 | Dunce Cap Problem

The dunce cap is obtained by gluing the three edges of a triangular sheet of paper to each other. [After gluing the first two edges you get a cone, with the glued edges forming a seam connecting the cone point with the rim. In the final step, wrap the seam around the rim, gluing all three edges to each other. To imagine how this work, it might help to think of the final result as similar to the shell of a snale.]

- Is the dunce cap a 2-manifold? Justify your answer.
- Give a triangulation of the dunce cap, making sure that no two edges connect the same two vertices and no two triangles connect the same three vertices.

Input

Is the Dunce Cap a 2-Manifold?

A 2-manifold (or surface) is a space where every point has a neighbourhood homeomorphic to an open disk in R^2

Justification:

Manifold-Like Behaviour in Most Regions:

Most points on the dunce cap behave like a typical surface (they have neighbourhoods that look like disks).

Singular Point at the Glued Edge:

The point where all three edges meet is problematic.

It has a self-intersecting structure.

It fails to have a neighbourhood homeomorphic to an open disk.

Instead, it has a more complex topology that prevents it from being a 2-manifold.

Output

Triangulation of the Dunce Cap

A triangulation is a way of dividing the surface into non-overlapping triangles, ensuring:

1. No two edges connect the same two vertices.
2. No two triangles share the same three vertices.

Steps for Triangulation:

1. Label the three vertices of the original triangular sheet as A, B, C.
2. Introduce an internal vertex (D) to help in constructing a triangulation that maintains the correct topology.
3. Define the triangles carefully so that no two triangles share the same three vertices.

Triangulation Example:

Let's define the triangles as follows:

1. Triangle 1: (A,B,D)(A, B, D)(A,B,D)
2. Triangle 2: (B,C,D)(B, C, D)(B,C,D)
3. Triangle 3: (C,A,D)(C, A, D)(C,A,D)

By gluing the edges appropriately (as per the dunce cap construction), we ensure all three original edges are identified.

The dunce cap is NOT a 2-manifold due to the singular gluing point.

A valid triangulation consists of three triangles: (A, B, D), (B, C, D), and (C, A, D), ensuring correct connectivity without duplications.

ALCP 106

Bin Packing Problem

The bin packing problem is the following. You are given n metal objects, each weighing between zero and one kilogram. You also have a collection of large, but fragile bins. Your aim is to find the smallest number of bins that will hold the *n* objects, with no bin holding more than one kilogram.

The first-fit heuristic for the bin packing problem is the following. Take each of the objects in the order in which they are given. For each object, scan the bins in increasing order of remaining capacity, and place it into the first bin in which it fits. Design an algorithm that implements the first-fit heuristic (taking as input the n weights w_1, w_2,... , w_n and outputting the number of bins needed when using first-fit) in O(n log n) time.

To solve the bin packing problem using the First-Fit Heuristic, the idea is to use a list of bins, each of which has a remaining capacity of 1 kilogram. For each object, we find the first bin in which it fits and place the object there.

- You are given n objects with weights w_1, w_2, ..., w_n, where each weight is between 0 and 1 kilogram.
- You have a collection of bins, each with a capacity of 1 kilogram.
- The task is to determine the minimum number of bins needed to pack all the objects, using the First-Fit Heuristic.

The First-Fit heuristic works by placing each item into the first available bin that has enough remaining capacity to fit the item. If no bin can accommodate an object, a new bin is added.

Input

n = 6

weights = [0.4, 0.8, 0.1, 0.3, 0.7, 0.2]

Step-by-Step Execution of the First-Fit Heuristic:

Initialize empty bins, each with a capacity of 1 kg.

Process each item in the given order and place it in the first bin where it fits.

Placing items in bins:

0.4 → Put in Bin 1 → Remaining: 0.6

0.8 → Doesn't fit in Bin 1 → Put in Bin 2 → Remaining: 0.2

0.1 → Fits in Bin 1 → Remaining: 0.5

0.3 → Fits in Bin 1 → Remaining: 0.2

0.7 → Doesn't fit in Bin 1 or Bin 2 → Put in Bin 3 → Remaining: 0.3

0.2 → Fits in Bin 1 → Remaining: 0.0

Output

Bin 1: [0.4, 0.1, 0.3, 0.2] → Full

Bin 2: [0.8] → Remaining: 0.2

Bin 3: [0.7] → Remaining: 0.3

Minimum number of bins required: 3

Sample Input	**Sample Output**

n = 5
weights = [0.9, 0.7, 0.8, 0.5, 0.2]
	Number of bins needed: 3

```
n = 6                                                     Number of bins needed: 3
weights = [0.6, 0.4, 0.8, 0.7, 0.3, 0.5]
```

ALCP 107 | The Power of DNA Sequence Comparison

DNA is formed over four elements or nucleotides called adenine (*A*), cytosine (*C*), guanine (*G*) and thymine (*T*), so in this work DNA sequences will be represented by strings containing elements of a 4-letter alphabet *A*, *C*, *G*, *T*.

Given two DNA sequences, S_1 and S_2, consisting of the nucleotides *A*, *T*, *G*, and *C*, the goal is to:

1. Identify the best alignment of *S1* and *S2* using a scoring system for matches, mismatches, and gaps.
2. Compute the optimal similarity score between the sequences.
3. Identify subsequences or regions of high similarity that may suggest functional or evolutionary relationships.

Input

Given two DNA sequences:

S1=ACTGACG

S2=GACTTAC

A scoring system with:

- Match score (+1): When two nucleotides are identical.
- Mismatch penalty (−1): When two nucleotides differ.
- Gap penalty (−2): When aligning a nucleotide with a gap.

Output

- The optimal alignment of S1and S2.
- The similarity score for the alignment.
- A visualization of aligned sequences (e.g., using gaps to show insertions/deletions).

Sample Input	**Sample Output**
S1 = ACTGACG	S_1: ACTGACG
S2 = GACTTAC	S_2: -ACTTAC
Scoring system: +1(match), −1(mismatch), −2 (gap)	Similarity Score: 2

ALCP 108 The Change Problem

Find the minimum number of coins needed to change a given amount of money.

Input

An amount of cents *M* and coins from denominations c = (c_1, c_2,..., c_d).

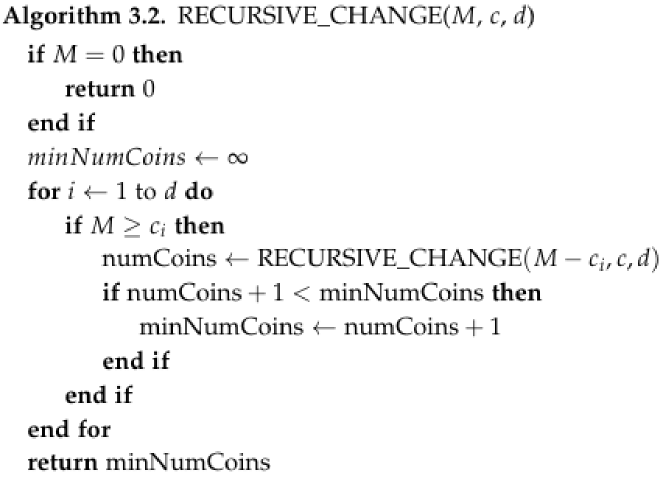
Output

The minimum number of coins *c* = (c_1, c_2,..., c_d) to change *M* cents.

First, if minNumCoins_M is the smallest number of coins required to change M cents, we can introduce the following recurrence relation:

$$\text{minNumCoins}_M = \min \begin{cases} \text{minNumCoins}_{M-c_1} + 1 \\ \vdots \\ \text{minNumCoins}_{M-c_d} + 1 \end{cases}$$

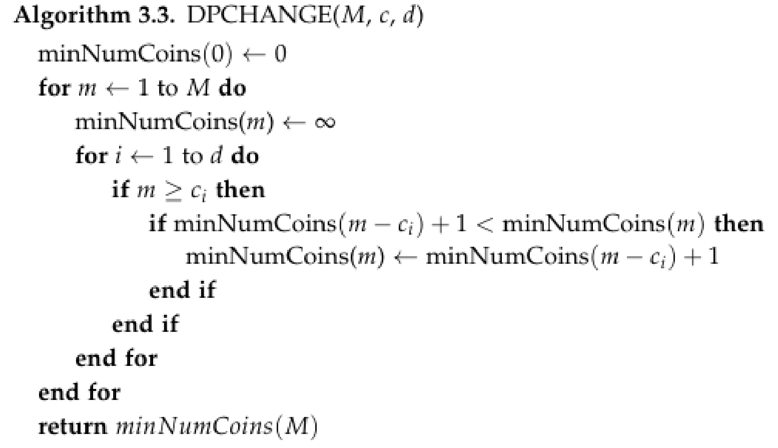
A simple recursive change algorithm could be designed straightforward based on the previous recurrence. Every time the amount of coins needed to change M cents is computed, it implies d recursive calls to find out the minimum number of coins needed to change M – c_i, with i = 1,...,d.



```
Algorithm 3.2. RECURSIVE_CHANGE(M, c, d)
  if $M = 0$ then
      return 0
  end if
  $minNumCoins \leftarrow \infty$
  for $i \leftarrow 1$ to $d$ do
      if $M \geq c_i$ then
          numCoins ← RECURSIVE_CHANGE($M - c_i, c, d$)
          if numCoins + 1 < minNumCoins then
              minNumCoins ← numCoins + 1
          end if
      end if
  end for
  return minNumCoins
```

The algorithm becomes unfeasible because of time complexity O(M^d)) to solve the problem.

Design a better optimal solution using dynamic programming approach. Since the optimal solution for *M* cents relies on optimal solutions for *M* – c_i, with *i* = 1,...,*d*, we can reverse the execution order and solve the problem for each increasing amount of money from 0 to *M*. It could seem we are facing a more complex task at first sight, but at every stage the algorithm is analyzing d precomputed values instead of recomputing them. This procedure is represented by the following faster algorithm, with running time O(*M.d*):



Algorithm 3.3. DPCHANGE(M, c, d)

```
minNumCoins(0) $\leftarrow 0$
for $m \leftarrow 1$ to $M$ do
    minNumCoins($m$) $\leftarrow \infty$
    for $i \leftarrow 1$ to $d$ do
        if $m \geq c_i$ then
            if minNumCoins($m - c_i$) $+ 1 <$ minNumCoins($m$) then
                minNumCoins($m$) $\leftarrow$ minNumCoins($m - c_i$) $+ 1$
            end if
        end if
    end for
end for
return $minNumCoins(M)$
```

Sample Input	**Sample Output**

M=11 cents
Coin denominations: c={1,5,10}
	Minimum number of coins: 222
	Explanation: 10+1=11 (1 coin of 10 and 1 coin of 1).

Exact Match with One Coin
M=25M = 25M=25 cents
Coin denominations: c={1,5,10,25}
	Minimum number of coins: 111
	Explanation: Use 1 coin of 25.

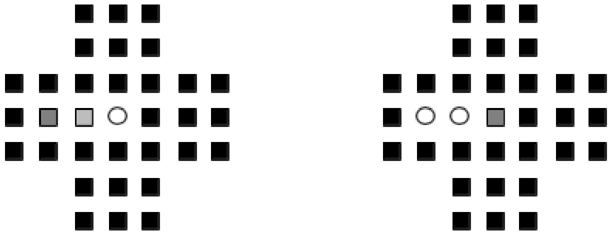
Multiple Denominations
M=7 cents
Coin denominations: c={1,3,4}
	Minimum number of coins: 2
	Explanation: 4+3=7 (1 coin of 4 and 1 coin of 3).

Large Amount
M = 63 cents
Coin denominations: c = {1, 5, 10, 25}
	Minimum number of coins: 6
	Explanation: 25+25+10+1+1+1 = 63

Non-Trivial Coin Set
M = 27 cents
Coin denominations: c = {1, 7, 10}
	Minimum number of coins: 3
	Explanation: 10 + 10 + 7 = 27 (2 coins of 10 and 1 coin of 7)

ALCP 109	Hi-Q Puzzle

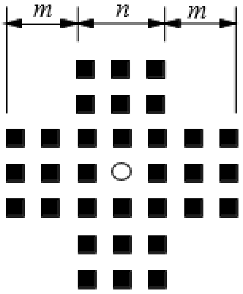
The following puzzle is called Hi-Q. You are given a board in which 33 holes have been drilled in the pattern shown in Figure below.



The initial configuration

After peg ▨
jumps peg □

A peg (indicated by a square) is placed in every hole except the center one (indicated by a circle). Pegs are moved as follows. A peg may jump over its neighbor in the horizontal or vertical direction if its destination is unoccupied. The peg that is jumped over is removed from the board. The aim is to produce a sequence of jumps that removes all pegs from the board except for one. The generalized HI-Q puzzle has $n(n + 4m)$ holes arranged in a symmetrical cross-shape as shown in Figure below, for any m and any odd n. Design an algorithm that solves the Hi-Q puzzle. Let $p = n(n + 4m)$. Your algorithm should run in time $O((4p)^{p-2})$.



Input

The initial board setup consists of 33 holes in a cross shape.

Each hole is occupied by a peg (■) except for the center hole, which is empty (○).

The game allows moves where a peg jumps over another peg into an empty hole, removing the jumped peg.

```
    ■ ■ ■
    ■ ■ ■
■ ■ ■ ■ ■ ■ ■
■ ■ ■ ○ ■ ■ ■
■ ■ ■ ■ ■ ■ ■
    ■ ■ ■
    ■ ■ ■
```

Output

```
    ○ ○ ○
    ○ ○ ○
○ ○ ○ ○ ○ ○ ○
○ ○ ○ ■ ○ ○ ○
○ ○ ○ ○ ○ ○ ○
    ○ ○ ○
    ○ ○ ○
```

Sample Input

```
    ■ ■ ■
    ■ ■ ■
■ ■ ■ ■ ■ ■ ■
■ ■ ■ ○ ■ ■ ■
■ ■ ■ ■ ■ ■ ■
    ■ ■ ■
    ■ ■ ■
```

Moves (Sample Solution Path)

Move peg at (3,1) → Jumps over (3,2) → Lands in (3,3)

Move peg at (5,3) → Jumps over (4,3) → Lands in (3,3)

Move peg at (3,5) → Jumps over (3,4) → Lands in (3,3)

Move peg at (1,3) → Jumps over (2,3) → Lands in (3,3)

Move peg at (3,3) → Jumps over (3,1) → Lands in (3,0)

Continue jumping until only one peg remains.

Sample Output

```
    ○ ○ ○
    ○ ○ ○
○ ○ ○ ○ ○ ○ ○
○ ○ ○ ■ ○ ○ ○
○ ○ ○ ○ ○ ○ ○
    ○ ○ ○
    ○ ○ ○
```

Final Board State - Only One Peg Left

ALCP 110 Ice Cream Cart Placement

The I-C Ice Cream Company plans to place ice cream carts at strategic intersections in Dallas so that for every street intersection, either there is an ice cream cart there, or an ice cream cart can be reached by walking along only one street to the next intersection. You are given a map of Dallas in the form of a graph with nodes numbered 1, 2,... ,*n* representing the intersections, and undirected edges representing streets. For each ice cream cart at intersection i, the city will levy \$$c_i$ in taxes. The I-C Ice Cream Company has *k* ice cream carts, and can make a profit if it pays at most \$d in taxes.

The goal is to write a backtracking algorithm that finds all possible ways to place k ice cream carts such that:

- Every intersection has either an ice cream cart or is adjacent to one.
- The total tax paid does not exceed d

Write a backtracking algorithm that outputs all of the possible locations for the k carts that require less than \$d in taxes. Analyze your algorithm.

Input

A graph representing Dallas' intersections and streets:

Nodes (intersections): {1, 2, 3, 4, 5}

Edges (streets): {(1,2), (2,3), (3,4), (4,5), (2,4)}

Cost of placing a cart at each intersection:

C = [3, 5, 2, 4, 6] (Cost for nodes 1,2,3,4,5 respectively)

Total budget available: D = 10

Number of carts to place: K = 2

```
 1 -- 2 -- 3
      |    |
      4 -- 5
```

Output

Each valid solution must:

Cover all intersections (either directly or via an adjacent cart).

Use exactly K = 2 carts.

Have a total cost ⩽ D = 10.

Sample Input	**Sample Output**
Simple Graph with Small Tax Costs Given graph 0 - 1 | | 2 – 3 Tax costs = [1, 2, 3, 4] K = 2 Max_tax = 5 **Simple Graph with High Tax Costs**	[[0, 1], [0, 2], [1, 2]]

Given graph [[0, 1]]

0 - 1

| |

2 – 3

Tax costs = [5, 6, 7, 8]

k (number of carts) = 2

max_tax = 10

Sparse Graph with More Carts

Given graph [[0, 1]]

0 - 1

|

2

Tax costs = [1, 3, 5]

k (number of carts) = 2

max_tax = 6

ALCP 111 | Postage Stamp Problem

The postage stamp problem is defined as follows. The Fussytown Post Office has n different postage stamp denominations, and regulations allow at most m stamps to be placed on any letter. The goal is to compute the length of the consecutive series of postage values (starting from 1) that can be legally realized under these rules.

Design an algorithm that, given n different postage stamp values in an array *P*[1..*n*] and the value of *m*, computes the length of the consecutive series of postage values that can legally be realized under these rules starting with the value 1. For example, if $n = 4$, $m = 5$, and the stamps have face value 1, 4, 12, and 21, then all postage values from 1 to 71 can be realized. Your algorithm should run in time $O(n^m)$. The algorithm aims to find the largest value of consecutive postage values starting from 1 that can be formed using at most m stamps from the given denominations. We will iterate through all possible postage values and check if it can be formed using the available stamps, keeping track of the smallest postage that cannot be formed.

Input

Number of stamp denominations: n = 4

Maximum stamps per letter: m = 5

Stamp denominations: P = [1, 4, 12, 21]

Output

We need to determine the longest consecutive sequence of postage values starting from 1 that can be formed using at most 5 stamps.

Step-by-Step Calculation:

We check all possible postage values starting from 1, ensuring they can be formed with at most m = 5 stamps.

Using {1}, we can generate: 1, 2, 3, 4, 5, ..., 5

Using {1, 4}, we can generate: 6, 7, ..., 16

Using {1, 4, 12}, we can generate: 17, 18, ..., 37

Using {1, 4, 12, 21}, we can generate: 38, 39, ..., 71

First missing postage value: 72 (cannot be formed with at most 5 stamps)

Thus, all postage values from 1 to 71 can be realized, but 72 is the first missing value.

Maximum consecutive postage values: 71

Sample Input	**Sample Output**

Simple Case with Small Values
N = 4
M = 5
Stamps = [1, 4, 12, 21]
	71

Small Number of Denominations
N = 2
M = 3
Stamps = [1, 5]
	9

Larger Number of Denominations	

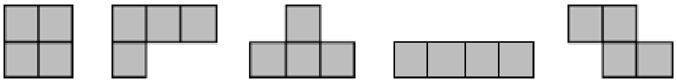
N = 5
M = 4
Stamps = [1, 2, 5, 10, 20]
	28

Maximum number of Stamps
N = 3
M = 10
Stamps = [1, 5, 7]
	55

Case with Larger Denominations and Higher m
N = 6
M = 6
Stamps = [1, 3, 5, 8, 12, 20]
	49

ALCP 112 — Polyomino or n-omino

A polyomino, or more precisely, an n-omino is a plane figure made up of n unit-size squares lined up on a grid. The figure below shows all of the 4-ominos. The challenge is to design an algorithm that generates all possible *n*-ominos, including all their rotations and reflections.



The goal is to:

- Generate all possible *n*-ominos, considering all rotations and reflections.
- Modify the algorithm to remove duplicates caused by rotations and reflections.

Generate all Polyominoes (O(3^*n*)): The algorithm can recursively build polyominoes by adding new squares to an existing shape. This can be done in time (3^n) because each new square has three possible directions in which it can be placed (left, right, up, down).

Remove Duplicates from Rotations and Reflections: After generating all polyominoes, rotate and reflect each polyomino in all possible ways (since the grid can be rotated or reflected). Store only unique polyominoes after applying all rotations and reflections.

- Design an algorithm that runs in time O(3^n) to generate all *n*-ominos, including all rotations and reflections. And modify your algorithm to remove all rotations and reflections.

Input

Value of n (size of the polyomino in squares): n = 4

Grid-based tiling rules:

Each new square must be connected to at least one existing square.

Can be placed in 3 directions (left, right, up, down) from the last placed square.

No duplicates from rotations or reflections.

Output

All Unique 4-Ominos, Considering Rotations and Reflections)

All possible 4-ominoes (also called tetrominoes) after removing duplicates:

1. ■■■■ (Straight)

2. ■■ (Square)
■■

3. ■■■
■ (L-shape)

4. ■■■
■ (T-shape)

5. ■■
■■ (S-shape)

Unique 4-Ominos

Unique 4-ominos:

```
1. ####
2. ##
    ##
3. ###
    #
4. ###
   #
5. ##
    ##
```

Total Unique Polyominoes (n = 4) after removing rotations/reflections: **5**
For n = 5, the number of unique polyominoes is `12`.

Sample Input	Sample Output
Simple Case with 1-Omino N = 1 (1-omino)	[[0, 0]]
Case with 2-Omino (Domino) N = 2 (2-omino)	[[[0,0], [1,0]], [[0,0], [0,1]]]
Case with 3-Omino N = 3 (3-omino)	[[[0,0], [1,0], [2,0]], [[0,0], [1,0], [1,1]], [[0,0], [1,0], [0,1]]]
Case with 4-Omino N = 4 (4-omino)	[[[0,0], [1,0], [1,1], [2,1]], [[0,0], [1,0], [1,1], [1,2]], [[0,0], [1,0], [1,1], [2,1]]]
Case with 5-Omino N = 5 (5-omino)	[[[0,0], [1,0], [1,1], [2,1], [3,1]], [[0,0], [1,0], [2,0], [2,1], [2,2]]]
Case with 6-omino N = 6 (6-omino)	[[[0,0], [1,0], [1,1], [2,1], [2,2], [3,2]], [[0,0], [1,0], [1,1], [2,1], [2,2], [3,2]]]
Remove Rotations and Reflections for 3-Omino N = 3	[[[0,0], [1,0], [2,0]], [[0,0], [1,0], [1,1]]]

ALCP 113 — Sum Decomposition into Ascending Positive Numbers

Design a backtracking algorithm that takes a natural number *C* as input and outputs all possible ways to express *C* as the sum of a group of distinct ascending positive integers. Each group must satisfy the following constraints:

- The numbers in the group must be strictly increasing.
- The sum of the numbers in the group must equal *C*.

The backtracking algorithm works as follows:

1. Start with an empty list to hold the current group of numbers.
2. Use a recursive function to attempt to add numbers to the group, starting from a minimum value (to ensure ascending order).
3. If the sum of the group equals *CCC*, add the group to the results.
4. If the sum exceeds *CCC*, backtrack by removing the last number and trying the next possibility.

Continue until all combinations have been explored.

Input

C = 6

Output

All possible ways to express 6 as a sum of distinct ascending positive integers:

1 + 2 + 3

1 + 5

2 + 4

Sample Input	Sample Output
C = 3	1 + 2 3
C = 6	1+2+3 1+5 2+4 6
C = 10	1+2+3+4 1+2+7 1+3+6 1+4+5 1+9 2+3+5 2+8 3+7 4+6 10
C = 15	1+2+3+4+5 1+2+3+9 1+2+4+8

1+2+5+7
1+2+13
1+3+4+7
1+3+5+6
1+3+11
1+4+10
1+5+9
1+6+8
1+14
2+3+4+6
2+3+10
2+4+9
2+5+8
2+6+7
2+13
3+4+8
3+5+7
3+12
4+5+6
4+11
5+10
6+9
7+8
15

ALCP 114 — Open Knight's Tour

A knight on a chessboard moves in an "*L*" shape: two squares in one direction and one square in a perpendicular direction (or vice versa). Using this movement pattern, the problems are defined as follows:

An open knight's tour is a sequence of n^2 moves by a knight on an $n \times n$ chessboard such that:

1. The knight starts on any square.
2. The knight visits every square on the chessboard exactly once.
3. The tour does not need to return to the starting square.

An open knight's tour is a sequence of n^2 1-knight's moves starting from some square of an n x n chessboard, such that every square of the board is visited exactly once. Design an algorithm for finding open knight's tours that runs in time $O(n^2 8^{n2})$.

A closed knight's tour is similar to an open knight's tour, but it satisfies an additional condition:

1. After visiting all squares on the board exactly once, the knight can make one final move to return to its starting square.

A closed knight's tour, as we have seen already, is a cyclic version of an open knight's tour (that is, the start point is reachable from the finish point a single knight's move). Design an algorithm for finding closed knight's tours that runs in time $O(8^{n2})$.

Input

Chessboard Size: n = 5 (A 5x5 chessboard)

Starting Position: (0, 0) (Top-left corner)

Output

A sequence of knight's moves that covers all the squares on a 5x5 chessboard exactly once, starting from the top-left corner (0, 0):

(0, 0) -> (2, 1) -> (4, 2) -> (3, 4) -> (1, 3) -> (0, 5)

(2, 4) -> (4, 3) -> (3, 1) -> (1, 2) -> (0, 4) -> (2, 5)

(4, 1) -> (3, 0) -> (1, 1) -> (0, 3) -> (2, 2) -> (4, 4)

(3, 3) -> (1, 4) -> (2, 0) -> (0, 1) -> (2, 3) -> (4, 0)

Start Position: (0, 0) (top-left corner)

The knight makes a series of moves such that every square is visited exactly once.

The knight doesn't return to the start (hence it's an open knight's tour).

Sample Input	**Sample Output**
Open Tour on a 5x5 Board	
N = 5	(1, 1) → (3, 2) → (5, 3) → (4, 1) → (2, 2) → ...
Starting position: (1, 1)	
Closed Tour on a 5x5 Board	
N= 5	(1, 1) → (3, 2) → (5, 3) → ... → (1, 1)
Starting position: (1, 1)	
Open Tour on a 6x6 Board	
N = 6	(1, 1) → (3, 2) → (5, 3) → ... → (4, 4)

Starting position: (1, 1)	
Closed Tour on a 8x8 Board	
N = 8	(1, 1) → (3, 2) → (5, 3) → … → (1, 1)
Starting position: (1, 1)	

ALCP 115 Optimal Student Pairing Problem

You have n students in a class, and you need to assign them into teams of two. You are given:

1. An integer *n* (assume *n* is even).
2. A 2D preference matrix *P*[1..*n*,1..*n*], where *P*[*i*][*j*] represents the preference score of student *i* for working with student *j*.

The weight of a pairing between student *i* and student *j* is defined as:

Weight (*i*, *j*) = *P*[*i*][*j*] + *P*[*j*][*i*]

Suppose you have n students in a class and wish to assign them into teams of two (you may assume that *n* is even). You are given an integer array *P* [1..*n*, 1..*n*] in which *P*[*i*, *j*] contains the preference of student *i* for working with student j. The weight of a pairing of student i with student j is defined to be *P* [*i*, *j*] *P* [*j*, *i*]. The pairing problem is the problem of pairing each student with another student in such a way that the sum of the weights of the pairings is maximized. Design an algorithm for the pairing problem that runs in time proportional to the number of pairings. Analyze your algorithm.

The goal is to determine a pairing of all *n* students (where each student is assigned to exactly one other student) such that the sum of the weights of all pairings is maximized.

Input

Number of students (n): 4

Preference matrix P:

```
P = [
 [0, 3, 2, 5],
 [3, 0, 4, 6],
 [2, 4, 0, 7],
 [5, 6, 7, 0]
]
```

P[i][j] represents the preference score of student i for working with student j.

Output

A pairing of students that maximizes the total weight:

Pairing:

Student 1 and Student 4

Student 2 and Student 3

Maximized Sum of Weights = 5 (for 1-4) + 7 (for 2-3) = 12

Sample Input	**Sample Output**
n = 2	Maximum pairing weight: 7
P = [[0, 3], [4, 0]]	Pairings: (1, 2)
n = 4	Maximum pairing weight: 22
P = [[0, 3, 5, 2],	Pairings: (1, 2) and (3, 4) OR (1, 3) and (2, 4)
[3, 0, 4, 6],	
[5, 4, 0, 8],	

[2, 6, 8, 0]]	
n = 6 P = [[0, 1, 2, 3, 4, 5], [1, 0, 6, 7, 8, 9], [2, 6, 0, 10, 11, 12], [3, 7, 10, 0, 13, 14], [4, 8, 11, 13, 0, 15], [5, 9, 12, 14, 15, 0]]	Maximum pairing weight: 72 Pairings: (1, 6), (2, 5), (3, 4)
n = 4 P = [[0, 5, 5, 5], [5, 0, 5, 5], [5, 5, 0, 5], [5, 5, 5, 0]]	Maximum pairing weight: 20 Pairings: Any valid pairing, e.g., (1, 2) and (3, 4)

ALCP 116 | Marble Distribution using Divide-and-Conquer

You have m marbles and *n* jars. The goal is to distribute all *m* marbles into *n* jars such that no jar holds more than *c* marbles, where $m \le c \cdot n$. Your task is to:

1. Generate all valid distributions of the marbles into jars, where the number of marbles in each jar satisfies the constraint $\text{jar}_i \le c$ for i=1, 2 ,..., *n*
2. Output the number of marbles in each jar for every valid distribution.

Use divide-and-conquer to design an algorithm that generates all of the ways of distributing the marbles into the jars so that no jar holds more than *c* marbles, where *m cn*. Your algorithm must print the number of marbles in each jar for each of the different ways of distributing the marbles. So, for example, with *m* = 10, *n* = 3, and *c* = 5, the output of your algorithm would be

0	5	5	3	2	5	4	3	3	5	3	2
1	4	5	3	3	4	4	4	2	5	4	1
1	5	4	3	4	3	4	5	1	5	5	0
2	3	5	3	5	2	5	0	5			
2	4	4	4	1	5	5	1	4			
2	5	3	4	2	4	5	2	3			

Input

Number of marbles (m): 10

Number of jars (n): 3

Maximum marbles per jar (c): 5

Output

All valid distributions of 10 marbles into 3 jars, where each jar can hold at most 5 marbles:

[0, 5, 5]

[1, 4, 5]

[2, 3, 5]

[3, 2, 5]

[4, 1, 5]

[5, 0, 5]

[0, 4, 6]

[1, 3, 6]

[2, 2, 6]

[3, 1, 6]

[4, 0, 6]

We need to distribute 10 marbles into 3 jars.

Each jar can hold a maximum of 5 marbles.

The solution generates all possible combinations where the number of marbles in each jar is between 0 and 5, and the total number of marbles in all jars is exactly 10.

Sample Input	**Sample Output**

M = 4
N = 2
C = 3
	0 4
	1 3
	2 2
	3 1
	4 0

M = 5
N = 3
C = 3
	0 0 5
	0 1 4
	0 2 3
	0 3 2
	0 4 1
	0 5 0
	1 0 4
	1 1 3
	1 2 2
	1 3 1
	1 4 0
	2 0 3
	2 1 2
	2 2 1
	2 3 0
	3 0 2
	3 1 1
	3 2 0
	4 0 1
	4 1 0
	5 0 0

M = 15
N = 3
C = 5
	5 5 5

ALCP 117 — Minimal-change Bit-String Generation

A minimal-change algorithm for generating bit-strings generates them in such an order that each string differs from its predecessor in exactly one place. The following algorithm generates all of the bit-strings of length *n* in minimal-change order. The array *b* is initially set to all zeros, and the bit-strings are generated with a call to generate (*n*).

```
Procedure generate (n)
  if n = 0 then
     process (b)
  else
     generate (n - 1)
     Complement b[n]
     generate (n -1)
```

- Prove that this algorithm does generate the 2^n bit-strings of length *n* in minimal-change order.
- Prove that for all *n* >= 1, the number of times that a bit of b is complemented (that is, the number of times that line 4 is executed) is 2^n -1. Hence, deduce the running time of the algorithm.

Input

n = 3

Output

All bit-strings of length 3 generated in minimal-change (Gray Code) order:

```
000
001
011
010
110
111
101
100
```

Sample Input	**Sample Output**
N = 2	00 01 11 10
N = 3	000 001 011

	010 110 111 101 100
N = 4	0000 0001 0011 0010 0110 0111 0101 0100 1100 1101 1111 1110 1010 1011 1001 1000

ALCP 118 — One Processor Scheduling Problem

The one-processor scheduling problem is defined as follows. We are given a set of *n* jobs. Each job *i* has a start time t_i, and a deadline d_i. A feasible schedule is a permutation of the jobs such that when the jobs are performed in that order, then every job is finished before the deadline.

- Design a greedy algorithm for the one-processor scheduling problem processes the jobs in order of deadline (the early deadlines before the late ones).
- Show that if a feasible schedule exists, then the schedule produced by this greedy algorithm is feasible.

Input

Array of jobs, each with t_i and d_i.

Output

Sort: Sort the jobs by their deadlines in non-decreasing order.

Schedule: Process the jobs in the sorted order.

Sample Input	**Sample Output**
Feasible Schedule Exists Jobs: [(t1=1,d1=3),(t2=2,d2=5),(t3=1,d3=6)]	Feasible schedule exists: Jobs are processed in the order 1→2→3.
Feasible Schedule Does Not Exist Jobs: [(t1=3,d1=3),(t2=2,d2=4),(t3=2,d3=5)]	No feasible schedule exists.
Jobs with Same Deadline Jobs: [(t1=2,d1=5),(t2=3,d2=5),(t3=1,d3=5)]	Feasible schedule exists: Jobs are processed in the order 3→1→2
All Jobs Have Sufficient Deadlines Jobs: [(t1=1,d1=10),(t2=2,d2=15),(t3=3,d3=20)]	Feasible schedule exists: Jobs are processed in the order 1→2→3

ALCP 119 | Max-cost Spanning Tree

A max-cost spanning tree is a spanning tree in which the sum of the edge weights is maximized. This problem focuses on directed graphs where the edges have weights, and we are tasked with finding a directed spanning tree under specific conditions. Design an algorithm for the max-cost spanning tree.

A directed spanning tree of a directed graph is a spanning tree in which the directions on the edges point from parent to child.

Suppose Prim's algorithm is modified to take the cheapest edge directed out of the tree.

- Does it find a directed spanning tree of a directed graph?
- If so, will it find a directed spanning tree of minimum cost?

Suppose Kruskal's algorithm is used to construct a directed spanning tree.

- Does it find a directed spanning tree of a directed graph?
- If so, will it find a directed spanning tree of minimum cost?

Consider the following algorithm. Suppose Prim's algorithm is modified, and then run n times, starting at each vertex in turn. The output is the cheapest of the n directed spanning trees.

- Does this algorithm output a directed spanning tree of a directed graph?
- If so, will it find a directed spanning tree of minimum cost?

Input

n: The number of vertices in the graph.

m: The number of directed edges in the graph.

edges: A list of directed edges with weights. Each edge is represented by three integers: u, v, and w, where:

u is the starting vertex (parent),

v is the ending vertex (child),

w is the weight of the edge from u to v.

n = 4 (number of vertices)

m = 5 (number of edges)

edges = [

(0, 1, 3), (edge from vertex 0 to vertex 1 with weight 3),

(0, 2, 4), (edge from vertex 0 to vertex 2 with weight 4),

(1, 3, 2), (edge from vertex 1 to vertex 3 with weight 2),

(2, 3, 5), (edge from vertex 2 to vertex 3 with weight 5),

(0, 3, 6) (edge from vertex 0 to vertex 3 with weight 6)

]

Output

Spanning Tree Edges: A list of directed edges that form a max-cost spanning tree. Each edge is represented by two integers u, v, indicating the directed edge from vertex u to vertex v.

Total Weight: The sum of the edge weights of the chosen directed spanning tree.

Spanning Tree Edges: [(0, 3), (0, 1), (2, 3)]

Total Weight: 6 + 3 + 5 = 14

Sample Input	Sample Output
Simple Directed Graph Graph with 4 vertices and directed edges: Edges: 1 → 2 (cost = 10) 2 → 3 (cost = 15) 3 → 4 (cost = 20) 4 → 1 (cost = 25)	Modified Prim's algorithm (cheapest edge directed out of the tree): • Directed spanning tree: Yes. • Minimum-cost spanning tree: No (this finds the max-cost tree). Modified Kruskal's algorithm: • Directed spanning tree: Yes. • Minimum-cost spanning tree: No. Prim’s algorithm run n times (cheapest tree among n): • Directed spanning tree: Yes. • Minimum-cost spanning tree: Yes (only if starting at every vertex is feasible).
Disconnected Graph Graph with 5 vertices and directed edges: Edges: 1 → 2 (cost = 5), 2 → 3 (cost = 6), 4 → 5 (cost = 7).	Modified Prim's algorithm: • Directed spanning tree: No (graph is disconnected). • Minimum-cost spanning tree: Not applicable. Modified Kruskal's algorithm: • Directed spanning tree: No. • Minimum-cost spanning tree: Not applicable. Prim’s algorithm run n times: • Directed spanning tree: No. • Minimum-cost spanning tree: Not applicable.

ALCP 120

Dynamic Investment Strategy for Maximizing Returns

You have $1 and want to invest it for *n* months. At the beginning of each month, you must choose from the following three options:

- Purchase a savings certificate from the local bank. Your money will be tied up for one month. If you buy it at time t, there will be a fee of CS(*t*) and after a month, it will return *S*(*t*) for every dollar invested. That is, if you have $k at time *t*, then you will have $\$(k - C_S(t)).S(t)$ at time *t* + 1.
- Purchase a state treasury bond. Your money will be tied up for six months. If you buy it at time *t*, there will be a fee of CB(*t*) and after six months, it will return B(*t*) for every dollar invested. That is, if you have $*k* at time t, then you will have $(k - CB(*t*)).B(t) at time *t* + 6.
- Store the money in a sock under your mattress for a month. That is, if you have $*k* at time *t*, then you will have $*k* at time *t* + 1.

Suppose you have predicted values for *S, B, CS, CB* for the next *n* months. Devise a dynamic programming algorithm that computes the maximum amount of money that you can make over the n months in time O(n).

Input

n: Number of months.

S(t): List of multipliers for savings certificates at each month t.

B(t): List of multipliers for state treasury bonds at each month t.

CS(t): List of fees for savings certificates at each month t.

CB(t): List of fees for state treasury bonds at each month t.

n = 6

S = [1.1, 1.2, 1.3, 1.4, 1.5, 1.6] # Multipliers for Savings Certificates

B = [1.5, 1.4, 1.3, 1.2, 1.1, 1.0] # Multipliers for State Treasury Bonds

CS = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6] # Fees for Savings Certificates

CB = [0.3, 0.2, 0.1, 0.0, 0.1, 0.2] # Fees for State Treasury Bonds

Output

The maximum amount of money you can have at the end of n months.

Maximum amount at the end of 6 months: 2.9036

Sample Input

Number of months (n): 8

Predicted values for S(t), B(t), CS(t), and CB(t)

Month	S(t)	B(t)	CS(t)	CB(t)
1	1.10	1.50	0.02	0.05
2	1.12	1.55	0.02	0.05
3	1.08	1.60	0.02	0.05
4	1.15	1.65	0.02	0.05
5	1.20	1.70	0.02	0.05

Sample Output

Maximum amount of money after n=8n = 8n=8 months: **$1.6984**

Investment strategy:

- Month 1: Invest in savings certificate (S(1).
- Month 2: Store money in sock.
- Month 3: Buy treasury bond (B(3).
- Month 8: Store money in sock.

6	1.10	1.75	0.02	0.05
7	1.05	1.80	0.02	0.05
8	1.18	1.85	0.02	0.05

ALCP 121	Tower of Hanoi and the End of the Universe

The Tower of Hanoi problem involves moving a set of *n* disks from one needle (peg) to another, following these rules:

1. Only one disk may be moved at a time.
2. A disk can only be placed on an empty needle or on top of a larger disk.
3. The goal is to move all disks from the starting needle to the target needle using an auxiliary needle.

According to legend, there is a set of 64 gold disks on 3 diamond needles being solved by priests of Brahma. The Universe is supposed to end when the task is complete. If done correctly, it will take *T* (64) = 2^{64} - 1 = 1.84 x 10^{19} moves. At one move per second, that's 5.85 x 10^{11} years. More realistically, at 1 minute per move (since the largest disks must be very heavy) during working hours, it would take 1.54 × 10^{14} years. The current age of the Universe is estimated at ≈ 10^{10} years. Perhaps the legend is correct.

There is an algorithm that will do this using at most 2^n -1 moves. There is even one that is probably optimal.

Input

The input consists of a single integer n which represents the number of disks.

n = 3

Output

The output will display the minimum number of moves required to solve the Tower of Hanoi problem for n disks, which is calculated as 2^n - 1.

Minimum number of moves required for 3 disks: 7

Sample Input	**Sample Output**
Number of disks: n=4	Number of moves: 2^4 - 1=15 Move disk 1 from Source to Target. Move disk 2 from Source to Auxiliary. Move disk 1 from Target to Auxiliary. Move disk 3 from Source to Target. Move disk 1 from Auxiliary to Source. Move disk 2 from Auxiliary to Target. Move disk 1 from Source to Target.

ALCP 122 — Winning a War

A general is planning a war campaign where multiple battles are scheduled, and the goal is to maximize the overall victory score. The general has limited troops and resources to allocate to each battle. Each battle has:

1. A minimum troop requirement for participation.
2. A victory score based on the number of troops allocated.
3. A restriction that no two battles can occur simultaneously (overlapping time intervals).

The challenge is to design an algorithm to decide which battles to participate in, how to allocate troops, and how to maximize the total victory score, considering the constraints of troop availability and non-overlapping schedules.

Input

1. An integer *n*: the number of battles.
2. s_i, e_i, r_i, v_i : start time, end time, required troops, and victory score.
3. An integer *T*: the total number of troops available.

Output

1. An integer representing the maximum total victory score.
2. A list of selected battles and the troop allocation for each.

Sample Input	**Sample Output**
5	320
1, 4, 10, 50	1, 4, 10
3, 5, 20, 100	6, 8, 15
6, 8, 15, 70	9, 11, 25
9, 11, 25, 200	
8, 10, 10, 80	
50	

ALCP 123 Decrypting Permuted Text

A common but insecure method of encrypting text is to permute the letters of the alphabet. In this method, each letter of the alphabet is consistently replaced in the text by some other letter, ensuring that:

1. Each letter maps uniquely to another letter.
2. No two letters are replaced by the same letter.

The encryption must also be reversible. Your task is to decrypt several encoded lines of text, assuming the following:

1. Each line uses a different letter permutation for encryption.
2. The decrypted text must only consist of valid words found in a dictionary of known words.

Input

1. The first line contains an integer *n*, which specifies the number of words in the dictionary.
2. The next n lines contain the dictionary words in alphabetical order. All words are in lowercase.
3. Following the dictionary are several lines of encrypted text. Each line:
   - Contains only lowercase letters and spaces.
   - Does not exceed 80 characters in length.

There are no more than 1,000 dictionary words, and no word exceeds 16 characters in length.

Output

For each encrypted line:

- Print the decrypted line if the text can be successfully decrypted such that all words are found in the dictionary.
- If there is no valid decryption, print the line with every letter replaced by an asterisk (*).

Sample Input

```
6
and
dick
jane
puff
spot
yertle
bjvg xsb hxsn xsb qymm xsb rqat xsb pnetfn
xxxx yyy zzzz www yyyy aaa bbbb ccc dddddd
```

Sample Output

```
dick and jane and puff and spot and yertle
**** *** **** *** **** *** **** *** ******
```

ALCP 124 | Maximum Skill Utilization

You have *n* employees and *m* projects, where *n*=*m*. Each employee has a certain skill level for each project, and your goal is to assign one project to each employee such that the total skill level is maximized.

Input

1. Integer *n*: Number of employees/projects.
2. A 2D matrix S[1..*n*][1..*n*],where S[i][j] is the skill level of employee *i* for project *j*.

Output

1. Maximum total skill level.
2. Assignment of employees to projects.

Sample Input	**Sample Output**
n = 3	Maximum skill level: 34
S = [[10, 2, 3],	Assignment: Employee 1 -> Project 1,
[4, 15, 6],	Employee 2 -> Project 2,
[7, 8, 9]]	Employee 3 -> Project 3

ALCP 125 | Efficient Longest Monotonic Sequence

If you could design a data structure that would return the maximum value of L_j for all $x_j \leq x_i$ in O(log *n*) time then we may be able to obtain a better bound with the simpler recurrence for the longest monotonic sequence. Note that this data structure must support insertion of new points as we scan from left to right. Since the points are known in advance we can pre construct the BST skeleton (see Figure 1) and fill in the actual points as we scan from left to right, thereby avoid-ing dynamic restructuring. As we insert points, we update the heap data along the insertion path. Work out the details of this structure and the analysis to obtain an O(n log n) algorithm.

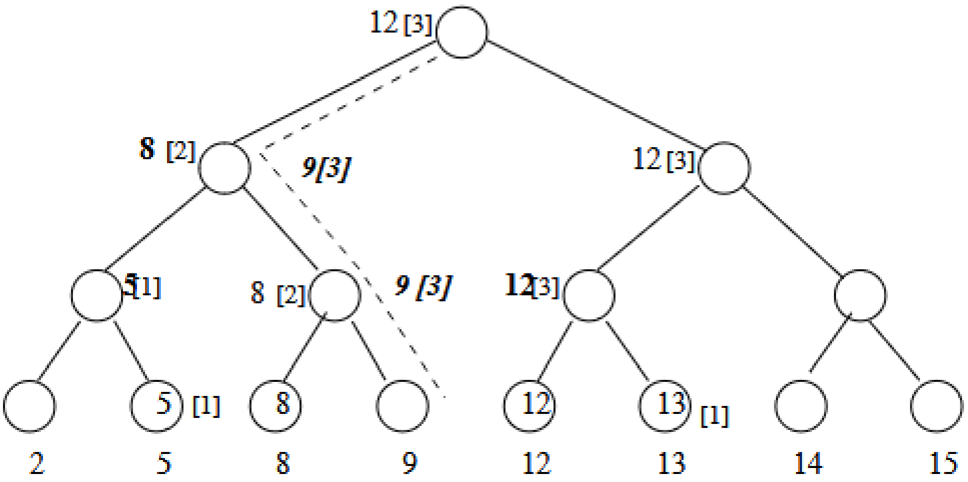


Figure 1

Input

1.Integer n: The number of points.
2.Array *X* (size *n*): The coordinates $x_1, x_2, \ldots, x_n$ (sorted in ascending order).

3.Array *L* (size *n*): The values $L_1, L_2, \ldots L_n$,associated with the coordinates.

Output

Integer: Length of the longest monotonic sequence.
Array: Indices of the points that form the sequence (1-based indexing).

Sample Input

n = 6

X = [2, 5, 8, 9, 12, 13]

L = [1, 2, 1, 3, 1, 1]

Sample Output

Longest Monotonic Sequence Length: 3

Sequence: [1, 2, 4]

ALCP 126 Matrix chain product

Given a chain (*A1, A2 . . . An*) of matrices where matrix A_i has dimensions p_{i-1} × p_i, we want to compute the product of the chain using minimum number of multiplications.

1. In how many ways can the matrices be multiplied?
2. Design an efficient algorithm that does not exhaustively use (1).

Input

1. An integer *n* (*n*≥2*n*), representing the number of matrices in the chain.
2. An array of integers *p*[] of size *n*+1, where p[*i*− *1*] and p[i] are the dimensions of the i^{th} matrix A_i(i.e., matrix A_i has dimensions *p[i*− *1*]×*p[i])*.

Output

1.Integer: An integer representing the minimum number of scalar multiplications required to multiply the entire chain of matrices.
2.(Optional) A string showing the optimal parenthesization of the matrix chain.

Sample Input	**Sample Output**
4	30000
10 20 30 40 30	$((A_1(A_2A_3))\ A_4)$

ALCP 127 Optimal BST

We are given a sequence $K = \{k_1, k_2 \ldots k_n\}$ of n distinct keys in sorted order with associated probability pi that the key ki will be accessed. Moreover, let q_i represent the probability that the search will be for a value (strictly) between k_i and k_{i+1}. So $\sum i\, p_i + \sum j\, q_j = 1$. How would you build the tree so as to optimise the expected search cost? (The more probable value should be closer to the root.)

Input

1.An integer n ($n \geq 1$), representing the number of distinct keys.
2.A list of n integers *K*[], representing the keys in sorted order.
3.A list of n floating-point numbers *P*[], where *P[i]* represents the probability pi that the key *K[i]* will be accessed.
4.A list of n+1 floating-point numbers Q[], where Q[j] represents the probability *qj* of searching for a value strictly between *K*[*j*−1] and *K*[*j*], with *Q*[0] being the probability of searching for a value before the first key.

Output

1.A floating-point number representing the minimum expected search cost of the optimal binary search tree.
2.The structure of the optimal BST, represented as a nested list, where each node is described as [key, left subtree, right subtree]. The left and right subtrees are recursively represented in the same format.

Sample Input

```
3
10 20 30
0.3 0.2 0.1
0.2 0.1 0.1 0.1
```

Sample Output

```
2.0
[20, [10, None, None], [30, None, None]]
```

ALCP 128 — Maximizing Taxi Driver's Profit problem

A taxi-driver has to decide about a schedule to maximize his profit based on an estimated profit for each day. Due to some constraint, he cannot go out on consecutive days. For example, over a period of 5 days, if his estimated profits are 30, 50, 40, 20, 60, then by going out on 1st, 3rd and 5th days, he can make a profit of 30+40+60 =130. Alternately, by going out on 2nd and 5th days, he can earn 110. First, convince yourself that by choosing alternate days (there are two such schedules) he won't maximize his profit. Design an efficient algorithm to pick a schedule to maximize the profit for an *n* days estimated profit sequence.

Input

1.An integer *n* (*n*≥1), representing the number of days.

2. A list of n integers *P*[], where *P*[*i*] represents the estimated profit on the *i*th day.

Output

1.A single integer representing the maximum profit the taxi driver can earn.
2.A list of integers representing the days the driver should work to achieve the maximum profit.

Sample Input	**Sample Output**
5	130
30 50 40 20 60	[1, 3, 5]

ALCP 129 Optimizing File Replication Across Servers

Suppose you want to replicate a file over a collection of n servers, labeled $S_1, \dots, S_n$. To place a copy of the file at server S_i results in a placement cost of ci for an integer $c_i > 0$.

Now if a user requests the file from server S_i, and no copy of the file is present at S_i, then the servers $S_{i+1}, S_{i+2}, \dots$ are searched in order until a copy of the file is finally found, say at server S_j , $j > i$. This results in an access cost of $j - i$. Thus, the access cost of S_i is 0 if S_i holds a copy of the file, otherwise it is $j - i$ where $j > i$ is the smallest integer greater than i such that S_j has a copy of the file. We will require that a copy of the file be always placed at S_n, the last server, so that all such searches terminate.

Now you are given the placement cost c_i for each server S_i. We would like to decide which servers should contain a copy of the file so that the sum of the placement cost and the sum of access costs for all the servers is minimized. Give an efficient algorithm which solves this problem.

Input

1. An integer n ($n \geq 1$), representing the number of servers.

2. A list of n integers *C[]*, where *C[i]* represents the placement cost c_i for server S_{i+1}.

Output

1. A single integer representing the minimum total cost.
2. A list of integers representing the indices of servers that should contain a copy of the file.

Sample Input	**Sample Output**
5	90
10 20 30 40 50	[1, 5]

ALCP 130 Finding the Longest Bitonic Subsequence

Bitonic sequence of numbers $x_1, x_2, x_3 \ldots x_k$ is such that there exists an i, $1 \le i \le k$ such that $x_1, x_2 \ldots x_i$ is an increasing sequence and $x_i, x_{i+1} \ldots x_n$ is a decreasing sequence. It is possible that one of the sequences is empty, i.e., strictly increasing (decreasing) sequences are also considered bitonic. For example, 3, 6, 7 ,5, 1 is a bitonic sequence where 7 is the discriminating number. Given a sequence of n numbers, design an efficient algorithm to find the longest bitonic subsequence. In 2, 4, 3 , 1, -10, 20, 8, the reader can verify that 2,3,20, 8 is such a sequence of length 4.

Input

1. An integer n ($n \geq 1$), representing the length of the sequence.

2. A list of n integers $A[]$, where $A[i]$ represents the i^{th} element of the sequence.

Output

1. An integer representing the length of the longest bitonic subsequence.
2. A list of integers representing the elements of the longest bitonic subsequence.

Sample Input	**Sample Output**
7	4
2 4 3 1 -10 20 8	[2, 3, 20, 8]

ALCP 131 Dart Game and Analysis

A dart game Imagine that an observer is standing at the origin of a real line and throwing n darts at random locations in the positive direction. At point of time, only the closest dart is visible to the observer.

(i) What is the expected number of darts visible to the observer?

(ii) Can you obtain a high-probability bound assuming independence between the throws?

(iii) Can you apply this analysis to obtain an O(n log n) expected bound on quicksort?

Input

1. An integer *n* (*n*≥1), representing the number of darts thrown

2. A list of n floating-point numbers D[], where D[i] represents the location of the i^{th} dart on the real number line.

Output

1. A floating-point number representing the expected number of visible darts.
2. A floating-point number representing the high-probability bound on the number of visible darts.
3. A proof or reasoning of how this analysis applies to establish an O(nlog n) expected runtime for quicksort.

Sample Input	Sample Output
5 1.2 2.3 0.9 3.5 0.5	2.5 4.1 "Using the dart visibility analysis, the expected number of comparisons in quicksort is proportional to the sum of distances, leading to an O(n log n) expected runtime."

ALCP 132 Deferred data structure

When we build a dictionary data structure for fast searching, we expend some initial overheads to build this data structure. For example, we need O(n log n) to sort an array so that we can do searching in O(log *n)* time. If there were only a few keys to be searched, then the preprocessing time may not be worth it since we can do brute force search in O(n) time which is asymptotically less that O(n log *n*).If we include the cost of preprocessing into the cost of searching then the total cost for searching k elements can be written as $\sum_{i=1}^{k} q(i) + P(k)$ where *q(i)* represents the cost of searching the i^{th} element and *P* (*k*) is the preprocessing time for the first k elements. For each value of *k*, balancing the two terms would give us the best performance. For example, for *k* = 1, we may not build any data structure but do brute force search to obtain *n*. As *k* grows large, say *n*, we may want to sort. Note that *k* may not be known in the beginning, so we may want to build the data structure in an incremental manner. After the first brute force search, it makes sense to find the median and partition the elements.

Describe an algorithm to extend this idea so as to maintain a balance between the number of keys searched and the preprocessing time.

Input

1. An integer *n* (*n*≥1), representing the total number of elements in the dictionary.

2. A list of *n* integers *A*[], representing the elements of the dictionary.

3. An integer *q* (*q*≥1), representing the number of search queries.

4.A list of *q* integers *S*[], representing the elements to be searched.

Output

1. A floating-point number representing the total cost of searching *q* elements while balancing preprocessing and search costs.
2. A list of steps showing the incremental building of the data structure and the search process.

Sample Input	**Sample Output**
7	22.5
[3, 8, 5, 1, 7, 10, 2] 4 [5, 7, 1, 10]	["Brute force search for 5", "Brute force search for 7", "Build median partition after 2 searches", "Binary search for 1 in partitioned data", "Binary search for 10 in partitioned data"]

ALCP 133 Latin Square

A Latin square of size n is an $n \times n$ table where each table entry is filled with one of the numbers $\{1, 2, . . . , n\}$. Further no row or column contains a number twice (and so, each number appears exactly once in each row and each column). For example, a Latin square of size 3 is shown below:

2	1	3
3	2	1
1	3	2

You are given a $k \times n$ table, where $k \leq n$. Again, each table entry is a number between 1 and n, and no row or column contains a number more than once. Show that this table can be extended to a Latin square, i.e., there is a Latin square of size n such that the first k rows of the Latin square are same as the rows of the given table.

Input

1. Two integers k,n ($1 \leq k \leq n$), representing the number of given rows and the size of the Latin square.

2. A $k \times n$ table T [], where $T[i][j]$ contains an integer between 1 and n, ensuring that no row or column contains a repeated number.

Output

1. An $n \times n$ times Latin square where the first k rows match the input table.
2. If no valid completion exists, return "No valid Latin square possible.

Sample Input

```
2 3
[[2, 1, 3],
 [3, 2, 1]]
```

Sample Output

```
[[2, 1, 3],
 [3, 2, 1],
 [1, 3, 2]]
```

ALCP 134 Valid Sudoku Completion

Sudoku is a 9×9 grid-based puzzle divided into 9 sub-grids (3×3 boxes). The objective is to fill the grid so that:

- Each row contains numbers 1 to 9 without repetition.
- Each column contains numbers 1 to 9 without repetition.
- Each 3×3 sub-grid contains numbers 1 to 9 without repetition.

Given a partially filled Sudoku grid, some cells are pre-filled with numbers (1 to 9), while others are empty (denoted by 0). Your task is to determine whether the Sudoku can be completed into a valid solution that satisfies all the above constraints. If a valid solution exists, return the completed grid. Otherwise, return "No valid Sudoku possible."

Constraints

- The input grid is always 9×9 in size.
- The given numbers in the grid are between 1 to 9 or 0 (empty).
- At least one empty cell exists in the grid.

Input

A 9×9 grid with numbers from 1 to 9 or empty cells represented as 0.

Output

1. A completed Sudoku grid if possible.
2. If no valid completion exists, return "No valid Sudoku possible."

Sample Input

```
[[5, 3, 0, 0, 7, 0, 0, 0, 0],
 [6, 0, 0, 1, 9, 5, 0, 0, 0],
 [0, 9, 8, 0, 0, 0, 0, 6, 0],
 [8, 0, 0, 0, 6, 0, 0, 0, 3],
 [4, 0, 0, 8, 0, 3, 0, 0, 1],
 [7, 0, 0, 0, 2, 0, 0, 0, 6],
 [0, 6, 0, 0, 0, 0, 2, 8, 0],
 [0, 0, 0, 4, 1, 9, 0, 0, 5],
 [0, 0, 0, 0, 8, 0, 0, 7, 9]]
```

Sample Output

```
[[5, 3, 4, 6, 7, 8, 9, 1, 2],
 [6, 7, 2, 1, 9, 5, 3, 4, 8],
 [1, 9, 8, 3, 4, 2, 5, 6, 7],
 [8, 5, 9, 7, 6, 1, 4, 2, 3],
 [4, 2, 6, 8, 5, 3, 7, 9, 1],
 [7, 1, 3, 9, 2, 4, 8, 5, 6],
 [9, 6, 1, 5, 3, 7, 2, 8, 4],
 [2, 8, 7, 4, 1, 9, 6, 3, 5],
 [3, 4, 5, 2, 8, 6, 1, 7, 9]]
```

If no valid Sudoku exists, output:
"No valid Sudoku possible."

ALCP 135 Completing a Partially Filled N-Queens Board

The *N*-Queens problem is a classic chess-based puzzle where the goal is to place *n* queens on an $n \times n$ chessboard such that:

- No two queens share the same row.
- No two queens share the same column.
- No two queens share the same diagonal (both main and anti-diagonal).

In this variation, you are given a partially filled board with some queens already placed at specific coordinates, and your task is to complete the board while maintaining the constraints of the *N*-Queens problem. Your solution must check whether the given queens' positions allow for a valid completion. If possible, return a valid board configuration with *n* queens placed. Otherwise, return "No valid *N*-Queens configuration possible."

Input

1. An integer *n* (where $4 \le n \le 12$).
2. A list of coordinates (r,c) where some queens are already placed.

Output

1. A valid $n \times n$ board with all *n* queens placed.
2. "No valid N-Queens configuration possible." otherwise.

Sample Input

```
4
[(1, 2), (3, 4)]
```

Sample Output

```
[['.', 'Q', '.', '.'],
 ['.', '.', '.', 'Q'],
 ['Q', '.', '.', '.'],
 ['.', '.', 'Q', '.']]
```

If no valid completion exists, output:
"No valid N-Queens configuration possible."

ALCP 136 Magic Square Transformation

A magic square is a 3×3 matrix where:

1. The sum of the elements in each row, each column, and both diagonals is the same constant value called the magic constant.
2. For a 3×3 magic square, this constant is always 15 when using the numbers from 1 to 9.

Given an arbitrary 3×3 matrix of integers, your task is to determine the minimum transformation cost required to convert it into a magic square. The transformation cost is defined as the sum of the absolute differences between the elements in the original matrix and the corresponding elements in the target magic square.

Input

A 3×3 matrix containing arbitrary integers.

Output

Print the minimum transformation cost to convert the given matrix into a magic square.

Sample Input	Sample Output
[[4, 8, 7], [3, 9, 6], [1, 5, 2]]	15

ALCP 137 | Smart Warehouse Item Retrieval

A smart warehouse contains a grid-based layout where each cell represents a location in the warehouse. Some cells contain items that need to be retrieved. The goal is to determine the shortest retrieval path that collects all required items in the minimum distance from a given starting position. The movement is allowed in four directions (up, down, left, right), and obstacles may exist, making some paths inaccessible. Using a search with multiple goals*, find the optimal route that minimizes travel distance while ensuring all items are collected.

Input

1. Two integers *n* and *m* — are the dimensions of the warehouse grid.
2. An *n*×*m* matrix representing the warehouse layout:
   - '.' represents an open space.
   - '#' represents an obstacle.
   - '*S*' represents the starting position.
   - '*I*' represents an item that needs to be collected.

Output

1. A single integer represents the minimum distance required to collect all items.
2. If it is impossible to collect all items, return -1.

Sample Input

```
5  5
S . #  .  I
. #  .  .  .
. I  #  .  .
.  .  .  # .
I  .  .  . I
```

Sample Output

```
Minimum Retrieval Distance: 12
```

ALCP 138 — Merging Two Upper Hulls

Design an algorithm to merge two upper hulls in O(n) time where n is the sum of the vertices in the two hulls. Further show how to find a bridge between two separable upper hulls in O(log n) steps using some variation of the binary search. Note that the bridge is incident on the two hulls as a tangents on points p_1 and p_2. Use binary search to locate these points on the two convex hulls by pruning away some fraction of points on at least one of the hulls.

Input

1.An integer n_1 ($n_1 \geq 1$), representing the number of points in the first convex hull.

2.A list of $n_1$tuples H_1[], each containing two floating-point numbers representing the x and y coordinates of the points in the first hull, sorted by increasing x-coordinate.

3.An integer n_2 ($n_2 \geq 1$), representing the number of points in the second convex hull.

4.A list of n_2 tuples H_2[], each containing two floating-point numbers representing the x and y coordinates of the points in the second hull, sorted by increasing x-coordinate.

Output

1.A tuple (p_1, p_2), where p_1 and p_2 are the coordinates of the points forming the upper bridge between the two hulls.

2.A list of tuples representing the merged upper hull, sorted by increasing x-coordinate.

Sample Input	**Sample Output**
3	Upper Bridge: ((2, 5), (4, 6))
(1, 3) (2, 5) (3, 4)	
3	
(4, 6) (5, 3) (6, 1)	Merged Upper Hull: [(1, 3), (2, 5), (4, 6), (5, 3), (6, 1)]

ALCP 139 Dynamic Maintenance of Convex Hull

If we want to maintain a convex hull under arbitrary insertion and deletion of points without recomputing the entire hull, we can use the following ap-proach. Use a balanced BST to store the current set of points. At each of the internalnodes, store the bridge between the upper hull of the points stored in the left and right subtrees. Use this scheme recursively - the leaves store the original points. In case of any changes to the point set the bridges may be re-computed using the algorithm in the previous problem. Provide all the missing details to show that this data structure can be maintained in O(log^2) time per update, either insertion or deletion.

Input

1 An integer *q* (*q*≥1), representing the number of operations.

2. A list of *q* operations, where each operation is one of the following:

- "INSERT *x y*": Insert a point (*x*,*y*) into the convex hull.
- "DELETE *x y*": Remove the point (*x*,*y*) from the convex hull.

Output

1. After each operation, output the updated convex hull in sorted order of *x*-coordinates.

2. The convex hull should be efficiently maintained using the BST-based approach.

Sample Input

```
5
INSERT 1 3
INSERT 2 5
INSERT 3 4
DELETE 2 5
INSERT 4 6
```

Sample Output

```
Convex Hull After Operation 1: [(1, 3)]
Convex Hull After Operation 2: [(1, 3), (2, 5)]
Convex Hull After Operation 3: [(1, 3), (2, 5), (3, 4)]
Convex Hull After Operation 4: [(1, 3), (3, 4)]
Convex Hull After Operation 5: [(1, 3), (3, 4), (4, 6)]
```

ALCP 140 | Finding All Intersecting Pairs

Given a set *S* of *n* line segments, design an efficient algorithm that outputs all intersecting pairs of segments. Note that the answer can range from 0 to(2^n), so it makes sense to design an output sensitive algorithm whose running time is proportional to the number of intersecting pairs.

Hint: You may want to target a running time of O((*n* + *h*) log n) algorithm where *h* is the number of intersecting pairs.

Input

1. An integer *n* (*n*≥1), representing the number of line segments.

2.A list of *n* tuples, where each tuple represents a line segment as:

(x_1, y_1, *x2,y2*) → denoting a segment from point (x_1,y_1) to (x_2,y_2).

Output

1. An integer *h*, representing the number of intersecting pairs.

2. A list of *h* intersecting pairs, where each pair is represented as:

((x_1, y_1,x_2,y_2),(x_3,y_3,x_4,y_4)), denoting the two intersecting segments.

Sample Input

```
4
1 1 4 4
2 3 5 2
3 1 6 3
1 4 4 2
```

Sample Output

```
2
((1, 1, 4, 4), (2, 3, 5, 2))
((2, 3, 5, 2), (3, 1, 6, 3))
```

ALCP 141 Escape the Maze

In this problem, you are given a maze represented by an $m \times n$ grid, where each cell can be one of the following types:

- # represents a wall, which blocks movement.
- . represents an empty space, which is traversable.
- *S* represents the starting point, where you begin.
- *E* represents the exit, which is your destination.

The goal is to navigate from the starting point *S* to the exit *E* using the shortest path possible. If there are multiple paths with the same length, you can return any one of them. If no path exists from *S* to *E*, return "No Exit."

The maze can have various obstacles and open spaces that require navigating through them efficiently. The problem is a classic application of the shortest path in a grid, where you need to explore the grid while avoiding walls and trying to find the quickest route to the exit.

Input

1. Two integers *m* and *n* ($2 \leq m,n \leq 100$) representing the maze dimensions.
2. An $m \times n$ grid with the elements '*S*', '*E*', '#', and '.'

Output

1. A list of directions ('UP', 'DOWN', 'LEFT', 'RIGHT') representing the shortest path.
2. "No Exit" if no path exists.

Sample Input

```
5 5 #####
#S..#
#.###
#..E# #####
```

Sample Output

```
'DOWN', 'DOWN', 'RIGHT', 'RIGHT', 'UP'
```

ALCP 142 | Weighted Word Chain

In this problem, you are given a dictionary of words, each associated with a weight. The goal is to form a chain of words where the last letter of one word matches the first letter of the next word. The challenge is to maximize the total weight of the chain while adhering to the rule that each word can only be used once.

For example, given the list of words:

- apple (weight 3),
- elephant (weight 5),
- tiger (weight 2),
- rat (weight 4),
- ant (weight 6),

You need to create a sequence of words such that the last letter of a word matches the first letter of the next word. Additionally, the sum of the weights of the words used in the chain should be maximized.

The key task is to find a way to select words that can form a valid sequence and simultaneously maximize the total weight. In this case, the sequence could be something like "apple" -> "elephant" -> "tiger" -> "rat", where the last letter of each word matches the first letter of the next, and the total weight of the chain (3 + 5 + 2 + 4) is maximized.

Input

1. An integer 1≤*k*≤10,000) representing the number of words.
2. A list of tuples (word,weight), where word is a string and weight is an integer.

Output

1. The maximum total weight of the chain.
2. The chain of words used**.**

Sample Input	**Sample Output**
5 [("apple", 3), ("elephant", 5), ("tiger", 2), ("rat", 4), ("ant", 6)]	Maximum Weight: 14Chain: ["apple", "elephant", "tiger", "rat"]

ALCP 143

Land Division Optimization

In this problem, you are given a rectangular piece of land with dimensions $w \times h$ (width × height) and are tasked with dividing it into exactly k regions. The challenge is to ensure that each of these regions has equal area, and the divisions can only be made either horizontally or vertically. Additionally, each resulting region must remain a rectangle.

The problem is a geometric optimization problem where the goal is to determine the correct dimensions for the k regions, such that:

1. **Equal Area**: Each of the k regions must have the same area. This means the total area of the land ($w * h$) must be divisible by k. If this is not the case, it's impossible to divide the land into exactly k regions of equal area.
2. **Horizontal or Vertical Divisions**: The land can only be divided along horizontal or vertical lines. This means that all boundaries between the regions must be straight lines either horizontally or vertically, resulting in rectangular regions.
3. **Rectangular Regions**: Each region formed after division must remain a rectangle. This limits the ways in which divisions can be made, as the division must align with the grid formed by the width w and height h.

The challenge lies in determining if it's possible to divide the land into k regions, and if so, finding the exact dimensions of each region.

Input

1. Two integers w and h ($1 \le w, h \le 104$).
2. An integer k ($1 \le k \le 1001$).

Output

1. A list of dimensions for the k regions if possible.
2. "Division not possible" otherwise.

Sample Input	**Sample Output**
4 4 4	Regions: [(2, 2), (2, 2), (2, 2), (2, 2)]

ALCP 144 — Robo-Arena

In this problem, you are tasked with designing a competitive game where two robots navigate an *n*×*n* grid, referred to as the arena. The grid contains several elements:

- *R* represents rewards that the robots aim to collect, with each reward giving them 1 point.
- *T* represents traps, which block the robots' movement.
- . represents open spaces where the robots can move freely.

Each robot has a starting position on the grid, and they take turns moving within the grid. The game ends when both robots are unable to move due to the obstacles (traps or the boundaries of the grid). The objective is to determine the scores of both robots, which are the total number of rewards collected, and to track their movements within the arena.

Input

1. An integer *n* (5≤n≤20).
2. A grid with:
   - *R* for rewards (1 point each),
   - *T* for traps,
   - . for open spaces.
3. Starting positions of the two robots.

Output

1. The scores of both robots.
2. The paths taken by each robot.

Sample Input

```
5
......R.T..T.R.
..T..
.....
(1, 1), (5, 5)
```

Sample Output

```
Robot 1 score: 2, Path: [(1, 1), (2, 2)] Robot 2 score: 1,
Path: [(5, 5), (4, 4)]
```

ALCP 145 | The Matrix Illumination

You are given an *n*×*n* grid, where some cells contain light sources. Each light source illuminates the entire row, column, and both diagonals it resides on. Your task is to determine the following:

1. Which cells are illuminated (represented as 1).
2. The number of cells that remain unilluminated (represented as 0).
3. The placement of additional light sources required to ensure the entire grid is illuminated. The objective is to use the minimum number of additional light sources.

For example, if a light source is placed in position (*i, j*), it will illuminate all cells in row *i*, column *j*, and both diagonals that intersect at (*i, j*).

You are tasked with computing the final grid after determining which cells are illuminated by the existing light sources, counting the unilluminated cells, and finding the minimum number of additional light sources required to illuminate the whole grid.

Input

1. An integer *n* (1 ≤ *n* ≤ 1000) representing the size of the grid.
2. A list of tuples (*r, c*) where each tuple represents the coordinates of a light source, where *r* and *c* are 1-based indices indicating the position of the light source on the grid. The total number of light sources can range from 0 to n^2.

Output

1. A matrix of size *n*×*n* where:

- Cells illuminated by light sources are marked as 1.
- Cells that remain unilluminated are marked as 0.
- The number of unilluminated cells in the grid.

2. The list of coordinates where additional light sources need to be placed in order to fully illuminate the grid, using the minimum number of new light sources.

Sample Input	Sample Output

n = 5
light_sources = [(1, 1), (3, 3)]
	Illuminated Grid:1 1 1 1 11 0 0 0 11 0 1 0 11 0 0 0 11 1
	1 1 1
	Unilluminated Cells: 7
	Additional Light Sources Required: [(2, 2), (4, 4)]

ALCP 146 | Friendship Graph

A Friendship Graph represents social connections between individuals, where each person is a node, and a friendship between two people is an edge. Given a friendship network with *n* people and *m* friendships, the goal is to optimize connectivity while minimizing disruptions when people leave or form new friendships.

You need to design an efficient algorithm to analyze and optimize the friendship graph based on the following complex problems:

Critical Friend Detection:

Find the most influential people (nodes) whose removal would disconnect the network significantly.

Use this to identify key social influencers.

Friendship Recommendation System:

Suggest new connections to maximize closeness centrality (shortest paths between friends).

Maximum Social Circle:

Identify the largest group of friends who are all connected directly or indirectly.

Fastest Information Spread:

Find the minimum number of people who should receive a message first so that everyone in the network gets the information in the shortest time.

Input

- An integer *n*: the number of people in the network.
- An integer mmm: the number of friendships.
- A list of mmm pairs (*u, v*) meaning there is a friendship between person *u* and person *v*.

Output

- A list of critical people whose removal would disconnect the network.
- A list of recommended new friendships for each person.
- The largest connected friend group.
- The minimum set of people to start spreading a message for the fastest reach.

Sample Input

n = 6
m = 5
friendships = [(1, 2), (2, 3), (3, 4), (4, 5), (5, 6)]

Sample Output

Critical People: [3, 4]

Recommended Friendships: [(1, 3), (2, 4), (3, 5), (4, 6)]
Largest Friend Group: [1, 2, 3, 4, 5, 6]

Minimum Message Spreaders: [3]

ALCP 147 — Circuit Board Placement

In designing printed circuit boards (PCBs), the primary challenge is to route electrical connections between different components efficiently while minimizing space, avoiding overlaps, and reducing manufacturing costs.

Given a circuit board with multiple components and pre-defined connection points, the goal is to determine an optimal wire routing strategy while satisfying the following constraints:

1. Minimize Wire Length – Reduce the total length of the wires used to connect components.
2. Avoid Overlapping Wires – Ensure that no two wires cross over each other.
3. Use Minimum Number of Routing Layers – Optimize for multi-layer circuit boards.
4. Minimize the Number of Turns (Bends) – Each turn increases resistance and manufacturing complexity.

Input

- An integer *n* — the number of components (nodes).
- An integer *m* — the number of required connections (edges).
- A list of m pairs (*u, v*) — representing required connections between component *u* and *v*.
- A 2D Grid Representation of PCB Layout — defining obstacles, wire paths, and available layers.

Output

- A set of non-overlapping paths (wires) connecting all required components with minimized length and bends.
- Total wire length used in the layout.
- Whether the routing is planar or requires additional layers.

Sample Input

```
Input:
n = 5
m = 4
connections = [
    (1, 2),
    (2, 3),
    (3, 4),
    (4, 5)
]

PCB Grid:
0  0  0  0  0
0  1  0  1  0
0  0  0  0  0
0  1  0  1  0
0  0  0  0  0
```

Sample Output

```
Wire Paths: [(1 → 2), (2 → 3), (3 → 4), (4 → 5)]

Total Wire Length: 8 units

Planar Routing: Yes
```

ALCP 148 Finding All Intersecting Pairs

Given a set S of n line segments, design an efficient algorithm that outputs all intersecting pairs of segments. Note that the answer can range from 0 to(2^n), so it makes sense to design an output sensitive algorithm whose running time is proportional to the number of intersecting pairs. Hint: You may want to target a running time of O($(n + h)$ log n) algorithm where h is the number of intersecting pairs.

Input

1. An integer n ($n \geq 1$), representing the number of line segments.

2.A list of n tuples, where each tuple represents a line segment as:

(x_1,y_1,x_2,y_2) → denoting a segment from point (x_1,y_1) to (x_2,y_2).

Output

An integer h, representing the number of intersecting pairs.

2. A list of h intersecting pairs, where each pair is represented as:

$((x_1,y_1,x_2,y_2),(x_3,y_3,x_4,y_4))$, denoting the two intersecting segments.

Sample Input

```
4
1 1 4 4
2 3 5 2
3 1 6 3
1 4 4 2
```

Sample Output

```
2
((1, 1, 4, 4), (2, 3, 5, 2))
((2, 3, 5, 2), (3, 1, 6, 3))
```

ALCP 149 — The Second Shortest Path Problem

In a weighted graph with non-negative edge weights, the goal is to find the second shortest path between two vertices *u* and *v*. The second shortest path must satisfy the following conditions:

1. It must be a valid path from *u* to *v*.
2. It must differ from the shortest path by at least one edge.
3. It may have the same total weight as the shortest path but cannot be identical in edge composition.

This problem is useful in network routing, transportation planning, and alternative pathfinding, where a backup route is needed in case of congestion or failure.

Input

The input consists of multiple lines:

- The first line contains two integers *N* (number of vertices) and *M* (number of edges).
- The next *M* lines contain three integers *u*, *v*, *w*, representing an edge between *u* and *v* with weight *w*.
- The last line contains two integers *s* and *t*, representing the start and destination vertices.

Output

Output a single line containing the second shortest path weight from s to t. If no such path exists, output -1.

Sample Input

```
5 6
1 2 10
1 3 5
2 3 2
2 4 1
3 4 9
4 5 3
1 5
```

Sample Output

```
14
```

ALCP 150

Determining Tournament Victory Using Maximum Flow

There are n teams playing in a tournament. Each pair of teams will play each other exactly k times. So far p_{ij} games have been played between every pair of teams i and j. Assume that in every game, one of the team wins (draw is not an option). You would like to know if there is any possible scenario in which your favourite team, team 1, can have more wins than any other team at the end of the tournament. In other words, you would like to know if you could decide on the winner of every remaining game, then is it possible that team 1 ends up with more wins than any other team. Show how you can formulate this problem as a maximum flow problem.

Input

The input consists of multiple lines:

1. An integer n ($n \geq 2$) — the number of teams.

2.An integer k ($k \geq 1$) — the number of times each pair of teams plays each other.

3.A $n \times n$ matrix $P[]$, where $P[i][j]$ represents the number of games already played between team i and team j.

4.A list $W[]$, where $W[i]$ represents the current number of wins for team i.

Output

A boolean value (True or False), indicating whether team 1 can possibly end with more wins than any other team.

A list of match assignments (if possible) showing a valid allocation of remaining game results.

Sample Input

```
3
2
[[0, 1, 1],
 [1, 0, 2],
 [1, 2, 0]]
[2, 3, 1]
```

Sample Output

```
True
["Assign win to team 1 in (1,2)",   "Assign win to
team 1 in (1,3)",   "Assign win to team 3 in (2,3)"]
```