-1 A fair warning

Like in regular ICPC, today's problems are not sorted by difficulty. Do **not** attempt to solve them in order! Instead, use your brain and/or the scoreboard to figure out which problem to attempt next.

0 Some more C++ tips

0.1 Printing floating point numbers

The default settings for `cout` do not print with enough decimal digits to pass the precision bounds of typical ICPC problems. Use the following commands to increase the precision of your output:

```
// 15 digits of precision
cout << fixed << setprecision(15) << my_double << endl;
```

You may need to `#include <iomanip>`.

0.2 One header to rule them all

Instead of including `<vector>`, `<algorithm>` etc. etc. all by one, you may use

```
#include <bits/stdc++.h>
```

to pull in the entire C++ standard library.

0.3 Compiler flags for debugging

If your program misbehaves and you can't figure out why, it may help to use the following compiler flags for debugging:

```
g++ -std=gnu++14 -O2 -g -Wall -Wextra -Wconversion -Wshadow -
    ↪ fsanitize=address -fsanitize=undefined my_program.cpp
```

This will do two things:

- Turn on a huge amount of warnings
- Enable runtime checks for out-of-bounds, division by zero and a bunch of other typical bugs

Caveat: Your program will take more time to compile and run much slower (but, of course, only locally, the judge doesn't care how you compiled your programm on your machine).

0.4 Runtime Estimate

Think before coding! Before implementing your algorithm estimate roughly the number of steps s you will need in the worst case. Consider the bounds given in the input description.

- If $s \leq 10^7$: The solution will almost surely be fast enough.
- If $10^7 \leq s \leq 10^8$: Your solution should be fast enough, as long as you do not use very expensive operations in an inner loop. Examples for expensive operations are mod and divisions.
- Otherwise: The algorithm is too slow.

1 Selling CPUs

1.1 Problem

You are very happy that you got a job at ACME Corporation's CPU factory. After a hard month of labor, your boss has given you c identical CPUs as payment. Apparently, ACME Corporation is low on cash at the moment.

Since you can't live on CPUs alone, you want to sell them on the market and buy living essentials from the money you make. Unfortunately, the market is very restrictive in the way you are allowed to conduct business. You are only allowed to enter the market once, you can only trade with each merchant once, and you have to visit the traders in a specific order. The market organizers have marked each merchant with a number from 1 to m (the number of merchants) and you must visit them in this order. Each trader has his own price for every amount of CPUs to buy.

1.2 Input

The input consists of:

- one line with two integers c and m ($1 \leq c, m \leq 100$), where c is the number of CPUs and m is the number of merchants;
- m lines describing the merchants. Each merchant is described by
  - one line with c integers $p_1, \ldots, p_c$ ($1 \leq p_i \leq 10^5$ for all $1 \leq i \leq c$), where p_i is the amount of money the merchant will give you for i CPUs.

1.3 Output

Output the maximum amount of money you can make by selling the CPUs you have.

Note that you don't have to sell all CPUs.

1.4 Sample Data

Input	Output
5 3 1 4 10 1 1 1 1 8 1 1 1 1 9 1 1	13

2 Knapsack

2.1 Problem

The Knapsack problem is a very well-known optimization problem: Given n items with size w_i and value p_i and a knapsack of size M, select some items which fit in the knapsack and maximize the total value of the selected items. In other words, find some $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} p_i$ is maximized under the additional constraint $\sum_{i \in S} w_i \leq M$.

In 1972, Richard Karp included this problem in his seminal paper „21 problems you won't believe are actually probably hard" and it made another appearance in the 2000 article „Solve one of THESE easy problems and become a millionaire!" by the Clay Mathematics Institute. As of May 18, 2022, no polynomial time solution for the Knapsack problem is known.

Year after year, students around the world rejoice when they discover an algorithm for the Knapsack problem that runs in time $\mathcal{O}(n \cdot M)$. However, when they demonstrate their algorithm to their professors using an example instance containing their bank account number, the professors ask them to leave their office, muttering something about „binary representation" and „pseudopolynomial algorithm", whatever that means.

To add insult to injury, here at Passau University, we like spoiling all the fun, so if you didn't already hear the solution in your algorithms or TI lecture, you learned about it last Thursday. Man, we really are no fun.

So to make it up to you, we will give you the chance to at least implement the solution so you can feel the rush of seeing it actually give the correct answer. However, working on such an important problem is probably pretty intimidating for you, so we've decided to make the problem a little bit easier for you by setting $p_i = w_i \in \{1, 2, 3\}$ for all i. However, the solution from the lecture slides is too slow for the judge, so you have to come up with an idea exploiting the advantages of the special case we consider here.

2.2 Input

The input consists of a single testcase, which consists of two lines. The first line contains two integers, n and M ($1 \leq n, M \leq 2 \cdot 10^5$): The number of items and the capacity of the knapsack.

The second line contains n integers w_i ($1 \leq w_i \leq 3$) separated by single spaces: The weight of the items, which is also the value of the items.

2.3 Output

Output a single integer: The value of an optimal solution to the Knapsack problem.

2.4 Sample Data

Input	Output
3 5 3 3 1	4

3 Speedrun

3.1 Problem

A *speedrun* is a playthrough of a game with the intention to complete it as quickly as possible. When speedrunning, you usually follow a pre-planned path through the game. Along this path, there may be some places where you have to pull off a difficult technique, or *trick*, which may cause a delay if you fail to pull it off successfully. Luckily you can *reset* the game at any time: if you have made a few mistakes, you can start a new run, losing your progress but instantaneously starting over with a clean slate. You can do this as often as you like.

The game you are currently speedrunning has a record of r seconds, which you intend to beat. You have discovered a path through the game that, in the best case, takes $n < r$ seconds. There are some tricks along the way, though: you know exactly where along the run they occur, what the probability is that you will pull them off successfully, and how many seconds you have to spend to recover if they fail.

Given this data, you want to find the optimal strategy for when to reset the game to minimise the expected time to set a new record. Write a program to determine what this smallest possible expected time is.

3.2 Input

The input consists of:

- One line with three integers n, r and $m (2 \leq n < r \leq 5000, 1 \leq m \leq 50)$, where n and r are as described above and m is the number of tricks.
- m lines, each containing three numbers describing a trick:
  - An integer $t (1 \leq t < n)$, the time in the route (assuming no failed tricks before) at which the trick occurs,
  - a real number $p (0 < p < 1$ and p has at most 6 digits after the decimal point), the probability that the trick succeeds, and
  - an integer $d (1 \leq d \leq 1000)$, the number of seconds required to recover in case the trick fails.

The tricks are given in sorted order by t, and no two tricks occur at the same time t in the route. You may assume that, without resetting, a single playthrough has a probability of at least 1 in 50000 to succeed at improving the record.

3.3 Output

Output the expected time you will have to play the game to set a new record, assuming an optimal strategy is used. Your answer should have an absolute or relative error of at most 10^{-6}.

Explanation of Sample Input 1: The record for this game is 111 seconds, and your route takes 100 seconds if everything goes right.

After playing for 20 seconds, there is a trick with a 50% success rate. If it succeeds, you keep playing. If it fails, you incur a 10 second time loss: now the run will take at least 110 seconds. It is still possible to set a record, but every other trick in the run has to be successful. It turns out to be faster on average to reset after failing the first trick.

Thus you repeat the first 20 seconds of the game until the trick is successful: with probability $\frac{1}{2}$, it takes 1 attempt; with probability $\frac{1}{4}$, it takes 2 attempts; and so on. On average, you spend 40 seconds on the first 20 seconds of the route.

Once you have successfully performed the first trick, you want to finish the run no matter the result of the other tricks: it takes 80 seconds, plus on average 1 second loss from each of the remaining 4 tricks. So the expected time until you set a record is 124 seconds.

3.4 Sample Data

Input	Output
100 111 5 20 0.5 10 80 0.5 2 85 0.5 2 90 0.5 2 95 0.5 2	124
2 4 1 1 0.5 5	3
10 20 3 5 0.3 8 6 0.8 3 8 0.9 3	18.9029850746
10 50 1 5 0.5 30	15

4 Coin Change

Note: This problem has an unusally large time limit of 15 seconds per input file.

Bob is really afraid of germs and, thus, hates touching anything that hasn't been thoroughly disinfected. He especially hates coins. Just think of where those might have been and what forms of life might be evolving in your wallet right now!

Unfortunately, Bob still owes Jimmy X€ and Jimmy wants his money back. As electronic payment is not an option for Jimmy, Bob wants to figure out how to pay back Jimmy while exchanging a minimum number of coins.

Assume that Bob's debt was 4€. Then Bob would have to give Jimmy two 2€ coins and therefore two coins would have to be exchanged. Now assume that the debt was 4.99€. In this case it would be best if Bob gave Jimmy a 5.00€ bank note [1] and Jimmy returned a 0.01€ coin. Again, two coins would be exchanged. If the debt was 19.01€, then Bob should give Jimmy a 20€ bank note as well as a 0.01€ coin and return a 1€ coin.

Help Bob by writing a program that computes the minimum number of coins that have to be transferred given the values of the coins and Bob's debt. You may assume that Bob and Jimmy have an infinite number of every coin at their disposal.

4.1 Input

The first line contains the number of test cases (at most 20). The first line of every test case contains two integers X ($0 \leq X \leq 10^7$) and C ($0 < C \leq 10$) in this order. The debt is given by X and C is the number of different coin values.

Next a line with C integers separated by spaces follows. These are the values of the coins. They are given in ascending order and you may assume that no two coins exist with the same value. Also, there will always be a coin with value 1 and the maximum coin value will not exceed 3000.

4.2 Output

For every test case output a single line containing the minimum number of coin exchanges needed.

[1] Bob hates bank notes just as much as he hates coins, so there is no point in discerning between the two.

4.3 Sample Data

Input	Output
3 400 5 1 100 200 500 2000 499 5 1 100 200 500 2000 1901 5 1 100 200 500 2000	2 2 3

5 Farida

5.1 Problem

Once upon time there was a cute princess called Farida living in a castle with her father, mother and uncle. On the way to the castle there lived many monsters. Each one of them had some gold coins. Although they are monsters they will not hurt. Instead they will give you the gold coins, but only if you didn't take any coins from the monster directly before the current one. To marry princess Farida you have to pass all the monsters and collect as many coins as possible. Given the number of gold coins each monster has, calculate the maximum number of coins you can collect on your way to the castle.

5.2 Input

The first line of input contains T, the number of test cases, $1 \leq T \leq 10$.
Each test case starts with a number N, the number of monsters, $0 \leq N \leq 10^5$.
The next line will have N numbers m_i, indicating the number of coins each monster has, $1 \leq m_i \leq 10^5$. Monsters are described in the order they are encountered on the way to the castle.

5.3 Output

For each test case print the maximum number of coins you can collect in a separate line.

5.4 Sample Data

Input	Output
2 3 10 15 10 4 1 100 1 100	20 200

6 Hedge

6.1 Problem

We have N rectangular planks, numbered from 1 to N ($0 < N < 2001$). The planks are of equal width and the K-th plank has a height of K centimeters.

We build a fence by lining up all N planks. Dependent on the order of alignment, we form several sections of consecutive planks of decreasing height. Usually there is more than one section, but it is possible that the whole fence forms one decreasing sequence. It is also possible that some sections consist of only one plank.

For example, if we put down 9 planks so that their heights form the sequence 6, 4, 5, 3, 9, 7, 2, 1, 8, we have the four sections $(6, 4)$, $(5, 3)$, $(9, 7, 2, 1)$ and (8).

Write a program that finds the number of all possible arrangements with exactly P sections ($0 < P < N + 1$).

6.2 Input

Each line in the input (there are at most 1000) contains the two numbers N and P separated by a single space. The input is terminated by a line containing two zeroes, which are not to be processed.

6.3 Output

For each pair (N, P) the program should print the number of possible arrangements of N planks so that they form exactly P sections modulo 1020847.

6.4 Sample Data

Input	Output
4 3 20 2 0 0	11 27708