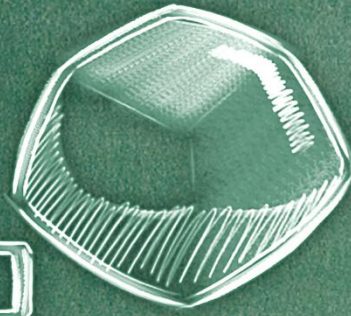


bubble
cup



Student coding contest

Problem Set -booklet



BUBBLE CUP 7DC

**Student programming contest
Microsoft Development Center Serbia**

Problem set & Analysis From the Finals and Qualification rounds

Belgrade, 2012

Scientific committee:

Andreja Ilić
Andrija Jovanović
Milan Vugdelija
Mladen Radojević
Miroslav Bogdanović
Dražen Žarić
Dimitrije Filipović
Željko Nikoličić
Milan Novaković

Qualification analyses:

Nikola Milosavljević
Boris Grubić
Dušan Zdravković
Dimitrije Dimić
Uros Joksimovic
Milos Biljanovic
Dejan Pekter
Bartek Dudek
Linh Nguyen
Petar Veličković
Ivan Stošić
Goran Žužić
Bartosz Tarnawski
Vanja Petrović Tanković
Vladislav Haralampiev
Aleksandar Ivanović
Stjepan Glavina
Patrick Klitzke
Teodor Von Burg
Filip Pavetić
Predrag Milošević
Danilo Vunjak
Marko Baković

Cover:

Sava Čajetinac

Typesetting:

Andreja Ilić

Proofreader:

Andrija Jovanović

Volume editor:

Dragan Tomić

Contents

<i>Preface</i>	6
<i>About Bubble Cup and MDCS</i>	7
<i>Bubble Cup Finals 2012</i>	8
Problem A: Good sets	9
Problem B: Wheel Of Fortune	13
Problem C: MaxDiff	15
Problem D: Cars.....	18
Problem E: Triangles.....	22
Problem F: Olympic Games	25
Problem G: Matrix.....	27
Problem H: String covering	31
Problem I: Polygons.....	34
<i>Qualification</i>	37
Problem R1 01: November Rain (code: RAIN1)	39
Problem R1 02: Ambiguous Permutations (code: PERMUT2).....	41
Problem R1 03: Roll Playing Games (code: RPGAMES)	43
Problem R1 04: Manhattan Wire (code: MMAHWIRE)	45
Problem R1 05: Spheres (code: KULE).....	48
Problem R1 06: Sightseeing (code: GCPC11H)	52
Problem R1 07: Segment Flip (code: SFLIP)	54
Problem R1 08: K12 - Building Construction (code: KOPC12A)	57
Problem R1 09: Its a Murder! (code: DCEPC206)	59
Problem R1 10: Words on graphs (code: AMBIG)	61
Problem R2 01: Zig-Zag Permutation (code: ZZPERM)	63
Problem R2 02: DEL Command II (code: DELCOMM2)	65
Problem R2 03: Boxes (code: BOX)	67
Problem R2 04: Cryptography (code: CRYPTO).....	72
Problem R2 05: Slow Growing Bacteria (code: SBACT)	74
Problem R2 06: Reverse the sequence (code: REVSEQ)	76

Problem R2 07: Cover the string (code: MAIN8_E)	78
Problem R2 08: Dynamic LCA (code: DYNALCA)	80
Problem R2 09: Magic Bitwise AND Operation (code: AND)	82
Problem R2 10: Contaminated City (code: 1CONTCITY 19)	84
Problem R3 01: Four Mines (code: MINES4)	87
Problem R3 02: Lost in Madrid (code: LIM)	93
Problem R3 03: Circles (code: CIRCLES)	96
Problem R3 04: Bridges! More bridges! (code: BRII)	100
Problem R3 05: Polynomial $f(x)$ to Polynomial $h(x)$ (code: POLTOPOL)	103
Problem R3 06: Factorial challenge (code: FUNFACT)	105
Problem R3 07: Hi6 (code: HISIX)	106
Problem R3 08: Frequent values (code: FREQUENT)	109

Preface

Dear Finalist of BubbleCup 5,

Thank you for participating in the fifth edition of the Bubble Cup. On behalf of Microsoft Development Center Serbia (MDCS), I wish you a warm welcome to Belgrade and I hope that you will enjoy yourself.

MDCS has a keen interest in putting together a world class event. Most of our team members participated in similar competitions in the past and have passion for solving difficult technical problems.

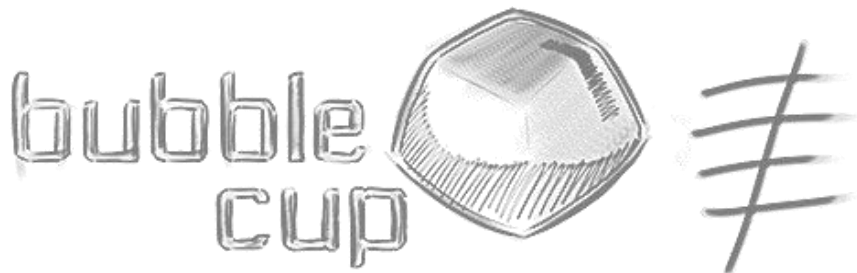
This edition of the Bubble Cup is special. It is its fifth anniversary and it is the most international event that we have had so far. Not only do we have the participants from the region (Bulgaria, Croatia, Serbia) but teams from Germany and Poland fought their way to the Finals too. This means that BubbleCup is reaching more and more fans every year.

Given that we live in a world where technological innovation will shape the future, your potential future impact on humankind will be great. Take this opportunity to advance your technical knowledge and to build relationships that could last you a lifetime. I wish you all warm welcome to Belgrade.

Thanks,

Dragan Tomić

MDCS Group Manager/Director



About Bubble Cup and MDCS

BubbleCup is a coding contest started by Microsoft Development Center Serbia in 2008 with a purpose of creating a local competition similar to the ACM Collegiate Contest, but soon that idea was outgrown and the vision was expanded to attracting talented programmers from the entire region and promoting the values of communication, companionship and teamwork.

The contest has been growing in popularity with each new iteration. In its first year close to 100 participants took part and this year while in 2012 this number is over 500.

This year the emphasis was on keeping intact all of the things that made BubbleCup work in previous years but taking every opportunity to tweak and subtly improve the format of the contest. The third qualifying round was added this year, where contestants had the opportunity to choose the problems themselves.

Microsoft Development Center Serbia (MDCS) was created with a mission to take an active part in conception of novel Microsoft technologies by hiring unique local talent from Serbia and the region. Our teams contribute components to some of Microsoft's premier and most innovative products such as SQL Server, Office & Bing. The whole effort started in 2005, and during the last 7 years a number of products came out as a result of great team work and effort.

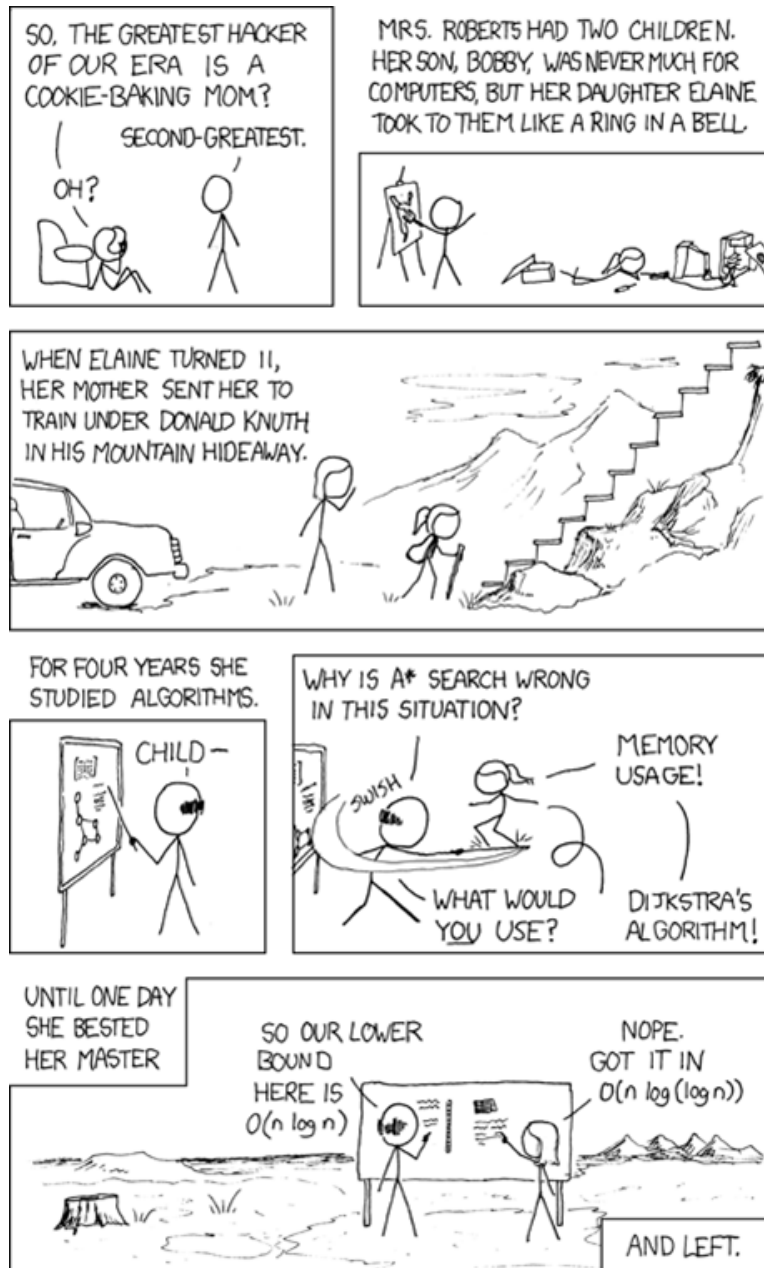
Our development center is becoming widely recognized across Microsoft as a center of excellence for the following domains: computational algebra engines, pattern recognition, object classification, computational geometry and core database systems. The common theme uniting all of the efforts within the development center is applied mathematics. MDCS teams maintain collaboration with engineers from various Microsoft development centers around the world (Redmond, Israel, India, Ireland and China), and Microsoft researchers from Redmond, Cambridge and Asia.



Microsoft[®]
Development Center Serbia

Bubble Cup Finals 7DC

Problem set & Analysis



Taken from xkcd.com – A web comic of Romance, Sarcasm, Math, and Language

Finals 2012

The finals of BubbleCup 5 were held on September 8th 2012, with 17 teams competing at the Faculty of Electrical Engineering in Belgrade. There were 9 problems and five hours to solve them.

There were no changes to the rules from the previous years. As a reminder: the team which solves the most problems wins. In case of ties, the team with less penalty points (gained for incorrect submissions and deducted based on qualification results) is preferred. Programming style is not taken into account.

The difficulty of the problems was relatively balanced – no problem was solved by more than 12 teams (unlike last year, when 3 problems were solved by every team), and one problem remained unsolved. The accent was mostly on problems which required original thinking, with only a few that were tricky to implement.

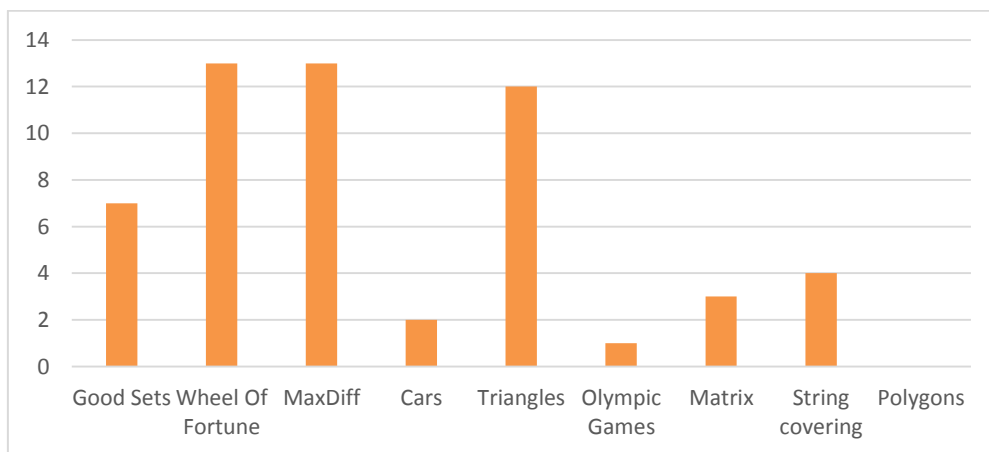


Figure 1. Number of accepted solutions per problems

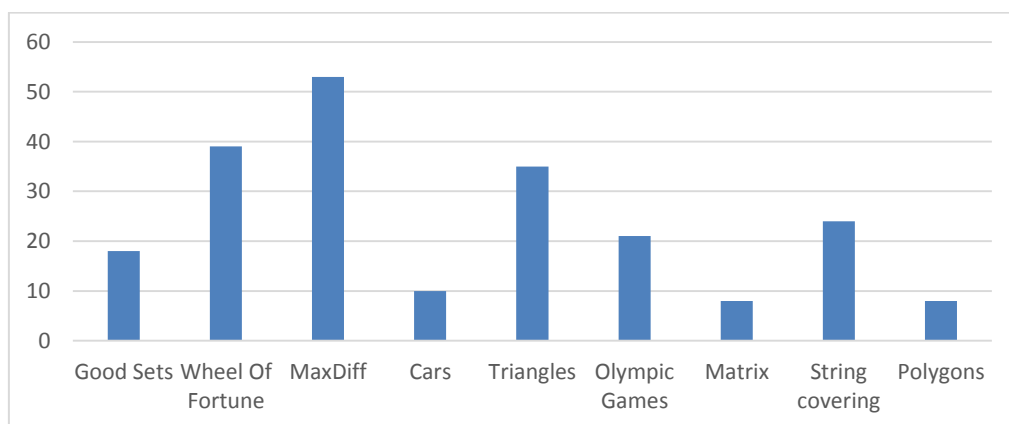


Figure 2. Number of submissions per problems

The competition was initially very close, but towards the end the team koko koko euro spoko pulled away from the rest and had no problems winning the first place with 8 solved problems. The fight for second and third was very exciting and continued until the last couple of minutes. In the end the second place went to ☺S-Force☺ and the third to Suit Up! (finishing in the top three for the third consecutive year).

Similar to the previous year, the Scientific Committee decided to give some special awards:

- The first accepted solution - *Silver lightning*: **koko koko euro spoko (Poland)**
- The shortest accepted solution - *Vertipaq coders*: **H-Rast (Croatia)**
- The most persistent team - *while (! accepted)*: **The Code Breathers (Germany)**
- BubbleCup friend - *The best mentor*: **Dusko Obradovic, team Gimnazija Sombor (Serbia)**

ID	Problem Name	ACC/ SUB ratio	Min elapsed until first accepted solution
A	Good Sets	0.39	33
B	Wheel Of Fortune	0.34	15
C	MaxDiff	0.24	17
D	Cars	0.20	201
E	Triangles	0.34	87
F	Olympic Games	0.05	211
G	Matrix	0.37	167
H	String covering	0.17	104
I	Polygons	/	/

Table 1. Statistic for time and accepted / submitted ratio

The Scientific Committee would like to thank all the teams and individuals for their interest, enthusiasm and hard work.

Problem A: Good sets

Author: **Milan Vugdelija**

Implementation and analysis: **Milan Vugdelija**

Statement:

Let A be the set $\{1, 2, \dots, n\}$, where n is a given natural number. Set B is called *good* if it has the following properties:

- B is a subset of A ;
- For every x , if x belongs to B , then $2x$ doesn't belong to B ;
- No other set C can have properties a) and b) and a greater number of elements than B ;

For example, if $n = 12$, then $A = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$, and the set $\{1, 3, 4, 5, 7, 9, 11, 12\}$ is good, while $\{1, 4, 5, 6, 7, 9, 11\}$ is not good (note that set C from the third property doesn't have to be a superset of B).

Given positive integer numbers n and b compute the following:

- The number of elements in every good set;
- With how many zeros the total number of good sets ends, if written in base b .

Input:

The first and only line of input contains two integers n and b , separated with one empty space, representing cardinality of the set A and the given base b , respectively.

Output:

Output contains only one line with two integers, separated with one empty space: the number of elements in every good set and number of zeros at the end of the total number of good sets in base b , respectively.

Example input:

12 3

Example output:

8 1

Example explanation:

All good sets consist of 8 elements and there are 6 of them - $6_{(10)} = 20_{(3)}$.

Constraints:

- $1 \leq n \leq 4 \cdot 10^9$
- $2 \leq b \leq 100$
- Number b is a prime number.

Time and memory limit: 0.5s / 64 MB

Solution and analysis:

Divide set A into chains such that each chain starts with an odd number from A and contains repeatedly doubled values from A . For example, if $n = 12$, set A is $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ and the chains

are $1 - 2 - 4 - 8$, $3 - 6 - 12$, $5 - 10$, 7 , 9 , 11 . Here the first chain contains 4 elements, the second chain 3, the third 2, and there are three one-element chains. Now elements are chosen from each chain independently.

Let us denote with $d[n]$ the maximal number of elements from a chain of length n such that no two are consecutive. It is not hard to see that if a chain has an odd number of elements, there is only one way to pick the maximal number of elements from that chain (take every element with an odd index). So, for a chain with $2k - 1$ elements, that maximal number is k . In other words, we have that $d[2k - 1] = k$.

For chains with an even number of elements, say $2k$, maximal number of elements that can be taken is k . Can we apply some sort of induction here? For $2k + 2 = 2(k + 1)$ elements we see that we must use exactly one element out of the last two (because otherwise we would have to select $k + 1$ elements from $2k$, which is not possible by induction). If we choose the last one, then from the first $2k$ we must select k of them and this can be done in $d[2k]$ ways. In the second case, if we do not select the last one, then we would have to select $k + 1$ elements from first $2k + 1$. From prior discussion we have that there is only one way to do this. Now we have:

$$d[2k + 2] = d[2k + 1] + d[2k] = 1 + d[2k]$$

Finally, from induction we have that $d[2k] = k + 1$, because $d[2] = 2$.

It remains to count chains of each different length, add up maximal numbers of elements for each chain, and multiply ways to choose such elements from each chain. Actually, instead of computing the exact number of ways to form a good set, it is required only to compute how many times this number is divisible by a given prime number b .

Problem B: Wheel Of Fortune*Author: Dražen Žarić**Implementation and analysis: Dražen Žarić***Statement:**

You are on a quiz show playing the game Wheel of Fortune. The wheel has N fields of the same size, and each field $1, 2, \dots, N$ is associated with a value: $D[1], D[2], \dots, D[N]$. Each time you spin the wheel you have equal probability of hitting any of the N fields. You will spin the wheel K times. When you spin the wheel for the i -th time and it stops on field j , if it is your first time hitting that field, $D[j]$ dollars is added to your prize and field j gets *marked*. If the wheel stops at a marked field, meaning you've hit that field in some of your previous spins ($i - 1, i - 2, \dots, 1$), your score does not increase.

What is the expected value of the prize you'll take home?

Input:

The first line contains two integers, N – number of fields on the wheel, and K – number of times you get to spin the wheel. The following N lines contain one integer each, representing values of the fields - $D[1], D[2], \dots, D[N]$.

Output:

Output contains exactly one real number – expected value of your overall prize, rounded to 5 decimal places.

Example input:

```
2 2
10
20
```

Example output:

```
22.50
```

Constraints:

- $1 \leq N \leq 10^4$
- $1 \leq K \leq N$
- $1 \leq D[i] \leq 10^4$

Time and memory limit: 0.5s / 64 MB

Solution and analysis:

In order to calculate the expected prize value, we can observe the expected value that we can gain from each field. By the rules of the game, for each field j we can get either 0 points if we never hit that field in our K

spins, or exactly $D[j]$ points if we hit j at least once (i.e. we get the same score for field j no matter how many times the wheel stops at that field). Thus, the expected overall prize can be calculated as the expected sum of prizes each field will give us, so we have:

$$E[\text{Prize}] = E\left[\sum_{j=1}^N \text{Prize}_j\right] = \sum_{j=1}^N E[\text{Prize}_j]$$

Given that we have N fields of the same size, and we are making K random draws (i.e. wheel spins), it is obvious that for each field of the wheel we have the same binomial distribution over the number of hits after K spins:

$$P_k(K, p) = \binom{K}{k} p^k (1-p)^{K-k}$$

Here P_k stands for the probability of exactly k hits after K spins, and p denotes probability of hitting the field in a single spin, so $p = \frac{1}{N}$.

Now we can write the distribution over prize value for each field:

$$\text{Prize}_j = \begin{pmatrix} 0 & D[j] \\ P_0(K, p) & \sum_{i=1}^K P_i(K, p) \end{pmatrix}$$

In order to avoid dealing with binomial coefficients, we can rewrite the above distribution in simpler terms:

$$\text{Prize}_j = \begin{pmatrix} 0 & D[j] \\ P_0(K, p) & 1 - P_0(K, p) \end{pmatrix}$$

so we end up with:

$$\text{Prize}_j = \begin{pmatrix} 0 & D[j] \\ \left(1 - \frac{1}{N}\right)^K & 1 - \left(1 - \frac{1}{N}\right)^K \end{pmatrix}$$

Finally, we can calculate the expected overall prize as:

$$E[\text{Prize}] = \sum_{j=1}^N E[\text{Prize}_j] = \sum_{j=1}^N D[j] \cdot \left(1 - \left(1 - \frac{1}{N}\right)^K\right)$$

which yields an easy $O(N)$ solution.

Problem C: MaxDiff

Author: **Milan Novaković**

Implementation and analysis: **Andrija Jovanović**

Statement:

You are given an array of integers A of length N . We will define $S(A)$ as the sum of absolute differences between all pairs of consecutive elements in A . More formally, assuming that A is zero-based:

$$S(A) = \sum_{i=1}^{N-1} |A[i] - A[i-1]|.$$

Your task is to find the permutation $p(A)$ of the array A for which the value $S(p(A))$ is maximized.

Input:

The first line of input will contain one integer N , representing the size of the array A . The second line will contain N space-separated integers, representing the elements of the array.

Output:

The first and only line of output should contain a single integer equal to the largest sum of differences of consecutive elements obtainable from A as described in the problem statement.

Example input:

```
3
2 3 5
```

Example output:

```
5
```

Example explanation:

There are six possible ways to reorder the array: $(2, 3, 5)$; $(2, 5, 3)$; $(3, 2, 5)$; $(3, 5, 2)$; $(5, 2, 3)$; $(5, 3, 2)$. The sums of differences are then respectively 3, 5, 4, 5, 4 and 3, and the largest among them is 5.

Constraints:

- $1 \leq N \leq 1,000,000$
- $-2^{32} \leq A[i] \leq 2^{32} - 1$ ($0 \leq i < N$)

Time and memory limit: 1.0s / 64 MB

Solution and analysis:

It is obviously infeasible to generate all permutations of A , calculate the value S for each one and pick the maximum, so let's try to observe some things about the problem that will help us reduce the space of possible solutions.

We will assume that all elements in the array are distinct. The proofs for the case when equal elements are allowed are slightly more difficult and there is a number of corner cases that have to be taken care of, so they will be left to the reader as an exercise ☺

First, let's notice a relatively obvious but very important fact: there will always exist an optimal solution in

which the elements are sorted in a “zig-zag” manner, i.e. it will not contain a triple of consecutive elements such that $A[i] < A[i + 1] < A[i + 2]$ (or $A[i] > A[i + 1] > A[i + 2]$). Proving this is easy: if we have a triple satisfying this condition, we can just pull out its middle element and place it at the end of the array – it is trivial to verify that S cannot decrease after this transformation.

The other fact is slightly harder to notice. Let’s denote the median of A with m . (A reminder: the median of an array is the middle element of the sorted array if the number of elements is odd, and the average of the two middle elements if the number of elements is even). Clearly m depends only on the elements of A and not on the permutation. We will prove the following:

Lemma. There is an optimal solution in which there are no two consecutive elements that are either both larger or both smaller than the median.

Proof. The first thing to notice here is that, due to the “zig-zag” principle discussed above, an optimal solution can’t contain a sequence of exactly two consecutive elements on the same side of the median. Let’s assume that there are at least three such consecutive elements. It is easy to see that we can always pick *exactly* three consecutive elements from this sequence such that $A[i] > A[i + 1] < A[i + 2]$. Since the rest of the array now has at least two more elements that are under the median than elements that are over it, we can use the same reasoning to conclude that somewhere else in the array there are three consecutive elements under the median, ordered as $A[j] < A[j + 1] > A[j + 2]$. Since $A[i + 1] > A[j + 1]$ (the former is over the median and the latter under it), we can swap these two elements and get a solution that preserves all the inequalities and is strictly better than the previous one.

Now we have enough information to deduce the most important statement:

If the order of elements in A satisfies the two principles described above, its value of S is

$$S(A) = |A[0] - m| + |A[N - 1] - m| + \sum_{i=1}^{N-1} 2 \cdot |A[i] - m|,$$

where m is the median of A .

It should be clear that this holds from the following argument: since for all i elements $A[i]$ and $A[i + 1]$ aren’t on the same side of the median, their absolute difference is $|A[i] - A[i + 1]| = |A[i] - m| + |A[i + 1] - m|$. For each element except the first and the last one the term $|A[i] - m|$ appears twice in the final sum, while for the two edge elements it appears just once.

This gives us the final step in the solution: since all terms in the sum are non-negative, we just have to minimize the value $|A[0] - m| + |A[N - 1] - m|$. If the total number of elements is even, we pick the two middle elements for the ends – otherwise the zig-zag property would not hold. If it is odd, we pick the median element at one end and the element closest to it by absolute value at the other.

Note that we don’t even have to generate the exact permutation, since all permutations $p(A)$ constructed in this way will have the same value $S(p(A))$ and the above discussion gives us the guarantee that permutations which don’t satisfy these conditions cannot possibly result in a better solution.

The implementation ends up being very simple: first we find the median of the array A , then we find the edge elements as described in the previous paragraph, and finally we sum up the absolute differences of the elements from the median, multiplying by two for all except the leftmost and the rightmost element.

There are still some traps that need to be avoided – edge cases with a very small number of elements need to be dealt with, the solution has to be kept in a 64-bit value, repeating values can pose a problem for

certain implementations. However, none of that should present a serious challenge for any competitor with decent technique.

Complexity

There is a choice for the algorithm used to calculate the median. The simplest way is to sort all the elements and pick the one(s) in the middle, which takes $O(N \log N)$ time. We can do better – the well-known **quickSelect** algorithm gives expected $O(N)$ time. Although its running time depends on the pivot choice and its worst-case complexity is $O(N^2)$, median-of-three or just random pivot choice should be enough since none of the test cases targeted this scenario (at least not intentionally). For the more paranoid contestants, the pivot can be chosen using the **median-of-medians algorithm**, which guarantees $O(N)$ running time but is tricky to implement and slower on average than simple quickSelect.

The rest of the algorithm can be done in a single pass of the array, giving an overall $O(N)$ time complexity of the algorithm. The memory complexity is obviously $O(N)$.

Problem D: Cars

Author: **Mladen Radojević**

Implementation and analysis: **Dimitrije Filipović**

Statement:

There are n cars parked at the parking lot and a new car is arriving. The parking lot is a space between two walls and cars are parked along one line between those walls. The driver will park his car if there is a free parking spot that is long enough (at least as long as the car). Otherwise, he will have to move a few cars in order to make appropriate space for his car. The car can be moved to the left or to the right along the parking lot, but at most until it reaches a wall or another car.

Your task is to find the minimal total distance by which currently parked cars have to be moved in order to provide enough space for the arriving car.

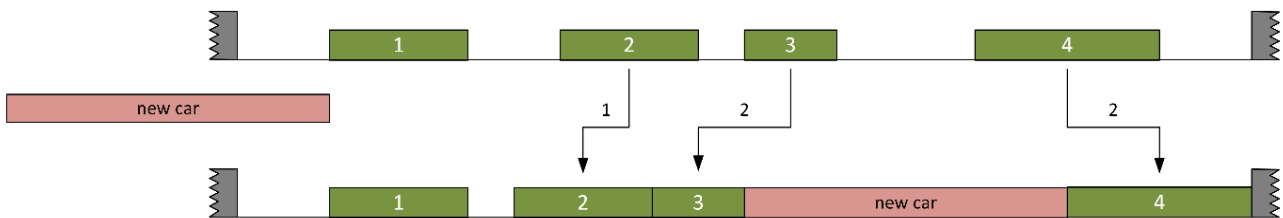


Figure 1. Optimal car moves for the given example.

Input:

The first line of input contains three space-separated integers n , lw and rw . They denote the number of cars already parked, the coordinate of the left wall and the coordinate of the right wall, respectively. Each of the following n lines contains two integers, describing a parked car: p – the coordinate of the leftmost point of the car, and l – the length of the car. The last line of the input contains one integer l_{new} , the length of the arriving car.

Output:

Output contains only one line with one integer – the sum of distances by which parked cars have to be moved to provide enough space for the arriving car. If a solution doesn't exist the output should be -1 .

Example input:

```
4 0 22
2 3
7 3
11 2
16 4
7
```

Example output:

```
5
```

Example explanation:

The best way is to push the second car to the left by 1, the third car to the left by 2 and the fourth car to the right by 2. It will create an empty space of length 7, so the new car can be parked there. The sum of all movement lengths is 5 ($= 1 + 2 + 2$).

Constraints:

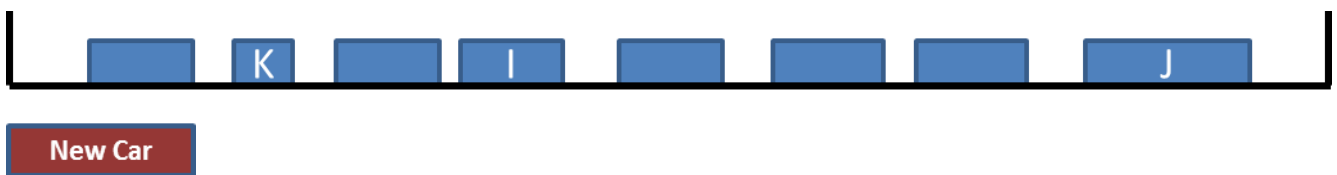
- $0 \leq n \leq 10^5$
- $0 \leq lw, rw, p[i] \leq 2 \cdot 10^9$
- $1 \leq l[i], lnew \leq 10^6$

Time and memory limit: 0.5s / 64 MB

Solution and analysis:

Let's enumerate the cars with 1 to n from left to right. (To be able to do that, we will need to sort the array of cars first). Consider each car in turn. For each car K , find the first car J ($J > K$), such that the sum of free parking spots between K and J is greater than the length of the new car. For each such pair k, j we will find the optimal solution, and then use these to compute the global minimum.

Let's consider a given pair of cars (K, J) . In the case that the pair $(K + 1, J)$ also satisfies the total empty length constraint, we can narrow down the search space by removing car K from the set of cars for which we will consider moves. We will repeat this process as long as removing the leftmost car will still satisfy the total empty space length constraint. Let's denote the leftmost car remaining in this set with I .



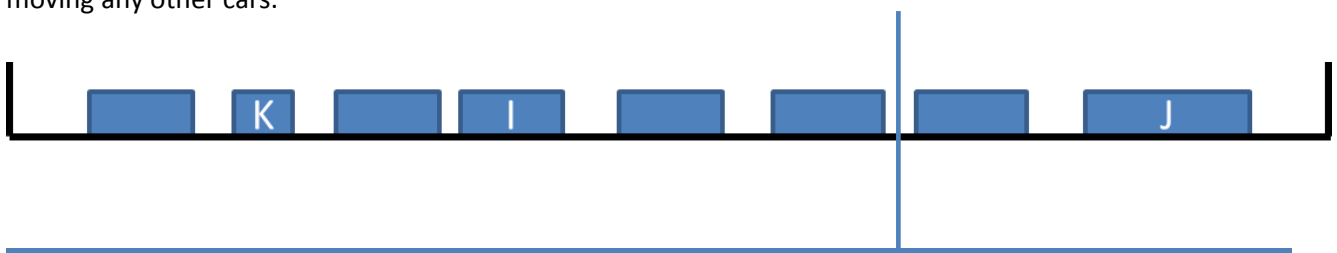
Let's enumerate empty spaces between cars I and J with numbers $0..(J - I)$. We will find the first empty space m such that $\sum_{y=0}^m Length(y) \geq \sum_{y=0}^{J-I} Length(y) / 2$. We will move every car in the direction of whichever car (I or J) is closer to it, in order to reduce the total distance covered. In this way, the upper bound on the distance any given car can move is $\sum_{y=0}^{j-i} Length(y) / 2$, whereas if we moved any car in the other direction (to the car which is farther away from it) this upper bound would be greater.

Let's define four values for every car:

- $FS_L(I) = \sum Free\ space\ to\ the\ left\ of\ I$ – Cost for moving car I to the leftmost position possible (if all cars before it were parked consecutively from the left wall, with no space between cars).
- $CM_L(I) = \sum_{X=0}^I FS_L(I)$ – total cost for moving all cars from 0 to I (inclusive) to the left wall.
- Another pair of arrays FS_R, CM_R , representing free space and total cost of moving to the right wall.

These values can be pre-computed in $O(n)$ time with two passes through the array.

We would like to compute the cost of moving all cars between I and J away from the middle (m), without moving any other cars.





Consider only cars I, L (The rightmost car moved to the left) and $I - 1$. We will separately calculate the cost of moving cars to the left and to the right. The cost can be calculated in the following way:

The cost of moving car L and all cars left of it to the left wall is $CM_L(L)$.



If we moved all cars left of car $I - 1$ (inclusive) to the left wall and all cars from I to L to the car I , the cost difference between this configuration and the previous is $FS_L(I) * (L - I)$. (It is the cost of moving $L - I$ cars to the right by the sum of empty spaces left of I).



Now we can determine the cost of moving only cars from I to L to the left towards car I . Since the cost difference between this configuration and the previous is $CM_L(I - 1)$,

This cost can be calculated as

$$\text{Cost}(I, L) = CM_L(L) - CM_L(I - 1) + FS_L(i) * (L - I)$$



In the same way (using pre-computed FS_R, CM_R), we can determine the cost of moving remaining cars between I and J to the right in the consecutive configuration without moving J .



The only issue here is that we may have created more space than what is needed for the new car. To reduce the cost we should first compare $L - I$ and $J - L$ (the number of cars we moved to the left and the number of cars we moved to the right).

Let W be the sum of empty spaces between cars I and J , and let S be the space we created for the new car.

$$W = FS_L(J) - FS_L(I)$$

$$S = \text{Length}(\text{NewCar}) - W$$

Let's say that

$$L - I \geq J - L$$

In this case, the cost for creating the empty space to fit the new car should be reduced by $S \cdot (L - I)$

$$\text{OptimumCost} = \text{Cost} - (\text{Length}(\text{NewCar}) - W) * (L - I)$$

In the case that we moved more cars to the right than to the left the same logic applies using the values from FS_R and CM_R .

After calculating *OptimumCost* for each car as a first car (I), we should easily be able to pick the best one. The time complexity of the solution is dominated by the initial sort – the rest of the algorithm is linear. This means that the overall time complexity is $O(N \log N)$.

Problem E: Triangles

Author: **Mladen Radojević**

Implementation and analysis: **Mladen Radojević**
Milan Vugdelija

Statement:

You are given an array of positive integers. Find the maximal substring (i.e. a subset of at least three consecutive elements of the array) so that any three distinct elements from that substring can form the sides of a triangle. Also, find the maximal subsequence (a subset consisting of at least three elements, not necessarily consecutive) with the same property.

Input:

The first line of input contains one integer n , the number of elements in the array. The next n lines contain the elements of the array.

Output:

Output consists of exactly two lines, each containing one integer– the length of the maximal substring and the maximal subsequence with the property described above, respectively. If such substring or subsequence doesn't exist, the corresponding value is zero.

Example input:

```
5
60
30
20
40
60
```

Example output:

```
3
4
```

Constraints

- $1 \leq n \leq 100,000$
- Elements of the array are positive integers, each less than or equal to 10^9 .

Time and memory limit: 1.5s / 64 MB

Solution and analysis:

Input size limit 100000 suggests that any solution to this problem should work in $O(n \log n)$ time (or faster).

Part a (substring):

To verify that some substring fulfills the requirement, it is enough to check if the sum of two smallest numbers in a substring is greater than the largest number in that substring. Indeed, if the inequality holds for these three elements, it will hold for any three numbers in that substring.

There are several ways to find the length of the longest such substring, two of which will be explained here.

Solution 1:

Suppose that we want to check if there exists a substring of length k with the described property. We can divide input array into slots of length k (the last slot may have less than k elements). For each slot $a[0] \dots a[k - 1]$, we can compute the following arrays:

- Prefix maximum: $leftMax[i] = \max\{a[0], a[1], \dots, a[i - 1]\}$
- Suffix maximum: $rightMax[i] = \max\{a[i + 1], a[i + 2], \dots, a[k - 1]\}$
- Prefix minimum: $leftMin[i] = \min\{a[0], a[1], \dots, a[i - 1]\}$
- Suffix minimum: $rightMin[i] = \min\{a[i + 1], a[i + 2], \dots, a[k - 1]\}$
- Prefix second minimum: $leftMin2[i] = \min(\{a[0], a[1], \dots, a[i - 1]\} \setminus \{leftMin[i]\})$
- Suffix second minimum: $rightMin2[i] = \min(\{a[i + 1], a[i + 2], \dots, a[k - 1]\} \setminus \{rightMin[i]\})$

With this pre-calculation, we can find the maximum, minimum and second minimum of any substring of input array of length k in constant time. Namely, any substring of length k covers entirely one slot or lies in two consecutive slots, so min and max are straightforward to compute using suffix arrays of the left slot and prefix arrays of the right slot, while computing second min requires several comparisons between minima and second minima of two parts of the substring.

Acting as described, it is possible to check all substrings of length k in linear time. Doing binary search on k gives us an $O(n \log n)$ algorithm for the original problem.

Solution 2:

We start with the substring consisting of the first three elements of the input array. If the current substring has the required property, we move the right boundary of the substring forward, introducing a new element into it; otherwise we move forward the left boundary of the substring, removing one element from the substring.

To check if the substring has the triangle property, we can use one heap that extracts maximum, and one that extracts minimum - let's call the heaps M and m respectively. When the right boundary moves, we just put a new element into both heaps. Moving the left boundary requires removing one particular element from both heaps. Instead of removing that element immediately, we can use two auxiliary heaps (again, one for max and one for min, call them M_a and m_a) and put the element that should have been removed there. As long as that element is not equal to max of the heap M , it doesn't matter if it is present in the heap or not. So every time we extract the max of heap M , we also extract the max of M_a ; if they are not equal, the max is regular and we can use it (we just put back the max of M_a); if the two maxima are equal, we do the (delayed) removal from both M and M_a , and get another max from both until they differ. We do the same with heaps m and m_a for extraction of the minimal element.

Getting min/max from the heap and putting a new element into all 4 heaps requires $O(\log n)$ time, so we can check one particular substring in logarithmic time. Since after each check one of the substring's boundaries moves forward, there are $O(n)$ substrings to check, so the total running time is again $O(n \log n)$.

Part b (subsequence):

It is easy to prove that if some elements of a sorted array form a subset with the described property, then the entire segment (from minimal to maximal element of the subset) also has the property. So, the maximal subset of the original array with the described property is a substring of the sorted array.

Therefore, to solve part b, it is enough to sort the input array and then search for the longest substring

using (any) solution of part a. The running time of such algorithm would be $O(n \log n)$ for sorting and $O(n \log n)$ for finding the longest substring, which gives $O(n \log n)$ in total.

It is also possible to find the longest substring with the given property in a sorted array more directly. Obviously, the two smallest elements are the first two elements of a substring, and the largest is the last one, so there is no need to use heaps or auxiliary prefix/suffix arrays to find minima and maxima of a substring. The running time in this case is still $O(n \log n)$ due to sorting, even though finding the longest substring in a sorted array can be done in linear time.

Problem F: Olympic Games

Authors: **Mladen Radojević**

Implementation and analysis: **Mladen Radojević**

Statement:

Young boy, Oliver, has watched the Olympic Games this year for the first time. The number of countries which participated in the Olympics is n . There are k different sports and each country had its own representative in some of the k sports. In each sport the gold medal is won by exactly one country among the ones that have representatives for that sport. And, of course, for every sport there is at least one country which competes in it.

Oliver noticed that a small number of countries won a huge number of gold medals, and that a lot of countries didn't win any. Now, he is wondering what could be the minimal difference in the number of gold medals between the country which took the most and the country which took the least. Oliver is still too young to figure out the answer to this question, so please help him.

Input:

The first line contains n , the number of participating countries. The second line contains k , the number of sports. The third line contains the total number of competitors, m . Each of the next m lines contains two integers, c and s , which mean that country c had a representative in sport s .

Output:

Output contains only one integer – the minimal possible difference in the number of gold medals between the country which took the most gold medals and the country which took the least.

Constraints:

- $2 \leq n \leq 100$
- $1 \leq k \leq 500$
- $0 \leq c_i \leq n - 1, 0 \leq s_i \leq k - 1$ for each i
- No pair (c_i, s_i) is contained in the input more than once

Example input:

```
3
4
6
0 0
0 1
0 2
1 2
1 3
2 3
```

Example output:

```
1
```

Time and memory limit: 0.5s / 64 MB

Solution and analysis:

Let's describe the algorithm for solving this problem. Initially, no sport is assigned to any country. Going circularly through all countries we try to assign new sport to the currently considered country, while maintaining the number of sports assigned to other countries (actually, we try to find an ***augmenting path*** in a bipartite graph which starts from the country being considered and ends at some yet unused sport). If at some moment no new augmenting path for a particular country exists, we can skip that country in all subsequent iterations in order to save time. Since for every sport there is at least one country which competes in it, after a certain number of iterations each sport will be assigned to some country.

By this algorithm we get a matching where the number of gold medals taken by country with the minimal number of gold medals is maximal and the number of gold medals taken by the country with the maximal number of gold medals is minimal. Therefore, this matching has the desired property that the difference between these two values is minimal.

As the number of iterations which can find augmenting paths is at most $k + n - 1$ (there are at most $n - 1$ unsuccessful findings) and each augmenting path can be found in $O(E)$ time, where E is the number of edges, the time complexity of the solution is $O((n + k) \cdot n \cdot k)$. The memory complexity is $O(n \cdot k)$.

Problem G: Matrix

Author: **Andreja Ilić**

Implementation and analysis: **Andreja Ilić**

Statement:

You are given a square binary matrix A of dimension $N \times N$. Elements on the main diagonal are all ones. We want to compute the $77,686,783^{\text{th}}$ power of this matrix (MDCS written in ASCII codes is (77,68,67,83)). To make things more interesting, we will define binary operations $+$ (addition) and \cdot (multiplication) as the following:

+	0	1
0	0	1
1	1	1

\cdot	0	1
0	0	0
1	0	1

So basically, addition is logical OR and multiplication is logical AND.

The input matrix is too big for normal time constraints, so it will be given by listing all positions of ones in it. Also, for the output only the number of ones in the $77,686,783^{\text{th}}$ power of the matrix is sufficient.

Input:

The first line contains two integers, N and M - dimension of the square matrix A and the number of ones in it. Each of the next M lines contains two integers x and y – which means that $A_{x,y}$ is equal to 1.

Output:

Output contains only one integer– the number of ones in the $77,686,783^{\text{th}}$ power of the given matrix.

Example input:

4 8
1 1
1 3
1 4
2 2
2 3
3 1
3 3
4 4

Example output:

11

Example explanation:

From the input we have that $A = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$. Its $77,686,783^{\text{th}}$ power is $\begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, which has 11 ones in it.

Constraints:

- $1 \leq N \leq 5,000$
- $N \leq M \leq 200,000$

- Indices x and y from the input satisfy the condition $1 \leq x, y \leq N$ and these pairs are unique.
- The matrix elements are 0 or 1 and the elements on the main diagonal all ones. All elements that are not listed in the input have zero value.

Note:

For two square matrices A and B with dimensions $N \times N$, we say that matrix C , with the same dimension, is product of these two matrices if:

$$C_{i,j} = A_{i,1} \cdot B_{1,j} + A_{i,2} \cdot B_{2,j} + \dots + A_{i,N} \cdot B_{N,j}, \text{ for every } 1 \leq i, j \leq N$$

Time and memory limit: 2.0s / 64 MB

Solution and analysis:

At first glance, this problem requires fast multiplication of the sparse binary matrices with a given definition for operations. This is known and very hard problem for implementation. The standard approach for matrix multiplication gives as time complexity $O(n \cdot n^3)$ which is very big for the problem constraints. Logarithmic powering is also too slow. But, in our case solution has nothing to do with this – this is a **graph theory problem**.

Before we start the analysis of this problem, let's look at the adjacency matrix A of an arbitrary directed graph $G = G(V, E)$. As we know, the adjacency matrix is a binary one and its element $A[v][u]$ is equal to 1 if and only if there is an edge from vertex v to vertex u (directed edge). The adjacency matrix is a square one, so it is allowed to consider powers of this matrix: A^k , for every $k \in N_0$. From now on we will assume that operations are defined as in the problem statement.

Can we, with some corresponding graph property, define the square of the matrix A ? The element at position (v, u) is going to be equal to one if and only if there is a vertex k such that $A[v][k] = 1$ and $A[k][u] = 1$. This means that $A^2[v][u]$ is equal to 1 if and only if there is a path of length 2 in the starting graph G . Using mathematical induction we can prove the following property:

$$A^k[v][u] = 1 \text{ if and only if there is a path from vertex } v \text{ to vertex } u \text{ of length } k.$$

We have an additional property of the start matrix: elements on the main diagonal are ones. This means that all of the vertices have loops. In other words, we can "circle" around any vertex for an arbitrarily long time. So, if there is a path of length k between vertices v and u , then there is path of length k_1 between them for every $k_1 > k$ (we can just append a "circle" of length $k_1 - k$ to path). This means that, with the above property of matrix A , we have

$A^k[v][u] = 1$ if and only if there is a path from vertex v to vertex u of length less than or equal to k .

Now we can go back to our original problem. In the input we have a directed graph G with n vertices and m edges, where every vertex has a loop. From the above definition of the element $A^k[v][u]$, we have that starting from $k > n$ power of matrix A is not going to change. This is very important, because in this way we have to calculate the n -th power, and of course $n < 77,686,783$. The problem can be reformulated as:

*find the number of edges in the **transitive closure** of the given graph G ,
i.e. for every vertex calculate the number of vertices that are reachable from it*

Naïve approach for the transitive closure leads to complexity $O(n \cdot m)$ – graph tour, DFS or BFS, from every

vertex separately. A better idea is to find the **strongly connected components (SCC)** first. In this way, submatrices for every component have all elements equal to one (so we do not want to “waste” time there). After finding the SCCs, we can shrink every component to just one “vertex”. In this way we can obtain a **directed acyclic graph (DAG)**.

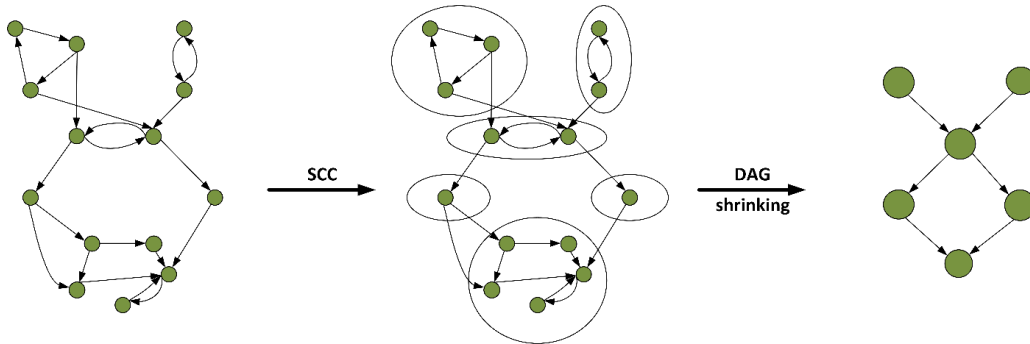


Figure 1. Creating DAG graph from SCC components

Things are a little bit easier. If we assume that for every component we have a list of all components that are reachable from it, we can easily transfer this to the start graph and calculate the final result. But how can we initialize these lists with the given time constraints?

For DAG we can find its **topological sort** order. We can initialize the lists for every component in this order, because it holds that by the time when we are examining some component, all components reachable from it are already initialized. Let us denote with *currentC* the current component for which we want to initialize the list. Unfortunately, we cannot simply “connect” all lists from their neighbors, because there can be some duplicates (see Figure 2 for example). On the other hand, these lists can be $O(n)$ long, so the union of these sets must require passing through them multiple times.

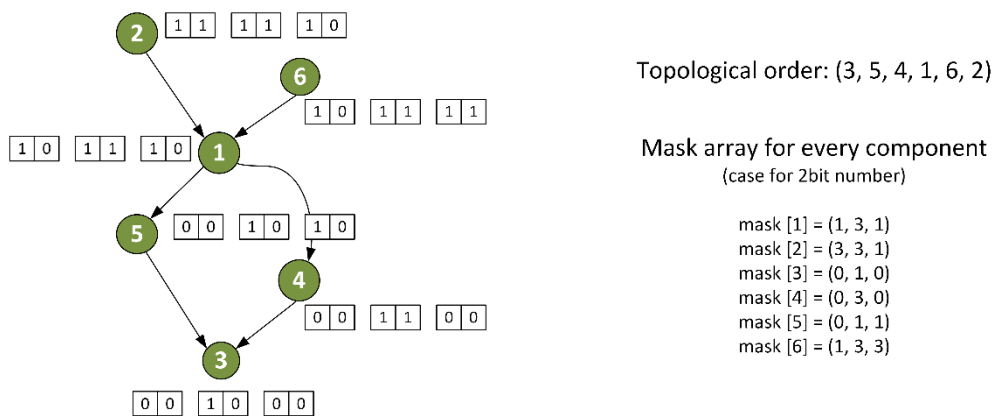


Figure 2. Example of a topological sort and mask arrays (for the case of 2-bit numbers)

The best way to maintain these “lists” is to store them in some sort of marked array (when a component is in the “list” we are going to mark the corresponding element). If we use simple boolean arrays, complexity will again be $O(n \cdot m)$. Idea is to use **bit masks**. For every component we are going to store an array $mask_{currentC}$ of length $\lceil \frac{numComponents}{64} \rceil$ of long type. This way we can mark some component *c* in this array as

$$mask_{currentC}[c \text{ DIV } 64] = 2^{c \text{ MOD } 64}$$

In other words, for each component there is a unique corresponding bit in every array. Now, we can initialize the array for *currentC* by simply *OR*-ing the arrays for its neighbors element-wise (which represents union). Note that although the complexity remains $O(n \cdot m)$, the reduction of the constant factor is very significant.

Complexity:

In this problem we have quite a pipeline of graph algorithms. First, the complexity of finding SCCs and building a DAG is $O(m)$. Finding the topological order has the same complexity. Finally, performing the dynamic programming approach for initialization of bit mask arrays as described above has $O(\frac{n \cdot m}{64})$ complexity if we use a 64-bit integer type for bit masks. Indeed, every array is going to be iterated for every component which has an edge directed at the corresponding component for that array. So, for every edge we have one tour through some array. This brings us to the final complexity of $O(\frac{n \cdot m}{64})$.

Test data:

The data corpus for this problem consists of 24 test cases. Test cases are created with one (or more) of the following methods:

- Random generation of a binary matrix with given probability for 1 and 0
- Generation of a matrix that corresponds to a tree with some additional cross or / and up edges
- Generation of a matrix that corresponds to a path with some additional edges
- Creating a SCC graph from a tree with cross edges and blossoming a SCC components in every vertex
- Special cases (triangle, all ones, only loops...)

Num	N	M	Comment
00	4	8	test case from problem statement
01	10	31	by hand
02	100	5187	Random with $p = 0.5$
03	1000	1000	All zeros except on the main diagonal
04	500	125250	One in the upper triangle
05	2000	161804	Random with $p = 0.0.3$
06	2500	127736	Random upper triangle with $p = 0.03$
07	3000	5999	Random tree structure
08	3000	36871	Path with down edges
09	3000	193412	Expanded SCC graph with $numComp = 450$
10	5000	161017	Expanded SCC graph with $numComp = 1000$
11	4000	67879	Random with $p = 0.0.3$
12	3000	182515	Random tree structure with cross edges
13	5000	9999	Random tree structure
14	5000	59870	Random tree structure with down cross edges
15	5000	194846	Expanded SCC graph with small $numComp$
16	5000	179948	Random with $p = 0.0.3$
17	5000	196411	Path with down edges
18	5000	189283	Random tree structure with cross edges
19	5000	188570	Random tree structure with down cross edges
20	5000	184753	Expanded SCC graph with $numComp = 1000$
21	1	1	Only one vertex
22	4500	118341	Components: SCC, Tree, Path, Random
23	5000	143657	Components: big SCC, Tree, Path, Random

Table 1. Test data description

Problem H: String covering

Author: **Andreja Ilić**

Implementation and analysis: **Andreja Ilić**

Statement:

We say that string B covers string A if A can be obtained by putting together several copies of string B , where overlapping between two successive copies of B is allowed but the overlapped parts must match. After connecting these copies the whole generated string must match string A .

For example, string $B = abab$ covers string $A = ababab$, with two copies (see Figure 1). String $B = aba$ does not cover A , because the last character b cannot be covered.

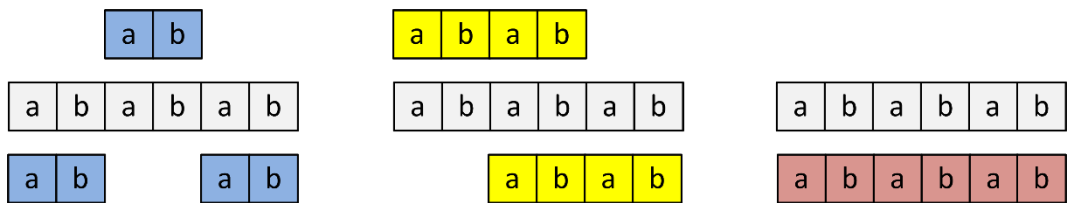


Figure 1. All possible coverings of string $A = ababab$.

You are given string A . Write a program that calculates how many strings B exist that cover string A in this way.

Input:

The first and only line of the input contains string A .

Output:

Output contains only one integer – the number of different strings that cover given string A .

Constraints:

- $1 \leq \text{length}(A) \leq 100,000$
- String A consists exclusively of letters 'a' - 'z'.
- Output the final solution modulo $10^9 + 9$.

Example input:

ababab

Example output:

3

Time and memory limit: 1.0s / 64 MB

Solution and analysis:

Problem H on this year's BubbleCup finals was a string problem. For a given string A it was asked to find how many strings there exist that covers this string. We say that string B covers string A if A can be

obtained by putting several copies of string B where overlapping between two successive copies of B is allowed but the overlapped parts must match. After connection of these copies whole generated string must match string A . First thing that we can observe is that necessary condition for string B is that it is some prefix of the string A . So, the asked modulo in the task description is only a small trick.

Now we know that the final answer is going to be smaller or equal to n (solution is equal to n if and only if all char in A are equal). Naïve approach would be to check every prefix of the string A . When testing for the prefix $A^k = a_1a_2 \dots a_k$ we have to find all occurrences of it as substrings in A and then see if these occurrences covers whole string A (here we can see that string B has to be an suffix as well). Complexity of this algorithm is approximately $O(n^3)$, but it can be reduced to quadratic with some hash functions for substrings. In any case this is too big for our constraints.

Can we, in some other way, track these occurrences for prefixes? Let us assume that for the prefix A^k occurrences in the string A starts at positions $1 = s_1 < s_2 < \dots < s_m$. For the prefix A^{k+1} start positions are subset of the start positions for A^k . Idea is to store these occurrences in some nice way that can be updated fast when adding new character on the end (moving to the next prefix).

For this we can use data structure: **suffix array**. Let us denote with s suffix array - $s[i]$ is equal to the start position of the i -th suffix in the lexicographic order. Then the positions for prefixes A^k are successive in the array s . For every prefix we can define some segment in the suffix array $[l_k, r_k]$ which means that suffixes in this segment starts with given prefix. Nice thing is that these segments create an inclusive chain: $[l_{k+1}, r_{k+1}] \subset [l_k, r_k]$. In this way we can easily obtain segments.

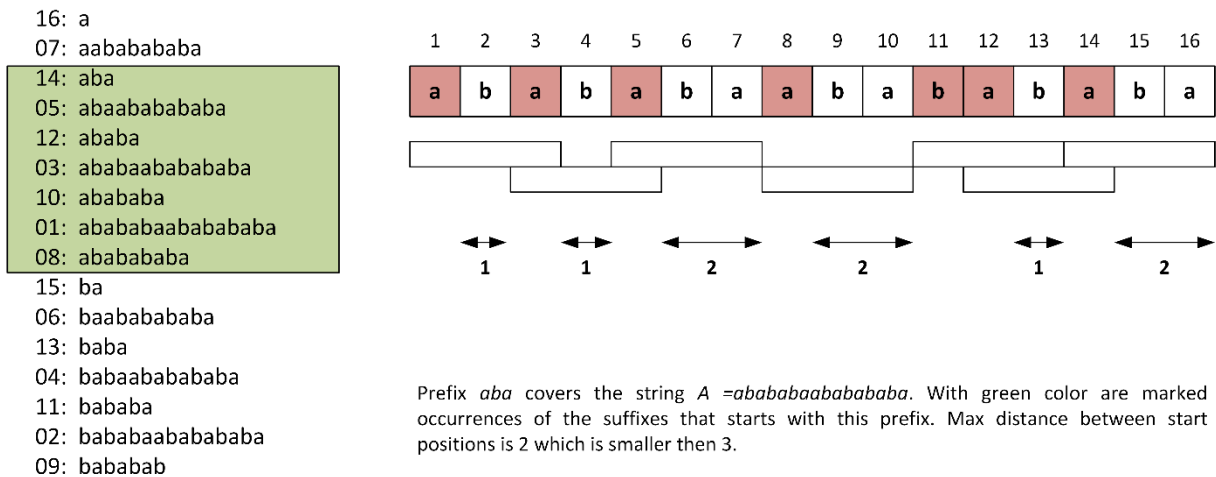


Figure 1. Example of suffix array and prefix check.

Now we have the start positions of occurrences for any prefix. Problem is to, in some efficient way, see if they cover the whole string A . For this we must use additional data structure – **max heap**. In heap we are going to store the lengths between two successive positions - gaps. When we remove some occurrence, we will remove two distances / gaps (from prior to current one and from current one to next occurrence) and add the new one which is the sum of the removed ones (from prior to next). Finally, we can state that current prefix covers the whole string if and only if the max element in the heap (max distance between two successive occurrences) is smaller or equal to the prefix length.

```
=====
01 initialization of the suffix array S;
02 sol = 0;
03 for k = 1 to n - 1 do
```

Problem H: String covering

```

04     add in heap key-value pair (k, 1);
05     segmentLeft = 1; segmentRight = n;
06     for k = 1 to n do
07         while (char at position k of segmentLeft-th suffix is different from A [k]) do
08             remove from heap key prior and segmentLeft;
09             add in heap key-value pair (prior, next - prior);
10             inc(segmentLeft);
11         while (char at position k of segmentRight-th suffix is different from A [k]) do
12             remove from heap key prior and segmentLeft;
13             add in heap key-value pair (prior, next - prior);
14             dec(segmentRight);
15
16         if (max in heap <= k)
17             inc(sol);
18     endfor

```

=====

Figure 1. Pseudo code for described algorithm

Complexity:

Sorting the suffixes with suffix array takes $O(n \log n)$ time. This can be implemented in the linear time but in our case this is sufficient. After that for every prefix, in order as in the given string, we are going to maintain the above segments in linear time overall (in every step we are going to shrink current segment with only one comparing of chars). For heap, every position is going to be added exactly one time and removed at most one time. Taking all of this in to account we get the final complexity: $O(n \log n + n \cdot 3 \log n) = O(n \log n)$.

Test data:

Test corpus consists of 15 test cases. In the random string we can expect only one covering string – the string itself. Many ideas for test generation is to create some “recursive” string that has many occurrences of prefixes in it. The “worst” case for this is to use small number of different chars. Description of the test data with some comments is given below.

Num	N	Solution	Comment
00	6	3	<i>ababab</i>
01	45690	9138	Concatenation of the string <i>aabcc</i>
02	90010	10	Concatenation of the string <i>a..ab..ba..ag</i>
03	90000	10000	Concatenation of the string <i>aaaabcaaaa</i> with random change
04	80.000	1	Many <i>a</i> with random char <i>b</i>
05	10.002	3334	Concatenation of the string <i>aab</i>
06	98.904	8242	Concatenation of the string <i>aaaaaabaaaaaaaaaabaaaaa</i>
07	50.003	1	Random concatenation of strings <i>aab</i> and <i>aaab</i>
08	96.048	16	Concatenation of the string <i>acbacb</i> with some <i>xyz</i>
09	99.999	1	String is of the form <i>a..ab...ba..a</i>
10	99.999	99.999	All chars are the same
11	78950	3158	Concatenation <i>abcd...xyz</i>
12	100.000	1	Many <i>v</i> with random char <i>w</i>
13	1	1	Only one char
14	90.000	5628	Concatenation of the string <i>abababaababababa</i>
15	100.000	100	Concatenation of the string <i>aa..ab</i>

Table 1. Test data description

Problem I: Polygons

Author: **Miroslav Bogdanović**

Implementation and analysis: **Miroslav Bogdanović**

Statement:

You are given n points with integer coordinates in the plane. After that you are given q queries.

Each query gives you a list of indices in the originally given set of points. The points in this list form a simple polygon (one that does not intersect itself). For each query you should output how many points from the original set are on the inside of the given polygon.

Input:

The first line of the input contains two integers n and q , separated with an empty space. Next n lines each contain two numbers, x and y - coordinates of a point. The q lines that come after that each start with a number p , the number of points that form that polygon. The rest of the line consists of p space-separated numbers that represent indices (indices are from 0 to $n - 1$) of originally given points that form the polygon.

Output:

You should output q lines, one for each query. Output for each query should be just one integer number: the number of points from the original set on the inside of the polygon given in that query.

Example input:

```
7 2
0 0
0 4
4 0
4 4
2 1
3 2
2 3
3 0 1 3
3 0 2 3
```

Example output:

```
1
2
```

Example explanation:

First polygon is a triangle whose vertices are $(0, 0)$, $(0, 4)$ and $(4, 4)$. There is one point on its inside: $(2, 3)$. Second triangle has the vertices $(0, 0)$, $(4, 0)$ and $(4, 4)$. There are two points on its inside $(2, 1)$ and $(3, 2)$.

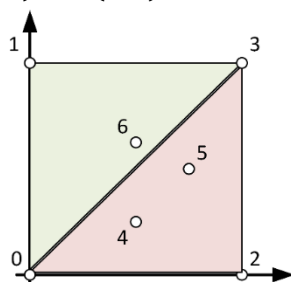


Figure 1. Visualization of the given example.

Constraints:

- $1 \leq n \leq 1,000$
- $1 \leq q \leq 10,000$
- Coordinates of a point are in the segment $[0, 10^6]$.
- No two points are the same. Also, no three different points are on the same line.
- Vertices of the polygon do not count as being inside of it.

Time and memory limit: 6.0s / 64 MB

Solution and analysis:

Taking each query and checking each point is on the inside of that polygon in a straightforward manner would take $O(n^2 \cdot q)$ time, which is too slow for the given constraints.

We are going to deal with this by precalculating some things, which will allow us to answer each query in linear time with respect to the number of vertices of the polygon. For each two points from the original set, we calculate the number of points under the line segment connecting them (that are contained in the quadrilateral formed from the endpoints of the line segment and their projections on the x axis; we don't count points on the edges of this quadrilateral). Also, we calculate the number of points directly under each point from the set (i.e. the ones that have equal x and smaller y coordinates).

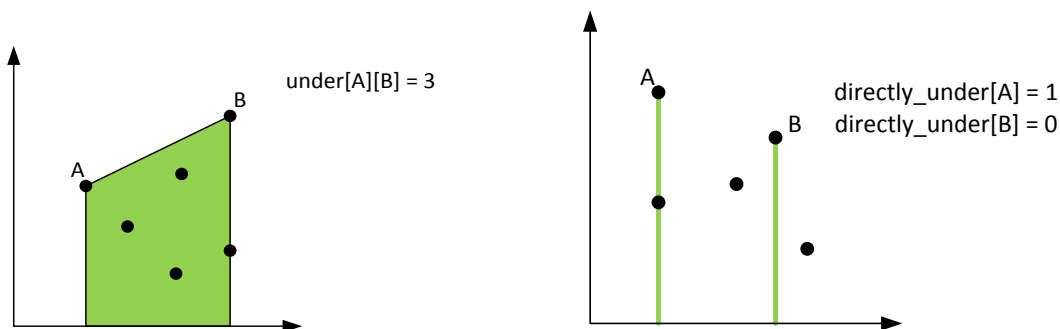


Figure 1. What is precalculated.

First we sort the points by x coordinate, sorting points with equal x by y . The number of points directly under each point is easily calculated in linear time from this sorted array.

Now for each point (we'll call it point A) we take all the points after it in this sorted array (those that have larger x , or equal x and larger y coordinate) and sort them by angle in respect to point A. We go through these points in this order (we'll call the current point B). For each point A we keep an array that counts the number of times each x coordinate has appeared in points B that we went through until now. We keep this array as a cumulative table in order to be able to do insertions and calculations of the sum of the first k elements in $\log(MAX_X)$ time for each operation, where MAX_X is the maximal value of x coordinates among the set of points. For each point B we do the following:

1. We add 1 to the cumulative table in the position of the x coordinate of point B.
2. We calculate the number of points under the line segment A-B as sum in cumulative table to the position of the x coordinate of point B – 1.

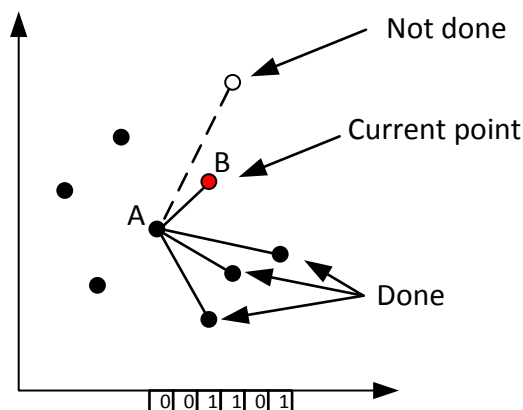


Figure 2. A typical state of a precalculation step.

We respond to each query in the following way. We go through the polygon in clockwise direction and for each edge, if it goes to the right (the x coordinate of its second point is larger than the x coordinate of its first point) we add the number of points under that edge to the sum, otherwise we subtract this number from the sum. For each vertex, if we go through it going to the right (the edge coming into it and the one going out of it are both to the right) we add the number of points under it to the sum, if we go through it going to the left we subtract the number from the sum.

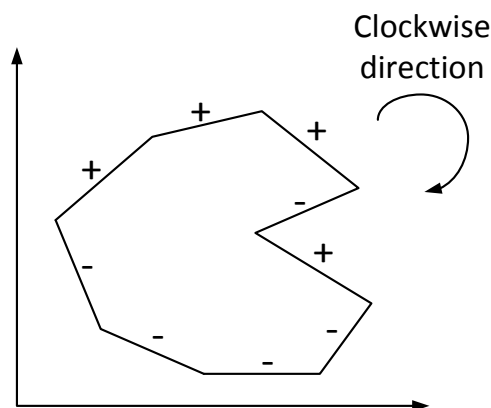


Figure 3. Responding to a query.

After going through all edges and vertices of the polygon, each point outside of the polygon is counted zero times and each point on the inside of the polygon is counted exactly once, which is exactly what we need.

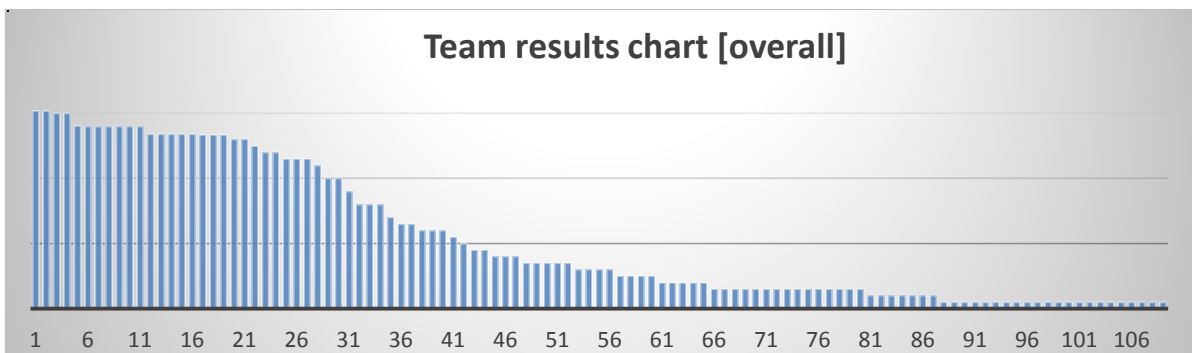
Complexity:

Time complexity of initial calculation is $O(n^2 \cdot \log(MAX_X))$. After that each query is resolved in $O(n)$ time. The total complexity is therefore $O(n^2 \cdot \log(MAX_X) + q \cdot n)$.

Qualification

The qualification rounds were originally intended to have the same format as in the previous years – two rounds with 10 problems each, with each problem in the first round being worth 1 point and each second-round problem being worth 2 points. The most important change was that the Timus online judge system was no longer used - Sphere Online Judge (www.spoj.pl) provided the problems and the judging system this year.

BubbleCup has continued getting more and more popular with every year, and this year's edition again broke all the records, with 109 teams submitting at least one correct solution. It also continued spreading geographically, with teams from countries such as Germany, Poland, Taiwan and Vietnam participating for the first time and having lots of success as well – one German and one Polish team ended up participating in the finals.



The strength of the teams has increased as well. The tasks were not any easier than in the previous editions, but the results were excellent. This meant that 29 teams solved all problems in the first round, and 4 of those solved all problems in the second round as well, so for the first time in BubbleCup history there was a team which got the maximal number of points in the qualifications. This record-breaking team was “Vanja, Nenad and the sandwich maker”, and they were quickly followed by “S-Force”, “NO Ex1t” and “koko koko euro spoko”.

In the end, the problems from the qualification rounds were solved so well that it was not possible to separate the teams. So the scientific committee decided to organize a third round for the first time in BubbleCup history. In the third round, teams with 26 points were called upon to select tasks for each other to solve, adding yet another strategic dimension to the contest.

The full statistics from the qualification rounds are shown in the tables below:

Num	Problem name	Code	Accepted solutions
01	November Rain	RAIN1	65
02	Ambiguous Permutations	PERMUT2	172
03	Roll Playing Games	RPGAMES	50
04	Manhattan Wire	MMAHWIRE	44
05	Spheres	KULE	67
06	Sightseeing	GCPC11H	58
07	Segment Flip	SFLIP	43
08	Building Construction	KOPC12A	114
09	It's a Murder!	DCEPC206	104
10	Words on Graphs	AMBIG	51

Table 1. Statistics for Round 1

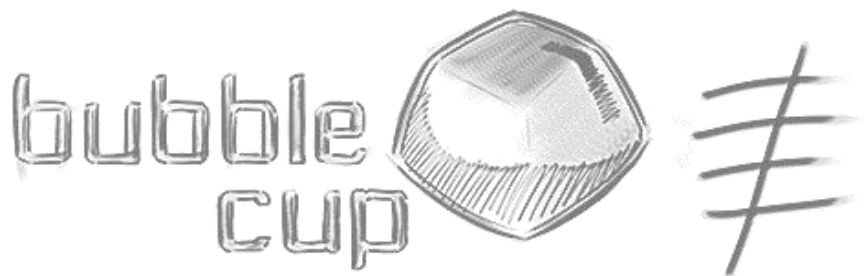
Num	Problem name	Code	Accepted solutions
01	Zig-Zag Permutation	ZZPERM	25
02	DEL Command II	DELCOMM2	17
03	Boxes	BOX	4
04	Cryptography	CRYPTO	34
05	Slow Growing Bacteria	SBACT	44
06	Reverse the Sequence	REVSEQ	35
07	Cover the String	MAIN8_E	48
08	Dynamic LCA	DYNALCA	28
09	Magic Bitwise AND Operation	AND	30
10	Contaminated City	CONTCITY	32

Table 2. Statistics for Round 2

Num	Problem name	Code	Accepted solutions
01	Four Mines	MINES4	5
02	Lost in Madrid	LIM	11
03	Circles	MINES4	9
04	Bridges! More bridges!	BRIL	9
05	Polynomial $f(x)$ to Polynomial $h(x)$	MINES4	10
06	Factorial Challenge	FUNFACT	10
07	Hi6	HISIX	8
08	Frequent Values	FREQUENT	11

Table 3. Statistics for Round 3

The organizers would like to express their gratitude to everyone who participated in writing the solutions.

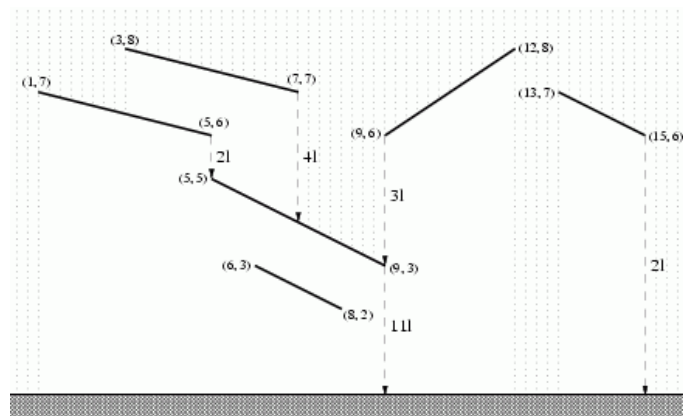


Problem R1 01: November Rain (code: RAIN1)

Resource: ACM Central European Programming Contest, Warsaw 2003

Time Limit: 13 second

Contemporary buildings can have very complicated roofs. If we take a vertical section of such a roof it results in a number of sloping segments. When it is raining the drops are falling down on the roof straight from the sky above. Some segments are completely exposed to the rain but there may be some segments partially or even completely shielded by other segments. All the water falling onto a segment as a stream straight down from the lower end of the segment on the ground or possibly onto some other segment. In particular, if a stream of water is falling on an end of a segment then we consider it to be collected by this segment.



For the purpose of designing a piping system it is desired to compute how much water is down from each segment of the roof. To be prepared for a heavy November rain you should count one liter of rain water falling on a meter of the horizontal plane during one second.

Write a program that:

- reads the description of a roof,
- computes the amount of water down in one second from each segment of the roof,
- writes the results.

Input

The input begins with the integer t , the number of test cases. Then t test cases follow.

For each test case the first line of the input contains one integer n ($1 \leq n \leq 40.000$) being the number of segments of the roof. Each of the next n lines describes one segment of the roof and contains four integers x_1, y_1, x_2, y_2 ($0 \leq x_1, y_1, x_2, y_2 \leq 1.000.000, x_1 < x_2, y_1 \neq y_2$) separated by single spaces. Integers x_1, y_1 are respectively the horizontal position and the height of the left end of the segment. Integers x_2, y_2 are respectively the horizontal position and the height of the right end of the segment. The segments don't have common points and there are no horizontal segments. You can also assume that there are at most 25 segments placed above any point on the ground level.

Output

For each test case the output consists of n lines. The i -th line should contain the amount of water (in liters)

down from the i -th segment of the roof in one second

Sample

input	output
1	2
6	4
13 7 15 6	2
3 8 7 7	11
1 7 5 6	0
5 5 9 3	3
6 3 8 2	
9 6 12 8	

Solution:

In this task we are asked to compute the amount of water that falls down from each segment of the roof. First we want to calculate the amount of rain that is falling onto each segment directly from the sky above, and then add the amount of all the water falling onto each segment from the lower end of some other one. Notice that the coordinates are nonnegative integers less or equal then one million; this allows us to iterate through all points on the ground level. While doing so we store the indexes of segments, which are located somewhere above the current X coordinate on the ground, in a list (there will be at most 25 stored segments at any moment during the iteration, as given in the task). So, for each step in this iteration we do the following:

- If a left end of a roof is encountered add it to the list. (we are doing this by moving one pointer in the sorted list of roof segments by X coordinate of left end)
- For each segment in the list whose lower end is equal to the current X coordinate determine the segment under it which will collect the water falling from it. (this can be done by considering Y coordinates of points located on the current segment and each one in the list, both with the current X coordinate, in order to find the closest such point with lower Y coordinate than the point on the current segment)
- If a right end of a roof is encountered remove it from the list. (Pass through all list elements and remove the one with right end equal to the current X coordinate)
- Find the topmost segment from the current list and increase the rain counter for it by one. (similar to the second step, we find the topmost point located on some segment from the list with the current X coordinate)

Now the only thing left is adding the falling water from the segments above. Let's consider each roof as a node and each connection between two segments (two segments are connected if water is falling from one to another) as a directed edge with an end in the one above. We end up with a directed acyclic graph in which for each node we need to compute the sum of all rain in the nodes reachable from it. This can be simply done using **depth first search**, iterate through all nodes and if we don't have wanted information for the current one calculate it by summing all rain collected in the child nodes recursively.

Solution by:

Name: *Dimitrije Dimić*
 School: *School of Computing, Belgrade*
 E-mail: *dimke92@gmail.com*

Problem R1 02: Ambiguous Permutations (code: PERMUT2)

Resource: Adrian Kuegel, used in University of Ulm Local Contest 2005

Time Limit: 10 second

Some programming contest problems are really tricky: not only do they require a different output format from what you might have expected, but also the sample output does not show the difference. For an example, let us look at permutations.

A permutation of the integers 1 to n is an ordering of these integers. So the natural way to represent a permutation is to list the integers in this order. With $n = 5$, a permutation might look like 2, 3, 4, 5, 1.

However, there is another possibility of representing a permutation: You create a list of numbers where the i -th number is the position of the integer i in the permutation. Let us call this second possibility an inverse permutation. The inverse permutation for the sequence above is 5, 1, 2, 3, 4.

An ambiguous permutation is a permutation which cannot be distinguished from its inverse permutation. The permutation 1, 4, 3, 2 for example is ambiguous, because its inverse permutation is the same. To get rid of such annoying sample test cases, you have to write a program which detects if a given permutation is ambiguous or not.

Input

The input contains several test cases.

The first line of each test case contains an integer n ($1 \leq n \leq 100000$). Then a permutation of the integers 1 to n follows in the next line. There is exactly one space character between consecutive integers. You can assume that every integer between 1 and n appears exactly once in the permutation.

The last test case is followed by a zero.

Output

For each test case output whether the permutation is ambiguous or not. Adhere to the format shown in the sample output.

Sample

input	output
4	ambiguous
1 4 3 2	not ambiguous
5	ambiguous
2 3 4 5 1	
1	
1	
0	

Solution:

This was the easiest problem of all bubble cup qualification problems ever. You just need to make inverse array from array A (defined as $\text{inverse}[A[i]] = i$ for each i) and check if $A[i] == \text{inverse}[i]$ for every i ($1 \leq i \leq N$). It's easy to prove that it's enough to check only that $A[A[i]] == i$ for every i ($1 \leq i \leq N, A[i] \leq i$) so you can do this task with only one array and one loop

through the array.

The code for this task is really short, so we can run a shortest code competition ☺. Here is my shortest code in C, which passed all test cases on SPOJ.

```
n,i,b;
main() {
    while(scanf("%d", &n), n) {
        int a[n];
        for(b=i=1; i<=n;) scanf("%d", a+i), b &= a[i] > i | a[a[i]]==i++;
        puts("not ambiguous" + b*4);
    }
    return 0;
}
```

The length of this code is only 135 non-whitespace characters, and I want to thank all the guys who helped me shortening it.

Solution by:

Name: Dušan Zdravković

Organization: School of Computing, Belgrade.

E-mail: duxxud@gmail.com

Problem R1 03: Roll Playing Games (code: RPGAMES)

Resource: ACM East Central North America Regional Programming Contest 2004

Time Limit: 15 second

Phil Kropotnik is a game maker, and one common problem he runs into is determining the set of dice to use in a game. In many current games, non-traditional dice are often required, that is, dice with more or fewer sides than the traditional 6-sided cube. Typically, Phil will pick random values for all but the last die, then try to determine specific values to put on the last die so that certain sums can be rolled with certain probabilities (actually, instead of dealing with probabilities, Phil just deals with the total number of different ways a given sum can be obtained by rolling all the dice). Currently he makes this determination by hand, but needless to say he would love to see this process automated. That is your task.

For example, suppose Phil starts with a 4-sided die with face values 1, 10, 15, and 20 and he wishes to determine how to label a 5-sided die so that there are a) 3 ways to obtain a sum of 2, b) 1 way to obtain a sum of 3, c) 3 ways to obtain 11, d) 4 ways to obtain 16, and e) 1 way to obtain 26. To get these results he should label the faces of his 5-sided die with the values 1, 1, 1, 2, and 6. (For instance, the sum 16 may be obtained as 10 + 6 or as 15 + 1, with three different "1" faces to choose from on the second die, for a total of 4 different ways.) Note that he sometimes only cares about a subset of the sums reachable by rolling all the dices (like in the previous example).

Input

Input will consist of multiple input sets. Each input set will start with a single line containing an integer n indicating the number of dice that are already specified. Each of the next n lines describes one of these dice. Each of these lines will start with an integer f (indicating the number of faces on the die) followed by f integers indicating the value of each face. The last line of each problem instance will have the form

$$r \ m \ v_1 \ c_1 \ v_2 \ c_2 \ v_3 \ c_3 \ \dots \ v_m \ c_m$$

where r is the number of faces required on the unspecified die, m is the number of sums of interest, v_1, \dots, v_m are these sums, and c_1, \dots, c_m are the counts of the desired number of different ways in which to achieve each of the respective sums.

Input values will satisfy the following constraints: $1 \leq n \leq 20$, $3 \leq f \leq 20$, $1 \leq m \leq 10$, and $1 \leq r \leq 6$. Values on the faces of all dice, both the specified ones and the unknown die, will be integers in the range 1 .. 50, and values for the v_i 's and c_i 's are all non-negative and are strictly less than the maximum value of a 32-bit signed integer.

The last input set is followed by a line containing a single 0; it should not be processed.

Output

For each input set, output a single line containing either the phrase "Final die face values are" followed by the r face values in non-descending order, or the phrase "Impossible" if no die can be found meeting the specifications of the problem. If there are multiple dice which will solve the problem, choose the one whose lowest face value is the smallest; if there is still a tie, choose the one whose second-lowest face value is smallest, etc.

Sample

input	output
1 4 1 10 15 20 5 5 2 3 3 1 11 3 16 4 26 1	Final die face values are 1 1 1 2 6 Impossible Final die face values are 3 7 9 9
1 6 1 2 3 4 5 6 6 3 7 6 2 1 13 1	
4 6 1 2 3 4 5 6 4 1 2 2 3	
3 3 7 9 8 1 4 5 9 23 24 30 38 4 4 48 57 51 37 56 31 63 11	
0	

Solution:

Maybe the first impression was that this task is a little bit confusing, but considering the constraints that were given in the task it turns out that you only needed to find a way to see if it is possible to create the last die so it fits the conditions, while keeping in mind the time complexity of your algorithm.

First we create a 2D matrix M in which we will keep the number of ways in which you could get all sums with $n - 1$ dice (there are n given dice). In this matrix the rows would represent the number of dice, and the columns would represent the numbers possible to get from the sum of the dice (e.g. if we had $M[4][2] = 5$ that would mean that we could find 5 ways to achieve the sum 2 with the first four dice). We would calculate fields in this matrix like this:

for every die (with an index j),

and for all its sides (s_1, s_2, \dots, s_m)

$$M[i][j] += M[i - 1][j - v[s_i]], \text{ where } v[s_i] \text{ is be the value of the } i\text{-th side of the die}$$

Then, for the last die, when we need to find out if we can create it, we can try all possible values for its sides using brute force. If we have two arrays (like c and v from the input for the last die) then we would after performing the operation

$$c[j] -= M[n][v[j] - i] \text{ (where } i \text{ is any index for the die side)}$$

check if $c[j]$ is negative. If that is the case then we know that we can't make the last die in this way. In the other case, $c[j]$ is a positive number and we would with a recursion try to find the value for the next side of the last die. After the recursive call we would set back the value of $c[j]$ so we can do another recursion in the next iteration.

Next we would check if the combination is ok (just go through the given conditions and see if everything fits).

After every recursion, if we found a combination of sides for the last die, we still need to check if the c array is empty. If it is, then we have a valid solution, and we are finished. If we went through all combinations and didn't find a valid one then there is no solution.

Solution by:

Name: Uros Joksimovic, Milos Biljanovic, Dejan Pekter

School: School of Computing, Belgrade.

E-mail: uros.joksimovic92@gmail.com, miloshb92@hotmail.com, deximat@gmail.com

Problem R1 04: Manhattan Wire (code: MMAHWIRE)

Resource: Yokohama 2006

Time Limit: 3.0 second

There is a rectangular area containing $n \times m$ cells. Two cells are marked with “2”, and another two with “3”. Some cells are occupied by obstacles. You should connect the two “2”s and also the two “3”s with non-intersecting lines. Lines can run only vertically or horizontally connecting centers of cells without obstacles.

Lines cannot run on a cell with an obstacle. Only one line can run on a cell at most once. Hence, a line cannot intersect with the other line, nor with itself. Under these constraints, the total length of the two lines should be minimized. The length of a line is defined as the number of cell borders it passes. In particular, a line connecting cells sharing their border has length 1.

Fig. 1(a) shows an example setting. Fig. 1(b) shows two lines satisfying the constraints above with minimum total length 18.

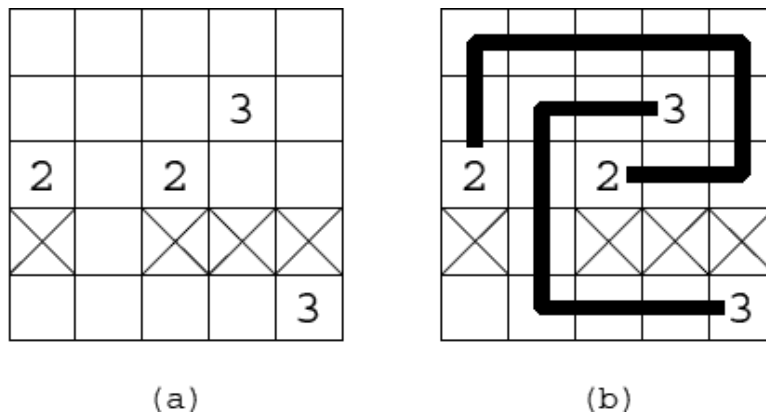


Figure 1: An example of setting and its solution

Input

The input consists of multiple datasets, each in the following format.

```

n m
row1
...
rown
    
```

n is the number of rows which satisfies $2 \leq n \leq 9$. m is the number of columns which satisfies $2 \leq m \leq 9$. Each row_i is a sequence of m digits separated by a space. The digits mean the following.

- 0: Empty
- 1: Occupied by an obstacle
- 2: Marked with “2”
- 3: Marked with “3”

The end of the input is indicated with a line containing two zeros separated by a space.

Output

For each dataset, one line containing the minimum total length of the two lines should be output. If there is

no pair of lines satisfying the requirement, answer “0” instead.

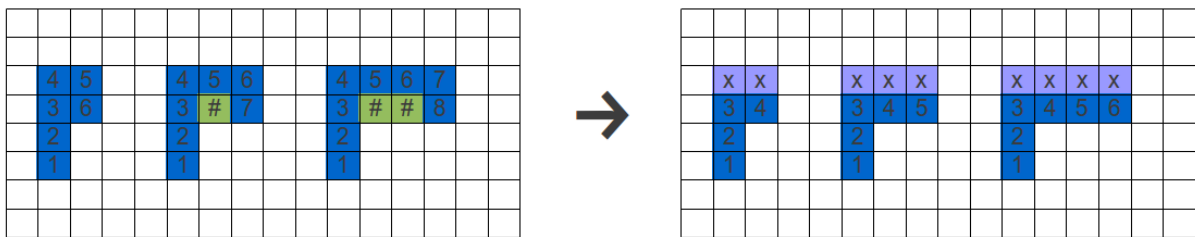
Sample

input	output
5 5	18
0 0 0 0 0	2
0 0 0 3 0	17
2 0 2 0 0	
1 0 1 1 1	
0 0 0 0 3	
2 3	
2 2 0	
0 3 3	
6 5	
2 0 0 0 0	
0 3 0 0 0	
0 0 0 0 0	
1 1 1 0 0	
0 0 0 0 0	
0 0 2 3 0	
0 0	

Solution:

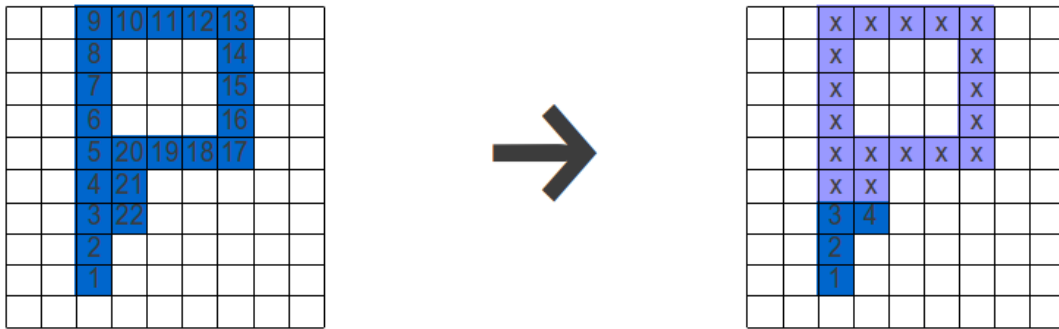
The approach for this problem is very straight-forward: let's try every possible placement of the line connecting “2”s and then look at the shortest available line between “3”s and take the best result. Of course, that solution is too slow and we need some optimizations to prune the search tree to fit the time. Let's look at some situations which will surely lead to a suboptimal solution:

a) U-shape turns (see Figure 1) - if there are no obstacles, it can be easily replaced by a shorter piece of wire. Note: This optimization is the most important one!



If some of the green fields (marked #) are blocked, than it might be good to go on U-shape
 Otherwise – if the green fields are free – you will surely get a better result when you go like in the second picture

b) let's say that we are in field A and there is an adjacent field B that has been visited a long time ago (meaning not in the previous step) - it's bad, because it would have been better to go straight from B to A and avoid the loop – see Figure 2.



c) suppose that we have constructed some part of the line between “2”s. If at this moment there is no path between “3”s (we can check this easily using breadth-first search) we can stop searching, backtrack and try some different way. It turns out that checking for such situations in every step is quite slow, but if we do it on every 300th step, for example, it will considerably speed up our program.

We can also deduce some steps in the very beginning – as long as there's only one possible movement from any “2” or “3” - let's do it and mark it on the grid. It will always help our program.

Obviously, many more optimizations can be applied, but an efficient implementation of the abovementioned ideas make our program easily fit the time limit.

Solution by:

Name: **Bartek Dudek**

Organization: XIV LO Wrocław

E-mail: bardek.dudek@gmail.com

Problem R1 05: Spheres (code: KULE)

Time Limit: 2.0 second

John has a certain number of spheres. Almost all of them have identical weight apart from one. There are a lot of them and John cannot say which one differs from the other ones by himself. You can help him to determine which sphere it is by using the pair of scales.

Input

In the first line of the input there is one integer n ($3 \leq n \leq 100.000$) that stands for the number of spheres which John has. The spheres are numbered from **1** to n .

You can give John two types of orders (just print them to standard output):

- **WEIGHT m a_1 a_2 ... a_m b_1 b_2 ... b_m**

Weighting spheres. All numbers should be separated with a space and they stand for: m - number of spheres that should be put on one of the scales (there should be the same amount of spheres on both of the scales), a_1 a_2 ... a_m - identifiers of spheres that you want John to put on the left scale and b_1 b_2 ... b_m - identifiers of spheres that you want John to put on the right scale.

After conducting the weighting John will tell you about the outcome (which you will be able to read from the standard input). Possible answers are LEFT - spheres on the left scale are heavier, RIGHT - spheres on the right answer are heavier, EQUAL - spheres on both scales have equal weight.

After conducting the weighting John is ready right away to execute the next order.

However, you should remember that if the weighting's number is too high John can become quite bored...

- **ANSWER k**

Answering. This order is to give information that k is the identifier of the searched sphere; however if the sphere we are looking for is lighter than the other ones you should precede that with a '-' sign.

John no longer needs you after that command (your program should end).

Output

Output the month number the accountant-robot will rust in. Months are numerated 1 to p .

Sample

input / output	
John:	3
You:	WEIGHT 1 1 2
John:	LEFT
You:	WEIGHT 1 1 3
John:	EQUAL
You:	ANSWER -2

Remark:

Program should clear the output buffer after printing each line. It can be done using `fflush(stdout)` command or you can set the proper type of buffering at the beginning of the execution - `setlinebuf(stdout)`.

Solution:

The problem is one of the variants of the Coin-Weighting problem: Given n coins, one of which is counterfeit, and a pair of scales (two-pan balance) without weights, what is the minimum number of weightings needed to find the counterfeit coin? There are variants in which we know/don't know if the counterfeit is lighter/heavier, in which we are required to only indicate counterfeit/determine its weight, in which we have/don't have some additional genuine coins etc. The following theorem gives us the required minimum number of weightings for the corresponding variant of the problem and its constructive proof can easily be implemented to obtain the required algorithm:

Theorem 1: Given n coins, numbered from 1 through n , $n - 1$ of which are genuine and with exactly one counterfeit among them, the minimum number of weightings needed to determine index of the counterfeit **and** if it's lighter or heavier than the others is $\lceil \log_3(2n + 3) \rceil$.

Proof: Let k be the optimal number of weightings. Since we have n candidates for counterfeit coin, and in each case the counterfeit can be lighter or heavier than the genuine coin, **there are $2n$ possible outcomes in total**. On the other hand, let us mark the result of each weighting with a number from set $S = \{-1, 0, 1\}$; -1 denotes that the left pan of the scales was heavier, 0 denotes that the scales were balanced and 1 denotes that the right pan was heavier. The arrangement of the coins on the scale pans on the i -th weighting depends only on the results of previous weightings and previous rearrangements, which were themselves induced by the previous weighting results. Therefore, after the arrangement of the coins on the scale pads for the first weighting is fixed, the subsequent rearrangements depend only on the results of previous weightings (with if-then conditions for new rearrangements) and the final outcome (counterfeit coin) depends only on the weighting results. It follows that the outcome of the k weightings **can be uniquely described as a sequence (a_1, a_2, \dots, a_k) of individual weighting results $(a_i \in S)$** .

We just showed that k weightings can distinguish between at most 3^k different outcomes; since we must be able to recognize $2n$ outcomes, it follows $2n \leq 3^k$. Since 3^k is odd, we can rewrite the inequality as $2n \leq 3^k - 1$ (instead of the parity argument, we could also notice that weighting sequence $(0, 0, \dots, 0)$ cannot tell whether the counterfeit is lighter or heavier – because of that only $3^k - 1$ sequences are valid). However, this inequality is a bit weak; we will improve it.

Let m be the number of coins which will be put in the first weighing on each pan. If in the first weighing the pans are balanced, the counterfeit coin is one of $n - 2m$ coins not participating in this weighting. The remaining $k - 1$ weightings must be enough to identify the counterfeit among $n - 2m$ coins and to find out if it is heavier or lighter. As before, it follows $2(n - 2m) \leq 3^{k-1} - 1$ (because of the parity argument). However, if in the first weighing the pans aren't balanced, the counterfeit is among $2m$ coins which participated in this weighting. Unlike the previous case, this time it is enough to find **only an index** of a counterfeit coin in the following $k - 1$ weightings; its weight can be determined from the first weighting. Therefore, we are only interested in $2m$ outcomes (not $2 \cdot 2m$) and it follows $2m \leq 3^{k-1} - 1$ (parity argument again). Adding $2(n - 2m) \leq 3^{k-1} - 1$ and doubled $2m \leq 3^{k-1} - 1$ results in $2n \leq 3^k - 3$, which is (using the fact that k is an integer) equivalent to

$$k \geq \lceil \log_3(2n + 3) \rceil.$$

We will now prove that this bound is achievable by explicitly constructing the weightings. It suffices to construct the required k weightings for all $n \in \left(\frac{3^{k-1}-3}{2}, \frac{3^k-3}{2}\right]$ (for $n \leq \frac{3^{k-1}-3}{2}$, less than k weightings are

needed). It is assumed that $k \geq 3$ since $k < 3$ gives trivial cases with $n \leq 3$ coins.

First, let us recall of a method for determining a counterfeit coin among 3^m coins, when **it is known** whether the counterfeit is lighter/heavier than others, in m weightings: we divide coins into 3 equal groups of size 3^{m-1} and put any two groups on different pans. Since we know if the counterfeit is lighter/heavier, result of weighting will identify the group with counterfeit and we will apply the same method recursively. After m weightings, the counterfeit will be identified. This also works for any $3^{m-1} < n \leq 3^m$ number of coins (i.e. m weightings suffice): We divide coins into 3 as equal as possible groups: (x, x, x) or $(x + 1, x, x)$ or $(x + 1, x + 1, x)$ (at least 2 of them will be equal) and weight the equal ones. After that, we will narrow our search to a group of a size at most $\lfloor \frac{n}{3} \rfloor + 1$ (if $3 \nmid n$) or $\frac{n}{3}$, if $3 \mid n$. In either case, the resulting group will have less than or equal to 3^{m-1} coins, and we can apply the method recursively again. We will call this method the **simple method**.

Back to the optimal weighting construction. First, let us assume that $n = \frac{3^k - 3}{2}$ (which is an edge case, but is the simplest for construction). We divide n coins into 3 equal groups A, B, C with $\frac{3^{k-1} - 1}{2} = 1 + 3 + \dots + 3^{k-2}$ coins each. Now, we divide each group into $k - 1$ subgroups $\{A_0, A_1, \dots, A_{k-2}\}, \{B_0, B_1, \dots, B_{k-2}\}, \{C_0, C_1, \dots, C_{k-2}\}$; subgroups A_i, B_i, C_i have 3^i coins each. In the first weighting, we place group A on the left pan and group B on the right pan. Regardless of the result, in the second weighting, we remove subgroup A_{k-2} from the left pan, move subgroup B_{k-2} from the right pan to the left and put subgroup C_{k-2} to the right pan.

If the result of the second weighting differs from the first one ($a_1 \neq a_2$), then the counterfeit coin belongs to some of the subgroups $A_{k-2}, B_{k-2}, C_{k-2}$; moreover, **from these 2 weightings we can precisely deduce which of the 3 subgroups contains the counterfeit and if the counterfeit is lighter/heavier than the others**. To see this, it suffices to consider all 6 possibilities (A_{k-2} vs. B_{k-2} vs. C_{k-2} and lighter vs. heavier) – they all give different weighting result sequence (a_1, a_2) . After these 2 weightings, we have a group of 3^{k-2} coins with counterfeit coin of known weight – we can solve this using the *simple method* in $k - 2$ weightings, which gives k weightings in total.

However, if the result of the second weighting remains the same ($a_1 = a_2$), it follows that all the coins from the subgroups $A_{k-2}, B_{k-2}, C_{k-2}$ are genuine. In that case, we repeat the same process as in the second weighting, only this time with subgroups $A_{k-3}, B_{k-3}, C_{k-3}$. If $a_3 \neq a_2$, we use the *simple method* on 3^{k-3} coins as in the previous paragraph, otherwise we “rotate” subgroups $A_{k-4}, B_{k-4}, C_{k-4}$ etc.

This algorithm always uses k weightings for identifying counterfeit and its weight: if counterfeit belongs to one of the subgroups A_i, B_i, C_i , we will use one starting weighting, $k - i - 1$ subgroup “rotations” and i *simple method’s* weightings – k weightings in total, as required.

What if $\frac{3^{k-1} - 3}{2} < n \leq \frac{3^k - 3}{2}$? Again, we divide n coins into 3 groups A, B, C : if $n = 3x$ then $|A| = |B| = |C| = x$; if $n = 3x + 1$ then $|A| = |B| = x + 1, |C| = x - 1$; if $n = 3x + 2$ then $|A| = |B| = x + 1, |C| = x$. In either case, using $\frac{3^{k-1} - 3}{2} < n \leq \frac{3^k - 3}{2}$, we have

$$1 + 3 + \dots + 3^{k-3} = \frac{3^{k-2} - 1}{2} < |A| = |B| \leq \frac{3^{k-1} - 1}{2} = 1 + 3 + \dots + 3^{k-2}.$$

Therefore, we can write $|A| = |B| = 1 + 3 + \dots + 3^{k-3} + X$, where $1 \leq X \leq 3^{k-2}$. Now, just like in an edge case, we divide each group A, B, C into $k - 1$ subgroups $\{A_0, A_1, \dots, A_{k-2}\}, \{B_0, B_1, \dots, B_{k-2}\}, \{C_0, C_1, \dots, C_{k-2}\}$; subgroups A_i, B_i, C_i have 3^i coins each, for $0 \leq i \leq k - 3$, and subgroups $A_{k-2}, B_{k-2}, C_{k-2}$ have X coins each. Since $X \leq 3^{k-2}$, a *simple method* can handle this subgroup with at most $k - 2$ weightings. Now, with the edge case algorithm, we are able to determine the counterfeit coin in k weightings.

However, note that the group C might have fewer coins than the groups A, B . This is not a problem – for $k > 3$ we can leave subgroups C_0 and C_1 one coin short ($|A| - |C| \leq 2$); if, during an algorithm, we reach those subgroups, it means that all coins in subgroups A_i, B_i, C_i for $i > 1$ are genuine, and we can use them to fill those 2 subgroups. For $k = 3$, in particular for $n = 7$ and $n = 10$, group division should be $(2, 2, 2)$ and $(3, 3, 3)$, respectively (the only cases with $|C| < X$). This completes all steps of the algorithm.

With this, Theorem 1 is proved and the optimal weighting sequence is explicitly constructed. ■

Implementation:

Direct simulation of a mentioned algorithm with subgroup “rotations” and the *simple method*. Since $k = O(\log n)$, we can traverse all the coins during each weighting with a total complexity of $O(n \log n)$.

Note:

The online judge system requires a sharp bond on weighting number for this task. If contestant’s code exceeds the optimal number of weightings, John will not report the outcome of the weighting and TLE (time limit exceeded) will occur.

Other variants of the Coin-Weighting problem and some different (and generalized) weighting constructions can be found in [1].

References:

- [1] Marcel Kolodziejczyk, *Two-pan balance and generalized counterfeit coin problem*
- [2] <http://www.cut-the-knot.org/blue/OddCoinProblems.shtml>
- [3] <http://www.mathplayground.com/coinweighing.html>

Solution by:

Name: **Nikola Milosavljević**

School: Faculty of Mathematics, University of Niš

E-mail: nikola5000@gmail.com

Problem R1 06: Sightseeing (code: GCPC11H)

Resource: German Collegiate Programming Contest 2011 (Author: Moritz Kobitzsch)

Time Limit: 1.0 second

As a computer science student you are of course very outdoorsie, so you decided to go hiking. For your vacation this year, you located an island full of nice places to visit. You already identified a number of very promising tracks, but are still left with some problems. The number of choices is so overwhelming, that you had to select only a "small" subset of at most 105 sights.

And if that is not enough, you are very picky about the order in which you want to visit the sights. So you have already decided on an order in which you want to visit the preselected tracks. The problem you are left with is to decide in which direction to travel along each single track, and whether you may have to reduce your choice of tracks even further. After identifying the travel time between the endpoints of different tracks, you decide to write a program to figure out if you can make all your trips within the time you have planned for your vacation. Since you also do not want to waste any precious time, you only care about an optimal solution to your problem. Furthermore, the tracks can get pretty challenging. That's why you do not want to hike along a track more than once.

Input

The first line of the input gives the number of test cases C ($0 < C \leq 100$). The first line of each such test case holds two integers N, T the number of tracks of the current hiker ($1 \leq N \leq 105$) and the maximal time spent hiking throughout the vacation ($0 \leq T \leq 106$). Each of the following N lines holds five integers cp, cbb, cbe, ceb and cee that describe a track (in order of importance). cp gives the length of the track in minutes. cxy gives the travel time of the official begin or end of a track to the beginning or end of the next most important track, where x and y are either b or e . All values given are non-negative integers not greater than 106. Since you have to get back to your car, the list is circular. Furthermore, we will ignore the time it takes you to get to the start of your trip with your car.

Output

For each test case print one line. The output should contain a list of either F or B for every track (in order) indicating whether you have to hike the track in forward direction or backward direction. If you cannot make the full trip within the planned time T , you should print IMPOSSIBLE to indicate that these trips are just too much hiking. You can assume that the optimal solution is always unique.

Sample

input	output
3	FF
2 100	BB
4 7 8 2 3	IMPOSSIBLE
1 4 6 1 2	
2 20	
4 2 3 7 8	
1 1 2 4 6	
3 5	
1 2 2 2 1	
1 1 2 2 2	
1 2 2 1 2	

Solution:

At the first glance, this appears to be a problem requiring finding the shortest path. The issue is to find the correct vertices. Obviously, keeping the index of the road is not enough, as there are two ends. Plus, you need to keep track of whether you have traveled across that road or not.

Thus, define a new graph $G'(V', E')$, where each vertex is described by a triple $(idx, isBegin, isHiked)$ which uniquely identifies your current position:

- idx is the index of the track you are currently in.
- $isBegin$ is a boolean value which describes whether you are at the beginning or the end of the track.
- $isHiked$ is a boolean value describing whether you have hiked the corresponding track or not.

The set of edges can be found quite easily. This is left for the readers as an exercise.

A sightseeing tour now becomes a path from the vertex $(0, isBegin, false)$ to itself. ($isBegin$ can be true or false).

First solution: Dijkstra's algorithm

This is the direct way to solve the problem, and the implementation is quite straightforward. The time complexity is $O(N \log N)$, as the number of edges is proportional to N . It should be enough to pass all of the test cases.

Second solution: Dynamic programming

For readers who don't like 'slow' solutions, there are more to explore.

It's not difficult to recognize that the state $(idx, true, *)$ depends solely on $(idx, false, *)$, and similarly, the state $(idx, false, *)$ is dependent on $(idx - 1, true, *)$.

The DP solution shares the same idea as the first solution. The recursion is not difficult and is performed by calculating in the following order:

$$(0, false, *) \rightarrow (0, true, *) \rightarrow (1, false, *) \rightarrow \dots$$

As any state is visited exactly once, the time complexity of the algorithm should be $O(N)$.

Solution by:

Name: **Linh Nguyen**

School: Vietnam National University

E-mail: ll931110@yahoo.com

Problem R1 07: Segment Flip (code: SFLIP)

Resource: Proposed by venkateshb

Time Limit: 1.0 second

You are given N number a_1, a_2, \dots, a_N . In a segment flip, you can pick a contiguous segment $a_i, a_{(i+1)}, \dots, a_j$ of these numbers, where $i \leq j$ and negate all the numbers in this segment.

You are permitted at most K segment flip operations overall. Also, no 2 segments that you pick can overlap. That is, if you flip a_i, \dots, a_j and a_k, \dots, a_l then either $j < k$ or $l < i$.

Your aim is to maximize the sum of all the numbers in the resulting sequence by applying appropriate segment flip operations meeting these constraints.

For instance, suppose the sequence is $-5, 2, -3$ and you are allowed a single segment flip. The best sum you can achieve is 6, by flipping all 3 numbers as a single segment to $5, -2, 3$.

Input

The first line contains 2 integers N and K . The next line contains N integers, the initial values of a_1, a_2, \dots, a_N .

Output

A single integer denoting the maximum possible sum of the final array.

Constraints

- $0 \leq K \leq N$
- $-10000 \leq a_i \leq 10000$
- $1 \leq N \leq 100000$

Sample

input	output
3 1 -5 2 -3	6

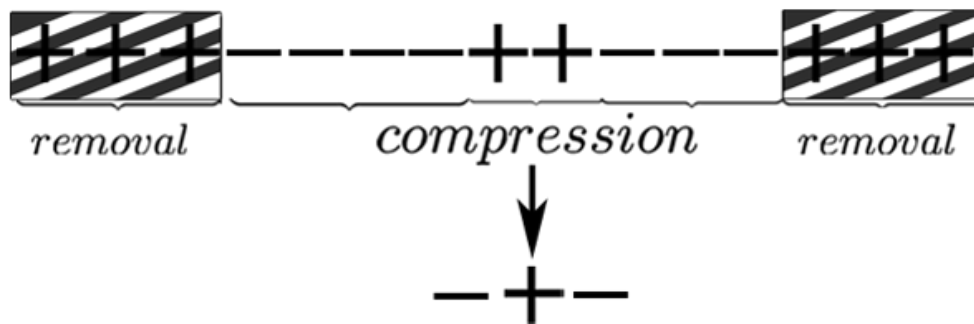
Solution:

This interesting problem was definitely one of the hardest ones on Round 1, and the fact it had the lowest amount of correct solutions out of all the problems of the round confirms that. The main idea is not very hard to conceive but implementing it correctly can get a bit frustrating.

The problem statement sticks to formal mathematics and hence doesn't require any particular decoding: we are asked to determine the maximal sum of a sequence of integers after performing no more than K "segment flip" operations on disjoint segments (where one operation negates an entire range of the sequence). This problem is not extensively covered in literature, however there exists one paper which names this kind of problem the **Maximal-Scoring Segment K-Set problem**. It is a problem that arises commonly in bioinformatics, most notably DNA and RNA analysis.

Before implementing the main algorithm, it's wise to first transform the given sequence into a more special case, which is equivalent to the initial sequence but will make our calculations simpler. First of all, it is fairly

easy to notice that if there are positive integers on the ends of the sequence, they certainly won't need to be flipped, hence we can immediately exclude them from the range we're observing and add them to the final solution. That way we're left with negatives on both ends of the observed range, and this is a property we can constantly maintain as we build our set of segments; it will later be clarified how. The next transformation we can immediately apply is to "compress" successive runs of positive/negative integers into a single integer with the value of the sum of all integers in that run. *Why can we do this?* Let's say a negative integer is contained in the final set of flipped segments; it is obvious that including all the negatives adjacent to it can only benefit to the solution. Analogously, the only reason why we would want to flip a positive integer is to join together two segments of negatives, and we can't do this if we don't flip all the other positives in that run as well. **In the end we are left with an alternating sequence of negative and positive integers.** For visualization, refer to Fig. 1.



Let's say that after the given transformations are applied, there are exactly M negative integers in the sequence. *What is the maximal sum we can obtain with M segment flips?* It is obviously the sum obtained when we flip all the negatives separately. If $M \leq K$, then this is also the optimal solution to the problem. Otherwise, the solution is derived from the M -segment solution using a **theorem** that is formally stated in [1], albeit with its proof omitted. It states that a solution for $X - 1$ segment flips can be obtained from the solution for X segment flips by either **merging two segment flips** or **excluding one**. Using this we can find the solution for K segment flips in $M - K$ iterations.

It is easy to conclude that the segment to be excluded/merged through in any iteration must be the segment with the **minimal absolute value** (its removal/flipping would pose minimal "damage" to the sum obtained with M flips), so we are required to store segments we are observing in a structure which will efficiently provide this segment; one of the possible data structures we can use for this is a **min-heap**.

Now let's discuss the algorithm itself; in every iteration, the algorithm extracts the segment with the currently minimal absolute value from the heap. In the case that this segment is on **one of the ends** of the currently observed range, we know that we have run into a negative segment and that we will never have to re-include it again; hence, it is optimal to **remove** both it and the positive run adjacent to it from consideration, so we are left with a negative segment at both ends again. If the extracted segment is not at either end, the action taken depends on the sign of the segment:

- If the segment is positive, we flip it to merge the two negative segments adjacent to it;
- If the segment is negative, we exclude it for now; it is possible to re-include it, but only together with both positive segments adjacent to it.

If we look at this a little closer, we can conclude that same actions can be performed regardless of the sign: **subtract** the segment's absolute value from the optimal solution obtained in the previous

iteration, **remove** the segment and its two adjacent segments from the heap and **insert** a segment which is obtained by merging those three. With a little optimization to assign an ID to each segment, and to constantly store the IDs of segments directly to the left and right of them, the updates described above can be done quite efficiently. This concludes the algorithm description, and it turns out to be fairly short and easy to code (requiring only ~50 lines of code for the main algorithm).

The time complexity of each iteration of the algorithm is $O(\log n)$ for updating the heap, hence the overall asymptotic time complexity of this solution is $O(n \log n)$. The memory complexity of the solution is $O(n)$.

For a more in-depth look at the Maximal-Scoring Segment K-Set problem, its generalizations, as well as formal statements of the theorem and algorithm mentioned here and the algorithm's application in analyzing biomolecules, refer to [1].

References:

[1] Miklós Csűrös: *Algorithms for Finding Maximal-Scoring Segment Sets (extended abstract)*, *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (2004)

Solution by:

Name: **Petar Veličković**

School: Matematička Gimnazija

E-mail: petrov.velickovic@gmail.com

Problem R1 08: K12 - Building Construction (code: KOPC12A)

Time Limit: 1.0 second

Given N buildings of height $h_1, h_2, h_3 \dots h_n$, the objective is to make every building has equal height. This can be done by removing bricks from a building or adding some bricks to a building. Removing a brick or adding a brick is done at certain cost which will be given along with the heights of the buildings. Find the minimal cost at which you can make the buildings look beautiful by re-constructing the buildings such that the N buildings satisfy

$$h_1 = h_2 = h_3 = \dots = h_n = k \text{ (} k \text{ can be any number).}$$

For convenience, all buildings are considered to be vertical piles of bricks, which are of same dimensions.

Input

The first line of input contains an integer T which denotes number of test cases. This will be followed by $3 * T$ lines, 3 lines per test case. The first line of each test case contains an integer n and the second line contains n integers which denotes the heights of the buildings $[h_1, h_2, h_3 \dots h_n]$ and the third line contains n integers $[c_1, c_2, c_3 \dots c_n]$ which denotes the cost of adding or removing one unit of brick from the corresponding building.

$$T \leq 15; n \leq 10000; 0 \leq H_i \leq 10000; 0 \leq C_i \leq 10000;$$

Output

The output must contain T lines each line corresponding to a testcase.

Sample

input	output
1 3 1 2 3 10 100 1000	120

Solution:

First, let *low* be the height of the shortest building, and let *high* be the height of the tallest building. It isn't hard to notice that the solution k will satisfy the inequality $low \leq k \leq high$. This is easy to prove - if there is a solution k smaller than *low*, then we need to remove bricks from every single building. The solution is guaranteed not to be optimal because removing a brick from each building will cost less. The proof of the latter, $k \leq high$, is analogous.

Since the limitations for heights for this problem are quite low, the following algorithm with time complexity $O(N + high)$ solves the problem.

Let $f(i, k)$ be the cost of changing the height of the i -th building to k . Clearly, $f(i, k) = |h_i - k|c_i$. Let $F(k)$ be equal to $\sum_{i=1}^N f(i, k)$. Clearly, the total cost to change the heights of all buildings to k will be $F(k)$. Now we define $e(i, k) = f(i, k) - f(i, k - 1)$. Similarly, we define $E(k) = \sum_{i=1}^N e(i, k)$. Notice that this is also equal to $F(k) - F(k - 1)$. The reason why we are doing this is because we want to be able to

quickly calculate $F(x)$ if we know $F(x - 1)$ and $E(x)$, for some x . We can calculate $F(0)$ in $O(N)$, but, how can we calculate $E(x)$ for some x ? Since $e(i, k) = -c_i$ for $k \leq h_i$ and c_i otherwise (in other words, it decreases by c_i every time we increase the target height if it's smaller than the original height, and increases otherwise), we can calculate $E(x)$ from its definition as $\sum_{i=1}^N e(i, x)$. This leads to an $O(N \cdot high)$ algorithm, which is still too slow.

So, let's revisit the above procedure. Define $d(i, k)$ as $e(i, k) - e(i, k - 1)$. Notice that $d(i, k) = 2c_i$ for $k = h_i + 1$, and 0 otherwise. Now define, yes - you guessed it: $D(k) = \sum_{i=1}^N d(i, k)$. We observe that $D(k) = E(k) - E(k - 1)$. Now, if we could quickly calculate $D(x)$ for every x , then we could calculate $E(x)$ in constant time if we knew $E(x - 1)$ and $D(x)$. Also, we can calculate $E(1)$ in linear time by simply iterating through all buildings.

Now, back to computing the function $D(x)$. Initialize the array D to all zeros. Iterating through the buildings, increase $D[h_i + 1]$ by $2c_i$ for every building encountered. From all this information, one can compute $F(x)$ for all x and then simply choose the value of x where the function has a minimum.

Complexity: $O(N + high)$ time and $O(N + high)$ space.

Solution by:

Name: **Ivan Stošić**

School: *Gymnasium "Svetozar Marković", Niš*

E-mail: *ivan100sic@gmail.com*

Problem R1 09: Its a Murder! (code: DCEPC206)

Time Limit: 0.5 second

Once detective Saikat was solving a murder case. While going to the crime scene he took the stairs and saw that a number is written on every stair. He found it suspicious and decides to remember all the numbers that he has seen till now. While remembering the numbers he found that he can find some pattern in those numbers. So he decides that for each number on the stairs he will note down the sum of all the numbers previously seen on the stairs which are smaller than the present number. Calculate the sum of all the numbers written on his notes diary.

Input

First line gives T , number of test cases. $2T$ lines follow. First line gives you the number of stairs N . Next line gives you N numbers written on the stairs.

$T \leq 10$; $1 \leq N \leq 10^5$; All numbers will be between 0 and 10^6 .

Output

For each test case output one line giving the final sum for each test case.

Sample

input	output
1 5 1 5 3 6 4	15

Solution:

All you needed to do to solve this task is to sum up for every given number all past numbers which are smaller than that number. Sounds simple enough.

So a naive solution would be that you have an array of numbers so that every given number is added in $O(1)$ time to that array, and after that the required sum can be computed in $O(n)$ time. This algorithm is slow given the constraints (there can be 10^5 numbers per test case) so we need to think of another way to implement these two operations.

This can be solved by **cumulative tables**, where cumulative stands for “how much so far” and that is what we need. The time complexity for add and sum operations are $O(\log n)$, which is fast enough to pass.

The whole idea behind the cumulative tables is if we can represent a number in a binary notation (e.g . $14 = 2^3 + 2^2 + 2^1$ or 1110 in binary) , then we can also represent any sum of numbers as a sum of specific subtotals.

For an example if we need the sum of numbers that are less than 14, then we would have a dynamic array where we would keep our subtotals, so for that sum we would only need to look at a few subtotals to get the result, which can be done in $O(\log n)$. Why is the complexity logarithmic? The binary presentation of 14 is 1110, so to get the subtotals, we would delete the rightmost one in binary and with that new number we would use it as index to get the subtotal from the dynamic array. We would repeat the process until we

delete all ones (get zero as index) and along the way sum up all the subtotals. There are about $\log n$ operations to execute because there can be at most $\log n$ digits in the binary representation of n .

For adding an element to our dynamic array, so the subtotals that we have are correct, we will do the opposite of what we were doing when we were getting the required sum. We only need to update subtotals that depend on the number we are adding so to speak, so there are at most $\log n$ subtotals that need to be updated, so the time complexity for adding is $O(\log n)$ too.

Solution by:

Name: Uros Joksimovic, Milos Biljanovic, Dejan Pekter

School: Racunarski fakultet (RAF)

E-mail: uros.joksimovic92@gmail.com, miloshb92@hotmail.com, deximat@gmail.com

Problem R1 10: Words on graphs (code: AMBIG)

Time Limit: 1.0 second

Input

The input is a directed (multi)graph. The first line gives the number of edges M and the number of nodes N (≥ 2). Then each edge is described by a line of the form "FROM TO LABEL". Nodes (FROM, TO) are numbers in the range $0..N - 1$ and labels are also numbers. All numbers in the input are nonnegative integers < 2000 .

Output

Print "YES" if there are two distinct walks with the same labelling from node 0 to node 1, otherwise print "NO".

Sample

input	output
4 4 0 2 0 0 3 0 2 1 1 3 1 2	NO
10 9 0 2 0 2 1 0 2 3 0 3 4 0 4 2 0 2 5 0 5 6 0 6 7 0 7 8 0 8 2 0	YES

Solution:

The problem statement is very brief, but there is a lot of information hidden among the test cases. Let's first dissect what we should do. We are given a **directed labeled multigraph** and we need to find out if there are at least two distinct paths from node 0 to node 1, such that we can only traverse edges with the same labeling. The previous sentence needs some explanations: when you traverse edges finding requested paths you need to have the same sequence of edge labelling, e.g. if you have a path that contains edges with labeling 1-5-10-10, the other path also must have 1-5-10-10 labeling of its edges. The distinction criteria is based on the visited nodes, so in the above example if we did have two paths with same labeling but at least one node differs, we do have two distinct paths and the answer is YES.

So now that we understand what we should do, let's talk about possible solutions.

Let's consider a naive solution first. Let us find all the paths, select pairs of paths that have the same labeling and check if there is a difference in visited nodes. As you may guess this is way too slow, it has exponential time of execution and for 2000 nodes/edges it wouldn't produce a solution even if we had 1000 years of spare time.

We need to abstract things a bit so that we can prune paths that we don't need.

The question is: Do we even need whole paths?

The answer is: No, well after thinking a bit on this problem you would probably come to the same kind of abstraction. The key point here is to search for distinct paths in parallel. Let's consider that we have two paths, but we don't really need all the information: what nodes path contain, what edges path contains, and so on... We only need to know what are the nodes of path at some point, and if the paths contained different nodes at some point. Let's define state in two paths after traversing N edges as:

- Current node on path 1
- Current node on path 2
- Did we have a distinction (any state so far has distinct property or if current nodes differ in this state)

This state can be described as three-parameter function $f(\text{node}, \text{node}, \text{bool})$, where we have $n \cdot n \cdot 2$ states, which is about 8 million at most.

So what you need to do is perform a **breadth-first search** (or an iterative depth-first search, because recursive DFS will fail due to stack overflow when it tries to call itself several million times) which is going to traverse graph in parallel. So the BFS should look like this:

```
parallel_bfs(0, 0, false)
  queue.add(state(0, 0, false));
  state = getStateFromQueue();
  for every pair of state.node1's and state.node2's neighbor whose edges have same labeling
    if ( state (node1, node2, distinct || (node1 != node2) is not visited)
      mark state as visited
      put that state in the queue
```

You should keep all the states in a boolean matrix $state[node1][node2][distinct]$. In the end the solution to this problem will be contained in $state[1][1][true]$, if it is true it means that we made it to a state where we finished both of our paths in node 1 and we did at some point have distinction in nodes (it doesn't matter where).

This solution has quadratic complexity and should pass the time constraint, but if it is not implemented well you will need some optimisations, and here is a nice one.

Let's think if there is way not to do all the computation, but to make a conclusion a bit earlier. If we could transform our graph before all this processing to a graph that always contains the path from every node to node 1 we could conclude that we found distinct path if we are in state $state[anyNode1][anyNode2][true]$, where $anyNode1$ and $anyNode2$ are equal, otherwise there is no such path. So there is no need to go to node 1 if we know that there is a path from every node to node 1. When we come to the state with equivalent current nodes and distinct paths we can connect that path with node 1 in any way we want to.

How to do this transformation?

Pretty easy, we only need to delete all the nodes from which we can't reach node 1. That can be done by reversing all the edges of the graph, and running regular BFS on that graph. After the BFS is done the visited nodes are the only ones we are interested in. Our new graph, with only the selected nodes and edges reversed back, has the property described above. Now we can run our parallel BFS algorithm and break out from it as soon as we hit the state with the same nodes and the *distinct* flag on. This optimisation speeds up the algorithm a lot, but its complexity is a bit hard to prove.

Solution by:

Name: Uros Joksimovic, Milos Biljanovic, Dejan Pekter

School: Racunarski fakultet (RAF)

E-mail: uros.joksimovic92@gmail.com, miloshb92@hotmail.com, deximat@gmail.com

Problem R2 01: Zig-Zag Permutation (code: ZZPERM)

Time Limit: 1.0 second

In the following we will deal with nonempty words consists only of lower case letters 'a','b',..., 'j' and we will use the natural 'a' < 'b' < ... < 'j' ordering. Your task is to write a program that generates almost all zig-zag words (zig-zag permutations) from a given collection of letters. We say that a word $W=W(1)W(2)...W(n)$ is zig-zag iff $n = 1$ or $W(i) > W(i + 1)$ and $W(j) < W(j + 1)$ for all odd $0 < i < n$ and for all even $0 < j < n$ or $W(i) > W(i + 1)$ and $W(j) < W(j + 1)$ for all even $0 < i < n$ and for all odd $0 < j < n$. For example: "aabcc" is not zig-zag, "acacb" is zig-zag, "cac" is zig-zag, "abababc" is not zig-zag. If you imagine all possible zig-zag permutations of a word in increasing lexicographic order, you can assign a serial number (rank) to each one. For example: the word "aabcc" generates the sequence: 1 <-> "acacb", 2 <-> "acbca", 3 <-> "bacac", 4 <-> "bcaca", 5 <-> "cabac", 6 <-> "cacab".

Input

The input file consists several test cases. Each case contains a word (W) not longer than 64 letters and one positive number (D). The letters of each word are in increasing order. Input terminated by EOF.

Output

For each case in the input file, the output file must contain all of the zig-zag permutations of W whose zig-zag serial is divisible by D, in increasing lexicographic order - one word per line. In the next line you have to print the total number of zig-zag permutations of W. There is no case that produces more than 365 lines of output. Print an empty line after each case.

Sample

input	output
j 1	j
abc 2	1
aaabc 1	bac
aaabb 2	cab
aaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbccdd	4
123456	abaca
	acaba
	2
	1
	babacbcabacabadababababababababababab
	213216

Solution:

The first thing you should notice about the problem is that there are at most only $|\{ 'a', 'b', 'c', \dots, 'j' \}| = 10$ distinct letters in the word W. After that, let's first focus our efforts on solving the first part of the problem: counting the number of zig-zag permutations of W.

The solution will be an application of **dynamic programming**. Let's say we constructed the first K characters

of a specific zig-zag permutation and we need to construct $N - K$ more of them. The only things we need to know are: what's the last (K -th) character, whether the next character should be greater or lesser than the K -th, and how many characters of which kind are remaining. Given that data, we can count the number of ways we can extend the word to form a complete zig-zag permutation.

For example, let's say we constructed the prefix and the state is now

last_char, next_char_greater?, characters_remaining) = ('c', greater, "aabbcdde")

We can obviously transition from that state to ('d', lesser, "aabbcdde") and ('e', lesser, "aabbcdde"). Associating the number of zig-zag permutations with the state enables us to calculate the first required number. The only thing left to do is to pick the first character and the relationship between the first two characters and sum up the numbers for each choice.

From a complexity point of view, the solution isn't perfect. If the original word W had 6 occurrences of each of the 10 letters, then the number of states would be $10 \cdot 2 \cdot 7^{10}$, because there are 10 possibilities for the last character, 2 possible directions and every character can occur from 0 to 6 times, yielding a total of 7^{10} states. The said number is about $5 \cdot 10^9$ and is generally too big to handle within the time limit. Luckily, no such test cases exist and the said algorithm is good enough to run within the allowed time.

As for the second part of the problem (finding the D -th zig-zag permutation), it's amazingly simple to implement after we know how to solve the first part via dynamic programming. You can easily find the first character of the string by counting how many permutations there are with a certain first character and stopping when the cumulative number exceeds D . A similar algorithm can be used to find the second character, etc. Such an algorithm is sufficient to solve the problem.

Solution by:

Name: **Goran Žužić**

School: FER, Zagreb

E-mail: zuza777@gmail.com

Problem R2 02: DEL Command II (code: DELCOMM2)

Resource: Chinese National Olympiad in Informatics 1997, Day 1

Time Limit: 35 second

It is required to find out what's the maximum number of files that can be deleted from MS-DOS directory executing the DEL command of MS-DOS operation system only once. There are no nested subdirectories.

DEL command has the following format: DEL wildcard

The actual wildcard as well as a full file name can be made up of a name containing 1 up to 8 case-sensitive characters. In a wildcard the characters '?' and '*' can be used. A question mark substitutes exactly one character of the full file name, an asterisk any sequence of characters even empty one.

MS-DOS system can permit maybe other wildcards but they can not be used in this task. File names consist only of Latin letters and digits.

Input

The first line of the input is an integer M, then a blank line followed by M datasets. There is a blank line between datasets.

Input data for each dataset contains a list of full file names without any extra empty lines and spaces. Each name is written in a separate line of input data file and ended with a control sign: '+' for delete or '-' for keep. Full file names are not repeated. The list comprises at least one file, and at least one file is marked to be deleted. There are no more than 250 files.

Output

For each dataset, write to the first line of output the maximum number of files one DEL command can delete.

Sample

input	output
2 BP + BPC + TURBO - EXCHANGE + EXT + HARDWARE + MOUSE - NETWORK -	2 2

For the two tests above, the corresponding DEL commands are DEL BP* and DEL EX*.

Solution:

This problem requires of us to find a wildcard pattern which matches the largest amount of files, but not any of the files which are marked as "need to be saved". It means that we can create a list for each file, containing all patterns which match it. Then we just need to find the pattern which is contained in the

maximal number of “safe” lists, but in none of the “unsafe” ones.

The first problem with this idea is that the list of patterns can get really big, so we have to leave out some of them. Let’s see which patterns can be safely left out. For example, if a pattern contains “***” it’s equivalent to the pattern where one of the “*” is left out. Further, a pattern such as “?*?*?” is equivalent to “???*?” which is then equivalent to “???*”. Generalizing this insight, we conclude that we don’t have to look at any patterns containing consecutive “*”s or a “?” immediately following a “*”. This gives us a more manageable list – for example, a file named “AB” is matched by the following list of wildcard patterns:

??	?*B	A?*	A*B*	*AB	*A*
??*	?*B*	AB	A*	*AB*	*B
?B	?*	AB*	*A?	*A*B	*B*
?B*	A?	A*B	*A?*	*A*B*	*

It can be calculated that the worst case is when we have a file name containing 8 different characters, and that for this case, taking into account that the maximum allowed length of a pattern is 8 characters as well, the pattern list contains somewhere around 8000 elements.

To find a pattern which is contained in the maximal number of lists, we will make use of the **trie** data structure. Each pattern from each list is added to the trie, and we add 1 to the value of the node in the trie if the pattern was pulled out from a list corresponding to a file which can be deleted, and a *-infinity* value is added if the list was for a file that is not allowed to be deleted. we

In the end we just have to go through the trie and find the node with the maximal value.

The time complexity of inserting a pattern to the trie is $O(max_patern_length)$, which gives us the total time complexity $O(nF \cdot nW \cdot max_pattern_length)$ (where nF denotes the number of files ($nF \leq 250$) and nW the maximal number of wildcard patterns for some file ($nW \leq 8000$, as discussed)). The memory complexity has the same order as the number of nodes in the trie, so some care has to be taken not to allocate any unnecessary memory.

Solution by:

Name: *Dušan Zdravković*

School: *Računarski fakultet (RAF)*

E-mail: *duxxud@gmail.com*

Problem R2 03: Boxes (code: BOX)

Resource: POI III, stage 3; Special thanks to Lei Huang

Time Limit: 0.5 - 15 second

There are n boxes on the circle. The boxes are numbered from 1 to n in clock wise order. There are balls in the boxes, and the number of all the balls in the boxes is not greater than n .

The balls should be displaced in such a way that in each box there remains no more than one ball. In one move we can shift a ball from one box to one of it's neighboring boxes.

Write a program that: reads from the standard input the number of boxes n and the arrangement of balls in the boxes, computes the minimal number of moves necessary to displace the balls in such a way that in each box there remains no more than one ball, writes the result in the standard output.

Input

The first line of the input file contains an integer t representing the number of test cases. Then t test cases follows. Each test case has the following form:

- The first line contains one positive integer n - the number of boxes
- The second line contains n nonnegative integer separated by single spaces. The i -th number is the number of balls in the i -th box.

Output

For each test case, output one nonnegative integer - the number of moves necessary to displace the balls in such a way that in each box there remains no more than one ball.

Sample

input	output
1 12 0 0 2 4 3 1 0 0 0 0 0 1	19

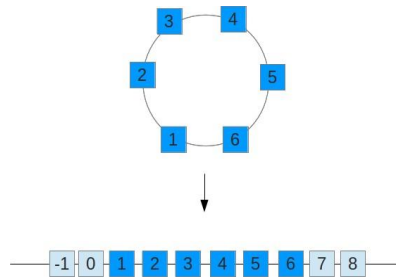
Note

There are two input files. In the first input file, $t=19$, $n \leq 1000$, time limit=0.5 second; In the second input file, $t=3$, $n \leq 200000$, time limit=15 seconds.

Solution:

In order to solve the problem above, we will first reformulate it.

We number the boxes in a clockwise order from 1 to n , beginning from an arbitrary box. Let $b(i)$ denote the box in which ball number i was placed at the beginning. We number the balls so that $i > j \Rightarrow b(i) \geq b(j)$. Then we "stretch" the boxes 1 ... n into a line and add infinitely many boxes before and after the line (they are numbered ... - 2, -1, 0 and $n + 1, n + 2, n + 3, \dots$ respectively).

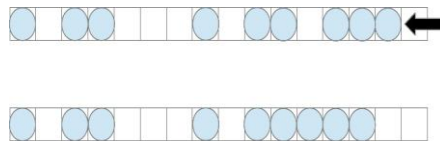


We displace the balls so that at most one ball remains in each box. The new position of each ball will be denoted $b^0(i)$. We will call $b^0(i)$ for $i = 1, 2, \dots, j$ a *solution* for balls $1, \dots, j$. Furthermore, a solution will be called *optimal* if its cost $S = \sum_{i=1}^m |b(i) - b'(i)|$ is minimal.

We will call the distance between the first and the last occupied box in a given solution (i.e. the number of the boxes between them plus 1) *span*. We want to find the optimal solution for balls $1, \dots, m$ with span at most $n - 1$.

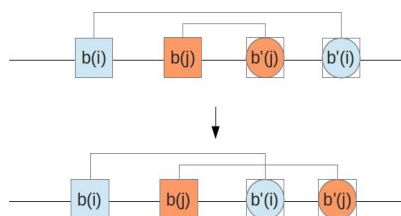
Before we do that, we will define an operation and introduce four lemmas.

We define an operation “push left” on a given solution. We take the rightmost ball and move it one box to the left. If the box is not empty, we take the ball from it and again move one box to the left. We continue this process until a ball is inserted into an empty box. In other words, we move the rightmost continuous block of balls one box to the left. This operation changes the cost of the solution by ΔS . If ΔS is a negative number, than the solution is improved. “Push right” is defined in an analogous way.



Lemma 1 There exists an optimal solution in which $i > j \Rightarrow b'(i) \geq b'(j)$ (1).

Let us take any optimal solution and assume that there exists such a pair (i, j) that $i > j$ and $b'(i) < b'(j)$. Since $b(i) \geq b(j)$, $|b(i) - b'(j)| + |b(j) - b'(i)| \leq |b(i) - b'(i)| + |b(j) - b'(j)|$. Therefore we may swap $b'(i)$ and $b'(j)$ and we obtain another optimal solution. Continuing this process for different pairs, like during a sorting algorithm, we may finally obtain an optimal solution where (1) is satisfied.



Lemma 2 When we perform subsequent “push left” operations, each time the change of the cost ΔS is greater or equal than the change last time. (The same holds for the “push right” operation.)

Let us assume that in the rightmost continuous block of balls there are l balls x such that $b'(x) \leq b(x)$, and r balls such that $b'(x) > b(x)$. The “push left” operation changes the cost of the solution by

$\Delta S = l - r$. After the operation l can increase or stay the same and r can decrease or stay the same. This proves the lemma.

Lemma 3 Let us consider all solutions such that boxes with indices greater than x are empty. We choose the optimal one among them and call it A . Now we want to find the optimal solution A' among all solutions in which boxes with indices greater than $x - 1$ are empty. If box number x was free in solution A , then we may take $A' = A$. Else we may obtain A' by performing “push left” on A . This follows from Lemmas 1 and 2.

Lemma 4 If i is the leftmost occupied box and j is the rightmost occupied box in an optimal solution with span $s = j - i$, then there exists an optimal solution with span $s - 1$ such that all occupied boxes are either in range $[i + 1, j]$ or $[i, j - 1]$.

Lemma 4 will be left without a proof. It can be proven by contradiction, when we consider rightmost continuous blocks of balls in both solutions and estimate their lengths.

Algorithm

We may use these lemmas to construct the algorithm.

1. Firstly, we want to find an optimal solution with any span. We keep in memory an optimal solution with a cost S for the balls $1 \dots i - 1$, (if there is more than one, we assume that we have any of them). Now we consider the ball number i . According to (1) we know that it is the rightmost ball in the optimal solution for the balls $1 \dots i$.

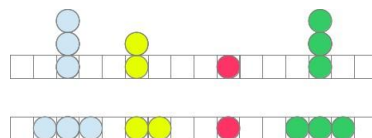
If $b(i)$ is free we set $b'(i) := b(i)$ and the new solution’s cost is S , so it is optimal.

If $b(i)$ is occupied, then we place the ball number i to the right of the rightmost ball in the previous solution. We perform “push left” as long as it decreases the cost of the solution. According to Lemma 3 this leads to an optimal solution.

2. After adding all m balls we get an optimal solution with span s . As long as s is greater than $n - 1$ we perform either “push left” or “push right”, each time we choose the operation that makes a smaller increase in the cost of the solution. According to the Lemmas 4 it leads to an optimal solution with a span $n - 1$.

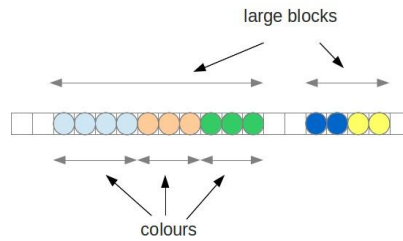
Implementation

Let us assign the same colour to balls which were in the same box at the beginning (which have the same $b(x)$). Balls in the same colour form one continuous block in the solution produced by the algorithm — this is an observation based on Lemma 1 and the workflow of the algorithm. The block of balls of the same colour (from the same box) will be called simply a *colour*. Each colour i in the solution may be characterized by three parameters: $l(i)$ — its leftmost ball, $r(i)$ — its rightmost ball and $b(i)$ — the primary position (box) of all the balls in that colour. Instead of considering single balls, we will consider colours. The figure below shows an example of an initial and final placement of the balls.



Furthermore, we will create a structure which will store information about continuous blocks of balls in the

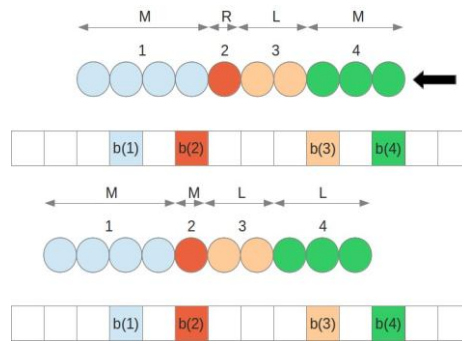
solution. We will call them *large blocks*. Large blocks consist of one or more colours.



We will divide colours in each large block into three groups:

- L — colours entirely to the left of their box of origin ($r(x) < b(x)$),
- R — colours entirely to the right of their box of origin ($l(x) > b(x)$),
- M — one of the balls in the colour x is in its box of origin ($l(x) \leq b(x) \leq r(x)$).

When we perform the operation “push left”, we move the rightmost large block one box to the left. It may make the colours in the large block change their groups ($L \leftarrow M \leftarrow R$).



If there is only a one-box distance between the rightmost and the second rightmost large block, than after “push left” we have to merge these two into one large block. In order to make the algorithm sufficiently fast, we have to concentrate on performing “push left”, updating the current cost of the solution and merging large blocks.

When we perform “push left”, we have to change the cost of the solution by

- $r(x) - l(x)$ for each colour x in L
- $(b(x) - l(x)) - (r(x) - b(x) - 1)$ for each colour x in M
- $-(r(x) - l(x))$ for each colour x in R

To update the cost of the solution in a fast way, we remember the sum of the lengths of the colours in L in R . Thanks to that, we calculate this part of ΔS in constant time. We calculate the change of the cost for each colour in M separately. This is performed in amortized constant time, because each colour x may be pushed left at most $r(x) - l(x)$ (its length) times when it is in group M , then it will travel left to its primary position and end up in group L . The sum of the lengths of all colours equals m .

For each large block, we may store information about colours from L and R in **binary search trees** (e.g. *set* from the C++ STL) and colours from the group M in a dynamic array (e.g. *vector* from the C++ STL). The colours in the BST structures are ordered by the number of remaining “push left” operations which will make them change their group. If this number for the minimal colour in R is equal to the number of

operations performed, than that colour is deleted from R and added to M . Colours in M are updated separately after each “push left”, they are added to L when they leave M . Colours in L cannot change their group any more after “push left”. However, it is useful to store information about them in a BST — we use it to construct a similar data structure to enable the “push right” operation in the second part of the algorithm.

The last thing that remains to be done is merging two large blocks. When we merge two groups of the same kind, we always add elements from a smaller container to a larger one. Thanks to this, every element will be inserted at most $\log m$ times. The most time-consuming operation in the algorithm is merging binary search trees, which in total takes $O(m \log^2 m)$ time (insertion of a single element into a BST takes $O(\log m)$ time).

Solution by:

Name: **Bartosz Tarnawski**

School: *Szramek High School Katowice*

E-mail: *bartek.tarnawski@gmail.com*

Problem R2 04: Cryptography (code: CRYPTO)

Time Limit: 1 – 11 second

Your task is to work as a cryptographer for some time, the reason is ...

Blue Mary has set a problem using English. Since the problem is too easy and it will be boring when solving it, she has deleted all the whitespaces and punctuations in the original problem description, and lowercased all the capital latin letters. Then, she randomly chose a permutation of the English lowercase letter alphabet, and then used the corresponding letters in place of the letters in the original text.

The encrypted text can be downloaded [here](#).

There is no example for this problem.

Blue Mary's note: some tricky test cases were added on Nov. 25th, 2007 and the time limit has been changed. Programs have been rejudged and some "accepted" solutions got Wrong Answer. However, this problem can still be solved by quite clean code with length less than 1KB.

Solution:

This task is a bit different from the others, since we are given an encrypted problem statement. Therefore, to solve the task, we need to decipher the statement. This also happens to be the main and interesting part of this problem, since the task itself is rather easy (except for some ambiguity in the real statement).

In the introduction to the task statement, we are provided with some important facts – we are dealing with lowercase Latin letters, the language is English and, the most important fact, the method of encryption is a form of substitution cipher.

One of the simplest and most commonly used methods for breaking the substitution cipher is frequency analysis. This method, coupled with some manual tweaking, is enough to decipher the task statement. It consists of counting the number of occurrences of each letter in the ciphertext – analysis of the frequency of each letter in the ciphertext. Then, the more frequent letters in the ciphertext are mapped to the more frequent letters in the English alphabet, while the less frequent are mapped to the less frequent letters in the English alphabet. For this, we will, of course, need relative frequencies of letters in the English alphabet. Then, in the best case, the most frequent letter in our ciphertext will map to the letter 'e' (the most frequent letter in English language), the second most frequent letter will map to the letter 't' and so on... Of course, the process of decryption would be too easy then, so this is not the best case and we will have to do some guessing. We can start by mapping the most frequent letters in the given text to the most frequent letters in the alphabet, and then build it up from there by looking at the text we are getting when applying our partial key, mixing the mappings a little bit, trying to get meaningful words and parts of words. We can also use the frequency analysis of bigrams (two adjacent letters in the string) or trigrams. For example, the most frequent trigram in the given text is 'nte', and it maps to the most frequent trigram in the English language 'the'. Slowly, by correctly guessing the parts of the needed key (permutation), we will be getting more and more meaningful words, ultimately leading to this (spaces and punctuation included additionally to show the real beauty of the statement):

"I have told you that this problem is very easy, just like the problem life the universe and everything.

Yes, it's very easy, but you will have some trouble to get accepted if you don't care with the problem description.

A very easy problem is not always a very boring problem.

Sometimes it will be interesting from the time you try it to the time you solve it because after lots of WA/TLE you will feel very well.

The sun is shining outside the window. The birds are singing in the tree. The nature is harmonious.

What's the first problem of an online judge?

It's A plus B problem in common except sphere online judge.

This problem will be the first problem of most of the online judges.

The data limitations are multiple test cases.

All the numbers are separated by some whitespaces. A and B are integers and less than MAX_LONG_INT in C or C++ language.

The input file will less than four megabytes.

I hope you can solve it as fast as possible because it's very very easy and the test cases are very weak. But if you got WA/TLE you will work for a long time to find the mistake of your program possibly.

Enjoy this problem “

Alright, the text may not be so beautiful at the end, and actually, it does not describe the problem very well. Most of the people probably felt excitement and relief after successfully deciphering the text, but it was only the beginning of a bumpy road to 'Accepted'. The task is to print the sum of two numbers, but the format of the input (and output) is still unknown, and the constraints are vague. Therefore, the method of trial and error would have to continue to find out all the tricks that are hidden in this task. The first, and the easier problem is to find out how the input looks like. By testing different solutions, it turns out that the input consists of an array of $2N$ integers (there is no number in the input that tells us the value of N). Let V be the given array. You have to print exactly N integers (separated by some whitespace), where each integer represents the value of expression $V[2i-1]+V[2i]$, for each $i=1,\dots,N$. The second and more tricky part of solving this problem is figuring out the constraints. It is stated that all the integers will be less than MAX_LONG_INT. A lot of people initially assumed that their absolute value is less than MAX_LONG_INT. But, it is a very wrong assumption, and, as it turns out, there is no lower bound on the value of numbers, other than the size of the input file. Therefore, big integer arithmetics has to be used to solve the task. Since the only operation is addition (and subtraction in case of two numbers with different signs), the implementation is straightforward. Time limit should not be an issue in this task, since the time complexity of the solution is linear, but reading the input should be done with care.

Solution by:

Name: Vanja Petrović Tanković

School: School of Computing, Belgrade

E-mail: vpetrovictankovic@gmail.com

Problem R2 05: Slow Growing Bacteria (code: SBACT)

Resource: MNNIT IOPC 2010, Co-author: jitendra_kumar

Time Limit: 2.0 second

Given an $n \times n$ grid of cells, a bacteria colony can colonize these cells. Their growth after every second is governed by the following rules:

- 1) One new bacteria is born in cell (i, j) if and only if either one of its four neighboring cells or the cell (i, j) itself has a bacteria population more than or equal to the threshold value, k .
- 2) Already living bacterias do not die.

Given, the initial state of the $n \times n$ cell grid, you need to accurately estimate the time by when the total bacteria population reaches m .

Input

First line contains t , number of test cases. Each test case starts with n (side length of grid) , k (growth threshold) and m (final population). Next n lines contain an $n \times n$ grid of integers, where i th row, j th column has an integer representing the number of bacteria's present initially at cell (i, j) . $1 < n \leq 100$; $0 < k \leq 2^{45}$, $0 < m \leq 2^{45}$; There are no more than n cells with initial population equal to or greater than k .

Output

For each test case print the number of seconds required for the total bacteria population to reach m . If m can never be reached print "Not possible" (quotes for clarity).

Sample

input	output
1 3 5 15 0 0 0 0 3 0 0 0 5	3

Solution:

Given an $n \times n$ array of integers called A and an integer K , find the first moment when the sum of the elements in the array is greater or equal to a given integer M . Each second we apply the rule: if the value in the cell (i, j) or its neighbours is greater or equal to K , increase it by one.

Basic idea:

The first thing to notice is that the constraints on n are relatively small , while K and M can be huge. So the complexity of the solution must not depend on K and M much (maybe logarithmically proportional, but not more).

There are two basic ideas on similar problems – do a binary search on the answer or just simulate the process. The binary search doesn't seem to be a good idea because when we fix the answer, there is no easy way (at least I couldn't find one) to check if the answer is achievable (the only thing which comes to mind is again simulation). So, our next stop is to think whether simple simulation is possible within the

constraints.

With n up to 100 we can easily simulate each step, but the problem is that we may need to simulate billions of steps. But, many of them will do the same thing – in fact we will have only $O(n \cdot n)$ steps in which something changes. Really, the only change occurs when the value of a cell becomes greater or equal to K (because it activates its neighbours), and this can happen at most once per cell. This idea must be enough to solve the problem!

So, we are ready with the basis of the solution – group the steps and use a single iteration to simulate the whole group.

Implementation:

There are different ways to implement the above idea with complexities ranging from $O(n^4)$ to $O(n^2 \log n)$, and maybe even better. I will explain the $O(n^2 \log n)$ solution I used.

I maintain three groups of cells:

- already with value greater or equal to K
- With value less than K , but increasing
- The rest of the cells.

Apparently, the work we do each moment only changes when something from 2) moves to 1). I remember this (future) moment for each item in 2) and in each step choose the group in 2) with the smallest moment (I group the elements in 2) with equal "move" times; this is not necessary). Then, for the time from the previous interesting point (of course, the first interesting point is 0) to this one I know how the total count has changed because it depends only on the sizes of 1) and 2) and the interval of time between the two points. After finishing with this, I update the information for the neighbours of the current cells. If I use a data structure which supports the priority queue operations in $O(\log n)$ time (for example, `set` in the C++ STL), I get the desired complexity.

Also, when 2) becomes empty, nothing will change from then to infinity, so it is easy to count the remaining time (if any) for the sum to reach M . There are some things to be careful about:

- 1) Not to forget to use long long (or similar)
- 2) If the sum in the table is fine in the beginning, we should just print 0 and stop
- 3) If in the beginning we do not have active elements and 2) is not satisfied, print "Not possible"
- 4) Be careful in the initialisation – it is easy to send an element not in the right group
- 5) Also, remember to round up when dividing
- 6) One last thing - we must not forget to break the simulation when the sum reaches M

A final note about the complexity:

For each of the cells we will look at most once at its four neighbours and maybe put them in the priority queue (if they are not already there). This takes $O(n^2 \log n)$ time. This time dominates all other steps, so the total complexity is $O(n^2 \log n)$ as well.

Solution by:

Name: *Vladislav Haralampiev*

School: *SU "St. Kliment Ohridski"*

E-mail: *vladislav_haralampiev@abv.bg*

Problem R2 06: Reverse the sequence (code: REVSEQ)

Time Limit: 1 second

This is a very ad-hoc problem. Consider a sequence (N, N-1, ..., 2, 1). You have to reverse it, that is, make it become (1, 2, ..., N-1, N). And how do you do this? By making operations of the following kind.

Writing three natural numbers A, B, C such that $1 \leq A \leq B < C \leq N$ means that you are swapping the block (block = consecutive subsequence) of elements occupying positions A..B with the block of elements occupying positions B+1..C. Of course, the order of elements in a particular block does not change.

This means that you can pick any two adjacent blocks (each of an arbitrary length) and swap them. The problem can easily be solved in N-1 operations, but to make it more difficult, you must think of a faster way.

Input

A natural number $1 < N < 100$.

Output

Output at most 50 operations, one per line. Each operation is represented by three numbers as described above.

Sample

input	output
5	2 3 5 1 2 4 2 3 5

Explanation of the sample output: (5 4 3 2 1) --> (5 2 1 4 3) --> (1 4 5 2 3) --> (1 2 3 4 5)

Solution:

Since the limitations for N are $2 \leq N \leq 99$, and we shouldn't output more than 50 operations, we're looking for a solution which will take around N/2 steps. We will describe an algorithm which takes N/2 + 1 steps for even N and (N-1)/2 + 1 for odd N (or, simply, N div 2 + 1 steps for all N), except for N=2 where 1 step is sufficient.

Even N

To demonstrate how the algorithm reverses the sequence in the given number of steps, let N=10. The sequence looks like this at first:

10, 9, 8, 7, 6, 5, 4, 3, 2, 1

Phase 1

Starting from the number with the index N/2 - 2, we let the first sequence be that number and the next one, and we let the second sequence be the following N/2 - 1 numbers. We then swap those two sequences.

10, 9, **8, 7**, 6, 5, 4, 3, 2, 1 => 10, 9, 6, 5, 4, 3, **8, 7**, 2, 1

Now, we swap the sequences of the same length (two and N/2 - 1), but, this time the starting index will be N/2 - 3.

10, **9, 6**, 5, 4, 3, 8, 7, 2, 1 => 10, 5, 4, 3, 8, **9, 6**, 7, 2, 1

We repeat this procedure - the lengths of the sequences to be taken are not changed, and the index of the number from the beginning of the first sequence is decreased by 1 until we run out of indices. That means this phase takes $N/2 - 2$ steps.

10, 5, 4, 3, 8, 9, 6, 7, 2, 1 => 4, 3, 8, 9, **10, 5**, 6, 7, 2, 1

After these steps the sequence will, in general, look like this:

4, 3, $N/2 + 3$, $N/2 + 4$, $N/2 + 5$, ..., N , 5, 6, 7, ..., $N/2 + 2$, 2, 1

For $N=10$: 4, 3, 8, 9, 10, 5, 6, 7, 2, 1

Phase 2

Now we are just three steps away from reversing the sequence!

First, swap the sequences $[2, N/2]$ and $[N/2 + 1, N]$:

4, **3, 8, 9, 10**, 5, 6, 7, 2, 1 => 4, 5, 6, 7, 2, 1, **3, 8, 9, 10**

Now, swap $[1, N/2 - 1]$ and $[N/2, N/2 + 2]$:

4, 5, 6, 7, 2, 1, 3, 8, 9, 10 => **2, 1, 3**, **4, 5, 6, 7**, 8, 9, 10

Now swap the first two numbers, and you're done! The first phase takes $N/2 - 2$ steps, and the second phase takes 3 steps, for a total of $N/2 + 1$ steps.

Odd N

The general idea for odd N is the same as for even N . In phase 1, we start reversing sequences of lengths 2 and $(N+1)/2$ from the index $(N-1)/2 - 1$. This takes $(N-1)/2 - 1$ steps, that is, $N \text{ div } 2 - 1$ steps. The sequence will look like this after phase 1:

3, 2, $(N-1)/2 + 3$, $(N-1)/2 + 4$, ..., N , 4, 5, 6, 7, ..., $(N-1)/2 + 2$, 1

The phase two, however, takes just two steps - swap $[2, (N+1)/2]$ and $[(N+1)/2 + 1, N]$ and $[1, (N-1) / 2]$ and $[(N+1)/2, (N+1)/2 + 1]$.

Demonstration of phase 2 for $N=11$:

3, **2, 8, 9, 10, 11**, 4, 5, 6, 7, 1 => 3, 4, 5, 6, 7, 1, **2, 8, 9, 10, 11**
3, 4, 5, 6, 7, 1, 2, 8, 9, 10, 11 => **1, 2**, **3, 4, 5, 6, 7**, 8, 9, 10, 11

The phase 1 takes $(N-1)/2 - 1$ steps, and phase two takes 2 steps, for a total of $N \text{ div } 2 + 1$ steps.

Small N

For $N=3$ and $N=4$, phase 1 is omitted. For $N=2$, we just swap those two numbers.

Conclusion

The implementation of the algorithm is easy, although reaching it is far more difficult. The complexity of the program itself is $O(N)$ - it doesn't compute anything, just prints some predetermined numbers.

Solution by:

Name: **Ivan Stošić**

School: Gymnasium "Svetozar Marković", Niš

E-mail: ivan100sic@gmail.com

Problem R2 07: Cover the string (code: MAIN8_E)

Resource: Mahesh Chandra Sharma, used for NSIT-IIITA Main contest #8

Time Limit: 4.0 second

Given two strings A and B. You have to find the length of the smallest substring of A, such that B is the subsequence of this substring.

Formally speaking: you have to find the length of smallest possible string S such that

S is the substring of A

B is the subsequence S

Note: Subsequence of an string S is obtained by deleting some (possibly none) of the characters from it. for example "ah" is the subsequence of "aohol"

Input

First line contains T, the number of test cases. Then T test cases follow, 2 lines for each test case, 1st contains A and 2nd contain B.

$|A| \leq 20000$, $|B| \leq 100$

Output

For each test case print the answer in a new line. if no such substring exists print -1 instead.

Sample

input	output
2	10
aaddddcckhssdd	3
acs	
ghkklhllhjkthkjjht	
hh	

Solution:

Since we are looking for the smallest possible string S it is quite obvious that S will start with the first character of string B and end with the last character of string B. Let all strings be zero-indexed and let N be equal to |A| and M be equal to |B|. We will try to build the solution from any occurrence of character B[0] in string A. Observe that there is no point in skipping characters of string B in string A.

For example, let's say that string A is equal to "aabcbbcddeef" and string B is equal to "abc". The solution will of course be "aabc" and not "aabcbbc", so we conclude that we need to use some kind of a greedy algorithm to solve this task. The main idea of the algorithm is that after we have string S_i , which is a substring of string A and has first $i + 1$ characters of string B as its subsequence, we try to build string S_{i+1} which will be a substring of string A and will have first $i + 2$ characters of string B as its subsequence and string S_i will be prefix of string S_{i+1} .

So how do we do that?

We can expand string S_i to string S_{i+1} by concatenating all characters that come after the last character of string S_i until we reach character B[i] (and that character will be the last character of string S_{i+1}).

From all strings S_{M-1} the one with the smallest length will be the solution.

Now we just need to find some quick way to expand any string S_i to string S_{i+1} . We will need some matrix $Next[N][26]$ and $Next[i][j]$ will tell us the position of the first occurrence of j -th character of the alphabet (0 – indexed) after position i in string A .

How to build matrix $Next$?

We traverse the string A in reverse and we update array $Last[26]$ where $Last[i]$ is equal to the last position of i -th character while traversing the string in reverse and we can see that when we get to position i , the array $Last$ will represent row $Next[i]$.

Now let's summarize the solution:

- Build matrix $Next$
- Try to build string S_{M-1} starting from any occurrence of character $B[0]$ in string A
- Use matrix $Next$ to expand any string S_i to S_{i+1}

If we were unable to build any string S_{M-1} we output -1 , otherwise we output the length of the smallest string S_{M-1} .

Solution by:

Name: **Aleksandar Ivanović**

School: Prva kragujevačka gimnazija

E-mail: aleksandar.ivanovic.94@gmail.com

Problem R2 08: Dynamic LCA (code: DYNALCA)

Time Limit: 0.5 second

A forest of rooted trees initially consists of N ($1 \leq N \leq 100,000$) single-vertex trees. The vertices are numbered from 1 to N .

You must process the following queries, where ($1 \leq A, B \leq N$) :

- link A B : add an edge from vertex A to B, making A a child of B, where initially A is a root vertex, A and B are in different trees.
- cut A : remove edge from A to its parent, where initially A is a non-root vertex.
- lca A B : print the lowest common ancestor of A and B, where A and B are in the same tree.

Input

The first line of input contains the number of initial single-vertex trees N and the number of queries M ($1 \leq M \leq 100,000$). The following M lines contain queries.

Output

For each lca query output the lowest common ancestor (vertex number between 1 and N)

Sample

input	output
5 9	1
lca 1 1	2
link 1 2	3
link 3 2	2
link 4 3	
lca 1 4	
lca 3 4	
cut 4	
link 5 3	
lca 1 5	

Solution:

First of all, if there were no "cut" and "link" operations, and all nodes were connected, how would we solve the problem? One of the methods to find LCA is this: create an empty array and make a DFS order of the tree. As we traverse edges (either forwards or backwards), whenever we enter a node, append that node to the end of the array. This array will contain $2n$ elements in the end. Note that this does not mean each node appears twice in the array! Now to find LCA of nodes A and B, pick any of their occurrences in the array and find the node with minimal depth between them. This can be done in $O(\log n)$ using any range-minimum-query (RMQ) data structure.

Let's see what happens to the array containing nodes in DFS order on operation "cut": if we cut node A, the interval beginning from the first occurrence of node A and ending at the last occurrence of node A is being cut from the array and becomes a new array. What about operation "link A B"? The array starting and ending with node A is plugged in the array containing node B between two consecutive B's. Note that cutting and linking may require slight modifications like duplicating a node or removing a duplicate to fix

the DFS order.

The last obstacle to solving this problem is making all operations fast. There's an awesome data structure called the **splay tree**. Using splay trees we can easily cut and paste intervals and query for minimum in an interval. The idea is to just naively implement operations, and make sure that we splay any node we were looking for by traversing any tree from the root. This magically gives $O(\log^2 n)$ per operation.

Solution by:

*Name: **Stjepan Glavina***

School: FER Zagreb

E-mail: stjepang@gmail.com

Problem R2 09: Magic Bitwise AND Operation (code: AND)

Resource: Fudan University Local Contest #3, by g201513

Time Limit: 23 second

Given n integers, your task is to pick k out of them so that the picked number are minimum when do bitwise AND among all of them.

Input

There are multiple test cases for this problem. The first line of the input contains an integer denoting the number of test cases.

For each test case, there are two integers in the first line: n and k , denoting the number of given integers and the number of integers you are asked to pick out. ($1 \leq n \leq 40, 1 \leq k \leq n$)

The second line contains the n integers. You may assume that all integers are smaller than 260.

Note: There are about one thousand randomly generated test cases. Fortunately 90% of them are relatively small.

Output

For each test case, output only one integer - the smallest possible value.

Sample

input	output
2	Case #1: 4
3 2	Case #2: 9
5 6 7	
8 2	
238 153 223 247 111 252 253 247	

Solution:

For solving this task we will require some basic knowledge of binary numbers and bitwise operations. The solution for this task is some kind of a pruning search. We will use the most basic recursion for finding all the k -combinations of n elements and use some optimizations to make it faster. Observe that for all numbers A and $B, B \geq A$, the following inequality holds: $A \& B \leq A$, where $\&$ represents the bitwise AND operation. So first of all, we will sort the numbers in non-decreasing order. We set the initial solution to be some very large number (for the limits in this task $2^{63} - 1$ will suffice). We use the following optimizations: if we select exactly K numbers or we reach the end of the array we don't go any further in the recursion. Also, since we concluded that including larger numbers in the combination will make that combination less or equal to the previous one, we can check if including all the numbers after the current position will give us some number that is greater or equal to the solution and if that is the case we don't go further in the recursion either. For that optimization we will need to use some preprocessed array $DP[N]$, where $DP[i]$ is equal to $A[i] \& A[i + 1] \& \dots \& A[N - 1]$ (all arrays are zero-indexed). We can preprocess array DP in either linear or quadratic complexity, it doesn't affect the total time that much. One more optimization is to check in every step if the current combination is smaller than the best solution so far and update the solution if that is the case. Using all of the above optimizations will be enough to solve the task within the

given time limit. The recursion itself is very simple to code and very short. The implementation follows:

```
void Solve ( int idx, int Cnt, lld Curr )
{
    if ( Curr < Sol ) Sol = Curr;
    if ( ( Cnt == K ) || ( idx == N ) ) return;
    if ( ( Curr & DP[idx] ) >= Sol ) return;
    Solve ( idx + 1, Cnt + 1, Curr & A[idx] );
    Solve ( idx + 1, Cnt, Curr );
}
```

(*idx* represents the current position in the array, *Cnt* represents the quantity of numbers that we take in the combination and *Curr* represents the value of the combination.)

Solution by:

Name: **Aleksandar Ivanović**

School: Prva kragujevačka gimnazija

E-mail: aleksandar.ivanovic.94@gmail.com

Problem R2 10: Contaminated City (code: 1CONTCITY 19)

Time Limit: 2.0 second

In a far away country there is a city facing a big problem. The city is plagued by a deadly gas. Many people have died, but there are groups of survivors at places around the city. Between these places there are roads connecting two distinct places that can still be traversed safely. These roads can be traversed in both directions. It's known the number of days necessary to traverse each road and the two places that it connects. It's also known the number of survivors at each location. Each survivor can get to other places following a sequence of roads.

The mayor will send several helicopters to rescue these people, each having a capacity, a limit on the number of crew (people that it can rescue). Each helicopter will land on a certain day and place.

You should answer an important question for the mayor. How many days are needed to rescue all survivors? If it's not possible to rescue all people you should answer how many of them can be rescued.

Input

The first line of input file have the number of test cases T ($T \leq 40$). The first line of each test case have N , M , and H , the number of places considered, the number of roads between the places and the number of helicopters that will be sent, respectively. Each place is uniquely identified by a number between 1 and N . The next N lines will have N integers, the i -th line have the number of survivors in place i , X_i . Each of next M lines will have three numbers A_j , B_j and D_j , meaning that there is a way between places A_j and B_j that last D_j days to traverse. The input can contain several roads between the same pair of places. Each of next H lines will have three integers D_h , P_h , and C_h (in this order), meaning that a helicopter with capacity C_h will arrive at place P_h at day D_h . The sum of survivors will not be more than 200. If a survivor can get a helicopter following a sequence of roads, the total time to get the helicopter will not be more than 1000.

Constraints

- $1 \leq N, H \leq 50$
- $1 \leq M \leq 1500$
- $1 \leq A_j, B_j, P_h \leq N$
- $1 \leq D_j, D_h \leq 1000$
- $1 \leq C_h \leq 200$
- $0 \leq X_i \leq 200$

Output

For each test case there is one line in output. If all people can be rescued "All people can be rescued in D day(s) ." should be printed, where D is the minimum number of days to rescue all people. If it is impossible to rescue all people " X survivor(s) can be rescued." should be printed, where X is the maximum number of survivors that can be rescued.

Sample

input	output
2 4 4 4 3 4 5 6 1 2 7 2 3 3 3 4 3 4 1 4 4 4 7 6 3 2 5 2 3 3 1 6 4 2 3 2 2 3 1 1 4 3 2 3 3 2 4 2 3 2 4 3 3 2	All people can be rescued in 6 day(s) . 7 survivor(s) can be rescued.

Solution:

Let's reformulate this task in graph theory. Cities and helicopters will represent vertices in this graph and there will be an edge e between city u and helicopter v if there is a path between u and the city that helicopter v lands on.

Firstly, let's find the maximum number of people that can be rescued without regard for the time needed. This subproblem can be easily solved using the well-known **max flow** algorithm. We can see that costs of edges and times of helicopter landing don't play any role here, so we can forget about them. Let's create two new vertices: *source* and *sink*, with *source* being connected to each city through an edge with capacity equal to the number of people in city u and *sink* being connected to every helicopter v through an edge with capacity equal to the capacity of helicopter v . We will set the capacity of edges between cities and helicopters to infinity. Now it's easy to see that the flow between *source* and *sink* will be the maximum number of people that can be rescued.

Now, if not all people can be rescued, we should just output this number. Otherwise we need to find the minimal time needed to rescue all the people. In order to do that, we will make a use of the previous idea. Let's try to answer the question whether we can save all the people within time D . If we remove all the helicopters that land after time D , and if we remove all the edges (u, v) between the city u and the helicopter v where there is no path of a distance no more than D , then the flow between *source* and *sink* will tell us maximum number of people that can be rescued within time D .

Finally, let's note that if we can save all the people within time D , then we can save all the people within time $D+1$, $D+2$, ... as well. Because of this we can do a binary search over time D , and find the minimal time

needed to save all the people.

Time complexity of this algorithm is: $O(N^3 + MaxSurvivors * (N + M) * \log(MaxTime))$ where *MaxSurvivors* is the maximal number of people that can be saved (up to 200), and *MaxTime* is the maximal time needed to reach a helicopter from a city (no more than 1000). The $O(N^3)$ factor is there because we have to find the minimal distance between each two cities, and it can be done using the **Floyd-Warshall algorithm**.

Solution by:

Name: Boris Grubić

School: Gimnazija Jovan Jovanovic Zmaj

E-mail: borisgrubic@gmail.com

Problem R3 01: Four Mines (code: MINES4)

Resource: ThreeMines from TopCoder SRM 315 extended

Time Limit: 16 second

A Company that Makes Everything (ACME) has entered the surface mining business. They bought a rectangular field which is split into cells, with each cell having a profit value. A mine is a non-empty rectangular region, and the profit of a mine is equal to the sum of the values of all its cells. ACME wants to extract ore from four different non-overlapping mines. You are to choose these mines to maximize the total profit.

Input

The first line contains an integer T ($1 \leq T \leq 5$), denoting the number of test cases.

For each test case, the first line contains two positive integers R and C ($2 \leq R, C \leq 100$), denoting the number of rows and columns of a rectangular field.

Each of next R lines contain C integers between -10000 and 10000 , denoting a profit value for each cell in that row.

Output

For each test case, print a number on its own line, denoting the maximum total profit that can be extracted from four mines.

Sample

input	output
2	99
5 5	60
10 10 -1 -1 10	
10 -1 -1 -1 10	
-1 -1 -1 -1 -1	
-1 -1 -1 10 10	
10 -1 -1 10 10	
2 3	
-1 -2 -3	
-4 -5 66	

Solution:

As it turned out, this was the hardest problem in the BubbleCup qualification rounds. Here we are asked to put four rectangles on a given rectangular field without overlapping such that the sum of values inside those four rectangles is maximized.

This problem can be solved using dynamic programming. It's easy to see that a "state" can be easily represented by 5 numbers $r1, c1, r2, c2, left$ where $r1, c1$ represents upper left and $r2, c2$ bottom right corner of a rectangle in a given rectangular field where we are trying to put $left$ number of rectangles such that we maximize the sum of numbers inside those rectangles. Let's represent these values by a matrix $dp[r1][c1][r2][c2][left]$.

For $left = 2$ we can see that however we set two rectangles, there will always exists such a vertical or horizontal line that doesn't pass through any rectangle (see figure 1). The same holds for $left = 3$, as

shown in figure 2. For $left = 4$ there exist only 2 cases where this is not possible and those two cases are shown in figure 3. Let's first solve the problem not considering those 2 cases.

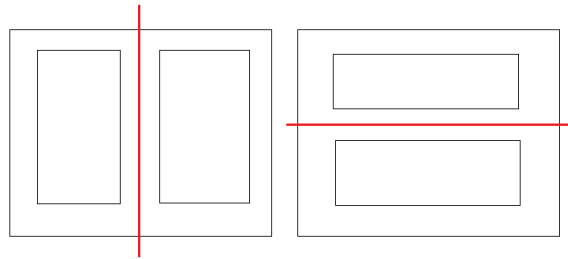


Figure 1

To solve this we will make a use of this property we have just noted. So, since every case can be “separated” by a vertical or horizontal line, we can just simply try every possible way. So, we can derive the following recurrent equations (for $left \geq 2$):

$$\begin{aligned}
 HorMax &= \max(dp[r1][c1][nr - 1][c2][left1] + dp[r1][c1][nr][c2][left2]) \\
 VerMax &= \max(dp[r1][c1][r2][nc - 1][left1] + dp[r1][nc][r2][c2][left2]) \\
 dp[r1][c1][r2][c2][left] &= \max(HorMax, VerMax) \\
 &\text{for } r1 < nr \leq r2, c1 < nc \leq c2, 1 \leq left1, left2 \text{ and } left1 + left2 = left
 \end{aligned}$$

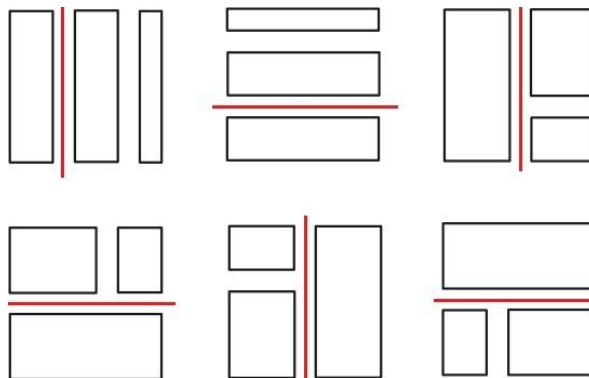


Figure 2

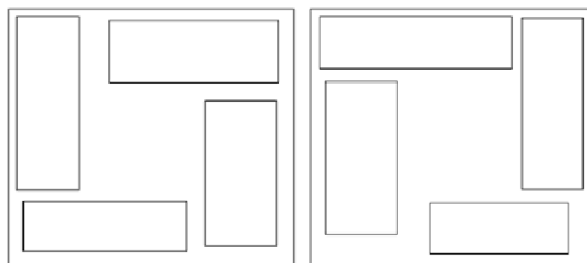


Figure 3

For $left = 1$, we can note that we will either take the whole observed rectangle, or we will remove the first or the last column or the first or the last row. So if we represent the sum of cells in rectangle $(r1, c1) -$

$(r2, c2)$ by a function $sum(r1, c1, r2, c2)$ we have:

$$\begin{aligned} Row &= \max(dp[r1 + 1][c1][r2][c2][1], dp[r1][c1][r2 - 1][c2][1]) \\ Col &= \max(dp[r1][c1 + 1][r2][c2][1], dp[r1][c1][r2][c2 - 1][1]) \\ dp[r1][c1][r2][c2][1] &= \max(sum(r1, c1, r2, c2), Row, Col) \end{aligned}$$

Please note that in the given equations we should take care of special cases where $r1 = r2$ or $c1 = c2$. In the former case we should set $HorMax = -infinity$ and $Row = -infinity$ and in the latter $VerMax = -infinity$ and $Col = -infinity$.

Now let's deal with this sum function. This is well known and used really often. Let's represent by $s[r][c]$ the sum of cells in the rectangle $(1,1) - (r, c)$. How to calculate it? Well it's easy to see that the following holds:

$$\begin{aligned} s[r][c] &= a[r][c] + s[r - 1][c] + s[r][c - 1] - s[r - 1][c - 1], \text{ for } r, c \geq 2 \\ s[r][c] &= a[r][c] + s[r - 1][c], \text{ for } r \geq 2, c = 1 \\ s[r][c] &= a[r][c] + s[r][c - 1], \text{ for } c \geq 2, r = 1 \\ s[r][c] &= a[r][c], \text{ for } r = 1, c = 1 \end{aligned}$$

where $a[r][c]$ represents the value of the cell in row r , column c in the given rectangular field.

So, how can we make a use of this matrix to calculate the sum in rectangle $(r1, c1) - (r2, c2)$? Well, we can use the inclusion-exclusion principle and derive the following:

$$\begin{aligned} sum(r1, c1, r2, c2) &= s[r2][c2] - s[r1 - 1][c2] - s[r2][c1 - 1] + s[r1][c1], \text{ for } r1, c1 \geq 2 \\ sum(r1, c1, r2, c2) &= s[r2][c2] - s[r1 - 1][c2], \text{ for } r1 \geq 2, c1 = 1 \\ sum(r1, c1, r2, c2) &= s[r2][c2] - s[r2][c1 - 1], \text{ for } r1 = 1, r2 \geq 2 \\ sum(r1, c1, r2, c2) &= s[r2][c2], \text{ for } r1 = 1, r2 = 1 \end{aligned}$$

Let's analyze the running time of this solution so far... One thing to note is that starting from rectangle $(1,1) - (n, m)$, where n and m represent number of rows and number of columns of a given rectangular field respectively, in one cut we can get to one of the following rectangles (see figure 4):

$$(1,1) - (x, m), (x, 1) - (n, m), (1,1) - (n, y), (1, y) - (n, m)$$

Note that here we have $O(n)$ states.

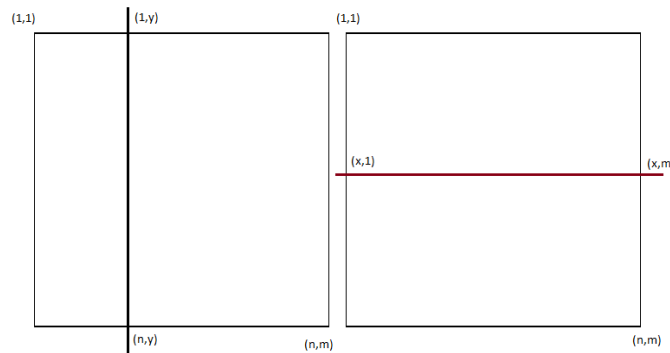


Figure 4

Following this, in two cuts we can get into following configurations (see figure 5):

$$(1,1) - (x, y), (1, y) - (x, m), (x, 1) - (n, y), (x, y) - (n, m), (1, x) - (n, y), (x, 1) - (y, m)$$

Here we've got $O(n^2)$ different states.

Again... in three cuts we can get into following configurations (see figure 6):

$$(x1,1) - (x2, y), (1, y1) - (x, y2), (x1, y) - (x2, m), (x, y1) - (n, y2)$$

And finally here we have $O(n^3)$ different states.

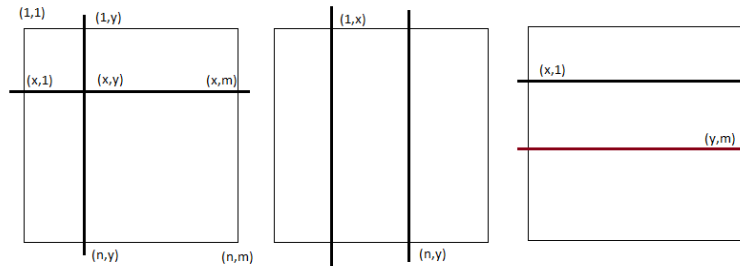


Figure 5

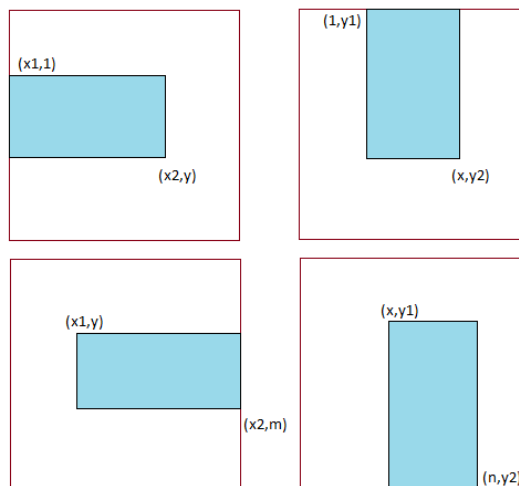


Figure 6

Note that after three cuts we have $left = 1$ and we have pretty nice dynamic there in $O(n^4)$ (we aren't looping through rows or columns to find a cut line), but in other cases we do have that loop and it takes $O(n)$ time. So the running time is $O(1 * n + n * n + n^2 * n + n^4) = O(n^4)$. This is pretty good, but we can do better. Note that the only thing that is slowing out program is the case for $left = 1$ and there we haven't used the property that we need only $O(n^3)$ different states.

How to speed that case up? Here we will solve only for the case $(x1,1) - (x2, y)$, and the other 3 cases are solved similarly. Let's represent the solution of this case by $dp1[x1][x2][y]$. Note that we will either take the whole rectangle, or we will erase the first or last column or the first or last row. Let's forget for a moment about erasing the first or the last column. So we've got:

$$Row = \max(dp1[x1 + 1][x2][y], dp1[x1][x2 - 1][y])$$

$$dp1[x1][x2][y] = \max(Row, sum(x1,1, x2, y))$$

Of course, for case $x1 = x2$, we should set $Row = -infinity$.

Now after we've got this solution, let's get back to erasing columns. After erasing first or last column multiple times, we will end up with a rectangle of type $(x1, t1) - (x2, t2)$. Note that our rows are fixed. So, the solution will be the sum of consecutive columns where rows are fixed. Let's say $p[t] = \text{sum}(x1, t, x2, t)$. The sum of rectangle $(x1, t1) - (x2, t2)$ will be $S = \sum_{i=t1}^{t2} p[i]$, and we are looking for maximum value. This is pretty well known problem. As we increase y from 1 to m we keep a track of maximum sum of consecutive columns ending at y . Once this sum goes below 0, we set it to 0. The following pseudocode describes this fully:

```

For d->1 to n:
  For x1->1 to n-d:
    x2 = x1 + d
    best = -infinity
    sum = 0
    For y->1 to m:
      sum = sum + p[y]
      best = max(best, sum)
      dp1[x1][x2][y] = best
      if d > 1:
        dp1[x1][x2][y] = max(dp1[x1][x2][y], dp1[x1+1][x2][y])
        dp1[x1][x2][y] = max(dp1[x1][x2][y], dp1[x1][x2-1][y])
      if sum < 0:
        sum = 0
    endfor
  endfor
endfor

```

Note that the running time of this algorithm is $O(n^3)$. So the total running time is $O(n^3)$ now, which is great.

Now let's crack those two cases in picture 3. Here we will "crack" only the first case, because the second one can be solved similarly. Here we will use more dynamic programming. Let's fix two columns and one row, like in Figure 7. Now we would like to compute the best way to put the left and the bottom rectangle such that the sum of cells inside them is maximal, the left rectangle is bounded by column $c1$ and the bottom rectangle is bounded by row r and column $c2$.

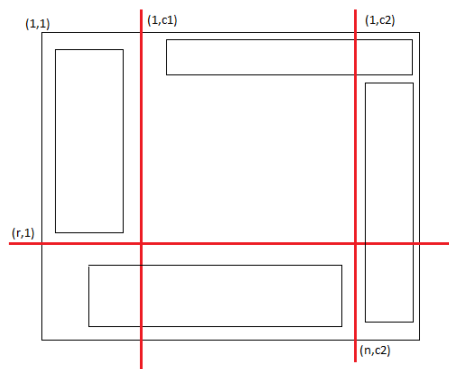


Figure 7

Let's represent this value by $f[c1][c2][r]$. It's easy to see that $f[c1][c2][r]$ is at least as $f[c1][c2][r + 1]$. This is really nice, because the only thing that's left to be checked is if bottom rectangle is as large as possible. So, we have the following recurrent relation:

$$A = dp[r][1][n][c2][1] + dp[1][1][r - 1][c1][1]$$

$$B = f[c1][c2][r + 1]$$

$$f[c1][c2][r] = \max(A, B)$$

Of course, for $r = n$ we set $B = -infinity$, and note that $2 \leq r \leq n$.

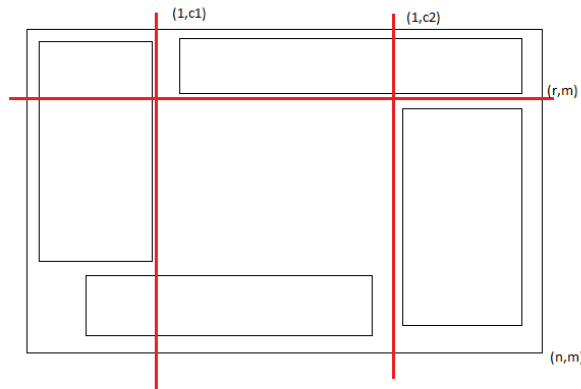


Figure 8

We are almost there. Now again, fix two columns and a row for the top rectangle, as shown in Figure 8, and make a use of previously calculated values to calculate best profit. In pseudocode:

```

res1 = -infinity
For c1 -> 2 to m - 1:
  For c2 -> c1 to m - 1:
    For r -> 1 to n - 1:
      A = dp[1][c1+1][r][m] + dp[r+1][c2+1][n][m]
      B = f[c1][c2][r+1]
      res1 = max(res, A + B)

```

The running time of this part of algorithm is $O(n^3)$ and the whole solution has time complexity of $O(n^3)$.

One last thing to worry about in this problem is the memory limit. Memory complexity in this solution is $O(n^4)$, which is too high. But note that we are using $O(n^4)$ memory only in dp matrix, and we know that at least one of the parameters of $r1, c1, r2, c2$ of $dp[r1][c1][r2][c2][left]$ has some nice value. Either $r1 = 1$ or $c1 = 1$ or $r2 = n$ or $c2 = m$, so we can make a use of it, by representing all the states where $r1 = 1$ by $dp[1][c1][r2][c2][left]$, all the states where $c1 = 1$ by $dp[2][r1][r2][c2][left]$, all the states where $r2 = n$ by $dp[3][r1][c1][c2][left]$ and finally all the states where $c2 = m$ by $dp[4][r1][c1][r2][left]$. Note that in this way we decreased our memory complexity to $O(n^3)$, which is sufficient for this problem and completes the solution.

Solution by:

Name: **Boris Grubić**

School: *Gimnazija Jovan Jovanovic Zmaj*

E-mail: *borisgrubic@gmail.com*

Problem R3 02: Lost in Madrid (code: LIM)

Time Limit: 10 second

Programming contests can be very exhausting. After five hours of intensive programming, you want to get some well-deserved rest and make yourself on the way to your hotel. Unfortunately, you don't quite remember the way to get there... but that doesn't matter: In good spirits (due to a successful contest?) you set out.

As you don't know the exact way, you decide to walk around in the following fashion: Start at the contest site (denoted by id 0) and choose a street at random. Follow the street to the next intersection, and choose another street at random. Every street at an intersection has the same probability of being chosen. You might even decide to take the street back where you came from. As you're on foot, you can use the streets in both directions, unlike in "Madrid's One Way Streets".

Your walk stops once you encounter your hotel (id = 300) or one of the tourist information booths (id > 290) where you can ask for the way. You can assume there is at least one path connecting you to either type of object.

Because you don't speak a lot of spanish (apart from some verbs that you can conjugate thanks to problem "Spanish Verbs"), you'd like to know the probability that you arrive at your hotel directly, without first arriving at a tourist information booth.

Input

The input consists of several testcases, separated by an empty line.

Each testcase starts with S , the number of streets. The following S lines contain two numbers $0 \leq A, B \leq 300$ each. This means that there is a street connecting intersection A to intersection B . The same street will not appear multiple times in the input.

The input ends with $S=0$. This testcase should not be processed.

Output

For each testcase, print the probability to arrive directly at the hotel, rounded to three decimal places.

Sample

input	output
3	0.333
0 291	1.000
0 292	0.000
0 300	0.579
2	
0 300	
291 300	
2	
0 291	
291 300	
7	
0 292	
0 88	
0 14	
0 300	
292 88	
88 300	
14 300	
0	

Solution:

The problem seems to be quite difficult when you look at it for the first time. Nearly instantly, simulation comes into mind. If we begin at one position of the graph and choose our way according to the probabilities of the graph randomly the algorithm should work for a small graph. But there are some problems, since we cannot be sure how often we have to run the algorithm to get good results, especially, because we do not know very much about the structure of the graph.

So we have to look at the problem differently. Let's call the k -th node k and the probability that one arrives at the hostel directly from the k -th node $p(k)$. If the node we look at is the hostel the probability is one and if it is a tourist information it is zero. For the other nodes, we look at the neighbors. According to the problem statement, each way has the same probability of being chosen. That is why we can use the following simple formula:

$$p(k) = \sum_{(k,k_2) \in \text{outEdges}(k)} \frac{p(k_2)}{|\text{outEdges}(k)|}$$

For each node we get one linear equation. All in all we have n equations with at most n unknown values. These equations are always solvable, since it is guaranteed that we can reach either a tourist information or the hostel from our start position. If we have solved all the equations we know the probability of our start position.

Figure 1 shows a small example. The graph is taken from the last test case in the problem statement. We have the following equations which we need to solve.

$$p(0) = \frac{1}{4}(p(14) + p(88) + p(292) + p(300))$$

$$p(14) = \frac{1}{2}(p(0) + p(300))$$

$$p(88) = \frac{1}{3}(p(0) + p(292) + p(300))$$

Qualifications

$$p(292) = 0$$

$$p(300) = 1$$

If we solve these equations, we get the required result. The time complexity of the solution is $O(n^3)$ if we use the standard Gauss algorithm, which one learns at school.

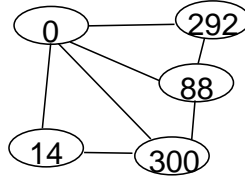


Figure 1: example graph

Solution by:

Name: Patrick Klitzke

School: Saarland University

E-mail: philologos14@googlemail.com

Problem R3 03: Circles (code: CIRCLES)

Resource: POI III, stage 3; Special thanks to Lei Huang

Time Limit: 30 second

Little Gary plays the following video game. Circles pop up on the screen and disappear from it. When the screen flashes, Gary can draw a straight line on the screen and win as many points as there are circles intersected by the line. As a born-to-be-winner, Gary wants to maximize his score. Please, help him, and write a program that will determine the maximum number of points he can win each time the screen flashes.

Input

The first line of the input contains M ($1 \leq M \leq 1000$), the number of events during the game. The next M lines contain descriptions of the events, one per line. They can be in one of the following three formats:

1 x y r

representing a circle of radius r popping up with the position of its center at (x, y) in the plane

2 i

representing a circle i disappearing, where circle i is the ith circle that popped up since the beginning of the game; and

3

representing the screen flashing.

x, y, and r are real numbers with at most two decimals, $-106 < x, y, r < 106, r > 0$.

Notes: A line intersects a circle if it has at least two common points with it. At any time, no two Circles on the screen have a common point. At any time, there is no line that "touches" more than two circles (a line touches a circle if they have exactly one common point). At any time, there are no more than 100 circles on the screen. Each i determines a circle that is on the screen at the moment of removal. No circle is removed twice.

Output

Each time the screen flashes, write an integer to a separate line, which is the maximum number of circles Gary can intersect.

Sample

input	output
9	2
1 3.00 0.00 1.00	2
1 -2.00 0.00 1.00	3
3	2
1 2.00 3.00 1.50	
3	
1 2.00 -4.00 1.00	
3	
2 3	
3	

Solution:

Here we are presented with a rather interesting problem which is clearly geometric in nature. It doesn't seem easy at first glance, however its solution is relatively simple to obtain (although formally proving the mathematics behind it might be tough). Like most geometry-related problems, CIRCLES can be solved in a number of ways; the approach we will explain here seems intuitively the easiest to comprehend.

Let's first analyze what the problem gives us, and what we must calculate. We are required to support three kinds of operations: **adding/removing** a circle from the visible set, and answer **queries** asking for the **maximal amount** of currently visible circles which can be **intersected** by an arbitrarily drawn straight line.

Before going any further with the analysis, let's prove a theorem which will be the basis of our solution.

Theorem: For all possible placements of circles (with respect to the task's constraints), at least one of the lines which intersects the maximal amount of them is infinitesimally "close" to one of the common tangents for one of the pairs of circles.

Proof: Let p be an arbitrary line of the form $y = kx + n_0$ which intersects the maximal amount of circles, and let $f: \mathbb{R} \rightarrow \{ \top, \perp \}$ be a function such that $f(n)$ is true if $y = kx + n$ is a line which intersects the maximal amount of circles, and false otherwise. As we start off with $n = n_0$ and increase / decrease the parameter n , after some time there will certainly be a value of the parameter, n_1 for which $f(n_1) = \perp$. Since this function covers the entire real number line (which is continuous by definition), we can use **Dedekind's Axiom** (Axiom of Continuity) to deduce that there exists a specific real number t where this function changes its value. It is obvious that this real number marks the moment when $y = kx + t$ is a **tangent** to one of the circles. If we move the parameter infinitesimally in the other direction, we will once again have a line which intersects the maximal amount of circles. Hence, we have proven that one of the sought lines must be infinitesimally "close" to a tangent of one of the circles.

Let T be the point of tangency of the line $y = kx + t$. If we exclude the circle which this line is a tangent to (and we can easily re-include it by infinitesimally shifting this line), we still have a line which intersects the maximal amount of circles. Now, let's shift the point of tangency in one direction ("rotate" the tangent around the circle). Analogously as in the first paragraph (by introducing a new function and applying Dedekind's Axiom to it), we can deduce that at one particular moment this tangent will no longer intersect the maximal amount of circles. It is obvious that at this point, this line is tangent to another circle, i.e. it is a **common tangent of two circles**. If we were to infinitesimally shift/rotate (depending on whether the common tangent is internal or external) this line in the proper direction, we would again obtain a line which intersects the maximal amount of circles. This concludes the proof.

This theorem is of great importance, as by proving it we have greatly reduced the amount of lines we need to consider: when we check for intersections, we will only check the lines which are **common tangents** to each pair of circles. Since no pair of circles has common points, each pair will have **four** common tangents: two **internal** and two **external** ones. Now we have to come up with a correct way of getting those tangents.

Let A and B be the centers of the circles, r_1 and r_2 their respective radii, d the distance between the centers, C and D the points of tangency for one of the common tangents, and \vec{n} the unit vector perpendicular to that tangent.

In this situation we have the following system of equations:

$$\begin{aligned} \vec{n} \cdot \vec{n} &= 1 \text{ (}\vec{n} \text{ is a unit vector)} \\ \vec{C} &= \vec{A} + r_1 \cdot \vec{n} \text{ (radius at point of tangency is parallel to } \vec{n}\text{)} \\ \vec{D} &= \vec{B} \pm r_2 \cdot \vec{n} \text{ (same as prev.)} \\ \vec{n} \cdot \overrightarrow{CD} &= 0 \text{ (orthogonality)} \end{aligned}$$

Deducing from this system:

$$\begin{aligned} \vec{n} \cdot \overrightarrow{CD} &= \vec{n} \cdot (\overrightarrow{AB} \pm r_2 \cdot \vec{n} - r_1 \cdot \vec{n}) = \overrightarrow{AB} \cdot \vec{n} - (r_1 \mp r_2) = 0 \\ \Leftrightarrow \overrightarrow{AB} \cdot \vec{n} &= (r_1 \mp r_2) \\ \Leftrightarrow \vec{v} \cdot \vec{n} &= \frac{(r_1 \mp r_2)}{d} \text{ (}\vec{v} = \frac{\overrightarrow{AB}}{|\overrightarrow{AB}|}\text{)} \end{aligned}$$

We have obtained a linear equation where the unknown is the vector \vec{n} . Once we obtain \vec{n} it is easy to obtain the desired **points of tangency**:

$$\begin{aligned} C_x &= A_x + r_1 \cdot n_x \\ C_y &= A_y + r_1 \cdot n_y \\ D_x &= B_x \pm r_2 \cdot n_x \\ D_y &= B_y \pm r_2 \cdot n_y \end{aligned}$$

When we have the **points of tangency** for each of the four possible tangents, it's easy to reformulate each tangent in the form $y = kx + n$, which we can use to handle the lines more easily later on.

A new question arises now: how can we correctly check **whether a line intersects a circle**? There are many ways in which this can be done; here we will present a trigonometric approach. What we do essentially is calculate the **distance between the center and the line**, and compare it to the radius. If it is lesser, then the line **intersects** the circle. To calculate the distance we will construct a **right triangle** using the distance we are looking for as one of the catheti, and the **vertical distance** (following the Y axis) between the center and the line as the hypotenuse.

The vertical distance can be calculated easily; if we were to draw a line parallel to the given one which intersects the center, this distance is the change in the **y-intercept** parameter (since the slope doesn't change):

$$\begin{aligned} k \cdot A_x + m &= A_y \\ m &= A_y - k \cdot A_x \\ m - n &= A_y - k \cdot A_x - n \end{aligned}$$

And now we involve some trigonometry to calculate the final distance:

$$\begin{aligned} d &= (m - n) \cdot \sin \alpha = (m - n) \cdot \cos \beta \text{ (complementary angles)} \\ \cos \beta &= \frac{1}{\sqrt{1 + \tan^2 \beta}} \\ \tan \beta &= k \text{ (definition of slope)} \\ \Leftrightarrow d &= \frac{(A_y - k \cdot A_x - n)}{\sqrt{1 + k^2}} \end{aligned}$$

To avoid precision errors with the square root and possible sign issues, it is wise to calculate the **square** of this distance instead, and compare it with the square of the radius. When we get the final amount of

intersections, we should return the result **plus 2**, because as described in the theorem, we can infinitesimally shift or rotate the tangent to include the two circles it is common to, without excluding any other circle.

Now the only remaining issue is how to store the circles efficiently. A good structure for this in C++ is a **map**, since we can use it to easily assign IDs to circles with insertion and removal times of $O(\log n)$. To avoid calculating tangents each time, we can store their parameters (k and n) in a **four-dimensional array** of size $1000 \cdot 1000 \cdot 4 \cdot 2$, where $param[x][y][z][f]$ will store the f^{th} (1^{st} is k , 2^{nd} is n) parameter of the z^{th} tangent between circles with IDs x and y . This narrowly fits our memory limits, so it is a plausible optimization.

Our algorithm is now easy to construct: when we receive an insertion command we insert a new circle in the map and store common tangents between this circle and all other currently visible ones. When we receive a deletion command we simply erase the given ID from the map, and when we are given a query we calculate the amount of intersected circles for each common tangent of pairs of visible circles, and output the maximum. An obvious optimization would simply output the previously calculated solution immediately if we receive two query commands in a row.

The time complexity of extracting common tangents is $O(1)$, and the time complexity of checking the amount of intersections for a single line is $O(n)$. This means that insertion commands are processed in $O(n)$ time, deletion commands in $O(\log n)$ time, and queries in $O(n^3)$ time. Hence, the overall asymptotic time complexity of our algorithm is $O(q \cdot n^3)$, where q is the amount of queries. With the given constraints, this solves the problem well ahead of the time limit. The memory complexity is $O(n^2)$ if we opt to store all the previously calculated tangents, and $O(n)$ otherwise.

Solution by:

Name: **Petar Veličković, Teodor Von Burg**

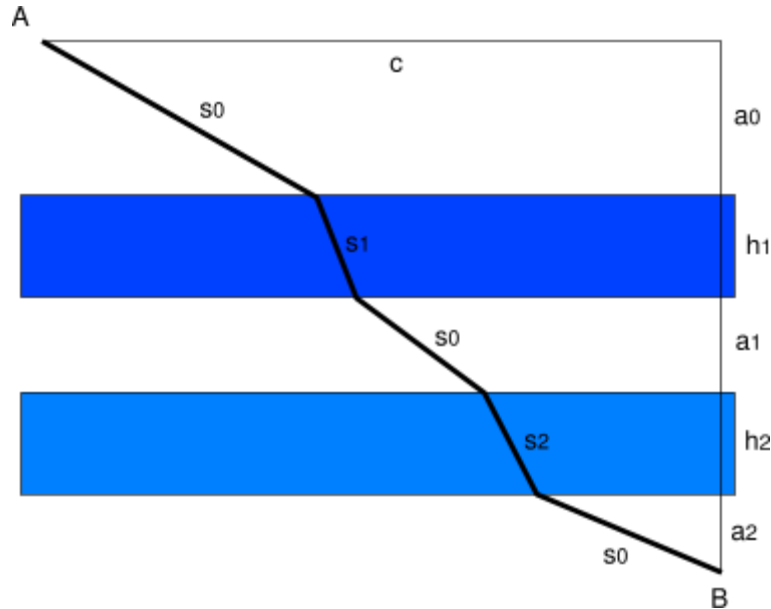
School: *Matematička Gimnazija*

E-mail: *petrov.velicovic@gmail.com, teodor.vonburg@gmail.com*

Problem R3 04: Bridges! More bridges! (code: BRII)

Time Limit: 7 second

Problem BRIDGE has shown that you are able to build the cheap bridge through the river very quickly. Now you will not have problems with time limit. You will have problems with number of bridges.

**Input**

There is a single positive integer T on the first line of input. It stands for the number of test cases to follow. Each test case is exactly five lines, containing description of the route between two cities A and B , located on opposite sides of the rivers.

```

n
a0 a1 a2 ... an
h1 h2 ... hn
c
s0 s1 s2 ... sn

```

Here n is the number of the rivers which are parallel to each other, a_i - the distances between rivers or between rivers and cities, h_i - the widths of the rivers, c - the distance between A and B along the axis parallel to the river, s_i - the costs of the unit of the bridge through i th river and s_0 - the cost of the unit of the road. Example for $n=2$ you can see on the picture.

All integers in input are positive and less than 50, except c - it is less than 2000.

Output

For each test case your program should write a single number to the standard output, equal to the minimal total cost of the route between A and B , accurate up to two digits after the decimal dot.

Sample

input	output
1 2 1 1 1 1 1 1 1 1 1	5.10

Solution:

Let's first denote things differently than in the problem statement in order to simplify the expressions which come after. Let $s_0 \dots s_{2n}$ stand for costs of building bridges or roads in tracks starting from the upper one and let $a_0 \dots a_{2n}$ be track widths. Imagine now we've built bridges and roads in each track. Let $x_0 \dots x_{2n}$ denote their projections on horizontal axis. We can observe that $x_0 \dots x_{2n} \geq 0$ and that there will never exist a 'zigzag' structure while building the roads (meaning we'll never drive back with respect to the horizontal axis). It is obvious that $x_0 + x_1 + \dots + x_{2n} = c$.

Now we can state the problem formally. We want to minimize cost function

$$f(x_0, x_1, \dots, x_{2n}) = s_0 \cdot \sqrt{(x_0^2 + a_0^2)} + \dots + s_{2n} \cdot \sqrt{(x_{2n}^2 + a_{2n}^2)}$$

given the constraint

$$x_0 + x_1 + \dots + x_{2n} = c.$$

This problem can be solved by using **Lagrange multipliers** [1][2]. Let's define Lagrangian function G as

$$G(x_0, x_1, \dots, x_{2n}, L) = f(x_0, \dots, x_{2n}) + L \cdot (x_0 + x_1 + \dots + x_{2n} - c)$$

In a point of optimal solution $\text{grad}(G) = 0$ has to hold.

So let's look at the partial derivative of G with respect to x_i :

$$\frac{\partial G}{\partial x_i} = L + s_i \cdot \frac{x_i}{\sqrt{x_i^2 + a_i^2}}$$

This has to be zero in a point of optimum so:

$$L + s_i \cdot \frac{x_i}{\sqrt{x_i^2 + a_i^2}} = 0$$

Let's use the above observation to express x_i :

$$x_i = L \cdot \frac{a_i}{\sqrt{s_i^2 - L^2}}$$

Note that, in order for x_i to be real, $L \leq s_i$ must hold for all i . One more important fact is that x_i is a monotonic function of L (if L increases the numerator increases and the denominator decreases, so it follows that x_i increases, and the other way around).

Let's plug in expressions for x_i into $x_0 + x_1 + \dots + x_{2n} = x_{0(L)} + x_{1(L)} + \dots + x_{2n(L)} = c$.

Now we have a sum of monotonic functions which has to be equal to a constant c . We can simply binary search L and find when the equality holds. From this point on it is trivial to reconstruct the solution - when we found L just calculate all x_i and then the total cost using the expressions given above.

References:

1. <http://www.slimy.com/~steuard/teaching/tutorials/Lagrange.html>
 2. http://en.wikipedia.org/wiki/Lagrange_multiplier
-

Solution by:

Name: *Filip Pavetić*

School: *FER Zagreb*

E-mail: *fpavetic@gmail.com*

Problem R3 05: Polynomial $f(x)$ to Polynomial $h(x)$ (code: POLTOPOL)

Resource: Tjandra Satria Gunawan

Time Limit: 20 second

Given polynomial of degree d , $f(x) = c_0 + c_1x + \dots + c_dx^d$. For each polynomial $f(x)$ there exists polynomial $g(x)$ such that:

- $f(x) = g(x) - g(x - 1)$ for each integer x
- $g(0) = 0$

Your task is to calculate polynomial $h(x) = \frac{g(x)}{x}$.

Note : degree of polynomial $h(x)$ = degree of polynomial $f(x)$.

Input

The first line of input contain an integer T , T is number of test cases ($0 < T \leq 10^4$). Each test case consist of 2 lines:

- First line of the test case contain an integer d , d is degree of polynomial $f(x)$ ($0 \leq d \leq 18$)
- Next line contains $d + 1$ integers c_0, \dots, c_d separated by space, represent the coefficient of polynomial $f(x)$ ($-231 < c_0, c_1, \dots, c_d < 231$ and $c_d \neq 0$)

Output

For each test case, output the coefficient of polynomial $h(x)$ separated by space. Each coefficient of polynomial $h(x)$ is guaranteed to be an integer.

Sample

input	output
5	13
0	0 1
13	1 1
1	1 2 3
-1 2	31 41 59 26
1	
0 2	
2	
2 -5 9	
3	
23 9 21 104	

Solution:

Since $g(0) = 0$ we have that the free member of the polynomial $g(x)$ is 0, so $h(x) = \frac{g(x)}{x}$ is a polynomial, and it is obvious that their degrees are equal: $deg(h(x)) = deg(f(x))$.

Let us define coefficients of the polynomial $g(x)$ respectively with a_1, a_2, \dots, a_{d+1} ($a_0 = 0$). Since $f(x) = g(x) - g(x - 1)$ for every integer x we have:

$$c_0 + c_1x + \dots + c_dx^d = a_{d+1}x^{d+1} + a_dx^d + \dots + a_1x - a_{d+1}(x-1)^{d+1} - a_d(x-1)^d - \dots - a_1(x-1).$$

Using the binomial formula

$$(x+a)^n = \sum_{k=0}^n \binom{n}{k} x^k a^{n-k}$$

for $a = -1$, we get that :

$$a_{d+1}x^{d+1} + a_dx^d + \dots + a_1x - a_{d+1}(x-1)^{d+1} - a_d(x-1)^d - \dots - a_1(x-1) = a_{d+1}x^{d+1} + a_dx^d + \dots + a_1x - a_{d+1} \left(\binom{d+1}{0}x^{d+1} - \binom{d+1}{1}x^d + \dots + (-1)^{d+1} \right) - a_d \left(\binom{d}{0}x^d - \binom{d}{1}x^{d-1} + \dots + (-1)^d \right) - \dots - a_1(x-1)$$

for every integer x .

Polynomials $f(x)$ and $g(x) - g(x-1)$ must be equal for every integer x , so their coefficients must be the equal, therefore it is necessary to hold:

$$\begin{aligned} c_d &= a_{d+1} \binom{d+1}{1} \\ c_{d-1} &= a_{d+1} \binom{d+1}{2} + a_d \binom{d}{1} \\ &\dots \\ c_i &= (-1)^{d-i} a_{d+1} \binom{d+1}{d-i+1} + (-1)^{d-i-1} a_d \binom{d}{d-i} + \dots + a_{i+1} \binom{i+1}{1} \\ (1) \quad &\dots \\ c_i &= a_{d+1} (-1)^{d-i} \binom{d+1}{d-i+1} + a_d (-1)^{d-i-1} \binom{d}{d-i} + \dots + a_{i+1} \binom{i+1}{1} \end{aligned}$$

From the last formula we can easily see that if we know coefficients $a_{d+1}, a_d, \dots, a_{i+1}$, we can easily calculate coefficient a_i (it is guaranteed that every coefficient is an integer, so $c_i - a_{d+1}(-1)^{d-i} \binom{d+1}{d-i+1} + a_d(-1)^{d-i-1} \binom{d}{d-i} + \dots + a_{i+1} \binom{i+1}{1}$) must be divisible by $i+1$).

From here it is easy to find the polynomial $h(x)$.

Since $|c_i| < 2^{31}$, from formulas (1) we can prove $|a_i| < 2^{63}$.

Complexity:

Given the total number of test cases $T \leq 10^4$ and $d \leq 18$, binomial coefficients must be determined before calculating the coefficients a_i . Then the overall complexity is $O(Td^2)$.

Solution by:

Name: **Predrag Milošević**
 School: Gymnasium "Svetozar Marković", Niš
 E-mail: predrag_93@live.com

Problem R3 06: Factorial challenge (code: FUNFACT)

Time Limit: 15 second

ing: Stir, let's go out and play our favorite game.

Stir: I am already having fun with my first factorial program.

Ling: Than I will give you a challenge on factorials. If you fail in it, you will have to come.

Stir: ok..

Ling gives Stir a number x and the challenge is to find the largest value of n such that $n!$ is not greater than the largest value that can be formed by x digits. Stir is stuck with the problem and needs your help. Now, it's your turn to make sure that Stir can continue having fun with factorials.

Input

The first line of the input contains a number t (about 10^5), the number of the test cases. The next t lines contain a number x ($1 \leq x \leq 10^9$).

Output

Output a total of t lines with each line containing the value n corresponding to the input case.

Sample

input	output
2	3
1	10
7	

Solution:

This problem turned out to be one of the easiest in special round. It is obvious that any iterative calculation of factorial will time out. We can use **Stirling's formula** (approximation) to calculate the factorial faster. Stirling's formula is in fact the first approximation to the following series:

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{51849n^3} - \frac{571}{2488320n^4} + \dots\right)$$

Once we can calculate $n!$, it is easy to calculate number of digits of $n!$ as $\log_{10} n!$, so $\log_{10} n! \sim \left(n \ln n - n + \frac{1}{2} \ln 2\pi n + \frac{1}{12n}\right) / \ln 10$. The last observation needed is that $\log_{10} n!$ is a monotonically increasing function, so we can use binary search to speed up the solution. A properly implemented solution should work in time, but there are still some possible mistakes. Pay attention to how you are rounding. I prefer rounding up. And pay attention to the base case of the binary search. There can be several numbers such that number of digits of their factorials is the same for those distinct numbers. This can cause WA.

Solution by:

Name: **Danilo Vunjak**

School: Faculty of technical science

E-mail: kingarthurie@gmail.com

Problem R3 07: Hi6 (code: HISIX)

Resource: Daniel Ampuero

Time Limit: 7 - 15 second

"I read somewhere that everybody on this planet is separated by only six other people. Six degrees of separation between us and everyone else on this planet. The President of the United States, a gondolier in Venice, just II in the names. I find it A) extremely comforting that we're so close, and B) like Chinese water torture that we're so close because you have to find the right six people to make the right connection... I am bound to everyone on this planet by a trail of six people." - Ouisa Kitteridge, "Six Degrees of Separation"

Is widely know that one is separated from everyone in the world in no more than 6 degrees of separation. A degree of separation is defined by the minimum numbers of connections you need to make to contact someone else. For instance, if you know personally another person, then you are separated by one degree. If you know somebody through some friend but not directly (a friend of a friend), then you are separated by two degrees, and so on.

Nevertheless, young Kevin Smith is not convinced about this theory and wants to probe it false. To achieve this, he has hacked the Hi6! social network and requested you to help him knock down the theory of six degrees of separation.

Input

The first line contains an integer T, which specifies the number of test cases. Then, T test case descriptions will follow. Each test case will start with a line with one positive integer, N meaning the number of connections. The next N lines will contain the following pattern:

<name_1> <name_2> <D>

meaning that person "<name_1>" is connected with the person "<name_2>" by making D connections and viceversa. Note that both persons can know each other by a lower degree of separation using other connections.

Output

For each input case you must print the string "Case #i: ", where i is the test case number, starting from 1, following by the maximum degree of separation between the specified people. If there is someone that cannot connect to another person, print "INFINITE" instead.

Constraints

- All names will be non-empty strings composed only by lowercase characters.
- All names will have between 1 and 10 characters, inclusive.
- "<name_1>" will be different than "<name_2>" for all connections.
- There will be no pair of connections between the same pair of persons.
- D will be an integer between 1 and 1000, inclusive, for all connections.
- T will be between 1 and 100, inclusive.
- N will be between 1 and 10^5 , inclusive.

Sample

input	output
3	Case #1: INFINITE
2	Case #2: 4
john judy 1 mary peter 1	Case #3: 3
3	
john judy 7 john peter 2 judy peter 2	
7	
john judy 3 katie peter 4 john peter 2 judy mary 1 peter mary 2 john katie 1 katie mary 1	

Solution:

This problem is a very fine example of how a seemingly “straightforward” algorithm fails to perform in time without some necessary optimization. It did prove to be one of the harder tasks of the round, even though the ideas behind it are all relatively simple and easy to implement.

First of all let's analyze the problem and reformulate it. The introductory part is backstory and not relevant to the solution. From the input explanation one can easily conclude that what we are given in the problem's input is an **undirected weighted graph**, where each node corresponds to a person, each edge represents a connection between two people, and its weight is the connection level between them. The output explanation tells us what we are required to do. The task at hand is to find the lengths of the **shortest paths** (smallest “degree of separation”) for each pair of people, and output the **maximal** length out of those (or 'INFINITE' if there exists a disconnected pair).

Before getting to the main algorithm, let's first discuss the way we can easily (and effectively) transform the input data into a graph. It is clearly inconvenient for the nodes to be stored as strings. So the first thing we want to do is to transform each string into an integer; to do this we use **hashing**. The constraints on the string lengths and the characters it can contain in this problem make hashing an ideal option, because we can make a **perfect hashing** function (which is a one-to-one function – two different strings can't map into the same integer). There are at most 10 characters in each name, and they only consist of lowercase characters of the English alphabet. By observing each character as a number from 1 – 26 (where 'a' corresponds to 1, 'b' to 2, etc...) we can observe each string as an integer in base 27. Hence, the largest mapped integer value we can get, knowing the constraints (if we were to map the string 'zzzzzzzzz'), is $27^{10} - 1$. This easily fits in a single 64-bit integer variable, so we have no need for any kind of modular arithmetic here. It is obvious why the hashing function described here is perfect.

There are numerous ways in which we can use the obtained integers to assign an ID (ranging from 1 to N, where N is the number of nodes) to each node in the graph. One of the ways is to store all the integers obtained so far in a **binary search tree**, while storing the ID within each node of the tree as well as the integer that represents it. For each new edge we are given, we take the mapped values for the names involved in that edge and try to insert them in the tree; if they are already inserted then we know which ID corresponds to them, and if not then they are assigned the first unused ID (for this we can use a variable which starts off at 1 and increases every time a new leaf is added to the tree). This code can be typed very quickly in C++ using the **map** structure, which is essentially a **RB-Tree** (balanced binary search tree) and

easily maps one value to another. The time complexity of the hashing is linear on the length of the string (essentially constant with this task's constraints), and the time complexity of each insertion in the tree is $O(\log |V|)$. This is done for each edge, so the overall complexity of the input processing is $O(|E| \cdot \log |V|)$, where $|E|$ is the amount of edges in the graph.

Now that we have a “normalized” graph, we can start discussing the solution to the problem. Let's first get the case when the answer is 'INFINITE' out of the way; this is the answer iff the graph is **not connected**. Connectivity can easily be checked in $O(|V|)$ time complexity using a **depth-first search** algorithm - a graph is connected iff a depth-first search visits all the nodes in a single execution. Once we are sure that our graph is connected, we can carry on. There are two algorithms that come to mind for solving this problem – **Floyd-Warshall** algorithm and **Dijkstra's** algorithm with a **binary heap-implemented priority queue**. Both of these algorithms, in their naïve form, will likely fail to solve the problem in time. Here we will mention a key optimization to Dijkstra's algorithm which makes it successfully process the given test data a few seconds before the time limit.

Dijkstra's algorithm, unlike Floyd-Warshall, is a **single-source shortest path** algorithm, meaning that in one execution it's going to find the shortest paths from one node to all others. It is clear that for this problem we need to execute Dijkstra's algorithm $N - 1$ times, to get all the possible shortest path lengths. However, with each next execution we have more information than before, which we can use to lessen the amount of operations the algorithm will do. When it is executed for the first time, Dijkstra's algorithm will initially only put one node in the priority queue (the first source we picked), while setting the distances associated with all others to infinity. When we execute Dijkstra's algorithm for the K -th time, we can use the shortest path lengths from nodes $\{1, 2, \dots, K-1\}$ to node K as their initial distance (since the shortest path from node A to node B in an undirected graph has the same length as the shortest path from B to A), and immediately insert them in the priority queue and mark them as visited. This will avoid repeating any unnecessary calculations we have done before while calculating the mentioned path lengths, and is sufficient to pass the given time limit. It should be noted, however, that this is in no way the only (or the best, for that matter) optimization which solves this task in time. It is simply one of the most obvious ones and it is very easy to modify the already-made code for Dijkstra's algorithm to support it.

The asymptotic time complexity of a single execution of Dijkstra's algorithm is $O((|E| + |V|) \cdot \log |V|)$, and we have to execute the algorithm once for each node in the graph, which gives the total complexity of $O(|V| \cdot (|E| + |V|) \cdot \log |V|)$, which dominates the time required for processing the input and checking for connectivity, so this is the overall asymptotic time complexity of the solution. The memory complexity (for storing the previously calculated shortest path lengths) is $O(|V|^2)$.

Solution by:

*Name: **Petar Veličković***

*School: **Matematička Gimnazija***

*E-mail: **petrov.velickovic@gmail.com***

Problem R3 08: Frequent values (code: FREQUENT)

Resource: University of Ulm Local Contest 2007

Time Limit: 5 second

You are given a sequence of n integers a_1, a_2, \dots, a_n in non-decreasing order. In addition to that, you are given several queries consisting of indices i and j ($1 \leq i \leq j \leq n$). For each query, determine the most frequent value among the integers a_i, \dots, a_j .

Input

The input consists of several test cases. Each test case starts with a line containing two integers n and q ($1 \leq n, q \leq 100000$). The next line contains n integers a_1, \dots, a_n ($-100000 \leq a_i \leq 100000$, for each $i \in \{1, \dots, n\}$) separated by spaces. You can assume that for each $i \in \{1, \dots, n-1\}$: $a_i \leq a_{i+1}$. The following q lines contain one query each, consisting of two integers i and j ($1 \leq i \leq j \leq n$), which indicate the boundary indices for the query.

The last test case is followed by a line containing a single 0.

Output

For each query, print one line with one integer: The number of occurrences of the most frequent value within the given range.

Sample

input	output
10 3	1
-1 -1 1 1 1 1 3 10 10 10	4
2 3	3
1 10	
5 10	
0	

Solution:

It's important to notice that the array is in order, so equal values are in a group. Assume we are given "from i to j " query. Let $P_{i,j}$ be the number of appearances of the most frequent number in the given interval. We split the interval into two parts, $[i, k]$ and $[k + 1, j]$. It's easy to notice that:

- If $a_k \neq a_{k+1}$ then $P_{i,j} = \max(P_{i,k}, P_{k+1,j})$
- If $a_k = a_{k+1}$ then $P_{i,j} = \max(P_{i,k}, P_{k+1,j}, \text{number of appearances of } a_k \text{ in } [i, k] + \text{number of appearances of } a_{k+1} \text{ in } [k + 1, j])$

Answering the queries can be done using a data structure known as the **segment tree**. Every node of the tree will contain the following:

- Number of appearances of the most frequent number in the nodes interval (cnt)
- Value and number of appearances of the left end number ($leftValue, leftCnt$)

- Value and number of appearances of the right end number ($rightValue, rightCnt$)

Also, by $leftValue[i, j]$ we denote $leftValue$ of the node in the segment tree which contains the interval $[i, j]$. The same goes for other values of nodes ($rightValue, leftCnt, \dots$)

Initialization of the tree: For every node which contains the interval $[i, i]$ it's clear that $cnt = leftCnt = rightCnt = 1, leftValue = rightValue = a_i$. If we had initialized nodes which contain intervals $[i, k]$ and $[k + 1, j]$, we can initialize a node which contains the interval $[i, j]$. It's obvious that $leftValue[i, j] = leftValue[i, k]$ and $rightValue[i, j] = rightValue[k + 1, j]$. If $leftValue[i, k] = leftValue[k + 1, j]$ ($a_i = a_{k+1}$), it's clear that all the values from i to $k + 1$ are equal (since the array is in order), hence $leftCnt[i, j] = leftCnt[i, k] + leftCnt[k + 1, j]$. If not, $leftCnt[i, j]$ is equal to $leftCnt[i, k]$. The same holds for $rightCnt$. Finally, the number of appearances of the most frequent number:

- If $rightValue[i, k] \neq leftValue[k + 1, j]$ ($a_k \neq a_{k+1}$) we notice that intervals $[i, k]$ and $[k + 1, j]$ do not contain the same numbers, hence $cnt[i, j] = \max(cnt[i, k], cnt[k + 1, j])$.
- If not, $cnt[i, j] = \max(cnt[i, k], cnt[k + 1, j], rightCnt[i, k] + leftCnt[k + 1, j])$.

The only thing left now is answering the queries, which is easy once we have all the information from the segment tree. All we have to do is to split the interval until we get intervals for which we have information from the segment tree, and then we combine informations in the same way as we initialize the segment tree.

Let's summarize:

- Initialization of the segment tree takes $O(N \log N)$ time. Answering the queries by going down the tree and combining the data takes $O(\log N)$ time per query.

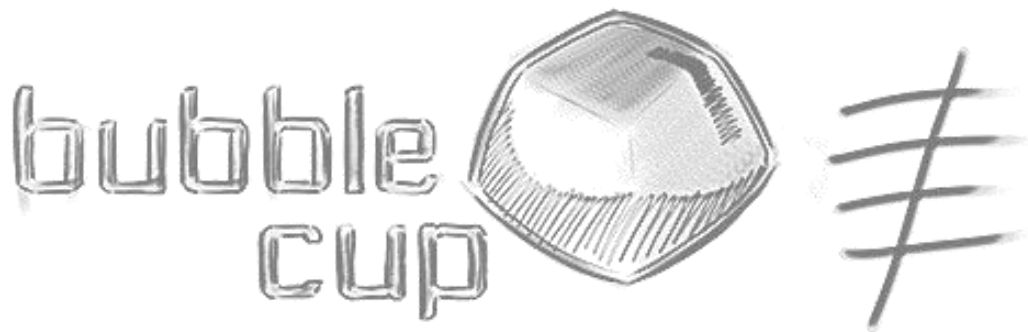
So the time complexity is $O(N \log N + Q \cdot \log N)$. Memory complexity is $O(N)$.

Solution by:

Name: Marko Baković

School: First gymnasium in Kragujevac

E-mail: markobakovic95@gmail.com



The scientific committee would like to thank everyone who did important behind-the-scenes work. We couldn't have done it without you.

For the next year, BubbleCup Crew have many ideas. A lot of changes will happen. Stay with us and see you next year. We'll be back...

Bubble Cup Crew



bubble
cup  

Student coding contest