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

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($\mathbf{N}^2$), 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 $\mathbf{K}_{\mathbf{ij}}$, then the square (**i**, **j**) is blocked during second **t**.

If $\mathbf{K}_{\mathbf{ij}}$ 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 $\mathbf{K}_{\mathbf{ij}}$ is less than 1000, the problem can be solved using prime factorization. Notice that, if $\mathbf{K}_{\mathbf{ij}}$ 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 $\mathbf{K}_{\mathbf{ij}}$.

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

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

$$\mathbf{F}(\mathbf{p}_1^{\mathbf{q}_1}) \mathbin{\&} \mathbf{F}(\mathbf{p}_2^{\mathbf{q}_2}) \mathbin{\&} \ldots \mathbin{\&} \mathbf{F}(\mathbf{p}_k^{\mathbf{q}_k}) \tag{1}$$

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 **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 [**x**, **y**], the following:

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

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 **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 **t**, we can simply apply it to the current matrix of possible positions using a bitwise AND, thus obtaining the final matrix for second **t**.

Necessary skills:

prime numbers, bit operations

Tags:

mathematics

COCI 2011/2012	**Task RASPORED**
Round 2, November 19th, 2011	**Author:** Gustav Matula

Let's say that we are given some permutation (order in which pizza's are made) π_i, and we wish to calculate the tip we'll get:

$$\sum_{i=1}^{n}(R\pi_i - \sum_{j=1}^{i} V\pi_j) = \sum_{i=1}^{n} R\pi_i - \sum_{i=1}^{n}(n-i+1)V\pi_j$$

We can now conclude that order with regard to deadlines is not important, and that order with regard to durations must be increasing, so that pizza's that are baked longer are multiplied by smaller coefficient.

Brute force solution that sorts the pizzas after each update and calculates the above sum has complexity O(N log N) or O(N) and is worth 50 points.

Constraint given to **V_i** was sort of a hint that can lead to a simple solution: in some array **cnt** we can use **cnt[x]** to store the number of pizzas having baking time **x**. For **V_i** under 1000 we calculate every query by traversing the **cnt** array. This approach was worth 80 points.

We can use some data structure like segmented array (BIT) for implementing the **cnt** array, and aditional sequence that keeps the **V_i** sums (**sum[x]** = **cnt[x]*x**). To remove duration **V_p** for some pizza **p**, we must increase our solution by

$$V_p \cdot \sum_{i=0}^{V_p} cnt[i] + \sum_{i=V_p+1}^{maxV} sum[i]$$

Now we decrease **cnt[V_p]** by 1 and **sum[V_p]** by **V_p**. In order to insert **V_p** into our structure, we do the same thing, but with inversed signs and order (change **cnt** and **sum** first and then decrease the solution). **R_i** sums are trivial to calculate and don't require any data structures.

We can answer each query in O(log maxV), and total complexity is O((N+P) log maxV) which is good enough for obtaining maximum points for this task.

There is alternative solution that doesn’t depend on maxV. Idea is to maintain the sorted sequence of all the baking durations, along with the sums and corresponding coefficients. This can be achieved by using balanced tree (online solution), and by using segmented array built on top of the sorted sequence of all the durations (offline solution).

Complexity of this approach is O((N + P) log (N + P)). Offline implementation can be found in official solutions.

Required skills: mathematic problem analysis, data structures

Category: data structures

Notice the following: if, for a given position *i* in the large word (the one describing the street), we can find the longest short word (tile) that can be placed starting at position *i*, the problem is reduced to finding the union of intervals, which is solvable using a simple sweep.

The remaining problem is efficiently finding the longest tile for each position, which can be done in one of the following ways:

a) suffix array

A large number of problems with strings, including this one, can be solved using a suffix array. It is a sorted array of all suffixes of a string and can be computed using multiple methods. The simplest one, with complexity O($\mathbf{N} \log^2 \mathbf{N}$), using hashing is sufficient for this problem.

It is important to notice that the indices of suffixes, whose prefix is one of the given **M** words, are grouped together in a suffix array, forming a suffix interval. This interval can be found using binary search for each word. After finding all the intervals, we need to find, for each suffix, the longest of **M** words whose interval covers that suffix, which can be found using sweep. After that, another sweep can easily determine the tileable positions, using the longest word that can be placed at each position.

There are other efficient solutions using hashing.

b) Aho-Corasick tree

The Aho-Corasick tree can be used to find many short words in a long one using a single pass over the long word. The tree can be constructed with complexity O(sum_of_short_word_lengths), while finding matches depends on their number.

The basic idea behind the tree is well described in the following materials:
http://www.cbcb.umd.edu/confcour/CMSC858W-materials/Lecture4.pdf
http://www.cs.uku.fi/~kilpelai/BSA05/lectures/slides04.pdf

For the purposes of this problem, while building the tree, for each node we can keep the data about the longest of the given short words that is a suffix of the prefix represented by a particular node. Let M(*x*) denote that value for node *x*. M(*x*) is then the maximum of:

- depth(*x*), if a short word ends in *x*
- M(failureLink(*x*)) *

*failureLink is described in the given materials

Building the tree still has complexity O(sum_of_short_word_lengths), while the (since we are not interested in all match positions, but only the maximum length for each position in the long word) complexity of searching through the long word is linear in its length. When encountering a letter, we try to descend down the tree. If it is not possible, we follow the failureLinks until finding either a node which we can descend from, or the root of the tree. In the node where we have ended up, we take the previous computed maximum and set the interval bounds for the future sweep.

Required skills: suffix array, Aho-Corasick tree, sweepline

Category: strings, sweepline