



Editorial

Tasks, test data and solutions were prepared by: Ivan Janjić, Sofija Velkovska, and Jakov Celin.

Implementation examples are given in attached source code files.

Task Škare

Prepared by: Sofija Velkovska

Required knowledge: loop statement (for), decision statement (if), arrays

One way to solve this task is by keeping track of the positions of the cuts relative to the original strip, instead of storing all of the current strips separately. Let the array *rez* represent the original strip. Let the *i*-th element of *rez* be equal to 1 if there is a cut after the *i*-th centimetre of the original strip, and 0 otherwise. So, each of the strips obtained by cutting the original strip is represented by a segment of the array, with 1s determining where the strip begins and ends (or in the case of the first and the last strip, the beginning/end is represented by the beginning/end of the array), and 0s between them.

Suppose we want to cut the *x*-th strip after its *l*-th centimetre. If we want to cut the first strip, we should consider the segment from the first element of *rez* up until the first 1. If we want to cut the last strip, we should consider the segment from the last 1 up until the last element in the array *rez*. Otherwise, the *x*-th strip is located between the (*x* - 1)-th and *x*-th 1 in the array. Let's denote the position of the (*x* - 1)-th 1 in the array by *i*. The *l*-th centimetre of the *x*-th strip is the (*i* + *l*)-th centimetre of the original strip. This is where the cut should be made - we assign 1 to this position in the array *rez*.

After performing all the cuts, we calculate the length of each strip as the difference between the positions of the *x*-th and (*x* - 1)-th 1 in *rez*, or by using the appropriate modification for the special cases of the first and the last strip. We can keep track of the number of distinct strip lengths by using another 0/1 array *duljine*. We initially set the value of each element in *duljine* to 0. When we encounter a strip of length *l*, we set the value of *duljine*[*l*] to 1. Finally, the number of distinct strip lengths is equal to the number of 1s in *duljine*.

Task Težina

Prepared by: Jakov Celin

Required knowledge: complexity analysis, prefix sums

The problem could be solved in various ways, but we will describe the two easiest.

For each type of weight *i* where $1 \leq i \leq k$, it is necessary to calculate the so-called weight power. For each item of mass *a_x*, the value $\lfloor a_x/i \rfloor \cdot (a_x + 2)$ is computed, and if the result exceeds 10^8 , it is replaced with 10^8 . The power of weight *i* is equal to the sum of these values for all items, and the desired answer is the sum of the powers of all weights.

Direct computation for each pair *i* and *x* would be too slow because it would require $O(nk)$ operations.

One solution is to iterate over the weights in order and group the values for which the quotient $\lfloor a_x/i \rfloor$ takes the same value.

For a fixed *i*, it holds that $\lfloor a_x/i \rfloor = j$ for all values *a_x* satisfying $i \cdot j \leq a_x < i \cdot (j + 1)$. Thus, for each value of *j*, we can consider the interval $[i \cdot j, i \cdot (j + 1) >$ and process all elements of the array whose masses belong to this interval at once.

Let *C* be the number of elements in this interval, and *S* their sum. The contribution of all these elements to the total result is equal to $j \cdot (a_x + 2)$ for each element, so their total contribution is $j \cdot (S + 2C)$.

It is also necessary to take into account the 10^8 limit. Since $a_x \leq 10^5$, for sufficiently large values of *i*, the quotient $j = \lfloor a_x/i \rfloor$ becomes very small, so in this case it is impossible for the expression $j \cdot (a_x + 2)$ to



exceed the 10^8 limit. Therefore, it is sufficient to explicitly check the limit for approximately the first 101 values of i and cap the value at 10^8 if necessary. For larger values of i , this check is no longer needed.

For a fixed i , the number of different values of the quotient j is approximately a/i , where a is the maximum mass value (up to 10^5). The total number of iterations over all weights then becomes $a + a/2 + a/3 + \dots + a/k$, which is $O(a \log k)$.

The overall time complexity of the algorithm is $O(B \cdot n + a \log k)$, where B is the number of first weights for which the 10^8 limit can still be reached (approximately 101). This complexity is sufficient for the given constraints.

Alternative (and official) solution used the claim that for a fixed value x , $\lfloor a_x/i \rfloor$ can take $O(\sqrt{x})$ different values, and then directly computed the contribution of a single array element for $O(\sqrt{x})$ intervals of weights.

Task Pet

Prepared by: Jakov Celin

Required knowledge: bitset, ad-hoc, dynamic programming

We consider the cells of a matrix with value 1 as the vertices of a graph. From a cell (x, y) , Maša can jump to any other cell in the same row or column, with the type of jump alternating (row, column, row, ...). We need to find the number of paths that visit exactly 5 different cells.

First, let's calculate the number of all paths of length 5 without the condition that all cells are distinct. Define:

$f(x, y, i)$ – the number of ways to end at (x, y) after i jumps if the last jump changed the row

$g(x, y, i)$ – the number of ways to end at (x, y) after i jumps if the last jump changed the column.

Notice that $f(x, y, k)$ and $g(x, y, k)$ are equal to 0 if the cell (x, y) in the matrix is 0.

For the remaining cells, the transitions in the dynamics are:

$$f(x, y, i) = \sum_{z=1}^m g(x, z, i-1) - g(x, y, i-1)$$

$$g(x, y, i) = \sum_{z=1}^n f(z, y, i-1) - f(x, y, i-1).$$

Row and column sums can be maintained during the computation, so the total number of paths of length 5 can be calculated in $O(nm)$.

In this counting, repetition of cells is allowed. In a path of length 5, this is only possible if the path forms a rectangle $(r_1, c_1) \rightarrow (r_1, c_2) \rightarrow (r_2, c_2) \rightarrow (r_2, c_1) \rightarrow (r_1, c_1)$.

Therefore, we need to subtract all such paths. Each rectangle gives 8 different paths (we choose four starting vertices and two directions of traversal).

We count the number of rectangles as follows: for each pair of rows, we calculate the number of columns x where both rows have value 1. Then they contribute $x(x-1)/2$ rectangles. The intersection of ones in two rows can be quickly obtained using bitsets, in time $O(\frac{m}{64})$.

The final result is obtained as (number of all paths of length 5) $- 8 \cdot$ (number of rectangles). The total time complexity of the algorithm is $O(\frac{n^2 m}{64})$.

Task Slaganje

Prepared by: Ivan Janjić

Required knowledge: graph theory, ad hoc

One possible path that leads to the solution is to carefully choose a single embedding of the tree and rotate



it N times. The question now becomes when does the embedding and its rotations cover all sides and diagonals.

Define *length* of a side or a diagonal between vertices x and y as $d(x, y) := \min\{|x - y|, N - |x - y|\}$. Observe that $1 \leq d(x, y) \leq \lfloor \frac{N}{2} \rfloor$ and that every integer in that range is achieved for some x and y . Additionally, observe that rotations of a single edge cover all diagonals(sides) of some fixed length.

If we want to cover all sides and diagonals, our embedding has to have an edge for every possible length. If there exists a vertex that is an end of all edges, that is the tree is a star, every embedding works. We will try to decompose the tree into small disjoint stars in the hopes that there are enough edges part of the stars to cover all lengths.

There are multiple ways to achieve a decomposition with enough edges, here we describe only one. Root the tree arbitrarily and let's focus on a vertex on maximum depth x . Let $p(x)$ be the label for the parent of x . We add a star with a center in $p(x)$. Other vertices of the star are children of $p(x)$ (including x). Remove $p(x)$ and its children from the tree. We continue this process while we can.

Take some edge between vertices y and $p(y)$ that is not in any star. It's not hard to see that, from the way we constructed the decomposition, this means that y is a center of some star. In other words, according to the notation from the previous section, $y = p(x)$ for some x . Putting it all together, for every edge that is not in any star (between y and $p(y)$) we have found an edge that is in some star (between x and $p(x)$). It is obvious that there is no overlap between such pairs of edges. In conclusion, there are as many edges that are part of some star as there are edges that are not part of any star. This means that the number of edges that are part of some star is at least $\lceil \frac{N-1}{2} \rceil = \lfloor \frac{N}{2} \rfloor$.

It remains to arrange the stars so that they cover every possible length. Remove some stars and shrink others so that the union has exactly $\lfloor \frac{N}{2} \rfloor$ edges. We have 4 pointers a, b, c, d to vertices. If N is odd, at the beginning set $a = c = 1, b = \lfloor \frac{N}{2} \rfloor + 1, d = \lfloor \frac{N}{2} \rfloor + 2$. If N is even, at the beginning set $a = c = 1, b = d = \lfloor \frac{N}{2} \rfloor + 1$. If the first star has k (if N is even we make sure that $k \geq 2$) edges, put the center of the star on vertex a , and the other k vertices on vertices $b, b - 1, \dots, b - k + 1$. Change pointers in the following way: $a := a, b := b - k, c := c - 1, d := d + k - 1$. Of course, vertex 0 is the same as vertex N , vertex $N + 1$ the same as vertex 1 and so on. If the second star has l edges, put the center of the star on vertex c , and the other l vertices on vertices $d, d + 1, \dots, d + l - 1$. Change pointers in the following way: $a := a + 1, b := b - k + 1, c := c, d := d + k$. For the third star repeat the steps for the first star, for the fourth star repeat the steps for the second star and so on. It's not hard to see that the first star covers lengths $\lfloor \frac{N}{2} \rfloor, \lfloor \frac{N}{2} \rfloor - 1, \dots, \lfloor \frac{N}{2} \rfloor - k + 1$, second covers $\lfloor \frac{N}{2} \rfloor - k, \lfloor \frac{N}{2} \rfloor - k - 1, \dots, \lfloor \frac{N}{2} \rfloor - k - l + 1$ and so on.

For implementation details, see the official solution.

Task Struktura

Prepared by: Jakov Celin

Required knowledge: modular arithmetic, fast exponentiation, matrix exponentiation, combinatorics

Petar chooses a sequence of length n such that each element is selected completely at random from the set $1, 2, \dots, k$. The total number of possible sequences is k^n , which needs to be computed using the fast exponentiation technique. The desired probability is equal to the ratio of the number of sequences that satisfy the structure conditions to the total number of possible sequences.

If $k < n$, then the sequence cannot form a structure because each number from 1 to n must appear exactly once. In that case, the answer is 0.

Therefore, assume $k \geq n$. Since each number from 1 to n appears exactly once, the sequence a is actually a permutation of these numbers. An additional structure condition is $|a_i + i - n - 1| \leq 1$ for each i .

Let $f(n)$ be the number of permutations that satisfy the structure condition. Consider the position of the largest number n . It can only be in the first or second position (this follows directly from observing the condition for each i).



If $a_1 = n$, then the remaining $n - 1$ elements must form a structure of the same type on the numbers $1, \dots, n - 1$. In this case, we have $f(n - 1)$ possibilities.

If $a_2 = n$, then it must be that $a_1 = n - 1$, and the remaining $n - 2$ elements again form a structure on the numbers $1, \dots, n - 2$. In this case, we have $f(n - 2)$ possibilities.

It follows that $f(n) = f(n - 1) + f(n - 2)$. The initial conditions are $f(1) = 1$ and $f(2) = 2$, so $f(n)$ is precisely the n -th Fibonacci number.

The value of $f(n)$ can be computed in $O(\log n)$ time using matrix exponentiation for the standard Fibonacci recurrence. The total number of favorable sequences is $f(n)$, and the total number of all sequences is k^n . The desired probability is $f(n)/k^n$. The time complexity of the algorithm is $O(\log n)$.