IPSC 2013

problems and sample solutions

Problem P: Plus one

You have just realized that the average of all the numbers in the world is too small. Therefore, you decided to increase some of them inconspicuously.

Problem specification

You are given several integers. Increment each of them by one.

Input specification

The input consists of several lines, each line contains single integer x.
The absolute value of each integer lies between 0 and 1 000, inclusive.

Easy subproblem P1: There are exactly 50 lines of input.
Hard subproblem P2: There are exactly 1 000 lines of input.

Output specification

For each integer x in the input, output a single line with the integer $x + 1$.
Use precisely the same format that was used for x.

Example

input

```
1
46
41
```

output

```
2
47
42
```

Note: This example has only 3 lines of input.
The input files `p1.in` and `p2.in` contain 50 and 1 000 lines, respectively.

Also note: Please do NOT submit any programs.
For each subproblem, just produce and submit a correct output file.

Task authors

Problemsetter:	Jano Hozza
Task preparation:	Jano Hozza, Peter 'Bob' Fulla

Solution

This task was easy but it had a malicious twist – after all, practice problems have to prepare you for the villainy on the real contest :)

Did you fall into our trap and write a program before actually looking at the input files? Or did the phrase "Use precisely the same format that was used for x." ring a warning bell?

Either way, once you found out what is really going on, there were many ways of dealing with the problem. Possibly the simplest way is to write a semi-interactive program: if the input is an integer, it increments it automatically, otherwise it prompts the user for the correct output. As there were only 20 special numbers in each input, this approach was really feasible.

Solving the hard subproblem probably required googling and using some language dictionary.

Here are the trickier inputs from the hard subproblem and their correct outputs:

- `evil matching` → `fair coin toss` (last year's IPSC problems)
- `saturn` → `uranus` (planets)
- `ekans` → `arbok` (pokémon – both by evolution and by official ID)
- `di-di-di-di-dah dah-dah-dah-dah-dit` → `di-di-di-di-dit dah-dah-dah-dah-dah` (phonetic Morse code – note that only a trailing dot is `dit`.)
- `one-one-zero-one-one-one` → `one-one-one-zero-zero-zero` (binary digits)
- `neves-ytfif` → `thgie-ytfif` (English backwards)
- `abcdefghijklmnopqrstuxyzwv` → `abcdefghijklmnopqrstuxzvwy` (permutations in lexicographic order)
- `51x7y-n1n3` → `53v3n7y` (l33t sp34k, use the same l33t letters as in the input)
- `gold` → `mercury` (elements by proton number)
- `osemdesiatdevat` → `devatdesiat` (89 and 90 in Slovak, without diacritics)

Hopefully you can forgive us if some of the test cases in the hard input seemed ambiguous to you. We did our best to help you out – the grader was verbose and always reported the first mistake you made. We were also helpful if you contacted us with a clarification request. And getting a few *Wrong answer*s in the practice session is actually a good thing – at least you saw how they look like! :)

Problem Q: Quite the cheater!

Your physics lab report is due tomorrow. However, you had no time to do the required experiments, as you spent all your time practicing for the IPSC. Therefore you decided to write a fake report quickly. Here is how to get a good grade for your lab report:

- It has to contain a lot of measurements.
- You already know the correct value you were supposed to measure. The mean of all "measured" values in your report has to be equal to that value.
- The values must look sufficiently random to avoid suspicion that you made them all up. (Yeah, right.) More formally, they must have a sufficient variance.

Problem specification

You are given two integers: the desired mean μ and the desired variance v.

Pick a number of measurements n and the values of those measurements $a_1, \dots, a_n$ such that the mean of those values is exactly μ and their variance is (easy subproblem: at least v / hard subproblem: exactly v). Formally, your values must satisfy the following conditions:

- $10 \leq n \leq 1000$
- Each a_i is an integer between -10^9 and 10^9, inclusive.
- The value μ is exactly the mean: $\mu = (a_1 + \cdots + a_n)/n$.
- The variance of your values[1] is computed as follows: $(1/n) \cdot \Big((a_1 - \mu)^2 + \cdots + (a_n - \mu)^2 \Big)$.
- In the easy subproblem Q1: the variance of your values must be at least v.
- In the hard subproblem Q2: the variance of your values must be exactly v.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case contains a single line with two integers: μ and v.

You may assume that $t \leq 100$, $|\mu| \leq 10^6$, and $0 \leq v \leq 10^9$.

Output specification

For each test case, output two lines. The first line should contain the number of values n, the second line a space-separated list of values $a_1, \dots, a_n$. Any valid solution will be accepted.

Example

input

```
1

47 2080
```

output

```
11
34 -7 102 117 16 8 0 130 36 34 47
```

This would be a correct solution to both subproblems. I.e., this sequence of 11 values has mean exactly 47 and variance exactly 2080.

[1] If you are a statistics buff, note that we are not using the unbiased sample variance formula (the one with $1/(n-1)$ instead of $1/n$), as in our case the mean is known a priori. If the previous sentence makes no sense, just ignore it and use the formula in the problem statement.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek, Lukáš 'lukasP' Poláček

Solution

This was an easy task. Don't get me wrong, it's scary, with all the math, but once you deal with your fear and actually read the statement, you probably quickly realized that there is not much to solve here.

The mean is easy. You want a sequence with mean 47? Here's one: 47, 47, 47, 47, 47, 47.

Now what about variance? This sequence clearly has variance zero – it does not vary at all. How would you make a sequence that varies, and still has the same mean? The simplest way of doing so: increase some values, and decrease other ones by the same amount. Here is a sequence with variance 100: 37, 57, 37, 57, 37, 57.

And just like that, we can solve the easy subproblem. For a given μ and v, just pick $n = 10$ and output five values $\mu - 10^6$ and five values $\mu + 10^6$. The mean is clearly μ, and the variance is 10^{12}, which is more than any valid input v can be.

Now for the hard subproblem. We cannot just take values that are wildly greater and smaller than the mean, we have to get the variance just right.

First of all, we'll do a small trick: we will rewrite the requirement $v = (1/n)\cdot\Big((a_1-\mu)^2+\cdots+(a_n-\mu)^2\Big)$ as follows: $nv = (a_1 - \mu)^2 + \cdots + (a_n - \mu)^2$. Why did we do this? Because now we have to deal with integers only, there is no division.

Additionally, we can already pick some n. Let's go with the largest possible $n = 1000$, as it gives us the most freedom in choosing the elements. (Later on we shall see that almost any even n would do.)

Another thing that can simplify our construction: let's first find a solution for $\mu = 0$. A solution for a different value μ can then easily be constructed by adding μ to all elements of the original solution.

What does that leave us with? We are looking for a sequence $a_1, \dots, a_n$ such that $a_1 + \cdots + a_n = 0$ and $a_1^2 + \cdots + a_n^2 = nv$.

We will pick the values in pairs: $a_2 = -a_1$, $a_4 = -a_3$, and so on. That alone will make sure that the sum of all a_i will be 0 at the end. Now, how to choose their actual values? It turns out that a simple greedy approach works.

For example, suppose that you want to have $a_1^2 + \cdots + a_n^2 = 4700$. What is the largest $|a_1|$ you can take? You have to have $a_1^2 + a_2^2 \le 4700$, in other words, $a_1 \le \sqrt{4700/2}$. As we want to get as close as possible to 4700, we pick $a_1 = \lfloor\sqrt{4700/2}\rfloor = 48$ and $a_2 = -a_1 = -48$. And this leaves us with a smaller problem: we need to find $a_3, \dots, a_n$ such that $a_3^2 + \cdots + a_n^2 = 4700 - 2 \cdot 48^2 = 92$.

We can easily see that the greedy approach always works (as long as the desired value is even). We cannot get stuck – if the desired value is still positive, we can always decrease it by two, by taking $a_{2k+1} = 1$ and $a_{2k+2} = -1$. Actually, this greedy solution converges quite quicky, as each step decreases the desired value to approximately the square root of the previous one. And that's it.

Did you like the problem? For an additional challenge, try solving it with n as small as possible.

Problem R: Rearranged alphabet

The string `abcabac` has a peculiar property: each permutation of `a`, `b`, and `c` occurs in it as a subsequence:

```
abcabac
-------
a   b c
a c b
 b a  c
 bca
  cab
  c ba
```

Your task is to find such a string for the entire English alphabet. That is, a string such that each of the 26! permutations of `a` through `z` occurs in it as a subsequence.

Input specification

There is no input.

Output specification

Your output must contain a single line with a string of lowercase English letters.
For the easy subproblem R1 we will accept any valid string that has at most 660 letters.
For the hard subproblem R2 we will accept any valid string that has at most 640 letters.
(There are valid answers shorter than 630 letters.)

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek, Monika Steinová

Solution

It's trivial to find a solution with 26^2 letters: we can just take 26 consecutive copies of the whole alphabet (actually, each of them can be in any order we like). Any permutation can be found in such a string – from each alphabet we can pick one letter we need.

Unluckily, this is over the limit for the easy subproblem – but not by much: $26^2 = 676$, and we are supposed to get it to 660 or below. That seems like all it needs is a small tweak.

The easy subproblem

One possible tweak: Assume that we have a string that consists of 25 copies of the alphabet, each in its usual order. This string is almost a solution! Why? Almost any permutation contains two consecutive letters such that the first one is smaller than the second one. And in such a case, we can take both those letters from the same copy of the alphabet. For example, when looking for the permutation `kjbd...`, we can take `k` from the first alphabet, `j` from the second one, and both `b` and `d` from the third one.

There is only a single permutation that cannot be found in our shorter string: the reversed alphabet. Luckily, this is easily fixed by appending one more `a` to our string.

This is the solution we get:

$$(\texttt{abcdefghijklmnopqrstuvwxyz})^{25}\texttt{a}$$

The length of this string: $25 \times 26 + 1 = 651$, well within the limit for the easy subproblem.

The hard subproblem: analysis

The hard subproblem was intended to be the hardest subproblem of this year's practice session. Solving this task optimally (for a general alphabet) is actually still an open problem. The limit of 640 was chosen so that there would be some known constructions that can do strictly better. Below, we show one such construction (originally by Leonard Adleman). For an alphabet with $n \geq 3$ letters it produces a string of length $n^2 - 2n + 4$.

We will start by slightly generalizing the solution to the easy problem. Take any fixed alphabet and form an infinite string S by concatenating its copies. Here is an example for the alphabet `a` through `g` (with $n = 7$ letters):

$$S = \texttt{abcdefgabcdefgabcdefgabcdefgabcdefg...}$$

What do we already know about this string? We know that its prefix of length $n(n-1)+1$ already contains all possible permutations of the alphabet. Now the generalization: Consider all strings of length k that do not contain any letter twice. Let's call them diverse strings of length k. (There are no such strings for $k > n$, and $k = n$ are precisely our permutations.) Given k, what is the shortest prefix of S that contains each of the diverse strings of length k?

The answer is pretty easy: $k(n-1)+1$ letters are enough. For example, `abcdefg` contains all 1-letter, and `abcdefgabcdef` contains all 2-letter diverse strings over the alphabet `a` through `g`. The proof is essentially the same as for permutations, we leave the details as an exercise :)

In fact, not only the prefixes of S have this property. Any substring of S of length $k(n-1)+1$ works. (This is because each substring of S looks exactly the same, only the order of letters in the alphabet changes.)

We can now use this observation to construct a shorter solution to our original problem. Consider the following string (with spaces inserted for clarity):

```
abcdefgabcdef h gabcdef
```

Obviously, this string contains each diverse string of the form `??h?`, where each `?` is a letter from `a` to `g`.

Another similar string to consider:

```
abcdefgabcdefgabcde h f
```

This one clearly contains each diverse string of the form `???h` (over the same alphabet). This is because the first part of the string is long enough to contain each 3-letter diverse string made of the letters `a` through `g`. (The last `f` is actually not necessary, but we kept it anyway – now the only difference between this example and the previous one is that the `h` moved six positions to the right.)

Of course, we can take the same string `abcdefgabcdefgabcdef` we used in the previous two examples, and insert multiple `h`s at the same time:

```
abcdefgabcdef h gabcde h f
```

This one clearly contains each diverse string of the form `??h?`, and also each diverse string of the form `???h` (over the same alphabet).

The hard subproblem: construction

We can now use the above observations to come up with the following construction for a general n:

Consider the infinite string S for the first $n-1$ letters of the alphabet. Take its prefix P of length $(n-1)(n-2)+2$. For each i between 0 and $n-1$, inclusive, insert the last letter of the alphabet after $1+i(n-2)$ letters of P.

Here is the entire string (with spaces for clarity) for $n=8$ and the alphabet `a` through `h`:

```
a h bcdefg h abcdef h gabcde h fgabcd h efgabc h defgab h cdefga h b
```

It should be obvious that this string contains each of the 8! possible permutations of its alphabet: E.g., for any permutation of the form `???h????` we can use the fourth `h` in our string and ignore the other `h`s. The string over `a-g` before the fourth `h` is long enough to contain any 3-letter diverse string, and the same holds for the part after the chosen `h` and 4-letter diverse strings.

This construction starts with a string of length $(n-1)(n-2)+2$, and then inserts n additional characters. Thus the total length of the constructed string is n^2-2n+4, as promised.

One final note: This construction gives the shortest possible solution for $n \leq 7$. It is also known that there is an even better solution already for $n=10$. As stated above, the general problem of determining the best solution is still open. Are you up for a challenge? :)

Problem A: Advice for Olivia

Olivia is going to work in the candy shop during the summer. However, she is afraid she'll have to work at the cash register. Whenever the cash register tells her to return some money to the customer, Olivia panics, because she can't decide which denominations to use. And if she takes too long (i.e., uses more than p pieces), a long checkout queue of nervous people will soon form.

Problem specification

In this problem we shall consider the Euro, as this is the currency used where Olivia lives.

The cash register holds an *infinite supply* of each of the following denominations: 1c, 2c, 5c, 10c, 20c, 50c, 1 Euro, 2 Euro, 5 Euro, 10 Euro, 20 Euro, 50 Euro, and 100 Euro. (The "c" denotes cents. There are 100 cents in 1 Euro.)

For the given sum s, find one way of paying s using at most p pieces of currency.

In the easy subproblem A1, p equals 10^6.
In the hard subproblem A2, p equals 200. (Using fewer pieces is harder!)
In both subproblems, each s will be between 0 and 100 Euro, inclusive.

Input specification

The first line of the input file contains an integer $t \leq 100$ specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of a single line containing two space-separated integers: e and c. The sum s for this test case is "e Euro c cents". You may assume that $0 \leq c \leq 99$.

Output specification

For each test case, output one line containing 13 space-separated integers n_1 through n_{13}, where each n_i represents the number of pieces of the i-th currency type in the list above. That is, n_1 is the number of 1c coins, ..., and n_{13} is the number of 100 Euro banknotes Olivia should use.

The total number of pieces of currency you use ($n_1 + \cdots + n_{13}$) must be less than or equal to p. Any such output that pays exactly the desired sum s will be considered correct.

Example

input

```
3

10 50

0 1

18 47
```

output

```
0 0 0 0 0 1 0 0 0 1 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 2 0 1 1 1 1 0 0 0
```

In the first test case, Olivia pays "10 Euro 50 cents" using a 10-Euro banknote and a 50-cent coin. In the second test case, she pays a single cent using a 1-cent coin. In the last test case, she pays the sum "18 Euro 47 cents" as 10 Euro + 5 Euro + 2 Euro + 1 Euro + 20 cents + 20 cents + 5 cents + 2 cents.

Note: Please do NOT submit any programs.
For each subproblem, just produce and submit a correct output file.

Task authors

Problemsetter:	Vlado 'Usamec' Boža
Task preparation:	Baška Klembarová

Solution

Hopefully, we managed to trick many experienced coders into implementing an algorithm that pays each sum using the smallest possible number of pieces of money.

This can be done in two ways: either by using dynamic programming, or by greedily taking always the largest denomination that does not exceed the remainder of the sum. The dynamic programming approach works for any set of denominations, the greedy algorithm does not – but luckily for Euros it does.

Anyway, both approaches were too complicated, our task was actually solvable in two lines of code.

In the easy subproblem p is so large that we can pay the entire sum using 1c coins. So we can just print $100e + c$ and then 12 zeros.

And the hard subproblem is also not that hard: the limits $s \leq 100.00$ and $p \leq 200$ allow us to pay all the Euros using 1 Euro coins, and all the cents using 1c coins. So we can just print c, then 5 zeros (for the other cent coins), then e, and then another 6 zeros (for the larger Euro coins and banknotes).

Problem B: Boredom buster

Gillian is normally a very lively child. Most of the time she plays with her friends and tries to indulge in some mischief. But today is different, today Gillian woke up with the flu so she has to stay in bed – still and bored. To entertain her, her brother came up with the following game.

When Gillian has an integer x greater than 1, she can split it up into two positive integers y and z such that $x = y + z$. After performing this operation, her brother gives her $y \cdot z$ hazelnuts. However, not all pairs of y, z are valid – there are some rules Gillian must comply with. These rules differ between the easy and hard subproblems; they are listed in the problem specification section.

Numbers that are obtained as a result of this operation can be also split up. At the beginning of the game, Gillian starts with a single integer n. She performs a series of operations described above until she is left with n copies of number 1. What is the maximum number of hazelnuts she can win if she chooses her moves wisely?

Problem specification – easy subproblem B1

For any $x > 1$ there is exactly one valid way of splitting:

- if x is divisible by 3, then $y = x/3$ and $z = 2x/3$;
- if x is divisible by 2 (but not by 3), then $y = z = x/2$;
- otherwise, $y = 1$ and $z = x - 1$.

Problem specification – hard subproblem B2

Gillian can pick any integer k satisfying $1 < k \leq x$ and split up number x into $y = \lfloor x/k \rfloor$ and $z = x - \lfloor x/k \rfloor$.

Input specification

The first line of the input file contains an integer $t \leq 1000$ specifying the number of test cases. Each test case is preceded by a blank line.

Each test case contains a single line with Gillian's initial integer $n > 1$. You may assume that $n \leq 10^6$ in the easy subproblem B1, and $n \leq 10^9$ in the hard subproblem B2.

Output specification

For each test case, output a single line with the maximum number of hazelnuts.

Example

input

```
1

5
```

output

```
10
```

This answer happens to be correct for both subproblems.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Peter 'Bob' Fulla

Solution

Solving the easy subproblem was a really straightforward task: One just needs to follow the deterministic splitting rule described in the problem statement.

We can store the integers Gillian currently has (other than 1) in almost any data structure (such as a queue or a stack). While the data structure is nonempty, we remove an integer x from it and split x according to the instructions. After that we insert the outcomes back into the data structure, and increment the number of hazelnuts by the appropriate amount.

Such an approach does not work for the hard subproblem – the initial integer n is too large. Moreover, there are too many valid ways of splitting an integer and we have to choose the best one carefully.

Or do we?

Actually, any sequence of operations yields the same result: $\binom{n}{2}$. And this is true for arbitrary splitting rules – those in the problem statement were just there to confuse you! Even if we were allowed to split any $x > 1$ into any $y, z > 0$ such that $x = y + z$, we would always collect the same total number of hazelnuts.

One way to prove it is by induction. However, a combinatorial proof is more interesting, as it can help us intuitively understand why the claim is true. Instead of an integer x imagine a complete graph on x vertices (K_x). When we split it up into K_y and K_z ($x = y + z$), we lose all the edges between the separated vertices. That is, we lose exactly $y \cdot z$ edges, and we get a hazelnut for each of them. At the end, we have n isolated vertices, so during the splitting we must have gotten rid of all the edges, and we now have a hazelnut for each of the $\binom{n}{2}$ edges of the original graph.

Problem C: Code Inception

In each subproblem of this task you are given a piece of code. Ultimately, the code produces (somehow, somewhen) a single readable English word.

Problem specification

Your only goal: recover the word.
And remember: you may need to go deeper.

Input specification

For each subproblem, you are given two files, each containing the same program, written in a different language. (One is in C++, the other in Python3. We did our best to make the programs as similar to each other as possible.) Each program produces the same single English word.

Difficulty is subjective. You may find subproblem C2 easier to solve than C1. But solving C1 is still worth 1 point and solving C2 is worth 2, just as in the other tasks.

Output specification

Submit a text file containing the recovered word, in UPPERCASE.

Example

input

```
for x in "olleh"[::-1]:
    print(x)

------------------------------


#include <iostream>
#include <string>
std::string s = "olleh";
int main() {
    for (auto c=s.rbegin();
         c!=s.rend();
         ++c)
            std::cout << *c;
}
```

output

```
HELLO
```

Note that the answer is given in uppercase.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek, Michal 'Žaba' Anderle

Solution

Both subtasks were actually solvable without fully understanding what the code computes. In the solutions below we go into more detail than was necessary.

Both subtasks feature code that runs for a very long time. Most of the solution lies in understanding how to compute the same thing faster.

Subproblem C1

About one half of the program is the mysterious `t` function:

```
def t(n):
    z=n
    for a in range(2,n):
        if t(a)>=a:
            if n%a==0: z//=t(a) ; z*=t(a)-1
    return min(z+1,n)
```

What does it compute? Well, first of all we can try running it. For example, we can add the following line: `for n in range(50): print(n,t(n))` and then we can run the program. We will get the following output (compressed into multiple columns to save space):

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

Afterwards, the computation starts to be quite slow – after all, the function `t` calls itself quite extensively. One way of speeding it up is obvious: as `t` is a function in the mathematical sense (its output only depends on its input), we can speed it up by adding memoization. This step was optional, and in this solution we will just skip it.

Instead, we can take a closer look at the code of `t`. There is one conditional: `if t(a)>=a`. When does this happen for small values of `a`? For the values we know this happens for 0, 1 (but we may ignore those, as the cycle for `a` starts from 2), 2, 3, 5, 7, 11, 13, 17, 19, and 23. And in all those cases we have `t(a)==a`. This brings us to a simple hypothesis: `t(a)>=a` is only true for 0, 1, and the primes, and for those inputs we always have `t(a)==a`. (If we added the memoization, we can now verify this hypothesis for more values of `a`.) Once we trust the hypothesis, we can now understand `t` as follows:

```
def t(n):
    z=n
    for each prime a that divides n: z = (z // a) * (a-1)
    return min(z+1,n)
```

The contestants more experienced in maths probably recognized this formula: this is basically Euler's totient formula (the number of integers that are smaller than n and relatively prime to n), only it's incremented by 1. In order to have a formally correct solution, we could now prove this by induction, and the hypothesis that `t(a)>=a` only for primes follows trivially.

(Note that we had two reasons for the +1 at the end. First, it makes the code a bit shorter. Second, thanks to the +1 the sequence of values of our `t` does not appear in the OEIS.)

But even if we have never heard of Euler's totient formula, we can now easily improve the efficiency of its implementation:

```
def prime_factors(n):
    answer = []
    tmp = n
    for a in range(2,n):
        if a*a > tmp: break
        while tmp % a == 0:
            answer.append(a)
            tmp //= a
    if tmp > 1: answer.append(tmp)
    return answer

def is_prime(n):
    return len( prime_factors(n) ) == 1

def t(n):
    z = n
    for a in set( prime_factors(n) ): z = (z // a) * (a-1)
    return min(z+1,n)
```

There is one last cheap trick to make the code run long: `for i in range(z): A = A[1:] + A[:1]`. In this line we rotate the (very short) array `A` awfully many times – as `z` is close to a billion. Obviously, this is useless, we can get the same result by only rotating the array `(z % len(A))` times. Once we make this final fix and execute the code, we get the final message:

```
CHANGE 1607055075 TO 853225920
```

Wait, what?

This is not the final message! We were promised a single English word, what is this? After the initial shock passes, we can notice that this is an instruction how to modify our program! The array `A` does actually contain the element `1607055075`. Let's change it as instructed and see what happens:

```
CHANGE 853225920 TO 646197696
```

Oh, okay. This might go on for a while. If we are lazy to apply the changes by hand, we can always write a shell script to do it for us:

```
#!/bin/bash
output=$( python3 c1.py )
echo $output
output=$( echo $output | grep CHANGE )
if [ "$output" == "" ] ; then exit ; fi
from=$( echo $output | cut -d ' ' -f 2 )
to=$( echo $output | cut -d ' ' -f 4 )
sed -e "s/␣$from,/␣$to,␣/" < c1.py > tmp ; mv tmp c1.py
```

Here is the list of all necessary changes (again, split into multiple columns):

```
CHANGE 1607055075 TO 853225920     CHANGE 60212160 TO 45005760     CHANGE 3779136 TO 2519424
CHANGE 853225920 TO 646197696      CHANGE 45005760 TO 33006528     CHANGE 2519424 TO 1679616
CHANGE 646197696 TO 443952576      CHANGE 33006528 TO 22006080     CHANGE 1679616 TO 1119744
CHANGE 443952576 TO 295970112      CHANGE 22006080 TO 16158528     CHANGE 1119744 TO 746496
CHANGE 295970112 TO 197318592      CHANGE 16158528 TO 10777536     CHANGE 746496 TO 497664
CHANGE 197318592 TO 135442368      CHANGE 10777536 TO 7978176      CHANGE 497664 TO 331776
CHANGE 135442368 TO 90310464       CHANGE 7978176 TO 5458752       CHANGE 331776 TO 221184
CHANGE 90310464 TO 60212160        CHANGE 5458752 TO 3779136       CHANGE 221184 TO 147456
```

And after we make the last change, suddenly the behavior of the program changes substantially, and it prints the desired password: `MATTER`.

Bonus questions: Where do all those letters come from? Is it possible to read the answer right at the beginning, without finding the value 147456? And what's up with the change in the last step, where did the `CHANGE` disappear?

Subproblem C2

Now what's going on here? First of all, this program clearly outputs a sequence of pairs of floating point numbers. Those can be points in the plane, or something like that, but where is the message? But let's not get ahead of ourselves.

Clearly, `n` is a simple generator of pseudorandom numbers and `s` is its state. In the main program we have an infinite loop that generates those pseudorandom numbers into `b`, and whenever `g(b)` is `False`, the variables `x,y` get changed. Once every 1000 steps we get the variables printed to the output.

Okay, so let's find the values of `b` for which `g` returns `False`. Again, the best way is just to guess this by running `g` for many inputs. For instance, if we run `g` for `b` up to a million, we will get a bunch of such inputs. And the largest one of them will be 438. And if we extend the range to a few more million, there won't be any new values. Randomly chosen `b` will also always return `True`.

Hypothesis: there are only very few values of `b` such that `g(b)` is `False`, and all of those are small. This hypothesis is actually easily verified by reading the implementation of `g`: large values of `b` lead to large values of `i`, and those point out of `foo` and into the string `HeHeHe`. Once `i` exceeds 145, the return values will be periodic, and as we can see, the string `HeHeHe` clearly makes all of those return values `True`.

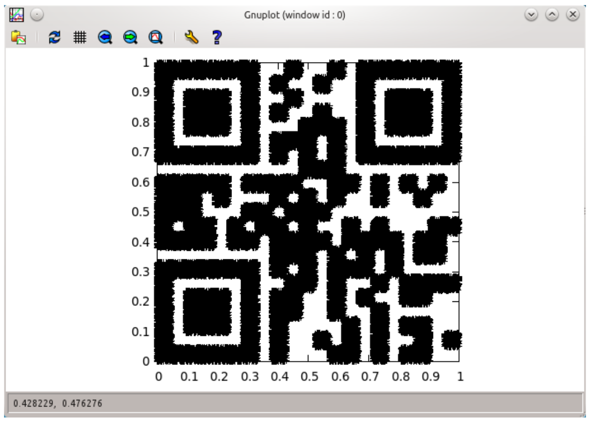
Conclusion #1: Consider the loop of 1000 iterations in the main program. Usually, none of those iterations will hit a good `b`, so there will be no change. Sometimes, if we are lucky, there will be one change, and very very rarely there may be two or more. So the output of our program will usually contain the same pair `x,y` many times.

Let's try speeding it up. We will use a standard random generator instead of `n`, we will only generate small values of `b` to increase the chance of hitting a good one, and we will output each new pair `x,y`.

```
x, y = 0., 0.
while True:
    b = randint(0,500)
    if not g(b):
        x, y = (b%21+x)/21, (20-b//21+y)/21
        print(x,y)
```

Yay, we have a steady stream of numbers! . . . and what now? Where is the answer?

We already mentioned the idea that the pairs `x,y` may be some points in the plane. Their names seem to agree, after all. Now that we have a lot of them, let's take a look at them! One simple tool that can be used to do this is `gnuplot`. Just generate a bunch of points (say 100k of them), save them into the file `data`, and then run `gnuplot` with the commands `set size square` and `plot 'data' linetype 0`. The output you should obtain is shown in the figure on the right.



Wow, that seems to be a QR code!
Maybe it hides our answer!

Nah, this is the Code Inception task. Scanning the QR code with your smartphone (or uploading it to an online service that reads QR codes) reveals the following text:

```
88767->7123398
```

And just as in C1, this is an instruction to modify the original program. In our case, we are supposed to change the constant `bar`. After we do so and repeat the same process, we get a new QR code, quite different from the previous one. Scanning this one reveals our final answer: `FOOTBALL`.

Bonus information: The values for which `g` is `False` actually represent the bitmap of the particular QR code, and they can be used to get its nicer version if the "impressionistic" one was causing you problems.

Note that even if we ran the program "infinitely long", the points it generates would never produce the exact QR code. This is because the picture generated by the program is *not* the full QR code. The picture is a *fractal*: each cell that should be black in the QR code actually contains a smaller copy of the fractal. See the Wikipedia entry "Iterated function system" for more information on such fractals.

Of course, the fractal is similar enough to the real deal, so it can be easily scanned.

Problem D: Do the grading

Did you take part in the practice session? If you did, you will be able to read this problem statement faster. But if you missed the practice session, don't worry, we will tell you all you need to know.

One of the practice tasks was the task *Rearranged alphabet*. In this task we asked the contestants to find a short string of lowercase letters such that each of the 26! permutations of `a` through `z` occurs in it as a (not necessarily contiguous) subsequence. For example, if the alphabet only consisted of `a`, `b`, and `c`, `abcabac` would be a valid answer, but `abccba` would not (both `bac` and `cab` is missing).

Solving the practice task was simple enough: you just have to find a pattern and print the corresponding string. On the other hand, *grading* the practice task is much more complicated: you have to read a string and check whether it actually contains all possible permutations.

Preparing the grader for the practice problem was quite fun. In fact, it was so much fun that we wanted to share it with you.

Problem specification

You are given a string of lowercase English letters.

In the easy subproblem D1, check whether each of 26! permutations of `a` through `z` occurs in the string as a subsequence.

In the hard subproblem D2, count the number of permutations of `a` through `z` that **do not** occur in the given string as a subsequence. As this number may be quite big, output it modulo 65 521.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of a single line containing a nonempty string of lowercase English letters.

In the easy subproblem D1, $t \leq 150$ and none of the strings is longer than 2 500 characters.

In the hard subproblem D2, $t \leq 20$ and the sum of lengths of all strings does not exceed 1 000.

Output specification

For each test case, output a single line with the answer.

That is, in the easy subproblem D1, output "`YES`" (without quotes) if the given string contains all permutations, or "`NO`" if it doesn't.

In the hard subproblem D2, output the number of missing permutations modulo 65 521.

Example

input

```
1

abc
```

In D2, note that 26! mod 65521 = 8297.

output for subproblem D1

```
NO
```

output for subproblem D2

```
8297
```

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Peter 'Bob' Fulla, Michal 'mišof' Forišek, Monika Steinová

Solution

Throughout this analysis we denote the set of lowercase English letters by Σ, its size by σ ($\sigma = 26$), the input string by w, and its length by n.

Easy subproblem

For every set of letters S ($S \subseteq \Sigma$), we compute the length of the shortest prefix of w such that each of the $|S|!$ permutations of letters from S occurs in this prefix as a subsequence. We denote this value by $f(S)$. Clearly, string w contains all permutations of Σ iff the value of $f(S)$ is defined for all S (i.e. the required prefix always exists).

For the empty set, the empty prefix satisfies the requirement, therefore $f(\varnothing) = 0$. If the values of $f(S')$ for all $S' \subsetneq S \subseteq \Sigma$ are already known, we can proceed to compute the value of $f(S)$. Let us pick a letter $a \in S$ and focus on the permutations of S that end with a. Now it is easy to determine the shortest prefix containing all such permutations – it must cover the prefix of length $f(S \setminus \{a\})$ and the first occurrence of a coming next. (If there is no occurrence of a following the prefix $f(S \setminus \{a\})$, the string w does not meet the requirements.) We try all letters $a \in S$ and assign to $f(S)$ the maximum of the prefix lengths we obtain.

To speed up the computation of f, we precompute for every position in w and every letter of Σ its first occurrence following the position. This can be easily done in time $O(\sigma n)$. After that we can compute the values of f in time $O(\sigma 2^\sigma)$.

Hard subproblem

We take a similar approach to solve the hard subproblem. Let us denote by $g(S, \ell)$ (for $S \subseteq \Sigma, 0 \leq \ell \leq n$) the number of permutations of letters from S that do not occur in the first ℓ characters of w as a subsequence. Hence the answer to our problem is the value of $g(\Sigma, n)$.

Again, we first compute the values of g for smaller arguments and then proceed to larger ones. To determine the value of $g(S, \ell)$, we sum the numbers of missing permutations ending with a for all $a \in S$. For a fixed a, the number of missing permutations is equal to $g(S \setminus \{a\}, i)$, where i is the last occurrence of a in the prefix ℓ.

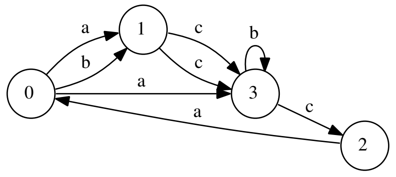
This solution runs in time $O(\sigma n 2^\sigma)$, which is quite a lot. However, our main problem are the memory requirements: To store all values of g we need about $100 \cdot 2^{26} \cdot 2\text{B} = 12.5\text{GB}$ of RAM on the longest input string. Fortunately, many of the values can be discarded during the computation – we use $g(S, \ell)$ only when ℓ is the last occurrence of some letter a and $a \notin S$. With this optimization we need only $26 \cdot 2^{25} \cdot 2\text{B} = 1.625\text{GB}$ of memory. Solving the entire hard input took us about one hour on a single processor core, but one can solve different test cases on different computers/cores in parallel and be done in a few minutes.

Problem E: Exploring the cave

After the success of "open sesame!", Ali Baba experimented with various other crops. Most of them didn't do anything out of the ordinary, until suddenly "open sugarcane!" caused one of the rocks to shift and reveal the entrance to a peculiar cave.

The cave consisted of several chambers. The entrance lead directly into one of these chambers, we will call it the starting chamber. Some pairs of chambers were connected by *one-way* tunnels. Each of the tunnels was of one of three types: some tunnels had abrasive walls, others had battered walls, and the rest had calcified walls. As you have probably already guessed, we will denote the tunnel types `a`, `b`, and `c`.

For any chamber, there could have been arbitrarily many tunnels entering it, and arbitrarily many tunnels leaving it – including multiple tunnels of the same type, or no tunnels at all. Also, there could have been tunnels that start and end in the same chamber.



An example of a cave with 4 chambers and 8 tunnels.

Of course, it's not really a good idea to explore a cave with one-way tunnels on your own. Luckily, Ali Baba can enlist the help of the forty thieves (and their infinitely many friends, if necessary). One round of cave exploration looks as follows:

1. Ali Baba chooses a finite (possibly empty) sequence of tunnel types (a string of letters).
2. One after another, the thieves repeat the following procedure:
   (a) The thief takes a long piece of rope and fastens one of its ends to his waist.
   (b) He enters the starting chamber.
   (c) He tries to follow a sequence of tunnels that 1. corresponds exactly to the sequence of types selected by Ali Baba, and 2. has not been traveled (as a whole) by any of the previous thieves.
   (d) If successful, the thief remains waiting in the final chamber reached by his walk. (We assume that each chamber is large enough to accommodate all the thieves that end their walks there.)
3. As soon as a thief is unable to perform his task (each possible sequence of tunnels has already been traversed by someone), the exploration round stops. The last, unsuccessful thief is removed from the cave – Ali Baba uses the thief's rope to pull him out.

   At this moment, consider the set of chambers that contain at least one thief. The set of chambers will be called *significant*. (Note that sometimes the significant set may even be empty.)
4. Ali Baba uses the ropes to pull all the thieves out of the cave.

Of course, different choices of the sequence in step 1 can lead to different significant sets of chambers in step 3. Consider the example above. If Ali Baba chooses the sequence `ac`, he will discover the significant set $\{2, 3\}$: there will be one thief going $0 \to 3 \to 2$ and two other thieves going $0 \to 1 \to 3$ (each of these two using a different tunnel to get from 1 to 3). The sequence `bcb` produces the significant set $\{3\}$, the empty sequence produces the significant set $\{0\}$, and the sequence `ccc` produces an empty significant set.

Problem specification

You are given the total number n of chambers in the cave. Ali Baba has also told you that they tried to explore the cave using all possible sequences of tunnel types (even though there's infinitely many of them!) and that they were able to find exactly d different significant sets of chambers.

Find whether such a cave system exists. If yes, find one example.

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of a single line with the two numbers: n and d.

In the easy subproblem E1, $1 \leq n \leq 10$ and $1 \leq d \leq 100$.

In the hard subproblem E2, $1 \leq n \leq 22$ and $1 \leq d \leq 10^9$.

Output specification

For each test case, there are two possible outputs.

If there is a cave with the given parameters n and d, output the description of one cave as a sequence of tunnels. In the first line, output the number $m \leq 5000$ of tunnels in your cave. (If there is a valid cave, there is always one with much less than 5000 tunnels.) In each of the following m lines, output one tunnel in the form "x y z", where x is the chamber where the tunnel starts, y is the chamber where it ends, and z is one of `a`, `b`, and `c`. (The chambers in the cave are numbered 0 through $n-1$, where chamber 0 is the starting chamber.)

If there is no such cave, output a single line with the integer -1 instead.

You may output additional whitespace. (Note that we do so in the example output for clarity.)

Example

input

```
3

2 3

4 7

1 100
```

In the first test case the cave we produced has three significant sets of chambers: $\emptyset$, $\{0\}$, and $\{1\}$.
In the second test case our answer is the cave shown on the previous page.
In the third test case it is obvious that there is no such cave.

output

```
1
0 1 a

8
0 1 a
0 1 b
0 3 a
1 3 c
1 3 c
3 3 b
3 2 c
2 0 a

-1
```

Task authors

Problemsetter: Michal ‘mišof’ Forišek
Task preparation: Michal ‘mišof’ Forišek

Solution

The first crucial observation: each significant set of chambers is a subset of the set of all chambers. Therefore, for a cave with n chambers, there can be at most 2^n significant sets. As we show below, this necessary condition is also sufficient: whenever $d \leq 2^n$ there will be a solution.

Counting the significant sets

Instead of doing the cave exploration as described in the problem statement, we can do it incrementally.

Imagine the starting chamber full of thieves. Now, Ali Baba announces a letter x, each of the thieves chooses a tunnel with that letter, and follows it to its destination. What have we obtained? The thieves now occupy precisely the significant set of chambers that corresponds to the letter x. Next, Ali Baba announces another letter y and again each of the thieves moves. Once the thieves move for the second time, we now have the significant set of chambers that corresponds to the string xy. (Of course, assuming that there are enough thieves to cover all possibilities.) And so on.

When doing the above process, moving the thieves one tunnel at a time, it is easy to note that each combination (current set of occupied chambers, the next letter) uniquely determines the next set of occupied chambers.

We can use this observation to construct all reachable significant sets incrementally. At the beginning, we know that the set $\{0\}$ is significant – it corresponds to the empty string. Now, we can construct (up to) three other significant sets: the set reached from $\{0\}$ by saying the letter `a`, the set reached for `b`, and the set reached for `c`. Then, for each of these sets we can repeat the same process (skipping sets we already discovered), and so on.

Effectively, we are discovering all the reachable significant sets using a breadth-first search.

The time complexity of this algorithm is $O(n2^n)$ – exponential in n, but significantly better than “try all possible strings” (there’s infinitely many of those, after all).

Note that we used this algorithm to check the correctness of your submissions. And if you discovered it, you could use it, too.

Easy subproblem

Given the above algorithm, we can try to get a feeling about the problem. A simple way of discovering how it all works: generate a bunch of random caves and for each of them compute the number of significant sets. And surprise: you will soon discover that you are seeing examples for all possible $d \leq 2^n$.

Indeed, the easy subproblem can be solved simply by generating enough random caves and picking the ones we want. It turns out that for each solvable (n, d) given in the easy subproblem there is plenty of such caves, so if we look long enough, we are bound to run into one.

Hard subproblem

The more experienced contestants probably saw through the disguise and rephrased the question in terms of automata: Find a nondeterministic finite automaton (NFA) with n states such that the equivalent deterministic automaton (DFA) constructed via the subset construction has exactly d reachable states.

It turns out that for a ternary alphabet each (n, d) such that $1 \le d \le 2^n$ has a solution. There is also a significantly stronger result: for each (n, d) such that $n \le d \le 2^n$ there is a *minimal* NFA with n states such that the corresponding DFA constructed via the subset construction is also minimal and has d states. (Note that when considering minimality we also have to take accepting states into account, which we won't do in our problem.)

One possible construction is given in the paper G. Jirásková: Magic Numbers and Ternary Alphabet (2009). Her construction also has the minimal-FA property described above. We are too lazy to include the construction here :)

Problem F: Feeling lucky?

Last year the IPSC was so successful that we earned n coins. And they are no ordinary coins: they are perfectly identical coins made of solid gold.

Sadly, there are some problems with our coins. First of all, we don't actually have them. The coins are locked in a vault in Absurdistan. And second, we just got word that one of our coins has been stolen and replaced by a fake one. The fake coin slightly differs from the real ones in weight, but we do not know whether it is heavier or lighter.

So far, our situation looks like one of those weighing puzzles, doesn't it? We bet you would love to take balance scales and start comparing the weights of some coins in order to identify the fake one as quickly as possible.

Well, it kind of does look like a weighing puzzle, but the weighing part is not the major issue here. Remember that all our coins are in a country far far away? We will not be the ones weighing the coins, we can only send our request to the natives.

Why does this change anything, you ask? Well, for a start, there is no Internet in Absurdistan. Each time we want to make some weighings, we write down a list of requests, send it by regular post and wait a week or so for the answers.

To add insult to injury, the natives in Absurdistan are very lazy. Each time a native is asked to weigh some coins, with probability $p = 0.7$ he will ignore the request and just give you a random answer instead. That is, only 30% of your requests will actually be executed, the other 70% will receive random answers. On the other hand, the natives are precise. If the native decides to perform the requested weighing, he will always get and report the correct result.

The scales used by the natives are extremely precise balance scales. They consist of two pans (we will call them "left" and "right") that are connected by a beam with a fulcrum in the middle. One may place some coins into the left pan, some other coins into the right pan, and then read off one of three possible results: either one of the pans is heavier, or the scales balance. (It only makes sense to place the same number of coins onto each pan. If you ask to place more coins into one pan than the other, the pan with more coins will always be heavier. Of course, even if this is the case, the native may still skip the weighing and report a random answer.)

Problem specification

There are n coins, labeled 1 through n. The labeling was chosen uniformly at random. Out of the coins, $n - 1$ are real and one coin is fake. The fake coin is either lighter or heavier than the real coins. As the counterfeiter was trying his best, both options are equally likely (probability 50%).

You will interact with our grader using multiple submissions. Each submission represents one letter sent to Absurdistan. You will first send some letters that require the natives to perform some weighings, and finally one letter announcing *which coin is the fake one and also whether it is heavier or lighter than the real ones.* In each letter in which you request weighings, you must request exactly k of them.

In each subproblem of this task there are two criteria you need to satisfy in order to solve the subproblem:

- You may send at most s letters. (That is, you may make at most $s - 1$ submissions, each requesting k weighings, and then you must submit your answer.)
- Are you feeling lucky? Well, today your luck has run out. If you are thinking about just taking a random guess, you can forget it right now.

  Your answer will only be accepted if *you can be at least 99% certain that your answer is correct*, based on the weighings you made, their outcomes you received, and the assumption that all random events were independent and had the stated probabilities.

If you fail (by guessing incorrectly, guessing without 99% certainty, or using up all s submissions without guessing at all), **the whole subproblem is restarted** and you can try again from the beginning. A new fake coin in chosen, and you get to make another s submissions in the "new game".

Constraints

In both subproblems, the probability of a particular native being lazy is $p = 0.7$. (That's a lot!)

Easy subproblem F1: The number of coins is $n = 81$. You may send at most $s = 6$ submissions (per restart), and in each submission that requests weighings, you have to request exactly $k = 50$ of them.

Hard subproblem F2: The number of coins is $n = 250$. You may send at most $s = 11$ submissions (per restart), and in each submission that requests weighings, you have to request exactly $k = 15$ of them.

A different rule replaces the standard limit of at most 10 submissions per subproblem. Here, only *Wrong answer*s count towards the limit. In each subproblem, you may only receive a *Wrong answer* message at most 9 times. If you receive a 10th *Wrong answer*, all further submissions will be rejected.

Submission specification

The first line of your file should contain a single letter: either '`G`' (a guess) or '`W`' (a list of k weighing requests).

If your submission is a guess, the second line of your submission should contain the number of the fake coin (between 1 and n, inclusive), a space, and a letter. The letter should be '`L`' if the fake coin is lighter than the real ones, and it should be '`H`' if the fake coin is heavier.

If your submission is a list of k weighing requests, it should contain exactly k more lines. Each of those lines should contain a string of n characters that describes one weighing request. The i-th character of a request should be '`L`' if coin i should be placed on the left pan of the scales, '`R`' for the right pan, and '`-`' if the i-th coin should remain off the scales.

Evaluation result specification

If your submission is syntactically incorrect, you will receive a *Wrong answer* with an explanation. A syntax error does not cause a restart, nor is it counted in the s allowed submissions.

If your submission is a successful guess, you will receive an *OK*, thus solving the subproblem. If your guess fails, you will receive a *Wrong answer* with an explanation, and the game is restarted.

If your submission is a list of weighing requests, you will receive a *Continue* and a string of k characters. The i-th character is the result of your i-th weighing request: '`L`' if the native claims the left pan is heavier, '`R`' if he claims the right pan is heavier, and '`=`' if he claims both pans are exactly equally heavy.

Note that each request on your list is handled by a different native, and (independently of each other) each of those natives generates his reply uniformly at random with probability p.

Continue messages do not affect the team's rank. They are not worth any points, nor do they add penalty time. *Wrong answer*s are scored as usual.

Good advice

If at first you don't succeed, try again!

As there are probabilities involved, even the best strategy might sometimes fail. If you trust that your game strategy makes sense, give it another attempt if the first one doesn't make it.

(Of course, if your strategy is bad, your chance to solve this problem would be zero even if we granted you a thousand attempts.)

Example

In the example below, we have $n = 6$ coins and we request $k = 3$ weighings at a time.
One possible first submission:

submission

```
W
LLRR--
-L---R
--LRLR
```

This is what you may receive as our response:

response

```
Continue: LL=
```

This means that you got the following responses:

- Coins 1+2 are heavier than coins 3+4.
- Coin 2 is heavier than coin 6.
- Coins 3+5 are exactly as heavy as coins 4+6.

If we trusted these answers, we could now conclude that coin 2 is the fake one, and it is heavier than the real coins. (Coins 3, 4, 5, 6 have to be real from the third answer, and then coin 2 is fake and heavier from the second answer.) We could then submit the corresponding guess:

submission

```
G
2 H
```

However, in this problem we have to be certain enough before making our guess.
And right now we shouldn't be too certain yet. After all, each of those three responses has probability 70% of being the result of a random choice. The submission would be evaluated as a *Wrong answer*, and the subproblem would be restarted.
At the moment (assuming $p = 0.7$) the actual probability that "`2 H`" is the correct answer is only about 36.96%. The second most likely answer is currently "`1 H`" with probability about 16.17%. (The answer "`1 H`" corresponds, among other possibilities, to the situation when the response to `-L---R` was generated at random and the other two are correct.)

Task authors

Problemsetter:	Monika Steinová
Task preparation:	Monika Steinová, Michal 'Žaba' Anderle, Michal 'mišof' Forišek

Solution

We will first show how to solve the easy subproblem without the probabilistic part, then we'll deal with the probability. Afterwards, we will show a better solution that is also able to solve the hard subproblem.

Deterministic weighing

For simplicity, let's first try to solve an exact version of the easy task: i.e., the version where we make weighing requests sequentially, and each of them will be answered correctly.

We will now show a simple way how one can solve the problem for $n = 3^q$ coins using $q+1$ weighings.

Divide the coins into three piles (A, B, and C), each containing $n/3 = 3^{q-1}$ coins. We will use the first two weighings to determine two things:
– which of the piles contains the fake coin,
– whether the fake coin is lighter or heavier than the real ones.

This can be done, for example, by first comparing A and B and then comparing A and C.

(If A and B are equally heavy, the fake coin is in C and the second weighing tells us whether it is heavier or lighter than the real ones. If A and B differ, C coins are real and the second weighing tells us whether A or B contains the fake coin. From the result of the first weighing we can then also deduce whether the fake coin is heavier or lighter.)

Now we have 3^{q-1} coins. One of those coins is fake and we know whether it is lighter or heavier than the real ones. It is now easy to find the fake coin in $q-1$ weighings.

The above solution is sufficient for our purpose, but it is not optimal. Optimally the problem can be solved in 3 questions for $n = 12$ and thus in $3+x$ questions for $n = 12 \cdot 3^x$.

Easy subproblem

Now, let's get back to the randomized version. We have $n = 81 = 3^4$ coins, so in the deterministic setting we could solve it in 5 weighings. Conveniently, in the easy subproblem we can send $s-1 = 5$ letters. We can now simply use majority voting: In each letter, request k copies of the same weighing. The outcome will be the answer you receive the most often.

The constraints were set so that in $\sim 80\%$ of your attempts this would give you the required certainty that your answer is correct.

Hard subproblem

The hard subproblem is not solvable with the above strategy. (Or, more precisely, its success probability is very very low). Below we present a different approach that could also be used to solve the easy subproblem.

If we know nothing about the fake coin, the best questions we can ask are precisely the questions we used in the first solution: if you divide the coins into three equally large piles and weigh two of them, each of the answers you might receive has probability $1/3$. (The answer to this question will give you the most information you can get by asking a single question.)

A good strategy to solve the hard subproblem is to use the above observation and randomization. The strategy: Send $s-1$ letters, each requesting a set of k random weighings, each of the above type. In the final round submit the answer that is consistent with most of them.

The parameters in the hard subproblems were set so that with this approach in $\sim 85\%$ of your attempts you would reach the 99% certainty that your answer is correct.

Beyond the hard subproblem

Why does the above strategy work? How are the solutions evaluated? And is there an even better strategy? These are the questions we will now answer.

Our grader uses Bayesian inference to derive the probabilities. At the beginning, the probability of each answer is $1/(2n)$. Observing the result of each weighing changes these probabilities.

More precisely: Let $P(c, l, x)$ be the probability that the coin c ($1 \leq c \leq n$) is the fake and has weight l ($l \in \{H, L\}$) after processing x weighing requests. Initially, we set $P(c, l, 0) := 1/(2n)$ for all (c, n). The values $P(c, l, x+1)$ are then computed from the values $P(c, l, x)$ as the appropriate conditional probability. $P(c, l, x+1) = Pr[(c, l)$ is the answer $|$ we saw the answer of weighing $x+1$ given the distribution $P(c, l, x)]$.

The same Bayesian inference can be used in our solution to (slightly) increase its chance of success. When sending the later letters, we will not choose the weighings at random. Instead, we will choose weighings that maximize the expected amount of information we'll get from them. (Or, as an easier alternative, we can always generate a lot of random requests, for each of them we can compute the expected amount of information, and then ask the k best ones.)

Our solution that used this approach could reach the required 99% certainty in $\sim 92.5\%$ of runs.

Problem G: Grid

You are given a rectangular grid consisting of $r \times c$ points. The lower left corner has coordinates $(1,1)$, the upper right corner has coordinates (c,r). The neighbors of a point (x,y) are the points $(x-1,y)$, $(x+1,y)$, $(x,y-1)$, and $(x,y+1)$, if they exist. A path is a sequence of points such that subsequent points are neighbors and each point appears on the path at most once.

Problem specification

Given two distinct points (x_s,y_s) and (x_f,y_f), find one longest path from (x_s,y_s) to (x_f,y_f).

Input specification

The first line of the input file contains an integer t specifying the number of test cases. Each test case is preceded by a blank line.

Each test case consists of a single line with six integers – r, c, x_s, y_s, x_f, y_f.

For both the easy subproblem G1 and the hard subproblem G2, you may assume that $1 \le r, c \le 100$ and $rc \ge 2$. Additionally, for the easy subproblem G1 you may assume that $r \le 5$.

Output specification

For each test case, output a single string describing one possible longest path. If there are multiple longest paths, output any one of them.

A path $a_1, a_2, \dots, a_k$ is described by a string consisting of $k-1$ letters U, D, L, R. The i-th letter in the string describes the move from point a_i to a_{i+1}:

If $a_i = (x,y)$ and $a_{i+1} = (x,y+1)$, the i-th letter should be U.
If $a_i = (x,y)$ and $a_{i+1} = (x,y-1)$, the i-th letter should be D.
If $a_i = (x,y)$ and $a_{i+1} = (x+1,y)$, the i-th letter should be R.
If $a_i = (x,y)$ and $a_{i+1} = (x-1,y)$, the i-th letter should be L.

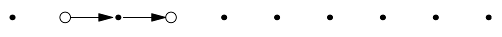
Example

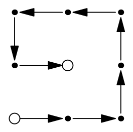
input

```
2

1 10 2 1 4 1

3 3 1 1 2 2
```

output

```
RR
RRUULLDR
```

first example: 

second example: 

Task authors

Problemsetter: Vlado ‘Usamec’ Boža
Task preparation: Vlado ‘Usamec’ Boža, Jano Hozza

Solution

Let’s assume without loss of generality that $r \leq c$. We will first calculate upper bound for the length of path. Denote color of point (x, y) as $(x + y)$ `mod` 2. If rc is even, then there is same number of points with color 0 as with color 1. If rc is odd, then we have one more points with color 0 than with color 1.

Now consider any path. Let s, f be the endpoints of the path. The path altenates between points with color 0 and color 1. Now we can derive following upper bounds on the length of the path:

- If rc is even and s, f have different color, then the upper bound for the length of the path is rc.
- If rc is even and s, f have same color, then the upper bound for the length of the path is $rc - 1$.
- If rc is odd and s, f have color 0, then the upper bound for the length of the path is rc.
- If rc is odd and s, f have color 1, then the upper bound for the length of the path is $rc - 2$.
- If rc is odd and s, f have different color, then the upper bound for the length of the path is $rc - 1$.

There are few more special cases when $r = 1, 2, 3$. More specifically:

- If $r = 1$, then the length of path is $|x_s - x_f| + 1$.
- If $r = 2$, $y_s \neq y_f$, and $|x_s - x_f| \leq 1$, then the length of path is: $\max(x_s + x_f, 2c - x_s - x_f + 2)$.
- If $r = 3$ and c is even and one of the endpoints has color 0 and the latter has color 1. We can assume, that s has color 1 and f has color 0. If at least one of the following holds, then the upper bound for the length of path is $rc - 2$:
  - $y_s = 2$ and $x_s < x_f$.
  - $x_s < x_f - 1$.

  If previous conditions hold when we change coordinates of endpoints from $(x_s, y_s), (x_f, y_f)$ to $(c - x_s + 1, y_s), (c - x_f + 1, y_f)$, then the upper bound for the length of the path is $rc - 2$.

Given r, c, x_s, y_s, x_f, x_f denote $U(r, c, x_s, y_s, x_f, y_f)$ the upper bound on the length of the path in given grid and endpoints calculated by the conditions above. It can be proved that there always exists a path with length given by this upper bound.

Now let’s find this path. If rc is sufficiently small (e.g. less than 30) we can find a path using brute-force. If rc is big, we can use following tricks:

We can find c_1, r_1 such that $x_s \leq c_1$, $x_f > c_1$ and $U(r, c, x_s, y_s, x_f, y_f) = U(r, c_1, x_s, y_s, c_1, r_1) + U(r, c - c_1, 1, r_1, x_f - c_1, y_f - c_1)$. This means that we split rectangle into two by some vertical line. The endpoints are in different rectangles. Then we try to find path from (x_s, y_s) to (c_1, r_1) and then path from $(c_1 + 1, r_1)$ to (x_f, y_f) such that these paths do not cross the splitting line. We will call this process a split. We can also split rectangle by some horizontal line.

The other process is stripping. We will again split our rectangle into two rectangles A, B by some vertical line. But now we will require the both endpoints to be in A and B to have even number of columns. Note that B contains Hamiltonian cycle. Denote the number of columns of A as c_2. Then if $U(r, c, x_s, y_s, x_f, y_f) = U(r, c_2, x_s, y_s, x_f, y_f) + (c - c_2) * r$ we can find the path by first finding path in the rectangle A and then expand it to go through rectangle B.

So we can recursively divide our rectangle into smaller ones and then merge path into final solution. This algorithms is adapted from algorithm given by Keshavarz-Kohjerdi et al.[2].

[2]Keshavarz-Kohjerdi, Fatemeh, Alireza Bagheri, and Asghar Asgharian-Sardroud. ”A linear-time algorithm for the longest path problem in rectangular grid graphs.” Discrete Applied Mathematics 160.3 (2012): 210-217.

Problem H: Histiaeus

Sometimes, you need to send someone a message without anyone knowing about the message's existence. This is the general principle of steganography. One of the early users of steganography was an Ancient Greek ruler named Histiaeus.

Histiaeus needed to send a secret message to Aristagoras, but worried that the slave carrying the message would be intercepted. So the slave was given some innocent letters to fool the enemy spies, but also carried another message, hidden in a clever manner devised by Histiaeus. The enemy didn't notice the secret message was there, but Aristagoras knew how to find it.

Inspired by Histiaeus, we decided to send you a secret message, hiding it the same way he did. We will play the role of Histiaeus, you'll be Aristagoras, and the problem statement of *Problem H – Histiaeus* is the slave we sent to you, carrying the secret message.

(However, *this* problem statement – the one that you are reading right now – does not hold any secrets. Searching for the secret message here would be a waste of time. Now, where else could it be?)

Problem specification

Do what Aristagoras did, and find the secret message.

Input specification

There is no input.

Output specification

For both subproblems, the correct output is a single English word written in UPPERCASE. The secret message will tell you which word it is.

Task authors

Problemsetter:	Tomi Belan
Task preparation:	Tomi Belan, Martin 'Imp' Šrámek

Solution

To hide the secret message from his enemies, Histiaeus shaved the head of his slave, tattooed the message on his head, and sent the slave to Aristagoras after his hair grew back.

According to the problem statement, we hid the message the same way Histiaeus did, and the problem statement itself is the slave. And really, its HTML `<head>` contains the following:

```
<link rel="secret message" type="text/tattoo"
    href="tattooed by Histiaeus on the slave's head">
<meta name="congratulations" value="that was easy, now find H2">
<meta name="H1 answer" value="SUBROUTINES">
```

So the correct output for H1 is `SUBROUTINES`.

(The PDF version of the problem statement says something slightly different from the online version. In the online version, "this problem statement" is the slave. In the PDF version, "the problem statement for H" is the slave, but not "the one you are reading right now".)

This only gives us the answer to H1. Where could H2 be hidden? Is there another secret message? Well, the problem statement has more than one "head". At the next level below HTML, there are the headers of its HTTP response:

```
H2: A blue monster approaches you and says: "I know the word you need for H2.
    But I won't talk to you unless you feed me first. I'm in the mood for some
    dessert... its name doesn't matter, but it must contain chocolate."
```

What kind of dessert is related to HTTP, has a name, contains something, and has something to do with a blue monster? It's an *HTTP cookie.* (The blue monster is a reference to Cookie Monster of *Sesame Street* fame.)

A simple way of creating the needed cookie is to enter `document.cookie = "a=chocolate"` in the browser's JavaScript console. Another is to use `curl` with the `-b a=chocolate` switch.

After the cookie is sent to the server, the HTTP headers change:

```
H2: "Yum! All right, if you answer this question, and do it quickly,
    I'll tell you the answer for H2."
H2-Question: How much is 127 plus 486?
H2-Answer: (your answer)
```

The final phase requires you to send your answer in a custom request header named `H2-Answer`. The appropriate request can be sent with `curl` like this:

```
curl .../h.html -H "H2-Answer: 613" -D result; cat result
```

If this is done quickly enough after the last request, the monster tells you the correct output:

```
H2: "Correct! The word you're looking for is CLASSIFICATION."
```

Problem I: Invisible cats

You are given a number of small grayscale images, each exactly 32 × 32 pixels in size. The images have been encrypted. The encryption is different for the easy and the hard subproblem. Both encryption types are described below.

There is a cat in the 21st image. There are also exactly ten other images with cats among the first 20 images. Find those ten cats!

Problem specification – easy subproblem

The pictures are encrypted in the following way: We picked a single random permutation on 32 elements. Then, for each picture we shuffled its columns using this permutation. That is, each column of pixels is still in its original order from top to bottom, but the order of columns is now different. (Note that we used the same permutation for all pictures.)

Problem specification – hard subproblem

The pictures are encrypted in the following way: We picked a single random permutation on 32 × 32 elements. Then, for each picture we shuffled its pixels using this permutation. That is, the set of pixels is now the same, they are just in different locations. (Again, note that we used the same permutation for all pictures.)

Input specification

For each subproblem you are given one set of encrypted images. Each set of images is provided in two different formats:

The first format is a ZIP archive that contains each encrypted image as a separate PGM file.

The second format is a single file that is formatted as follows: The first line contains a single integer: the number of images. For each image, you are then given 1024 integers, each in range from 0 (black) to 255 (white). The first 32 of these integers are the colors of the first row (left to right), the next 32 is the second row, and so on.

Output specification

Print ten whitespace-separated integers – the numbers of first ten pictures with cats, in ascending order. Do not use leading zeroes, even though the filenames in the ZIP archive have them.

Example

input	output
(a bunch of pictures)	`1 2 3 4 5 6 7 8 9 20`

Task authors

Problemsetter:	Vlado 'Usamec' Boža
Task preparation:	Vlado 'Usamec' Boža, Peter 'Bob' Fulla

Solution

Solving the easy subproblem was quite an easy task. You could either sort the columns by hand, or you could find some similarity measure for columns and rearrange them in a way that puts similar columns close to each other. One example of a good similarity measure is the sum of all differences between neighboring pixels.

The general idea for solving the hard subproblem is rather trivial: Find a measure of quality for decrypted images. Then, find a permutation of pixels that maximizes the quality of images.

Still, the details of this approach are quite tricky. Our quality measure for images was the sum of correlations between neighboring pixel values. For sufficiently large sets of reasonable images, maximizing this value should leads to perfect decryption. The biggest problem is finding a permution of pixels which maximizes this value. This problem is NP-hard (as finding shortest Hamiltonian path can be reduced to this problem). But real problem solvers cannot be stopped by some NP-hardness. We can for example try a hill-climbing approach. Let's start with some permutation of pixels. And then we can improve it by swapping pixels. If we discover that we are getting stuck in local optima of the value function, we can attempt fixing it by adding more transformations to the hill-climbing algorithm – like swapping bigger parts of the image, cyclic rotation of rows and colums, reversal of rows and columns, etc.

In our sample solution we find the best value by using simulated annealing.

Problem J: Just a single gate

Surely you have heard about logic gates, such as AND, NOT, XOR, and many others. A logic gate is a tiny device with some inputs and outputs that implements a Boolean function. That is, the inputs and outputs are boolean values (0 or 1), and for each particular gate each output is uniquely determined by its inputs.

For example, the traditional $NAND$ gate has two inputs (let's call them x and y) and one output (z). All possible outputs of this gate are given in the truth table shown below. This gate computes the "not and" function: the output is true if and only if the logical "and" of both inputs is false.

```
NAND:  x | 0 0 1 1
       y | 0 1 0 1
      ---+---------
       z | 1 1 1 0
```

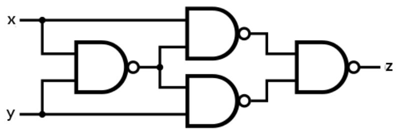
It is a well-known fact that the $NAND$ gate is *universal*: You can construct any other gate using only a finite set of suitably interconnected $NAND$s.

For example, consider the unary NOT gate – a gate that outputs 0 if the input is 1, and vice versa. This gate can be constructed using a single $NAND$ gate: $NOT(x)$ is the same as $NAND(x, x)$.

Of course, sometimes the construction is more involved. For example, to construct the binary XOR gate (a gate that returns 1 if the inputs are different and 0 if they are equal) we need at least four $NAND$ gates. One possible construction:

$$XOR(x, y) = NAND(\ NAND(x, NAND(x, y)),\ NAND(y, NAND(x, y))\ ).$$

The above expression has five $NAND$s, but $NAND(x, y)$ occurs twice, and can be implemented by a single gate in hardware, as shown in the figure below.



There are no universal unary gates, and only two universal binary gates: the gate $NAND$ described above, and the gate NOR that implements the Boolean function "not or".

Problem specification

In this problem we are interested in ternary gates: gates with three inputs and one output. An example of a ternary gate is the MAJ gate that returns the majority element – i.e., it returns 1 if at least two inputs are 1, and 0 if at least two inputs are 0. Below is the truth table of MAJ with inputs w, x, y and output z:

```
MAJ:  w | 0 0 0 0 1 1 1 1
      x | 0 0 1 1 0 0 1 1
      y | 0 1 0 1 0 1 0 1
     ---+-----------------
      z | 0 0 0 1 0 1 1 1
```

Easy subproblem J1: Find **at least seven** universal ternary gates.

Hard subproblem J2: Find **all** universal ternary gates.

Input specification

There is no input.

Output specification

Each line of your output file must describe a universal ternary gate, written as a space-separated list of zeroes and ones: the output row of its truth table, in order.

For the easy subproblem, the output must contain *at least* seven distinct rows, for the hard subproblem it has to contain all universal ternary gates. (I.e., any correct output for the hard subproblem will also be accepted if you submit it as your answer to the easy subproblem.)

Example output

```
0 0 0 1 0 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1
```

This is a syntactically correct output file. It describes three ternary gates: the first row is MAJ, the second row is a gate that always returns 0, and the third row is the $AND3$ gate (ternary and): its output is the logical "and" of all three inputs.

(This is an incorrect output: it contains too few gates, and the gates it contains are not universal.)

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'mišof' Forišek, Peter 'Ppershing' Perešíni

Solution

The easy subproblem was solvable by hand. To solve the hard subproblem, you could generate all possible ternary gates and for each of them check whether it is universal. (This is quite tedious but it can be done.) Alternately, if you know what makes a gate universal, you can generate the list of universal gates much faster and with less effort.

Easy subproblem

We can easily generate a few universal ternary gates by hand. For example, the gate `11101110` has to be universal. Why? This is the following gate: "for inputs w, x, and y, return $NAND(w,x)$". Clearly we can use this gate to construct any other gate, just as we would use the binary $NAND$ gate – we just connect something arbitrary to the ignored input.

In addition to this gate, there are two other ternary gates that return the $NAND$ of two of their inputs; and there are three more gates that do the same with NOR.

At the moment, we have 6 universal ternary gates – we only need one more.

A good guess that can easily be verified is the ternary $NAND$ gate – one that returns 0 if and only if all three inputs are 1. (And by symmetry, ternary NOR works as well.)

Hard subproblem – generate and check

Generating the gates is easy, as there are precisely $2^8 = 256$ of them. But how does one check whether a gate is universal?

A ternary gate is certainly universal if we can construct every other ternary gate from its copies. (Actually, constructing any single gate that is known to be universal is also sufficient.)

In order to check whether a gate G is universal, we will be incrementally constructing the set of all ternary gates that can be obtained by wiring together one or more copies of our gate. What do all such schemes have in common? The final output has to be computed by a G gate. Now, that G gate has three inputs. Without loss of generality we may assume that each of them is computed independently by some scheme of G gates – in other words, by some gate we already know how to construct out of G gates.

As a scheme, this part of the construction looks as follows:

```
  +----+
--|    |
--| X  |--+
--|    |  |
  +----+  |
          |
  +----+  |  +----+
--|    |  +--|    |
--| Y  |-----| G  |--
--|    |  +--|    |
  +----+  |  +----+
          |
  +----+  |
--|    |  |
--| Z  |--+
--|    |
  +----+
```

Finally, the entire gate we are just constructing has to have only three inputs, and each of the three gates X, Y, and Z has to get some combination of these inputs. Here we could again use brute force to generate all possibilities, but there is a smarter way: During the construction we will assume that the three inputs are given in their original order to each of the three gates X, Y, and Z.

The rearrangements of inputs will be addressed at another time: whenever a new gate is constructed (including the very beginning when the first gate we know how to construct is G) we take the gate and generate all new gates that can be obtained by rearranging inputs of the original gate. If we do it this way, when we later choose the three gates X, Y, and Z, this choice also includes the way how the inputs are rearranged.

Thus the entire creation of a new gate looks as follows: for each triple (X, Y, Z) of gates we already know how to construct we combine the gates into a new one, and add this new one (and all the gates derived from it) into the set of gates we already know how to construct.

After no more new gates can be added, we check whether the set contains all 256 possible gates.

Hard subproblem – what makes a gate universal?

We will now define five properties a gate may have:

1. Truth preservation: A gate is called truth-preserving if it gives the output 1 when all inputs are 1.
2. Falsehood preservation: A gate is called falsehood-preserving if it gives the output 0 when all inputs are 0.
3. Monotonicity: A gate is called monotonous if changing an input from 0 to 1 never changes an output from 1 to 0.
4. Self-duality: A gate is called self-dual if negating all inputs always negates the output.
5. Affinity: A gate is called affine if each input either never influences the result, or always influences the result.

It is easily seen that each of this properties is preserved under composition: e.g., regardless of how many gates you have and how you connect them, if all of them are truth-preserving, so will be the newly constructed gate.

Also, for each of these properties we can easily find a ternary gate that does not have it.

Thus, we get a necessary condition: A gate can only be universal if it does not have any of the five properties.

It turns out that this necessary condition is also sufficient. This is a corollary of a more general result by Emil Post (the logician who became one of the pioneers of computability theory). The result states the following: Any set $\mathcal{S}$ of logical connectives (i.e., gates) is universal if and only if for each of the 5 properties listed above it contains at least one connective that does not have the property.

Given a particular gate, each of the above properties can be tested efficiently. And this gives us a simpler solution of the subproblem J2. Moreover, this solution can easily be generalized to gates with arity more than 3.

Problem K: Knee problems

You wander through a dark dungeon. All around you there are doors of different shapes and colors. You pick one, open it and enter.

"I knew you would come," said a voice in the dark. You come closer and see an old man with a long white beard sitting on the floor.

"I used to be a problem solver like you," he says, "but then I took an arrow in the knee."

"Seriously?" you ask him.

"Well... not really. It's just what all the kids were saying the last time I saw daylight."

"So what happened to you?" you ask and sit beside him.

"The truth is, I destroyed my kneecaps on the stairs. When I was younger, I did a lot of programming contests. And in one of them was a really nasty task. I had to determine the number of ways in which one can go up and down a staircase with n steps. Of course, there were some constraints: when going up, you can take two steps at a time, and when going down, you can take up to four steps at once."

He sighs deeply. "I had no idea how to solve the task, so I found a staircase and attempted to try every possibility. But there were so many of them that I overloaded my knees and now I can't even walk. So I'm sitting here and still wondering about a solution for that problem. Can you help me to finally put a close on this?"

Problem specification – easy subproblem K1

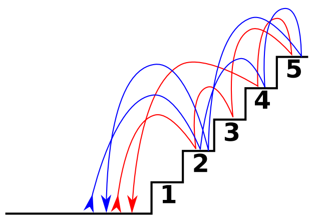
The staircase consists of n steps. Count the ways of going up and then down the staircase, given the following constraints:

- On the way up, you can take either 1 or 2 steps at a time.
- On the way down, you can take 1, 2, 3, or 4 steps at a time.

As the actual number of ways can be huge, compute the remainder it gives when divided by $10^9 + 9$.

For example, for $n = 5$ one valid way of going up and down the staircase looks as follows: Start on the ground, ascend to step 2, continue to step 3, and then go to step 5. Having now reached the top of the staircase, you turn around and walk down, first descending to step 4 and then going directly to the ground (which is, at that moment, 4 steps below).

The figure on the right shows two valid ways of going up and down the stairs for $n = 5$. The one described above is shown in red.



Problem specification – hard subproblem K2

The staircase consists of n steps. Count all ways of going up and then down the staircase, given the following constraints:

- On the way up, you can take either 1 or 2 steps at a time.
- On the way down, you can take 1, 2, 3, or 4 steps at a time.
- On the way down, you can only walk on the steps you used on the way up.

Again, your task is to compute the number of valid paths modulo $10^9 + 9$.

In the figure above, the red path is not valid for this subproblem: on the way down we walk on step 4, which was not used on the way up. The blue path $(0 \to 2 \to 4 \to 5 \to 2 \to 0)$ is valid.

Input specification

The first line of input contains one integer number t specifying number of test cases. Each test case is preceded by a blank line.

Each test case consists of a single line with the integer n $(1 \leq n \leq 100,000)$ – the number of steps.

Output specification

For each test case print a single line with one integer – the number of valid paths modulo $10^9 + 9$.

Example

input

```
2

3

5
```

output

```
12
120
```

This output is correct for the easy subproblem K1. For example, when $n = 3$ there are 3 ways to go up, and for each of them there are 4 ways to go back down.

input

```
2

3

5
```

output

```
8
52
```

This output is correct for the hard subproblem K2.

Task authors

Problemsetter:	Michal 'mišof' Forišek
Task preparation:	Michal 'Žaba' Anderle, Vlado 'Usamec' Boža

Solution

For the easy subproblem we know that the two parts of the path are independent. So if we find number of ways of going up and the number of ways of going down, we can multiply them to get the result.

Let's use $up(n)$ to denote the number of ways to go up n steps, and $down(n)$ the number of ways to go down n steps. These functions can be defined recursively: $up(n) = up(n-1) + up(n-2)$ and $down(n) = down(n-1) + down(n-2) + down(n-3) + down(n-4)$. (The initial conditions: $up(n) = down(n) = 0$ for $n < 0$, and $up(0) = down(0) = 1$.)

Their values can easily be computed by using dynamic programming or recursion with memoziation.

In the hard subproblem the situation is bit different: the way down depends on the way up. There are two tricks that can help us in such situations:

First of all, we will reverse the second part of the path. Instead of a person going up and then down (using only the set of steps used while going up), we will have two people going up the stairs, one after another. The second person still has the same constraints: he is able to move up to four steps at a time, and he must only use the steps used by the first person.

Now the second trick: we can construct both paths simultaneously.

In our case, it is better to focus on the person with larger stepsizes. Whenever he moves up 1 step, the other person must also move up 1 step. Whenever he moves up 2 steps, the other person has two options: either move up 2 steps, or move up 1 step twice. Similarly we get 3 options for the smaller-step-maker if the larger-step-maker moves up 3 steps ($1-1-1$, $1-2$, $2-1$) and 5 options if the larger-step-maker moves up 4 steps. Thus we derived the following recursive relation: $hard(n) = hard(n-1) + 2 \cdot hard(n-2) + 3 \cdot hard(n-3) + 5 \cdot hard(n-4)$.

Problem L: Labyrinth

To alleviate the stress you surely experience during programming competitions, we invite you to play a fun browser-based game.

Problem specification

Your objective in the game will be to find your way through a labyrinth. You win when you reach the finish tile.

The labyrinth is full of doors that will often block your way, and switches that can be used to open and close the doors. Every door and switch is marked by a letter (or a pair of letters) and a color. Pressing a switch toggles all doors that have the same marking.

Each of the two subproblems consists of 7 levels. Send us your solutions to all 7 levels to solve the subproblem. Good luck!

JavaScript application

The game is a browser-based JavaScript application. You can either open it from the online problem statement, or open `l/game.html` in the downloadable archive.

To move, use the arrows on your keyboard or the numeric keypad. To push a button you're standing on, press either "P", the numeric keypad "5", or Enter.

You'll need a reasonably modern browser to play. Old versions of Internet Explorer probably won't work.

Input specification

There is no input.

Output specification

Your steps through the labyrinth will be recorded as a string of letters 'U', 'D', 'L', 'R', and 'P' (meaning "up", "down", "left", "right", and "push", respectively). Collect the solution strings for all 7 levels in a text file and submit it.

The output file must contain 7 whitespace-separated strings. The i-th string must be a solution for the i-th level in the subproblem.

Your output file must not contain more than 1,000,000 characters.

Task authors

Problemsetter:	Martin 'Imp' Šrámek
Task preparation:	Tomáš 'Tomi' Belan

Solution

Easy subproblem

Just as the problem statement suggests, the easy subproblem can be solved by simply playing the game.

In the last two levels, the doors are organized into triplets. You have to get through each triplet, but to do so you only need to open one door. This should remind you of 3-SAT – you have to satisfy every clause, but to do so, you only need to satisfy one literal in it. This demonstrates how even such a simple game can in fact represent an NP-complete problem.

The 3-SAT instances in the easy subproblem were, however, very small and easily solvable by hand. Their only purpose was to prepare you for what you would find in the hard subproblem.

Hard subproblem

Every level in the hard subproblem had the shape of one long corridor with many triplets of doors. That means you had to solve a SAT instance. As the instances are quite large, these levels were actually rather impossible to be played. So, despite our promise of a game, you were better off programming the solution.

As you are not given the level maps directly, you must first parse them from the javascript code of the game. In this code, the levels are represented by numeric matrices. You must compare these matrices with what you see in the game to find out which number equals which feature (e.g. 0 is a wall, 1 is a corridor etc.). A JSON parser can be of help here.

After parsing the levels, you can start searching for triplets of doors that represent clauses. You may notice that they are always conveniently arranged on three adjacent vertical or horizontal tiles. Once you find all the clauses, you end up with a formula in the conjunctive normal form. To find a satisfying assignment, you can use a third party SAT-solver or program your own. A heuristic solver should suffice, as the formulae have many satisfying assignments.

The satisfying assignment assigns a 1 or 0 to every literal. In our game, flipping the values of boolean variables corresponds to toggling a button. Therefore, this assignment advises you on which buttons should be pressed.

At this point, you have two possibilities. You may play the game manually, following the list of buttons that should be pressed and ultimately travelling through the corridor once every triplet is open. Or you may write a program to do this instead of you.

Pathfinding in the level can be done by simply starting a breadth-first search from the player's current position. This way, you can find path to each button that has to be pressed and – of course – press it. Then, the only remaining part of the program is to find the path to the finish.

Problem M: Morning hassle

Peter was supposed to catch a morning train at 7:30. But as usual, he overslept the alarm set to 6:30 and he just woke up with 7:29 on the clock. He only managed a single "Oh crap!" before the train was gone. Luckily for Peter, everything can still be saved. He can take his old car out of the garage and drive it to his destination.

Peter lives in the middle of an abandoned countryside. There is a single long straight road going across the countryside. Peter's home and his destination both lie on this road.

Still, Peter has a valid reason to prefer the train. The whole countryside is covered by train tracks, and thus the road is riddled with railroad crossings. And as trains have priority over cars, you could easily end up waiting for a long time at some of those crossings.

Moreover, even if you are not waiting for a train to pass, you still need to approach the railroad crossings carefully – they are in pretty bad shape and if Peter were to drive carelessly, the crossing could easily break his old car.

Problem specification

For the purpose of this task, the road is an infinite straight line. Peter's home, his destination, each of the railroad crossings, and Peter's car should all be considered points. Peter's car is the only point that will be moving, all the other ones are stationary. The movement of Peter's car is continuous (not discrete).

We will be using a linear coordinate system on the road, with Peter's home being at 0 and his destination at some $x_{end} > 0$. All coordinates are in meters, all speeds are in meters per second, all accelerations are in meters per second squared.

All the railroad crossings lie strictly between Peter's home and his destination, at pairwise different coordinates x_i. In the input, their descriptions are ordered by their coordinate.

There are no other cars on the road. Peter's car can move freely along the line, including the parts that are not between his home and his destination. However, the movement of his car is subject to the following constraints:

- The car's acceleration (change of velocity over time) has to be between $-a_{max}$ and a_{max}, inclusive. (E.g., if your current speed is $v = 20$ and $a_{max} = 1.2$, after 0.5 seconds your speed can be anything between 19.4 and 20.6, inclusive.)
- The car cannot enter a crossing when the crossing is closed because of a passing train.
- In general, the speed of the car has no upper bound – it can go arbitrarily fast.

  However, the railroad crossings are special: The maximum allowed speed at a railroad crossing is v_{max} (an integer).

  But even when driving slower than v_{max}, the bumping while crossing the railroad sometimes tends to resonate parts of Peter's car and Peter fears that the car might break. He has already tested that this does not happen if the speed of his car is an integer less than or equal to v_{max}. He now refuses to drive over a railroad crossing using any other speed.

  Therefore, whenever the car crosses a railroad, its speed at that moment has to be a positive integer.

  (Note that zero is not allowed. Stopping at a railroad crossing is forbidden by law.)

Peter's car starts stationary ($v = 0$) at his home. Calculate the shortest time in which it is possible to park the car (i.e., have $v = 0$ again) at Peter's destination (x_{end}).

Input specification

The first line of the input file contains an integer $t \leq 500$ specifying the number of test cases. Each test case is preceded by a blank line.

Each test case starts with a single line containing four numbers:

- a floating-point number x_{end} ($0.5 \leq x_{end} \leq 1500.0$): the destination
- a floating-point number a_{max} ($0.1 \leq a_{max} \leq 10.0$): the maximum acceleration
- an integer v_{max} ($1 \leq v_{max} \leq 40$): the maximum speed over a crossing
- an integer n: the number of railroad crossings (constraints are given below)

Next n lines describe railroad crossings, one per line. Each line starts with two numbers. The first one a floating-point number x_i: the coordinate of this crossing. The second one is a nonnegative integer m_i: the number of trains that will be passing through the crossing. Then, $2m_i$ floating-point numbers follow: for each train the start $s_{i,j}$ and the end $e_{i,j}$ of the time interval when the crossing is blocked by the train.

You may assume the following things about the crossings:

- Their coordinates are in sorted order: $0 < x_1 < \cdots < x_n < x_{end}$.
- The intervals when the crossing is blocked are given in chronological order, they do not overlap, and they do not even touch (i.e., the end of one interval is always strictly less than the start of the next one). The first interval starts at 0 or later, the last interval ends at 10^6 or sooner.
- Assume that the intervals are open – if you arrive precisely at their start or end, you are still able to cross in either direction.

There should be no numerically unstable test cases in the test data. More precisely, for each test case we used: 1) if we make small changes to the values $s_{i,j}$ and $e_{i,j}$, the optimal path remains essentially the same; and 2) if we make a small change to a_{max}, the optimal path remains essentially the same.

Subproblem-specific constraints

In the easy subproblem M1, $0 \leq n \leq 1$ (i.e., there is at most one crossing) and $0 \leq m_i \leq 2$ (there are at most two trains per crossing). In the hard subproblem M2, $0 \leq n \leq 30$ and $0 \leq m_i \leq 25$.

Output specification

For each test case, output a single line with a floating-point number on it – the earliest time Peter can be parked ($v = 0$) at position x_{end}. Output sufficiently many decimal places. Answers with absolute or relative error up to 10^{-6} will be accepted.

Example

input

```
1

10 1 3 0

10 1 30 1
5 1 2 3
```

output

```
6.32455532034
6.32882800594
```

In the first case, there is no crossing and so Peter may drive directly to his destination. The optimal strategy is to accelerate until he reaches $x = 5$, and then to brake for the rest of the way. Note that his maximum speed during this trip will exceed v_{max}.

In the second case the train will leave the crossing before Peter can possibly reach it. Still, the crossing limits the car speed. In the optimal solution Peter will cross the railroad crossing having speed $v = 3$.

Task authors

Problemsetter:	Peter 'Ppershing' Perešíni
Task preparation:	Peter 'Ppershing' Perešíni, Michal 'mišof' Forišek

Solution

All the conditions in the problem statement looks scary, don't they? The good news is that because the speed at any railroad crossing can only be an integer (and not a big one), we can easily enumerate all possible car speeds at all crossings. But that is the second part of the story.

First thing we need to revise are the equations tying velocity and distance. In its simplest form, the free-fall equation is $s = at^2/2$ and the speed is $v = at$. If we start with a fixed speed v_0 instead of zero, the equation will become $s = v_0t + at^2/2$ when accelerating and $s = v_0t - at^2/2$ when decelerating.

This is quite enough to solve the special case where there are zero railroad crossing. The fastest way to reach the destination in this case is to fully accelerate till we reach the midpoint between start and destination and then fully brake until we stop precisely at the destination. The time needed to perform the manouver is $2t$ where t is free-fall time for distance $s/2$, i.e. $t = \sqrt{s/a}$.

Easy

Now, let us concentrate on the case that there is a single crossing and no trains. We can divide the task into two subtasks – before and after the crossing. To tie these together, we need to match the speed in both parts. We thus need to solve the problem "accelerate from start to the crossing, reaching speed v" and its counterpart "brake from speed v at the crossing and arrive at the destination". You may notice that the subtasks are isomorphic and thus we just need to solve only one of them. Let it be first task – given starting position $x = 0$, $v = 0$ at time $t = 0$, what is the earliest time we can be at position $x = x_{cross}$ and speed $v = v_{target}$?"

By t_{acc} let us denote minimal time needed just to accelerate to the target speed, $t_{acc} = v_{target}/a$. Similarly, let x_{acc} be minimal distance needed to reach target speed, $x_{acc} = 1/2a * t_{acc}^2$. If $x = x_{acc}$ then the answer to our question is clearly t_{acc}. If $x > x_{acc}$ then by the time we accelerated to the target speed, we are still away from the crossing. Naturally, the fastest way would be to continue accelerating till we reach midpoint between x_{acc} and x_{cross}, then to delecerate and reach the crossing with exactly required speed. Thus, we have $t_{min} = t_{acc} + 2 * t'$ where $(x_{cross} - x_{acc})/2 = v_{target} * t' + 1/2a * t'^2$. By solving the quadratic equation one can easily get the result. Finally, if $x < x_{acc}$ then we cannot accelerate and thus it is impossible to reach the crossing with enough speed, isn't it? Well, let us look at the problem statement again – you may have noticed some pecularities there like "including the parts that are not between his home and his destination". Does it sometimes pay off to go in reverse gear? Why yes, it does! In our case we need to go back for at least $x_{reverse} = x_{acc} - x$. As a bonus, we already know how much time we need to traverse distance $x_{reverse}$ as this is the same equation as in the no-railroad case.

To put it all together, we know the soonest time we can reach the railroad crossing for each target speed v_{target} and we can use the same formula to determine the minimum braking time after we cross the railrod. By looking at the minimum over all possible crossing speeds, we can pick the best solution.

In case the crossing is closed because of a train, we simply need to wait till it opens. Note however that waiting directly before the crossing is not going to help as we will loose the speed. Instead, one can simply wait the beginning (when we have speed zero) and give Peter some time to properly wake up.

One final note: One favorite mistake in the easy subproblem was treating the start and the destination as crossings with no trains. This is incorrect because there should not be a speed restriction when crossing the start/the destination.

Hard

We start with a simple observation that for any valid solution the sequence X of crossing ids we crossed satisfies $|X_{j+1} - X_j| \leq 1$. More precisely, if we crossed railroad r from left to right, in the next step we can either cross $r+1$ from left to right, or cross r from right to left. The situation is symmetrical when crossing r in the opposite direction.

We can therefore divide the whole task into several subtasks, one subtask per each between-the-crossings region on the line. Inside each region, the car can travel freely without any restrictions. Between the regions, the car can travel only by the rules described previously (i.e. either we cross to the next region in the same direction or we return to the region we came from). Let us therefore define function $T_{i+,v}$ as a set of all possible times the car can be at position $x_i + \varepsilon$ after crossing x_i with speed v. Note that $0 < v \leq v_{max}$. Similarly, let $T_{i-,v}$ to be set of all possible times the car can be at position $x_i - \varepsilon$ after crossing x_i with speed v, $-v_{max} \leq v < 0$. In the absence of trains, $T_{i+,v}$ can be computed from values of $T_{i-,v'}$ and $T_{i-1+,v'}$ as

$$T'_{i+,v} = \bigcup_{v'} \{t_1 + t_2 \,:\, t_1 \in T_{i-1+,v'},\, t_2 \in \text{time to cross region } i \text{ starting with speed } v' \text{ and ending with } v\}$$

$$\cup \bigcup_{v'} \{t_1 + t_2 \,:\, t_1 \in T_{i-,v'},\, t_2 \in \text{time to go back starting with speed } v' \text{ (to the left) and ending with } v\}$$

and there is a similar equation for $T_{i-,v}$. Taking crossing closing times into the account is quite simple – we just intersect the previous result with all allowed times for that crossing

Let us focus now on the set of all times t the car can take to cross the region. We start with the feasibility question: Is is possible to accelerate/decelerate from starting speed v to ending speed v' in distance at most s? Without loss of generality, we can assume $v < v'$ as the situation is symmetrical (you can switch start and end). (The fastest time to change the speed is $t_{change} = (v' - v)/a$ and the distance travelled during this time is $s_{change} = v * t + 1/2a * t^2$. If $s \geq s_{change}$, the change is possible otherwise we hit another railroad crossing before we had the chance to reach the desired speed.

Assuming $s \geq s_{change}$, We can calculate the fastest time to travel the distance s in similar fashion as before – we take the rest of the way after we accelerate to the desired speed, divide in in two halves and we acclelerate till we reach the midpoint, then we brake, i.e. $t_{min} = t_{change} + t'$ where for t' we have $(s - s_{change})/2 = v * t' + 1/2a * t'^2$.

Which other times t we can achieve? We can start introducing little braking at the beginning. The more we brake at the beginning, the bigger distance it will take until we change the distance and the less there will be left to go over the speed v'. In the extreme, we will brake till the point from which we will need to accelerate just to catch up with the destination speed.

To calculate t_{max}, we follow the traditional recipe, this time however we will change the speed from v to v' at the end and of the region and we will use remaining distance to slow us down as much as possible – $t_{max} = t'' + t_{change}$ where we have $(s - s_{change})/2 = v * t'' - 1/2a * t''^2$. Note that the last quadratic equation may have two roots $t''_1 < t''_2$, both of them greater than zero. In practice, it means we have two solutions, t''_1 being what we described and t''_2 being a new fascinating behavior – brake till stop, then reverse and go backwards for some time and only then start accelerating (braking again and going forward). If we investigate this behavior in more details, we may notice that:

- if $s \geq 1/2v^2/a + 1/2v'^2/a$ then we can always stop and prolong our time indefinitely. This means we can achieve times $[t_{min}, \infty)$.

- if $s < 1/2v^2/a + 1/2v'^2/a$ and $s > 1/2v'^2/a$ ($s > 1/2v^2/a$ follows from $v < v'$) we have two options – either we just slow a little and then accelerate back or we decide to continue braking more and in order to accelerate back to the required speed, we will need to go backwards (note that we do not go outside of the region). Moreover, once we stop, we can prolong our journey indefinitely. This gives us possible times $[t_{min}, t''_1 + t_{change}] \cup [t''_2 + t_{change}, \infty)$
- otherwise we won't be able to completely stop/accelerate the car without going outside of the region. The only possible times are $[t_{min}, t''_1 + t_{change}]$.

Determining all possible times between we cross the same railroad twice is similar to the previous case, albeit a bit simpler. Again assume $|v| < |v'|$. If $s > 1/2v'^2/a$, we simply cannot keep the car inside the region. Otherwise, the optimal strategy for minimizing the time is to stop the car at distance $d = 1/2v'^2/a$. We already know what is the minimum time to go across d starting with zero speed and finishing with speed v – that was the case we solved for the easy subproblem. Because we can stay at the point where we stopped for any time, the maximum time is unbounded.

Finishing touches

Putting it together we have a nice bunch of recursive set equations. As a first thing, we observe that the actual sets $T_{i\pm,v}$ are continuous and thus can be represented by a bunch of intervals. So instead of working with sets, we will work with interval sets where union, intersection and "plus product" $\{x+y|x \in X, y \in Y\}$ are mostly trivial operations. Second thing we need to observe is that the recursive equations can be solved by iterating – the equations are monotonic [3] (i.e. $f(X,Y) \subseteq F(X',Y')$ iff $X \subseteq X', Y \subseteq Y'$) and therefore according to Tarski's fixed-point theorem iterating will converge towards a fixed point (our solution). Here we note that for all test cases the number of iterations is reasonably small as in iteration i we compute all reachable states after crossing up to i railroads.

[3] With the exception of the first crossing in the left-to-right direction where we should not forget about the possibility that the car just started and this is it's first crossing.