UVA 10276 HANOI TOWER TROUBLES AGAIN!

What is the problem? It has nothing to do with the "Tower of Hanoi" problem.

Stack as many balls as possible onto *N* pegs. Balls must be processed in order: ball 1, then ball 2, ball 3, and so on. A ball can be stacked on top of another if and only if the sum of their numbers is a square. For example, ball 3 can be stacked on top of ball 1 since 3+1=4=2^2, but 3 cannot be stacked on top of 2 since 5 is not a square.

How to solve the problem? Let's try playing the game with four pegs.

To stack as many pegs as possible, we take a greedy approach. It is also a good idea to be systematic. Every time we try to stack a ball, we will first consider peg#1, then peg#2, peg#3, and so on. We will place the ball on the very first peg that satisfies the rule (when the sum is a square number). That's our strategy; let's begin.

		Peg1	Peg2	Peg3	Peg4
_ _ _ _		Initially, every peg is empty			
1 _ _ _	**ball 1**	✓			
1 2 _ _	**ball 2**	1+2=3 not square	✓		
3 / 1 2 _ _	**ball 3**	1+3=4=2^2 ✓			
3 / 1 2 4 _	**ball 4**	3+4=7 not square	2+4=6 not square	✓	
3 5 / 1 2 4 _	**ball 5**	3+5=8 not square	2+5=7 not square	4+5=9=3^2 ✓	
6 / 3 5 / 1 2 4 _	**ball 6**	3+6=9=3^2 ✓			
6 / 3 7 5 / 1 2 4 _	**ball 7**	6+7=13 not square	2+7=9=3^2 ✓		
6 / 3 7 5 / 1 2 4 8	**ball 8**	6+8=14 not square	7+8=15 not square	5+8=13 not square	✓
6 9 / 3 7 5 / 1 2 4 8	**ball 9**	6+9=15 not square	7+9=16=4^2 ✓		
10 / 6 9 / 3 7 5 / 1 2 4 8	**ball 10**	6+10=16=4^2 ✓			
10 / 6 9 11 / 3 7 5 / 1 2 4 8	**ball 11**	10+11=21 not square	9+11=20 not square	5+11=16 ✓	
10 / 6 9 11 / 3 7 5 / 1 2 4 8	**ball 12**	10+12=22 not square	9+12=21 not square	11+12=23 not square	8+12=20 not square

what have we done? what can we observe?

We solve the problem one ball one peg at a time; observed that every time we try to stack a ball onto a peg, we only look at the top-most ball. We never consider how many balls a peg has, or which balls are on the peg, etc. That means, at any time we only need to keep one piece of information for each peg – the ball on the top. We will use a one-dimensional array for this.

write the program exactly as how we have traced for a solution (above).

```
Declare an array of length n; a[i] = the ball on the top of the peg i
Initially, a[i]=0 for all i
Loop through the balls in order starting at b=1
  For each peg i from 1 through n+1
     If the sum of the top of peg i and the ball b is a square
        Then stack b on the top of peg i, set a[i] = b, and process the next ball
     Else if i==n+1 then return b-1 //We've reach the limit, there is no more peg
     Else stack b on peg i, set a[i] = b, and process the next ball //An empty peg
```

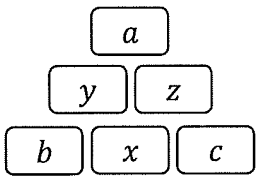
UVA 11040 ADD BRICKS IN THE WALL

What is the problem?

The wall has a fixed shape and size. It is triangular. It has 9 rows. Row i has exactly i bricks. We need to output the number on every brick. The rule is the number on a brick equals the sum of the two bricks below it.

How to solve the problem?

If we know every number on the bottommost row, then we can calculate all other numbers by a simple addition. The problem boils down to finding the missing numbers on row 9. Let's consider a smaller pyramid shown below. We know a, b, and c. We need to calculate the value of x.

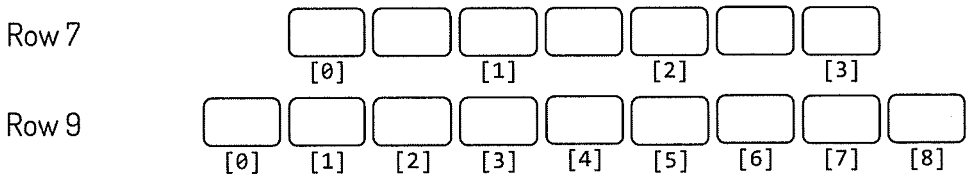


$$a = y + z \qquad a = (b + x) + (x + c)$$
$$y = b + x \quad \Rightarrow \quad a = b + 2x + c$$
$$z = x + c \qquad x = (a - b - c)/2$$

That is, we can use the numbers given in row 7 and row 9 to fill in every missing one in row 9. We can then work up one row at a time from the bottom to the top to fill in everything else. The numbers given in other rows are redundant; they are in fact not needed.

How to implement a program?

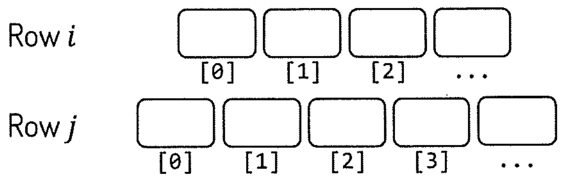
Step 1: calculate the missing numbers in row 9. Below, we draw row 7 and row 9 and label the bricks from left to right. In row 7, we are interested only on ones whose number is provided. In row 9, we wish to find the values on the odd bricks; even ones are given.



We write this in detail:

```
           Row9[1] = (Row7[ 0 ] - Row9[ 0 ] - Row9[ 2 ]) /2
           Row9[3] = (Row7[ 1 ] - Row9[ 2 ] - Row9[ 4 ]) /2
           Row9[5] = (Row7[ 2 ] - Row9[ 4 ] - Row9[ 6 ]) /2
           Row9[7] = (Row7[ 3 ] - Row9[ 6 ] - Row9[ 8 ]) /2
Therefore, Row9[k] = (Row7[k/2] - Row9[k-1] - Row9[k+1]) /2   for k=1,3,5,7
```

Step 2: use row 9 to complete row 8, then row 7, 6, ..., 2, and 1.



We have:

```
Row_i[k] = Row_j[k] + Row_j[k+1]
```

To get an output, keep a string as you fill the bricks and build it back to front. This follows from the fact that we fill the bricks bottom-up, but the output prints top down. It is also easier to fill each row right to left.

UVA 11953 BATTLESHIPS

How many ships are alive in a battlefield?

The width of the ship is exactly one cell, while its length varies. Ships are placed either vertically or horizontally. No two ships are overlapped or touched. A ship's cell may be alive or hit. A ship is sunk if all of its cells are hit, marked by '@'. A ship is alive if at least one if its cells is alive, marked by 'x'. A character '.' marks an empty cell.

Problem solving strategies...

Are we counting the number of 'x' cells? No! A single ship is likely to contain more than one alive cell. If we just count every 'x' cell then many ships will be counted more than once, giving us a wrong answer.

How about we locate connected 'x' cells and count each of these chunks as one alive ship? Still incorrect. Hit cells can be in the middle of any alive ship. In this case, two alive chunks belong to one same ship, not two.

So, how do we do this? We locate any 'x' cell, counting it as one ship being alive. Let's call this ship S. Next, we identify every cell, hit or alive, that belongs to S, We mark all these cells as processed so we do not consider them again. Then, we move on to another 'x' cell and repeat the steps until there is no more alive cells. Will this strategy work? Yes, it will. If you are not convinced, try work out a small battlefield by hands.

Develop some systematic procedures...

I We store the battlefield in a two-dimensional array of characters. The rows are indexed 0 to N-1; the top row is indexed zero. The columns are indexed 0 to N-1; the left column is indexed zero.

II To locate an 'x' cell, we look at one cell at a time from (0,0) to (N-1,N-1). We do row 1, row 2, ..., row N-1.

III To identify all cells belonging to S starting at 'x', we check the neighbors of 'x'. There are four neighbors, following the requirement that ships are arranged vertically or horizontally (there are no diagonals).

We start with the North, extending one cell at a time. If it is an 'x' or an '@' then we keep going (it is a part of a ship). Otherwise, we stop. The same process is repeated for the East, the South, and the West. Note: only at most two of these four directions will be extended. Can you reason why? Hint: the ship is one-cell wide.

IV To mark a cell as processed, we simply mark it as '.' or empty. This works because in **II** we only look for 'x' or alive cells; empty cells are ignored. A processed cell marked as empty will consequently be ignored.

Put it all together...

```
Read the battlefield into a 2D array of characters
For every row r, starting from row 0
   For every column c, starting from column 0
      If the cell (r,c) is alive, i.e. it is marked with an 'x'
         Mark the cell (r,c) empty with an '.'
         Increase the count of alive ships by one
         For every neighbors, in the order of North, East, South, and West
            Extend one cell at a time
               keep going as long as the cell is not empty and mark it empty
Output the count
```

UVA 10099 THE TOURIST GUIDE

What is the problem?

Find the minimum number of trips required for a guide, Mr.G, to guide T of his tourists from city S to T. There are N cities and R roads. Each road has a limit on how many passengers its bus service can carry at one time.

Why does the first test case require five trips?

By observation, the path from city#1 to #7 that can carry the maximum number of passenger is 1-2-4-7 and it can carry 25 passengers per trip (though 30 tourists can go from 1-to-2, only 25 can go from 2-to-4, and onto 4-to-7). There are 99 tourists. By simple mathematics, the number of trips Mr.G needs equals the upper-bound of 99 divides by 25, which is 4 trips. The answer, however, is 5.

What's going on? Did you forget the guide? Mr.G has to go with every trip; his tourists cannot traveling alone. One of the easiest way to include this condition into the map is to reduce the limit of every road by one. This deduction accounts for Mr.G's seat; the remaining (i.e., the new limit) is then exclusively for tourists. Hence, path 1-2-4-7 can only carries 24 tourists at once and 99 tourists require ceiling(99/24) = 5 trips.

Problem representation ... in a way that we can program it.

A map can handily be described as a graph where nodes represent cities and edges represent roads that connect any two cities. A graph in data structure is stored in a two-dimensional array, known as an adjacency matrix. The rows and the columns are the nodes. The cell at row=r and column=c, written (r, c), represents an edge between node r and node c. Here, we show a matrix for the first test case (with weights reduced for Mr.G's seat).

	1	2	3	4	5	6	7
1	▨	29	14	9			
2	29	▨		24	59		
3	14		▨	39		19	
4	9	24	39	▨			34
5		59			▨		19
6			19			▨	29
7				34	19	29	▨

Algorithmic thinking ... we will try to work out a few paths.

The adjacency matrix above shows the case when we use only direct connections. Hopping between cities would enable us to construct more paths. We begin by considering node 1. That is, we will find additional path by passing through node 1. For example, there is no direction connection from node 2 to node 3, but hopping through node 1 gives us a path 2-1-3. The diagram below illustrates how we work this out in our matrix.

Along column *c*: from which node is there a path to *c*, e.g. 2-*1*, 3-*1*, and 4-*1*

Along row *r*: to which node is there a path from *r*, e.g. *1*-2, *1*-3, and *1*-4

	1	2	3	4	5	6	7
1	▨	29	**14**	9			
2	**29**	▨	**14**	24	59		
3	14		▨	39		19	
4	9	24	39	▨			34
5		59			▨		19
6			19			▨	29
7				34	19	29	▨

Step 1: node 1 → column 1 → we can go from 2-to-1 with a capacity of 29

Step 2: node 1 → row 1 → we can go from 1-to-3 with a capacity of 14

Step 3: combine → we can go 2-1-3 with a capacity of min(29, 14) =14

Similarly, there are connections 2-1, and 1-4, giving us a new path 2-1-4 capacity 9. There is, however, already a connection 2-4 capacity 24. Since 24>9, the old connection is preferred. Thus, we do not update cell (2,4).

We can look at this problem as a "maximin". First, we take a minimum when we combine multiple connections (i.e. step 1-3). Then, we take a maximum when we compare different paths (i.e. the case of 2-1-4).

Doing it by hands, a complete solution one step at a time.

I Initial state: use only direct edges.

	1	**2**	**3**	**4**	**5**	**6**	**7**
1	▨	29	14	9			
2	29	▨		24	59		
3	14		▨	39		19	
4	9	24	39	▨			34
5		59			▨		19
6			19			▨	29
7				34	19	29	▨

II Adding node 1 (try hopping through node 1).

	1	**2**	**3**	**4**	**5**	**6**	**7**
1	▨	29	14	9			
2	29	▨	**14**	24	59		
3	14	**14**	▨	39		19	
4	9	24	39	▨			34
5		59			▨		19
6			19			▨	29
7				34	19	29	▨

III Adding node 2 (hopping pass either 1 or 2).

	1	**2**	**3**	**4**	**5**	**6**	**7**
1	▨	29	14	**24**	**29**		
2	29	▨	14	24	59		
3	14	14	▨	39	**14**	19	
4	**24**	24	39	▨	**24**		34
5	**29**	59	**14**	**24**	▨		19
6			19			▨	29
7				34	19	29	▨

IV Adding node 3.

	1	**2**	**3**	**4**	**5**	**6**	**7**
1	▨	29	14	24	29	**14**	
2	29	▨	14	24	59	**14**	
3	14	14	▨	39	14	19	
4	24	24	39	▨	24	**19**	34
5	29	59	14	24	▨	**14**	19
6	**14**	**14**	19	**19**	**14**	▨	29
7				34	19	29	▨

V Adding node 4.

	1	**2**	**3**	**4**	**5**	**6**	**7**
1	▨	29	**24**	24	29	**19**	**24**
2	29	▨	**24**	24	59	**19**	**24**
3	**24**	**24**	▨	39	**24**	19	**34**
4	24	24	39	▨	24	19	34
5	29	59	**24**	24	▨	**19**	**24**
6	**19**	**19**	19	19	**19**	▨	29
7	**24**	**24**	**34**	34	**24**	29	▨

VI No changes when adding 5 and 6. Adding 7.

	1	**2**	**3**	**4**	**5**	**6**	**7**
1	▨	29	24	24	29	**24**	24
2	29	▨	24	24	59	**24**	24
3	24	24	▨	39	24	**29**	34
4	24	24	39	▨	24	**29**	34
5	29	59	24	24	▨	**24**	24
6	**24**	**24**	**29**	**29**	**24**	▨	29
7	24	24	34	34	24	29	▨

At the end, the value in each cell (r, c) is the maximum number of tourists that a path from city r to city c can carries. Though the problem only concerns about a certain pair of cities, going from one starting city to one destination city, we have calculated for all-pairs. It seems as if we choose to do work than needed. The source code, however, is much simpler this way. This is a very fair trade-off for competitive programming.

Time to write a pseudocode.

```
Set up a matrix dp of size NxN
dp[i][j] = capacity of a link from i to j, 0 if there is no link from i to j
For each node k //hop node k
    For each node i //from node i
        For each node j //to node j
            If there is a connection i->k and k->j
                dp[i][j] = max(dp[i][j], min(dp[i][k], dp[k][j]))
Return dp[starting city][destination city]
```