Computer Science Competition
Invitational B 2026
Programming Problem Set

I. General Notes

1. Do the problems in any order you like. They do not have to be done in order from 1 to 12.

2. All problems have a value of 60 points.

3. There is no extraneous input. All input is exactly as specified in the problem. Unless specified by the problem, integer inputs will not have leading zeros. Unless otherwise specified, your program should read to the end of file.

4. Your program should not print extraneous output. Follow the form exactly as given in the problem.

5. A penalty of 5 points will be assessed each time that an incorrect solution is submitted. This penalty will only be assessed if a solution is ultimately judged as correct.

II. Names of Problems

Number	Name
Problem 1	Asa
Problem 2	Abraham
Problem 3	Manning
Problem 4	Rey
Problem 5	Jun
Problem 6	Jonnie
Problem 7	Brady
Problem 8	Vick
Problem 9	Alicia
Problem 10	Rahul
Problem 11	Noco
Problem 12	Clea

1. Asa

Program Name: Asa.java **Input File: none**

Asa knows the judging team could always use some extra love. She decided to create a banner for you to send to the judges to show how much love you have for them.

Input:
None

Output:
The phrase "`Happy Valentine's Day to the World's Greatest CompSci Judges!!!`" rotating each word one time each to the left and repeat this pattern 3 times. It is okay to have a single trailing space at the end of each line.

Sample input:
```
None
```

Sample output:
```
Happy Valentine's Day to the World's Greatest CompSci Judges!!!
Valentine's Day to the World's Greatest CompSci Judges!!! Happy
Day to the World's Greatest CompSci Judges!!! Happy Valentine's
to the World's Greatest CompSci Judges!!! Happy Valentine's Day
the World's Greatest CompSci Judges!!! Happy Valentine's Day to
World's Greatest CompSci Judges!!! Happy Valentine's Day to the
Greatest CompSci Judges!!! Happy Valentine's Day to the World's
CompSci Judges!!! Happy Valentine's Day to the World's Greatest
Judges!!! Happy Valentine's Day to the World's Greatest CompSci
Happy Valentine's Day to the World's Greatest CompSci Judges!!!
Valentine's Day to the World's Greatest CompSci Judges!!! Happy
Day to the World's Greatest CompSci Judges!!! Happy Valentine's
to the World's Greatest CompSci Judges!!! Happy Valentine's Day
the World's Greatest CompSci Judges!!! Happy Valentine's Day to
World's Greatest CompSci Judges!!! Happy Valentine's Day to the
Greatest CompSci Judges!!! Happy Valentine's Day to the World's
CompSci Judges!!! Happy Valentine's Day to the World's Greatest
Judges!!! Happy Valentine's Day to the World's Greatest CompSci
Happy Valentine's Day to the World's Greatest CompSci Judges!!!
Valentine's Day to the World's Greatest CompSci Judges!!! Happy
Day to the World's Greatest CompSci Judges!!! Happy Valentine's
to the World's Greatest CompSci Judges!!! Happy Valentine's Day
the World's Greatest CompSci Judges!!! Happy Valentine's Day to
World's Greatest CompSci Judges!!! Happy Valentine's Day to the
Greatest CompSci Judges!!! Happy Valentine's Day to the World's
CompSci Judges!!! Happy Valentine's Day to the World's Greatest
Judges!!! Happy Valentine's Day to the World's Greatest CompSci
Happy Valentine's Day to the World's Greatest CompSci Judges!!!
```

2. Abraham

Program Name: Abraham.java Input File: abraham.dat

Abraham is running for class president, and he needs you to write a program to count how many votes he got, and if he won. He wins if he obtains more than half the possible votes. Each given line will be a vote for any of the possible candidates.

Input:
The input will begin with one integer, `n` (`0 < n <= 1000`). Each test case will consist of one string, which could be any name. If the name is "`Abraham`", then he gets the vote, otherwise he does not.

Output:
For each test case, output the integer value of the number of votes Abraham got. On the following line, if he got more than half of the possible votes, output the string "`Honest Abe`", otherwise, output "`Better find a hobby`".

Sample input:
```
5
Abraham
Abraham
Jonathan
Abraham
Juliet
```

Sample output:
```
3
Honest Abe
```

3. Manning

Program Name: Manning.java Input File: manning.dat

Manning is an intern at a currency trading firm. Manning's manager suspects that there might be arbitrage opportunities in the foreign exchange market and Manning has asked you to write a program to detect them.

An arbitrage opportunity exists when you can start with some amount of a currency, perform a sequence of exchanges, and end up with more of the original currency than you started with. For example, if 1 USD = 0.9 EUR, 1 EUR = 120 JPY, and 1 JPY = 0.0095 USD, then starting with 1 USD, you could convert to 0.9 EUR, then to 108 JPY, then back to 1.026 USD, making a profit of 2.6%.

Given a list of currencies and exchange rates between some pairs of them, determine whether an arbitrage opportunity exists.

Input:
The first input line will contain a single integer K (1 <= K <= 100) denoting the number of test cases to follow. The following K lines will begin with two integers, n and m (1 <= n <= 100, 0 <= m <= 10000), representing the number of currencies and the number of exchange rates. The next n lines each contain a string representing the name of a currency (names contain only uppercase letters and have length at most 10). The next m lines each contain two currency names and a positive real number r, indicating that 1 unit of the first currency can be exchanged for r units of the second currency. Note that this does not mean that the second currency can be exchanged for 1/r units of the first currency.

Output:
If an arbitrage opportunity exists, output "Cha Ching!!" on a single line. Otherwise, output "gg".

Sample input:
```
1
3 3
USD
EUR
JPY
USD EUR 0.9
EUR JPY 120.0
JPY USD 0.0095
```

Sample output:
```
Cha Ching!!
```

4. Rey

Program Name: Rey.java **Input File: rey.dat**

The school store is selling t-shirts and hoodies to raise money for the winter dance and to increase school spirit. Rey is setting up the invoices for each homeroom's orders. Hoodies are $30 each and t-shirts are $15 each. Available sizes are S, M, L, XL, and XXL. There is an additional $2.50 charge if the size is XL or XXL. Finally, compute the total due from each classroom.

Input:
The data file will be formatted as follows: a collection of lines starting with a teacher name (a single string, i.e., `Mr.Miller`). The next line is an integer (n) which is the number of orders from that classroom, followed by n lines with a garment and size.

Output:
Each homeroom report as shown and formatted below, with a single blank line separating each team/s information. All monetary values should be displayed to two (2) decimal places. (dollars.cents). There is to be a single blank line between each invoice.

Sample input:
```
Mr.Miller
3
Hoodie M
TShirt XL
Hoodie XL
Ms.Garcia
2
TShirt S
TShirt M
```

Sample output:
```
INVOICE FOR: Mr.Miller
--------------------------
Hoodie (M)       $  30.00
TShirt (XL)      $  17.50
Hoodie (XL)      $  32.50
--------------------------
Total Due:       $  80.00

INVOICE FOR: Ms.Garcia
--------------------------
TShirt (S)       $  15.00
TShirt (M)       $  15.00
--------------------------
Total Due:       $  30.00
```

5. Jun

Program Name: Jun.java **Input File: jun.dat**

Jun is working on a meteorological project for geosciences class. There are weather stations across the state which report the high temperature (in Fahrenheit), precipitation (in inches), and high wind speed (in miles per hour). That information has been fed into a data file which is noted alongside the date of the reading and the city where the station is installed. Jun needs to provide a summary of the data for each city which will include the average temperature, total precipitation, and the windiest day along with the wind speed. At the end, Jun also needs to report the city, date, and temperature of the coldest reading—these readings are nuanced enough so as to not have any duplicates.

Input:
The data file contains an unspecified number of comma-delimited lines of data organized in the following order: date (in the format YYYY-MM-DD), city, temperature (in Fahrenheit) as a real number, precipitation (in inches) as a real number, and wind speed (in miles per hour) as an integer.

Output:
Begin the report with the title `City Summaries` followed by a blank line. This is followed by a set of 4 lines for each city in alphabetical order—the first line will be the name of the city; the eecond line will begin with a single space followed by `Average Temp:`, the temperature to 1 decimal place followed by `F`; the third line will begin with a single space followed by `Total Precip:`, the amount of precipitation to 2 decimal places followed by `in`; the fourth line will being with a single space followed by `Windiest Day:`, the date, a space, the wind speed in parentheses `(XX mph)`. Separate each city summary with a single blank space. After the last city summary and blank line, title the final part `Coldest Reading Overall`. The final line of output will begin with a single space followed by *city* `on` *date*`:` *TT.T* `F`. (to one decimal place).

Note: All dates should be displayed as their 3-letter abbreviation followed by the number date with no leading zeros. Also, any ties in the data should be broken by the latest date

Sample input:

```
2026-01-01,Amarillo,28.4,0.10,14
2026-01-01,Dallas,35.2,0.00,8
2026-01-01,Houston,72.5,0.30,12
2026-01-02,Amarillo,25.0,0.00,10
2026-01-02,Dallas,30.1,0.05,15
2026-01-02,Houston,74.0,0.10,9
2026-01-03,Amarillo,22.5,0.12,18
2026-01-03,Dallas,29.0,0.00,20
2026-01-03,Houston,71.8,0.40,14
2026-01-04,Amarillo,27.1,0.00,11
2026-01-04,Dallas,34.0,0.02,7
2026-01-04,Houston,73.2,0.00,10
2026-01-05,Amarillo,29.6,0.00,9
2026-01-05,Dallas,36.4,0.00,6
2026-01-05,Houston,75.1,0.20,13
```

Sample output:

```
City Summaries

Amarillo
 Average Temp: 26.5 F
 Total Precip: 0.22 in
 Windiest Day: Jan 3 (18 mph)

Dallas
 Average Temp: 32.9 F
 Total Precip: 0.07 in
 Windiest Day: Jan 3 (20 mph)

Houston
 Average Temp: 73.3 F
 Total Precip: 1.00 in
 Windiest Day: Jan 3 (14 mph)

Coldest Reading Overall
 Amarillo on Jan 3: 22.5 F
```

6. Jonnie

Program Name: Jonnie.java Input File: jonnie.dat

Jonnie has begun playing N-th dimensional scrabble with his friends. You will be given a list of possible words to spell. You need to determine, based on the words already spelled, what are the possible words you can spell given the letters you have in your bar, to maximize points. Because the game is N-th dimensional, you can use any letter in any previously spelled word to spell your words, and any letter can be used an infinite number of times. Player 1 will go first, followed by 2, and so on, cycling back to one after the last player, and any player can use any word on the board for their turn. If there are no words a player can spell using the letters they have and the words already played, then they will receive 0 points for that turn, and the string "`NO POSSIBLE WORDS`" should be printed instead of their results for their turn. The first player each time will have to play a word of their choice, without any other words on the board, so they can only use their letters. You score points by playing a word using your letters, and one letter currently on the board as part of your word, and totaling the sum of the points of all the letters in the given word. The point values for each letter are as follows:

Letter	Value				
A	1	B	3	C	3
D	2	E	1	F	4
G	2	H	4	I	1
J	8	K	5	L	1
M	3	N	1	O	1
P	3	Q	10	R	1
S	1	T	1	U	1
V	4	W	4	X	8
Y	4	Z	10		

Input:
The input will begin with one integer, `n` (`0 < n <= 1000`) , p (`0 < p <= 10`), and w (`0 < w <= 1000`), denoting the number of test cases, the number of players in the game, and the number of legal words in the game, respectively. The following line will contain `w` space-separated strings made up of only uppercase characters denoting all the legal words that can be played. The following `n` lines will each contain a line of uppercase characters denoting the letters of the player whose turn it currently is.

Output:
For each test case (turn) output the string that can obtain the highest value when played, given your current 9 letters, and all the words played up until that point, followed by a space, followed by the score obtained by this word. After all test cases, output the string "`Player` ", followed by the number of the player who won (indexed starting at 1), followed by the string " `won with a score of` " followed by the score of the winning player.

Sample input:
```
3 3 5
CATTY BATTY TAGGED GABBY ACTIVE
ATYTCVEIB
GAGADEIVI
YTBAGTBCI
```

Sample output:
```
ACTIVE 11
TAGGED 9
GABBY 13
Player 3 won with a score of 13
```

7. Brady

Program Name: Brady.java Input File: brady.dat

Large language models like ChatGPT work by predicting the next word given the previous words. In this problem, you will build a simple version of this: an N-gram language model.

An N-gram model predicts the next word by looking at the previous N-1 words (called the "context"). To build one:

1. Scan through the training text, examining every sequence of N consecutive words.
2. For each sequence, record that the Nth word followed the context (the first N-1 words).
3. To predict: look up the given context and return the word that followed it most frequently.

Example with N=3 (trigram model):
Training text: "the cat sat on the cat sat down"

We record what word follows each 2-word context:
- "the cat" is followed by "sat" (2 times)
- "cat sat" is followed by "on" (1 time), "down" (1 time)
- "sat on" is followed by "the" (1 time)
- "on the" is followed by "cat" (1 time)
- "sat down" has no following word (end of text)

Query: What comes after "the cat"? Answer: "sat" (appeared 2 times)
Query: What comes after "cat sat"? Answer: "down" (tie: alphabetically first)

Input:
The first line contains a single integer K (1 <= K <= 100), the number of test cases. Each test case begins with a line containing two integers N and Q (2 <= N <= 5, 1 <= Q <= 100), where N is the N-gram size and Q is the number of queries. The next line contains the training text: a sequence of lowercase words separated by single spaces (1 to 10000 words, each word 1-20 characters). The following Q lines each contain a query: exactly N-1 lowercase words separated by single spaces. Note that N will never be greater than the number of words for a given test case.

Output:
For each query, output the most likely next word on a single line. If multiple words are tied for most frequent, output the one that comes first alphabetically. If the context never appeared in the training text, output "unknown".

Sample input:

```
3
3 4
the cat sat on the cat sat down
the cat
cat sat
sat down
on the
2 3
to be or not to be
to
not
be
2 2
the dog the cat the bird
the
missing
```

Sample output:

```
sat
down
unknown
cat
be
to
or
bird
unknown
```

8. Vick

Program Name: Vick.java Input File: vick.dat

You have been hired to build the core matching engine for a new stock exchange. A matching engine maintains an order book and matches incoming orders against resting orders according to price-time priority.

Order Book Basics:
The order book has two sides: bids (buy orders) and asks (sell orders). Each limit order specifies a side, price, and quantity. Orders rest in the book until they are filled or canceled.

Matching Rules:
When a new order arrives, it attempts to match against the opposite side of the book:
- A BUY order matches against asks with price <= the buy order's price, starting with the lowest ask price
- A SELL order matches against bids with price >= the sell order's price, starting with the highest bid price
- At each price level, orders are matched in time priority (earlier orders fill first)
- Trades execute at the resting order's price (the order already in the book)
- If the incoming order is not fully filled, any remaining quantity rests in the book

Order Types:
LIMIT <id> <side> <price> <qty> - A limit order that can match or rest in the book
CANCEL <id> - Remove the order with the given ID from the book (if it exists and has remaining quantity)
ICEBERG <id> <side> <price> <qty> <visible> - An iceberg order with hidden quantity. Only <visible> shares are shown at a time. When the visible portion fills completely, it refills from the hidden quantity and goes to the back of the time priority queue at that price level.

Example:
LIMIT 1 SELL 100 10 -> Order 1 rests: selling 10 @ 100
LIMIT 2 SELL 101 5 -> Order 2 rests: selling 5 @ 101
LIMIT 3 BUY 100 7 -> Matches 7 from Order 1 @ 100. Trade: 7 @ 100. Order 1 has 3 left.
LIMIT 4 BUY 101 8 -> Matches 3 from Order 1 @ 100, then 5 from Order 2 @ 101. Trades: 3 @ 100, 5 @ 101.

Input:
The first line contains K (1 <= K <= 10), the number of test cases. Each test case begins with N (1 <= N <= 10000), the number of operations. The following N lines each contain one operation. Order IDs are unique positive integers up to 1000000. Prices are positive integers up to 1000000. Quantities are positive integers up to 10000. For ICEBERG orders, visible quantity is always <= total quantity.

Output:
For each test case, output the trades that occurred in order, one per line, in the format:
TRADE <buy_order_id> <sell_order_id> <price> <quantity>
After all trades for a test case, output a blank line.

Sample input:

```
3
4
LIMIT 1 SELL 100 10
LIMIT 2 SELL 101 5
LIMIT 3 BUY 100 7
LIMIT 4 BUY 101 8
5
LIMIT 1 BUY 50 100
LIMIT 2 SELL 50 30
CANCEL 1
LIMIT 3 SELL 50 50
LIMIT 4 BUY 50 10
```

Sample input cont. next column

Sample input continued

```
6
ICEBERG 1 SELL 100 20 5
LIMIT 2 BUY 100 7
LIMIT 3 BUY 100 3
LIMIT 4 BUY 100 6
LIMIT 5 SELL 100 2
LIMIT 6 BUY 100 10
```

Sample output:

```
TRADE 3 1 100 7
TRADE 4 1 100 3
TRADE 4 2 101 5

TRADE 1 2 50 30
TRADE 4 3 50 10

TRADE 2 1 100 5
TRADE 2 1 100 2
TRADE 3 1 100 3
TRADE 4 1 100 5
TRADE 4 1 100 1
TRADE 6 1 100 4
TRADE 6 5 100 2
```

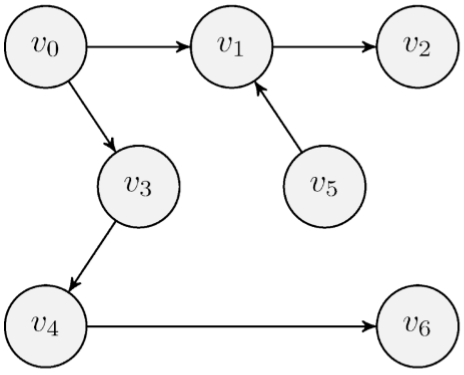
9. Alicia

Program Name: Alicia.java **Input File: alicia.dat**

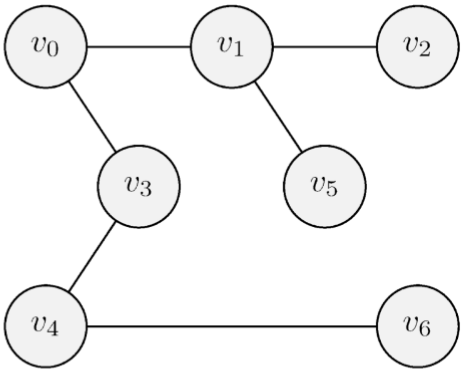
Alicia is currently learning about the formal definitions of Graphs and Trees. While she is easily able to visually classify a set of vertices and edges as either a graph or a tree, Alicia is struggling in being able to formally prove which classification it is. Here is what Alicia has learned so far:

- A graph $G = (V, E)$ is formally defined to be composed of a set of vertices $V = \{v_1, v_2, \dots, v_n\}$ and edges $E = \{e_1, e_2, \dots, e_m\}$ where edge $e_i = (u, v)$, $u, v \in V$ indicates that a relationship exists between vertex u and vertex v.
  - A graph G is said to be "undirected" if for all $e_i = (u, v) \in E$, then $e_j = (v, u) \in E$ (i.e., if a relationship exists between vertex u and vertex v, then a relationship also exists between vertex v and vertex u).
    - A graph G is said to be "directed" if it is not undirected.
  - A graph G is said to be "cyclic" if there exists some subset of edges $\{e_1, e_2, \dots, e_{m'}\} = \{(v_i, v_j), (v_j, v_k), \dots, (v_l, v_i)\} \subseteq E$. (i.e., there exists some path of the form $v_i \to v_j \to v_k \rightsquigarrow v_i$). Note that at least three vertices must be included in such a path for it to be considered cyclic.
    - A graph G is said to be "acyclic" if it is not cyclic.
  - A graph G is said to be "connected" if every pair of vertices v_i, v_j is connected in G. A pair of vertices v_i, v_j is said to be connected if there exists some simple path from v_i to v_j (denoted $v_i \rightsquigarrow v_j$) via some subset of edges in E.
    - An *undirected* graph G is said to be "disconnected" if it is not connected.
    - A *directed* graph G is said to be "weakly connected" if by replacing all directed edges in E with undirected ones (call this new set E'), then graph $G' = (V, E')$ is connected.
    - A *directed* graph G is said to be "strongly connected" if for every pair of vertices $v_i, v_j \in V$, then *both* $v_i \rightsquigarrow v_j$ and $v_j \rightsquigarrow v_i$ hold. That is, a graph is strongly connected if it is a connected directed graph.
    - A *directed* graph G is said to be "unilaterally connected" if, for every pair of vertices $v_i, v_j \in V$, then *either* $v_i \rightsquigarrow v_j$ and/or $v_j \rightsquigarrow v_i$ hold.
    - A *directed* graph G is said to be "disconnected" if it is not weakly, strongly, or unilaterally connected.
- A tree $T = (V, E)$ is formally defined as a connected, undirected, acyclic graph.

Thankfully, Alicia has a sage, oracle friend who has already determined the connectedness of each graph ahead of time. With this, help Alicia classify a given set of vertices and edges as either a graph or a tree, being as specific as possible.



Directed, Weakly Connected, Acyclic Graph



Tree (Undirected, Connected, Acyclic Graph)

Input:
The first line of input will consist of a single integer T, $1 \leq T \leq 15$, denoting the number of test cases to follow. The next T test cases will each consist of a single line denoting the serialized string of the object consisting of the set of vertices and edges of the graph. Each graph object can be described by two integers n and m, $2 \leq n \leq 10^3$,
$1 \leq m \leq n \cdot (n-1)$ denoting the number of vertices and edges. You may assume that graph G has $V = \{v_0, v_1, \ldots, v_{n-1}\}$ for the specified value of n, and each edge $e = (v_i, v_j) \in E$ will have $0 \leq i, j < n$, $i \neq j$ hold. You may also assume that the `connectedness` field has been filled out with the corresponding `Connectedness` enum value.

Output:
For each test case, on its own line, output a single string in the form of "`<directedness>, <connectivity>, <cyclic> graph`". If the graph is also a tree, start the line off with the word "`Tree `", and include the other output for this line in parentheses.

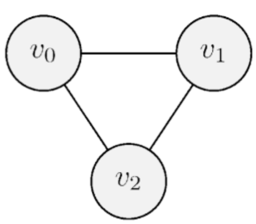
Sample input: (*lines truncated at …*)

```
5
rO0ABXNyAAVHcmFwaAAAAAAAAAABAgAGSQABbUkAAW5MAA1jb25uZWN0ZWRuZXNzdAAPTENvbm5lY3R...
rO0ABXNyAAVHcmFwaAAAAAAAAAABAgAGSQABbUkAAW5MAA1jb25uZWN0ZWRuZXNzdAAPTENvbm5lY3R...
rO0ABXNyAAVHcmFwaAAAAAAAAAABAgAGSQABbUkAAW5MAA1jb25uZWN0ZWRuZXNzdAAPTENvbm5lY3R...
rO0ABXNyAAVHcmFwaAAAAAAAAAABAgAGSQABbUkAAW5MAA1jb25uZWN0ZWRuZXNzdAAPTENvbm5lY3R...
rO0ABXNyAAVHcmFwaAAAAAAAAAABAgAGSQABbUkAAW5MAA1jb25uZWN0ZWRuZXNzdAAPTENvbm5lY3R...
```

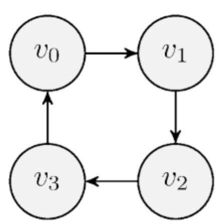
Sample output:

```
Directed, Weakly Connected, Acyclic Graph
Tree (Undirected, Connected, Acyclic Graph)
Undirected, Connected, Cyclic Graph
Directed, Strongly Connected, Cyclic Graph
Directed, Unilaterally Connected, Acyclic Graph
```

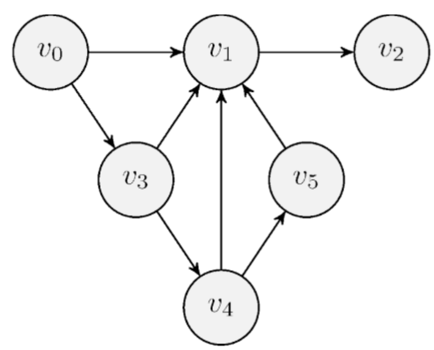
Explanation:
The first test case corresponds to the graph in the figure on the left, and the second test case corresponds to the graph in the figure on the right. The three additional test cases have been illustrated below.



Undirected, Connected, Cyclic Graph



Directed, Strongly Connected, Cyclic Graph



Directed, Unilaterally Connected, Acyclic Graph

Important Notes:
There are many factors that affect what is output by the `marshall()` method that don't pertain to what the object logically represents. Because of this, you should avoid changing the names of existing methods/fields, creating additional `public`/`protected` methods (`private` is fine) and avoid creating any new fields as this affects what data is associated with the encoding of the object. You may set the `toString()` method to simply return the empty string if you do not wish to make use of it in your implementation.

Starter Class:

```
import java.io.Serializable;
import java.util.HashMap;
import java.util.HashSet;

class Graph implements Serializable {
    public static final long serialVersionUID = 1L;

    private HashMap<Integer, HashSet<Integer>> g;
    private int n, m;

    public Graph(HashMap<Integer, HashSet<Integer>> g, int n, int m) {
        this.g = g;
        this.n = n;
        this.m = m;
    }

    private Directedness directedness;

    public Directedness getDirectedness() {
        // TODO: Implement
    }

    private Cyclicity cyclicity;

    public Cyclicity getCyclicity() {
        // TODO: Implement
    }

    protected Connectedness connectedness;

    @Override
    public String toString() {
        // TODO: Implement
    }
}
```

Serialization Code:

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.Base64;

class Serializer {
    public static String marshall(Serializable o) throws IOException {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(baos);
        oos.writeObject(o);
        String encodedData = Base64.getEncoder().encodeToString(baos.toByteArray());
        baos.close();
        oos.close();
        return encodedData;
    }

    public static Object unmarshall(String s) throws ClassNotFoundException, IOException {
        byte[] data = Base64.getDecoder().decode(s);
        ByteArrayInputStream bais = new ByteArrayInputStream(data);
        ObjectInputStream ois = new ObjectInputStream(bais);
        Object o  = ois.readObject();
        bais.close();
        ois.close();
        return o;
    }
}
```

Sample Usage:

```
String byteStream = file.readLine();
Object o = null;
try {
    o = Serializer.unmarshall(byteStream);
} catch (Exception e) {
    System.out.println("Come on problem setter, get your classes straight!");
    return;
}

Graph g = null;
if (o instanceof Graph) {
    g = Graph.class.cast(o);
} else {
    System.out.println("Come on problem setter, get your classes straight!");
    return;
}
```

10. Rahul

Program Name: Rahul.java Input File: rahul.dat

Rahul has recently becomes obsessed with number sequences, increasing, repeating, decreasing, etc. He now is looking into zigzag sequences, where each values is greater than or less than the last, in alternating order (1 < 2 > 1 < 2 > 1 and so on). You need to write a program to find the longest possible zigzag subsequence of the given list of integers, given that the subsequence does not need to be consecutive.

Input:
The input will begin with one integer, `n` (`0 < n <= 1000`). Each test case will consist of a line of an unspecified number of space separated integers denoting the list to check for zigzag subsequences.

Output:
For each test case, output the maximum length of a zigzag subsequence within the given integers. This value will always be at least 1.

Sample input:
```
3
1 5 4
1 4 5
10 22 9 33 49 50 31 60
```

Sample output:
```
3
2
6
```

11. Noco

Program Name: Noco.java **Input File: noco.dat**

Noco is an incredibly whimsical person and has recently been appointed as the head of the Christmas Light Strand Satisfaction division of his state's government. As head of this department, Noco is responsible for ensuring that each municipality that he has jurisdiction over is as satisfied as possible with the strand of Christmas Lights that they receive each holiday season. While this might sound like a simple task, each municipality tends to have its own set of preferences when it comes to how the lights are arranged.

Each municipality submits a preferred strand pattern, represented as a string consisting only of the characters: `R` – denotes a red light, `G` – denotes a green light, `W` – denotes a white light. To fairly compare two municipalities' preferences, Noco aligns their strands by possibly inserting gaps (−) into either strand so that both resulting sequences have the same length. He then evaluates the alignment using the following Festive Harmony Score:

$$FHS = (\#\text{matches}) \times a - (\#\text{mismatches}) \times b - (\#\text{gaps}) \times c$$

Where a match is when two aligned characters are identical, a mismatch is when two aligned characters differ, a gap is when a character is aligned with − in the other strand, a is the score awarded for each match, b is the penalty for each mismatch, and c is the penalty for each gap. Your task is to help Noco compute the maximum possible harmony score between two given strands.

Input:
The first line of input will denote a single integer T, $1 \leq T \leq 20$, denoting the number of test cases to follow. Each test case will consist of three lines of input. The first line will consist of five single-space separated integers n, m, a, b, and c denoting the length of the first strand of lights, length of the second strand of lights, award for each match, penalty for each mismatch, and penalty for each gap, respectively. It should be noted that $1 \leq n, m \leq 2 \cdot 10^3$ and $1 \leq a, b, c \leq 10$ hold. The second line denotes the first strand of Christmas lights, and the third denotes the second strand. It should be noted that each strand is guaranteed to only consist of the characters `R`, `G`, and `W`.

Output:
For each test case, on its own line, print the maximum value for FHS given the two strands denoted by the test case.

Sample input:
```
4
3 3 2 1 2
RGW
RWW
3 3 2 6 2
RGW
RWW
6 5 1 2 3
RGGGWR
RGGWR
13 14 10 2 3
RGGWRRRWRWWRG
WRRGRRWWGRWWRR
```

Sample output:
```
3
0
2
75
```

Explanation:
In the first example, the two sequences remain unchanged as creating a gap is more costly than simply allowing the mismatch. In the second example, one optimal alignment for the two sequences are `R-GW` and `RW-W`, respectively (alternatively `RG-W` and `R-WW`, respectively). In the third case, the optimal alignment for the two sequences are `RGGGWR` and `RGG-WR`, respectively. Lastly, one optimal alignment for the last case would be the following:

```
-RGGWRRRW-RWWRG
WRRG-RRWWGRWWRR
```

12. Clea

Program Name: Clea.java **Input File: clea.dat**

Clea is the Chief Archivist of the Royal Genealogy Office. Her job is to maintain the official family records of every noble in the kingdom. Unfortunately, centuries of wars, marriages, and political cover-ups have left the records in chaos. Two independent genealogical registries were kept over time – The Silver Registry and The Obsidian Registry – each have attempted to track every person's *primary* recorded parent.

Both registries describe the same set of people, but they often disagree on who someone's primary parent was. Each person may have at most one primary parent, and no one can be their own ancestor. Clea must now create a single official family record that is:

- Biologically valid – every person has at most one parent.
- Historically consistent – no ancestry cycles are allowed.
- Registry consistent – each relationship in the new official record must originate from either the Silver or Obsidian Registries (or both).

As faithfully as possible, the final record should preserve as many parent relationships from the two registries as possible. Each registry records primary-parent–child relationships. For every ordered pair (p, i) where $p \neq 0$, the registry claims that p is the primary biological parent of i. Clea assigns a score to each such claim:

- If a relationship (p, i) appears in either registry, it is worth 1 point.
- If it appears in both registries, it is worth 2 points.

When $p = 0$, we assume that the registry is unaware of the parent of i, and such a relationship bears 0 points. Clea must choose a set of primary-parent-child relationships that will form the final official genealogy. Her goal is to maximize the total score of the chosen relationships, subject to the biological, historical, and registry constraints.

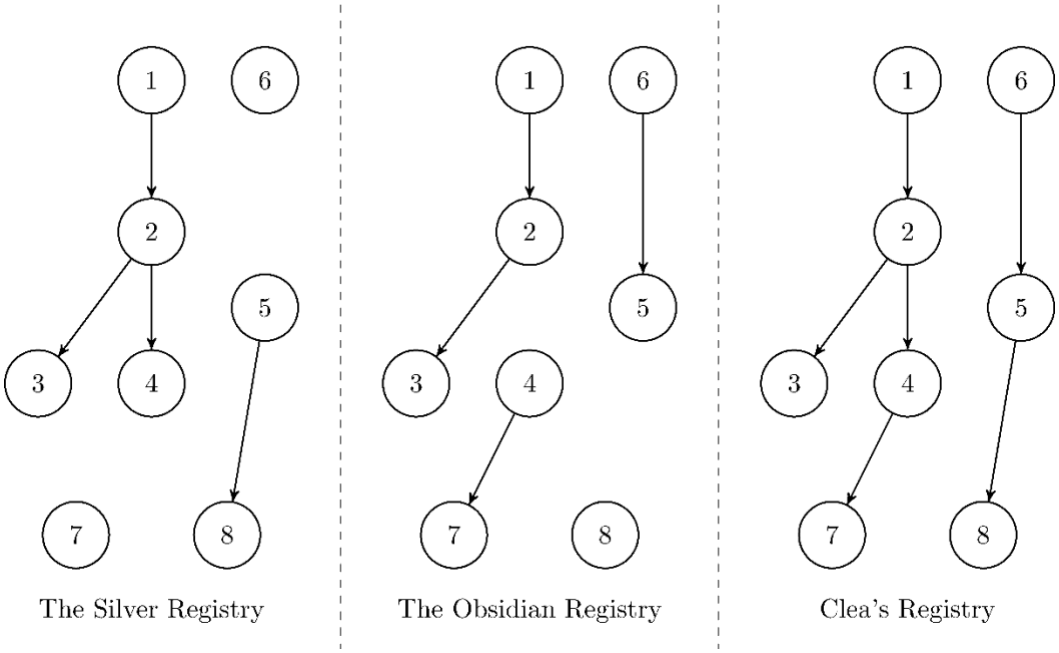
Input:
The first line of input will consist of a single integer T, $1 \leq T \leq 20$, denoting the number of test cases to follow. Each test case will consist of three lines. The first line will denote a single integer n, $1 \leq n \leq 2 \cdot 10^5$, denoting the number of people that the two registries are tracking. Each person is numbered 1 through n. The next line will consist of n single-space separated integers $p_1, p_2, \ldots, p_n$, denoting that person i has primary parent p_i in the Silver Registry. In the same fashion, the third line will denote the Obsidian Registry's contents. It is guaranteed that both $1 \leq i \leq n$ and $0 \leq p_i \leq n$ hold for both registries.

Output:
For each test case, on its own line, output the total score of Clea's maximized scoring official record.

Sample input:
```
2
8
0 1 2 2 0 0 0 5
0 1 2 0 6 0 4 0
7
0 1 1 2 0 5 0
3 0 2 0 1 5 5
```

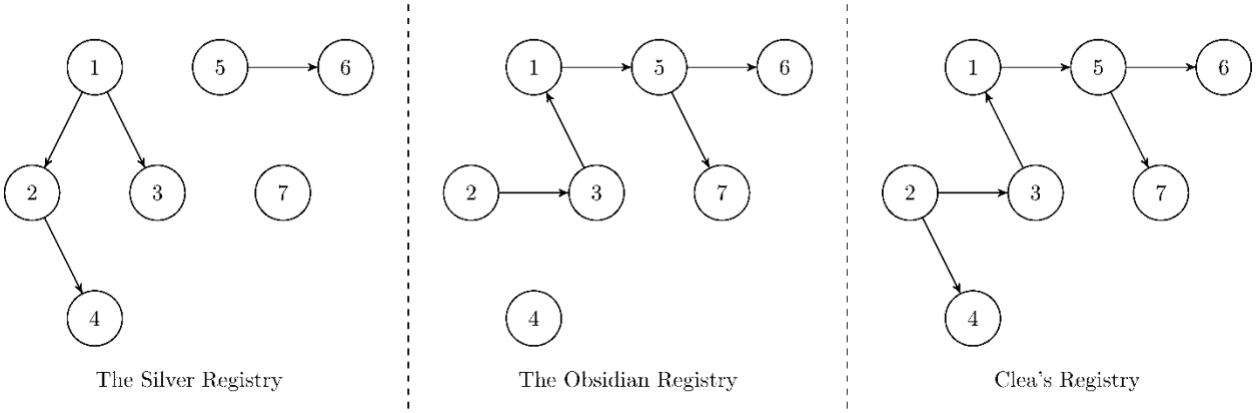
Sample output:
```
8
7
```

Explanation:
The following is a visualization of (i) the Silver Registry, (ii) the Obsidian Registry, and (iii) Clea's master Registry for the first test case in the sample input/output:



Since Clea's registry has 2 edges that exist in both registries, and 4 edges that are unique to either the Silver Registry, or the Obsidian Registry, then that equals $2(2) + 4(1) = 8$ points, which is the maximum points possible given the two registries provided. Clea's Registry also maintains the three rules described above regarding violations/consistencies in forming the merged family trees.

The following is a visualization again of the three registries for the second test case in the sample input/output:



Since Clea's registry has 1 edge that exists in both registries, and 5 edges that are unique to either the Silver Registry, or the Obsidian Registry, then that equals $1(2) + 5(1) = 7$ points, which is the maximum points possible given the two registries provided. Note that there are multiple versions of the registry that Clea could have created which equated these 7 maximal points; however, more than 7 points is not possible due to requiring that cycles are not formed.