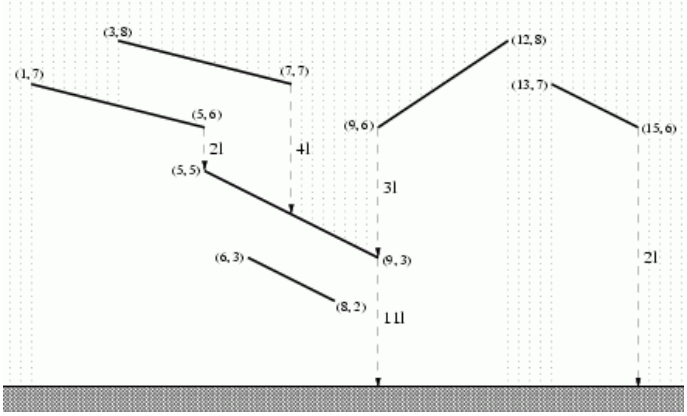
Problem R1 01: November Rain (code: RAIN1)

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 $x1, y1, x2, y2$ $(0 \leq x1, y1, x2, y2 \leq 1.000.000, x1 < x2, y1 \neq y2)$ separated by single spaces. Integers $x1, y1$ are respectively the horizontal position and the height of the left end of the segment. Integers $x2, y2$ 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 6 13 7 15 6 3 8 7 7 1 7 5 6 5 5 9 3 6 3 8 2 9 6 12 8	2 4 2 11 0 3

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

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 `inverse[A[i]] = i` for each `i`) and check if `A[i] == inverse[i]` for every `i` (`1 ≤ i ≤ N`). It's easy to prove that it's enough to check only that `A[A[i]] == i` for every `i` (`1 ≤ i ≤ N, A[i] ≤ 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

$r\ m\ v1\ c1\ v2\ c2\ v3\ c3\ \cdots\ vm\ cm$

where r is the number of faces required on the unspecified die, m is the number of sums of interest, $v1, \ldots, vsm$ are these sums, and $c1, \ldots, cm$ 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 $1 \ldots 50$, and values for the vi's and ci'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 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	Final die face values are 1 1 1 2 6 Impossible Final die face values are 3 7 9 9

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 $(s_1, s_2, \ldots, s_m)$

$M[i][j] += M[i-1][j - v[s_i]]$, where $v[s_i]$ is be the value of the i-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

$c[j] -= M[n][v[j] - i]$ (where i 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.

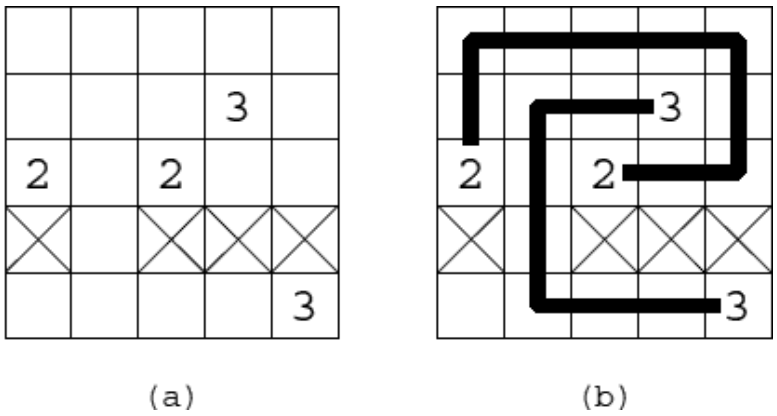


Figure 1: An example of setting and its solution

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 $rowi$ 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.

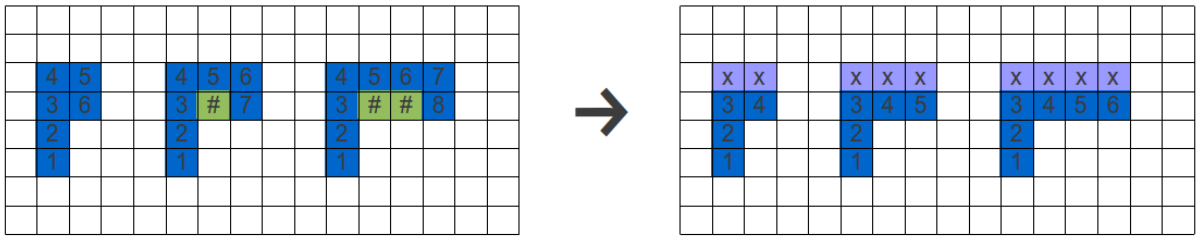
Sample

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

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):

- $WEIGHT\ m\ a1\ a2 \ldots am\ b1\ b2 \ldots bm$

  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), $a1\ a2 \ldots am$ - identifiers of spheres that you want John to put on the left scale and $b1\ b2 \ldots bm$ - 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 weighting 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 weighting 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 $\left\lfloor \frac{n}{3} \right\rfloor + 1$ (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 $\frac{3^{k-1}-1}{2} = 1 + 3 + \cdots + 3^{k-2}$ coins each. Now, we divide each group into $k-1$ subgroups $\{A_0, A_1, \ldots, A_{k-2}\}, \{B_0, B_1, \ldots, B_{k-2}\}, \{C_0, C_1, \ldots, C_{k-2}\}$; 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

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

Therefore, we can write $|A|=|B|=1+3+\cdots+3^{k-3}+X$, where $1\leq X\leq 3^{k-2}$. Now, just like in an edge case, we divide each group A,B,C into into $k-1$ subgroups $\{A_0,A_1,\ldots,A_{k-2}\},\{B_0,B_1,\ldots,B_{k-2}\},\{C_0,C_1,\ldots,C_{k-2}\}$; 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. Thats 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 2 100 4 7 8 2 3 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	FF BB IMPOSSIBLE

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:

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

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 $a1, a2, \ldots, aN$. In a segment flip, you can pick a contiguous segment $ai, a(i+1), \ldots, aj$ 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 $ai, \ldots, aj$ and $ak, \ldots, al$ 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 $a1, a2, \ldots, aN$.

Output

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

Constraints

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

Sample

input	output
3 1 -5 2 -3	6

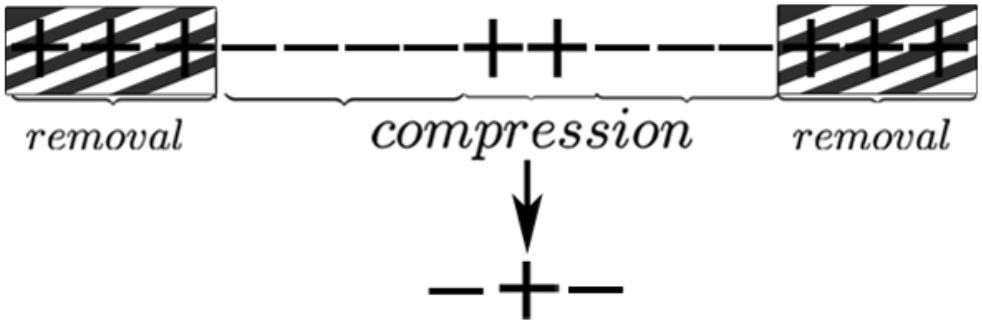
Solution:

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: petrovy.velickovic@gmail.com

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

Time Limit: 1.0 second

Given N buildings of height $h1, h2, h3 \ldots hn$, 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

$$h1 = h2 = h3 = .. = hn = k \text{ (} \text{ can be any number).}$$

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 $[h1, h2, h3 \ldots . hn]$ and the third line contains n integers $[c1, c2, c3 \ldots cn]$ which denotes the cost of adding or removing one unit of brick from the corresponding building.

$T \leq 15; n \leq 10000; 0 \leq Hi \leq 10000; 0 \leq Ci \leq 10000;$

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 $\boldsymbol{T}$, number of test cases. $\boldsymbol{2T}$ lines follow. First line gives you the number of stairs $\boldsymbol{N}$. 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 naïve 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(node, node, 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