



INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal - 500 043, Hyderabad, Telangana

Department of Computer Science and Engineering

Knowledge and Skill Assignment	:	Complex Problem Solving (CPS)
Number of Problem Statements	:	150
Course Code	:	ACSD08
Course Name	:	Data Structures
Semester	:	B.Tech III 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 solutions (CA)	Support sustainable development solutions by ensuring functional requirements, minimize environmental 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.	

DSCP Data Structures Manual

Problem solving is an essential part of every scientific discipline. To solve a given problem by using a computer, one needs to write a program for it. A program consists of two components, algorithm and data structure. For efficient problem solving, one needs to select a combination of algorithms and data structures that provide maximum efficiency. Data structures give a thorough understanding in the theories and application of computer algorithms, abstract data types, underlying data structures and their integration to produce efficient programmes.

From the data structures point of view, following are some important categories of algorithms:

- Search: Algorithm to search an item in a data structure.
- Sort: Algorithm to sort items in a certain order.
- Insert: Algorithm to insert item in a data structure.
- Update: Algorithm to update an existing item in a data structure.
- Delete: Algorithm to delete an existing item from a data structure.

Knowledge of data structures helps to develop the skills needed to analyse problems and then design, implement, and analyse, effective algorithmic solutions. These are fundamental to computer science and software development. Their efficient use allows for better performance, scalability, and resource management in various applications.

Applications of Data Structures

1. Social Media Platforms

Search engines like Facebook, Twitter, and Instagram use priority queues to rank posts based on relevance and engagement. Graph-based algorithms (BFS, DFS) analyse mutual friends and user behaviour to suggest connections. WhatsApp and Messenger use queues for real-time message processing and hash tables for quick retrieval.

2. Cybersecurity and Blockchain

Firewalls and IDS systems use bloom filters to detect malicious IPs. Bitcoin and Ethereum use Merkle trees for efficient transaction validation.

3. Navigation and GPS Systems

Google Maps and Uber use Dijkstra's or A* algorithm for real-time route optimization. Live traffic updates use heaps for congestion analysis and hash tables for quick lookup.

4. Online Shopping and E-Commerce

Amazon and Flipkart use collaborative filtering (graph-based) to recommend products. FIFO queues manage customer orders, while stacks help with last-minute undo operations. Warehouses use heaps for priority restocking and BSTs for product lookup.

5. Banking and Financial Services

Banks use graph analysis to detect fraudulent transactions and unusual patterns. ATMs process transactions in a queue, while stack structures handle undo operations. Trading platforms use time-series data structures for real-time price tracking.

6. Healthcare and Medical Systems

Emergency room queues prioritize critical patients using min-heaps. Bioinformatics applications use suffix trees for fast genetic sequence analysis. Quick lookup and storage of patient records using hash tables and B-Trees.

Table of Contents

S No.	Name of the Problem
DSCP 1	The Word Ladder Problem
DSCP 2	The Knight's Tour Problem
DSCP 3	Sierpinski Triangle
DSCP 4	Hot Potato
DSCP 5	Printing Tasks
DSCP 6	Randomized Hashing Analysis
DSCP 7	The Lost Treasure map
DSCP 8	Grade Report
DSCP 9	Book Management
DSCP 10	Merge Point of two Intersecting Singly Linked Lists
DSCP 11	Maximum Sum in Sliding Window
DSCP 12	Circular Queue and Element Counting
DSCP 13	Queue to Stack Transfer with Order Preservation
DSCP 14	Check Consecutive Pairs in a Stack
DSCP 15	Interleave Two Halves of a Queue
DSCP 16	Mirror Tree
DSCP 17	Zigzag Tree Traversal
DSCP 18	Vertical Sum of Binary Tree
DSCP 19	Isomorphic Tree
DSCP 20	Quasi-isomorphic Trees
DSCP 21	Lowest Common Ancestor in a BST
DSCP 22	Floor and Ceiling in a BST
DSCP 23	Counting Nodes in the Range [a, b]
DSCP 24	BST Inorder Transformation
DSCP 25	Pruning a BST
DSCP 26	Balanced BST Over Complete Binary Tree for Data Storage
DSCP 27	Josephus Problem

DSCP 28	Move-to-front (MTF)
DSCP 29	Stack Generability
DSCP 30	Binary Search with only Addition and Subtraction
DSCP 31	Throwing Eggs from a Building
DSCP 32	Hot or Cold
DSCP 33	Random Connections
DSCP 34	Sublinear Extra Space
DSCP 35	Nuts and Bolts
DSCP 36	Median-of-5 Partitioning
DSCP 37	Priority Queue with Explicit Links
DSCP 38	Counting White Regions in an $n \times n$ Grid
DSCP 39	Reversed and Mirrored Tree Traversals
DSCP 40	Binary Tree Sorting
DSCP 41	Software Store Management
DSCP 42	Latin-to-English Vocabulary Reversal
DSCP 43	Cross-Reference Generator
DSCP 44	Digital Trees for Spell Checking
DSCP 45	Analysis of Tournaments in Directed Graphs
DSCP 46	Quick-Fit Memory Allocation with Block Coalescing
DSCP 47	Efficient Substitute Teacher Allocation Using BST
DSCP 48	Hash Function Efficiency with Linear Probing
DSCP 49	Airline Ticket Reservation System
DSCP 50	Linked List-Based Line Editor
DSCP 51	High Score Entries for a Video Game
DSCP 52	Tic-Tac-Toe
DSCP 53	Out-of-Order Packets
DSCP 54	Matching Tags in a Markup Language
DSCP 55	Optimizing Alice's Chances in a Game
DSCP 56	Crossing a bridge

DSCP 57	Card Hand
DSCP 58	Air-Traffic Control Simulation
DSCP 59	Top log(n) Frequent Flyers
DSCP 60	Stock Trading System
DSCP 61	Unmonopoly Game
DSCP 62	CPU Job Scheduler
DSCP 63	Counting the Number of Occurrences of Words in a Document
DSCP 64	Flight Query System
DSCP 65	Spell Checker
DSCP 66	Maximum Non-Overlapping Daily Jobs Scheduling
DSCP 67	Building Bridge
DSCP 68	Video-phone system
DSCP 69	Telephone network
DSCP 70	Build a computer network
DSCP 71	Maximum Bottleneck Path Problem
DSCP 72	NASA's Communication Network
DSCP 73	Sir Paul' s Treasure Hunt
DSCP 74	Flight Scheduling
DSCP 75	Karen's algorithm
DSCP 76	Balanced Load Distribution in Server Cluster
DSCP 77	Secure File Storage
DSCP 78	Fraud Detection in Banking
DSCP 79	URL Shortener
DSCP 80	Anagram Detection
DSCP 81	Length of connected cells
DSCP 82	Merging Point of Two Linked Lists
DSCP 83	Reverse the Linked List in Pairs
DSCP 84	Last Survivor in a Circular Elimination Game
DSCP 85	Modular node

DSCP 86	Reverse Blocks of K Nodes
DSCP 87	Implementing Three Stacks in One Array
DSCP 88	Stock Span
DSCP 89	Largest rectangle under histogram
DSCP 90	Snake or Snail
DSCP 91	Fractional Node in a Singly Linked List
DSCP 92	Median in an Infinite Series of Integers
DSCP 93	Addition of Two Linked Lists Representing Large Numbers
DSCP 94	Stack with O(1) Minimum Retrieval
DSCP 95	Valid Stack Permutations
DSCP 96	Intersection of Two Linked Lists using Stacks
DSCP 97	Stacks of Flapjacks
DSCP 98	Checking Consecutive Pairs
DSCP 99	Minimum Halls for Event Scheduling
DSCP 100	Interleaving the First and Second Half of a Queue
DSCP 101	Water Pouring
DSCP 102	Bipartite Check for Professional Wrestlers' Rivalries
DSCP 103	Second-Best Minimum Spanning Tree
DSCP 104	Professor Adam's Children's School Routes
DSCP 105	Escape Problem in an n x n Grid
DSCP 106	Poll Analysis for Hit Parade Selection
DSCP 107	Verifying Connectivity in a One-Way Street Network
DSCP 108	Maximizing Expected Profit for Yuckdonald's Restaurant
DSCP 109	Pirate Division Problem: Strategic Distribution of Wealth
DSCP 110	Vito's Family
DSCP 111	ShoeMaker's Problem
DSCP 112	Currency Arbitrage Detection Using Exchange Rates
DSCP 113	Freckles
DSCP 114	The Hacker's Encrypted Filesystem

DSCP 115	The Infinite Scroll Newsfeed
DSCP 116	The Maximum Flow of the River
DSCP 117	The Enchanted Spellbook Sorting
DSCP 118	The Task Scheduler
DSCP 119	The Grand Library Search
DSCP 120	The Maze of the Graph King
DSCP 121	Parentheses Matching in Code Compilation
DSCP 122	The City of Balanced Skyscrapers
DSCP 123	The Enchanted Spellbook Sorting
DSCP 124	The AI Traffic Control System
DSCP 125	The Hacker's Cache
DSCP 126	The Haunted Castle's Door Puzzle
DSCP 127	The Kingdom's Road Network
DSCP 128	Professor Diogenes' Chip Testing
DSCP 129	Jug Pairing Problem with Minimum Comparisons
DSCP 130	Compactifying a Doubly Linked List
DSCP 131	Longest-Probe Bound for Hashing in Open-Addressed Hash Tables
DSCP 132	Professor Bunyan's Binary Search Tree
DSCP 133	Validating Red-Black Tree Insert Fix-Up Stability
DSCP 134	The Bitonic Euclidean Traveling Salesman
DSCP 135	Validating Red-Black Tree Insert Fix-Up Stability
DSCP 136	The Bitonic Euclidean Traveling Salesman
DSCP 137	Filtering
DSCP 138	Circular Rotations of a String
DSCP 139	Stack Pop Sequences
DSCP 140	Radix Sort Simulation using Queues
DSCP 141	Missionaries and Cannibals
DSCP 142	Optimal Change Maker
DSCP 143	Birthday Paradox

DSCP 144	Expression Tree Representation
DSCP 145	Finding the k-th Smallest Element in a Binary Search Tree
DSCP 146	Finding Floor and Ceiling in a Binary Search Tree
DSCP 147	Convert BST by Replacing Subtree Size with Inorder Previous Node Data
DSCP 148	Traffic Congestion Analysis
DSCP 149	Airline Ticket Reservation System
DSCP 150	Scalable Ride-Sharing System

DSCP 01**The Word Ladder**

Consider the puzzle called a word ladder. Transform the word "FOOL" into the word "SAGE". In a word ladder puzzle you must make the change occur gradually by changing one letter at a time. At each step you must transform one word into another word, you are not allowed to transform a word into a non-word. The word ladder puzzle was invented in 1878 by Lewis Carroll, the author of Alice in Wonderland. The following sequence of words shows one possible solution to the problem posed above.

FOOL

POOL

POLL

POLE

PALE

SALE

SAGE

There are many variations of the word ladder puzzle. For example you might be given a particular number of steps in which to accomplish the transformation, or you might need to use a particular word. Solve this problem using a graph algorithm.

Represent the relationships between the words as a graph.

Use the graph algorithm known as breadth first search to find an efficient path from the starting word to the ending word.

Input

The first line contains two words:

start_word (string) – The word to start the transformation from.

end_word (string) – The word to transform into.

The second line contains an integer N ($1 \leq N \leq 10^5$) – The number of words in the dictionary.

The next N lines contain N valid words (each of the same length as start_word and end_word).

Output

Print the minimum number of transformations required to reach the end_word from start_word.

If it is not possible, print -1.

Sample Input

FOOL SAGE

6

POOL

POLL

POLE

PALE

SALE

SAGE

Sample Output

6

HIT COG

-1

5

HOT

DOT

DOG

LOT

LOG

DSCP 02**The Knight's Tour**

The knight's tour puzzle is played on a chess board with a single chess piece, the knight. The object of the puzzle is to find a sequence of moves that allow the knight to visit every square on the board exactly once. One such sequence is called a "tour." The knight's tour puzzle has fascinated chess players, mathematicians and computer scientists alike for many years. The upper bound on the number of possible legal tours for an eight-by-eight chessboard is known to be; however, there are even more possible dead ends. Clearly this is a problem that requires some real brains, some real computing power, or both.

The knight's tour is a puzzle played on a chessboard using a single knight. The goal is to move the knight such that it visits every square on the board exactly once. A sequence of moves satisfying this condition is called a "tour". Given a starting position on an $N \times N$ chessboard, find a valid knight's tour or determine if no tour exists.

Represent the legal moves of a knight on a chessboard as a graph.

Use a graph algorithm to find a path of length where every vertex on the graph is visited exactly once.

Input

The first line contains an integer N ($1 \leq N \leq 8$) - The size of the chessboard.

The second line contains two integers r and c ($0 \leq r, c < N$) - The row and column index of the starting position of the knight.

Output

If a knight's tour exists, print an $N \times N$ matrix where each cell represents the step number in which the knight visits that cell.

If no valid tour exists, print -1.

Sample Input

5
0 0

3
0 0

Sample Output

0 3 16 9 20
15 8 1 18 5
4 21 10 7 12
11 6 13 24 19
22 17 2 23 14

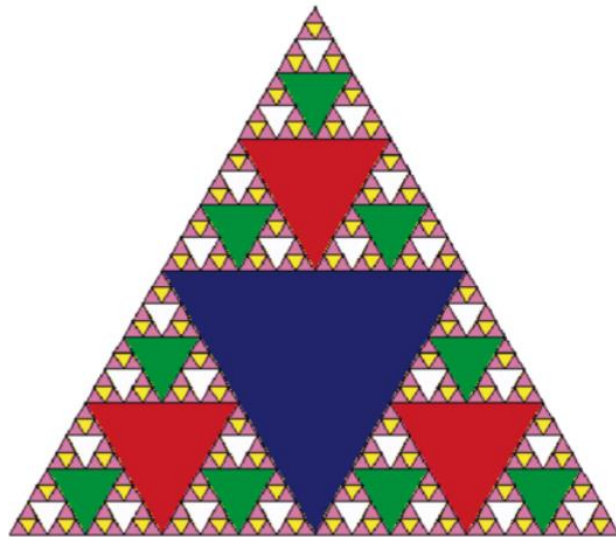
-1

DSCP 03**Sierpinski Triangle**

The Sierpinski triangle illustrates a three-way recursive algorithm. The procedure for drawing a Sierpinski triangle by hand is simple. Start with a single large triangle. Divide this large triangle into three new triangles by connecting the midpoint of each side. Ignoring the middle triangle that you just created, apply the same procedure to each of the three corner triangles. Each time you create a new set of triangles, you recursively apply this procedure to the three smaller corner triangles. You can continue to apply this procedure indefinitely if you have a sharp enough pencil. Before you continue reading, you may want to try drawing the Sierpinski triangle yourself, using the method described.

- The Sierpinski Triangle is a fractal that demonstrates a three-way recursive pattern. It is constructed as follows:
- Start with an equilateral triangle.
- Divide it into four smaller equilateral triangles by connecting the midpoints of its sides.
- Remove the central triangle.
- Recursively apply the same procedure to the three remaining corner triangles.
- Continue this process up to a given recursion depth.

Given a recursion depth N , generate the Sierpinski Triangle as ASCII art.

**Input**

A single integer N ($0 \leq N \leq 5$) - The recursion depth.

Output

A triangular ASCII representation of the Sierpinski Triangle of depth N .

The triangle consists of * (asterisks) for filled areas and spaces for empty areas.

Sample Input

0

1

Sample Output

*

*

* *

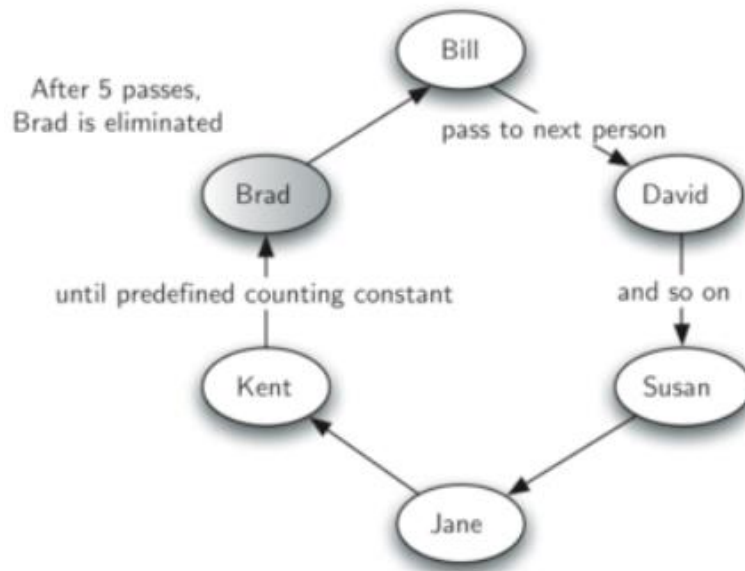
2

*
* *
* *
* * * *

3

*
* *
* *
* * * *
* * *
* * * *
* * * *
* * * * *

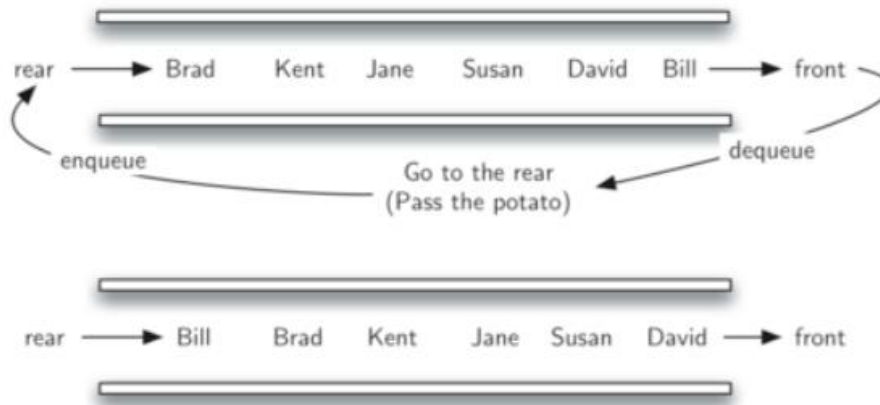
Consider the children's game Hot Potato. In this game children line up in a circle and pass an item from neighbour to neighbour as fast as they can. At a certain point in the game, the action is stopped and the child who has the item (the potato) is removed from the circle. Play continues until only one child is left.



This game is a modern-day equivalent of the famous Josephus problem. Based on a legend about the famous first-century historian Flavius Josephus, the story is told that in the Jewish revolt against Rome, Josephus and 39 of his comrades held out against the Romans in a cave. With defeat imminent, they decided that they would rather die than be slaves to the Romans. They arranged themselves in a circle. One man was designated as number one, and proceeding clockwise they killed every seventh man. Josephus, according to the legend, was among other things an accomplished mathematician. He instantly figured out where he ought to sit in order to be the last to go. When the time came, instead of killing himself, he joined the Roman side. You can find many different versions of this story. Some count every third man and some allow the last man to escape on a horse. In any case, the idea is the same.

We will implement a general simulation of Hot Potato. Our program will input an array of names and a constant, call it "num," to be used for counting. It will return the name of the last person remaining after repetitive counting by num.

To simulate the circle, we will use a queue. Assume that the child holding the potato will be at the front of the queue. Upon passing the potato, the simulation will simply dequeue and then immediately enqueue that child, putting her at the end of the line. She will then wait until all the others have been at the front before it will be her turn again. After num dequeue/enqueue operations, the child at the front will be removed permanently and another cycle will begin. This process will continue until only one name remains (the size of the queue is 1).



The Hot Potato game is a simulation of a well-known counting game. A group of children stand in a circle and pass an item (the "potato") around as quickly as possible. After a certain number of passes, the child holding the potato is eliminated. This process continues until only one child remains.

Given a list of names and a counting number num , determine the last child remaining.

Input

The first line contains an integer N ($1 \leq N \leq 100$) – the number of children playing.

The second line contains N space-separated names (each name consists of uppercase and lowercase English letters, with length ≤ 10).

The third line contains an integer num ($1 \leq num \leq 1000$) – the number of passes before elimination.

Output

Print a single line containing the name of the last remaining child.

Sample Input

5
Alice Bob Charlie David Eve
3
6
John Mike Sarah Emma Tom Lily
5

Sample Output

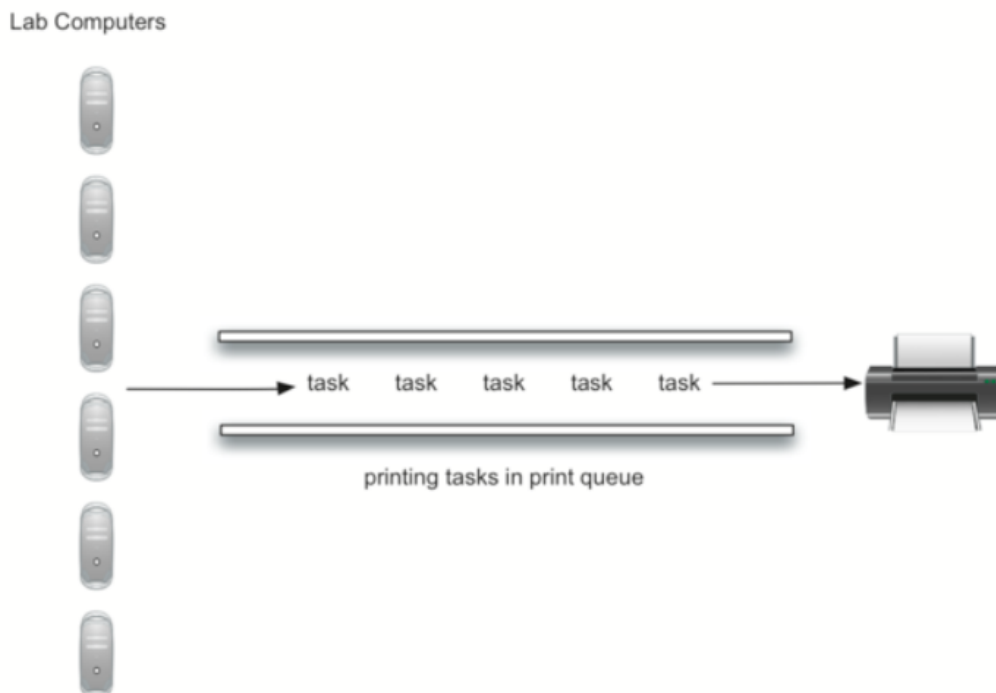
Charlie

Mike

Students send printing tasks to the shared printer, the tasks are placed in a queue to be processed in a first-come first-served manner. Many questions arise with this configuration. The most important of these might be whether the printer is capable of handling a certain amount of work. If it cannot, students will be waiting too long for printing and may miss their next class.

Consider the following situation in a computer science laboratory. On any average day about 10 students are working in the lab at any given hour. These students typically print up to twice during that time, and the length of these tasks ranges from 1 to 20 pages. The printer in the lab is older, capable of processing 10 pages per minute of draft quality. The printer could be switched to give better quality, but then it would produce only five pages per minute. The slower printing speed could make students wait too long. What page rate should be used?

We could decide by building a simulation that models the laboratory. We will need to construct representations for students, printing tasks, and the printer. As students submit printing tasks, we will add them to a waiting list, a queue of print tasks attached to the printer. When the printer completes a task, it will look at the queue to see if there are any remaining tasks to process. Of interest for us is the average amount of time students will wait for their papers to be printed. This is equal to the average amount of time a task waits in the queue.



Students may print a paper from 1 to 20 pages in length. If each length from 1 to 20 is equally likely, the actual length for a print task can be simulated by using a random number between 1 and 20 inclusive. This means that there is equal chance of any length from 1 to 20 appearing.

If there are 10 students in the lab and each prints twice, then there are 20 print tasks per hour on average. What is the chance that at any given second, a print task is going to be created? The way to answer this is to consider the ratio of tasks to time. Twenty tasks per hour means that on average there will be one task every 180 seconds:

$$\frac{20 \text{ tasks}}{1 \text{ hour}} \times \frac{1 \text{ hour}}{60 \text{ minutes}} \times \frac{1 \text{ minute}}{60 \text{ seconds}} = \frac{1 \text{ task}}{180 \text{ seconds}}$$

For every second we can simulate the chance that a print task occurs by generating a random number between 1 and 180 inclusive. If the number is 180, we say a task has been created. Note that it is possible that many tasks could be created in a row or we may wait quite a while for a task to appear. That is the nature of simulation. You want to simulate the real situation as closely as possible given that you know general parameters.

Input

First line: An integer M ($1 \leq M \leq 10$) - the number of modes to test (each mode has a different page rate).

Next M lines: Each line contains an integer P ($1 \leq P \leq 10$) - the page rate (pages per minute) of the printer for that mode.

Output

For each page rate, print the average wait time (in seconds) for a print job in that mode. The output should contain M lines, each formatted as:

Page Rate P : Average Wait Time = X seconds

where X is rounded to two decimal places.

Sample Input

2
10
5

3
8
6
4

Sample Output

Page Rate 10: Average Wait Time = 45.25 seconds
Page Rate 5: Average Wait Time = 183.90 seconds

Page Rate 8: Average Wait Time = 78.65 seconds
Page Rate 6: Average Wait Time = 145.30 seconds
Page Rate 4: Average Wait Time = 275.80 seconds

DSCP 06**Randomized Hashing Analysis**

A Hash function is defined as $H(k) = r_i$ where r_1, r_2, \dots, r_n is a sequence of random numbers between 1 and n (each integer appears exactly once).

1. Prove that if the hash table is not full then this hashing always resolves collision.
2. Does this technique eliminate clustering?
3. If I be the load factor of the table, what is the expected time for a
 - a. Successful search?
 - b. Unsuccessful search?

A hash function is defined as:

$$H(k) = r_i$$

where r_1, r_2, \dots, r_n is a random permutation of numbers from 1 to n (each integer appears exactly once).

This problem requires us to analyze the properties of this hashing technique in terms of collision resolution, clustering, and search efficiency.

Input

An integer n ($1 \leq n \leq 10^6$) – the size of the hash table.

An integer m ($1 \leq m \leq n$) – the number of keys inserted.

A list of m integers representing the keys to be inserted.

Output

A proof statement confirming that collision resolution works if the table is not full.

A statement whether clustering occurs or not.

Two floating-point values rounded to four decimal places:

Expected time for a successful search.

Expected time for an unsuccessful search.

Sample Input

10

5

12 45 78 23 56

20

10

5 15 25 35 45 55 65 75 85 95

Sample Output

Collision resolution is guaranteed if the table is not full.

Clustering does not occur due to random placement.

Successful search expected time: 1.5000

Unsuccessful search expected time: 2.0000

Collision resolution is guaranteed if the table is not full.

Clustering does not occur due to random placement.

Successful search expected time: 1.9091

Unsuccessful search expected time: 2.2222

DSCP 07**The Lost Treasure map**

Long ago, a famous pirate named Captain Hashbeard buried a vast treasure on a secret island. To keep his riches safe, he divided the treasure's location into N clues and assigned each clue a unique key. He then entrusted these clues to his crew members, who scattered them across the seven seas.

Years later, a young adventurer named Alex discovered an ancient scroll containing a mysterious hash function that Captain Hashbeard had used to store the clues:

$$H(k) = r_i$$

where r_1, r_2, \dots, r_n is a random sequence of numbers from 1 to N , ensuring that no two keys are assigned the same location.

To find the treasure, Alex must:

1. Insert all the discovered clues into the hash table.
2. Resolve collisions if two clues map to the same location.
3. Search for missing clues using an efficient lookup process.

Prove that Captain Hashbeard's method ensures collision resolution if there's space left.

Determine if clue clustering occurs when storing the information.

Calculate how efficiently Alex can find a clue (successful search) and determine if a missing clue exists (unsuccessful search).

Can Alex unlock the secrets of Captain Hashbeard's hashing technique and retrieve the lost treasure before rival pirates arrive?

Input

An integer N ($1 \leq N \leq 10^6$) → Size of the hash table.

An integer M ($1 \leq M \leq N$) → Number of discovered clues.

A list of M integers → The keys (clue identifiers) that Alex has found.

Output

A statement confirming whether collision resolution is guaranteed.

A statement confirming whether clustering occurs or not.

Two floating-point values rounded to four decimal places:

Expected time for a successful search.

Expected time for an unsuccessful search.

Sample Input

10
5
12 45 78 23 56

20
10

Sample Output

Collision resolution is guaranteed if the table is not full.

Clustering does not occur due to random placement.

Successful search expected time: 1.5000
Unsuccessful search expected time: 2.0000

Collision resolution is guaranteed if the table is not full.

5 15 25 35 45 55 65 75 85 95

Clustering does not occur due to random placement.

Successful search expected time: 1.9091

Unsuccessful search expected time: 2.2222

The midsemester point at your local university is approaching. The registrar's office wants to prepare the grade reports as soon as the students' grades are recorded. Some of the students enrolled have not yet paid their tuition, however. If a student has paid the tuition, the grades are shown on the grade report together with the grade point average (GPA). If a student has not paid the tuition, the grades are not printed. For these students, the grade report contains a message indicating that the grades have been held for non-payment of the tuition. The grade report also shows the billing amount. The registrar's office and the business office want your help in writing a program that can analyze the students' data and print the appropriate grade reports. The data is stored in a file in the following form:

345

studentName studentID isTuitionPaid numberOfCourses

courseName courseNumber creditHours grade

courseName courseNumber creditHours grade

...

studentName studentID isTuitionPaid numberOfCourses

courseName courseNumber creditHours grade

courseName courseNumber creditHours grade

...

Input

The first line indicates the tuition rate per credit hour. The students' data is given thereafter. A sample input file follows:

345

Lisa Miller 890238 Y 4

Mathematics MTH345 4 A

Physics PHY357 3 B

ComputerSci CSC478 3 B

History HIS356 3 A

...

The first line indicates that the tuition rate is \$345 per credit hour. Next, the course data for student Lisa Miller is given: Lisa Miller's ID is 890238, she has paid the tuition, and is taking 4 courses. The course number for the mathematics class she is taking is MTH345, the course has 4 credit hours, her midsemester grade is A, and so on.

Output

The desired output for each student is in the following form:

Student Name: Lisa Miller

Student ID: 890238

Number of courses enrolled: 4

Course No	Course Name	Credits	Grade
-----------	-------------	---------	-------

CSC478	ComputerSci	3	B
HIS356	History	3	A
MTH345	Mathematics	4	A
PHY357	Physics	3	B
Total number of credits: 13			
Midsemester GPA: 3.54			
This output shows that the courses must be ordered according to the course number. To calculate the GPA, we assume that the grade A is equivalent to 4 points, B is equivalent to 3 points, C is equivalent to 2 points, D is equivalent to 1 point, and F is equivalent to 0 points.			
Sample Input		Sample Output	
A file containing the data in the form given previously. For easy reference in the rest of the discussion, let us assume that the name of the input file is stData.txt.		A file containing the output of the form given previously. Let us assume that the name of the output file is stDataOut.txt.	

Some of the characteristics of a book are the title, author(s), publisher, ISBN, price, and year of publication. Design the class `bookType` that defines the book as an ADT. Each object of the class `bookType` can hold the following information about a book: title, up to four authors, publisher, ISBN, price, and number of copies in stock. To keep track of the number of authors, add another data member. Include the member functions to perform the various operations on the objects of `bookType`. For example, the typical operations that can be performed on the title are to show the title, set the title, and check whether a title is the same as the actual title of the book. Similarly, the typical operations that can be performed on the number of copies in stock are to show the number of copies in stock, set the number of copies in stock, update the number of copies in stock, and return the number of copies in stock. Add similar operations for the publisher, ISBN, book price, and authors. Add the appropriate constructors and a destructor (if one is needed). Write the definitions of the member functions of the class `bookType`. Write a program that uses the class `bookType` and tests the various operations on the objects of class `bookType`. Declare a vector container of type `bookType`. Some of the operations that you should perform are to search for a book by its title, search by ISBN, and update the number of copies in stock.

Input

Number of books to be added.

For each book, input the following details:

- Title (String)
- Number of Authors (Integer, up to 4)
- Author Names (Strings)
- Publisher (String)
- ISBN (String)
- Price (Float)
- Number of Copies in Stock (Integer)

Menu-driven operations:

- 1 → Search for a book by title
- 2 → Search for a book by ISBN
- 3 → Update the number of copies in stock
- 4 → Display book details
- 5 → Exit

Output

1. Search by Title

If the book is found:

Book Found:

Title: <Book Title>

Authors: <Author 1>, <Author 2>, ...

Publisher: <Publisher Name>

ISBN: <ISBN Number>

Price: \$<Price>

Copies in Stock: <Stock Count>

If the book is not found:

Book not found.

2. Search by ISBN

If the book is found:

Book Found:

Title: <Book Title>

Authors: <Author 1>, <Author 2>, ...

Publisher: <Publisher Name>

ISBN: <ISBN Number>

Price: \$<Price>

Copies in Stock: <Stock Count>

If the book is not found:

Book not found.

3. Update Copies in Stock

If the ISBN is found and stock is updated:

Stock updated successfully.

New Copies in Stock: <Updated Stock Count>

If the ISBN is not found:

Book not found. Stock update failed.

4. Display All Books

If there are books in the system:

List of Books:

1. Title: <Book Title>

Authors: <Author 1>, <Author 2>, ...

Publisher: <Publisher Name>

ISBN: <ISBN Number>

Price: \$<Price>

Copies in Stock: <Stock Count>

2. Title: <Book Title>

Authors: <Author 1>, <Author 2>, ...

Publisher: <Publisher Name>

ISBN: <ISBN Number>

Price: \$<Price>
Copies in Stock: <Stock Count>
If no books are available:
No books available in the system.

5. Exit Message
Exiting the program. Thank you!

Sample Input

Enter number of books: 2

Enter book details:
Title: Introduction to Algorithms
Number of Authors: 2
Author 1: Thomas H. Cormen
Author 2: Charles E. Leiserson
Publisher: MIT Press
ISBN: 9780262033848
Price: 75.50
Copies in Stock: 10

Title: Clean Code
Number of Authors: 1
Author 1: Robert C. Martin
Publisher: Prentice Hall
ISBN: 9780132350884
Price: 40.00
Copies in Stock: 5

Menu:
1. Search by Title
2. Search by ISBN
3. Update Copies in Stock
4. Display All Books
5. Exit
Enter choice: 1
Enter title to search: Clean Code

Sample Output

Book Found:
Title: Clean Code
Authors: Robert C. Martin
Publisher: Prentice Hall
ISBN: 9780132350884
Price: \$40.00
Copies in Stock: 5

If not found:
Book not found.

When updating copies in stock:
Enter ISBN to update stock: 9780132350884
Enter new stock quantity: 8
Stock updated successfully.

When displaying all books:
List of Books:
1. Title: Introduction to Algorithms
Authors: Thomas H. Cormen, Charles E. Leiserson
Publisher: MIT Press
ISBN: 9780262033848
Price: \$75.50
Copies in Stock: 10

2. Title: Clean Code
Authors: Robert C. Martin
Publisher: Prentice Hall

ISBN: 9780132350884

Price: \$40.00

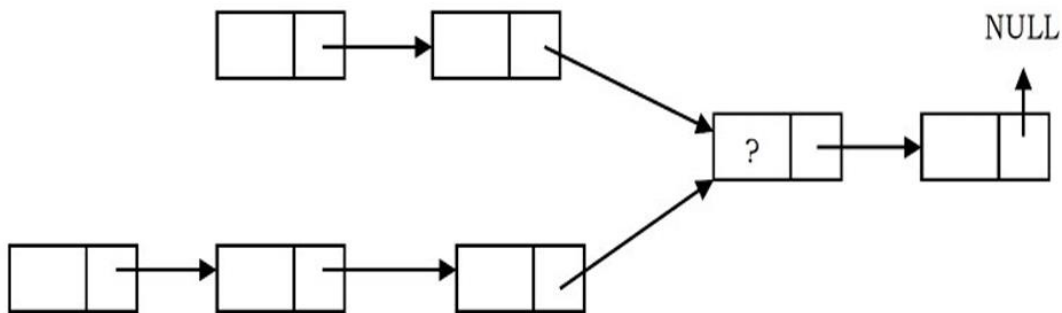
Copies in Stock: 8

On exiting the program:

Exiting the program. Thank you!

DSCP 10**Merge Point of Two Intersecting Singly Linked Lists**

Suppose there are two singly linked lists both of which intersect at some point and become a single linked list. The head or start pointers of both the lists are known, but the intersecting node is not known. Also, the number of nodes in each of the lists before they intersect is unknown and may be different in each list. List1 may have n nodes before it reaches the intersection point, and List2 might have m nodes before it reaches the intersection point where m and n may be $m = n$, $m < n$ or $m > n$. Give an algorithm for finding the merging point.

**Input**

Two singly linked lists represented as sequences of node values.

The lists may have different lengths before they intersect.

The intersection is represented by a common node (not just by value but by reference).

List1: $4 \rightarrow 1 \rightarrow 8 \rightarrow 4 \rightarrow 5$

List2: $5 \rightarrow 6 \rightarrow 1 \rightarrow 8 \rightarrow 4 \rightarrow 5$

Output

The output should be the value of the intersecting node.

If no intersection exists, return null or -1.

8

If the lists do not intersect:

-1

Sample Input

List 1:

1 -> 2 -> 3 -> 4 -> 5 -> 6

List 2:

9 -> 8 -> 4 -> 5 -> 6

List 1:

1 -> 2 -> 3 -> 7 -> 8 -> 9

Sample Output

Merge point value: 4

Merge point value: 7

List 2:

4 -> 5 -> 6 -> 7 -> 8 -> 9

List 1:

1 -> 2 -> 3

No merge point found.

List 2:

4 -> 5 -> 6

DSCP 11**Maximum Sum in Sliding Window**

Maximum sum in sliding window: Given array $A[]$ with sliding window of size w which is moving from the very left of the array to the very right. Assume that we can only see the w numbers in the window. Each time the sliding window moves rightwards by one position. For example: The array is $[1\ 3\ -1\ -3\ 5\ 3\ 6\ 7]$, and w is 3.

Window Position	Max
$[1\ 3\ -1]\ -3\ 5\ 3\ 6\ 7$	3
$1\ [3\ -1\ -3]\ 5\ 3\ 6\ 7$	3
$1\ 3\ [-1\ -3\ 5]\ 3\ 6\ 7$	5
$1\ 3\ -1\ [-3\ 5\ 3]\ 6\ 7$	5
$1\ 3\ -1\ -3\ [5\ 3\ 6]\ 7$	6
$1\ 3\ -1\ -3\ 5\ [3\ 6\ 7]$	7

Input

A long array $A[]$, and a window width w .

Output

An array $B[]$, $B[i]$ is the maximum value from $A[i]$ to $A[i+w-1]$.

Sample Input

Array: $[1, 3, -1, -3, 5, 3, 6, 7]$

Window size (w): 3

Array: $[2, 1, 5, 1, 3, 2]$

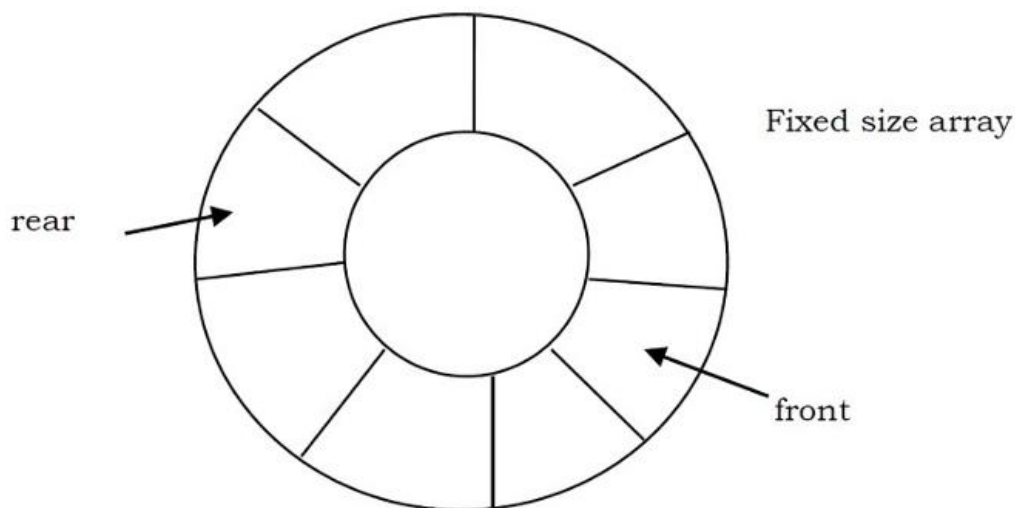
Window size (w): 3

Sample Output

Maximum sum in sliding window of size 3: 10

Maximum sum in sliding window of size 3: 9

A queue is set up in a circular array $A[0..n-1]$ with front and rear defined as usual. Assume that $n-1$ locations in the array are available for storing the elements (with the other element being used to detect full/empty condition). Give a formula for the number of elements in the queue in terms of rear, front, and n .



- Rear of the queue is somewhere clockwise from the front.
- To enqueue an element, we move rear one position clockwise and write the element in that position.
- To dequeue, we simply move front one position clockwise.
- Queue migrates in a clockwise direction as we enqueue and dequeue.
- Emptiness and fullness to be checked carefully.
- Analyze the possible situations (make some drawings to see where front and rear are when the queue is empty, and partially and totally filled).

Formula to calculate the number of elements in the queue:

$$\text{Number of elements} = (\text{rear} - \text{front} + n) \% n$$

Input

First Line:

An integer n , representing the size of the circular queue (i.e., the total number of positions available in the array).

Second Line:

Two space-separated integers front and rear, representing the indices of the front and rear pointers of the queue.

Third Line (optional, depending on the context):

A list of integers representing the elements currently in the queue (if applicable). This line is not necessary for calculating the number of elements based on the formula, but it can be provided for additional context.

Output

Single Line Output:

A single integer representing the number of elements in the queue.

Sample Input

Queue size: n = 5

Elements in the queue: 3, 4, 5

front = 2

rear = 0

Queue size: n = 6

Elements in the queue: 10, 20, 30, 40

front = 3

rear = 1

Queue size: n = 5

Elements in the queue: None

front = 0

rear = 0

Queue size: n = 6

Elements in the queue: 1, 2, 3, 4, 5

front = 0

rear = 5

Queue size: n = 4

Elements in the queue: 100

front = 2

rear = 3

Sample Output

Number of elements in the queue: 3

Number of elements in the queue: 4

Number of elements in the queue: 0

Number of elements in the queue: 5

Number of elements in the queue: 1

DSCP 13**Queue to Stack Transfer with Order Preservation**

Given a queue Q containing n elements, transfer these items on to a stack S (initially empty) so that front element of Q appears at the top of the stack and the order of all other items is preserved. Using enqueue and dequeue operations for the queue, and push and pop operations for the stack, outline an efficient $O(n)$ algorithm to accomplish the above task, using only a constant amount of additional storage.

Given a queue Q containing n elements, we need to transfer the elements to a stack S such that the front element of the queue appears at the top of the stack. The order of all other elements is preserved when transferred. The transfer should be done using the following operations:

Queue Operations:

Enqueue: Add an element to the end of the queue.

Dequeue: Remove the front element from the queue.

Stack Operations:

Push: Add an element to the top of the stack.

Pop: Remove the top element from the stack.

Input

First Line:

An integer n ($1 \leq n \leq 10^5$), representing the number of elements in the queue.

Second Line:

A list of n space-separated integers representing the elements in the queue.

Output

Single Line Output:

A space-separated list of integers, representing the elements of the stack after transferring them from the queue.

Sample Input

5
1 2 3 4 5

4
10 20 30 40

3
5 7 9

Sample Output

5 4 3 2 1

40 30 20 10

9 7 5

DSCP 14**Check Consecutive Pairs in a Stack**

Given a stack of integers, how do you check whether each successive pair of numbers in the stack is consecutive or not. The pairs can be increasing or decreasing, and if the stack has an odd number of elements, the element at the top is left out of a pair. For example, if the stack of elements are [4, 5, -2, -3, 11, 10, 5, 6, 20], then the output should be true because each of the pairs (4, 5), (-2, -3), (11, 10), and (5, 6) consists of consecutive numbers.

The task is to check whether each successive pair of integers in a stack are consecutive or not. A pair is consecutive if the difference between the two numbers is exactly ± 1 . If the stack has an odd number of elements, the last element at the top is left out of pairing.

Input

First Line:

An integer n ($1 \leq n \leq 10^5$), representing the number of elements in the stack.

Second Line:

A list of n integers representing the elements in the stack, starting from the top of the stack to the bottom.

Output

Single Line Output:

A Boolean value (True or False) indicating whether each successive pair of elements in the stack is consecutive or not.

Sample Input

9

20 6 5 11 10 -3 -2 5 4

9

4 5 -2 -3 11 10 5 6 20

7

9 8 4 3 12 11 20

Stack has Odd Number of Elements

5

3 2 1 7 8

Sample Output

False

True

True

False

DSCP 15**Interleave Two Halves of a Queue**

Given a queue of integers, rearrange the elements by interleaving the first half of the list with the second half of the list. For example, suppose a queue stores the following sequence of values: [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. Consider the two halves of this list: first half: [11, 12, 13, 14, 15] second half: [16, 17, 18, 19, 20]. These are combined in an alternating fashion to form a sequence of interleave pairs: the first values from each half (11 and 16), then the second values from each half (12 and 17), then the third values from each half (13 and 18), and so on. In each pair, the value from the first half appears before the value from the second half. Thus, after the call, the queue stores the following values: [11, 16, 12, 17, 13, 18, 14, 19, 15, 20].

The task is to rearrange the elements of a queue by interleaving the first half of the queue with the second half. The elements from the first half and second half are combined alternately, such that each element from the first half is followed by the corresponding element from the second half.

For example, if the queue contains the elements [11, 12, 13, 14, 15, 16, 17, 18, 19, 20], we first divide the queue into two halves:

First half: [11, 12, 13, 14, 15]

Second half: [16, 17, 18, 19, 20]

Then, we interleave them alternately as:

[11, 16, 12, 17, 13, 18, 14, 19, 15, 20]

Input

First Line:

An integer n ($1 \leq n \leq 10^5$), representing the number of elements in the queue.

Second Line:

A list of n integers representing the elements in the queue, where the front of the queue is the first element in the list.

Output

Single Line Output:

A list of integers representing the elements of the queue after rearranging them by interleaving the first and second halves.

Sample Input

10
11 12 13 14 15 16 17 18 19 20

6
1 2 3 4 5 6

7
10 20 30 40 50 60 70

Sample Output

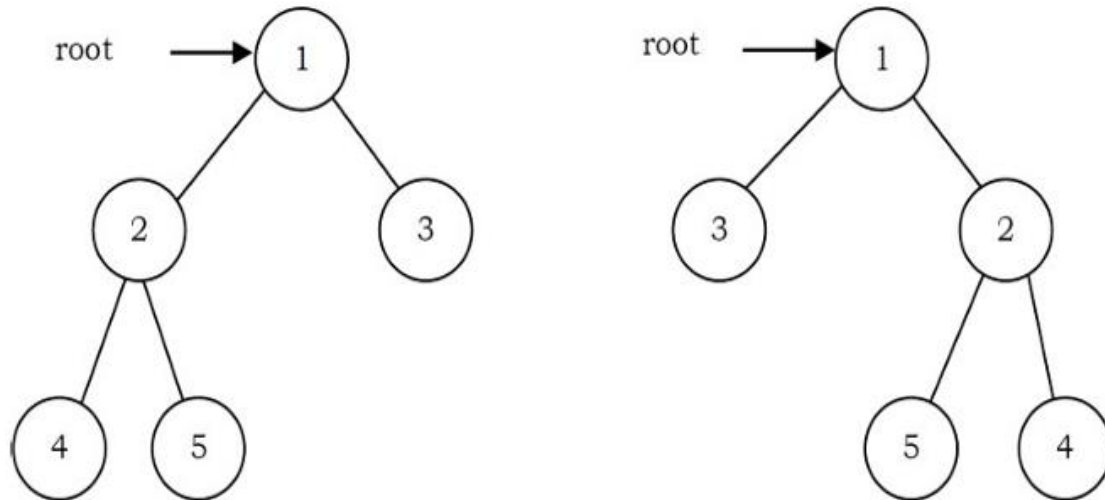
11 16 12 17 13 18 14 19 15 20

1 4 2 5 3 6

10 40 20 50 30 60 70

DSCP 16**Mirror Tree**

Give an algorithm for converting a tree to its mirror. Mirror of a tree is another tree with left and right children of all non-leaf nodes interchanged. The trees below are mirrors to each other.



The task is to convert a given binary tree into its mirror image. In a mirror tree, for each node, the left and right children are swapped. This means that if a node has a left child and a right child, their positions will be swapped.

Original Tree:

```
1
 /\
2 3
 /\
4 5
```

Mirror of the Tree:

```
1
 /\
3 2
 /\
5 4
```

Input

First Line:

An integer n ($1 \leq n \leq 10^5$), the number of nodes in the tree.

Next n Lines:

Each line contains a node value, along with its left and right child values. If a child does not exist, -1 is used to represent it.

Output

Single Line Output:

The mirror image of the tree represented in the same format as the input.

Sample Input

Sample Output

5

1

1 2 3

/\

2 4 5

3 2

3 -1 -1

/\

4 -1 -1

5 4

5 -1 -1

1

1 3 -1

/\

3 -1 -1

2 3

2 5 4

/\

5 -1 -1

4 5

4 -1 -1

1

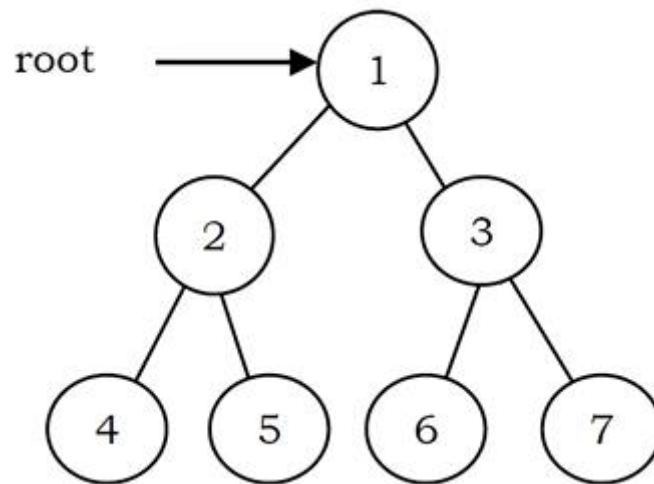
1 -1 -1

1 -1 -1

1

DSCP 17**Zigzag Tree Traversal**

Give an algorithm to traverse a binary tree in Zigzag order. For example, the output for the tree below should be: 1 3 2 4 5 6 7



The problem requires performing a zigzag traversal (also called spiral order traversal) of a binary tree. In a zigzag traversal, you traverse the tree level by level, but alternate the direction of traversal at each level.

- At level 1, you traverse from left to right.
- At level 2, you traverse from right to left.
- At level 3, you traverse from left to right, and so on.

Input

First Line:

An integer n ($1 \leq n \leq 10^5$), the number of nodes in the binary tree.

Next n Lines:

Each line contains a node value and its left and right child values. If a child does not exist, it is represented by -1.

Output

Single Line Output:

The nodes in the tree visited in zigzag order, separated by spaces.

Sample Input

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

Sample Output

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

7 -1 -1

1

/\

2 3

/\/\

4 5 6 7

1

1

1 -1 -1

1

Find the vertical sum of a binary tree. Given a tree which has 5 vertical lines

Vertical-1: nodes-4 => vertical sum is 4

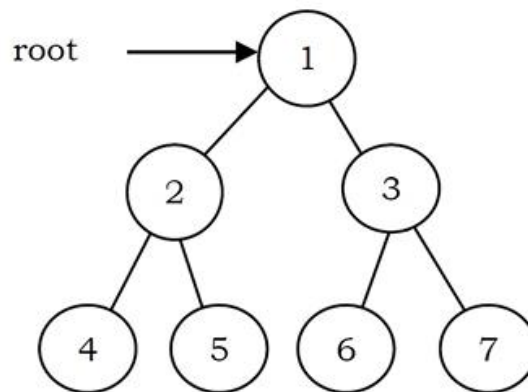
Vertical-2: nodes-2 => vertical sum is 2

Vertical-3: nodes-1,5,6 => vertical sum is $1 + 5 + 6 = 12$

Vertical-4: nodes-3 => vertical sum is 3

Vertical-5: nodes-7 => vertical sum is 7

We need to output: 4 2 12 3 7



The problem requires calculating the vertical sum of a binary tree. Each node in a binary tree has a horizontal distance (HD) from the root:

The root is at HD = 0.

The left child of a node has HD = parent's HD - 1.

The right child of a node has HD = parent's HD + 1.

The vertical sum is obtained by summing all the nodes that share the same horizontal distance.

Step-by-step Calculation of Vertical Sums

Thus, the output should be:

4 2 12 3 7 (from leftmost to rightmost vertical line)

Vertical Line	Nodes Present	Vertical Sum
Vertical - 2	4	4
Vertical - 1	2	2
Vertical 0	1, 5, 6	$1 + 5 + 6 = 12$
Vertical 1	3	3
Vertical 2	7	7

Input

First Line: An integer n ($1 \leq n \leq 10^5$) representing the number of nodes.

Next n Lines: Each line contains three integers:

- node_value left_child right_child
- If a node does not have a left or right child, -1 is used.

Output

A single line containing the vertical sums from leftmost vertical to rightmost vertical, separated by spaces.

Sample Input

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

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

```
5
10 20 30
20 40 -1
30 -1 50
40 -1 -1
50 -1 -1
```

```
    10
   / \
  20 30
 /   \
40    50
```

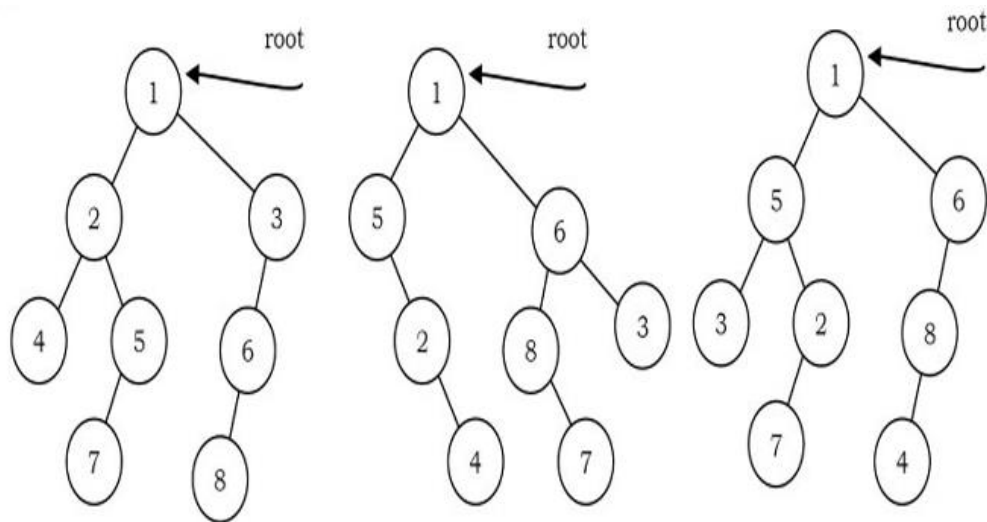
Sample Output

```
4 2 12 3 7
```

```
40 20 10 30 50
```

Two binary trees are said to be isomorphic if they have the same structure, regardless of the values of their nodes. This means both trees must have the same number of nodes and the left and right children of corresponding nodes must be arranged in the same way.

Two binary trees root1 and root2 are isomorphic if they have the same structure. The values of the nodes does not affect whether two trees are isomorphic or not. In the diagram below, the tree in the middle is not isomorphic to the other trees, but the tree on the right is isomorphic to the tree on the left. Given two trees, check whether the trees are isomorphic to each other or not?



Input

First, provide two binary trees as level-order input (or NULL if a node is missing).

The trees can be given in the form of node values and structure.

Tree 1:

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

Tree 2:

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

Here, -1 represents a NULL node.

Output

Output "Yes" if the two trees are isomorphic.

Output "No" if they are not.

Sample Input

Tree 1:

1

2 3

4 5 -1 -1

-1 -1 -1 -1

Tree 2:

1

2 3

4 5 -1 -1

-1 -1 -1 -1

Tree 1:

1

2 3

4 5 -1 -1

-1 -1 -1 -1

Tree 2:

1

2 -1

4 3

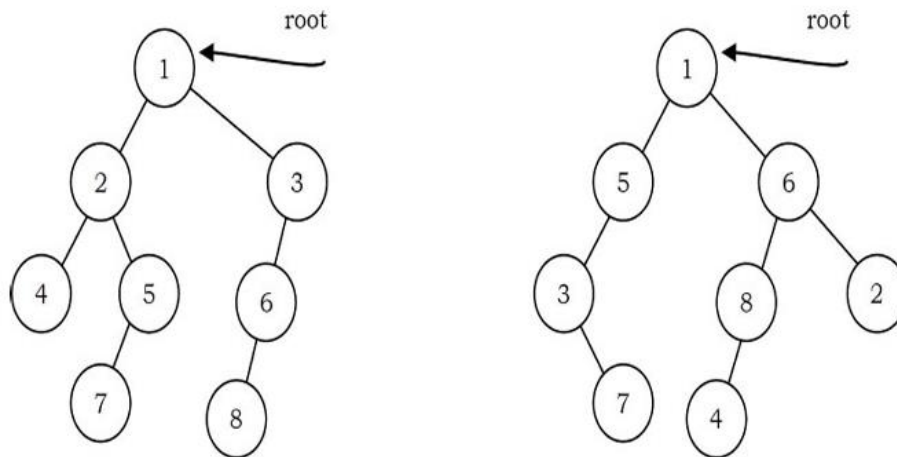
-1 5 -1 -1

Sample Output

Yes

No

Two trees root1 and root2 are quasi-isomorphic if root1 can be transformed into root2 by swapping the left and right children of some of the nodes of root1. Data in the nodes are not important in determining quasi-isomorphism; only the shape is important. The trees below are quasi-isomorphic because if the children of the nodes on the left are swapped, the tree on the right is obtained. Given two trees, check whether they are quasi-isomorphic to each other or not?



Quasi-Isomorphic Trees

Tree 1	Tree 2
1	1
/ \	/ \
2 3	3 2
/ \ \	/ / \
4 5 6	6 5 4

In Tree 1, if we swap the left and right children of node 1, we get Tree 2. Thus, Tree 1 and Tree 2 are quasi-isomorphic.

Non-Quasi-Isomorphic Trees

Tree 1	Tree 2
1	1
/ \	/ \
2 3	3 2
/ \ \	/ \
4 5 6	6 4

Input

The input consists of two binary trees given in level-order format (or NULL for missing nodes). Each tree is described as a series of values with -1 representing NULL nodes.

Tree 1:

1
2 3

4 5 -1 6
-1 -1 -1 -1 -1 -1

Tree 2:

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

Here, -1 represents a NULL node.

Output

"Yes" if the two trees are quasi-isomorphic.

"No" if they are not.

Sample Input

Tree 1:
1
2 3
4 5 -1 6
-1 -1 -1 -1 -1 -1

Tree 2:
1
3 2
6 -1 5 4
-1 -1 -1 -1 -1 -1

Tree 1:
1
2 3
4 5 -1 6
-1 -1 -1 -1 -1 -1

Tree 2:
1
3 2
6 -1 -1 4
-1 -1 -1 -1

Sample Output

Yes

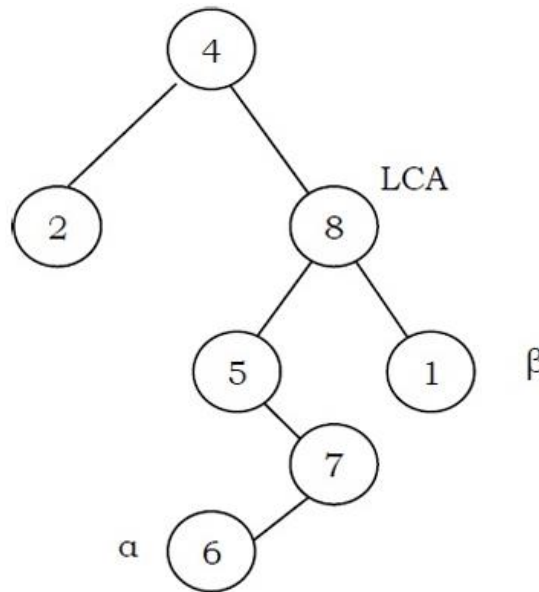
No

Given a Binary Search Tree (BST) and two nodes n_1 and n_2 , the Lowest Common Ancestor (LCA) of n_1 and n_2 is the lowest node in the tree that has both n_1 and n_2 as its descendants (where we allow a node to be a descendant of itself).

Key Properties of a BST

- The left subtree of a node contains only nodes with values less than the node's value.
- The right subtree of a node contains only nodes with values greater than the node's value.
- This property allows us to efficiently find the LCA in $O(\log N)$ time for a balanced BST.

Given pointers to two nodes in a binary search tree, find the lowest common ancestor (LCA). Assume that both values already exist in the tree.



Input

The input consists of a BST in level-order format (or -1 for NULL nodes).

The values of two nodes (n_1 and n_2) for which we need to find the LCA.

BST:

```

    20
   / \
  10  30
 / \
5   15

```

$n_1 = 5, n_2 = 15, \text{LCA} = 10$

Output

The value of the Lowest Common Ancestor (LCA).

Sample Input

BST:

```
    20
   / \
  10  30
 / \
5   15
```

n1 = 5, n2 = 15

BST:

```
    20
   / \
  10  30
 / \  \
5   15 40
```

n1 = 5, n2 = 30

BST:

```
    20
   / \
  10  30
 / \  \
5   15 40
```

n1 = 20, n2 = 30

Sample Output

10

20

20

In a Binary Search Tree (BST), Floor and Ceiling of a given key are defined as follows:

Floor of a key: The largest key in the BST that is less than or equal to the given key.

Ceiling of a key: The smallest key in the BST that is greater than or equal to the given key.

If a key is smaller than the root of the tree, then the floor must be in the left subtree. If a key is larger than the root, then the ceiling could be in the right subtree (but only if a smaller key exists in the right subtree). If not, the floor would be the root itself. If the key equals the root, then the root itself is both the floor and the ceiling.

If a given key is less than the key at the root of a BST then the floor of the key (the largest key in the BST less than or equal to the key) must be in the left subtree. If the key is greater than the key at the root, then the floor of the key could be in the right subtree, but only if there is a key smaller than or equal to the key in the right subtree; if not (or if the key is equal to the key at the root) then the key at the root is the floor of the key. Finding the ceiling is similar, with interchanging right and left. For example, if the sorted with input array is {1, 2, 8, 10, 10, 12, 19}, then

For x = 0: floor doesn't exist in array, ceil = 1, For x = 1: floor = 1, ceil = 1

For x = 5: floor = 2, ceil = 8, For x = 20: floor = 19, ceil doesn't exist in array.

Input

A Binary Search Tree (BST) is provided as input in the form of a root node.

An integer x for which we need to find the floor and ceiling.

Output

The output consists of two integers: the floor and the ceiling for the given key x.

If the floor or ceiling doesn't exist in the BST, return None.

Sample Input

BST:

```

    10
   /  \
  5    20
 / \  / \
2  8 15 30

```

x = 5

BST:

```

    10
   /  \
  5    20
 / \  / \
2  8 15 30

```

Sample Output

Floor = 5

Ceiling = 5

Floor = None

Ceiling = 2

x = 0

BST:

```
    10
   /  \
  5    20
 / \  / \
2  8 15 30
```

Floor = 5

Ceiling = 5

x = 5

BST:

```
    10
   /  \
  5    20
 / \  / \
2  8 15 30
```

Floor = 15

Ceiling = 15

x = 15

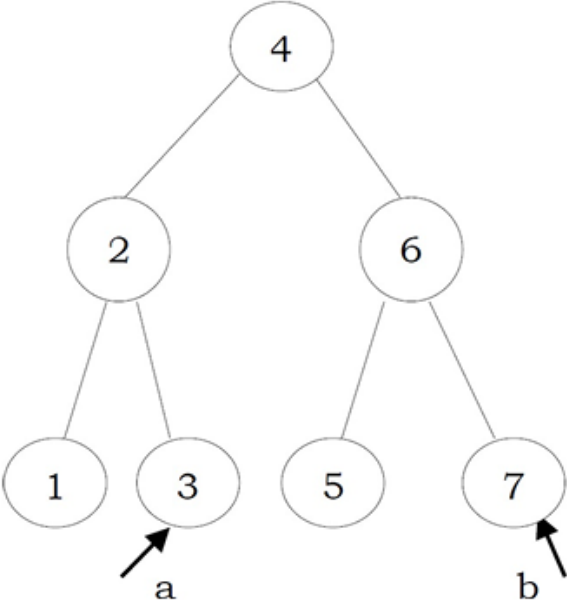
BST:

```
    10
   /  \
  5    20
 / \  / \
2  8 15 30
```

Floor = 15

Ceiling = 20

x = 18

DSCP 23	Counting Nodes in the Range [a, b]
<p>An AVL tree is a self-balancing binary search tree (BST), meaning that for every node, the height of the left and right subtrees differs by at most 1. Like a BST, an AVL tree maintains an ordered structure, making it efficient for searching operations.</p> <p>Given an AVL tree with n integer items and two integers a and b, where a and b can be any integers with $a \leq b$. Implement an algorithm to count the number of nodes in the range $[a, b]$.</p> 	
<p>Input</p> <p>An AVL Tree with n integer values.</p> <p>Two integers a and b, where $a \leq b$, representing the range.</p> <p>AVL Tree:</p> <pre> 20 / \ 10 30 / \ / \ 5 15 25 35 / 2 </pre> <p>$a = 10, b = 30$</p>	
<p>Output</p> <p>A single integer representing the count of nodes within the given range $[a, b]$.</p> <p>The numbers in the AVL tree that fall within $[10, 30]$ are $\{10, 15, 20, 25, 30\}$. So, the output is 5.</p>	
Sample Input	Sample Output

AVL Tree:

5

```
    20
   /  \
  10   30
 / \  / \
5  15 25 35
/
2
```

a = 10, b = 30

AVL Tree:

7

```
    15
   /  \
  10   20
 / \  / \
5  12 18 25
```

a = 5, b = 25

AVL Tree:

0

```
    20
   /  \
  10   30
 / \  / \
5  15 25 35
/
2
```

a = 40, b = 50

AVL Tree:

1

```
    10
```

a = 5, b = 15

DSCP 24**BST Inorder Transformation**

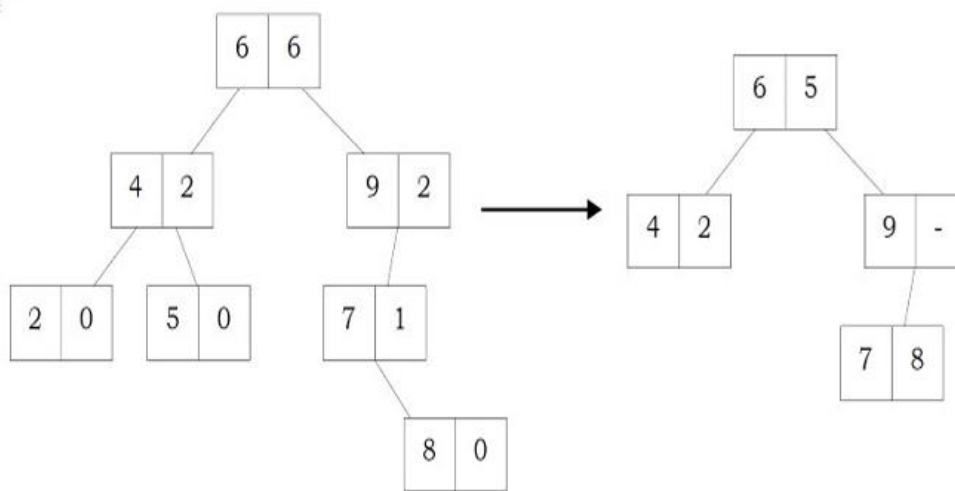
We are given a Binary Search Tree (BST) where each node has two pieces of information:

data → The value of the node.

subtree_size → The number of nodes in its subtree, including itself.

Given a BST where each node contains two data elements (its data and also the number of nodes in its subtrees) as shown below. Convert the tree to another BST by replacing the second data element (number of nodes in its subtrees) with previous node data in inorder traversal.

Note that each node is merged with inorder previous node data. Also make sure that conversion happens in-place.

**Input**

A Binary Search Tree (BST) where each node has:

data (integer)

subtree_size (total number of nodes in its subtree)

The BST is given as a set of node values.

Output

The modified BST where each node's subtree_size is replaced with its inorder predecessor's data.

If a node has no predecessor, its subtree_size is set to -1

Sample Input

```
10(7)
 /  \
5(3) 15(3)
 /  \  \
2(1) 7(1) 20(1)
```

Inorder traversal order: [2, 5, 7, 10, 15, 20]

Replacing subtree_size with previous node's data:

Sample Output

```
10(7)
 /  \
5(2) 15(10)
 /  \  \
2(-1) 7(5) 20(15)
```

2 has no previous \rightarrow subtree_size = -1

5 replaces with 2

7 replaces with 5

10 replaces with 7

15 replaces with 10

20 replaces with 15

8(5)
/ \
3(3) 10(1)
/ \
1(1) 6(1)

8(6)
/ \
3(1) 10(8)
/ \
1(-1) 6(3)

Inorder traversal

[1, 3, 6, 8, 10]

1 \rightarrow -1

3 \rightarrow 1

6 \rightarrow 3

8 \rightarrow 6

10 \rightarrow 8

DSCP 25

Pruning a BST

Given a Binary Search Tree (BST) and a range defined by two integers' min and max, the task is to remove all nodes from the BST that do not fall within this range. This process is known as BST pruning.

Pruning Approach

If a node's value is less than min, then the left subtree of that node is pruned, and we replace the node with its right subtree.

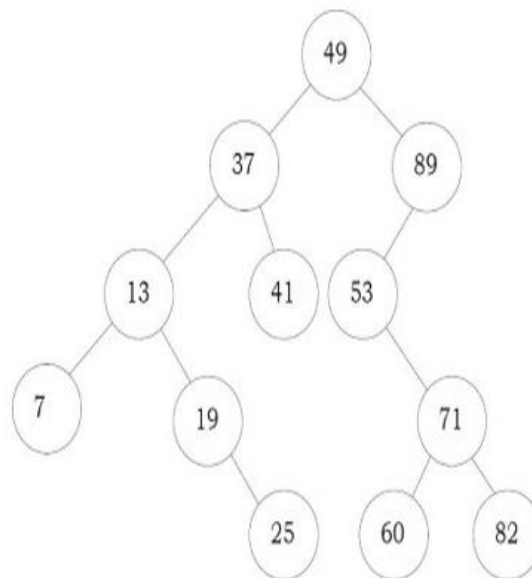
If a node's value is greater than max, then the right subtree of that node is pruned, and we replace the node with its left subtree.

Otherwise, recursively prune both left and right subtrees while keeping the node intact.

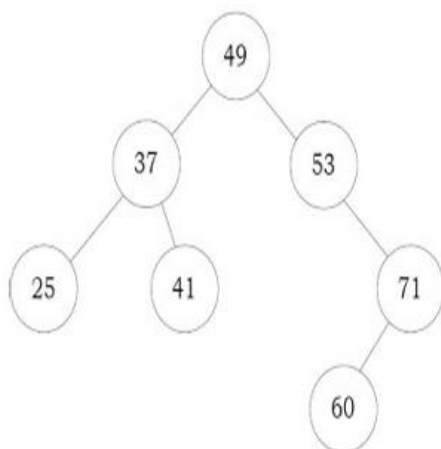
The function should return the new root after pruning.

Given a BST and two integers (minimum and maximum integers) as parameters, how do you remove (prune) elements that are not within that range?

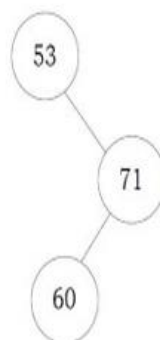
Sample Tree



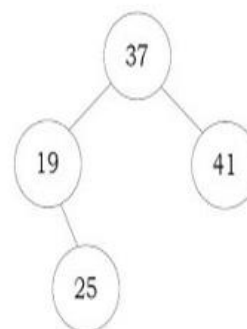
PruneBST(24,71);



PruneBST(53,79);



PruneBST(17,41);



Input

The BST is provided as a list of integers inserted sequentially.

Two integers' min and max, representing the valid range.

Output

The BST after pruning, printed in inorder traversal.

Sample Input

```
BST:      8
      /  \
     3    10
    /\    \
   1 6    14
  /\  /
 4 7 13
```

min = 5, max = 13

```
BST:      15
      /  \
     10   20
    /\  /\
   5 12 18 25
```

min = 10, max = 20

Sample Output

Pruned BST (inorder traversal): 6 7 8 10 13

Pruned BST (inorder traversal): 10 12 15 18 20

DSCP 26**Balanced BST Over Complete Binary Tree for Data Storage**

A Balanced Binary Search Tree (BST) and a Complete Binary Tree (CBT) are two common tree structures used for storing a set S of n numbers. However, when each node in the tree contains the number of nodes in its subtree, a Balanced BST has significant advantages over a Complete Binary Tree.

Assume that a set S of n numbers are stored in some form of balanced binary search tree; i.e. the depth of the tree is $O(\log n)$. In addition to the key value and the pointers to children, assume that every node contains the number of nodes in its subtree. Specify a reason(s) why a balanced binary tree can be a better option than a complete binary tree for storing the set S .

Input

n : Number of elements in the set S .

Set of n numbers: Values to be stored in the tree.

Output

Comparison results and an explanation of why a balanced BST is better.

Sample Input

$n = 7$

$S = [10, 20, 30, 40, 50, 60, 70]$

$n = 10$

$S = [5, 15, 25, 35, 45, 55, 65, 75, 85, 95]$

Sample Output

Balanced BST is better than a Complete Binary Tree because:

1. Searching for any element takes $O(\log n)$ time in a Balanced BST, but $O(n)$ in a CBT.
2. Insertions and deletions are faster in a Balanced BST ($O(\log n)$), whereas CBT restructuring is costly.
3. The number of nodes in any subtree can be retrieved in $O(1)$ time in a Balanced BST but requires $O(n)$ traversal in a CBT.

Balanced BST is preferred over a Complete Binary Tree because:

1. Searching for 35 in a Balanced BST takes $O(\log n)$, whereas it requires $O(n)$ in a CBT.
2. Finding the k -th smallest element is easy in a Balanced BST using subtree size, but not in a CBT.
3. Memory usage is more efficient in a Balanced BST.

DSCP 27**Josephus Problem**

The **Josephus problem** is a theoretical elimination game where N people stand in a circle and eliminate every M^{th} person until only one person remains. The goal is to determine the order in which people are eliminated and find the safe position where Josephus should sit.

In the Josephus problem from antiquity, N people are in dire straits and agree to the following strategy to reduce the population. They arrange themselves in a circle (at positions numbered from 0 to $N-1$) and proceed around the circle, eliminating every M^{th} person until only one person is left. Legend has it that Josephus figured out where to sit to avoid being eliminated. Write a Queue client Josephus that takes N and M from the command line and prints out the order in which people are eliminated (and thus would show Josephus where to sit in the circle).

```
% java Josephus 7 2
```

```
1 3 5 0 4 2 6
```

Input

Two integers, N (number of people) and M (step count for elimination), are provided as command-line arguments.

Output

A single line containing space-separated integers representing the order in which people are eliminated.

Sample Input

```
java Josephus 7 2
```

```
java Josephus 5 3
```

```
java Josephus 10 4
```

Sample Output

```
1 3 5 0 4 2 6
```

```
2 0 4 1 3
```

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

DSCP 28**Move-to-front (MTF)**

The Move-to-Front (MTF) strategy maintains a linked list of unique characters from a given input sequence. The list follows these rules:

If a new character appears, insert it at the front of the list.

If a duplicate character appears, remove it from its current position and reinsert it at the front of the list.

Read in a sequence of characters from standard input and maintain the characters in a linked list with no duplicates. When you read in a previously unseen character, insert it at the front of the list. When you read in a duplicate character, delete it from the list and reinsert it at the beginning. Name your program MoveToFront: it implements the well-known move-to-front strategy, which is useful for caching, data compression, and many other applications where items that have been recently accessed are more likely to be reaccessed.

Input

A sequence of characters read from standard input.

The input may contain letters, digits, or special characters.

The input ends when EOF (End of File) is reached.

Output

A single line of space-separated characters representing the linked list from front to back, after processing all input characters.

Sample Input

abracadabra

Processing Steps:

Read 'a' → Insert at front: a

Read 'b' → Insert at front: b a

Read 'r' → Insert at front: r b a

Read 'a' → Move 'a' to front: a r b

Read 'c' → Insert at front: c a r b

Read 'a' → Move 'a' to front: a c r b

Read 'd' → Insert at front: d a c r b

Read 'a' → Move 'a' to front: a d c r b

Read 'b' → Move 'b' to front: b a d c r

Read 'r' → Move 'r' to front: r b a d c

Read 'a' → Move 'a' to front: a r b d c

xyzxyz

Read 'x' → x

Read 'y' → y x

Read 'z' → z y x

Sample Output

a r b d c

z y x

Read 'x' \rightarrow x z y

Read 'y' \rightarrow y x z

Read 'z' \rightarrow z y x

DSCP 29**Stack Generability**

We have a sequence of push (0, 1, ..., $N-1$) and pop (-) operations on a stack. We need to determine: Stack Underflow Detection and Stack Generability.

Given a permutation of numbers (0, 1, ..., $N-1$), can it be a valid output sequence of the stack if we only use push (0, 1, ..., $N-1$) and pop (-) operations?

A valid permutation must be stack sortable, meaning numbers must be popped in the order they appear in the permutation while maintaining the Last-In-First-Out (LIFO) property.

Suppose that we have a sequence of intermixed push and pop operations as with our test stack client, where the integers 0, 1, ..., $N-1$ in that order (push directives) are intermixed with N minus signs (pop directives). Devise an algorithm that determines whether the intermixed sequence causes the stack to underflow. (You may use only an amount of space independent of N —you cannot store the integers in a data structure.) Devise a linear-time algorithm that determines whether a given permutation can be generated as output by our test client (depending on where the pop directives occur).

Input

For Stack Underflow Detection:

A sequence of push operations (0, 1, ..., $N-1$) and pop (-) operations.

For Stack Generability:

A single line containing N space-separated integers, representing a permutation of {0, 1, ..., $N-1$ }.

Output

For Stack Underflow Detection:

Print "Underflow" if a pop operation occurs when the stack is empty.

Print "No Underflow" if all pop operations are valid.

For Stack Generability:

Print "Yes" if the given permutation can be generated by a valid push/pop sequence.

Print "No" otherwise.

Sample Input

- 0 - 1 - 2 - 3 - -

Processing Steps:

Pop (-) → Underflow!

Push 0

Pop (-) → OK

Push 1

Pop (-) → OK

Push 2

Pop (-) → OK

Push 3

Pop (-) → OK

Sample Output

Underflow

Pop (-) → Underflow!

0 1 2 - 3 4 - - -

No Underflow

Processing Steps:

Push 0

Push 1

Push 2

Pop (-) → OK

Push 3

Push 4

Pop (-) → OK

Pop (-) → OK

Pop (-) → OK

4 3 2 1 0

Yes

Processing Steps:

Push 0 → Push 1 → Push 2 → Push 3 → Push 4

Pop 4 → Pop 3 → Pop 2 → Pop 1 → Pop 0

Valid stack sequence

DSCP 30	Binary Search with only Addition and Subtraction
<p>Given a sorted array of N distinct integers in ascending order, determine whether a target integer exists in the array. You must implement a binary search algorithm using only addition and subtraction (no multiplication, division, bitwise operations, or indexing using <code>arr[mid] = arr[low + high / 2]</code>).</p> <p>The goal is to achieve a logarithmic time complexity ($O(\log N)$) in the worst case while using only a constant amount of extra memory.</p> <p>Write a program that, given an array of N distinct int values in ascending order, determines whether a given integer is in the array. You may use only additions and subtractions and a constant amount of extra memory. The running time of your program should be proportional to $\log N$ in the worst case.</p>	
<p>Input</p> <p>The first line contains an integer N (size of the array).</p> <p>The second line contains N distinct integers in ascending order.</p> <p>The third line contains an integer T (target integer to search for).</p>	
<p>Output</p> <p>Print "Found" if T is in the array.</p> <p>Print "Not Found" if T is not in the array.</p>	
<p>Sample Input</p> <p>10</p> <p>1 3 5 7 9 11 13 15 17 19</p> <p>7</p> <p>7</p> <p>2 4 6 8 10 12 14</p> <p>5</p>	<p>Sample Output</p> <p>Found</p> <p>Not Found</p>

DSCP 31**Throwing Eggs from a Building**

You have an N -story building and an unlimited number of eggs. You need to determine the highest safe floor (F) from which you can drop an egg without breaking it. If an egg breaks when dropped from floor F or higher, all floors above F will also break the egg. Floors below F are safe.

Your task is to devise strategies for two different cases:

Unlimited eggs: Find F in $\sim \lg N$ throws with $\sim \lg N$ broken eggs. Then, optimize to $\sim 2 \lg F$ throws.

Two eggs only: Find F in at most $\sim 2\sqrt{N}$ throws and optimize to $\sim c\sqrt{F}$ throws.

Suppose that you have an N -story building and plenty of eggs. Suppose also that an egg is broken if it is thrown off floor F or higher, and unhurt otherwise. First, devise a strategy to determine the value of F such that the number of broken eggs is $\sim \lg N$ when using $\sim \lg N$ throws, then find a way to reduce the cost to $\sim 2 \lg F$.

Consider the question, but now suppose you only have two eggs, and your cost model is the number of throws. Devise a strategy to determine F such that the number of throws is at most $2\sqrt{N}$, then find a way to reduce the cost to $\sim c\sqrt{F}$. This is analogous to a situation where search hits (egg intact) are much cheaper than misses (egg broken).

Input

A single integer N representing the number of floors in the building.

A single integer F (hidden from the program, used for simulation) representing the threshold floor where the egg starts breaking.

Output

A sequence of floor numbers where eggs are dropped.

The final identified floor F .

The total number of egg drops used.

Sample Input

Unlimited Eggs ($\sim \lg N$ throws)

$N = 100$

$F = 37$

Optimized Unlimited Eggs ($\sim 2 \lg F$ throws)

$N = 100$

$F = 37$

Two Eggs ($\sim 2\sqrt{N}$ throws)

$N = 100$

$F = 37$

Optimized Two Eggs ($\sim c\sqrt{F}$ throws)

$N = 100$

$F = 37$

Sample Output

Floors tested: 50, 25, 37, 31, 34, 36, 37

Threshold Floor: 37

Total Drops: 7

Floors tested: 1, 2, 4, 8, 16, 32, 64, 48, 40, 36, 38, 37

Threshold Floor: 37

Total Drops: 11

Floors tested: 10, 20, 30, 40, 31, 32, 33, 34, 35, 36, 37

Threshold Floor: 37

Total Drops: 11

Floors tested: 9, 17, 24, 30, 35, 39, 36, 37

Threshold Floor: 37

Total Drops: 8

DSCP 32**Hot or Cold**

Your goal is to guess a secret integer between 1 and N . You repeatedly guess integers between 1 and N . After each guess you learn if your guess equals the secret integer (and the game stops). Otherwise, you learn if the guess is hotter (closer to) or colder (farther from) the secret number than your previous guess. Design an algorithm that finds the secret number in at most $\sim 2 \lg N$ guesses. Then design an algorithm that finds the secret number in at most $\sim 1 \lg N$ guesses.

You need to guess a secret integer (S) between 1 and N . After each guess:

If the guess is correct, the game stops.

Otherwise, you receive feedback:

"Hotter" if the new guess is closer to S than the previous guess.

"Colder" if the new guess is farther from S than the previous guess.

Your goal is to minimize the number of guesses to find S :

$\sim 2 \log N$ strategy (Basic version).

$\sim 1 \log N$ strategy (Optimized version).

Input

A single integer N (upper bound of the secret number).

A hidden integer S (the secret number, unknown to the player).

The program takes multiple guesses from 1 to N and provides feedback.

Output

A sequence of guessed numbers.

Feedback ("Hotter", "Colder", or "Correct!").

The total number of guesses used.

Sample Input

Basic Strategy ($\sim 2 \log N$ guesses)

$N = 100$

Secret = 73

Sample Output

Guess: 50 → No feedback

Guess: 80 → Colder

Guess: 60 → Hotter

Guess: 70 → Hotter

Guess: 75 → Colder

Guess: 72 → Hotter

Guess: 73 → Correct!

Total Guesses: 7

Optimized Strategy ($\sim 1 \log N$ guesses)

$N = 100$

Secret = 73

Guess: 1 → No feedback

Guess: 2 → Hotter

Guess: 4 → Hotter

Guess: 8 → Hotter

Guess: 16 → Hotter

Guess: 32 → Hotter

Guess: 64 → Hotter

Guess: 80 → Colder

Guess: 72 → Hotter

Guess: 75 → Colder

Guess: 73 → Correct!

Total Guesses: 6

DSCP 33**Random Connections**

Develop a UF client ErdosRenyi that takes an integer value N from the command line, generates random pairs of integers between 0 and $N-1$, calling `connected()` to determine if they are connected and then `union()` if not (as in our development client), looping until all sites are connected, and printing the number of connections generated. Package your program as a static method `count()` that takes N as argument and returns the number of connections and a `main()` that takes N from the command line, calls `count()`, and prints the returned value.

The problem requires simulating random connections between N sites (numbered 0 to $N-1$) using Union-Find (UF) data structure. The process is as follows:

Randomly generate pairs (p, q) , where $0 \leq p, q < N$.

Check if p and q are already connected using `connected(p, q)`.

If not connected, call `union(p, q)` to join them.

Repeat steps 1–3 until all sites are connected (i.e., exactly one connected component remains).

Print the total number of connections (pairs generated) to fully connect all sites.

Input

A single integer N from the command line representing the number of sites.

Output

The total number of random connections generated until all sites are connected.

Sample Input

$N = 10$

$N = 1000$

Sample Output

Total number of connections: 10

Total number of connections: 6153

DSCP 34**Sublinear Extra Space**

Develop a merge implementation that reduces the extra space requirement to $\max(M, N/M)$, based on the following idea: Divide the array into N/M blocks of size M (for simplicity in this description, assume that N is a multiple of M). Then, (i) considering the blocks as items with their first key as the sort key, sort them using selection sort; and (ii) run through the array merging the first block with the second, then the second block with the third, and so forth.

The goal is to implement Merge Sort while minimizing the extra space required to $\max(M, N/M)$.

Instead of using $O(N)$ extra space, we use the following divide-and-conquer approach:

- Partitioning: Divide the array into N/M blocks of size M .
- Block Sorting: Treat each block as an item, using its first key as a sorting key, and sort the blocks using Selection Sort.
- In-place Merging: Merge blocks iteratively: Merge the first block with the second. Merge the second block with the third. Continue merging until the entire array is sorted.
- Result: The array is fully sorted with only $\max(M, N/M)$ extra space instead of $O(N)$.

Input

An integer N : the number of elements in the array.

An integer M : the block size for partitioning.

An array of N integers to be sorted.

Output

The sorted array after applying the merge process.

Sample Input

$N = 8$

$M = 2$

Array = [8, 4, 7, 3, 2, 5, 1, 6]

$N = 12$

$M = 3$

Array = [12, 7, 9, 1, 6, 4, 8, 5, 11, 3, 10, 2]

Sample Output

Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8]

Sorted Array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

DSCP 35**Nuts and Bolts**

You are given N nuts and N bolts, where each nut matches exactly one bolt. Each bolt matches exactly one nut. You cannot directly compare two nuts or two bolts. You can only compare a nut with a bolt to determine whether:

The nut is smaller than the bolt.

The nut is larger than the bolt.

The nut and bolt match.

The goal is to efficiently match each nut to its corresponding bolt using the fewest number of comparisons.

You have a mixed pile of N nuts and N bolts and need to quickly find the corresponding pairs of nuts and bolts. Each nut matches exactly one bolt, and each bolt matches exactly one nut. By fitting a nut and bolt together, you can see which is bigger, but it is not possible to directly compare two nuts or two bolts. Give an efficient method for solving the problem.

Input

Integer N : the number of nuts and bolts.

Two arrays of size N :

nuts[]: An array containing N distinct nuts.

bolts[]: An array containing N distinct bolts.

These arrays are in random order initially.

Output

N pairs of matched (nut, bolt) in sorted order.

Sample Input

$N = 5$

Nuts = [4, 2, 1, 5, 3]

Bolts = [3, 1, 2, 5, 4]

$N = 4$

Nuts = [10, 7, 8, 6]

Bolts = [8, 10, 6, 7]

Sample Output

Matched Pairs:

Nut: 1 - Bolt: 1

Nut: 2 - Bolt: 2

Nut: 3 - Bolt: 3

Nut: 4 - Bolt: 4

Nut: 5 - Bolt: 5

Matched Pairs:

Nut: 6 - Bolt: 6

Nut: 7 - Bolt: 7

Nut: 8 - Bolt: 8

Nut: 10 - Bolt: 10

DSCP 36**Median-of-5 Partitioning**

The goal is to implement a Quicksort algorithm where the partitioning is done using the median of a random sample of five items from the subarray. This technique is expected to improve the performance of Quicksort by reducing the chances of worst-case behaviour (e.g., when the pivot is always the smallest or largest element).

Implement a quicksort based on partitioning on the median of a random sample of five items from the subarray. Put the items of the sample at the appropriate ends of the array so that only the median participates in partitioning. Run doubling tests to determine the effectiveness of the change, in comparison both to the standard algorithm and to median-of-3 partitioning (see the previous exercise). Devise a median-of-5 algorithm that uses fewer than seven compares on any input.

Additionally, you need to:

Compare the median-of-5 approach with the standard Quicksort partitioning (which typically uses the first element or a random element as a pivot).

Compare it with the median-of-3 approach, where the pivot is chosen as the median of the first, middle, and last elements of the subarray.

Input

Array A[]: An array of N integers that needs to be sorted.

Output

A sorted array.

Sample Input

Array = [12, 3, 5, 7, 19, 2, 8, 10, 17, 6]

Array = [21, 18, 13, 25, 10, 6, 15, 30]

Sample Output

Sorted Array: [2, 3, 5, 6, 7, 8, 10, 12, 17, 19]

Sorted Array: [6, 10, 13, 15, 18, 21, 25, 30]

DSCP 37**Priority Queue with Explicit Links**

Implement a priority queue using a heap ordered binary tree, but use a triply linked structure instead of an array. You will need three links per node: two to traverse down the tree and one to traverse up the tree. Your implementation should guarantee logarithmic running time per operation, even if no maximum priority-queue size is known ahead of time.

You are tasked with implementing a priority queue using a heap-ordered binary tree with a triply linked structure. Each node in the tree should have three links:

A link to its left child.

A link to its right child.

A link to its parent.

The priority queue should maintain the heap property, i.e., for a max heap, the value of each node is greater than or equal to the values of its children, and for a min heap, the value of each node is less than or equal to the values of its children. The implementation should support logarithmic time complexity for all operations, even if the maximum size of the priority queue is not known ahead of time.

Input

The first line of input contains an integer N , the number of elements to be inserted into the priority queue.

The next N lines each contain an integer representing the element to be inserted into the priority queue.

The subsequent lines contain the operations to be performed on the priority queue:

Insert x : Insert the element x into the priority queue.

Delete: Remove and return the highest (or lowest) priority element.

Peek: Return the highest (or lowest) priority element.

Size: Return the current number of elements in the queue.

IsEmpty: Return whether the queue is empty (yes/no).

Output

For Insert x : No output is required.

For Delete: Output the element that was removed from the queue.

For Peek: Output the current highest (or lowest) priority element.

For Size: Output the number of elements currently in the queue.

For IsEmpty: Output "yes" if the queue is empty, otherwise "no".

Sample Input

5
10
20
5
30

Sample Output

30
30
20
4
no

15

Insert 10

Insert 20

Insert 5

Insert 30

Insert 15

Peek

Delete

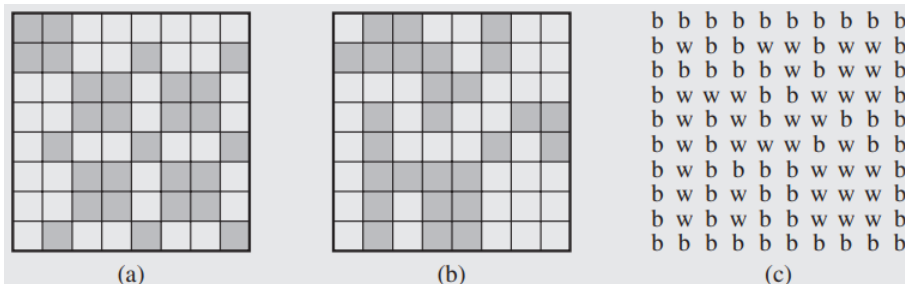
Peek

Size

IsEmpty

DSCP 38**Counting White Regions in an $n \times n$ Grid**

An $n \times n$ square consists of black and white cells arranged in a certain way. The problem is to determine the number of white areas and the number of white cells in each area. For example, a regular 8×8 chessboard has 32 one-cell white areas; the square in Figure (a) consists of 10 areas, 2 of them of 10 cells, and 8 of 2 cells; the square in Figure (b) has 5 white areas of 1, 3, 21, 10, and 2 cells. Write a program that, for a given $n \times n$ square, outputs the number of white areas and their sizes. Use an $(n + 2) \times (n + 2)$ array with properly marked cells. Two additional rows and columns constitute a frame of black cells surrounding the entered square to simplify your implementation. For instance, the square in Figure (b) is stored as the square in Figure (c).



Traverse the square row by row and, for the first unvisited cell encountered, invoke a function that processes one area. The secret is in using four recursive calls in this function for each unvisited white cell and marking it with a special symbol as visited (counted).

Input

An integer n (size of the square grid).

An $n \times n$ grid consisting of characters:

'W' for white cells

'B' for black cells

Output

An integer representing the number of white areas.

A list of integers representing the sizes of each white area, sorted in ascending order.

Sample Input

```
5
W B W W B
W B W W B
B W B B W
W W W B W
W B B W W
4
W W B W
B W W B
B B W W
W B W B
```

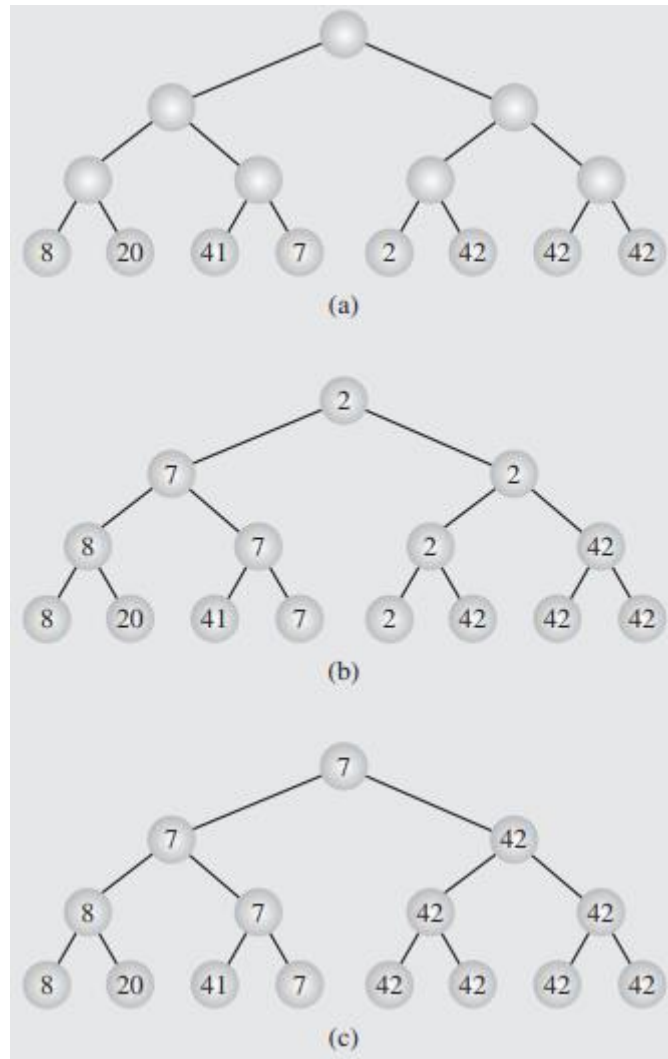
Sample Output

```
4
[1, 2, 5, 6]
3
[1, 3, 5]
```


DSCP 39	Reversed and Mirrored Tree Traversals
<p>Consider an operation R that for a given traversal method t processes nodes in the opposite order than t and an operation C that processes nodes of the mirror image of a given tree using traversal method t.</p> <p>$R(t)$: Processes nodes in the reverse order of traversal method t.</p> <p>$C(t)$: Processes nodes of the mirror image of a given tree using traversal method t.</p> <p>For the tree traversal methods—preorder, inorder, and postorder - determine which of the following nine equalities are true:</p> <p>$R(\text{preorder}) = C(\text{preorder})$</p> <p>$R(\text{preorder}) = C(\text{inorder})$</p> <p>$R(\text{preorder}) = C(\text{postorder})$</p> <p>$R(\text{inorder}) = C(\text{preorder})$</p> <p>$R(\text{inorder}) = C(\text{inorder})$</p> <p>$R(\text{inorder}) = C(\text{postorder})$</p> <p>$R(\text{postorder}) = C(\text{preorder})$</p> <p>$R(\text{postorder}) = C(\text{inorder})$</p> <p>$R(\text{postorder}) = C(\text{postorder})$</p> <p>Using inorder, preorder, and postorder tree traversal, visit only leaves of a tree. What can you observe? How can you explain this phenomenon?</p>	
<p>Input</p> <p>An integer n representing the number of nodes in the binary tree.</p> <p>n lines follow, each containing three values:</p> <p>Node Value</p> <p>Left Child (-1 if none)</p> <p>Right Child (-1 if none)</p> <p>A string t representing the traversal type ("preorder", "inorder", or "postorder").</p>	
<p>Output</p> <p>The standard traversal order of the given tree using traversal t.</p> <p>The result of applying operation $R(t)$.</p> <p>The result of applying operation $C(t)$.</p> <p>The result of applying both R and C on t.</p>	
<p>Sample Input</p> <p>7</p> <p>1 2 3</p> <p>2 4 5</p> <p>3 6 7</p>	<p>Sample Output</p> <p>Standard Preorder: 1 2 4 5 3 6 7</p> <p>$R(\text{Preorder})$: 7 6 3 5 4 2 1</p>

4 -1 -1	C(Preorder): 1 3 7 6 2 5 4
5 -1 -1	R(C(Preorder)): 4 5 2 6 7 3 1
6 -1 -1	
7 -1 -1	
preorder	
5	Standard Inorder: 2 4 1 3 5
1 2 3	R(Inorder): 5 3 1 4 2
2 -1 4	C(Inorder): 5 3 1 4 2
3 -1 5	R(C(Inorder)): 2 4 1 3 5
4 -1 -1	
5 -1 -1	
inorder	

A binary tree can be used to sort n elements of an array data. First, create a complete binary tree, a tree with all leaves at one level, whose height $h = \lceil \lg n \rceil + 1$, and store all elements of the array in the first n leaves. In each empty leaf, store an element E greater than any element in the array. Figure (a) shows an example for data = {8, 20, 41, 7, 2}, $h = \lceil \lg(5) \rceil + 1 = 4$, and $E = 42$. Then, starting from the bottom of the tree, assign to each node the minimum of its two children values, as in Figure (b), so that the smallest element e_{\min} in the tree is assigned to the root. Next, until the element E is assigned to the root, execute a loop that in each iteration stores E in the leaf, with the value of e_{\min} , and that, also starting from the bottom, assigns to each node the minimum of its two children. Figure (c) displays this tree after one iteration of the loop.



Input

An integer n representing the number of elements in the array.

A line containing n space-separated integers representing the elements of the array data.

Output

The initial complete binary tree representation with elements stored in the first n leaves.

The binary tree after assigning the minimum values to internal nodes.

The binary tree after each iteration of replacing the smallest element with E and rebalancing.
The sorted sequence obtained from the tree.

Sample Input

5
8 20 41 7 2

Sample Output

Initial Binary Tree (leaves filled with data, others with E = 42):

42

/ \

42 42

/ \ / \

8 20 41 7

/

2

Binary Tree after assigning min values:

2

/ \

7 7

/ \ / \

8 20 41 7

/

2

After one iteration (smallest element replaced with E = 42):

7

/ \

7 7

/ \ / \

8 20 41 7

/

42

Final sorted sequence:

2 7 8 20 41

Implement a menu-driven program for managing a software store. All information about the available software is stored in a file software. This information includes the name, version, quantity, and price of each package. When it is invoked, the program automatically creates a binary search tree with one node corresponding to one software package and includes as its key the name of the package and its version. Another field in this node should include the position of the record in the file software. The only access to the information stored in software should be through this tree. The program should allow the file and tree to be updated when new software packages arrive at the store and when some packages are sold. The tree is updated in the usual way. All packages are entry ordered in the file software; if a new package arrives, then it is put at the end of the file. If the package already has an entry in the tree (and the file), then only the quantity field is updated. If a package is sold out, the corresponding node is deleted from the tree, and the quantity field in the file is changed to 0. For example, if the file has these entries:

Adobe Photoshop	CS5	21	580
Norton Utilities		10	50
Norton SystemWorks	2009	6	50
Visual Studio Professional	2010	19	700
Microsoft Office	2010	27	150

then after selling all six copies of Norton SystemWorks 2009, the file is

Adobe Photoshop	CS5	21	580
Norton Utilities		10	50
Norton SystemWorks	2009	0	50
Visual Studio Professional	2010	19	700
Microsoft Office	2010	27	150

If an exit option is chosen from the menu, the program cleans up the file by moving entries from the end of the file to the positions marked with 0 quantities. For example, the previous file becomes

Adobe Photoshop	CS5	21	580
Norton Utilities		10	50
Microsoft Office	2010	27	150
Visual Studio Professional	2010	19	700

Input

An integer n representing the number of software packages in the initial file.

n lines follow, each containing:

Software Name (string)

Version (string)

Quantity (integer)

Price (integer)

A series of menu-driven commands:

1 ADD [Software Name] [Version] [Quantity] [Price] → Add a new package or update quantity.

2 SELL [Software Name] [Version] [Quantity] → Sell the specified quantity.

3 DISPLAY → Show the current inventory from the binary search tree.

4 EXIT → Clean up the file and terminate the program.

Output

A message confirming each operation (added, updated, sold, or error).

The inventory display as a table.

The final cleaned-up file contents when exiting.

Sample Input

```
5
Adobe Photoshop CS5 21 580
Norton Utilities 10 50
Norton SystemWorks 2009 6 50
Visual Studio Professional 2010 19 700
Microsoft Office 2010 27 150

1 ADD Adobe Photoshop CS5 5 580
2 SELL Norton SystemWorks 2009 6
3 DISPLAY
4 EXIT
```

Sample Output

Software "Adobe Photoshop CS5" updated with 5 more copies.

Sold 6 copies of "Norton SystemWorks 2009".

Current Inventory:

```
-----
| Software Name      | Version | Quantity | Price |
|-----|-----|-----|-----|
| Adobe Photoshop    | CS5     | 26      | 580   |
| Norton Utilities    | 10      | 50      | 50     |
| Norton SystemWorks | 2009    | 0       | 50     |
| Visual Studio Pro   | 2010    | 19      | 700    |
| Microsoft Office    | 2010    | 27      | 150    |
-----
```

File cleanup complete. Final File:

```
-----
```

Software Name	Version	Quantity	Price
----- ----- ----- -----			
Adobe Photoshop	CS5	26	580
Norton Utilities	10	50	50
Microsoft Office	2010	27	150
Visual Studio Pro	2010	19	700

Exiting program.			

Each unit in a Latin textbook contains a Latin-English vocabulary of words that have been used for the first time in a particular unit. Write a program that converts a set of such vocabularies stored in file Latin into a set of English-Latin vocabularies. Make the following assumptions: a. Unit names are preceded by a percentage symbol. b. There is only one entry per line. c. A Latin word is separated by a colon from its English equivalent(s); if there is more than one equivalent, they are separated by a comma. To output English words in alphabetical order, create a binary search tree for each unit containing English words and linked lists of Latin equivalents. Make sure that there is only one node for each English word in the tree. For example, there is only one node for and, although and is used twice in unit 6: with words ac and atque. After the task has been completed for a given unit (that is, the content of the tree has been stored in an output file), delete the tree along with all linked lists from computer memory before creating a tree for the next unit.

Here is an example of a file containing Latin-English vocabularies:

%Unit 5

ante : before, in front of, previously

antiquus : ancient

ardeo : burn, be on fire, desire

arma : arms, weapons

aurum : gold

aureus : golden, of gold

%Unit 6

animal : animal

Athenae : Athens

atque : and

ac : and

aurora : dawn

%Unit 7

amo : love

amor : love

annus : year

Asia : Asia

From these units, the program should generate the following output:

%Unit 5

ancient : antiquus

arms : arma

be on fire : ardeo

before : ante

burn : ardeo

desire : ardeo

gold: aurum
golden : aureus
in front of : ante
of gold : aureus
previously : ante
weapons : arma
%Unit 6
Athens : Athenae
and : ac, atque
animal : animal
dawn : aurora
%Unit 7
Asia : Asia
love : amor, amo
year : annus

Input

A text file (Latin.txt) containing Latin-English vocabulary for multiple units.
Each unit starts with %Unit X, where X is the unit number.
Each vocabulary entry is on a new line, formatted as:
Latin_Word : English_Equivalent1, English_Equivalent2, ...
Multiple English equivalents are separated by commas.

Output

A text file (English.txt) containing the reversed English-Latin vocabulary for each unit.
Each unit starts with %Unit X, maintaining the original unit order.
Each English word appears only once, followed by all corresponding Latin words, in alphabetical order.
Multiple Latin words corresponding to the same English word are separated by commas.

Sample Input

%Unit 5
ante : before, in front of, previously
antiquus : ancient
ardeo : burn, be on fire, desire
arma : arms, weapons
aurum : gold
aureus : golden, of gold

%Unit 6

Sample Output

%Unit 5
ancient : antiquus
arms : arma
be on fire : ardeo
before : ante
burn : ardeo

animal : animal

Athenae : Athens

atque : and

ac : and

aurora : dawn

%Unit 7

amo : love

amor : love

annus : year

Asia : Asia

desire : ardeo

gold : aurum

golden : aureus

in front of : ante

of gold : aureus

previously : ante

weapons : arma

%Unit 6

Athens : Athenae

and : ac, atque

animal : animal

dawn : aurora

%Unit 7

Asia : Asia

love : amor, amo

year : annus

DSCP 43**Cross-Reference Generator**

Write a cross-reference program that constructs a binary search tree with all words included from a text file and records the line numbers on which these words were used. These line numbers should be stored on linked lists associated with the nodes of the tree. After the input file has been processed, print in alphabetical order all words of the text file along with the corresponding list of numbers of the lines in which the words occur.

Input

A text file containing multiple lines of text.

Words are case-insensitive (e.g., "Apple" and "apple" should be treated as the same word).

Words consist only of alphabetic characters (punctuation should be ignored).

Each word should be stored only once in the BST, with a linked list of line numbers tracking its occurrences.

Output

Each word should be printed in alphabetical order, followed by the list of line numbers where the word appears.

The output format should be:

word: line_number1, line_number2, ...

Words should appear in lowercase in the output.

Sample Input

The quick brown fox jumps over the lazy dog.

A quick brown fox is very agile.

Foxes are known for their quick movements.

Sample Output

a: 2

agile: 2

are: 3

brown: 1, 2

dog: 1

fox: 1, 2

foxes: 3

for: 3

is: 2

jumps: 1

known: 3

lazy: 1

movements: 3

over: 1

quick: 1, 2, 3

the: 1

their: 3

very: 2

DSCP 44**Digital Trees for Spell Checking**

A variant of a trie is a digital tree, which processes information on the level of bits. Because there are only two bits, only two outcomes are possible. Digital trees are binary. For example, to test whether the word "BOOK" is in the tree, we do not use the first letter, "B," in the root to determine to which of its children we should go, but the first bit, 0, of the first letter (ASCII (B) 5 01000010), on the second level, the second bit, and so on before we get to the second letter. Is it a good idea to use a digital tree for a spell checking program, as was discussed in the case study?

In this problem, you will implement a digital tree and use it for spell checking. The tree should allow words to be inserted and checked for existence efficiently using bitwise operations. Given a set of words, the program should build the tree and determine if a given list of query words exist in the dictionary.

Construct a digital tree from a given dictionary of words.

Process multiple queries to check whether given words exist in the tree.

Input

An integer n representing the number of words in the dictionary.

n lines follow, each containing a single word (only uppercase English letters).

An integer q representing the number of query words.

q lines follow, each containing a single word to be checked.

Output

For each query word, print "YES" if it exists in the tree, otherwise print "NO".

Sample Input

5
BOOK
TREE
TRIE
BINARY
SEARCH

3

BOOK
CODE
TREE

6

HELLO
WORLD
PYTHON
PROGRAM
DATA
STRUCTURE

Sample Output

YES
NO
YES

YES
NO
YES
NO

4

PYTHON

JAVA

HELLO

CODE

DSCP 45**Analysis of Tournaments in Directed Graphs**

A tournament is a digraph in which there is exactly one edge between every two vertices. In this problem, you need to analyse various properties of tournaments and answer the following questions:

Number of Edges: Given a tournament with n vertices, determine the number of edges in the tournament.

Number of Different Tournaments: Given n vertices, determine the number of different tournaments that can be created.

Topological Sorting: Determine if every tournament can be topologically sorted.

Number of Minimal Vertices: Find out how many minimal vertices a tournament can have.

Transitive Tournament and Cycles: Determine whether a transitive tournament (where if there is an edge from v to u and from u to w , then there must be an edge from v to w) can have a cycle.

- How many edges does a tournament have?
- How many different tournaments of n edges can be created?
- Can each tournament be topologically sorted?
- How many minimal vertices can a tournament have?
- A transitive tournament is a tournament that has edge (vw) if it has edge (vu) and edge (uw) . Can such a tournament have a cycle?

Input

A single integer n representing the number of vertices in the tournament.

Output

Print five lines, each corresponding to the answers for the above questions:

The number of edges in the tournament.

The number of different tournaments possible with n vertices.

"YES" if every tournament can be topologically sorted, otherwise "NO".

The minimum number of minimal vertices in the tournament.

"YES" if a transitive tournament can have a cycle, otherwise "NO".

Sample Input

4

3

Sample Output

6

64

YES

1

NO

3

8

YES

1

NO

DSCP 46**Quick-Fit Memory Allocation with Block Coalescing**

Implement the following memory allocation method developed by W. A. Wulf, C. B. Weinstock, and C. B. Johnsson (Standish 1980) called the quick-fit method. For an experimentally found number n of the most frequently requested sizes of blocks, this method uses an array avail of $n + 1$ cells, each cell i pointing to a linked list of blocks of size i . The last cell ($n + 1$) refers to a block of other less frequently needed sizes. It may also be a pointer to a linked list, but because of possibly a large number of such blocks, another organization is recommended, such as a binary search tree. Write functions to allocate and deallocate blocks. If a block is returned, coalesce it with its neighbours. To test your program, randomly generate sizes of blocks to be allocated from memory simulated by an array whose size is a power of 2.

This method efficiently manages memory by maintaining an array avail of size $n+1$, where:

The first n cells point to linked lists of frequently requested block sizes. The $(n+1)^{\text{th}}$ cell handles less frequently requested sizes, possibly using a binary search tree or another suitable structure. When a block is allocated, it is taken from the appropriate list. If no exact match is found, a larger block is split. When a block is deallocated, it is returned to the list and merged (coalesced) with adjacent free blocks. Memory is simulated using an array of size power of 2.

Write functions to:

Allocate a block of a given size.

Deallocate a block, merging it with adjacent free blocks if possible.

Simulate memory operations by randomly generating block sizes for allocation and deallocation.

Input

An integer M (size of the memory, a power of 2).

An integer n (number of frequently requested sizes).

n integers representing the n most frequently requested block sizes.

An integer Q (number of memory operations).

Q operations in the form:

ALLOC $x \rightarrow$ Allocate a block of size x .

FREE $y \rightarrow$ Free the block at index y .

Output

For each ALLOC operation:

If allocation is successful, print "ALLOCATED <index>" (where <index> is the starting position in memory).

If allocation fails, print "FAILED".

For each FREE operation:

If the block is successfully freed, print "FREED <index>".

If the block is invalid or not allocated, print "INVALID FREE".

Sample Input

16

Sample Output

ALLOCATED 0

3	ALLOCATED 4
2 4 8	ALLOCATED 6
6	FREED 4
ALLOC 4	ALLOCATED 4
ALLOC 2	ALLOCATED 10
ALLOC 8	
FREE 1	
ALLOC 2	
ALLOC 4	
32	ALLOCATED 0
2	ALLOCATED 8
4 8	ALLOCATED 4
5	ALLOCATED 16
ALLOC 8	FAILED
ALLOC 4	
ALLOC 8	
ALLOC 8	
ALLOC 8	

DSCP 47**Efficient Substitute Teacher Allocation
Using BST**

The board of education maintains a database of substitute teachers in the area. If a temporary replacement is necessary for a certain subject, an available teacher is sent to the school that requests him or her. Write a menu-driven program maintaining this database.

The file substitutes lists the first and last names of the substitute teachers, an indication of whether or not they are currently available (Y(es) or N(o)), and a list of numbers that represents the subjects they can teach. An example of substitutes is:

Hilliard Roy Y 0 4 5

Ennis John N 2 3

Nelson William Y 1 2 4 5

Baird Lyle Y 1 3 4 5

Geiger Melissa N 3 5

Kessel Warren Y 3 4 5

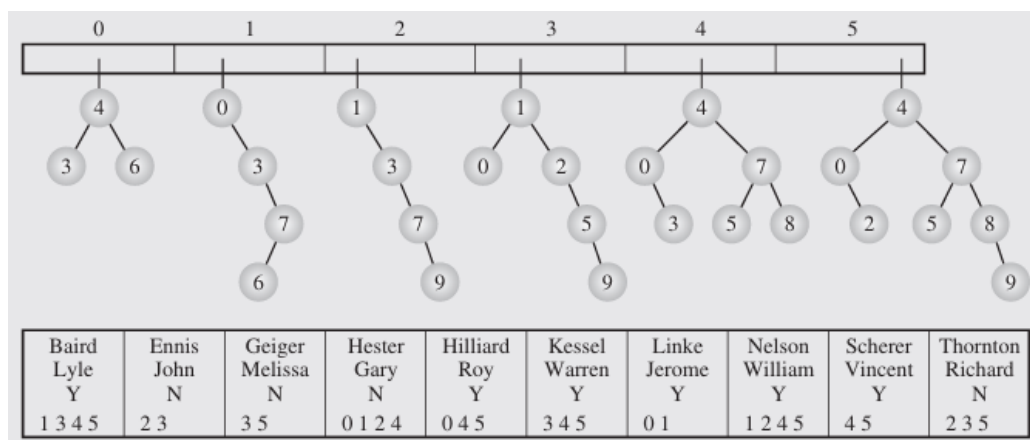
Scherer Vincent Y 4 5

Hester Gary N 0 1 2 4

Linke Jerome Y 0 1

Thornton Richard N 2 3 5

Create a class teacher with three data members: index, left child, and right child. Declare an array subjects with the same number of cells as the number of subjects, each cell storing a pointer to class Teacher, which is really a pointer to the root of a binary search tree of teachers teaching a given subject. Also, declare an array names with each cell holding one entry from the file. First, prepare the array names to create binary search trees. To do that, load all entries from substitutes to names and sort names using one of the algorithms discussed in this chapter. Afterward, create a binary tree using the function `balance()` and go through the array names, and for each subject associated with each name, create a node in the tree corresponding to this subject. The index field of such a node indicates a location in names of a teacher teaching this subject. (Note that `insert()` in `balance()` should be able to go to a proper tree.) Figure below shows the ordered array names and trees accessible from subjects as created by `balance()` for our sample file.



Allow the user to reserve teachers if so requested. If the program is finished and an exit option is chosen, load all entries from names back to substitutes, this time with updated availability information.

Input

<First Name> <Last Name> <Availability (Y/N)> <Subjects (space-separated integers)>

1. Show available teachers for a subject
2. Reserve a substitute teacher
3. Release a substitute teacher
4. Exit

Output

Show available teachers for a subject

Available teachers for subject <X>:

1. <First Name> <Last Name>
2. <First Name> <Last Name>

...

If no teachers are available:

No available teachers for subject <X>.

Reserve a substitute teacher

<First Name> <Last Name> has been reserved for subject <X>.

If the teacher is unavailable or does not teach the subject:

Error: <First Name> <Last Name> is not available for subject <X>.

Release a substitute teacher

<First Name> <Last Name> is now available.

If the teacher was not reserved:

Error: <First Name> <Last Name> is not currently reserved.

Exit and save data

Data saved successfully. Exiting...

Sample Input

Hilliard Roy Y 0 4 5
 Ennis John N 2 3
 Nelson William Y 1 2 4 5
 Baird Lyle Y 1 3 4 5
 Geiger Melissa N 3 5
 Kessel Warren Y 3 4 5
 Scherer Vincent Y 4 5
 Hester Gary N 0 1 2 4
 Linke Jerome Y 0 1
 Thornton Richard N 2 3 5

Sample Output

Menu:
 1. Show available teachers for a subject
 2. Reserve a substitute teacher
 3. Release a substitute teacher
 4. Exit
 Enter your choice: 1
 Enter subject number: 4
 Available teachers for subject 4:
 1. Hilliard Roy
 2. Nelson William
 3. Baird Lyle

4. Kessel Warren

5. Scherer Vincent

Enter your choice: 2

Enter subject number: 4

Enter teacher name: Nelson William

Nelson William has been reserved for subject 4.

Enter your choice: 3

Enter teacher name: Nelson William

Nelson William is now available.

Enter your choice: 4

Data saved successfully. Exiting...

DSCP 48**Hash Function Efficiency with Linear Probing**

Linear probing technique used for collision resolution has a rapidly deteriorating performance if a relatively small percentage of the cells are available. This problem can be solved using another technique for resolving collisions, and also by finding a better hash function, ideally, a perfect hash function. Write a program that evaluates the efficiency of various hashing functions combined with the linear probing method. Have your program write a table, which gives the averages for successful and unsuccessful trials of locating items in the table. Use functions for operating on strings and a large text file whose words will be hashed to the table. Here are some examples of such functions (all values are divided modulo TSize):

- FirstLetter(s) + SecondLetter(s) + ... + LastLetter(s)
- FirstLetter(s) + LastLetter(s) + length(s) (Cichelli)
- for (i = 1, index = 0; i < strlen(s); i++) index = (26 * index + s[i] - ' ');

Input

An integer TSize (size of the hash table).

A string hashMethod specifying which hash function to use (SUM, CICHELLI, or POLY).

A large text file containing words (one per line) to be inserted into the hash table.

A list of words to search, ending with STOP.

Output

The program should display the hash table after inserting words.

For each search query:

If the word is found, print "Found <word> in <probes> probes".

If the word is not found, print "Not found <word> after <probes> probes".

At the end, print a performance table summarizing the average probes for successful and unsuccessful searches.

Sample Input

```
100
SUM
words.txt
search.txt

words.txt
apple
banana
grape
orange
cherry
kiwi
mango
```

Sample Output

```
Hash Table (SUM Method, Size: 100)
Index | Word
-----
10    | apple
20    | banana
33    | grape
45    | orange
55    | cherry
77    | kiwi
88    | mango

Search Results:
Found apple in 1 probes
```

search.txt	Found grape in 1 probes	
apple	Not found watermelon after 3 probes	
grape	Performance Summary:	
watermelon	-----	
STOP	Hash Function Avg Probes (Success) Avg Probes (Failure)	

	SUM	1.2 3.5
	CICHELLI	1.5 3.0
	POLY	1.1 2.8

Write a simple airline ticket reservation program. The program should display a menu with the following options: reserve a ticket, cancel a reservation, check whether a ticket is reserved for a particular person, and display the passengers. The information is maintained on an alphabetized linked list of names. In a simpler version of the program, assume that tickets are reserved for only one flight. In a fuller version, place no limit on the number of flights. Create a linked list of flights with each node including a pointer to a linked list of passengers.

Design a simple airline ticket reservation system that allows users to:

Reserve a ticket for a passenger.

Cancel a reservation for a passenger.

Check if a ticket is reserved for a particular person.

Display the list of passengers for a flight.

The information is stored in an alphabetized linked list of names.

In the simpler version, assume there is only one flight.

In the fuller version, allow multiple flights by maintaining a linked list of flights, where each flight node contains a linked list of passengers.

The program should be menu-driven and ensure that passenger lists remain alphabetically ordered after each reservation or cancellation.

Input

An integer N (number of flights).

N lines containing flight numbers (unique identifiers).

A menu with the following options:

1. Reserve a ticket
2. Cancel a reservation
3. Check reservation status
4. Display passengers
5. Exit

For Option 1 (Reserve a ticket):

Input: <Flight Number> <Passenger Name>

For Option 2 (Cancel a reservation):

Input: <Flight Number> <Passenger Name>

For Option 3 (Check reservation status):

Input: <Flight Number> <Passenger Name>

For Option 4 (Display passengers):

Input: <Flight Number>

For Option 5 (Exit), the program should terminate.

Output

For Option 1 (Reserve a ticket):

If successful:

Reservation confirmed for <Passenger Name> on Flight <Flight Number>.

If the passenger is already booked:

<Passenger Name> is already reserved on Flight <Flight Number>.

For Option 2 (Cancel a reservation):

If successful:

Reservation for <Passenger Name> on Flight <Flight Number> has been canceled.

If the passenger is not found:

No reservation found for <Passenger Name> on Flight <Flight Number>.

For Option 3 (Check reservation status):

If found:

<Passenger Name> has a reservation on Flight <Flight Number>.

If not found:

No reservation found for <Passenger Name> on Flight <Flight Number>.

For Option 4 (Display passengers):

If passengers exist:

Passengers on Flight <Flight Number>:

1. Alice Brown
2. John Doe
3. Mark Smith

If no passengers:

No reservations for Flight <Flight Number>.

For Option 5 (Exit):

Exiting the airline reservation system...

Sample Input	Sample Output
2	Reservation confirmed for John Doe on Flight AI101.
AI101	
BA202	Reservation confirmed for Alice Brown on Flight AI101.
1	
AI101 John Doe	Reservation confirmed for Mark Smith on Flight AI101.
1	
AI101 Alice Brown	Passengers on Flight AI101:

1

AI101 Mark Smith

4

AI101

3

AI101 Alice Brown

2

AI101 Alice Brown

4

AI101

5

1. Alice Brown

2. John Doe

3. Mark Smith

Alice Brown has a reservation on Flight AI101.

Reservation for Alice Brown on Flight AI101 has been canceled.

Passengers on Flight AI101:

1. John Doe

2. Mark Smith

Exiting the airline reservation system...

Write a simple line editor. Keep the entire text on a linked list, one line in a separate node. Start the program with entering EDIT file, after which a prompt appears along with the line number. If the letter *I* is entered with a number *n* following it, then insert the text to be followed before line *n*. If *I* is not followed by a number, then insert the text before the current line. If *D* is entered with two numbers *n* and *m*, one *n*, or no number following it, then delete lines *n* through *m*, line *n*, or the current line. Do the same with the command *L*, which stands for listing lines. If *A* is entered, then append the text to the existing lines. Entry *E* signifies exit and saving the text in a file. Here is an example:

EDIT testfile

```
1> The first line
2>
3> And another line
4> I 3
3> The second line
4> One more line
5> L
1> The first line
2>
3> The second line
4> One more line
5> And another line
5> D 2
4> L
1> The first line
2> The second line
3> One more line
4> And another line
4> E
```

Design a simple line editor that maintains the text as a linked list, where each line is stored in a separate node. The editor should support the following operations:

Insert (*I n text*) – Insert a new line before line *n*. If *n* is not provided, insert before the current line.

Delete (*D n m*) – Delete lines from *n* to *m*. If *m* is not provided, delete only line *n*. If no number is provided, delete the current line.

List (*L n m*) – List lines from *n* to *m*. If *m* is not provided, list only line *n*. If no number is provided, list the entire text.

Append (*A text*) – Append new lines to the end of the text.

Exit (*E*) – Save the text to a file and exit the editor.

Input

First input: EDIT filename (specifies the file being edited).

User commands:

I n text → Insert "text" before line n. (If n is omitted, insert before the current line.)

D n m → Delete lines from n to m. (If m is omitted, delete only line n. If n is omitted, delete the current line.)

L n m → List lines from n to m. (If m is omitted, list only line n. If n is omitted, list all lines.)

A text → Append "text" to the existing lines.

E → Save and exit.

Output

The editor displays a prompt (<line_number> >) for user input.

After inserting, deleting, or appending lines, the program updates the numbering accordingly.

The L command prints the specified lines.

The E command saves the text and exits.

Sample Input

```
EDIT testfile
1> The first line
2>
3> And another line
4> I 3 The second line
5> L
6> D 2
7> L
8> E
```

Sample Output

```
1> The first line
2>
3> The second line
4> And another line

(After `D 2` - Delete line 2)
1> The first line
2> The second line
3> And another line

(Saves and exits upon `E`)
```

DSCP 51**High Score Entries for a Video Game**

A video game requires a data structure to efficiently store and manage a sequence of high score entries. The system should support the following operations efficiently:

Insert scores dynamically as new scores are achieved.

Maintain sorted order (highest scores first).

Limit the leaderboard size (e.g., top 10 scores).

Retrieve top-N scores efficiently.

Support player rank lookup.

The solution must be optimized for performance while ensuring efficient insertion, retrieval, and deletion operations.

Input

Number of scores (N) to store in the leaderboard.

List of player score entries, each with:

player_id: Unique identifier for the player.

score: The score achieved.

Output

Sorted leaderboard (highest scores first).

Rank of a given player (if queried).

Top-N scores if requested.

Sample Input

Leaderboard size: 5

Player Scores:

1 -> 500

2 -> 600

3 -> 450

4 -> 700

5 -> 620

6 -> 580

Sample Output

Top 5 High Scores:

1. Player 4 - 700

2. Player 5 - 620

3. Player 2 - 600

4. Player 6 - 580

5. Player 1 - 500

Rank of Player 3: Not in the top 5

DSCP 52**Tic-Tac-Toe**

Tic-Tac-Toe is a two-player game played on a 3×3 grid where players take turns marking an empty cell with 'X' or 'O'. The game must support the following operations efficiently:

Players take turns marking an empty cell with 'X' or 'O'.

The first player to get three marks in a row, column, or diagonal wins.

If the grid is full and no winner is found, the game results in a draw.

The problem is to design an efficient data structure to:

Store the Tic-Tac-Toe board.

Track player moves dynamically.

Check for a win or draw efficiently.

Handle invalid moves.

Input

A 3×3 grid, initialized with '-' (empty spaces).

Players take turns entering their move (row, column).

Game stops when a player wins or all cells are filled.

Output

Updated board after each move.

Message if a player wins ("Player X wins!") or game is a draw ("It's a draw!").

Error message for invalid moves (e.g., placing a mark in an occupied cell).

Sample Input

Player X: (0, 0)

Player O: (1, 1)

Player X: (0, 1)

Player O: (2, 2)

Player X: (0, 2) <-- X wins

Sample Output

X | X | X

| O |

| | O

Player X wins!

DSCP 53	Out-of-Order Packets
<p>In network communication, packets are often received out of order due to factors such as network congestion, varying transmission speeds, or different routing paths. Bob sends a message M to Alice over the Internet. The challenge is to store and reorder these packets efficiently before passing them to the application layer.</p> <p>The message M is split into n numbered packets (P_1, P_2, \dots, P_n).</p> <p>These packets arrive out of order at Alice's computer.</p> <p>Alice needs to reconstruct the original message in order before processing it.</p> <p>Alice knows the total number of packets (n) in advance.</p> <p>Design an efficient data structure for Alice to:</p> <ul style="list-style-type: none"> Store packets in memory as they arrive. Reorder packets efficiently. Output the final message in correct order. 	
<p>Input</p> <p>Total number of packets (n).</p> <p>A list of packet data, each containing:</p> <ul style="list-style-type: none"> packet_id: Unique sequence number (1 to n). data: The message fragment. <p>Packets may arrive in random order.</p>	
<p>Output</p> <p>The reassembled message in correct order.</p> <p>Error handling if packets are missing.</p>	
<p>Sample Input</p> <p>$n = 5$</p> <p>Packets received (out of order):</p> <p>(3, "lo "), (1, "Hel"), (4, "Wor"), (2, "l"), (5, "ld!")</p>	<p>Sample Output</p> <p>Reassembled message: "Hello World!"</p>

DSCP 54**Matching Tags in a Markup Language**

In markup languages like HTML and XML, elements are enclosed within opening (`<tag>`) and closing (`</tag>`) tags. Properly nested and matched tags are essential for the document's validity.

The tags must be properly nested and matched.

The goal is to validate whether a given markup document is well-formed.

Design an efficient stack-based algorithm to:

Scan an HTML/XML document for opening and closing tags.

Detect mismatched, missing, or improperly nested tags.

Output whether the document is valid or contains errors.

Input

A string containing an HTML/XML-like document.

Tags are enclosed in `< >`, and only proper pairs are allowed.

Self-closing tags like `
`, ``, etc., should be ignored

Output

"Valid Markup" if all tags are properly nested.

"Invalid Markup" with error details (mismatch, missing, or extra tags).

Sample Input**Sample Output****Input 1 (Matched tags):**

```
<html>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

Valid Markup

Input 2 (Mismatched tags):

```
<html>
  <body>
    <h1>Hello</h2>
  </body>
</html>
```

Invalid Markup: Mismatched tag, expected '`</h1>`' but found '`</h2>`'.

DSCP 55**Optimizing Alice's Chances in a Game**

Alice has two queues, Q and R , which can store integers.

Bob gives Alice 50 odd and 50 even integers, and she must distribute all 100 integers between Q and R . Bob randomly selects either Q or R and applies a round-robin scheduler for a random number of times.

The last processed number determines the winner:

If the last number is odd, Bob wins.

If the last number is even, Alice wins.

Alice should store all even numbers in one queue and all odd numbers in the other queue to maximize her chances.

Optimal distribution:

Queue $Q \rightarrow$ All 50 even numbers

Queue $R \rightarrow$ All 50 odd numbers

Since Bob picks a queue randomly (50% probability for each queue), Alice wins whenever Q is selected.

Probability of Bob picking Q (Even Queue) \rightarrow 50% (Alice wins)

Probability of Bob picking R (Odd Queue) \rightarrow 50% (Bob wins)

Thus, Alice's best possible winning probability is 50%.

Input

A list of 50 even numbers and 50 odd numbers.

A randomly chosen queue (Q or R).

A random number k (number of times round-robin scheduler runs).

Output

The last processed number.

Winner (Alice or Bob).

Sample Input

Even Numbers in Q : [2, 4, 6, ..., 100]

Odd Numbers in R : [1, 3, 5, ..., 99]

Selected Queue: Q

Number of Round-Robin Steps: 73

Sample Output

Last Processed Number: 88

Winner: Alice

DSCP 56**Crossing a bridge**

Suppose Bob has four cows that he wants to take across a bridge, but only one yoke, which can hold up to two cows, side by side, tied to the yoke. The yoke is too heavy for him to carry across the bridge, but he can tie (and untie) cows to it in no time at all. Of his four cows, Mazie can cross the bridge in t_1 minutes, Daisy can cross it in t_2 minutes, Crazy can cross it in t_3 minutes, and Lazy can cross it in t_4 minutes. Of course, when two cows are tied to the yoke, they must go at the speed of the slower cow. Describe how Bob can get all his cows across the bridge in minimum minutes.

The bridge can hold a maximum of two people at a time.

A flashlight is required for crossing, and at least one person must return with the flashlight after each crossing.

Each person has a unique crossing time, e.g., [1, 2, 5, 10] minutes.

The objective is to minimize the total crossing time.

Input

Number of cows

Mazie returns in minutes

Daisy returns in minutes

Crazy returns in minutes

Lazy returns in minutes

Output

The total time is to get all cows across the bridge in minutes.

Sample Input

4

Mazie 2

Daisy 4

Crazy 10

Lazy 20

Sample Output

34

DSCP 57**Card Hand**

A Card Hand that supports a person arranging a group of cards in his or her hand. The simulator should represent the sequence of cards using a single positional list ADT so that cards of the same suit are kept together. Implement this strategy by means of four "fingers" into the hand, one for each of the suits of hearts, clubs, spades, and diamonds, so that adding a new card to the person's hand or playing a correct card from the hand can be done in constant time.

The challenge is to design a data structure that:

Maintains cards grouped by suit (Hearts, Clubs, Spades, Diamonds).

Supports efficient insertion of a new card into the hand.

Allows a player to play a card in constant time ($O(1)$).

Uses a single positional list ADT to store the cards while keeping the suits separate.

Uses four "fingers" (pointers) to track the start of each suit's section within the list for fast access.

Input

Add a Card to Hand

Remove (Play) a Card from Hand

Display Hand

Output

Displays the current hand, grouped by suits.

Sample Input

2, "hearts"

5, "hearts"

3, "clubs"

8, "spades"

1, "diamonds"

4, "clubs"

Sample Output

Current Hand:

(2, 'hearts') (5, 'hearts') (3, 'clubs') (4, 'clubs')
(8, 'spades') (1, 'diamonds')

After Playing (5, 'hearts'):

Current Hand:

(2, 'hearts') (3, 'clubs') (4, 'clubs') (8, 'spades')
(1, 'diamonds')

DSCP 58**Air-Traffic Control Simulation**

An airport is developing a computer simulation of air-traffic control that handles events such as landings and takeoffs. Each event has a time stamp that denotes the time when the event will occur. The simulation program needs to efficiently perform the following two fundamental operations:

- Insert an event with a given time stamp (that is, add a future event).
- Extract the event with smallest time stamp (that is, determine the next event to process).

The time stamp is a unique integer representing the event's occurrence time.

Events occur in real-time order, so earliest events must be processed first.

The number of events may be large, requiring an efficient solution.

The system should be able to handle dynamic event scheduling (i.e., inserting new events while processing others).

Input

Insert a new event

Extract the next event to process

Display all scheduled events

Output

Prints all scheduled events in sorted order.

Sample Input

10, "Flight 101 Landing"

5, "Flight 205 Takeoff"

15, "Flight 301 Landing"

2, "Emergency Landing"

Sample Output

Scheduled Events: [(2, 'Emergency Landing'), (5, 'Flight 205 Takeoff'), (15, 'Flight 301 Landing'), (10, 'Flight 101 Landing')]

Next Event: (2, 'Emergency Landing')

Next Event: (5, 'Flight 205 Takeoff')

DSCP 59**Top $\log(n)$ Frequent Flyers**

Tamarindo Airlines wants to give a first-class upgrade coupon to their top $\log n$ frequent flyers, based on the number of miles accumulated, where n is the total number of the airlines' frequent flyers. The algorithm they currently use, which runs in $O(n \log n)$ time, sorts the flyers by the number of miles flown and then scans the sorted list to pick the top $\log n$ flyers.

There are n frequent flyers, each with a unique mileage count.

The system must return the top $\log(n)$ flyers with the highest miles.

The solution should not use full sorting (as it's $O(n \log n)$).

n is large, so an efficient selection method is necessary.

Find the system that identifies the top $\log n$ flyers in $O(n)$ time.

Input

Total number of frequent flyers (n).

Give flyer id and number of miles flown.

Output

List the top $\log(n)$ flyers in descending order.

Sample Input

8
(101, 50000)
(102, 120000)
(103, 75000)
(104, 30000)
(105, 90000)
(106, 60000)
(107, 200000)
(108, 110000)

Sample Output

Top Frequent Flyers: [(107, 200000), (102, 120000), (108, 110000)]

DSCP 60**Stock Trading System**

An online computer system for trading stocks needs to process orders of the form "buy 100 shares at \$x each" or "sell 100 shares at \$y each." A buy order for \$x can only be processed if there is an existing sell order with price \$y such that $y \leq x$. Likewise, a sell order for \$y can only be processed if there is an existing buy order with price \$x such that $y \leq x$. If a buy or sell order is entered but cannot be processed, it must wait for a future order that allows it to be processed. Insert new buy/sell orders in $O(\log n)$ time efficiently independent of whether or not they can be immediately processed.

Orders must be processed in $O(\log n)$ time.

If an order cannot be processed immediately, it remains in the system.

Buy orders prioritize the highest price first (Max Heap).

Sell orders prioritize the lowest price first (Min Heap).

Orders are FIFO when prices are the same.

Input

Number of orders (N)

Each order is in the format:

"BUY shares price"

"SELL shares price"

Output

Trade executions

Pending buy & sell orders

Sample Input

6

BUY 100 50

SELL 50 45

SELL 100 55

BUY 50 55

BUY 30 40

SELL 20 35

Sample Output

Trade executed: Buy 50 shares at \$45 each

Trade executed: Sell 50 shares at \$55 each

Pending Buy Orders: [(50, 50), (40, 30)]

Pending Sell Orders: [(55, 50)]

DSCP 61	Unmonopoly Game
<p>A group of children are playing Unmonopoly, a variation of Monopoly where the richest player gives half their money to the poorest player at each turn. The game continues until a specific stopping condition is met (e.g., a set number of rounds or wealth equality). Each turn, the richest player (max money) gives half their money to the poorest player (min money).</p> <p>The game continues until a stopping condition is met (e.g., a set number of rounds).</p> <p>We need an efficient way to:</p> <p>Find the richest player (max money) in $O(\log n)$ time</p> <p>Find the poorest player (min money) in $O(\log n)$ time</p> <p>Update a player's wealth in $O(\log n)$ time</p>	
<p>Input</p> <p>Number of children (N)</p> <p>List of initial money values</p> <p>Number of rounds (R)</p>	
<p>Output</p> <p>Each round, the richest player gives half their money to the poorest player.</p> <p>Final wealth distribution after R rounds.</p>	
<p>Sample Input</p> <p>5</p> <p>Alice 100</p> <p>Bob 50</p> <p>Charlie 200</p> <p>Daisy 30</p> <p>Eve 150</p> <p>3</p>	<p>Sample Output</p> <p>Round 1: Charlie gives \$100 to Daisy</p> <p>Round 2: Eve gives \$75 to Daisy</p> <p>Round 3: Charlie gives \$50 to Bob</p> <p>Final money distribution:</p> <p>Alice: 100</p> <p>Bob: 100</p> <p>Charlie: 50</p> <p>Daisy: 205</p> <p>Eve: 75</p>

DSCP 62**CPU Job Scheduler**

CPU scheduling in OS is a method by which one process is allowed to use the CPU while the other processes are kept on hold or are kept in the waiting state. This hold or waiting state is implemented due to the unavailability of any of the system resources like I/O etc. Each job is assigned a priority, which is an integer between -20 (highest priority) and 19 (lowest priority), inclusive.

From among all jobs waiting to be processed in a time slice, the CPU must work on a job with highest priority. In this simulation, each job will also come with a length value, which is an integer between 1 and 100, inclusive, indicating the number of time slices that are needed to process this job. For simplicity, you may assume jobs cannot be interrupted—once it is scheduled on the CPU, a job runs for a number of time slices equal to its length.

Output the name of the job running on the CPU in each time slice and must process a sequence of commands, one per time slice, each of which is of the form "add job name with length n and priority p" or "no new job this slice". Design a CPU scheduler that schedules simulated CPU jobs.

Input

Number of time slices (T)

Commands for each time slice:

"add job name with length n and priority p"

"no new job this slice"

Output

Job running in each time slice

Sample Input

10
add job A with length 3 and priority -5
add job B with length 2 and priority -10
no new job this slice
add job C with length 1 and priority 0
no new job this slice
no new job this slice
add job D with length 4 and priority -20
no new job this slice
no new job this slice
no new job this slice

Sample Output

Time slice 1: Running job B
Time slice 2: Running job B
Time slice 3: Running job A
Time slice 4: Running job A
Time slice 5: Running job A
Time slice 6: Running job C
Time slice 7: Running job D
Time slice 8: Running job D
Time slice 9: Running job D
Time slice 10: Running job D

DSCP 63**Counting the Number of Occurrences of Words in a Document**

Given a large text document, we need to efficiently count the occurrences of each word. The program should handle large-scale text files efficiently and output a list of words along with their respective counts. We need to count the occurrences of each word in a document efficiently.

The input is a large text document.

The output is a list of words and their respective counts.

The solution must be efficient, handling large-scale text files.

Common words like "the," "is," etc., should be counted separately.

Read the file line by line to avoid memory overflow.

Tokenize words, convert them to lowercase, and remove punctuation.

Use a hash map to store word frequencies.

Output words with their respective counts in $O(n \log n)$ if sorted.

Input

A text document

Output

Each word and its count

Sample Input

The quick brown fox jumps over the lazy dog.
The dog was not amused.

Sample Output

The: 2
quick: 1
brown: 1
fox: 1
jumps: 1
over: 1
the: 1
lazy: 1
dog: 2
was: 1
not: 1
amused: 1

There are several Web sites on the Internet that allow users to perform queries on flight databases to find flights between various cities, typically with the intent to buy a ticket. To make a query, a user specifies origin and destination cities, a departure date, and a departure time. Efficient search is required for fast response times.

search for flights in a database based on:

Origin City

Destination City

Departure Date

Departure Time

Efficient search is required to ensure fast response times.

The system should handle a large volume of flight data efficiently.

Flights should be sorted by departure time for easy access.

Users may enter a flexible time range, requiring fast retrieval of the nearest available flights.

Input

Flight database (list of flights)

Query (origin, destination, date, time)

Output

Matching Flight(s) Details

Sample Input

FLIGHTS DATABASE:

Flight("DELHI", "HYD", "2025-06-15", "08:00 AM", "Air India", 30000)

Flight("DELHI", "HYD", "2025-06-15", "02:00 PM", "SpiceJet", 40000)

Flight("DELHI", "MUMBAI", "2025-06-15", "09:00 AM", "Thai", 50000)

QUERY:

Find flights from "DELHI" to "HYD" on "2025-06-15" at "08:00 AM"

Sample Output

Flight Found:

Origin: DELHI

Destination: HYD

Date: 2025-06-15

Time: 08:00 AM

Airline: Air India

Price: Rs. 30,000/-

DSCP 65	Spell Checker						
<p>A spell checker takes a given word and checks if it is correctly spelled based on a dictionary of valid words. If the word is misspelled, the system should suggest the closest correct words to help the user correct their input.</p> <p>The spell checker should support efficient lookup ($O(1)$ or $O(\log n)$) for checking valid words.</p> <p>Suggestions should be generated based on the smallest edit distance (Levenshtein Distance) or similar metric.</p> <p>The system should handle large dictionaries efficiently.</p> <p>The suggestion mechanism should support insertions, deletions, substitutions, and transpositions of letters.</p> <p>Operations Required:</p> <p>Check if a word is valid (Fast lookup in the dictionary)</p> <p>Suggest similar words (Finding words that are close in spelling)</p> <p>Efficiency Considerations:</p> <p>Fast lookup for checking validity ($O(1)$ or $O(\log n)$)</p> <p>Efficient suggestion mechanism (Levenshtein distance or Trie-based autocomplete)</p>							
<p>Input</p> <p>Dictionary of valid words (preloaded from a dataset)</p> <p>User-input word to check spelling</p>							
<p>Output</p> <p>If the word is correct → "Correct spelling"</p> <p>If incorrect → "Did you mean: [suggested words]?"</p>							
<table> <tr> <th data-bbox="220 1283 392 1317">Sample Input</th><th data-bbox="810 1283 1002 1317">Sample Output</th></tr> <tr> <td data-bbox="220 1346 783 1379">{ "apple", "banana", "grape", "orange", "mango" }</td><td data-bbox="810 1346 1018 1379">Incorrect spelling.</td></tr> <tr> <td data-bbox="220 1395 443 1429">User Input: "appel"</td><td data-bbox="810 1395 1094 1429">Did you mean: ['apple']?</td></tr> </table>		Sample Input	Sample Output	{ "apple", "banana", "grape", "orange", "mango" }	Incorrect spelling.	User Input: "appel"	Did you mean: ['apple']?
Sample Input	Sample Output						
{ "apple", "banana", "grape", "orange", "mango" }	Incorrect spelling.						
User Input: "appel"	Did you mean: ['apple']?						

DSCP 66	Maximum Non-Overlapping Daily Jobs Scheduling
<p>Consider the following variation on the Interval Scheduling Problem from lecture. You have a processor that can operate 24 hours a day, every day. People submit requests to run daily jobs on the processor. Each such job comes with a start time and an end time; if the job is accepted to run on the processor, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the Interval Scheduling Problem.) Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n, the number of jobs. You may assume for simplicity that no two jobs have the same start or end times</p>	
<p>Input</p> <p>The first line contains an integer n ($1 \leq n \leq 100,000$), representing the number of jobs.</p> <p>The next n lines contain two integers s_i, e_i ($0 \leq s_i, e_i < 24$), representing the start and end times of the i^{th} job in 24-hour format.</p> <p>$s_i \neq e_i$, meaning no job has zero length.</p>	
<p>Output</p> <p>A single integer, representing the maximum number of non-overlapping jobs that can be scheduled.</p>	
Sample Input	Sample Output
5	3
1 5	
2 6	
23 4	
6 8	
7 10	

DSCP 67**Building Bridge**

There are eight small islands in a lake, and the state wants to build even bridges to connect them so that each island can be reached from another one via one or more bridges. The cost of constructing a bridge is proportional to its length. The goal is to connect all the islands with 7 bridges such that the total cost of constructing the bridges is minimized.

Graph Representation: The islands and bridges form a weighted undirected graph.

Connected Graph: All islands must be directly or indirectly connected.

Cost Minimization: The minimum spanning tree (MST) approach should be used.

Efficient Algorithm: The solution should run in $O(E \log V)$ time.

Input

The input is a distance matrix representing the distances between pairs of islands.

Output

The output is a list of the 7 bridges (edges) to be constructed, represented by the pairs of islands connected and their respective costs. These bridges form the MST with the minimum construction cost.

Sample Input

	1	2	3	4	5	6	7	8
1	-	240	210	340	280	200	345	120
2	-	-	265	175	215	180	185	155
3	-	-	-	260	115	350	435	195
4	-	-	-	-	160	330	295	230
5	-	-	-	-	-	360	400	170
6	-	-	-	-	-	-	175	205
7	-	-	-	-	-	-	-	305
8	-	-	-	-	-	-	-	-

Sample Output

Bridges to build:

Bridge between island 1 and island 8 with cost 120

Bridge between island 2 and island 4 with cost 175

Bridge between island 3 and island 5 with cost 115

Bridge between island 4 and island 6 with cost 160

Bridge between island 2 and island 6 with cost 180

Bridge between island 7 and island 8 with cost 305

Bridge between island 5 and island 7 with cost 170

Total construction cost: 1025

DSCP 68**Video-phone system**

A company named RT&T has a network of n switching stations connected by m high-speed communication links. Each customer's phone is directly connected to one station in his or her area. The engineers of RT&T have developed a prototype video-phone system that allows two customers to see each other during a phone call. In order to have acceptable image quality, however, the number of links used to transmit video signals between the two parties cannot exceed 4.

Suppose that RT&T's network is represented by a graph. For each station, compute the set of stations that it can reach using no more than 4 links. In other words, determine the stations that can be accessed from a given station through a path of length at most 4 edges in the network graph. The network graph is undirected, where each station is represented as a vertex, and each high-speed communication link is represented as an undirected edge between two vertices. The goal is to compute, for each station, the set of stations it can reach using no more than 4 links.

The graph is undirected, and no parallel edges exist (there is at most one edge between any pair of stations).

The number of stations n can be as large as 10,000, and the number of communication links m can be as large as 50,000.

The solution must efficiently compute the set of reachable stations for each station with a time complexity suitable for the problem size.

Input

The graph is represented as an undirected graph with n stations (nodes) and m communication links (edges).

For each station, we need to determine all other stations that can be reached using at most 4 edges (links).

Output

For each station, output the set of stations that can be reached using no more than 4 links.

Sample Input

5 6

1 2

1 3

2 3

2 4

3 5

4 5

Sample Output

For station 1: {2, 3, 4, 5}

For station 2: {1, 3, 4, 5}

For station 3: {1, 2, 4, 5}

For station 4: {2, 3, 5}

For station 5: {3, 4}

DSCP 69**Telephone Network**

RT&T is a telecommunications company that wants to compute the maximum possible time delay that may be experienced in a long-distance call. The delay depends on the number of communication links between the caller and callee, and it is represented as the length of the longest path in the telephone network.

The telephone network of RT&T is represented as a tree. A tree is a connected, acyclic graph where any two nodes are connected by exactly one path. The diameter of a tree is the length of the longest path between any two nodes in the tree.

The engineers of RT&T need to compute the diameter of the tree efficiently to estimate the maximum time delay for long-distance calls.

Given a tree T , compute the diameter of the tree. The diameter is defined as the length of the longest path between any two nodes in the tree.

The graph is a tree, meaning it is a connected acyclic graph with $n-1$ edges.

The number of nodes n can be as large as 10,000, and the number of edges is $n-1$.

The solution should be efficient, with a time complexity suitable for large values of n (approximately $O(n)$).

Input

The tree is represented by n nodes and $n-1$ edges.

Each edge connects two nodes and has a weight that indicates the communication delay for that link.

We need to compute the diameter of the tree, which is the longest path between any two nodes in the tree.

Output

The output should be the length of the longest path between any two nodes in the tree, which is the diameter of the tree.

Sample Input

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

Sample Output

The diameter of the tree is: 8

Tamarindo University, along with many other schools worldwide, is working on a joint multimedia project and has built a computer network connecting various schools. The network is structured as a tree, where each school is represented by a node, and each communication link is represented by an edge between nodes.

The schools have decided to install a file server at one of the schools to share data among all the connected schools. The cost of a data transfer is proportional to the number of communication links used for transmission, and the goal is to minimize this cost by selecting a "central" location for the file server. The transmission time is dominated by the link setup and synchronization, so choosing an optimal central node can significantly reduce the transmission time.

The eccentricity of a node v in a tree T is defined as the length of the longest path from v to any other node in T . A node with the minimum eccentricity is called a center of the tree.

Given a tree T , identify the center node(s) of the tree. The file server should be installed at the center of the tree to minimize the maximum transmission time.

The graph is a tree, meaning it is connected and acyclic, with $n-1$ edges.

The number of nodes n can be as large as 10,000, and the number of edges is $n-1$.

The solution must be efficient, with a time complexity suitable for large values of n (approximately $O(n)$).

Input

An integer n (number of nodes in the tree).

A list of $n-1$ edges where each edge is represented as a tuple (u, v, w) , where u and v are the nodes connected by the edge and w is the weight (length) of the edge.

Output

The center(s) of the tree.

If the center is unique, print the center node.

If there are two centers, print both nodes.

Sample Input

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

Sample Output

```
Center of the tree: 1
```

DSCP 71**Maximum Bottleneck Path Problem**

Consider a telephone network represented as an undirected graph G , where:

The vertices of the graph represent switching centers, denoted as nodes a and b .

The edges represent communication lines between pairs of switching centers, and each edge is marked with a bandwidth value, which represents the maximum transmission capacity of that link.

The bandwidth of a path is defined as the smallest bandwidth value on that path. Hence, the bandwidth of a path from node a to node b is determined by the edge with the minimum bandwidth along the path.

The objective is to find the maximum bandwidth path between two given nodes a and b . This path is the one that maximizes the minimum bandwidth along its edges.

The graph is undirected, meaning that communication between two nodes is bidirectional.

The number of nodes n can be as large as 10,000, and the number of edges m can be as large as 50,000.

The bandwidth of each edge is a positive integer and is at most 1,000,000.

Input

The graph is represented as an adjacency list

Output

Get the max bandwidth path between the two centres a and b .

Sample Input

graph =
0: [(1, 50), (2, 30)],
1: [(0, 50), (2, 40), (3, 60)],
2: [(0, 30), (1, 40), (3, 20)],
3: [(1, 60), (2, 20)]
 $a = 0$
 $b = 3$

Sample Output

40

DSCP 72**NASA's Communication Network**

NASA wants to link n stations spread over the country using communication channels. Each pair of stations has a different bandwidth available, and the bandwidth between any two stations is known beforehand. The goal is to select $n-1$ channels (the minimum possible number of channels to connect all stations) such that:

All stations are connected by the channels.

The total bandwidth (defined as the sum of the individual bandwidths of the selected channels) is maximized.

In other words, NASA needs to construct a Maximum Spanning Tree (MST) using $n-1$ edges (the minimum number of edges to connect all the nodes), where the weight of each edge represents the bandwidth available between two stations, and the tree maximizes the total sum of the selected bandwidths.

The graph is undirected, meaning that each edge represents a bidirectional communication channel.

The number of stations n can be as large as 10,000, and the number of edges m can be as large as 50,000.

The bandwidth w is a positive integer, and the maximum possible bandwidth value is 1,000,000.

Input

n : Number of stations.

bandwidth, station1, station2.

Output

Maximum Total Bandwidth

Maximum Spanning Tree edges

Sample Input

$n = 4$ # Number of stations

edges = [

(10, 0, 1), # Channel with bandwidth 10
between stations 0 and 1

(20, 1, 2),

(30, 2, 3),

(25, 0, 3),

(15, 0, 2),

(5, 1, 3)

]

Sample Output

Maximum Total Bandwidth: 65

Edges in Maximum Spanning Tree: [(2, 3, 30),
(0, 3, 25), (1, 2, 20)]

DSCP 73**Sir Paul's Treasure Hunt**

Inside the Castle of Asymptopia there is a maze, and along each corridor of the maze there is a bag of gold coins. The amount of gold in each bag varies. A noble knight, named Sir Paul, will be given the opportunity to walk through the maze, picking up bags of gold. He may enter the maze only through a door marked "ENTER" and exit through another door marked "EXIT." While in the maze he may not retrace his steps. Each corridor of the maze has an arrow painted on the wall. Sir Paul may only go down the corridor in the direction of the arrow. There is no way to traverse a "loop" in the maze. Given a map of the maze, including the amount of gold in each corridor to help Sir Paul pick up the most gold.

The graph contains n nodes ($2 \leq n \leq 10,000$) and m edges ($1 \leq m \leq 50,000$).

The amount of gold in each corridor is a positive integer ($1 \leq g \leq 1,000$).

The graph is acyclic, meaning there are no loops in the maze.

The graph is directed, meaning corridors must be traversed in the direction of the arrows.

Input

n : Number of rooms.

edges: start_room, end_room, gold_amount representing corridors.

enter: Index of the "ENTER" room.

exit: Index of the "EXIT" room.

Output

The path with the maximum sum of gold.

Sample Input

$n = 6$

edges = [

(0, 1, 10), # Room 0 → Room 1 with 10 gold

(0, 2, 5),

(1, 3, 20),

(2, 3, 15),

(2, 4, 10),

(3, 5, 30),

(4, 5, 25)

]

enter = 0

exit = 5

Sample Output

Maximum Gold Collected: 60

Path: [0, 1, 3, 5]

Suppose you are given a timetable, which consists of:

- A set A of n airports, and for each airport a in A , a minimum connecting time $c(a)$.
- A set F of m flights, and the following, for each flight f in F :
 - Origin airport $a_1(f)$ in A ◦ Destination airport $a_2(f)$ in A
 - Departure time $t_1(f)$
 - Arrival time $t_2(f)$

Find the earliest arrival time at airport b , starting from airport a at or after time t . In this problem, we are given airports a and b , and a time t , and we wish to compute a sequence of flights that allows one to arrive at the earliest possible time in b when departing from a at or after time t . Minimum connecting times at intermediate airports must be observed. What is the running time of your algorithm as a function of n and m ?

Input

n = Number of airports.

flights = List of (origin, destination, departure time, arrival time) tuples.

airports = Dictionary of minimum connecting times at each airport.

start_airport = Starting airport index.

end_airport = Destination airport index.

start_time = Earliest departure time.

Output

Earliest arrival time

Time-dependent shortest path

Sample Input

$n = 5$ # Number of airports

flights = (0, 1, 8, 10), # Flight from 0 to 1,
departs at 8, arrives at 10

(1, 2, 12, 14),

(0, 3, 9, 12),

(3, 2, 13, 16),

(2, 4, 17, 20),

(3, 4, 14, 19)

airports = {0: 0, 1: 1, 2: 2, 3: 1, 4: 0} # Minimum
connection times

start_airport = 0

end_airport = 4

start_time = 8

Sample Output

Earliest Arrival Time: 20

Path Taken: [(0, 3, 9, 12), (3, 4, 14, 19)]

DSCP 75**Karen's algorithm**

Karen has developed a new method for path compression in a tree-based union/find partition data structure, starting at a position p . The union/find structure consists of nodes with parent pointers, and each node points to a parent node, with the root of each tree pointing to itself.

Karen's algorithm works as follows:

Start at a position p and traverse the path from p to the root.

Put all positions on this path into a set S .

Scan through the set S , and for each node x in S , set its parent pointer to the parent's parent pointer (i.e., $\text{parent}(x) = \text{parent}(\text{parent}(x))$). The parent pointer of the root points to itself.

If any parent pointer changes during this pass, repeat the process. Continue repeating until a scan through S does not change any parent pointers.

The goal is to prove that Karen's path compression algorithm is correct and analyze its running time for a path of length h .

The tree has a depth of h ($2 \leq h \leq 10,000$).

Each node has a parent pointer that points to another node, with the root of each tree pointing to itself.

Input

A tree-based union/find data structure with a set of nodes, where each node has a parent pointer.

A position p in the tree, from which the path compression process starts.

Output

Finds the root of x using Karen's path compression.

Sample Input

uf = UnionFind(10)

uf.union(1, 2)

uf.union(2, 3)

uf.union(3, 4)

uf.union(4, 5)

Sample Output

1

[0, 1, 1, 1, 1, 1, 6, 7, 8, 9]

DSCP 76**Balanced Load Distribution in Server Cluster**

A company wants to efficiently distribute network traffic across multiple servers. Each server is represented as a node in a Balanced Binary Search Tree (BST), where:

The key represents the server's capacity.

The left subtree contains servers with lesser capacity.

The right subtree contains servers with greater capacity.

Given an incoming request with a certain processing demand d , design an efficient algorithm to:

1. Find the server with the closest higher capacity (i.e., the smallest node whose value is $\geq d$).
2. If multiple servers have the same capacity, pick the one with the lowest load.
3. If no server meets the requirement, return "No server available."

The BST is balanced, meaning the height of the tree is $O(\log n)$, where n is the number of nodes.

The number of nodes n can be as large as 10,000.

The processing demand d is a positive integer.

Input

Integer N → Number of servers in the BST.

N lines → Each line contains two integers:

- Server Capacity
- Current Load on Server

Integer Q → Number of incoming requests.

Q lines → Each line contains a single integer (Request Demand d).

Output

For each request, print the server capacity and load of the best available server. If no such server exists, print "No server available."

Sample Input

5
10 3
20 5
15 2
25 1
30 4
3
12
18
35

Sample Output

15 2
20 5
No server available.

DSCP 77	Secure File Storage
<p>A cloud storage provider wants to ensure file integrity using Binary Hash Trees (Merkle Trees). Given a binary tree representation of file blocks, where each leaf node contains a hash of a file block, and each internal node stores the hash of its children. A binary hash tree representing the file, with n blocks of the file. The tree has Leaf nodes storing hashes of file blocks. Internal nodes storing the hash of their children. A file block index and a proof (a set of hashes from the root to the block) for verification purposes. Implement an algorithm to:</p> <ol style="list-style-type: none"> 1. Verify file integrity efficiently (i.e., detect corruption with minimal computation). 2. Optimize retrieval by ensuring partial downloads can be verified efficiently. <p>The number of file blocks n can be as large as 1,000,000. The tree is balanced, and each internal node stores a hash of its two children. The hash function is cryptographically secure (e.g., SHA-256).</p>	
<p>Input</p> <p>Integer $N \rightarrow$ Number of file blocks.</p> <p>N lines \rightarrow Each line contains a file block (string).</p> <p>Integer $Q \rightarrow$ Number of integrity verification queries.</p> <p>Q lines \rightarrow Each line contains an index of a file block that needs to be verified.</p>	
<p>Output</p> <p>For each query:</p> <ul style="list-style-type: none"> • Print "Verified" if the hash of the given block matches the stored hash. • Print "Corrupted" if the hash does not match. 	
<p>Sample Input</p> <p>4</p> <p>block1_data</p> <p>block2_data</p> <p>block3_data</p> <p>block4_data</p> <p>2</p> <p>1</p> <p>3</p>	<p>Sample Output</p> <p>Verified</p> <p>Verified</p>

DSCP 78**Fraud Detection in Banking**

A banking system needs to identify duplicate or suspicious transactions in real-time. Each transaction has:

Transaction ID: A unique identifier for each transaction.

Timestamp: The time at which the transaction occurred.

Amount: The value of the transaction.

Sender ID: The unique identifier of the sender.

Receiver ID: The unique identifier of the receiver.

The goal is to detect if a similar transaction (same sender, receiver, and amount) occurs within 10 minutes of a previous one. If such a transaction is found, it is flagged as fraudulent.

The number of transactions can be large (up to millions).

Timestamps are given in a format that can be compared directly.

The system needs to handle real-time processing with low latency.

Input

Integer N → Number of transactions.

N lines → Each transaction consists of:

- Transaction ID (String)
- Timestamp (Integer, in minutes)
- Amount (Integer)
- Sender ID (String)
- Receiver ID (String)

Integer Q → Number of fraud detection queries.

Q lines → Each query consists of a Transaction ID to check.

Output

For each query:

- Print "Fraud Detected" if a duplicate transaction was found within 10 minutes.
- Print "No Fraud" otherwise.

Sample Input

5
T1 100 500 A B
T2 105 500 A B
T3 200 1000 X Y
T4 205 1000 X Y
T5 250 700 A C
3
T2
T3
T5

Sample Output

Fraud Detected
Fraud Detected
No Fraud

DSCP 79	URL Shortener
<p>A URL (Uniform Resource Locator) is the address used to access resources on the internet. It specifies the location of a resource and how to retrieve it. A URL typically consists of several components:</p> <p>Scheme: The protocol used to access the resource (e.g., http, https, ftp).</p> <p>Host: The domain name or IP address of the server hosting the resource (e.g., www.example.com).</p> <p>Path: The specific location of the resource on the server (e.g., /images/pic.jpg).</p> <p>Query (optional): Additional parameters used for requests (e.g., ?search=term).</p> <p>Fragment (optional): A specific section within the resource (e.g., #section1).</p> <p>A URL shortening service is required to convert long URLs into short, unique aliases. The system should:</p> <ol style="list-style-type: none"> 1. Generate a short URL using a hashing technique. 2. Store and retrieve URLs efficiently using a hash table. 3. Handle collisions efficiently. 4. Support high read/write throughput for millions of URLs. <p>The system should support millions of URLs.</p> <p>Short URLs must be unique and short (e.g., 6-8 characters).</p> <p>The system must handle hash collisions when two different URLs generate the same short URL.</p> <p>The system must support high read/write throughput to handle large traffic volumes.</p>	
<p>Input</p> <p>Integer N → Number of URLs to be shortened.</p> <p>N lines → Each line contains a long URL (string).</p> <p>Integer Q → Number of queries.</p> <p>Q lines → Each line contains a short URL to be resolved.</p>	
<p>Output</p> <p>For each query:</p> <ul style="list-style-type: none"> • Print the original long URL if it exists. • Print "URL not found" if the short URL does not exist. 	
<p>Sample Input</p> <p>3</p> <p>https://example.com/article/12345</p> <p>https://myblog.com/post/how-to-code</p> <p>https://news.com/story/abcdef</p> <p>2</p> <p>abc123</p> <p>xyz999</p>	<p>Sample Output</p> <p>https://example.com/article/12345</p> <p>URL not found</p>

DSCP 80**Anagram Detection**

Given two words, the goal is to determine if one is an anagram of the other. An anagram is formed by rearranging the letters of one word to produce another word, using exactly the same letters with the same frequency.

The system should efficiently handle large-scale text datasets, determining if two words are anagrams in an optimized manner.

The words contain only alphabetic characters (no spaces or punctuation).

Words can be of different lengths, but both are guaranteed to be non-empty.

The system should be optimized to handle large datasets with potentially millions of word comparisons.

Sorting Approach: Sort both words and compare if their sorted versions are identical. Time complexity: $O(n \log n)$.

Counting Approach: Count the frequency of each character in both words. If the frequencies match, the words are anagrams. Time complexity: $O(n)$, where n is the length of the words.

Input

Integer $N \rightarrow$ Number of word pairs to check.

N lines \rightarrow Each line contains two words (strings).

Output

For each pair:

- Print "Anagram" if the two words are anagrams.
- Print "Not Anagram" otherwise.

Sample Input

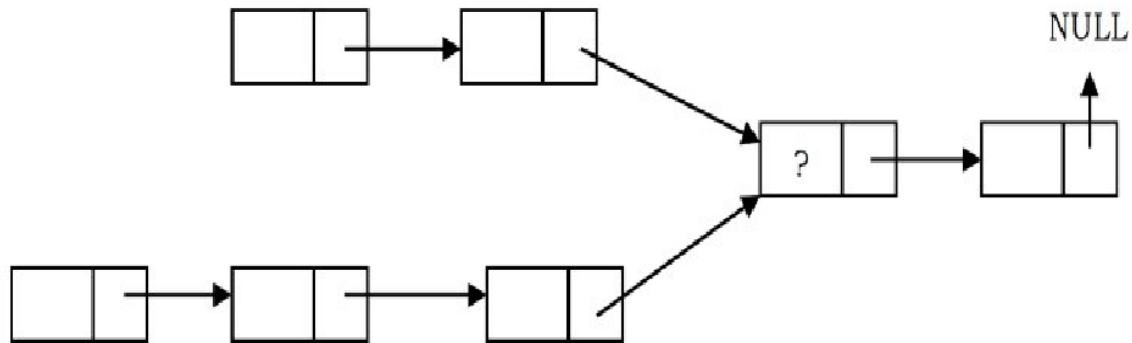
3
listen silent
hello world
binary brainy

Sample Output

Anagram
Not Anagram
Anagram

DSCP 81	Length of connected cells
<p>Given a binary matrix where each cell contains either a 1 or a 0, the task is to find the largest region formed by connected 1s. Two cells containing 1 are considered connected if they are adjacent either horizontally, vertically, or diagonally.</p> <p>A region consists of adjacent cells containing 1. The goal is to identify the largest region (the region with the maximum number of connected 1s) in the matrix.</p> <p>The matrix can be of any size, up to 1000 x 1000.</p> <p>Cells with a value of 0 do not belong to any region.</p> <p>Depth First Search (DFS): Traverse the matrix to find all the connected regions. For each unvisited 1, run a DFS to explore all adjacent cells that are part of the same region. Keep track of the size of each region and update the maximum size found.</p> <p>Breadth First Search (BFS): Alternatively, use BFS starting from each unvisited 1 to explore the entire region and count its size.</p> <p>Mark each visited cell to avoid counting the same region multiple times.</p> <p>Compare the size of each region and return the largest one.</p>	
<p>Input</p> <p>A binary matrix of size $n \times m$, where $1 \leq n, m \leq 100$.</p> <p>Each row contains m characters (0 or 1).</p> <p>The input ends when the matrix is fully given.</p>	
<p>Output</p> <p>A single integer representing the size of the largest connected region of 1s.</p>	
<p>Sample Input</p> <pre>11000 01100 00101 10001 01011</pre>	<p>Sample Output</p> <pre>5</pre>

Suppose there are two singly linked lists both of which intersect at some point and become a single linked list. The head or start pointers of both the lists are known, but the intersecting node is not known. Also, the number of nodes in each of the lists before they intersect is unknown and may be different in each list. List1 may have n nodes before it reaches the intersection point, and List2 might have m nodes before it reaches the intersection point where m and n may be $m = n$, $m < n$ or $m > n$. Give an algorithm for finding the merging point.



Input

You are given two singly linked lists that merge at some point and continue as a single linked list. The following details apply:

1. The structure of a linked list:
 - Each list consists of nodes where each node contains:
 - A value (data).
 - A pointer to the next node in the list.
2. Head pointers:
 - You are given two head pointers: head1 and head2.
 - head1 points to the first node of List1.
 - head2 points to the first node of List2.
3. Before the merge point:
 - The lists have different numbers of nodes before they merge.
 - List1 may have n nodes before merging, and List2 may have m nodes before merging.
 - The values in these nodes are arbitrary.
4. After the merge point:
 - The lists become a single linked list.
 - All nodes after the merge point are shared by both lists.

Two singly linked lists that merge at a certain node.

The head pointers of both lists are given.

Output

If an intersection exists:

Print the first merging node's value (i.e., the first node that is common between both lists).

If there is no intersection:

If the two lists never merge, print: No intersection

Sample Input	Sample Output
List1: 1 → 2 → 3 → 4 → 5 → 6 → 7	5
List2: 9 → 10 → 5 → 6 → 7	

DSCP 83	Reverse the Linked List in Pairs
<p>You are given a singly linked list where each node contains an integer value. Your task is to swap adjacent nodes in pairs while keeping the overall structure of the list intact. If there is an odd number of nodes, the last node should remain unchanged. If you have a linked list that holds $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow X$, then after the function has been called the linked list would hold $2 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow X$.</p> <p>Swapping Adjacent Nodes:</p> <p>Every two consecutive nodes should be swapped.</p> <p>The first node in the pair moves to the second position, and the second node moves to the first position.</p> <p>Handling Odd and Even Length Lists:</p> <p>If the number of nodes is even, all nodes are swapped in pairs.</p> <p>If the number of nodes is odd, the last remaining node is left unchanged.</p> <p>The Last Node (X) Represents the End of the List:</p> <p>The notation X in the problem represents the null pointer in a singly linked list.</p>	
<p>Input</p> <p>The input list may be empty (X).</p> <p>The list may contain only one node.</p> <p>The number of nodes can be odd or even.</p> <p>The values in the nodes are not necessarily sequential they can be any integers.</p>	
<p>Output</p> <p>The output should be a linked list where adjacent nodes are swapped in pairs.</p> <p>The last node remains unchanged if there is no pair for it.</p> <p>The modified linked list is printed in the same format as the input.</p>	
<p>Sample Input</p> <p>$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow X$</p>	<p>Sample Output</p> <p>$2 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow X$</p>

DSCP 84	Finding the Last Survivor in a Circular Elimination Game
<p>Josephus Circle: N people have decided to elect a leader by arranging themselves in a circle and eliminating every M^{th} person around the circle, closing ranks as each person drops out. Find which person will be the last one remaining (with rank 1).</p> <p>They stand in a circle, numbered 1 to N.</p> <p>Starting from person 1, they count up to M and eliminate the M^{th} person.</p> <p>The circle closes ranks (shifts left), and counting resumes from the next person.</p> <p>This process repeats until only one person remains.</p> <p>$1 \leq N \leq 105$ (Large values of N require an efficient solution.)</p> <p>$1 \leq M \leq N$</p>	
<p>Input</p> <p>$N \rightarrow$ Total number of people standing in the circle.</p> <p>$M \rightarrow$ Every M^{th} person is eliminated.</p>	
<p>Output</p> <p>The program should return the position (rank) of the last remaining person.</p>	
Sample Input	Sample Output
$N = 5, M = 2$	5

DSCP 85**Find modular node**

Given a singly linked list, write a function to find the last element from the beginning whose $n \% k == 0$, where n is the number of elements in the list and k is an integer constant. For example, if $n = 19$ and $k = 3$ then we should return 18th node. We are given a singly linked list and an integer k . Our task is to find the last node in the list whose position (1-based index) is divisible by k .

1. The list is singly linked, meaning each node has a value and a pointer to the next node.
2. The position of each node is counted from 1 (not 0).
3. We need to find the last node in the list where its position n is divisible by k .
4. If no such node exists, we return None.

Input

A singly linked list with n nodes.

An integer k (where $k > 0$).

Output

The data of the last node whose position is divisible by k .

If no such node exists, return None.

Sample Input

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

Sample Output

9

DSCP 86	Reverse Blocks of K Nodes
<p>Given a singly linked list containing n elements and an integer K ($K > 0$), modify the list such that every group of K nodes is reversed. If there are fewer than K nodes remaining at the end, they should not be reversed and should remain in their original order.</p> <p>Example: Input: 1 2 3 4 5 6 7 8 9 10.</p> <p>Output for different K values:</p> <p>For $K = 2$: 2 1 4 3 6 5 8 7 10 9</p> <p>For $K = 3$: 3 2 1 6 5 4 9 8 7 10</p> <p>For $K = 4$: 4 3 2 1 8 7 6 5 9 10</p> <p>$1 \leq K \leq n$ (i.e., K is always a positive integer).</p> <p>The number of nodes in the list can be large (up to 10^6).</p> <p>The solution should be efficient and run in $O(n)$ time.</p>	
<p>Input</p> <p>A single line of space-separated integers representing the linked list.</p> <p>A second line containing a single integer K (the block size for reversal).</p>	
<p>Output</p> <p>A single line of space-separated integers representing the modified linked list after reversing every K nodes.</p>	
<p>Sample Input</p> <p>1 2 3 4 5 6 7 8 9 10</p> <p>2</p>	<p>Sample Output</p> <p>2 1 4 3 6 5 8 7 10 9</p>

DSCP 87**Implementing Three Stacks in One Array**

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. Normally, stacks are implemented using separate arrays or linked lists, but in this problem, we need to implement three stacks using a single array. Given a single array of size N , implement three independent stacks within this array, supporting the following operations:

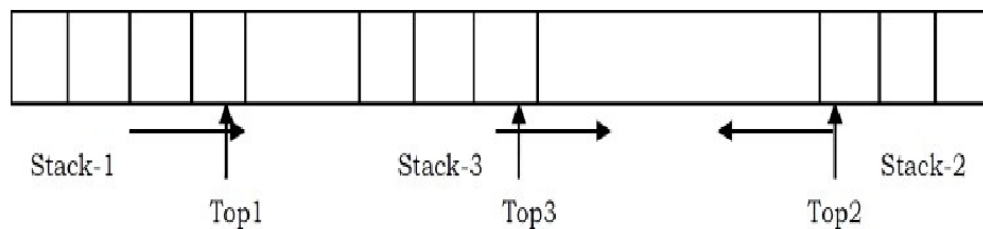
Push(stackNumber, value) → Push an element onto a specific stack (stackNumber can be 1, 2, or 3).

Pop(stackNumber) → Remove the top element from the specified stack.

Peek(stackNumber) → Return the top element of the specified stack without removing it.

isEmpty(stackNumber) → Check if the specified stack is empty.

isFull(stackNumber) → Check if there is no available space to push a new element.

**Input**

A single integer N representing the size of the array.

A series of operations (push, pop, peek, isEmpty, isFull) applied to one of the three stacks.

Output

After each operation, return the appropriate result (Pushed, Popped, Top Element, True/False).

If an invalid operation is performed (e.g., popping from an empty stack), return an error message.

Sample Input

$N = 9$

push(1, 10)

push(1, 20)

push(2, 30)

push(3, 40)

pop(1)

peek(2)

isEmpty(3)

Sample Output

Pushed 10 into Stack 1

Pushed 20 into Stack 1

Pushed 30 into Stack 2

Pushed 40 into Stack 3

Popped 20 from Stack 1

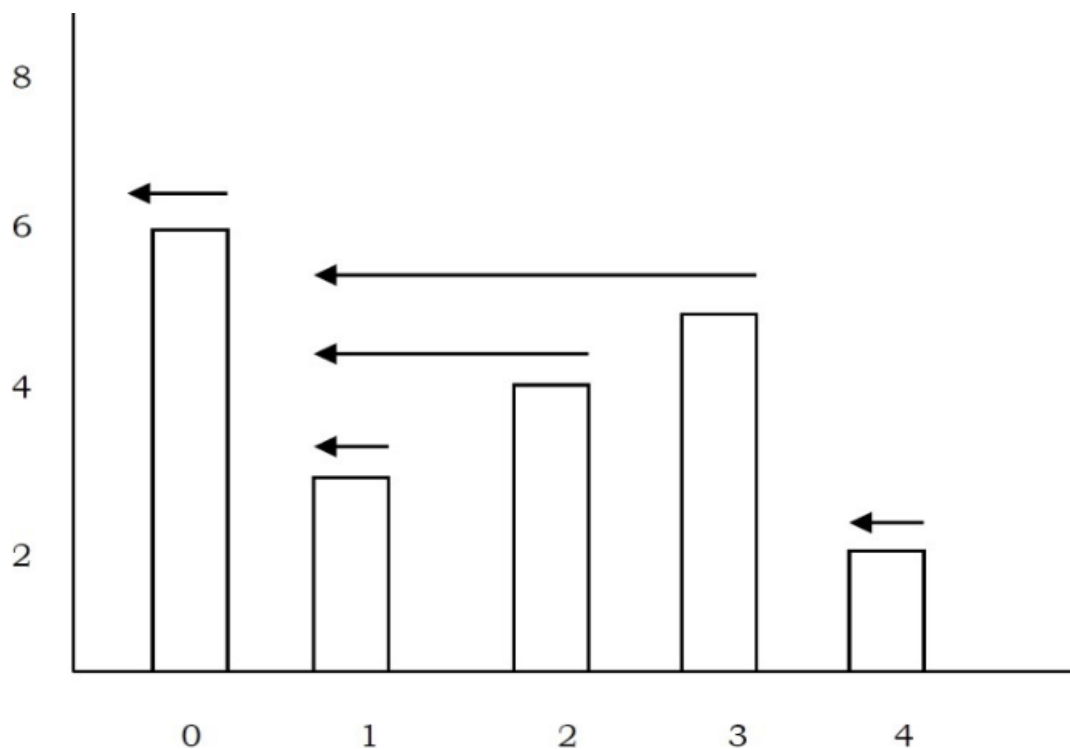
Top Element in Stack 2: 30

False

DSCP 88**Finding the Stock Span**

The stock span problem is a common financial analysis problem used to determine the peaks in stock prices. The span of a stock price on a given day i is defined as the maximum number of consecutive days (including i) for which the price of the stock was less than or equal to its price on day i . Alternatively, given an array A of integers, find the maximum of $j - i$ subjected to the constraint that $A[i] < A[j]$. This problem is widely used in stock market analysis to determine 52-week highs, resistance levels, and stock trends. Given an array A of size N , where $A[i]$ represents the stock price on day i , compute an array S such that:

$S[i]$ is the maximum number of consecutive elements $A[j]$ immediately preceding $A[i]$ and satisfying $A[j] < A[i]$.



Day: Index i	Input Array $A[i]$	$S[i]$: Span of $A[i]$
0	6	1
1	3	1
2	4	2
3	5	3
4	2	1

Input

An integer N representing the number of days.

A sequence of N space-separated integers representing stock prices.

Output

A single line containing N space-separated integers representing the span for each day.

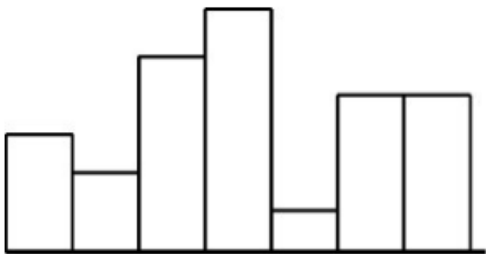
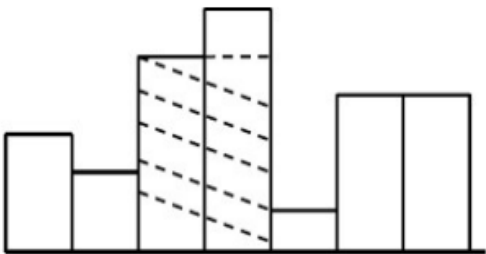
Sample Input

7

100 80 60 70 60 75 85

Sample Output

1 1 1 2 1 4 6

DSCP 89	Largest Rectangle under Histogram
<p>A histogram is a polygon composed of a sequence of rectangles aligned at a common base line. For simplicity, assume that the rectangles have equal widths but may have different heights. For example, the figure on the left shows a histogram that consists of rectangles with the heights 3,2,5,6,1,4,4, measured in units where 1 is the width of the rectangles. Here our problem is: given an array with heights of rectangles (assuming width is 1), we need to find the largest rectangle possible. For the given example, the largest rectangle is the shaded part</p> <div style="display: flex; justify-content: space-around; align-items: flex-end;">   </div> <p>$1 \leq n \leq 10^5$ (where n is the number of bars).</p> <p>$1 \leq \text{heights}[i] \leq 10^6$ (each bar height is a positive integer).</p>	
<p>Input</p> <p>A single integer N representing the number of bars in the histogram.</p> <p>A sequence of N space-separated integers representing the heights of the bars.</p>	
<p>Output</p> <p>A single integer representing the maximum area of a rectangle that can be formed.</p>	
<p>Sample Input</p> <p>7</p> <p>3 2 5 6 1 4 4</p> <p>6</p> <p>2, 1, 5, 6, 2, 3</p>	<p>Sample Output</p> <p>12</p> <p>10</p>

DSCP 90**Snake or Snail**

In linked list-based data structures, it is important to determine whether a linked list terminates properly or contains a cycle. Given a singly linked list L , each node points to the next node in the sequence. The list can have one of two structures:

1. Snake - The linked list ends properly, meaning the last node points to NULL.
2. Snail - The last node links back to an earlier node, forming a cycle.

A cycle in a linked list means that traversing the list indefinitely loops back to a previously visited node. Our goal is to determine whether a given linked list is a snake (terminates properly) or a snail (contains a cycle). Give an algorithm that tests whether a given list L is a snake or a snail

Input

The first line contains an integer T , the number of test cases.

Each test case consists of a single integer N ($1 \leq N \leq 10^6$), representing the number of nodes in the linked list.

The next line contains N space-separated integers, representing the linked list's nodes in order.

The last integer in the input specifies whether the last node points to an earlier node (cycle) or ends at NULL.

If it is -1 , the list is a snake (ends properly).

Otherwise, it is an index i (0-based), indicating that the last node points back to the i -th node, forming a cycle (snail).

Output

"The linked list is a snake (ends properly)." if there is no cycle.

"The linked list is a snail (has a cycle)." if a cycle exists.

Sample Input

```
2
4
1 2 3 4 -1
5
10 20 30 40 50 2
```

Sample Output

```
The linked list is a snake (ends properly).
The linked list is a snail (has a cycle).
```

DSCP 91**Fractional Node in a Singly Linked List**

In a singly linked list, each node contains a value and a pointer to the next node. Given a linked list of size N , we need to find the fractional node, which is the $\lfloor N / k \rfloor$ -th node in the list for a given integer k . Here, $\lfloor x \rfloor$ represents the floor function, which rounds down the value to the nearest integer.

Approach 1: Two Pass Solution

Pass 1: Traverse the list to count N .

Pass 2: Compute $\text{ceil}(N/k)$, traverse again to get that node.

Approach 2: Single Pass Using Two Pointers

Use two pointers:

One pointer moves every k steps

The other moves every step

When the first pointer reaches the end, the second pointer is at $\text{ceil}(N/k)$

Input

The first line contains an integer T ($1 \leq T \leq 10$) representing the number of test cases.

For each test case:

The first line contains two integers N and k ($1 \leq N \leq 10^5$, $1 \leq k \leq 10^5$) representing:

$N \rightarrow$ Number of nodes in the linked list.

$k \rightarrow$ The divisor for fractional node calculation.

The second line contains N space-separated integers representing the elements of the linked list.

Output

For each test case, print the value of the fractional node.

Sample Input

```
2
7 3
1 2 3 4 5 6 7
10 4
10 20 30 40 50 60 70 80 90 100
```

Sample Output

```
2
20
```

DSCP 92**Median in an Infinite Series of Integers**

In a continuous stream of integers, we need to dynamically find the median as new numbers are added. The median is defined as:

If the count of numbers is odd, the median is the middle element when sorted.

If the count of numbers is even, the median is the average of the two middle elements.

Since the numbers are coming in as an infinite stream, we must efficiently maintain the median without sorting the entire list each time. Given an infinite series of integers, continuously find the median after inserting each new number. The approach should efficiently handle real-time updates.

Using Two Heaps (Efficient Approach - $O(\log N)$ per insertion)

Maintain two heaps:

Max Heap (Left half of numbers)

Min Heap (Right half of numbers)

Insert new elements in the appropriate heap.

Balance both heaps so that their sizes differ by at most 1.

Median Calculation:

If both heaps have equal size, median is the average of top elements.

If one heap is larger, median is the top element of that heap.

Input

The first line contains an integer T ($1 \leq T \leq 10$) representing the number of test cases.

For each test case:

The first line contains an integer N ($1 \leq N \leq 10^6$) representing the number of elements in the stream.

The second line contains N space-separated integers, representing the elements of the stream in the order they arrive.

Output

For each test case, print the median after each insertion in a new line.

Sample Input

1
5
2 4 6 8 10

Sample Output

2
3.0
4
5.0
6

DSCP 93**Addition of Two Linked Lists Representing Large Numbers**

In many applications, we need to add two very large numbers that cannot be stored in standard data types like int or long. These numbers can be efficiently represented using linked lists, where each node stores a single digit of the number, and the head node contains the most significant digit (MSD). Given two singly linked lists, each node storing one digit, add the two linked lists as if they were large numbers and store the result in a new linked list.

Example 1: Adding Two Numbers

List1: $7 \rightarrow 5 \rightarrow 9 \rightarrow 4 \rightarrow 6$ (Represents 75946)

List2: $8 \rightarrow 4$ (Represents 84)

75946

+ 84

76030

$7 \rightarrow 6 \rightarrow 0 \rightarrow 3 \rightarrow 0$

Example 2: With Carry Over

List1: $9 \rightarrow 9 \rightarrow 9$

List2: 1

999

+ 1

1000

$1 \rightarrow 0 \rightarrow 0 \rightarrow 0$

Input

The first line contains an integer T ($1 \leq T \leq 10$) representing the number of test cases.

For each test case:

The first line contains an integer N ($1 \leq N \leq 1000$) representing the number of digits in List1.

The second line contains N space-separated integers representing the digits of List1.

The third line contains an integer M ($1 \leq M \leq 1000$) representing the number of digits in List2.

The fourth line contains M space-separated integers representing the digits of List2.

Output

For each test case, print the sum as a new linked list, where each node represents a single digit of the sum.

Sample Input

2

5

7 5 9 4 6

Sample Output

$7 \rightarrow 6 \rightarrow 0 \rightarrow 3 \rightarrow 0$

$1 \rightarrow 0 \rightarrow 0 \rightarrow 0$

2

8 4

3

9 9 9

1

1

DSCP 94**Stack with O(1) Minimum Retrieval**

Stacks are a fundamental data structure used for storing and retrieving elements in a Last In, First Out (LIFO) manner. A common problem in stack operations is to efficiently retrieve the minimum element in the stack at any given time.

Design a stack that supports the following operations in O(1) time complexity:

1. Push(x) – Insert an element into the stack.
2. Pop() – Remove the top element from the stack.
3. GetMinimum() – Retrieve the minimum element in the stack in O(1) time.

Design a stack that supports the following operations in O(1) time using an auxiliary stack (min stack) to keep track of the minimum value at every step.

Main stack	Min stack
5 → top	1 → top
1	1
4	2
6	2
2	2

After popping twice we get

Main stack	Min stack
4 - → top	2 → top
6	2
2	2

Input

The first line contains an integer T ($1 \leq T \leq 10$) representing the number of test cases.

For each test case:

The first line contains an integer N ($1 \leq N \leq 1000$) representing the number of operations.

The next N lines contain either:

"Push X" (where X is an integer to push onto the stack)

"Pop" (removes the top element)

"GetMinimum" (returns the minimum value in the stack)

Output

For each "GetMinimum" operation, print the minimum element on a separate line.

If a "Pop" operation is performed on an empty stack, print "Stack is empty".

If a "GetMinimum" operation is performed on an empty stack, print "Stack is empty".

Sample Input	Sample Output
2	1
7	2
Push 5	4
Push 2	4
Push 8	
Push 1	
GetMinimum	
Pop	
GetMinimum	
Pop	
5	
Push 4	
Push 6	
Push 4	
GetMinimum	
Pop	
GetMinimum	

DSCP 95**Valid Stack Permutations**

A stack is a Last In, First Out (LIFO) data structure, meaning that the last element pushed onto the stack will be the first to be popped. The problem at hand requires us to determine whether a given output sequence can be obtained from a specific push order of elements into a stack. Given a sequence of integers 1, 2, 3, 4, 5, 6 that are pushed onto an initially empty stack in the given order, determine whether a given output sequence can be obtained using a valid sequence of push (S) and pop (X) operations.

For example, consider the numbers 1, 2, 3, 4, 5, 6 pushed in order. We must check if a given output order can be achieved by performing valid push and pop operations.

Example 1: Valid Permutation (325641)

analyze if 325641 can be a valid output.

Operations (S = Push, X = Pop):

Push 1 → [1]

Push 2 → [1,2]

Push 3 → [1,2,3]

Pop → 3 (Output: 3)

Pop → 2 (Output: 2)

Push 4 → [1,4]

Push 5 → [1,4,5]

Pop → 5 (Output: 5)

Push 6 → [1,4,6]

Pop → 6 (Output: 6)

Pop → 4 (Output: 4)

Pop → 1 (Output: 1)

Final output sequence: 325641 (Valid)

Operations required: SSSXXSSXSXXX

Example 2: Invalid Permutation (154623)

Now, let's check if 154623 is valid.

Observing the required output:

1 must be popped first.

5 must be popped before 2 (but 2 is pushed before 5).

Since 2 is pushed much earlier than 5, it must be popped before 5, making this sequence impossible.

154623 is not a valid permutation

Input

The first line contains an integer T ($1 \leq T \leq 10$) denoting the number of test cases.

For each test case:

The first line contains an integer N ($1 \leq N \leq 10^6$) representing the number of elements.

The second line contains N space-separated integers representing the sequence that we need to

check.

Output

For each test case, output "Valid" if the given sequence can be obtained from a stack following the given push order. Otherwise, output "Invalid".

Sample Input

2

6

3 2 5 6 4 1

6

1 5 4 6 2 3

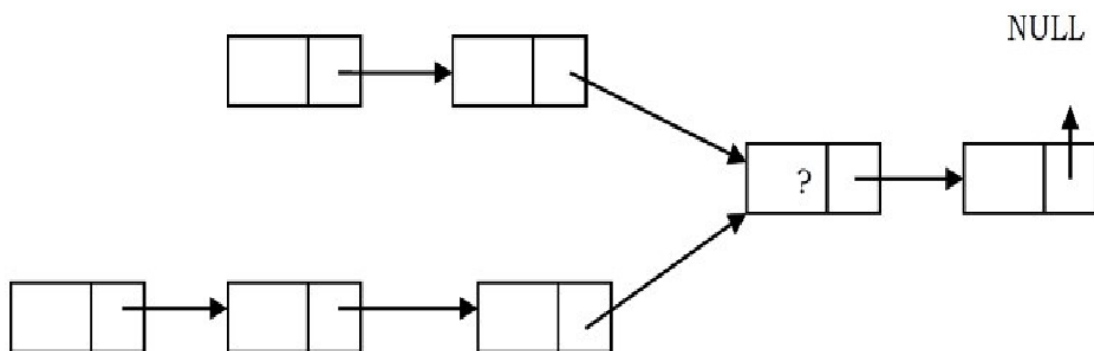
Sample Output

Valid

Invalid

DSCP 96**Intersection of Two Linked Lists using Stacks**

Suppose there are two singly linked lists which intersect at some point and become a single linked list. The head or start pointers of both the lists are known, but the intersecting node is not known. Also, the number of nodes in each of the lists before they intersect are unknown and both lists may have a different number. List1 may have n nodes before it reaches the intersection point and List2 may have m nodes before it reaches the intersection point where m and n may be $m = n$, $m < n$ or $m > n$. Can we find the merging point using stacks?

**Input**

The first line contains an integer T ($1 \leq T \leq 10$) denoting the number of test cases.

For each test case:

The first line contains two integers N and M ($1 \leq N, M \leq 10^6$) representing the number of nodes in List1 and List2, respectively.

The second line contains N space-separated integers representing nodes of List1.

The third line contains M space-separated integers representing nodes of List2.

The last part of both lists (the intersection part) is the same.

Output

For each test case, print the value of the merging node. If there is no intersection, print "-1".

Sample Input

```
2
6 5
10 20 30 40 50 60
15 25 40 50 60
4 3
1 2 3 4
5 6 7
```

Sample Output

```
40
-1
```

DSCP 97**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 position 1, 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 than 100 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

```

1 2 3 4 5
5 4 3 2 1

```

Sample Output

```

1 2 3 4 5
0

```

5 1 2 3 4

5 4 3 2 1

10

5 1 2 3 4

120

DSCP 98**Checking Consecutive Pairs**

You are given a stack of integers, and your task is to check whether each successive pair of numbers in the stack is consecutive. A pair of consecutive numbers can either be increasing or decreasing (i.e., their absolute difference is 1). If the stack has an odd number of elements, the element at the top will be left out of any pair.

If the stack has an even number of elements, divide the stack into successive pairs.

If the stack has an odd number of elements, leave the topmost element out and process the remaining pairs.

For each pair, check if the absolute difference between the two numbers is 1.

A stack with only one element (cannot form any pairs).

A stack where all pairs are consecutive or non-consecutive.

Input

The first line contains an integer T ($1 \leq T \leq 10$) denoting the number of test cases.

Each test case contains:

An integer N ($1 \leq N \leq 10^6$) representing the number of elements in the stack.

The next N space-separated integers represent the stack elements (top element is last in input)..

Output

For each test case, print "true" if all pairs are consecutive, otherwise print "false".

Sample Input

2

9

4 5 -2 -3 11 10 5 6 20

8

1 3 2 4 5 7 6 8

Sample Output

true

false

DSCP 99**Minimum Halls for Event Scheduling**

You are given n events, each defined by a start time and an end time. Each event must be scheduled in a hall, and no two overlapping events can be assigned to the same hall. Your task is to find the minimum number of halls required to schedule all the events without conflicts. This problem can be visualized as an interval graph coloring problem, where each event is represented as an interval on the real line. The goal is to assign a minimum number of colors (halls) such that no two overlapping intervals have the same color. Each event has a fixed start and end time. Two events cannot be scheduled in the same hall if their intervals overlap. The goal is to find the minimum number of halls required to accommodate all events without conflicts. The number of events (n) can be large, so an efficient algorithm is required which can also be applied in real-World applications like conference room Booking, assigning minimum rooms for overlapping meetings. airport runway Scheduling, Allocating runways for flights landing and departing, network bandwidth allocation: assigning bandwidth to simultaneous data streams.

Input

The first line contains an integer n ($1 \leq n \leq 100,000$), representing the number of events.

Each of the next n lines contains two integers s and e ($1 \leq s < e \leq 10^6$), representing the start time and end time of an event.

Output

A single integer, representing the **minimum number of halls** required to schedule all events without conflicts.

Sample Input

5
1 4
2 5
3 6
7 8
8 9

Sample Output

3

DSCP 100**Interleaving the First and Second Half of a Queue**

A shoe maker has N orders from customers which he must satisfy. The shoe maker can work on only one order at a time. Given a queue of integers, rearrange the elements by interleaving the first half of the list with the second half of the list. For example, suppose a queue stores the following sequence of values: [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. Consider the two halves of this list: first half: [11, 12, 13, 14, 15] second half: [16, 17, 18, 19, 20]. These are combined in an alternating fashion to form a sequence of interleaved pairs: the first values from each half (11 and 16), then the second values from each half (12 and 17), then the third values from each half (13 and 18), and so on. In each pair, the value from the first half appears before the value from the second half. Thus, after the call, the queue stores the following values: [11, 16, 12, 17, 13, 18, 14, 19, 15, 20]

Input

The first line contains an integer T ($1 \leq T \leq 10$) representing the number of test cases.

Each test case consists of:

An integer N (N is even, $2 \leq N \leq 10^6$) representing the number of elements in the queue.

The next N space-separated integers representing the queue elements (from front to rear).

Output

For each test case, print the queue elements after interleaving, separated by spaces.

Sample Input

```
2
10
11 12 13 14 15 16 17 18 19 20
6
1 2 3 4 5 6
```

Sample Output

```
11 16 12 17 13 18 14 19 15 20
1 4 2 5 3 6
```

Description:

This problem involves three containers of sizes 10 pints, 7 pints, and 4 pints. Initially, the 7-pint and 4-pint containers are full, while the 10-pint container is empty. The allowed operation is pouring water from one container into another until either the source is empty or the destination is full. The objective is to determine if there exists a sequence of pourings that results in exactly 2 pints of water in either the 7-pint or 4-pint container.

The problem is modeled as a graph where each state represents the amount of water in the three containers. The edges represent valid pouring actions between the containers. A graph traversal algorithm such as Breadth-First Search (BFS) or Depth-First Search (DFS) is used to explore possible states and determine if the desired condition can be achieved.

Input

Container sizes: (10 pints, 7 pints, 4 pints)

Initial state: (0, 7, 4) - where the first, second, and third numbers represent the amount of water in the 10-pint, 7-pint, and 4-pint containers, respectively.

Goal state: Any state where either the 7-pint or 4-pint container contains exactly 2 pints of water.

Output

A sequence of pouring operations leading to the goal state (if possible).

If no solution exists, return "No solution found."

Sample Input

(10, 7, 4)

Initial State: (0, 7, 4)

Goal: 2 pints in either the 7-pint or 4-pint container

Sample Output

Step 1: Pour from 4-pint to 10-pint → (4, 7, 0)

Step 2: Pour from 7-pint to 4-pint → (4, 3, 4)

Step 3: Pour from 4-pint to 10-pint → (8, 3, 0)

Step 4: Pour from 7-pint to 4-pint → (8, 2, 1)

Goal achieved: 2 pints in the 7-pint container.

In professional wrestling, wrestlers can be classified as either babyfaces (good guys) or heels (bad guys). Given n wrestlers and r rivalry pairs, the goal is to determine whether it is possible to divide the wrestlers into two groups such that each rivalry pair consists of a babyface and a heel. This problem can be modeled as a graph problem where wrestlers are represented as nodes and rivalries as edges between nodes.

You need to check if the graph is bipartite. A bipartite graph can be divided into two disjoint sets where no two nodes within the same set are adjacent. If the graph is bipartite, output the two groups (babyfaces and heels); otherwise, return that it is impossible to divide them.

Treat the wrestlers as nodes and the rivalries as edges.

Check if the graph is bipartite by trying to color the graph with two colors (representing babyfaces and heels).

Start from an unvisited node and assign it a color (either babyface or heel).

Visit all neighboring nodes and assign the opposite color to each of them.

If at any point two adjacent nodes have the same color, the graph is not bipartite, and the division is not possible.

If the graph is bipartite, divide the wrestlers into two groups based on their assigned colors.

If not bipartite, return "IMPOSSIBLE".

Input:

An integer n representing the number of wrestlers.

An integer r representing the number of rivalry pairs.

r pairs of integers, where each pair (a, b) represents a rivalry between wrestlers a and b .

Output

If a valid division is possible, output two lists: one for babyfaces and one for heels.

If not possible, output "Not Possible".

Sample Input

5

4

1 2

2 3

3 4

4 5

Sample Output

Babyfaces: 1 3 5

Heels: 2 4

DSCP 103	Second-Best Minimum Spanning Tree
<p>Given an undirected, connected graph $G = (V, E)$ with distinct edge weights, the goal is to compute the second-best minimum spanning tree (SB-MST). The SB-MST is a spanning tree whose total weight is the smallest among all spanning trees except the Minimum Spanning Tree (MST). The problem involves:</p> <p>Prove the uniqueness of MST and non-uniqueness of SB-MST.</p> <p>Show that the SB-MST can be obtained by swapping an edge in the MST with an edge outside the MST.</p> <p>Compute the maximum weight edge on paths in a given tree using an $O(V^2)$ algorithm. Design an efficient algorithm to find the second-best MST.</p>	
<p>Input</p> <p>An undirected, connected graph G with V vertices and E edges. Each edge (u, v) has a unique weight $w(u, v)$.</p>	
<p>Output</p> <p>The second-best minimum spanning tree (SB-MST) with its total weight. The edges of SB-MST.</p>	
<p>Sample Input</p> <p>5 7 1 2 3 1 3 1 2 3 4 2 4 2 3 4 5 3 5 6 4 5 7</p>	<p>Sample Output</p> <p>Second-best MST Weight: 12 Edges: (1,3), (1,2), (2,4), (3,5)</p>

DSCP 104**Professor Adam's Children's School Routes**

Professor Adam's two children refuse to walk on the same street block on the same day, though they can meet at corners. Given a map of the town with intersections and streets represented as a graph, determine if both children can reach the school without violating their constraints. The problem is formulated as a maximum flow problem, where we check if we can send two disjoint paths from the professor's house to the school while ensuring no street segment is used by both children.

The intersections are nodes, and the street segments are edges in a graph. We need to find two disjoint paths from the professor's house to the school. Split each intersection node into two nodes to ensure that each street segment is only used by one child at a time. For each intersection, connect the two nodes corresponding to that intersection with an edge with a capacity of 1.

The solution must efficiently handle the graph representation and compute the maximum flow in a reasonable amount of time.

Input

$N\ M \rightarrow$ Number of intersections (N) and streets (M)

$S\ T \rightarrow$ Source (Professor's house) and Destination (School)

M lines \rightarrow Each line contains two integers $U\ V$ representing a street between intersections U and V

Output

"YES" \rightarrow If both children can reach the school without stepping on the same street

"NO" \rightarrow If it is not possible

Sample Input

5 6

1 5

1 2

2 3

3 4

4 5

2 4

3 5

Sample Output

YES

DSCP 105**Escape Problem in an $n \times n$ Grid**

The Escape Problem involves determining whether it is possible to find vertex-disjoint paths from multiple starting points within an $n \times n$ grid to the boundary of the grid. Each vertex in the grid has four neighbors unless it's on the boundary of the grid. Given m starting points inside the grid, the problem asks if there are m vertex-disjoint paths that connect each starting point to a different boundary point.

The grid can be modeled as a flow network, where the vertices and edges have capacities. This problem can be reduced to an ordinary maximum-flow problem where the goal is to determine if there exists a flow of size m , such that m disjoint paths from the starting points to the boundary are found.

You can move in four directions: up, down, left, or right.

You are not allowed to pass through walls, but you can pass through free cells.

The grid will contain walls represented by 1 and free cells represented by 0.

Input

The grid is an $n \times n$ grid.

The input will include m starting points $(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$ representing the m starting positions inside the grid.

A grid with n rows and n columns.

Output

If there are m vertex-disjoint paths to m different boundary points, output "Yes".

Otherwise, output "No".

Sample Input

$n = 4$

$m = 2$

Starting points:

$(1, 1), (2, 2)$

Sample Output

$n = 3$

$m = 2$

Starting points:

$(1, 1), (2, 2)$

DSCP 106**Poll Analysis for Hit Parade Selection**

A company conducts a poll to determine the popularity of its products like records and tapes of hit songs. Respondents are categorized based on sex (male/female) and age (≤ 20 or > 20). Each respondent selects five hits from a predefined set (1 to N, where $N = 30$).

You are given the results of a poll conducted to select the top k songs for a hit parade. The poll consists of votes for each song from a set of voters. Each voter provides a ranked list of songs, and the goal is to determine the top k songs based on the votes.

The task is to analyze the votes and determine the top k songs that should be included in the hit parade. The songs should be selected in such a way that songs that appear at the top of voters' ranked lists are given higher priority. Identify respondents who mentioned one of the top three hits in their category as their first choice.

Input

An integer n representing the number of songs.

An integer m representing the number of voters.

A list of m votes, where each vote is a ranked list of n songs. The ranking list represents the order of preference for each voter, with the first element being the most preferred song.

The integer k representing the number of top songs to be selected.

Output

A list of the top k songs based on the poll results, ordered from the most preferred to the least preferred.

Sample Input

n = 5

m = 4

votes = [

[1, 2, 3, 4, 5], # Voter 1's ranking

[2, 1, 3, 5, 4], # Voter 2's ranking

[3, 1, 2, 4, 5], # Voter 3's ranking

[4, 5, 3, 2, 1] # Voter 4's ranking

]

k = 3

n = 4

m = 3

votes = [

[1, 2, 3, 4], # Voter 1's ranking

[4, 3, 2, 1], # Voter 2's ranking

[3, 2, 1, 4] # Voter 3's ranking

]

k = 2

Sample Output

[1, 3, 2]

[2, 3]

DSCP 107**Verifying Connectivity in a One-Way Street Network**

In the city of Computopia, all streets are one-way. The mayor claims that it is possible to drive from any intersection to any other intersection. The task is to verify whether this claim holds using a graph-theoretic approach, and if not, check whether it is always possible to return to the town hall from any reachable location.

To model this problem, intersections are represented as nodes in a directed graph, and streets are represented as edges between the nodes. The objective is to check two properties:

1. Strong Connectivity: Can you travel from any intersection to any other intersection, i.e., is the graph strongly connected?
2. If the graph is not strongly connected, check a weaker property: Is it always possible to return to the town hall (node 1, representing the starting intersection) from any reachable location? This corresponds to checking whether the graph, when reversed, is strongly connected from the town hall.

Input

An integer n ($1 \leq n \leq 100,000$), representing the number of intersections (nodes).

An integer m ($1 \leq m \leq 200,000$), representing the number of one-way streets (edges).

A list of m directed edges, where each edge is a pair (u, v) meaning there is a one-way street from intersection u to intersection v .

Output

"YES" if the graph is strongly connected, meaning that you can drive from any intersection to any other intersection.

If the graph is not strongly connected, check if the second property holds:

"YES" if you can always return to the town hall (intersection 1) from any reachable location.

"NO" if you cannot return to the town hall from some reachable location.

Sample Input

$n = 4$

$m = 4$

edges = [(1, 2), (2, 3), (3, 4), (4, 1)]

Sample Output

YES

$n = 4$

$m = 3$

edges = [(1, 2), (2, 3), (3, 4)]

NO

DSCP 108	Maximizing Expected Profit for Yuckdonald's Restaurant
<p>Yuckdonald's, a fast-food chain, is considering opening a series of restaurants along Quaint Valley Highway (QVH). The highway has n potential restaurant locations, each with a specific expected profit from opening a restaurant. The locations are specified by their distances from the starting point of the highway, which are given in increasing order.</p> <p>The challenge is to determine the maximum expected total profit from opening a series of restaurants along these locations, given the following constraints:</p> <p>A restaurant can be opened at each of the n locations.</p> <p>The expected profit from opening a restaurant at location i is p_i, where $p_i > 0$ for every location.</p> <p>If a restaurant is opened at location i, then no other restaurant can be opened within k miles of that location. This constraint means that if a restaurant is placed at location i, the next restaurant can only be placed at a location that is at least k miles away from i.</p> <p>The goal is to design an efficient algorithm that computes the maximum expected total profit subject to these constraints.</p>	
<p>Input</p> <p>n: The number of possible locations on the highway.</p> <p>k: The minimum distance in miles that must be maintained between any two restaurants.</p> <p>m_1, m_2, \dots, m_n: The distances of these n locations along the highway.</p> <p>p_1, p_2, \dots, p_n: The expected profits from opening a restaurant at locations m_1, m_2, \dots, m_n respectively.</p>	
<p>Output</p> <p>The maximum expected total profit that can be obtained from opening restaurants along the highway subject to the given constraints.</p>	
<p>Sample Input</p> <p>$n = 5$</p> <p>$k = 3$</p> <p>$m = [1, 4, 7, 10, 13]$</p> <p>$p = [10, 20, 15, 30, 25]$ 55</p>	<p>Sample Output</p> <p>55</p>

In the Pirate Division Problem, a group of pirates must divide a sum of money amongst themselves according to a strict set of rules. The process involves negotiations and voting, where each pirate has specific priorities: staying alive is the top priority, followed by maximizing the amount of money they receive. The rules for dividing the money are as follows:

1. The senior-most pirate (the one who proposes first) must present a division of the \$300 among all the pirates.
2. The remaining pirates vote on the proposed division. If the senior pirate gets at least half of the votes (including their own), the division stands, and the pirates receive their share.
3. If the proposal is rejected, the senior pirate is killed, and the next pirate (who is now the senior-most pirate) presents a new proposal.
4. This process continues until there are only two pirates left.

The pirates are all intelligent, meaning they will make decisions based on logic, considering both the desire to stay alive and to maximize their portion of the money. In particular, the senior pirate knows that if they are killed, the next pirate will have to make the decision, and that pirate will use the same reasoning.

Input

A single integer, N , which represents the number of pirates ($N = 6$ in this case).

The total sum of money, M , which is \$300.

The pirates' preferences for staying alive, followed by maximizing the money they receive.

Output

The division of the \$300 among the pirates.

The number of pirates who survive after the final division.

Sample Input

6

Sample Output

Pirate 1: \$100, Pirate 2: \$0, Pirate 3: \$0, Pirate 4: \$50, Pirate 5: \$50, Pirate 6: \$100

Surviving Pirates: 6

Vito's family has a rather peculiar tradition. They love organizing reunions, and during these reunions, they always figure out a fun activity to do together. One such tradition involves finding out the "median" of the ages of all family members, a task that Vito has been assigned. To calculate the median, Vito needs to sort the ages of all his relatives and then determine the middle value.

However, Vito has a problem: he doesn't know the ages of his family members in an easy-to-process format. He only knows how many years older or younger each relative is compared to himself. The good news is that Vito is clever enough to write a program to figure out the median from this information.

Given a list of relatives' ages relative to Vito's own age, Vito needs to calculate the median of all ages, including his own. The ages of Vito's family members are represented as differences relative to his age. Vito knows how many relatives there are, and he knows how old he is. From this information, your task is to calculate the median age of the entire family, where the median is the middle value when all ages are sorted.

Input

The input starts with an integer n ($1 \leq n \leq 1000$), the number of Vito's relatives. The next line contains n integers x_1, x_2, \dots, x_n ($-100 \leq x_i \leq 100$), representing the difference in age between Vito and each of his relatives. Vito's age is implicitly considered to be zero in this context. Therefore, if the input for x_i is 10, that means the corresponding relative is 10 years older than Vito. If x_i is -5, that means the relative is 5 years younger than Vito.

The input guarantees that the number of relatives is an odd number, meaning that there will always be a single median.

Output

You need to output a single integer representing the median age. The median is the middle value when all ages are sorted in ascending order.

Sample Input

5
10 -5 0 20 -10

Sample Output

0

A shoemaker operates a small shoe manufacturing unit that produces multiple types of shoes. The goal is to determine the optimal production plan for the shoemaker's unit, ensuring that the demand is met while minimizing the cost and time involved in production.

Each type of shoe requires specific quantities of raw materials (e.g., leather, rubber, etc.), labor (e.g., for stitching, assembly, etc.), and time (e.g., machine usage, drying). The challenge is to allocate limited resources such as raw materials, workers, and machines in an efficient manner to meet the required demand and minimize costs.

Key Elements of the Problem:

Demand: The number of shoes of each type that need to be produced to meet customer orders.

Resources: Raw Materials: The quantities of raw materials such as leather, rubber, and other components needed for each shoe.

Labor: The amount of labor required for different tasks such as stitching, assembly, etc.

Machines: The machine time needed to process, assemble, or finish the shoes.

Production Time: The time required to manufacture one pair of shoes, considering the use of labor and machine time.

Material Cost: The cost of raw materials used in manufacturing each shoe.

Labor Cost: The cost associated with labor to manufacture each shoe.

Machine Cost: The cost of operating machines for a specified time to produce shoes.

The main objective is to determine the most efficient production schedule that fulfills the demand while minimizing the total production cost, which includes:

Material costs,

Labor costs,

Machine operation costs,

Time constraints.

Additionally, the production should be optimized based on the available resources and constraints. The problem comes with several constraints that must be taken into account during the optimization process, such as:

Limited Availability of Raw Materials: There is a finite amount of each raw material available, and the production of shoes cannot exceed this availability.

Limited Number of Workers or Machines: There are a limited number of workers or machines available, which restricts the number of shoes that can be produced in a given period.

Minimum or Maximum Production Quantities: There may be constraints on the minimum or maximum number of shoes that must be produced for a specific type of shoe.

Delivery Deadlines: Shoes must be manufactured and delivered by a certain deadline, which impacts the production scheduling.

There may be specific combinations of raw materials, labor, or machines that are required for each shoe type, further complicating the optimization problem.

Input:

A list of shoe types that need to be produced.

The demand for each type of shoe.

The raw material required for each shoe type, including the cost per unit of material.

The labor required for each shoe type, including labor cost per unit.

The machine time required for each shoe type, including the machine cost per unit.

The available quantities of raw materials, labor, and machine time.

The maximum available production time.

Any additional constraints such as deadlines or limits on the number of shoes per type.

Output:

The optimal number of shoes to be produced for each type.

The total cost of production (including raw materials, labor, and machine time).

The production schedule that minimizes costs and time while meeting the demand and constraints.

Sample Input

Shoe Types: [Boots, Sneakers, Sandals]

Demand: [500, 800, 600]

Raw Material Requirements (Leather, Rubber):

Boots: [2, 1], Sneakers: [1, 2], Sandals: [1, 1]

Material Costs (Leather = 3, Rubber = 2)

Labor Requirements (hours):

Boots: 3, Sneakers: 2, Sandals: 1

Labor Costs (per hour = 10)

Machine Time (hours):

Boots: 2, Sneakers: 1, Sandals: 1

Machine Costs (per hour = 5)

Available Resources:

Leather: 1000 units, Rubber: 800 units

Labor: 1000 hours, Machine time: 1500 hours

Maximum Production Time: 1000 hours

Sample Output

Optimal Production Plan:

Boots: 500, Sneakers: 800, Sandals: 600

Total Cost: \$15,000

DSCP 112**Currency Arbitrage Detection Using Exchange Rates**

Currency arbitrage refers to the exploitation of discrepancies in the exchange rates between different currencies, allowing traders to make a profit by converting one currency into another and ultimately back to the original currency, with the end result being more than the initial amount. The problem is commonly modeled as a graph, where each currency is a node, and each exchange rate between two currencies is an edge with a weight that represents the exchange rate. By finding a cycle in the graph where the product of the exchange rates is greater than one, a trader can identify an arbitrage opportunity.

In this problem, we are provided with an $n \times n$ table of exchange rates, where each entry $R[i, j]$ represents the amount of currency c_j that can be obtained by exchanging one unit of currency c_i . The task is to find a cycle in the graph such that the product of the exchange rates around the cycle is greater than one, which would indicate an arbitrage opportunity.

Input:

An integer n ($1 \leq n \leq 100$) representing the number of currencies.

An $n \times n$ matrix R , where $R[i][j]$ represents the exchange rate from currency c_i to currency c_j .

Output

A boolean indicating whether an arbitrage opportunity exists.

If an arbitrage is found, the output should be the sequence of currencies that form the arbitrage cycle.

Sample Input

```
4
1.0 0.75 0.7 0.9
1.25 1.0 1.1 1.2
1.4286 0.9091 1.0 1.3
1.1111 0.8333 0.7692 1.0
```

Sample Output

```
Arbitrage detected: Yes
Arbitrage cycle: [1, 2, 3, 4]
```


DSCP 113	Freckles
<p>"Freckles" programming challenge revolves around analyzing a given grid of size $m \times n$, where each cell contains a character representing either a dot ('.') or a freckle ('*'). The task is to determine the number of freckle clusters that exist within the grid. A freckle cluster is defined as a contiguous region of adjacent freckle cells that are connected either vertically, horizontally, or diagonally.</p> <p>The challenge tests the contestant's ability to process 2D grids and utilize Depth First Search (DFS) or Breadth First Search (BFS) algorithms to traverse the grid and identify connected components.</p>	
<p>Input:</p> <p>First line: Two integers m and n (the dimensions of the grid).</p> <p>The next m lines: A grid of characters where each character is either '.' or '*'.</p>	
<p>Output:</p> <p>Output a single integer that represents the total number of freckle clusters in the given grid.</p>	
<p>Sample Input</p> <pre> 5 5*.. </pre>	<p>Sample Output</p> <pre> 3 </pre>

DSCP 114**The Hacker's Encrypted Filesystem**

A hacker group has encrypted files using a nested directory structure. Each folder contains subfolders and files, forming a tree. The only way to decrypt files is to process queries on this file system.

Operations include:

Count files in a given folder.

Find the largest file in a directory subtree.

Delete a folder (removing all its subfolders and files).

The system uses a Tree with DFS (Depth-First Search) and Segment Tree for fast operations.

Input:

An integer N ($1 \leq N \leq 10^5$) → Number of folders.

$N-1$ lines with two integers (P, C) → Folder C is a subfolder of P .

An integer M ($1 \leq M \leq 10^5$) → Number of files.

M lines with three values (Folder ID, File ID, Size) → File exists in a folder.

An integer Q ($1 \leq Q \leq 10^5$) → Number of queries.

Q queries of three types:

C X → Count files in folder X .

L X → Largest file in folder X .

D X → Delete folder X and its contents.

Output:

For each query, return the required result.

Sample Input

```
5
1 2
1 3
2 4
2 5
4
2 101 500
3 102 300
4 103 800
5 104 700
3
C 2
L 1
D 2
```

Sample Output

```
File Count in 2: 3
Largest File in 1: 800
Folder 2 Deleted
```

DSCP 115**The Infinite Scroll Newsfeed**

A social media app implements an infinite scroll newsfeed, where users see posts from their friends in real-time. Each post has:

A timestamp (newer posts appear first).

A popularity score (high-score posts rank higher).

A category (news, entertainment, tech, etc.).

Users request

The top K trending posts.

All posts in a given category.

The latest post by a specific user.

The system uses a Max-Heap (Priority Queue) and Hash Maps for fast retrieval.

Input:

An integer N ($1 \leq N \leq 10^5$) → Number of posts.

N lines with four values: Post ID, Timestamp, Score, Category.

An integer Q ($1 \leq Q \leq 10^5$) → Number of queries.

Q queries of three types:

T K → Get the top K posts.

C X → Get all posts in category X.

U X → Get the latest post by user X.

Output:

For each query, return the required result.

Sample Input

```
5
101 1700000000 500 news
102 1700000005 800 tech
103 1700000010 600 news
104 1700000020 1000 sports
105 1700000030 750 news
3
T 3
C news
U 102
```

Sample Output

```
Top 3 Posts: [104, 102, 105]
News Posts: [105, 103, 101]
Latest Post by User 102: 102
```

DSCP 116**The Maximum Flow of the River**

A river has several N tributaries. Water flows from these tributaries into different parts of the river, forming a network of pipes. The amount of water that can flow through each pipe is limited by its capacity. The goal is to determine the maximum amount of water that can flow from the source tributary to the sink tributary, given the capacity constraints of each pipe.

Implement the Edmonds-Karp algorithm for maximum flow using BFS and Ford-Fulkerson.

Track the flow across each pipe and update the capacities as the flow increases.

Determine if a bottleneck (a pipe with limited capacity) exists and how to address it.

Use priority queues (min-heaps) to find the most significant bottleneck in the network.

Input:

An integer N ($1 \leq N \leq 10^5$) → Number of tributaries.

An integer M ($1 \leq M \leq 10^5$) → Number of pipes.

M lines of pipes defined by three integers (u, v, c) representing a pipe from tributary u to tributary v with capacity c .

An integer S ($1 \leq S \leq N$) → Source tributary.

An integer T ($1 \leq T \leq N$) → Sink tributary.

Output:

Maximum flow from source S to sink T .

List of pipes involved in the bottleneck and their adjusted capacities.

Sample Input

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

Sample Output

```
Maximum Flow: 15
Bottleneck pipes: [(2, 4, 5)]
```

A genealogist needs to build a family tree for a historical society. Each individual is uniquely identified by an ID and has a set of relationships with other individuals, including parents, children, and siblings. The goal is to build the family tree and process various queries related to ancestor searches, relationship paths, and family size.

Build a binary tree where each node represents an individual, with links to parents and children.

Implement functions to find an individual's ancestors up to a certain generation.

Query for the shortest path between two individuals, given their relationship as either siblings or cousins.

Count the number of descendants for each individual.

Implement balanced insertion of nodes to maintain a balanced family tree (using AVL or Red-Black Tree).

Input:

An integer N ($1 \leq N \leq 10^6$) → Number of individuals.

$N-1$ lines describing relationships between individuals, in the form (parent, child).

An integer Q ($1 \leq Q \leq 10^6$) → Number of queries.

Q queries as:

A u → Find the ancestors of individual u .

P u v → Find the shortest path between individuals u and v .

D u → Find the number of descendants of individual u .

Output:

For each query, return the required result.

Sample Input

```
6
5
1 2
1 3
2 4
2 5
3 6
3
A 2
P 4 5
D 1
```

Sample Output

```
Ancestors of 2: [1]
Shortest Path between 4 and 5: [2]
Descendants of 1: 4
```

DSCP 118**The Task Scheduler**

In a distributed system, several tasks are assigned to workers. Each task has a priority, and some tasks are dependent on others. The system needs to schedule tasks while respecting their dependencies, ensuring that all tasks are executed in the correct order.

Implement a Topological Sort algorithm to determine the correct order of task execution.

Track the completion time for each task using a priority queue (min-heap).

Determine if there are any cyclic dependencies that prevent task scheduling.

Input:

An integer N ($1 \leq N \leq 10^6$) → Number of tasks.

An integer M ($1 \leq M \leq 10^6$) → Number of dependencies.

M lines of dependencies as pairs (u, v) , meaning task u depends on task v .

An integer Q ($1 \leq Q \leq 10^6$) → Number of queries.

Q queries of the form:

$S \rightarrow$ Return the schedule of tasks based on topological sort.

$C\ u\ v \rightarrow$ Check if task u depends on task v

Output:

For each query, return the required result.

Sample Input

4
3
1 2
2 3
3 4
2
S
C 2 4

Sample Output

Task Schedule: [1, 2, 3, 4]
Does task 2 depend on task 4? No

The Grand Library contains N books, each having a title and an associated publication year. The library uses a binary search tree (BST) to store the books in such a way that for any given node, the left subtree contains books with titles lexicographically smaller than the node's title, and the right subtree contains books with titles lexicographically larger.

The library has been under renovation, and the librarian needs to perform the following tasks:

1. Insert new books into the BST.
2. Delete books from the BST by their title.
3. Search for books within a given title range and find the book with the oldest publication year.
4. Perform in-order traversal of the BST to list all books in lexicographical order.

Implement a BST to manage the books. Support the following operations:

Insert book by title and publication year. Delete book by title. Search for books within a title range and find the book with the oldest publication year.

In-order traversal of the BST.

Input:

An integer N ($1 \leq N \leq 10^5$) → Number of books.

N lines of book titles and their publication years.

An integer Q ($1 \leq Q \leq 10^5$) → Number of operations.

Q lines of operations in the form:

I title year → Insert a book.

D title → Delete a book.

S start_title end_title → Search for books in the range.

T → Perform an in-order traversal.

Output:

For each operation, output the result:

For S , output the title of the book with the oldest publication year in the range.

For T , output the list of book titles in lexicographical order.

Sample Input

```
5
BookA 2001
BookB 1999
BookC 2005
BookD 1998
BookE 2010
4
I BookF 2000
S BookA BookC
D BookB
T
```

Sample Output

```
BookA
BookD BookF BookA BookC BookE
```

DSCP 120**The Maze of the Graph King**

Long ago, a great ruler known as the Graph King built a maze with N interconnected chambers. Each chamber had bi-directional tunnels leading to other chambers. The king stored his wealth in a hidden vault and placed a guardian monster to guard it. The only way to reach the treasure was to find the shortest path from the entrance (chamber 1) to the vault (chamber V).

To help adventurers, the king left behind a graph-based navigation system that records the tunnels and their lengths in an adjacency list. However, the map has missing and redundant paths, and the adventurer must efficiently find the shortest route.

Implement Dijkstra's algorithm or *A search** to determine the shortest path from chamber 1 to chamber V .

Detect if there exists a negative-weight cycle using Bellman-Ford's algorithm.

Find and mark redundant tunnels that do not contribute to the shortest path.

If the treasure is unreachable, determine the minimum number of tunnels to be added to connect the chambers.

Input

An integer N ($1 \leq N \leq 10^5$) → Number of chambers.

An integer E ($1 \leq E \leq 10^5$) → Number of tunnels.

E lines of three integers (u, v, w) representing a tunnel between chambers u and v with length w .

An integer V ($1 \leq V \leq N$) → Chamber number where the treasure is hidden.

Output

The shortest path length from chamber 1 to chamber V . If unreachable, return -1.

A list of redundant tunnels.

The number of extra tunnels needed to connect all chambers.

Sample Input

5 6

1 2 4

1 3 1

2 3 2

2 4 5

3 4 8

4 5 3

5

Sample Output

Shortest path: 8

Redundant tunnels: [(3, 4, 8)]

Extra tunnels needed: 0

In programming languages, parentheses play a critical role in the correct execution of code. A stack-based approach can be used to check if parentheses are correctly matched in a code string. The problem is to validate the balance of parentheses, which can include round brackets `()`, curly braces `{}`, and square brackets `[]`. The solution must utilize stack operations (push and pop) to match opening and closing parentheses and ensure their correct order.

The solution needs to:

1. Parse the string character by character.
2. Push opening parentheses onto the stack.
3. For each closing parenthesis, pop the stack and check if it matches the corresponding opening parenthesis.
4. Ensure that at the end of parsing, the stack is empty (all opening parentheses have been matched).

The algorithm should handle multiple types of parentheses and return whether the parentheses in the string are balanced or not.

Input

A string containing parentheses `()`, `{}`, `[]`, and other characters that should be ignored.

The length of the string is n .

Output

Return True if the parentheses are balanced, otherwise return False.

Sample Input

arduino
CopyEdit
String: "(a + b) * {x + y}"

Sample Output

graphql
CopyEdit
True

DSCP 122**The City of Balanced Skyscrapers**

A futuristic city consists of N skyscrapers, each having a height $H(i)$. The city wants to balance the skyline to improve energy efficiency by ensuring that each skyscraper has a balanced neighborhood height difference.

To achieve this, the city uses a Segment Tree to efficiently:

Query the maximum and minimum skyscraper heights in any given range $[L, R]$. Determine if a new skyscraper height H_{new} should be inserted at position P to maintain balance.

Find the smallest adjustment needed to make the height difference in any range $\leq K$.

Build a segment tree for fast queries.

Process Q operations:

$Q\ L\ R \rightarrow$ Find the max-min height difference in the range $[L, R]$.

$I\ P\ H_{\text{new}} \rightarrow$ Insert a new skyscraper height at position P .

$F\ L\ R\ K \rightarrow$ Find the minimum height adjustment needed to ensure the height difference in $[L, R]$ is $\leq K$.

Input

An integer N ($1 \leq N \leq 10^5$) \rightarrow Number of skyscrapers.

A list of N integers $H(i)$ \rightarrow Heights of skyscrapers.

An integer Q ($1 \leq Q \leq 10^5$) \rightarrow Number of queries.

Q lines of operations as described.

Output

For each query, return the required result.

Sample Input

```
5
3 8 2 10 6
4
Q 1 3
I 3 7
F 2 5 5
Q 2 5
```

Sample Output

```
6
Inserted at 3
1
4
```

DSCP 123**The Enchanted Spellbook Sorting**

An enchanted spellbook contains N magical spells, each represented as a string. The spells are inscribed in an ancient order that has been disrupted. The only way to restore the spellbook is to sort them based on an unknown lexicographic rule, governed by a priority weight.

Each spell has a unique weight $W(i)$, and the final sorting order must be determined as follows:

If two spells have the same starting character, the one with a higher weight appears first.

If two spells have different starting characters, they should be sorted in a custom order derived from an ancient permutation P of the alphabet.

Implement a custom sorting algorithm that follows the ancient permutation P .

Use Trie or Radix Sort to efficiently organize the spells.

Answer queries where a wizard requests:

K th spell in the sorted order.

All spells starting with a prefix.

Input:

An integer N ($1 \leq N \leq 10^5$) → Number of spells.

A string of length 26 → The permutation P of the alphabet.

N lines, each containing a string $S(i)$ and an integer $W(i)$ ($1 \leq W(i) \leq 10^6$).

An integer Q ($1 \leq Q \leq 10^5$) → Number of queries.

Q lines of two types:

Kx → Find the K -th spell in the sorted list.

P prefix → List all spells starting with prefix.

Output:

For each query, return the required result.

The updated state of the deque after all operations.

Sample Input

```
5
zyxwvutsrqponmlkjihgfedcba
fireball 50
flame 30
frost 80
shock 60
storm 70
3
K 2
P f
K 4
```

Sample Output

```
storm
frost fireball flame
flame
```

DSCP 124**The AI Traffic Control System**

In a smart city, an AI-driven traffic control system regulates intersections to reduce congestion. The system models the city's roads as a directed weighted graph, where intersections are nodes and roads are edges with weights representing congestion levels.

Traffic Spike Detection: If any road's congestion level exceeds T , suggest alternative routes.

Real-time Road Closure Handling: If a road is closed, dynamically update the graph and reroute vehicles. Predictive Congestion Analysis: Use Fenwick Tree (Binary Indexed Tree) to quickly update and query congestion levels.

Input:

N ($1 \leq N \leq 10^5$) \rightarrow Number of intersections.

E ($1 \leq E \leq 10^6$) \rightarrow Number of roads.

E lines of three integers (u, v, w) \rightarrow Road between intersections u and v with congestion level w .

T ($1 \leq T \leq 100$) \rightarrow Congestion threshold.

Q ($1 \leq Q \leq 10^5$) \rightarrow Number of queries.

Q lines:

$R\ S\ D \rightarrow$ Find the fastest route from S to D .

$C\ u\ v \rightarrow$ Close the road between u and v .

$U\ u\ v\ w \rightarrow$ Update the congestion level of the road (u, v) to w .

Output:

Fastest route length or -1 if unreachable.

Alternative route suggestions if congestion is above T .

Updated road network after closures.

Sample Input

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

Sample Output

```
Fastest route: 6
Road (2,3) closed.
Fastest route: 8
```

DSCP 125**The Hacker's Cache**

A hacker group is designing a Least Recently Used (LRU) Cache for a system that stores data blocks. The cache has a capacity K , and when it is full, the least recently used block is evicted.

The system receives requests for blocks, and if a block is in the cache, it moves to the most recently used position. If not, the block is loaded, and if the cache is full, the least recently used block is removed.

Implement an LRU Cache using Doubly Linked List and HashMap.

Process Q requests to access or insert blocks.

Return the state of the cache after every request.

Input:

Two integers N ($1 \leq N \leq 10^6$) \rightarrow Number of requests, K ($1 \leq K \leq 10^5$) \rightarrow Cache capacity.

N lines of requests:

A $X \rightarrow$ Access block X .

I $X \rightarrow$ Insert block X .

Output:

The state of the cache after every request.

Sample Input

6 3
I 5
I 10
A 5
I 20
I 30
A 10

Sample Output

Cache: [5]
Cache: [10, 5]
Cache: [5, 10]
Cache: [20, 5, 10]
Cache: [30, 20, 5]
Cache: [10, 30, 20]

DSCP 126**The Haunted Castle's Door Puzzle**

A haunted castle has N doors that form a complex circular locking mechanism. Each door is connected to two other doors, forming a cycle. To unlock the castle, one must open all doors in the correct sequence. However, the sequence is hidden in an encrypted pattern.

The doors are stored in a doubly linked list, and the puzzle master must:

1. Find the correct rotation that unlocks the doors.
2. Perform fast insertions and deletions when a new door is added or removed.
3. Check if a door sequence exists in the current cycle.
1. Rotate the doubly linked list to match the correct unlock pattern.
2. Efficiently insert and remove doors while maintaining the cycle.
3. Search for a given subsequence in the door pattern.

Input:

An integer N ($1 \leq N \leq 10^5$) → Number of doors.

N space-separated integers $D(i)$ → Door lock sequence.

An integer Q ($1 \leq Q \leq 10^5$) → Number of operations.

Q operations:

$R\ X$ → Rotate the sequence X times.

$I\ P\ V$ → Insert a new door V at position P .

$D\ P$ → Delete the door at position P .

$S\ K\ K_1\ K_2\ \dots\ K_k$ → Search if sequence K_1, K_2, \dots, K_k exists.

Output:

For each S operation, return "FOUND" or "NOT FOUND"

Sample Input

5
10 20 30 40 50
4
R 2
I 3 25
S 3 20 30 40
D 4

Sample Output

25 30 40 50 10 20
FOUND

DSCP 127**The Kingdom's Road Network**

A medieval kingdom consists of N cities connected by M roads. Each road has a difficulty level that makes it harder to travel. The king wants to establish an efficient road network that ensures:

1. The shortest possible travel time between any two cities.
2. The minimum number of roads required to keep all cities connected.
3. The best alternative route if the main road between two cities is blocked.

The kingdom's road network can be represented as a weighted, undirected graph.

Find the minimum spanning tree (MST) to ensure all cities are connected with minimal difficulty.

Find the shortest path between any two cities.

Find the second-best shortest path when the main road is closed.

Input:

Two integers N ($1 \leq N \leq 10^5$), M ($1 \leq M \leq 10^5$) → Cities and roads.

M lines of three integers u, v, w → Road between cities u and v with difficulty w .

An integer Q ($1 \leq Q \leq 10^5$) → Number of queries.

Q queries of the form $X Y$ → Find the best alternative route between X and Y .

Output:

MST total cost.

Shortest path length for each query.

Second-best shortest path if the best route is blocked.

Sample Input

```
5 6
1 2 4
1 3 1
2 3 2
2 4 5
3 4 8
4 5 3
2
1 5
2 4
```

Sample Output

```
MST Cost: 10
Shortest Path: 8
Alternative Path: 9
```

DSCP 128**Professor Diogenes' Chip Testing**

Professor Diogenes has N identical integrated-circuit chips. Each chip can test another chip and return a result. However, the results can only be trusted if at least one of the chips in the test pair is good. The professor needs to determine which chips are good using minimal tests.

Each test is conducted in a test jig that accommodates two chips at a time. When tested, each chip gives a verdict on the other. The results follow these rules:

1. If both chips are good → They both correctly identify each other as good.
2. If one chip is good and the other is bad → The good chip correctly identifies the bad chip, but the bad chip may give any random result.
3. If both chips are bad → They can report any result (randomly saying good or bad).

Since bad chips give unreliable results, the professor must find a strategy to determine the set of good chips with the fewest number of tests.

Efficiently identify at least one good chip (if one exists).

Determine the entire set of good chips using minimal tests.

Output the indices of all good chips in ascending order.

Input:

An integer N ($1 \leq N \leq 10^6$) → Number of chips.

A list of pairwise test results given as (A, B, resultA, resultB) for each test:

A, B ($1 \leq A, B \leq N$) → Indices of the two tested chips.

resultA (0 or 1) → Chip A's verdict on B (1 = Good, 0 = Bad).

resultB (0 or 1) → Chip B's verdict on A (1 = Good, 0 = Bad).

Output:

A sorted list of indices representing the good chips.

Sample Input

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

Sample Output

```
[1, 3, 5, 6, 9, 10]
```


DSCP 129	Jug Pairing Problem with Minimum Comparisons
<p>You are given n red water jugs and n blue water jugs, where each jug holds a unique amount of water. For each red jug, there is a corresponding blue jug that holds the same amount of water. Your task is to find the optimal grouping of these jugs into pairs, where each pair consists of one red jug and one blue jug, such that the amount of water in the red jug equals the amount of water in the blue jug.</p> <p>You are allowed to perform the following operation to compare the jugs:</p> <p>Pick a red jug and a blue jug, fill the red jug with water, and pour the water into the blue jug.</p> <p>This operation will tell you whether the red jug holds more, less, or the same amount of water as the blue jug. The goal is to minimize the number of comparisons required to identify the correct pairs of red and blue jugs.</p>	
<p>Input:</p> <p>An integer n ($1 \leq n \leq 10^6$) → The number of red and blue jugs (n red and n blue jugs).</p> <p>A list of n red jug volumes: A sequence of n unique integers representing the amount of water each red jug can hold.</p> <p>A list of n blue jug volumes: A sequence of n unique integers representing the amount of water each blue jug can hold.</p>	
<p>Output:</p> <p>A list of pairs of jugs where each pair consists of one red jug and one blue jug that hold the same amount of water. The output should be in the format of tuples (red_jug, blue_jug).</p>	
<p>Sample Input</p> <p>$n = 4$ Red jugs: [3, 5, 8, 2] Blue jugs: [5, 2, 3, 8]</p>	<p>Sample Output</p> <p>[(3, 3), (5, 5), (8, 8), (2, 2)]</p>

DSCP 130**Compactifying a Doubly Linked List**

In this problem, we are given a doubly linked list L with n nodes, represented by three arrays:

key[]: Stores the values associated with the nodes.

pre[]: Stores the previous pointers of each node.

next[]: Stores the next pointers of each node.

The arrays are part of a larger array of size m , where exactly n items belong to the doubly linked list L and the remaining $m-n$ items are available in a free list F . The free list is maintained using a doubly linked structure and is managed by the allocate-object and free-object procedures.

The task is to write a procedure compactify-list (L, F) that rearranges the nodes in list L so that they occupy array positions from 1 to n , while the free list F occupies positions from $n+1$ to m .

The procedure should operate in $O(n)$ time and use only constant space (ignoring the input and output arrays).

Input:

key[]: An array of size m where each element stores the key value of the node.

pre[]: An array of size m where each element stores the index of the previous node in the doubly linked list, or 0 if there is no previous node.

next[]: An array of size m where each element stores the index of the next node in the doubly linked list, or 0 if there is no next node.

free[]: A doubly linked free list of size $m-n$, where the elements store indices of free slots available for allocation.

Output:

The arrays key[], pre[], and next[] should be updated so that the nodes of list L occupy positions from 1 to n .

The free list should remain correct, occupying positions from $n+1$ to m .

Sample Input

key[]: [5, 8, 3, 2, 6, 9, 7, 4, 1]

pre[]: [0, 0, 1, 2, 3, 4, 5, 6, 7]

next[]: [1, 2, 3, 4, 5, 6, 7, 8, 0]

free[]: [10, 11, 12, 13, 14]

Sample Output

key[]: [5, 8, 3, 2, 6, 9, 7, 4, 1]

pre[]: [0, 1, 2, 3, 4, 5, 6, 7, 0]

next[]: [1, 2, 3, 4, 5, 6, 7, 8, 0]

free[]: [9, 10, 11, 12, 13]

In this problem, we explore the behavior of hash tables with open addressing during insertions. Given an open-addressed hash table of size m with n items, where $m = 2n$, we analyze the probability bounds and expected values related to probe lengths during insertions. The goal is to understand the longest probe sequence during n insertions under the assumption of uniform hashing.

The problem is divided into four parts:

Showing the probability bound for an insertion requiring more than k probes.

Showing the probability of an insertion requiring more than $2 \lg n$ probes.

Analyzing the probability that the maximum number of probes required by any insertion exceeds $2 \lg n$.

Calculating the expected maximum length of probe sequences.

Input:

An integer n ($1 \leq n \leq 10^6$) → The number of elements to be inserted into the hash table.

An integer m ($1 \leq m \leq 10^6$) → The size of the hash table, where $m = 2n$.

Output:

A statement showing that the probability is at most 2^{-k} for the insertion requiring strictly more than k probes.

A statement proving that the probability is $O(1/n^2)$ that the i^{th} insertion requires more than $2 \lg n$ probes.

A statement showing that the probability of the maximum number of probes required by any insertion exceeding $2 \lg n$ is $O(1/n)$.

An expected value calculation for the maximum number of probes, $E[X]$, for the longest probe sequence.

Sample Input

$n = 100$ $m = 200$

Sample Output

Part a: Probability that the i^{th} insertion requires more than k probes is at most 2^{-k} .

Part b: Probability that the i^{th} insertion requires more than $2 \lg n$ probes is $O(1/n^2)$.

Part c: Probability that the maximum number of probes exceeds $2 \lg n$ is $O(1/n)$.

Part d: Expected length of the longest probe sequence is $O(\lg n)$.

DSCP 132**Professor Bunyan's Binary Search Tree**

Professor Bunyan has made an intriguing claim about binary search trees (BSTs). He asserts that, when searching for a key k in a binary search tree, the search eventually reaches a leaf node. At this point, the tree can be divided into three sets:

Set A: All the keys to the left of the search path.

Set B: The keys that lie on the search path from the root to the leaf.

Set C: All the keys to the right of the search path.

Professor Bunyan's hypothesis is that for any three keys a in A , b in B , and c in C , the relationship must hold true:

$$a < b < c$$

In other words, for every possible combination of keys chosen from A , B , and C , the key from A should always be less than the key from B , and the key from B should always be less than the key from C .

The task is to disprove Professor Bunyan's claim by providing the smallest possible counterexample — a case where $a < b < c$ does not hold for all keys selected from A , B , and C .

Given a binary search tree and the key k being searched for, identify the three sets: A , B , and C .

Find the smallest counterexample where the keys a , b , c from the sets A , B , and C do not satisfy the relationship $a < b < c$.

Input:

The first line contains an integer N ($1 \leq N \leq 1000$), the number of nodes in the binary search tree.

The second line contains N integers representing the keys of the nodes in the tree in level-order.

The third line contains the key k ($1 \leq k \leq 1000$) to search for in the binary search tree.

Output:

If there is a counterexample to Professor Bunyan's claim, print "Counterexample found" and display the smallest counterexample in the form:

$$a < b < c \text{ (where } a \text{ belongs to } A, b \text{ belongs to } B, \text{ and } c \text{ belongs to } C).$$

If no counterexample is found, print "Professor Bunyan's claim holds".

Sample Input

7 50 30 70 20 40 60 80 40

Sample Output

Counterexample found 20 < 40 < 60

Professor Teach is analyzing the RB-INSERT-FIXUP algorithm used in Red-Black Trees and is concerned that it might set `T.nil.color` to RED. If this happens, the termination condition in line 1 of the algorithm may fail when the inserted node reaches the root, leading to an infinite loop.

A Red-Black Tree maintains five essential properties:

1. Each node is either red or black.
2. The root is always black.
3. All leaves (`T.nil`) are black.
4. If a node is red, both its children must be black (No two consecutive red nodes).
5. Every path from a node to its descendant leaves must have the same number of black nodes (black-height property).

To address the professor's concern, we must prove that RB-INSERT-FIXUP never sets `T.nil.color` to RED.

Implement the RB-INSERT-FIXUP procedure on a Red-Black Tree.

Show that the `T.nil.color` (representing the sentinel nil leaves) always remains black throughout the execution.

Verify that RB-INSERT-FIXUP correctly maintains Red-Black Tree properties after each insertion.

Provide evidence that no case in RB-INSERT-FIXUP results in `T.nil.color` being red.

Input:

An integer N ($1 \leq N \leq 10^5$) → Number of elements to be inserted into the Red-Black Tree.

N space-separated integers → Elements to be inserted in the given order.

Output:

A message confirming that `T.nil.color` remains black after all insertions.

The in-order traversal of the Red-Black Tree after all insertions.

The color (Red or Black) of each node in the final tree structure.

Sample Input

6
10 20 30 15 25 5

Sample Output

`T.nil.color` is always BLACK.
In-order Traversal: 5(B) 10(R) 15(B) 20(B)
25(R) 30(B)

The Euclidean Traveling Salesman Problem (TSP) requires finding the shortest possible route that connects a given set of n points on a 2D plane, forming a closed tour. This problem is NP-hard, meaning an optimal solution is computationally expensive for large n .

However, a Bitonic TSP simplifies the problem by imposing a restriction:

1. The tour starts at the leftmost point, moves strictly rightward to the rightmost point.
2. Then, the tour returns leftward to the starting point, ensuring a bitonic order (first increasing then decreasing in x-coordinates).

Sort the points by their x-coordinates.

Use dynamic programming to maintain the shortest paths for valid bitonic tours.

Compute the optimal bitonic tour using a recurrence relation.

Output the minimum bitonic tour length.

Input:

An integer n ($2 \leq n \leq 10^5$) → Number of points.

n lines with two space-separated real numbers x y → Coordinates of the points.

No two points share the same x-coordinate.

The algorithm assumes unit-time operations on real numbers.

Output:

A single floating-point value rounded to four decimal places, representing the minimum bitonic tour length.

Sample Input

```
7
1.0 2.0
2.5 3.5
3.0 1.0
4.5 5.0
5.0 2.0
6.0 4.0
7.0 3.0
```

Sample Output

```
14.7632
```

DSCP 135**Filtering**

You are given a sequence of N real numbers between 0 and 1 from standard input. Your task is to determine which operations require storing all N values and which can be performed using only a fixed number of variables or fixed-size arrays (independent of N). Which of the following require saving all the values from standard input (in an array, say), and which could be implemented as a filter using only a fixed number of variables and arrays of fixed size (not dependent on N)? For each, the input comes from standard input and consists of N real numbers between 0 and 1.

- Display the maximum and minimum numbers.
- Display the median of the numbers.
- Display the k^{th} smallest value, for k less than 100.
- Display the sum of the squares of the numbers.
- Display the average of the N numbers.
- Display the percentage of numbers greater than the average.
- Display the N numbers in increasing order.
- Display the N numbers in random order.

Input

The first line contains an integer N ($1 \leq N \leq 10^6$), the number of real numbers.

The next N lines each contain a single real number between 0 and 1.

Output

For each operation, output the result in a new line, following the order of operations given in the problem statement.

Sample Input

7
0.12
0.95
0.50
0.23
0.75
0.36
0.88

5
0.80
0.20
0.50
0.10
0.90

Sample Output

Max: 0.95, Min: 0.12
Median: 0.50
3rd Smallest: 0.36
Sum of Squares: 2.2749
Average: 0.5414
Percentage Greater than Average: 42.86%
Sorted: 0.12 0.23 0.36 0.50 0.75 0.88 0.95
Random Order: 0.75 0.12 0.36 0.88 0.23 0.50 0.95

Max: 0.90, Min: 0.10
Median: 0.50
3rd Smallest: 0.50
Sum of Squares: 1.99
Average: 0.50
Percentage Greater than Average: 40.00%
Sorted: 0.10 0.20 0.50 0.80 0.90
Random Order: 0.20 0.90 0.10 0.80 0.50

DSCP 136	Circular Rotations of a String
<p>Two strings, s and t, are considered circular rotations of each other if one string can be obtained by rotating the characters of the other in a circular manner. For example, ACTGACG is a circular shift of TGACGAC, and vice versa. Detecting this condition is important in the study of genomic sequences. Write a program that checks whether two given strings s and t are circular shifts of one another. The solution is a one-liner with <code>indexOf()</code>, <code>length()</code>, and string concatenation.</p> <p>"WATERBOTTLE" is a circular rotation of "BOTTLEWATER" because shifting the characters results in a match.</p>	
<p>Input</p> <p>The first line contains a string s.</p> <p>The second line contains a string t.</p> <p>Both s and t will have the same length and consist of uppercase English letters (A-Z).</p>	
<p>Output</p> <p>Print "YES" if s is a circular rotation of t.</p> <p>Otherwise, print "NO".</p>	
Sample Input	Sample Output
ACTGACG	YES
TGACGAC	
HELLO	YES
LOHEL	
ABCDE	YES
CDEAB	
ABCDE	NO
EBCDA	
WATERBOTTLE	YES
BOTTLEWATER	
COMPUTER	YES
PUTERCOM	
PYTHON	YES
THONPY	

DSCP 137**Stack Pop Sequences**

A stack follows the Last In, First Out (LIFO) principle, meaning that the last pushed item must be the first to be popped. A client performs an intermixed sequence of push and pop operations using a stack. The push operations insert integers from 0 to 9 in order onto the stack. The pop operations print out the values as they are removed. Your task is to determine which of the given sequences could not occur as a valid pop sequence of a stack.

- a. 4 3 2 1 0 9 8 7 6 5
- b. 4 6 8 7 5 3 2 9 0 1
- c. 2 5 6 7 4 8 9 3 1 0
- d. 4 3 2 1 0 5 6 7 8 9
- e. 1 2 3 4 5 6 9 8 7 0
- f. 0 4 6 5 3 8 1 7 2 9
- g. 1 4 7 9 8 6 5 3 0 2
- h. 2 1 4 3 6 5 8 7 9 0

Input

The input consists of a single line containing 10 space-separated integers, representing a sequence of popped values.

Output

Print "VALID" if the given sequence is a possible stack pop sequence.

Print "INVALID" if the sequence is not possible using a stack.

Sample Input

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

Sample Output

VALID
INVALID
INVALID
VALID
INVALID
INVALID
INVALID
INVALID

Radix Sort is a non-comparative sorting algorithm that sorts numbers digit by digit, starting from the least significant digit (ones place) to the most significant digit. A radix sort for base 10 integers is a mechanical sorting technique that utilizes a collection of bins, one main bin and 10 digit bins. Each bin acts like a queue and maintains its values in the order that they arrive. The algorithm begins by placing each number in the main bin. Then it considers each value digit by digit. The first value is removed and placed in a digit bin corresponding to the digit being considered. For example, if the ones digit is being considered, 534 is placed in digit bin 4 and 667 is placed in digit bin 7. Once all the values are placed in the corresponding digit bins, the values are collected from bin 0 to bin 9 and placed back in the main bin. The process continues with the tens digit, the hundreds, and so on. After the last digit is processed, the main bin contains the values in order.

The sorting process follows these steps:

Initialization: Place all numbers in a main bin (initial unsorted list).

Digit-wise Sorting:

- For each digit position (ones, tens, hundreds, etc.), distribute the numbers into 10 digit bins (0-9) based on the digit in that position.
- Collect the numbers back from bin 0 to 9, maintaining their order.
- Repeat the process for the next digit place until all digits have been processed.

Completion: After processing the most significant digit, the numbers in the main bin will be sorted in ascending order.

Input

The first line contains an integer N ($1 \leq N \leq 1000$) — the number of integers to sort.

The second line contains N space-separated non-negative integers (each $\leq 1,000,000$), representing the numbers to be sorted.

Output

Print the sorted list of numbers in ascending order, separated by spaces.

Sample Input

```
6
170 45 75 90 802 24
5
321 432 564 210 109
4
9 23 456 78
```

Sample Output

```
24 45 75 90 170 802
109 210 321 432 564
9 23 78 456
```

DSCP 139**Missionaries and Cannibals**

The Missionaries and Cannibals Problem is a classic river-crossing puzzle. Three missionaries and three cannibals come to a river and find a boat that holds two people. Everyone must get across the river to continue on the journey. However, if the cannibals ever outnumber the missionaries on either bank, the missionaries will be eaten.

However, there is a rule: If at any point on either side of the river, the number of cannibals exceeds the number of missionaries, the missionaries will be eaten. Your task is to determine a sequence of boat crossings that will allow all missionaries and cannibals to cross the river safely. Find a series of crossings that will get everyone safely to the other side of the river.

Input

The input consists of a single integer $N = 3$, representing the number of missionaries and cannibals on the starting side.

The boat can carry one or two people at a time.

Output

A sequence of moves, where each move is represented as:

(M, C) → Number of missionaries (M) and cannibals (C) in the boat for that trip.

The direction of travel (e.g., → for crossing to the other side and ← for returning).

The final output should confirm that all missionaries and cannibals have safely reached the other side.

Sample Input

3

Sample Output

(1,1) →

(0,1) ←

(2,0) →

(1,1) ←

(2,0) →

(0,1) ←

(1,1) →

(0,1) ←

(1,1) →

2

(1,1) →

(0,1) ←

(1,1) →

You are given two monetary values: amount charged (the total cost of a purchase) and amount given (the money paid by the customer). Your program should take two numbers as input, one that is a monetary amount charged and the other that is a monetary amount given. It should then return the number of each kind of bill and coin to give back as change for the difference between the amount given and the amount charged. The values assigned to the bills and coins can be based on the monetary system of any current or former government. Design your program so that it returns as few bills and coins as possible. Your task is to calculate the change and determine the minimum number of bills and coins needed to return to the customer. The program should follow the standard currency denominations for the chosen monetary system (e.g., US Dollars or Indian Rupees).

Input

The first input is a floating-point number representing the amount charged.

The second input is a floating-point number representing the amount given.

Output

If the amount given is less than the amount charged, print "Insufficient amount given."

Otherwise, print the minimum number of bills and coins required to make the change.

Each denomination should be displayed as:

<Denomination>: <Count>

Sample Input

Enter amount charged: 12.37

Enter amount given: 20.00

Enter amount charged: 374.50

Enter amount given: 500.00

Sample Output

Change: \$7.63

\$5 bill: 1

\$2 bill: 1

50¢ coin: 1

10¢ coin: 1

1¢ coin: 3

Change: ₹125.50

₹100 note: 1

₹20 note: 1

₹5 coin: 1

50p coin: 1

DSCP 141**Birthday Paradox**

The birthday paradox says that the probability that two people in a room will have the same birthday is more than half, provided n , the number of people in the room, and is more than 23. This property is not really a paradox, but many people find it surprising. Design a program that can test this paradox by a series of experiments on randomly generated birthdays, which test this paradox for $n = 5, 10, 15, 20, \dots, 100$.

Simulate this paradox by performing multiple trials with randomly generated birthdays for groups of sizes $n = 5, 10, 15, \dots, 100$ and determine the probability of at least two people having the same birthday for each group size.

Input

No direct user input is required.

The program will simulate groups of $n = 5, 10, 15, \dots, 100$.

Each group size should be tested with a large number of trials (e.g., 1,000 trials per group).

Output

For each group size n , print the probability that at least two people share the same birthday, formatted as a percentage.

Number of people: 5, Probability: 2.8%

Number of people: 10, Probability: 11.7%

Number of people: 15, Probability: 25.5%

Number of people: 20, Probability: 41.1%

Number of people: 23, Probability: 50.3%

Number of people: 30, Probability: 70.6%

Number of people: 40, Probability: 89.1%

Number of people: 50, Probability: 97.0%

Number of people: 100, Probability: 99.99%

Sample Input

$N = 5$

$N = 10$

$N = 15$

Sample Output

Simulating for 5 people...

Out of 1000 trials, 29 had at least two people with the same birthday.

Probability: 2.9%

Simulating for 15 people...

Out of 1000 trials, 255 had at least two people with the same birthday.

Probability: 25.5%

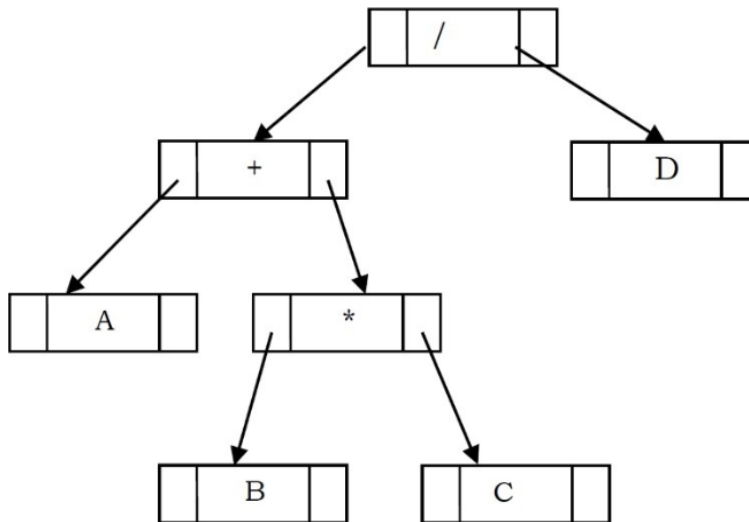
Simulating for 15 people...

Out of 1000 trials, 255 had at least two people with the same birthday.

Probability: 25.5%

N = 20	<p>Simulating for 20 people...</p> <p>Out of 1000 trials, 412 had at least two people with the same birthday.</p> <p>Probability: 41.2%</p>
N = 23	<p>Simulating for 23 people...</p> <p>Out of 1000 trials, 501 had at least two people with the same birthday.</p> <p>Probability: 50.1%</p>
N = 30	<p>Simulating for 30 people...</p> <p>Out of 1000 trials, 704 had at least two people with the same birthday.</p> <p>Probability: 70.4%</p>
N = 40	<p>Simulating for 40 people...</p> <p>Out of 1000 trials, 890 had at least two people with the same birthday.</p> <p>Probability: 89.0%</p>
N = 50	<p>Simulating for 50 people...</p> <p>Out of 1000 trials, 972 had at least two people with the same birthday.</p> <p>Probability: 97.2%</p>

A tree representing an expression is called an expression tree. In expression trees, leaf nodes are operands and non-leaf nodes are operators. That means, an expression tree is a binary tree where internal nodes are operators and leaves are operands. An expression tree consists of binary expression. But for a u-nary operator, one subtree will be empty. The figure below shows a simple expression tree for $(A + B * C) / D$.



Algorithm for Building Expression Tree from Postfix Expression

Constraints

- The expression contains single-character variables (A-Z) or numerical values.
- The operators are +, -, *, and / (binary operators).
- Parentheses determine the order of operations.
- The expression follows standard operator precedence rules:
 1. Parentheses ()
 2. Multiplication * and Division /
 3. Addition + and Subtraction -

Input

The input consists of:

1. A single string representing a mathematical expression in infix notation (e.g., "3 + (5 * 2) / 7").
2. The expression may contain whitespace, which should be ignored.

Output

The output consists of:

1. The constructed Expression Tree (represented in a structured format).

2. The evaluated result of the expression (if numerical values are provided).

Sample Input

$(A + B * C) / D$

Sample Output

(/)

/ \

(+) D

/ \

A (*)

/ \

B C

DSCP 142**Finding the kth Smallest Element in a Binary Search Tree**

A Binary Search Tree (BST) is a binary tree where the left subtree of a node contains only nodes with values less than the node's value. The right subtree of a node contains only nodes with values greater than the node's value. Both left and right subtrees are also BSTs.

Given a BST, the task is to find the k-th smallest element (where k is a positive integer). The smallest element corresponds to $k = 1$. The second smallest correspond to $k = 2$, and so on.

This problem is commonly encountered in scenarios such as:

- Database indexing (finding the k^{th} smallest record).
- Ranking systems (finding the k^{th} best/worst candidate).
- Median finding (finding the middle element efficiently).
- For a node with value X:
- Left subtree $\leq X \leq$ Right subtree
- The value of k is between 1 and the total number of nodes.
- The BST contains unique values (no duplicate elements).
- The BST structure is balanced or unbalanced.

Input

An integer k ($1 \leq k \leq N$), representing the rank of the smallest element to be found.

A Binary Search Tree (BST) containing N nodes.

The BST is given in level order format (or alternatively, it can be provided as a tree structure).

Output

A single integer representing the k-th smallest element in the BST.

Sample Input

BST:

5

/ \

3 8

/ \ \

2 4 10

k = 3

10

/ \

5 15

/ \ / \

2 8 12 20

Sample Output

4

10

/

1

k = 5

DSCP 143**Finding Floor and Ceiling in a Binary Search Tree**

If a given key is less than the key at the root of a BST then the floor of the key (the largest key in the BST less than or equal to the key) must be in the left subtree. If the key is greater than the key at the root, then the floor of the key could be in the right subtree, but only if there is a key smaller than or equal to the key in the right subtree; if not (or if the key is equal to the key at the root) then the key at the root is the floor of the key. Finding the ceiling is similar, with interchanging right and left. For example, if the sorted with input array is {1, 2, 8, 10, 10, 12, 19}, then For x = 0: floor does not exist in array, ceil = 1, For x = 1: floor = 1, ceil = 1 For x = 5: floor = 2, ceil = 8, For x = 20: floor = 19, ceil does not exist in array.

Input

A BST containing N unique elements.

A query integer x, for which we need to find the floor and ceiling.

The BST is given in level order format, or it can be provided as a tree structure.

Output

floor(x): The largest value $\leq x$ in BST (or None if not found).

ceil(x): The smallest value $\geq x$ in BST (or None if not found).

Sample Input

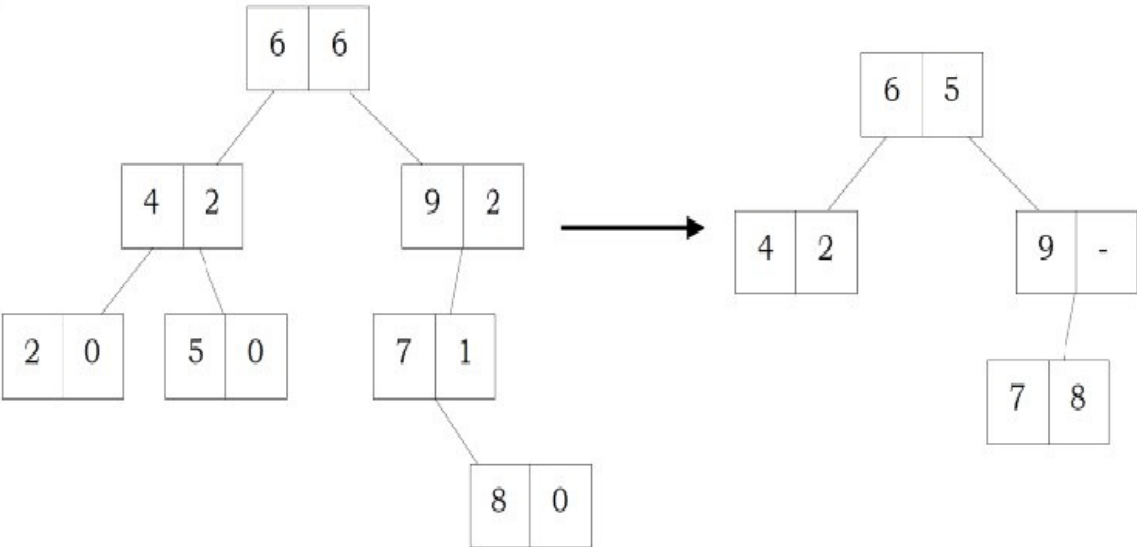
BST:

```
    10
   / \
  5   15
 / \   \
2  7  20
```

x = 6

Sample Output

Floor = 5, Ceil = 7

DSCP 144	Convert BST by Replacing Subtree Size with Inorder Previous Node Data
<p>Given a BST (applicable to AVL trees as well) where each node contains two data elements (its data and also the number of nodes in its subtrees) as shown below. Convert the tree to another BST by replacing the second data element (number of nodes in its subtrees) with previous node data in inorder traversal. Note that each node is merged with inorder previous node data. Also make sure that conversion happens in-place.</p> 	
<p>Input</p> <p>A BST with each node containing val (key) and size (number of nodes in the subtree). The BST is given in level-order format or as a tree structure.</p>	
<p>Output</p> <p>A modified BST where each node's size is replaced by its inorder predecessor's val. The tree structure remains unchanged.</p>	
<p>Sample Input</p> <p>BST:</p> <pre> (20, 5) / \ (10, 3) (30, 1) / \ (5, 1) (15, 1)</pre>	<p>Sample Output</p> <p>BST:</p> <pre> (20, 15) / \ (10, 5) (30, 20) / \ (5, None) (15, 10)</pre>

In a metropolitan city, traffic congestion is a frequent issue, especially during peak hours. The road network can be represented as a graph, where intersections are nodes and roads between them are edges. The weight of each edge represents the normal travel time between two intersections. Due to traffic congestion, the travel time on some roads increases dynamically. Given a list of roads with updated congestion-based travel times, determine the shortest possible time to travel from a given source intersection to a destination intersection, while considering these changes.

Challenges & Considerations

- Roads may have different travel times at different times of the day.
- Some roads may become completely blocked due to accidents or maintenance.
- The algorithm should be able to recompute paths efficiently when congestion levels change.

Input

An integer N representing the number of intersections.

An integer M representing the number of roads.

A list of M roads, where each road is represented as (U, V, W) , meaning there is a road from U to V with an initial travel time of W minutes.

A list of congested roads, where each road (U, V, C) means the travel time from U to V has increased by C minutes due to congestion.

Two integers S and D , representing the source intersection and the destination intersection.

Output

A single integer representing the shortest travel time from S to D , after considering congestion.

If D is unreachable, print -1.

Sample Input

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

Sample Output

```
7
```

DSCP 146**Airline Ticket Reservation System**

An airline company provides flights to various destinations. Each passenger is assigned a priority score based on multiple factors, including ticket price, frequent flyer status and business class or economy class.

The system should ensure that seats are assigned in the order of highest priority. The highest-priority passenger should always get the next available seat. If multiple passengers have the same priority, they are assigned based on their arrival time.

The airline should efficiently manage passenger bookings.

The system should process millions of passengers efficiently.

Dynamic updates should be handled as new passengers arrive.

Input

An integer N representing the number of passengers and each containing passenger name and priority score (higher values indicate higher priority).

The system will then receive multiple "book" commands, assigning seats in order of highest priority.

Output

A sequence of passenger names, printed one per line, in the order they receive seats.

Sample Input

4
Alice 95
Bob 90
Charlie 85
David 92
book
book
book
book

Sample Output

Alice
David
Bob
Charlie

DSCP 147**Scalable Ride-Sharing System**

Modern ride-sharing platforms like Uber and Lyft need to efficiently match passengers with drivers in real time. Given a set of drivers at various locations and a passenger requesting a ride, the system must quickly find the nearest available driver and provide an estimated time of arrival (ETA).

The core challenges in this problem include:

- Efficiently storing and updating driver locations as they move.
- Finding the closest available driver for a given pickup location.
- Computing ETA based on real-time traffic and road conditions.

Input

A list of drivers with their current locations and availability status:

("DriverID", Latitude, Longitude, Status)

A list of passenger requests, each containing a pickup and destination location:

("PassengerID", PickupLatitude, PickupLongitude, DestinationLatitude, DestinationLongitude)

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

Drivers:

D1, 12.9716, 77.5946, Available

D2, 12.9352, 77.6245, Available

D3, 12.9203, 77.6192, Busy

Passenger Request:

P1, 12.9304, 77.6219, 12.9856, 77.6515

Sample Output

Matched Driver: D2

Estimated Arrival Time: 8 minutes

DSCP 148**Blockchain Transaction Verification Using Merkle Trees**

Cryptocurrency systems like Bitcoin and Ethereum rely on blockchain technology to ensure transaction security. Every transaction must be verified efficiently while maintaining data integrity. A Merkle Tree is a cryptographic data structure used in blockchain for efficient and secure transaction verification. It allows users to verify whether a specific transaction exists in a block without needing to store the entire block.

The core challenges in blockchain verification include:

- Ensuring that transaction data has not been tampered with.
- Providing cryptographic proof that a transaction exists in a block.
- Verifying transactions quickly in a decentralized system.

Input

A list of transactions, each containing a unique ID, sender, receiver, amount, and hash:

("TransactionID", "Sender", "Receiver", "Amount", "Hash")

Output

The output of the Blockchain Transaction Verification Using Merkle Trees should include:

A single cryptographic hash representing the root of the Merkle tree.

A list of intermediate hashes needed to verify a specific transaction within the Merkle tree.

The proof allows verification without revealing the entire set of transactions.

A confirmation message stating whether the transaction exists in the block.

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

The answer is: 10

The answer is: 5

Sample Output

Merkle Root: 0xA3B4C56D

Transaction ID: T2

Merkle Proof: [HASH3, HASH4, HASH1]

Transaction T2 is VALID in the Merkle Tree.

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

DSCP 149**Smart Traffic Signal Management Using Graph Algorithms**

Traffic congestion in metropolitan cities is a significant challenge, leading to delays, increased fuel consumption, and higher pollution levels. A smart traffic signal management system is essential to dynamically adjust signal timings and ensure the smooth flow of traffic. One of the critical applications of such a system is emergency vehicle routing, where vehicles like ambulances, fire trucks, and police cars must reach their destination as quickly as possible. Your task is to design and implement an intelligent traffic signal system using graph algorithms, where:

The road network is represented as a weighted graph:

Nodes represent intersections. Edges represent roads, with weights indicating travel time (in seconds) under current traffic conditions.

Emergency Vehicle Routing:

Given an emergency vehicle's starting intersection and its destination, determine the shortest and fastest route using Dijkstra's Algorithm.

The algorithm should consider real-time traffic conditions (e.g., congestion levels, road closures).

Dynamic Traffic Signal Adjustment:

The system should monitor traffic density at each intersection.

If an intersection has a high density of vehicles, signal durations should be dynamically adjusted to improve traffic flow.

The adjustment should prioritize emergency vehicles, ensuring they receive green lights whenever possible.

Input

$n \rightarrow$ The number of intersections (nodes).

$m \rightarrow$ The number of roads (edges).

Next m lines describe each road with three values:

Source intersection (A-Z)

Destination intersection (A-Z)

Time taken (in seconds) under current traffic conditions

Last line: The starting and ending intersection for the emergency vehicle.

Output

The minimum time required for the emergency vehicle to reach its destination.

The optimal route (shortest path) that the vehicle should take.

Sample Input

5 6

A B 10

A C 5

B C 3

B D 8

Sample Output

Shortest Time: 13 seconds

Path: A \rightarrow C \rightarrow D \rightarrow E

C D 2

D E 6

A E

DSCP 150**Extract Minimum in a Sequence**

You are given a sequence of integers ranging from 1 to n , where each integer appears at most once in the sequence. Additionally, the sequence may contain an EXTRACT-MIN operation (denoted as 'E') at arbitrary positions. This operation removes and outputs the smallest element encountered so far in the sequence. Your task is to process the sequence efficiently and determine the output for each EXTRACT-MIN operation while maintaining the order of elements.

Key Considerations:

Each integer appears at most once in the sequence.

The sequence may contain multiple EXTRACT-MIN ('E') operations.

When 'E' appears, the smallest element so far is removed and printed.

The sequence is processed from left to right.

The output should be a list of extracted minimum elements in order of execution.

Input

The first line contains a single integer n ($1 \leq n \leq 100,000$), representing the maximum possible value in the sequence.

The second line contains a space-separated sequence consisting of integers in the range $[1, n]$ and the character 'E' (denoting EXTRACT-MIN operations).

Output

The first line contains a single integer m , representing the number of EXTRACT-MIN operations performed.

The second line contains m space-separated integers, representing the extracted minimum values in the order they were removed.

Sample Input

10
4 3 1 E 5 8 E 2 7 E 6 E

Sample Output

4
1 3 2 4