CROATIAN OPEN COMPETITION IN INFORMATICS

Round 1

SOLUTIONS

COCI 2011/12	Task JABUKE
Round 1, October 22nd, 2011	**Author:** Adrian Satja Kurdija

Movements of the boat can be simulated by keeping track of the leftmost and rightmost column occupied by the boat.

If the apple is falling down a column between those two columns (inclusive), we do not need to move the boat.

Otherwise, if the apple is falling to the left of the boat, we can move the boat just enough to catch the apple - the boat's leftmost column needs to align with the apple's column. The new positions are simple to calculate: the leftmost column is set to the apple's column; the distance that the boat has moved is equal to the difference of the old and new leftmost column positions, and the rightmost column also moves that same distance to the left.

If the apple is falling to the right of the boat, we apply a completely analogous move. In the end, we just output the sum of all move distances.

Necessary skills:

for - loop

Tags:

greedy algorithms

COCI 2011/12	Task MATRIX
Round 1, October 22nd, 2011	**Author:** Adrian Satja Kurdija

A solution that iterates over all possible squares (by selecting all combinations of an upper-left cell and side length) and computes sums of diagonals for each of them has complexity O($\mathbf{N}^4$), which exceeds the time limit.

A faster solution, with complexity O($\mathbf{N}^3$), works by selecting a central cell of a square and gradually expands the square outwards until reaching an edge, while keeping track of the current diagonal sums. Whenever expanding the current square, the diagonal sums are updated and the maximum beauty is updated as well, if the new beauty value is larger. In the end, we simply output the final maximum value.

The algorithm described above must cover two cases: when the square centre is a matrix cell (i.e. with an odd-length square side) and when it is a corner of four cells (with an even-length side). The latter case is marginally more involved to implement.

Necessary skills:

matrix manipulation

Tags:

ad-hoc

COCI 2011/12	**Task X3**
Round 1, October 22nd, 2011	**Author:** Ivan Katanić

The first useful observation is that individual binary digits of the friendship value are mutually independent, so they can be considered separately. If the $\mathbf{i}^{th}$ digit of a result is equal to 1, a value of $2^{\mathbf{i}}$ is added to the friendship value, otherwise 0 is added.

A digit of the result is equal to 1 only if the corresponding digits in extraterrestrial names differ. Since we are computing the total friendship value, we can count the number of appearances of $2^{\mathbf{i}}$ (digit 1 in position **i**) for each **i**.

Let us denote by $\mathbf{K_i}$ the number of extraterrestrials who have the digit 1 in position **i** in binary name notation. Then we add

$$(\mathbf{K_i} * (\mathbf{N} - \mathbf{K_i})) * 2^{\mathbf{i}}$$

to the sum of friendships, since that is exactly the number of pairs with differing digits in position **i**, multiplied by the weight of that digit position.

Necessary skills:

number systems

Tags:

ad-hoc

For simplicity, let us find partners for boys who wish to dance with shorter girls. The other case is solved analogously, by swapping boys' and girls' heights. The following algorithm solves the problem:

```
solution := 0
for each boy
        find the tallest girl without a partner shorter than him
        if no girl was found, continue
        pair up the current boy and the girl
        solution := solution + 1
```

A naive implementation of this algorithm has complexity O($\mathbf{N}^2$) and is worth 60% of points. A key observation is that boys will be paired up with increasingly shorter girls if we iterate over boys from the tallest to the shortest one. This can be implemented with complexity O(**N** log **N**) or O(**N**+**H**), where **H** is the maximum height difference. Alternatively, a solution with complexity O(**NH**) is also worth all points.

Proof of correctness:

Let us denote by **b** and **g** the boy and the girl, respectively, in the **first** dance pairing we've made. In every optimal pairing, at least one of them will be paired up. Let us consider two cases:

1) Only one of them has a partner.

Without loss of generality, assume that the boy b is the one who has a partner. If we pair up **b** with **g** instead of his current partner $\mathbf{g_2}$, the number of pairs doesn't decrease.

2) Both the boy and the girl have a partner.

Let **b** be paired up with a girl $\mathbf{g_2}$, and **g** with a boy $\mathbf{b_2}$. $\mathbf{g_2}$ cannot be taller than **g**. $\mathbf{b_2}$ is taller than **g**, therefore he is also taller than $\mathbf{g_2}$. Thus the number of pairs doesn't decrease if we pair up **b** with **g** and $\mathbf{b_2}$ with $\mathbf{g_2}$.

The first pairing reduces the problem to a smaller problem, which can be solved by the same algorithm.

Necessary skills:

proof of correctness of a greedy algorithm

Tags:

greedy algorithms

COCI 2011/12	**Task SORT**
Round 1, October 22nd, 2011	**Author:** Goran Gašić

Let us first show that the sorting algorithm is correct. Consider the leftmost smallest element in the sequence. Until it is in the first position, every pass of the algorithm will apply a reverse operation on a subsequence including that element, thus moving it towards the beginning of the sequence. Once it is in the first position, it can be ignored. The procedure continues on the remainder of the sequence until each element is in the correect position.

Notice that, after the first pass, all reverse operations will be applied to subsequences with length 2. Since every element will, in some step, exchange positions with every greater element in a smaller position and every smaller element in a greater position, we are left with a classical problem of computing the number of inversions in a sequence.

In the reference solution, a Fenwick tree is used to solve the problem with complexity O(**N** log **N**). It is also possible to solve the problem using an interval (tournament) tree or merge sort, with the same complexity, or a bucket-based algorithm with complexity O(**N** sqrt **N**).

Necessary skills:

Fenwick tree (or interval tree)

Tags:

data structures

A solution with complexity O(T * N^2):

For each second, starting with 0, we compute a matrix of 1s and 0s, where a 1 denotes a square reachable by the knight in the respective second. The matrix for second 0 has 0s in all positions, except for the starting position of the knight. From each matrix, we can compute the matrix for the next second, up to second **T**. The next matrix is computed by mapping all 1s to all squares reachable by any of the 8 possible jumps from that square, and then zeroing out all positions blocked during the next second.

The memory complexity of this algorithm is O(**N**^{2}), since we need only two matrices (the current and the next one) at any moment. This solution is worth 40% of points.

A solution with complexity O(T * N):

Instead of the matrix of 1s and 0s, we will use a sequence of numbers, where each number represents a row of the matrix and its binary digits correspond to the 1s and 0s from the previous solution. For simplicity, we will continue to use the term *matrix* in the remainder of the description.

The mapping of 1s in all 8 directions can now be implemented using bit operations, which is more efficient than the previous solution.

The remaining problem is quickly computing the matrix of squares blocked in second **t**. Notice that if **t** is divisible by $\mathbf{K_{ij}}$, then the square (**i**, **j**) is blocked during second **t**.

If $\mathbf{K_{ij}}$ is greater than 1000, we can simply generate all moments when the square will be blocked, since there will be less than 1000 such moments.

If $\mathbf{K_{ij}}$ is less than 1000, the problem can be solved using prime factorization. Notice that, if $\mathbf{K_{ij}}$ divides **t**, all prime factors of **t** (including the ones with exponent 0) must have greater or equal exponents than the corresponding exponents in the factorization of $\mathbf{K_{ij}}$.

Let us denote by $\mathbf{F}(\mathbf{p}^{\mathbf{q}})$ the matrix with a 1 in position ($\mathbf{i}$, $\mathbf{j}$) if $\mathbf{q}$ is greater than or equal to the exponent of the prime number $\mathbf{p}$ in the factorization of $\mathbf{K}_{\mathbf{ij}}$.

Now the matrix of squares blocked in second t can be expressed as:

$$\mathbf{F}(\mathbf{p}_1^{\mathbf{q}_1}) \,\&\, \mathbf{F}(\mathbf{p}_2^{\mathbf{q}_2}) \,\&\, \dots \,\&\, \mathbf{F}(\mathbf{p}_k^{\mathbf{q}_k}) \qquad (1)$$

where $\mathbf{p}_1$, $\mathbf{p}_2$, …, $\mathbf{p}_k$ are all primes less than 1000, $\mathbf{q}_1$, $\mathbf{q}_2$, …, $\mathbf{q}_k$ are their exponents in the factorization of $\mathbf{t}$, and & is the bitwise AND operation.

The direct computation of the above expression (1) is slow. However, notice that at most 7 primes in a factorization will have positive exponents, while all other exponents will be 0.

Before the main algorithm, we will precompute, for each interval [$\mathbf{x}$, $\mathbf{y}$], the following:

$$\mathbf{G}(\mathbf{x}, \mathbf{y}) = \mathbf{F}(\mathbf{p}_\mathbf{x}^{\mathbf{0}}) \,\&\, \mathbf{F}(\mathbf{p}_{x+1}^{\mathbf{0}}) \,\&\, \dots \,\&\, \mathbf{F}(\mathbf{p}_\mathbf{y}^{\mathbf{0}}).$$

The expression (1) can now be quickly computed by using $\mathbf{F}(\mathbf{p}^{\mathbf{q}})$ for all primes with a positive exponent, and the precomputed $\mathbf{G}$ for all other primes (with exponents of 0, grouped in at most 8 intervals), combining all values with a bitwise AND.

Now that we can compute the matrix of squares blocked in second $\mathbf{t}$, we can simply apply it to the current matrix of possible positions using a bitwise AND, thus obtaining the final matrix for second $\mathbf{t}$.

Necessary skills:

prime numbers, bit operations

Tags:

mathematics