G. Centroid of Point Masses

The "centroid" of a region in two dimensions can be thought of as the point at which the region would balance on the end of a pencil. Computing this would require a bit more effort than we have in mind for this problem, so we restrict our attention to finding the centroid of a collection of point masses.

In this case, we could view the plane as being a thin, massless sheet with a few heavy points on it and ask where the plane would balance. To be more rigorous, we present a mathematical definition of the centroid for this case.

Given the coordinates (x_i, y_i) of a set of n points and the mass m_i at each point, we define the x-moment of that set of points relative to a given point (a, b) as follows (note the x-moment is defined in terms of y differences, but we will need both moments, so it doesn't really matter which way this is done for this particular problem)

$$M_x = \sum_{i=1}^{n} m_i \left(b - y_i \right)$$

Similarly, the y-moment is defined as $M_y = \sum_{i=1}^{n} m_i \left(a - x_i \right)$

The centroid of that set of points is defined to be the point (a, b) for which both moments are zero.

Input:

Input will be sets of points. Each set will be specified by the number of points n in the set followed by n lines of three numbers representing x_i, y_i, and m_i values for $i = 1$ to n. All these numbers will be integers from 1 to 5000. That is, n will be from 1 to 5000 and all the coordinates and masses will also be from 1 to 5000, just to make input easier. End of input will be marked by a negative value of n. There will be extra white space in input so that judges can read the input cases easily. Do not assume any particular number of spaces before, between, or after the input values, and do not assume a particular number of blank lines between cases.

Output:

Print the coordinates of the centroid. Follow the format exactly: "`Case`", a space, the case number, a colon and one space, and the values of a and b rounded to two decimal places separated by one space. Input will be constructed so that rounding will not cause problems for values that are sufficiently close to correct. Do not print any trailing spaces.

Sample Input	Sample Output

`3`
`1 1 10`
`22 1 10`
`1 31 10`
 	
`3`
`10 10 100`
`20 20 50`
`10 40 30`
 	
`-4`
	`Case 1: 8.00 11.00`
	`Case 2: 12.78 17.78`

A • Quick Change

J.P. Flathead's Grocery Store hires cheap labor to man the checkout stations. The people he hires (usually high school kids) often make mistakes making change for the customers. Flathead, who's a bit of a tightwad, figures he loses more money from these mistakes than he makes; that is, the employees tend to give more change to the customers than they should get.

Flathead wants you to write a program that calculates the number of quarters ($0.25), dimes ($0.10), nickels ($0.05) and pennies ($0.01) that the customer should get back. Flathead always wants to give the customer's change in coins if the amount due back is $5.00 or under. He also wants to give the customers back the smallest total number of coins. For example, if the change due back is $1.24, the customer should receive 4 quarters, 2 dimes, 0 nickels, and 4 pennies.

Input

The first line of input contains an integer ***N*** which is the number of datasets that follow. Each dataset consists of a single line containing a single integer which is the change due in cents, **C**, ($1 \leq \mathbf{C} \leq 500$).

Output

For each dataset, print out the dataset number, a space, and the string:

```
Q QUARTER(S), D DIME(S), n NICKEL(S), P PENNY(S)
```

Where ***Q*** is he number of quarters, ***D*** is the number of dimes, ***n*** is the number of nickels and ***P*** is the number of pennies.

Sample Input	Sample Output
3 124 25 194	1 4 QUARTER(S), 2 DIME(S), 0 NICKEL(S), 4 PENNY(S) 2 1 QUARTER(S), 0 DIME(S), 0 NICKEL(S), 0 PENNY(S) 3 7 QUARTER(S), 1 DIME(S), 1 NICKEL(S), 4 PENNY(S)

D. Four-Tower Towers of Hanoi

Refer to problem three for a description of the classic three-tower version of the Towers of Hanoi problem.

For this problem, we extend the Towers of Hanoi to have four towers and ask the question "What are the fewest number of moves to solve the Towers of Hanoi problem for a given *n* if we allow four towers instead of the usual three?" We keep the rules of trying to move *n* disks from one specified post to another and do not allow a bigger disk to be put on top of a smaller one. What is new for this problem is to have two spare posts instead of just one.

For example, to move 3 disks from post A to post D, we can move disk 1 from A to B, disk 2 from A to C, disk 3 from A to D, disk 2 from C to D, and disk 1 from B to D, making a total of 5 moves.

Input:

Input will be positive integers (*n*), one per line, none being larger than 1,000. For each value of *n*, compute the fewest number of moves for the four-tower problem. Stop processing at the end of the file. (There is no end-of-data flag.)

Output:

Output the fewest number of moves. Follow this format exactly: "`Case`", one space, the case number, a colon and one space, and the answer for that case. You may assume the answer will fit in a 64-bit integer type. Do not print any trailing spaces.

Sample Input	Sample Output

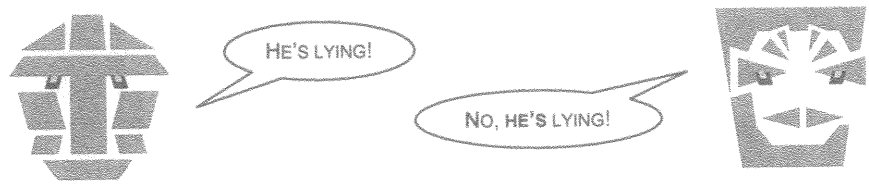
`1`
`3`
`5`
	`Case 1: 1`
	`Case 2: 5`
	`Case 3: 13`

UCF Local Contest — September 5, 2009

Tautobots and Contradicticons

filename: `logotron`

The planet Logotron is inhabited by two types of robots – the Tautobots and the Contradicticons. The Tautobots are programmed to always tell the truth, while the Contradicticons must always lie. Unfortunately, there is no simple way for outsiders to tell them apart, which often causes problems.



The Problem:

You are given a set of statements made by a group of robots. Every robot knows the type of every other robot, as well as itself. Each statement consists of an author (the robot that made the statement), a subject (the robot the statement is about), and the type of the subject (Tautobot or Contradicticon). For example, "Robot 5 says that Robot 2 is a Tautobot" is a valid statement. Note that if Robot 5 is a Contradicticon, then Robot 2 must also be a Contradicticon, since Robot 5 lied.

Given M statements made by N robots, you must find the number of distinct ways to assign a type to each robot, consistent with the statements. Two assignments are considered to be different if at least one robot is a Tautobot in one and a Contradicticon in the other.

The Input:

The first input line contains a positive integer T, indicating the number of test cases to be processed. This will be followed by T test cases.

Each test case is formatted as follows. The first line consists of the numbers N and M ($1 \le N \le 15$, $0 \le M \le 100$). This is followed by M lines, each of which represents a statement by one of the robots. A statement is formatted as "A S X". Here A and S are integers between 1 and N (inclusive) representing the author of the statement and its subject respectively (assume that A and S will be different robots). X will be one of the characters 'T' (for Tautobot) or 'C' (for Contradicticon).

Assume that a robot will not contradict itself (making a statement and then making the opposite of that statement) but different robots may contradict each other (in that case, there is no possible

answer, i.e., zero assignments). Also assume that we will not have the same statement repeated by a robot several times, i.e., no two input statements will be completely identical.

The Output:

For each test case, output a single line, formatted as: "Case #*t*:", where *t* is the test case number (starting from 1), a single space, and then the number of distinct assignments that can be made for that case. Follow the format illustrated in Sample Output.

Sample Input:

```
3
3 2
1 2 T
2 3 C
4 2
1 2 T
2 3 C
2 0
```

Sample Output:

```
Case #1: 2
Case #2: 4
Case #3: 4
```

Possible Passwords

After learning recursion, you've decided to look for some applications of the new problem solving technique you've learned. It turns out that there's an escalating rivalry between a couple student clubs on campus, "Unkempt Freshmen" and "Frustrated Student Underlings". The SGA at UCF suspects that both clubs are up to some fairly nefarious activities. In order to check up on the clubs, your boss has asked you to write a program that can guess passwords for the email accounts of each club. Luckily, after gathering some data, you know exactly how long each password is and what the possible letters are for each slot. As an example, it's possible you might have narrowed down a particular password to be three letters long where the first letter is from the set {'a', 'b', 'c'}, the second letter is from the set {'x', 'y'} and the third letter is from the set {'d', 'm', 'n', 'r'}. From this data, there are 24 possible passwords. You will have to write a program that can iterate through each possible password, in alphabetical order. Since printing out each of the passwords might create unnecessarily long output, to check to see that your program works, you'll only be asked to output specific alphabetically ranked possible passwords from the list, instead of the whole list itself.

The Problem

Given the length of a password, a list of possible letters for each letter in the password, and a desired alphabetical rank, determine the possible password of the given rank.

The Input

The first line of the input file will contain a single positive integer, c ($c \leq 100$), representing the number of input cases. The input cases follow, one per line. The first line of each input case will contain a single positive integer, m ($m \leq 20$), the length of the password. The following m lines will contain strings of distinct lowercase letters in alphabetical order representing each of the possible letters for each letter in the password. The i^{th} line in this set will store the possible letters for the i^{th} letter in the password, in alphabetical order. The last line of each test case will contain a single positive integer, r ($r \leq 1048576$), representing the rank of the possible password to output. (You are also guaranteed that the product of the lengths of these m lines won't exceed 1,000,000,000.)

The Output

For each case, output the correct possible password for the query, in all lowercase letters. It is guaranteed that all queries will be for a valid ranked password.

Sample Input

```
2
3
abc
xy
dmnr
10
2
abcdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
676
```

Sample Output

```
bxm
zz
```

Seating Arrangements

Filename: seating

You and your significant other enjoy going to the movies with your other couple friends. However, you've noticed that if any couple sits adjacent to each other, the corresponding public display of affection is not necessarily savory for the rest of the group. Thus, you want to ensure that whenever you go to the movies with your group of couple friends, the seating arrangements are such that no two couples are sitting next to each other but everyone is in one contiguous segment of seats, on a single row. To prove to the others that this is a good idea, you've decided to count the number of possible sets of seats the couples can arrange themselves adhering to this restriction. (You hope that by showing them that there are plenty of valid arrangements with the new restriction, the other couples will go along with your rule.)

For the purposes of this problem, the couples are indistinguishable from each other, as are the two members of each couple. Assume the seats are numbered 1, 2, ..., 2*n*, from left to right, where *n* is in the number of couples attending the movie. For example, if there are two couples, only one possible set of seats works: (1, 3), (2, 4). (Thus, we're NOT counting an arrangement where the couple in seats 1 and 3 swaps with the couple in seats 2 and 4, AND we're not counting differently if the couple themselves swap seats.) If there are three couples, the following five sets of seats works:

(1, 3), (2, 5), (4, 6)
(1, 4), (2, 5), (3, 6)
(1, 4), (2, 6), (3, 5)
(1, 5), (2, 4), (3, 6)
(1, 6), (2, 4), (3, 5)

The Problem

Given the number of couples attending a movie, determine the number of sets of seating arrangements on a contiguous segment of a single row that are valid such that no two couples are sitting next to one another.

The Input

The first line of the input file will contain a single positive integer, *n* ($n \leq 9$), representing the number of input cases to consider. Each of the test cases will follow, one per line. Each test case will have a single positive integer, *c* ($c \leq 9$), on a line by itself.

The Output

For each test case, print out the number of valid seating arrangements for that case.

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

Maximum Stock Return

Filename: `stock`

In your Economics class, you calculated the profit of buying and selling a stock once, over the course of a semester. However, real day traders tend to make more than one trade every season. In fact, as the term suggests, they make trades every day!!! In this problem, you want to build a better model that might earn you more money than making just one trade. To keep things simple however, at any given point in time, you're only allowed to own one stock out of two possible stocks. (You may also keep all of your money out of the market for a particular day.) If you own a particular stock, you will buy as many shares of that stock as possible.

In this problem you'll be given the daily stock history of two stocks, the initial amount of money you have to invest, as well as the cost of a transaction, known as a transaction fee. A transaction is the buying or selling of any quantity of shares of a stock.

As an example, consider the following three day data of Google stock and Apple stock:

Google: 759.68, 765.74, 770.17
Apple: 443.00, 457.82, 457.04

Imagine that we started with $10,000 and the transaction fee was $50. On day one, we buy Apple stock. The maximum we can buy is 22 shares, which costs us 22 x $443 + $50 = $9796. Thus we have $204 cash leftover. In one day, our 22 shares we own are worth 22 x $457.82 = $10072.04. Thus our current value, if we choose to cash out would be $10072.04 - $50 + $204 = $10226.04. If we then turn around at the end of that day and buy Google stock, we can get 13 shares for 13 x $765.74 + $50 = $10004.62, leaving us with $221.42 in cash. At the end of the last day, the value of our Google stock is 13 x $770.17 = $10012.21. At the end of our trading period (three days in this example) we are forced to sell the stock and our total value at the end of the three days for these set of actions is $221.42 (old cash) + $10012.21 (selling stock) - $50 (selling fee) = $10183.63. Alternatively, we could have simply chosen to not buy any stock on day three. In this scenario, we simply keep the $10226.04 that we had right after selling the Apple stock. (For this scenario, this is the optimal outcome.)

The Problem:

Based on this the daily stock history of two stocks for a fixed trading period, the initial amount of money you have to invest and the transaction fee, you need to calculate the maximum money you can have at the end of the last day of the trading period given, assuming that you cash out at the end of that day.

The Input:

The first line of the input file has a single positive integer, n ($n \leq 50$), representing the number of stock scenarios to evaluate. The first line of each stock scenario has the number of days for that scenario, d ($1 < d \leq 15$), followed by a space, then a positive real number given to two decimal places, T ($T \leq 100.00$), representing the transaction fee for the scenario, followed by a space and a positive real number given to two decimal places, M ($M < 10000000$), representing the amount of money available to invest at the beginning of the stock scenario in dollars. The second line of

each stock scenario will contain *d* positive real numbers given to two decimal places separated by spaces representing the value of stock #1 on days 1 through *d*, respectively. The third line of each stock scenario will contain *d* positive real numbers given to two decimal places separated by spaces representing the value of stock #2 on days 1 through *d*, respectively. All of the costs for each stock per share will be less than 1000.00 (in dollars).

The Output:

For each stock scenario, output a single line with the maximum value in dollars to decimal places that can be obtained by trading under the given restrictions of the problem.

Sample Input:

```
2
3 50.00 10000.00
759.68 765.74 770.17
443.00 457.82 457.04
2 5.99 29999.99
59.99 58.99
22.67 20.73
```

Sample Output:

```
10226.04
29999.99
```

B • Triangular Sums

The **n**th *Triangular* number, **`T(n) = 1 + ... + n`**, is the sum of the first **n** integers. It is the number of points in a triangular array with **n** points on side. For example **`T(4)`**:

```
X
X X
X X X
X X X X
```

Write a program to compute the weighted sum of triangular numbers:

$$W(n) = \texttt{SUM[k = 1..n; k*T(k+1)]}$$

Input

The first line of input contains a single integer **N**, ($1 \le$ **N** ≤ 1000) which is the number of datasets that follow.

Each dataset consists of a single line of input containing a single integer **n**, ($1 \le$ **n** ≤ 300), which is the number of points on a side of the triangle.

Output

For each dataset, output on a single line the dataset number, (1 through **N**), a blank, the value of **n** for the dataset, a blank, and the weighted sum , **W(n)**, of triangular numbers for **n**.

Sample Input	Sample Output
4 3 4 5 10	1 3 45 2 4 105 3 5 210 4 10 2145

Swan Boats

Filename: swan

You've decided to go downtown to the beautiful lake and rent the pedal powered swan boats. Since the rental cost is based on time, you'd like to minimize the time you spend in the boat while visiting each important location in the lake. Write a program to help you calculate the minimum possible cost for your rental. For the purposes of this problem, the lake is a circle centered at (0, 0) with a radius of 1000 meters.

The Problem

Given the starting location of your swan boat, each important location you would like to visit, and your pedaling speed in meters/minute, determine the minimum amount of time necessary to visit each location and return the swan boat to its starting location.

The Input

The first line of the input file will contain a single positive integer, n ($n \leq 50$), representing the test cases to process. Each of the test cases will follow. The first line of each test case will contain three space separated integers: k *(1 ≤ k ≤ 10)*, the number of important locations, a *(0 ≤ a < 360)*, representing the degree measure of the starting point in relation to the center of the lake, and s *(1 ≤ s ≤ 100),* representing your pedaling speed in meters per minute. (0^o represents east, 90^o represents north, 180^o represents west, and 270^o represents south.) The starting point is always on shore, exactly 1000 meters from the center of the lake. The following k lines will contain information about each important location. The i^{th} of these lines will have two non-negative integers, d_i *(1 ≤ d_i ≤ 1000)*, representing the distance of the i^{th} important location from the center of the lake, and a_i *(0 ≤ a_i < 360)*, representing the angle from the center of the lake of the i^{th} important location.

The Output

Output a single positive real number rounded to two decimal places representing the minimum time, in minutes, your journey could take.

Sample Input

```
2
1 0 100
1000 180
5 90 10
80 30
1000 180
280 40
678 235
995 209
```

Sample Output

```
40.00
420.12
```

C. One Move from Towers of Hanoi

For this problem, we are concerned with the classic problem of Towers of Hanoi. In this problem there are three posts and a collection of circular disks. Let's call the number of disks *n*. The disks are of different sizes, with no two having the same radius, and the one main rule is to never put a bigger disk on top of a smaller one. We will number the disks from 1 (smallest) to *n* (biggest) and name the posts A, B, and C. If all the disks start on post A, and the goal is to move the disks to post C by moving one at a time, again, never putting a bigger one on top of a smaller one, there is a well-known solution that recursively calls for moving *n*–1 disks from A to B, then directly moves the bottom disk from A to C, then recursively calls for moving the *n*–1 disks from B to C.

Pseudocode for a recursive solution to classic Towers of Hanoi problem:

```
move(num_disks, from_post, spare_post, to_post)
  if (num_disks == 0)
    return
  move(num_disks - 1, from_post, to_post, spare_post)
  print ("Move  disk ", num_disks, " from ",
    from_post, " to ", to_post)
  move(num_disks - 1, spare_post, from_post, to_post)
```

The problem at hand is determining the k^{th} move made by the above algorithm for a given *k* and *n*.

Input:

Input will be two integers per line, *k* and *n*. End of file will be signified by a line with two zeros. All input will be valid, *k* and *n* will be positive integers with *k* less than 2^n so that there is a k^{th} move, and *n* will be at most 60 so that the answer will fit in a 64-bit integer type.

Output:

Output the requested k^{th} move made by the above algorithm. Follow this format exactly: "`Case`", one space, the case number, a colon and one space, and the answer for that case given as the number of the disk, the name of the from_post, and the name of the to_post with one space separating the parts of the answer. Do not print any trailing spaces.

Sample Input	Sample Output
1 3 5 3 8 4 0 0	Case 1: 1 A C Case 2: 1 B A Case 3: 4 A C