The 2011 ACM ASIA Programming Contest Kanpur Site

Sponsored by IBM

Hosted by IIT Kanpur

8th December 2011
You get 19 Pages
10 Problems
&
300 Minutes

Rules for ACM-ICPC 2011 Asia Regional Kanpur Site:

1. Solutions to problems submitted for judging are called runs. Each run is judged as accepted or rejected by the judge, and the team is notified of the results. Submitted codes should not contain team or University name and the file name should not have any white space.

2. The public rank list will not be updated in the last one hour. However, notification of accepted/ rejected runs will **NOT** be suspended at the last one hour of the contest time.

3. A contestant may submit a clarification request to judges. If the judges agree that an ambiguity or error exists, a clarification will be issued to all contestants.

4. Contestants are not to converse with anyone except members of their team and personnel designated by the organizing committee while seated at the team desk. **But they cannot even talk with their team members when they are walking around the contest floor to have food or any other purpose.** Systems support staff may advise contestants on system-related problems such as explaining system error messages.

5. While the contest is scheduled for a particular time length (five hours), the contest director has the authority to alter the length of the contest in the event of unforeseen difficulties. Should the contest duration be altered, every attempt will be made to notify contestants in a timely and uniform manner.

6. **A team may be disqualified by the Contest Director** for any activity that jeopardizes the contest such as dislodging extension cords, unauthorized modification of contest materials, distracting behavior or communicating with other teams.

7. Team can bring up to 25 pages of printed materials with them. But they are not allowed to bring calculators or any machine-readable devices like mobile phone, CD, DVD, Pen-drive, IPOD, MP3/MP4 players, floppy disks etc. Blank papers will be provided, if needed.

8. With the help of the volunteers and external judges, the contestants can have printouts of their codes for debugging purposes. **Passing of printed codes to other teams is strictly prohibited.**

9. **The decision of the judges is final.**

10. **Teams should inform the volunteers if they do not get reply from the judges within 10 minutes of submission. Volunteers will inform the External Judges and the external judge will take further action. Teams should also notify the volunteers if they cannot log in into the PC^2 system. This sort of complains will not be entertained after the contest.**

11. Ten problems will be posed. So far as possible, problems will avoid dependence on detailed knowledge of a particular applications area or particular contest language. All problems require you to read test data from the file as specified in the problem and write results to the standard output.

12. You can use any of the standard library functions that your chosen programming language provides. In addition, you can use the math library in C/C++. You cannot use any other library that requires an extra flag to be passed to the compiler command. If you do this, the judges will probably find a code "compilation error" in your program.

13. Your program is not permitted to invoke any external programs. For example, you cannot use in C the system call ("grep xyz abc") to determine whether the file abc contains an occurrence of the string xyz. *Violation of this rule may lead to disqualification from the contest.*

14. Output must correspond exactly to the provided sample output format, including (mis)spelling and spacing. Multiple spaces will not be used in any of the judges' output, except where explicitly stated.

15. Your solution to any problem should be submitted for judging using the PC^2 software only. Once you have submitted the solution, it will reach the judges. The time it takes for your problem to be judged will depend, among other things, on how busy the judges are. Once your submission has been judged, you will receive a message through PC^2 indicating the judgement. The judgement may be "Yes", meaning that your submission was judged to be correct. Otherwise you will get a message indicating the problem with your program. For example, the message may be "Incorrect Output", "Output Format Error", "Compilation Error", "Runtime Error", "Run Time Limit Exceeded" etc.

16. **Programming style is not considered in this contest. You are free to code in whatever style you prefer. Documentation is not required. The judges will only test whether the input-output behavior of your program is correct or not.**

Problem A
Strongly Connected Chemicals

Input: Standard Input
Output: Standard Output

One type of chemical compounds is the ionic compound, because it is composed of positively charged ions (cations) and negatively charged ions (anions) in such a way that the compound as a whole is electrically neutral. Cations always attract anions and anions always attract cations.

Now you are given **m** cations and **n** anions, and their connectivity information about which cation attracts which anion. Your task is to find a sub group from the cations and anions where there is at least one cation and at least one anion, such that all the cations in the group attract all the anions in the group. We call such a group as '**Strongly Connected Chemicals**'. As there can be many such groups, we want to find a group which has the highest cardinality. Cardinality means the number of members (cation and anion) in the group.

Input

Input starts with an integer **T (≤ 100)**, denoting the number of test cases.

Each case starts with a line containing two integers: **m** and **n (1 ≤ m, n ≤ 50)**. Each of the next m lines contains **n** characters (without space), each either '**0**' or '**1**'. If the $\mathbf{j}^{th}$ character in the $\mathbf{i}^{th}$ line is **1**, that means the $\mathbf{i}^{th}$ cation attracts the $\mathbf{j}^{th}$ anion, otherwise it doesn't.

Output

For each case, print the case number and the maximum possible size of the strongly connected chemical group.

Sample Input	**Output for Sample Input**
2 3 4 1100 1110 0011 2 2 10 00	Case 1: 4 Case 2: 2

Problem B
Confusion in the Problem Set

Input: Standard Input
Output: Standard Output

A small confusion in a problem set may ruin the whole contest. So most of the problem setters try their best to remove any kind of ambiguity from the set. But sometimes it is not that important. For example the mock of last contest. As it was mock contest so we were not that serious with the set. We printed two problems, problem A in Page 1 and Problem B in Page 2. Then we remembered we have to give rule of the contest too. So we printed the rule page. But we did not notice that the rule page was printed with Page 2. We were stapling 3 pages together. First rule page, then Problem A and at the last Problem B. So page numbers were, 2, 1 and 2. This looked odd. But we already printed all the pages and if we want to fix the issue we had no other way but print all the three pages. One among us suggested an explanation- "Well, first 2 means there are 2 pages after this page. 1 also means there is 1 page after this page. But the last 2 means there are 2 pages before this page." Interesting observation indeed. So we came up with a rule which is, page numberings of all the n pages are valid, if the page number at a page denotes number of pages before this page or number of page after this page.

So with this rule, {3, 1, 2, 0} is valid but {3, 3, 1, 3} is not valid.

Input

First line of the input file contains number of tests cases **T (<60)**.

Then **T** test cases follow. First line of each test case contains a positive number **n (<=10^4)** number of pages in the problem set. Then there are n space separated numbers in the following line each of which ranges from **0 to 10^6.**

Output

For each test case output the test case number. Then output **"yes"** (without the quotes) if the pages can be shuffled somehow so that the pages numbering is valid. Otherwise output **"no"**.

For the exact format of the output please follow the sample.

Sample Input	**Output for Sample Input**
2 4 0 3 1 2 4 1 3 3 3	Case 1: yes Case 2: no

Explanation of the samples:

1. The pages can be shuffled in several ways so that the page numbering is valid. One of the valid shuffles are 3, 1, 2, 0.
2. There is no valid way to shuffle these.

Problem C
LCM Extreme

Input: Standard Input
Output: Standard Output

Find the result of the following code:

```
unsigned long long allPairLcm(int n){
    unsigned long long res = 0;
    for( int i = 1; i<=n;i++)
        for(int j=i+1;j<=n;j++)
           res += lcm(i, j);// lcm means least common multiple
    return res;
}
```

A straight forward implementation of the code may time out.

Input

Input starts with an integer **T (≤ 25000)**, denoting the number of test cases.

Each case starts with a line containing an integer **n (1 ≤ n ≤ $5*10^6$)**.

Output

For each case, print the case number and the value returned by the function **'allPairLcm(n)'**. As the result can be large, we want the result modulo **2^{64}**.

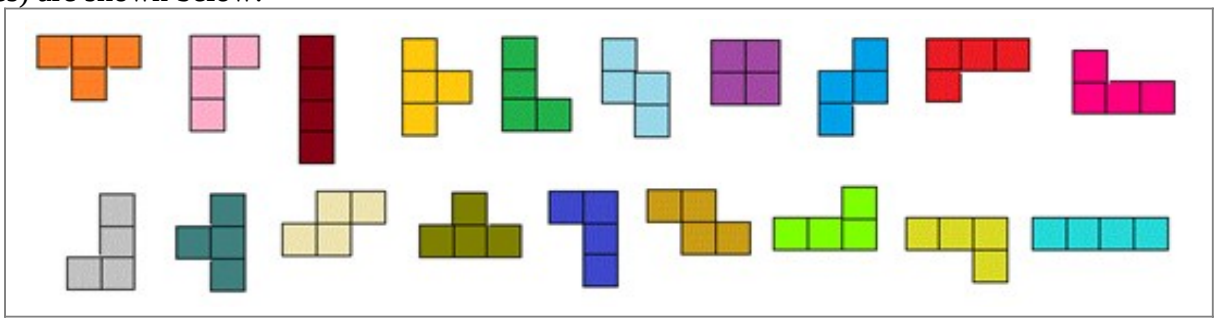
Sample Input	**Output for Sample Input**
4 2 10 13 100000	Case 1: 2 Case 2: 1036 Case 3: 3111 Case 4: 9134672774499923824

Problem D
Tetromino

Input: Standard Input
Output: Standard Output

Dexter wants to completely cover a **4 x N** board using **N** tetrominoes. Every cell in the grid has to be covered by exactly one piece and no piece is allowed to go outside the board. And no piece can be rotated or flipped.

A tetromino is a connected set of 4 tiles. All 19 possible tetrominoes that can be used (any number of times) are shown below:



Assume that all the 19 pieces have different colors. Now your task is to find the total number of ways Dexter can cover the board. Two board configurations are different if a cell can be found where the colors are different.

Input

Input starts with an integer **T (≤ 20)**, denoting the number of test cases.

Each case starts with a line containing an integer **N (1 ≤ N ≤ 10^9)**.

Output

For each case, print the case number and the number of ways to fill the board modulo **1000 000 007**.

Sample Input	**Output for Sample Input**
4 1 2 3 12	Case 1: 1 Case 2: 4 Case 3: 23 Case 4: 15747348

Problem E
Blade and Sword

Input: Standard Input
Output: Standard Output

You may have played the game 'Blade and Sword', it's an action game. However, in this problem we are actually solving one stage of the game.

For simplicity, assume that the game stage can be modeled as a **2D** grid which has **m** rows and **n** columns. The cells are categorized as follows:

1) **'.'** represents an empty space. The player can move through it.
2) **'#'** represents wall, and the player cannot move through it. You can assume that the boundaries of the grid will be walls.
3) **'*'** means a teleporting cell, once the player moves into this cell, he must choose any other teleporting cell where he will be taken to. But if he cannot find a desired teleporting cell, he will die. However, after moving to the desired teleporting cell, he can either move to an adjacent cell, or he can teleport again using the same procedure.
4) **'P'** means the position of the player and there will be exactly one cell containing '**P**'.
5) **'D'** means the destination cell and there will be exactly one cell containing '**D**'.

Now you are given a stage and the player starts moving. It takes one unit of time for the player to move to any adjacent cell from his current position. Two cells are adjacent if they share a side. One unit of time is needed for the teleporting service; that means taking the player from one teleporting cell to any other teleporting cell. Your task is to find the minimum possible time unit required for the player to reach the destination cell.

Input

Input starts with an integer **T (≤ 100)**, denoting the number of test cases.

Each case starts with a line containing two integers: **m** and **n (3 ≤ m, n ≤ 200)**. Each of the next **m** lines contains **n** characters denoting the stage. The given stage follows the restrictions stated above.

Output

For each case, print the case number and the minimum required time. If it's impossible for the player to reach the destination cell, print '**impossible**'. See the samples for details.

Sample Input	Output for Sample Input
2 4 10 ########## #.P..#*..# #*......D# ########## 3 9 ######### #P.#..D.# #########	Case 1: 6 Case 2: impossible

Problem F
The Falling Circle

Input: Standard Input
Output: Standard Output

The little Jerry is now developing a game. You know developing game is a complex thing as there are many challenges involved. For example little Jerry is now stuck with some geometric pattern and asks for your help. To help him you do not need to know every detail of the game, only information of the particular frame should suffice. Here it is:

Two circles are attached with the wall. They are fixed. A line is attached to the circles in such a way that it touches both the circles and each of the touching points has lower Y coordinate than that of the corresponding center of the circle. Now another circle is dropped on the set up from above (higher Y coordinate). The circle will fall along the Y axis with a constant velocity of 1 unit per second until it touches the line. When it touches the line, it starts to rotate along the line towards the circle that has lower Y coordinate touching point with a constant angular velocity of 1 revolution per second. When it touches the circle at the end, it stops. If both the touching points have same Y coordinate, i.e. the line is parallel to the X axis, then the falling circle stops as soon as it touches the line. Now given the setup, you need to find the time after which the falling circle will stop.

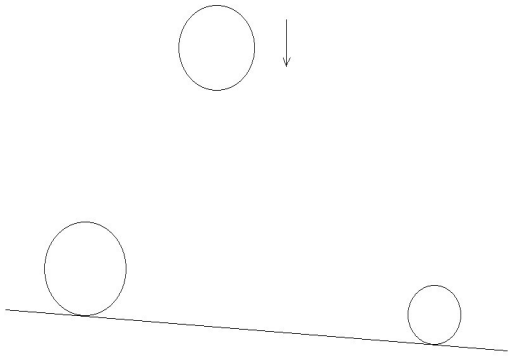


Figure 1: Initial Setup, the circle starts falling

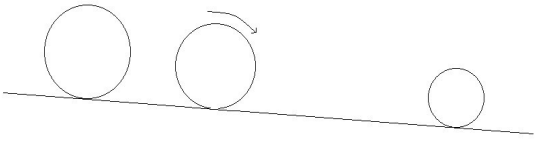


Figure 2: The circle touches the line and starts to rotate

Figure 3: The circle touches the circle in the end and stops

Input

The first line of the input will denote the number of cases **T (T ≤ 10000)**. Each of the test cases will contain 3 lines describing 3 circles. The first two lines will give information of the circles at the ends and the 3rd line will describe the circle that will fall. Each of the circles are described with 3 integers **x**, **y** and **r (-100000 ≤ x, y ≤ 100000; 0 < r ≤ 100)**. You are assured that the falling circle will always touch the line first and will drop between the fixed circles. Also, note that the falling circle will not bounce, no matter what its height is.

Output

For each case output the case number and the time required in seconds. Absolute difference less than **1e-6** will be considered correct.

Sample Input	**Output for Sample Input**
2 0 0 5 100 1 6 50 60 10 0 0 5 100 1 8 50 60 10	Case 1: 55.000000 Case 2: 56.511165

Problem G
My Visit to Spring Secret

Input: Standard Input
Output: Standard Output

Last year in ICPC Kanpur, we introduced you to the story of well-known programmers' mythology Spring Secret. If you never heard of this, don't bother just yet. It is not really important for the purpose of solving this problem. This year, powered by an assumption that this mysteriously unknown object is a place, I myself have decided to visit it. But being the puzzling mythology that it is, I don't really know if I will be able to recognize a spring secret when I see one. Your task is to help me there.

In the context of this problem, lets assume I have visited a series of places. Each city has a single capital English letter label. This label is not any unique identifier so multiple places may have the same label, don't get confused about that. Now we somehow know that if two consecutive places that I have visited has a label '**S**' then the first place is a Spring Secret. Whereas if a place with label '**S**' is followed by a place with any other label than '**S**', then the first place is an Winter Obvious. Please report the count of Spring Secret and Winter Obvious for each test case.

Input

There will be at most **100** test cases in the input file. Each case is contained in a line by itself that contains a series of capital English letters together. You can safely assume that there will be no blank space, punctuation marks or any invalid character in a test case. Each letter describes the label of a place as described above. The places are given in chronological order of my visit. You can also assume that I shall visit between **2** and **50** places in a single test case.

Output
For each test case, print a line in the format, "**Case X: Y / Z**", where **X** is the case number, **Y** is the number of Spring Secrets while **Z** is the number of Winter Obviouses present in the test case.

Sample Input	**Output for Sample Input**
2 SABCDPGSSMNSTRSS SSSSS	Case 1: 2 / 3 Case 2: 4 / 0

Explanation of the first sample case:

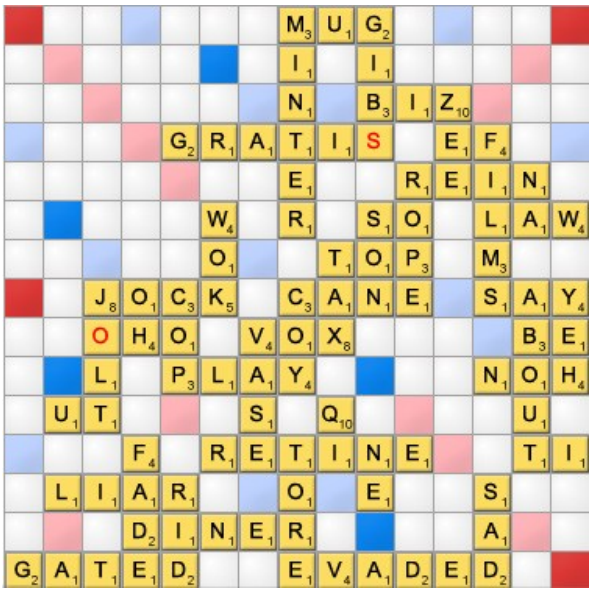
In the first sample test cases, the 2 **S** at position 8, &15 are Spring Secrets while the 3 **S** at position 1, 9 and 12 are Winter Obviouses.

Problem H
Scrabble

Input: Standard Input
Output: Standard Output

In this problem, you'll need to find the optimal word to place on a partially completed Scrabble game. Instead of using your previous knowledge of the game, only use the rules described below. Also, the rules described here are only a subset of all the rules of the game. But it should be enough for the purpose of the problem.

Game Description:



The game is played by two to four players on a square board with a 15-by-15 grid of cells (individually known as "squares") and 100 tiles. Each of the squares accommodates a single letter tile. The words are formed across and down in crossword fashion.

The game contains 100 tiles, 98 of which are marked with a letter and a point value ranging from 1 to 10. The other two tiles are blank and carry no point value._

Sequence of play:

Before the game a dictionary is selected for the purpose of adjudicating any challenges during the game. The letter tiles are put in an opaque bag. The players sit circularly and take turns in playing tiles from their 'racks'. During a turn, a player has to play at least one tile on the board, adding the value of all words formed to the player's cumulative score.

A proper play uses one or more of the player's tiles to form a contiguous string of letters that make a word (the play's "main word") on the board, reading either left-to-right or top-to-bottom. The main word must either use the letters of one or more previously played words or else have at least one of its tiles horizontally or vertically adjacent to an already played word. If words other than the main word are formed by the play, they are scored as well, and are subject to the same criteria of acceptability. Any contiguous string of letters on the board will always have to be a valid word.

When the board is blank, the first word played must cover the center square (8^{th} row and 8^{th} column). The word must consist of at least two letters, extending horizontally or vertically.

A blank tile may take the place of any letter. It then remains that letter for the rest of the game. It scores no points regardless of what letter it is.

Scoring:

- For each word formed in the play, add the normal point value of all other letters in the word (whether newly played or existing).

- If a player uses all **seven** of the tiles in the rack in a single play, a bonus of 50 points is added to the score of that play.

The problem:

Given a partially played valid scrabble board, a dictionary, point value of the letters and a rack, you have to find a placement of tiles such that point scored will be maximal.

Input

First line of input will contain an integer **N (N <= 1,800,000)**, no of words in the dictionary. Next **N** lines will contain one word each. Words will be given in lexicographic order and in lowercase letters. The length of a word will be between **2** and **15**.

Next line will contain **26** integers separated by a space. **i**-th integer will denote the point value of **i**-th character. So, first integer will be the point value of **A**, second integer will be the value of **B** etc.

Next line will contain an integer denoting number of test cases. After that **T (T <= 50)** test cases will follow. You will use the dictionary and point values described above on all subsequent test cases and queries.

Each test case starts with describing the current state of the board in **15** lines. Each line will correspond to one row of the board and will contain **15** characters. Each square on the board will either be a lowercase letter or a dot (‘**.**’), denoting an empty square. At most two of the squares can also be uppercase letters denoting a blank tile used as that particular character. You can assume that the board is valid (it is a result of sequence of valid plays).

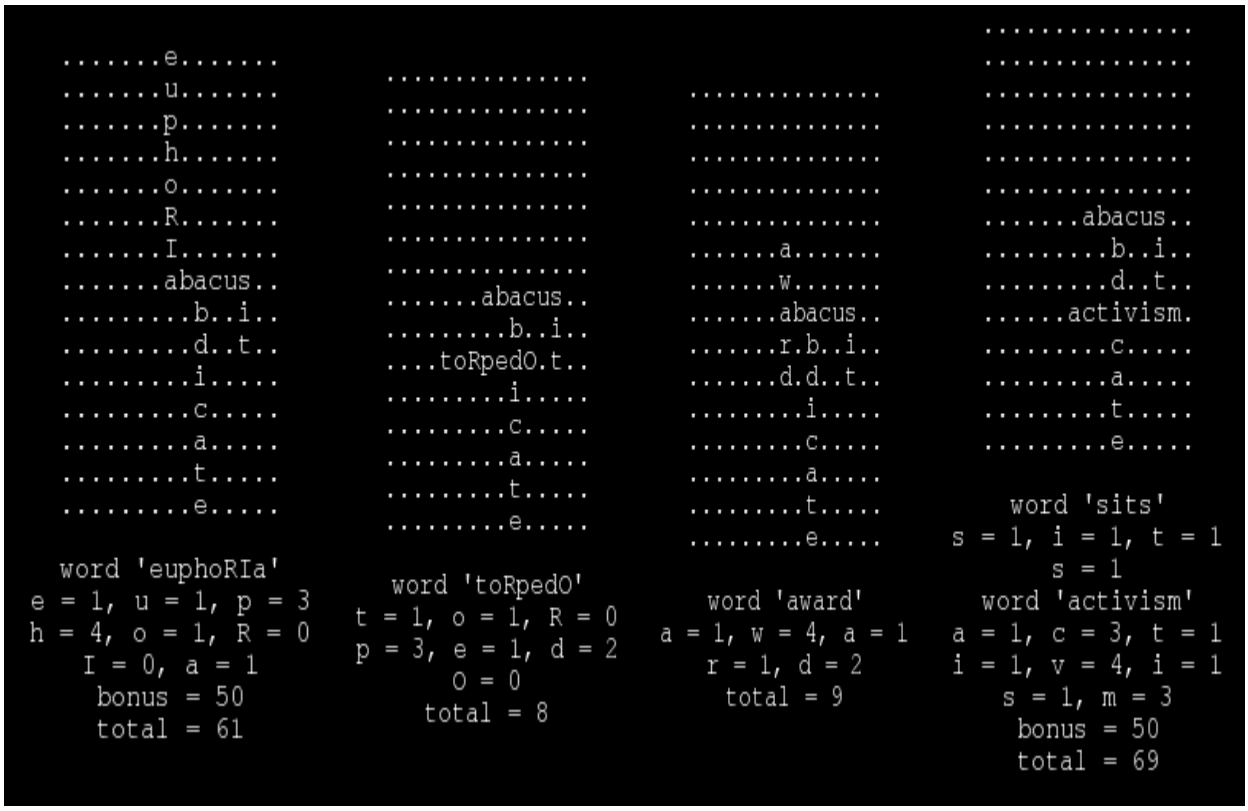
Next will be an integer **Q (Q <= 50)** denoting number of queries. Each of the next **Q** lines will contain a string of length at most **7** denoting the “**rack**”. Blank tiles will be denoted by ‘*****’. There will be at most 2 blank tiles. You can assume that the summation of number of blank tiles in board and in rack will be at most 2.

Output

The output for each case should start with a line in the format ‘**Case C**’ where **C** is the case number starting from 1. Then there will be **Q** lines in the format ‘**`Query i: total_points`**’ where **`i`** is the serial of the query (starting from 1), `total_points` is the maximum achievable points. Note that for some cases **`total_points`** can be zero if no valid placement can be found.

Sample Input	Output for Sample Input
12 abacus abdicate activism award chocoholic euphoria flexible sit sits torpedo torrential zoologist 1 3 3 2 1 4 2 4 1 8 5 1 3 1 1 3 10 1 1 1 1 4 4 8 4 10 1 ....abacus.. ....b..i.. ....d..t.. ....i..... ....c..... ....a..... ....t..... ....e..... 4 eupho** tope** rdwa activsm	Case 1 Query 1: 61 Query 2: 8 Query 3: 9 Query 4: 69

Illustration:



Here, upper case letters are used for blank tiles.

Problem I
An Average Game

Input: Standard Input
Output: Standard Output

Alice and Bob has just learned how to find average of some numbers. They got really excited and decided to come up with a game about finding average.

The game works like this, at the start of game a sequence of numbers is written.

Then there will be several rounds in the game. In first round Alice will say a number **x** and Bob has to select two index **i** and **j**. Let's say the average of the unique numbers between $\mathbf{i}^{\mathbf{th}}$ number and $\mathbf{j}^{\mathbf{th}}$ number is **y**. Then Alice gets **abs(x-y)** points in that round. In next round Alice and Bob switch the role. The game continues this way.

While Alice and Bob enjoy playing this game, they hate calculating average of unique numbers. So, they are asking you, their only programmer friend, to write a program that calculates the average for them.

Input

First line of input is a number **T (T <= 100)**. **T** test cases follow.

First line of each test case is an integer **n (0 < n < 10^4)** representing the length of number sequence. Next line consists n space separated integer representing the sequence. Each of these integers has absolute value less than **10^9**. Next line has an integer **q (q < 10^5)**. **q** lines follow, each representing a query. Each query has two space separated integer **i, j (1 <= i <= j <= n)**.

Output

For each case output **q** lines representing average of unique elements of corresponding range rounded **6** digits after decimal points. No case will have output whose rounding changes if the **10^-9** is added to or subtracted from answer.

Sample Input	Output for Sample Input
2 10 1 2 3 4 4 3 2 1 -1 0 4 1 4 1 10 3 5 8 10 3 1 1 0 1 1 3	Case 1: 2.500000 1.500000 3.500000 0.000000 Case 2: 0.500000

Explanation for second query in first sample

The range is (1,10) which includes ten numbers (1, 2, 3, 4, 4, 3, 2, 1, -1, 0). Unique numbers in this ranges are (1, 2, 3, 4, -1, 0) whose average is 1.5

Explanation for third query in first sample

The range is (3,5) which includes three numbers (3, 4, 4). Unique numbers in this ranges are 3 and 4 whose average is 3.5

Problem J
Permutation Counting

Input: Standard Input
Output: Standard Output

Dexter considers a permutation of first **N** natural numbers **good** if it doesn't have **x** and **x+1** appearing consecutively, where **(1 ≤ x < N)**. For example, for **N=3**, all **good** permutations are:

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

Input

Input starts with an integer **T (≤ 10000)**, denoting the number of test cases.

Each case starts with a line containing an integer **N (1 ≤ N ≤ 10^6)**.

Output

For each case, print the case number and the number of **good** permutations modulo **1000 000 007**.

Sample Input	**Output for Sample Input**
3 2 3 5	Case 1: 1 Case 2: 3 Case 3: 53