## A. Aztec Diamond

To solve this problem, one can define recursive functions that makes a vertical block at given coordinate. Here we provide one way to do it. Refer to the below image for terminologies about directions and coordinates.


```
void foo(int r, int c)
{
    if(arr[r][c]=='D') return;
    assert(arr[r][c] == 'L');
    if(arr[r-1][c]=='L'){
        rotate(r-1, c);
    }
    else if(arr[r-1][c]=='D'){
        foo(r-1, c+1);
        rotate(r-2, c);
        rotate(r-1, c);
    }
    return;
}
```

Function 'foo' takes the coordinate, then makes it the lower part of a vertical block. Function 'rotate' is where the actual rotation takes place. It takes ( $\mathrm{r}, \mathrm{c}$ ), the lower left part of the $2 * 2$ square, and rotates the square. It raises an error if the given coordinate is not the upper left part of a $2 * 2$ square. arr is the $2 \mathrm{n} * 2 \mathrm{n}$ matrix containing the current state of the diamond.
To make the whole process work, the order of calling the 'foo' function is important. Here, we call the function from left to right, from bottom to top:


The order in which foo is called (excluding recursive calls).
In fact, only the left half is necessary; if the left half is full of vertical blocks, the right half must also be full of vertical blocks.

It can be seen that when each cell is called, it must be either L or D. If the call is not a recursive one, all blocks on the left or below are oriented vertically, so the current cell cannot be U or R. If the call is a recursive one (the orange cell in the image on the right), the previously called cell (gray) is horizontal, and the cell above it is vertical, so the orange cell is L or D .


Recursive call.

Now, if the cell is L, call the rotate function on the cell above it. Otherwise, run a recursion and two rotations:


The gray cell in the above image is the initial cell where the function is called.
Can it get out of bounds? No, it will never reach the upper right boundary. Remind that initial call starts from somewhere in the left half. If we reach the upper right boundary, the cell cannot be L or D. This contradicts with the above observation that the called cell must be L or D.
If the maximum recursion depth is $i$, there are at most $2 i-1$ rotations. There are $i$ cells whose maximum recursion depth is $i$, so there are at most $\sum_{i=1}^{n} i(2 i-1)=n(n+1)(4 n-1) / 6 \leq n^{4}$ rotations.
The tightest bound is $n(n+1)(2 n+1) / 6$. Most implementations (including the above approach) will give an optimal solution. For more information, refer to the paper: https://arxiv.org/pdf/math/9201305.pdf

## B. Break Oven, Run Cookie!

The problem would have been much simpler if multiple cookies could be on one cell and they could still move after reaching a hole: to determine if all cookies can escape in $t$ seconds, connect cookie $i$ to hole $j$ iff it can reach there in $t$ seconds, and run a bipartite matching. All cookies can escape iff the maximum matching is $N$. Use binary search to find the minimum possible $t$. In fact, you can try this variant at https://www.acmicpc.net/problem/1348.
We can't use the above solution because of the additional requirements. But not all is lost; if we can somehow implement the requirements in the flow model, we can run maximum flow to determine if all cookies can escape in $t$ seconds! And to do this, we should determine the position of each cookie at each second, not just which cookie goes to which hole.
Make layers of grids, each layer representing each second of the grid. Use two nodes per cell so that each cell is used only once. Connect the source node to the cookies in the first layer. If the cell is weak, a cookie on that cell cannot move anymore, so connect it to the same cell in the next layer. Otherwise, connect it to the same cell and each adjacent cell in the next layer. Finally, connect the weak cells in the last layer to the sink node. All capacities are 1. If the maximum flow equals the number of cookies, then all cookies can escape in $t$ seconds.
What is an upper bound? Let's assume there is an answer such that for $1 \leq i \leq n$, cookie $C_{i}$ goes to hole $H_{i}$. Then we can think of other answer that every path of cookie is its shortest path to the corresponding hole. In addition, if the paths of the different cookies make up a cycle, it is always better to remove that, so do the preprocessing.
Now, the state of the cookies for each time can be expressed in a tree form. Each node in the tree represents a cell that any cookie is currently placed or intended to go next time. If a cookie at cell X want to go adjacent cell Y (whether other cookie is placed or not) next time, set Y to be the parent of X in the tree. In this case, each root of the tree always represents a cell without a cookie, and all other nodes represent cell with a cookie.


An example of the tree representation of the grid.
We can guarantee that there is no any cycle due to the preprocessing step.

Only cells that are adjacent on the grid can be connected in tree form. The maximum number of cookies trying to go to a cell is 3(Otherwise there would be a cycle), so the branching factor of tree is also up to 3 .

Now we can see the least number of cookies that can move to next position of its path. If you take a path from root to any leaf on tree, it's possible to all cookies included in the path move to the next cell, and all other cookies stay still. Therefore, at least [the sum of maximum depth of each tree] cookies can move next time. This value is minimized when all cookies belong to one tree(=every cookie has conflicting relationships), and the tree is balanced. That is, when there are n cookies remained, at least $\log _{3} n$ cookies can go to the next cell.

So how long does it take from the initial state to the time the last cookie escapes? The length of each path that cookie $C_{i}$ goes to the corresponding hole $H_{i}$ is up to $H W-N$, and the worst case is that cookies with the shortest remaining path move first. When do the simulation with this condition and the value of $H W$ is 100 , it can be seen that all of the cookies can be escaped within 1000 seconds in all possible cases. If you set the upper bound of the binary search to 1000, C++ solution takes close to 1 second for execution. However, as we used some strong assumptions that cannot happens in all possible cases, we can achieve stable execution times with smaller upper bound.

Despite the suggested upper bound, the actual answer is generally much smaller. Based on this fact, you can check if all cookies can escape in long enough time to avoid TLE (in case you use something like Python). From each cookie, run a BFS to determine which hole it can escape through. Connect each cookie to its usable holes, and run a bipartite matching.

## C. Coke Challenge

Each person should drink coke for $T=K / A$ seconds. Every $t_{i}+s_{i}$ seconds, the $i$ -th person drinks for $t_{i}$ seconds. Let's call it a "cycle". If $T$ is a multiple of $t_{i}$, they need $T / t_{i}$ cycles, but the last $s_{i}$ seconds of the last cycle is unnecessary. It means it takes $\left(t_{i}+s_{i}\right) T / t_{i}-s_{i}$ seconds. Otherwise, they need $\left\lfloor T / t_{i}\right\rfloor$ cycles and $k_{i}$ more seconds, where $k_{i}$ is the remainder from dividing $T$ by $t_{i}$.

Compute how many seconds each person needs to finish their coke, and output the minimum.

## D. Dev, Please Add This!

Imagine actually playing the game. There is a moment where you cross a "one-way road": this means when you roll the ball, sometimes you can't go back to where the ball originally was. In the first example below, you can freely move
between the four red cells, but not after collecting one of the stars:

$$
\begin{aligned}
& 37 \\
& \# 0 \cdot 0 \cdot 0^{\#} \\
& \# \cdot \# \# \# \# \\
& \text { ה. } \# \cdot \# \cdot 0^{\star}
\end{aligned}
$$

That means the red cells belong to the same strongly connected component. The first step is to find the SCC's. For simplicity, let's say SCC 0 is the SCC where the ball is originally placed on.

Suppose the ball is placed on SCC $x$. The player can safely collect all stars that can be collected while staying on SCC $x$. We would place a star on a few SCC's and require that the ball visit at least one of them. But how many SCC's for each star? At most two: one that corresponds to the left/right end of the star (green triangle in the below image), and one that corresponds to the up/down end of the star (blue square in the below image). Of course, you can also collect the star while going from SCC $x$ to SCC $y$, but that is equivalent to collecting it while staying in SCC $y$, which must be either the left/right end or up/down end of the star.


To collect a star, the ball must visit at least one of those two SCC's. This gives a clue for a 2-SAT approach. Denote $x_{i}$ as true if the ball visits SCC $i$, false otherwise. Then:

1. $x_{0}$ (because the ball is initially on SCC 0 )
2. $\neg x_{i}$ if SCC $i$ cannot be reached from SCC 0
3. $\neg x_{i} \vee \neg x_{j}$ if not both of SCC $i$ and $j$ can be reached
4. $x_{i} \vee x_{j}$ if the ball must be on SCC $i$ or $j$ to collect a star

It takes $O(E V)$ time to construct the formulae of the third type, so it might look like it won't work in time. But it does work because $E=O(V)$ in this graph. Solve the 2 -SAT problem and we get the answer.

There is another solution that constructs a 2-SAT formula using horizontal and vertical segments of the grid. (Thanks to cubelover, who used this during the open contest)

## E. Expectation of Games

Denote $x_{i}$ as the expected number of die rolls when the token is placed at the $i$ -th cell (outside of the board is the 0th cell). Denote $\operatorname{end}(j)$ as the resulting cell when the token just reaches $j: n$ if $j>n, s$ if there is a snake or a ladder that starts at $j$ and ends at $s$, and $j$ otherwise. Then we can construct a system of linear equations as $x_{i}=1+\sum_{j=i+1}^{i+m} \frac{x_{\operatorname{end}(j)}}{m}$. Use Gaussian Elimination to find $x_{0}$. If the system is unsolvable, output -1 .

One catch is that using double would result in wrong answer. Note that the answer can get very large. For example, set $(n, m, s)=(100,2,25)$ then place a snake from $27+3 i$ to $25-i$, for $0 \leq i \leq 24$. One would have to survive the $1 / 2$ probability 48 times in a row! Therefore, it's not unreasonable to assume that you would somewhere divide by a very small number during the Gaussian Elimination in order to get the large number.

It is indeed the reason why the error happens. In one of the test data, using the system of equation constructed above, you would divide by $2 / 205891132094649$. It was verified that long double can solve the problem, but unfortunately, some codes that used long double still got the wrong answer (though some of them made another mistake).

On a side note, the expected number of die rolls in the example mentioned above is about $2,335,199,806,784,699$. The largest answer in the test data is $79,036,724,938,709,836,284$.

## F. Faster Sorting

Define asc[i] as the last index of the non-decreasing subarray starting from the $i$-th element, and $\operatorname{desc}[i]$ similarly but with the decreasing subarray. Then asc $[n]$ is $n$, and $a s c[i]$ is $a s c[i+1]$ if the next element is larger than or equal to the $i$-th, otherwise $i$. The same reasoning goes to desc.

Now, we can process each query in $O(N / M I N R U N)$, since each subarray can be obtained in $O(1)$. However, it exceeds the time limit if all MINRUN values of each query are very small. Save the processed values and answer duplicate queries immediately to avoid the repetition.

The time complexity is $O(N / 2+N / 3+\cdots+N / N+Q)=O(N \log N+Q)$.

## G. God Game

After $\operatorname{lcm}(4,8,12,16,20)=240$ seconds, all obstacles come back to their initial position. It means there are at most $240 N M$ possible states of the game.

First, let's not consider the collisions at non-integer time; obstacles and the
player "warp" instead of moving. Make an $N$ by $M$ by 240 grid, and find which cell has an obstacle on each layer. This can be done by simulating the trajectory of each obstacle. For each obstacle, determine its position at each second, and indicate the presence of an obstacle at the corresponding cell and layer.

Then run a BFS: every time you move or stay, you automatically move to the next layer. (If you are on the last layer, you move back to the first.)
Since collisions can also happen at non-integer time, we have to save which obstacle is moving in which direction. One way to do it is bitmask: 1 is a wall, 2 is an obstacle moving left, 4 is an obstacle moving right, etc. The player cannot advance to the next cell if it moves against an obstacle. This way you can solve the problem using the same grid.

## H. Highway Track

Let $f(i, j)$ be the remaining amount of fuel right after reaching the $j$-th gas station (before refueling), starting from the $i$-th gas station. It can be negative, but keep computing anyway. Let $m(i)=\min \{f(i, j)\}$. $i$ is an appropriate starting point if and only if $m(i) \geq 0$. In fact, $m(i)$ is at most 0 since $f(i, i)=0$.
Compute all $f(i, j)$ for $i=1$. What happens when we compute all $f(2, j)$ ? Nothing except all values decrease by $f(1,2)$, a constant. Therefore $m(2)=m(1)-f(1,2)$. Similarly, $m(j)=m(1)-f(1, j)$. If $m(j)=0$, that means $m(1)=f(1, j)$. Therefore the answer is the number of $f(1, j)$ that equals $m(1)$.
Perhaps the best way to understand the solution is to draw the graph of $f(1, j)$. When the starting station changes, the shape does not change. Instead you push the graph upwards or downwards, "calibrating" it at the new starting station. To make the graph placed above the x-axis, you must calibrate it at the lowest point.

## I. Impossible Design

Run a naive solution on short permutations, and you'll notice something strange. There is a pattern in a permutation without intersecting sticks. Specifically, the input has no intersecting sticks if and only if it can be constructed this way:

1. Take two positive integers $a, b$ such that $a, b<n \leq a+b$ and $\operatorname{gcd}(a, b)=1$.
2. Write all integers $\equiv 0 \bmod a$ in order, then $\equiv b$, then $\equiv 2 b, \cdots$, finally $\equiv(a-1) b$. For example, if we take $(a, b, n)=(5,2,11)$, we get the permutation 0510 27491638.
3. Rotate the permutation. That is, take the last $k$ numbers and put them on the beginning of the permutation.
Once you figure this out, coding the solution is trivial. Determine $a$ and $b$ from the given permutation and check if the constructed permutation equals the input.

But how do we actually prove that the permutations constructed this way are
precisely all the permutations without intersecting sticks? This was actually the $6^{\text {th }}$ problem in IMO 2013. Since the solution is quite long, we will provide a link to two official solutions (page 33): http://imo-official.org/problems/IMO2013SL.pdf

## J. Jeong Lab

Let $m_{A}$ and $m_{B}$ be the maximum amount of A and B in $S$, respectively. Now let's consider each solution as a point in a coordinate plane, where $x$ coordinate is the amount of A and $y$ coordinate is the amount of B in the solution. Assume that the collection $S$ contains $\left(m_{A}, 0\right),\left(0, m_{B}\right)$, and ( 0,0 ). Then it is not difficult to observe that a solution $K$ is "bad" if and only if the representing point lies in the boundary and the interior of the convex hull consisting of points in $S$.
Suppose that Mr. Jeong only checks whether the solution is bad or not and does not add the his collection $S$ (i.e. $S$ is invariant). Then the queries are simply asking whether the given point lies in the convex hull. To answer these queries, we sort the points in $S$ in a counter-clockwise manner with respect to the origin. When the query point $(a, b)$ is given, we binary search the sorted points and find the appropriate position of $(a, b)$ as if we are trying to insert the point. (This can be done by functions such as lower_bound() in C++.) Then we check two points adjacent to the found position, and it is enough to check if the new point lies inside or outside the triangle consisting of that two points and the origin.
Now let's consider our original problem. Here, the set $S$ varies, so we need to update the convex hull efficiently. This is called dynamic convex hull problem, and it is difficult in general. But here, only the points in the first quadrant is added, and there is no delete operation, so the life becomes easier. Suppose we are trying to insert the point $(a, b)$. Similar to the previous paragraph, we first find the appropriate position to insert. Now we should "really" insert the point, but also we should delete some points that does not belong to the hull anymore. Observe that if we should delete the point $\left(a_{0}, b_{0}\right)$ in the hull, then all the points lying between $\left(a_{0}, b_{0}\right)$ and ( $a, b$ ) should be deleted too. So we could simply start from the position that we are trying to insert to, and iterate the sorted data structure to the front and the back until we can delete the point from the structure. We could use a data structure like red black tree (set in C++ and TreeSet in Java) to efficiently insert and delete from the sorted data. Note that we should also update $m_{A}$ and $m_{B}$.
Now let's check the time complexity. Using techniques like Graham scan, the convex hull can be built in $O(N \log N)$ time. Throughout the queries, we delete no more than $N+M$ points, so it takes $O((N+M) \log (N+M))$ time with red black tree for all $M$ queries. Therefore, the total time complexity is $O((N+M) \log (N+M))$ and it is enough for the limit $N \leqq 100,000$ and $M \leqq 100,000$.

## K. Kimino Ichi Wa。

First, realize that a graph satisfying the requirements is very specific. It is just a path, with at most one simple cycle attached to each intermediate vertex. For example:


Therefore, the length of a walk from a vertex A from another vertex $B$ is (The length of the shortest path from A to B) + (A linear combination of the lengths of the cycles that shares a point with that path). Denote $T, M$ as the starting point of Taki and Mitsuha, $V_{t}$ as the shortest path length from V to T , and $a_{1}, a_{2}, \cdots$, $a_{p}$ as the lengths of the cycle in the path. Define $V_{m}$ and $b_{1}, \cdots, b_{q}$ similarly but from V to M . The problem of deciding if Taki and Mitsuha can meet at V is then equivalent to answering "Is there a non-negative integer solution $\left\{x_{i}\right\},\left\{y_{i}\right\}$ such that $V_{t}+\sum_{i=1}^{p} a_{i} x_{i}=V_{m}+\sum_{i=1}^{q} b_{i} y_{i}$ ?
WLOG assume $V_{t} \leq V_{m}$. That is, M is farther than T from V . If Taki and Mistuha are to meet at V . Taki should spend time in a cycle, so $p>0$. Combined with $V_{t} \leq V_{m}$, we get $q>0$. From this, we prove that there is a non-negative solution iff $V_{m}-V_{t}$ is divisible by $g=\operatorname{gcd}\left(a_{1}, \cdots, a_{p}, b_{1}, \cdots, b_{q}\right)$.
$\sum_{i=1}^{p} a_{i} x_{i}-\sum_{i=1}^{q} b_{i} y_{i}$ is a multiple of $g$, so $V_{m}-V_{t}$ must also be a multiple of $g$ if there is a solution. If $V_{m}-V_{t}$ is a multiple of $g$, we can find an integer solution using extended euclidean algorithm. If one of the unknowns (say $x_{i}$ ) is negative, we can make it larger by picking any $y_{j}$, then adding $b_{j}$ from $x_{i}$ and $a_{i}$ from $y_{j}$. The equality still holds since $a_{i}\left(x_{i}+b_{j}\right)-b_{j}\left(y_{j}+a_{i}\right)=a_{i} x_{i}-b_{j} y_{j}$. By repeating this process, we obtain a non-negative integer solution. QED.
Now, for each vertex V , we check if Mitsuha and Taki can meet at V by computing the gcd of cycle lengths on the path from V to T , and from V to M . Among those vertices, pick the vertex closest to the starting station. It is unique
since it must be an intersection of the path and a cycle.

## L. Labor

Count the number of trapezoids with lower acute angles. The other type (upper acute angles) can be counted by switching the ropes and doing the same thing.
Let $L(i)$ be the number of lower points placed at the left side of the $i$-th upper point. Define $R(i)$ similarly, but at the right side. Then we have to compute $\sum_{i<j} L(i) R(j)$. This equals $\sum_{i}\left(L(i) \sum_{i<j} R(j)\right)$. Now, let $S(j)=\sum_{k=1}^{j} R(k)$. The formula then becomes $\sum_{i}(L(i)(S(n)-S(i))$ ). If we already have the values of $L$ and $S$, the counting can be done in linear time.
How do we compute $L$ ? Sort all the points; in case of a tie, The upper point comes first. Then sweep the points, keeping track of the number of lower points encountered. When you encounter an upper point, update $L(i)$. You can also compute $R(i)$ here: just subtract $L(i)$ from $M$, and subtract 1 more if the next point has the same x coordinate. Computing $S$ is much easier. Just set $S(1)=R(1)$ and $S(k+1)=S(k)+R(k+1)$.
It can be seen that the $y$-coordinates of the ropes are red herring that have no effect on the answer.

