Problem Tutorial: "Um_nik's Algorithm"

Solution is really simple: make 19 iterations of Dinic algorithm. I state that after k iterations size of matching is at least $\frac{k}{k+1}K$.

Lemma: If there are no augmenting chains that contains less than k edges from matching M, then the size of matching M is at least $\frac{k}{k+1}K$. Proof: Let's look at symmetric difference of matching M and maximum matching. $K - |M|$ is bounded by the number of chains of odd length that start and end with an edge from maximum matching. But each such chain is an augmenting chain for M. Each such chain should have at least k edges from M, so number of such chains is bounded by $\frac{|M|}{k}$. After solving linear inequality we will get $|M| \geq \frac{k}{k+1}K$.

After each iteration of Dinic algorithm the shortest path from S to T will increase. Initially it is 3, and it will be always odd, because graph is bipartite. Then after k iterations the shortest path will be at least $3 + 2k$. That actually means that there are no augmenting chains that contains less than k edges, so by the Lemma we are done.

Dinic algorithm actually creates $2(n_1+n_2+m)$ edges, which is not great, so you should use Hopcroft-Karp algorithm, which is just Dinic optimized for bipartite graphs.

Problem Tutorial: "String Algorithm"

Let's calculate polynomial hashes for all prefixes, now we can calculate hashes for all substrings in $O(1)$.

We can solve the problem for one k in $O(n)$ time as follows. For every string calculate hashes of initial string and k more strings, i-th of them being initial string with i-th symbol changed into `?`. By maintaining number of previous occurrences for each hash in hashmap we can calculate the answer.

We can also solve the problem for one k in $O\left(\left(\frac{n}{k}\right)^2\right)$ time as follows. Try all pairs of string, take difference of two hashes. We have to check if it is a difference of two hashes of strings that are different in at most 1 symbol. There are only $O(n \cdot |\Sigma|)$ possible values of such hashes, so if we have them in hashset, we can perform check in $O(1)$ time.

We can maintain a hashset with all interesting values while trying all k in increasing order, total time will be $O(n|\Sigma|)$.

If we will use first approach for $k \leq \sqrt{n}$ and second approach for $k > \sqrt{n}$, total complexity will be $O(n\sqrt{n})$. First part is obvious, second part we can bound by $\sum_{k=\sqrt{n}+1}^{n} \left(\frac{n}{k}\right)^2 \leq n^2 \cdot \int_{\sqrt{n}}^{n} \frac{dx}{x^2} = n^2 \cdot O\left(\frac{1}{\sqrt{n}}\right) = O\left(n\sqrt{n}\right)$.

Problem Tutorial: "StalinSort Algorithm"

Let $nxt(i)$ be the next element greater than a_i. Let's add elements to the answer one by one, call an element *stable* if we never remove it later. Let $f(i)$ be the maximum number of elements that can remain on the prefix of length i among all possible solutions where a_i is stable.

If we have a prefix of a solution where a_i is stable, what can be the next stable element of this solution? It turns out that it should be on the segment $\big[nxt(i), nxt(nxt(i))\big)$, because when we continue, we must first erase all element on the segment $\big(i, next(i)\big)$, then add $a_{nxt(i)}$. Then we either never erase it (so that $a_{nxt(i)}$ is stable), or replace it with some element (possibly, multiple times) before adding $a_{nxt(nxt(i))}$. Once $a_{nxt(nxt(i))}$ is added, previous elements can't be erased anymore.

We can show that the next stable element can be any element from the segment $\big[next(i), next(next(i))\big)$ with value greater than a_i. Indeed, to make a_j the next stable element, we can erase all elements till a_j after adding $a_{nxt(i)}$, and then erase $a_{nxt(i)}$. Note that for any j in this segment $a_j < a_{nxt(i)}$.

That means that we can calculate $f(i)$ using dynamic programming with transitions $f(i) + 1 \to f(j)$ for all j such that $j \leq nxt(i) < nxt(nxt(i))$ and $a_j > a_i$. To process the transition fast, use scanning line by value or by index and some data structure. The total complexity is $O(n \log n)$.

Problem Tutorial: "FFT Algorithm"

Carmichael function $\lambda(m)$ is the smallest positive integer x such that $a^x \equiv 1 \pmod m$ holds for all a coprime with m. Order of each element divides $\lambda(m)$, and for every m there exists a with order exactly $\lambda(m)$. This means that answer exists if and only if $2^k | \lambda(m)$.

If factorization of m is $m = \prod_{i=1}^{t} p_i^{\alpha_i}$, then $\lambda(m) = LCM(\lambda(p_1^{\alpha_1}), \lambda(p_2^{\alpha_2}), \ldots, \lambda(p_t^{\alpha_t}))$. 2 is prime, so if $2^k | \lambda(m)$ then there should be i such that $2^k | \lambda(p_i^{\alpha_i})$. We will check all the primes that can be good.

For prime $p \geq 3$ we know that $\lambda(p^\alpha) = (p-1)p^{\alpha-1}$; also, $\lambda(2^\alpha) = 2^{\alpha-2}$ (if $\alpha \geq 3$). So, if $2^k | \lambda(p^\alpha)$, p is either 2 (and $\alpha \geq k+2$), or has form $w2^k + 1$.

We can test all the primes up to $\sqrt{m2^{-k}}$, and all the primes of form $w2^k + 1$ for w up to $\sqrt{m2^{-k}}$. The only possibility left is that m itself is prime of that form. We can use Miller-Rabin primality test to check that possibility, but actually we can just run checking procedure (described below) assuming that it is prime and abort it if it cannot find the answer for too long :).

Now we have to try to find the answer for given prime power. If we could find the element with order $\lambda(p^\alpha)$, we would just take its $2^{-k}\lambda(p^\alpha)$ power and be done. The thing is the number of such elements is big — $\phi(\lambda(p^\alpha))$ — so we can just take one at random, take its $2^{-k}\lambda(p^\alpha)$ power and check that it is a valid answer, after several hundred tries probability of failure for good prime power is negligible.

And in the end you have to restore the answer for m using the answer for p^α. It can be done using Chinese Remainder Theorem.

Total complexity is $O(\sqrt{m2^{-k}} + \log^3 m)$.

Problem Tutorial: "Binary Search Algorithm"

Multiple solutions are possible. One solution is a binomial heap. Another solution is a binary heap with additional information: you should store which of the children is smaller.

However, the easiest solution is the segment tree: suppose each leaf stores p_x when x is in S, and ∞ otherwise. On update, we should recalculate the minimum value in each node that covers the position x. Only $(\log n + O(1))$ elements are the candidates for all these minimums, so we can just query all of them and then proceed as in usual segment tree.

Problem Tutorial: "Face Recognition Algorithm"

According to Euler's formula $V + F - E = 2$. Since each face has at least 3 edges, $3F \leq 2E$, and inequality turns into equality if and only if all faces are triangles. $E \leq 3V - 6$ for all planar graphs, and $E = 3V - 6$ if and only if all faces are triangles. Therefore, we can write a solution in $O(1)$.

Problem Tutorial: "Petr's Algorithm"

Note that, for a given value k, the earliest position element x can occur is $(x - k + 1)$. The probability that x is on this position is k^{-1}: the only option for this is when it is shuffled to the front on the step $(x - k + 1)$ and never shuffled later. The probability for a particular x being later than $(x - k + 1)$ is $(1 - k^{-1})$. The probability that for all $y \geq k$ their positions is not $(y - k + 1)$ is

$$(1 - k^{-1})^{n-k} \leq \left[(1 - k^{-1})^k\right]^{\frac{n}{k}-1} \leq e^{-19}.$$

So we can assume that it never happens and output the maximum of $(p_x - x + 1)$ for all x.

Problem Tutorial: "Greedy Algorithm"

Applying operation to rows doesn't change differences between cells lying in one row (neither in the row we modified nor the other ones). Same works for the columns as well. This means that we can solve the problem for rows and columns independently and then sum up the results.

Let's describe the case when we only modify rows. Denote a_i as the number of operations applied to the i-th row. Coin between cells (i, j) and $(i \bmod n+1, j)$ appears if and only if $h_{i,j}+a_i = h_{i \bmod n+1,j}+a_{i \bmod n+1}$. This can be rewritten as $h_{i \bmod n+1,j} - h_{i,j} = a_{i \bmod n+1} - a_i$. If we denote $d_i = a_{i \bmod n+1} - a_i$, then $d_1 + d_2 + \ldots + d_n = 0$ must hold. Let $f_i(d)$ be the number of indices $1 \le j \le m$, such that $h_{i \bmod n+1,j} - h_{i,j} = d$. Then the problem can be formulated as follows: find $d_1, \ldots, d_n$, such that $d_1 + \ldots + d_n = 0$ and $f_1(d_1) + \ldots + f_n(d_n)$ is maximal. Note that each function f_i has no more than m non-zero elements. Also, $f_i(d) = 0$ when $|x| > 500$. One can prove that for any array d with zero sum we can find a suitable array a.

Consider the case where at least one of $f_i(d_i) = 0$ in optimal solution. Then we can maximize $f_j(d_j)$ for $j \ne i$ independently and then choose d_i so that the sum of all d is equal to 0, which means that such solution can be found greedily.

Another case where all $f_i(d_i) \ne 0$ can be solved as a knapsack problem. We have only $O(m)$ options for d_i in each row. All d_i's must have the absolute value not greater than 500, so the absolute value of intermediate sum of d_i's will always be less or equal to $500n$. The total complexity will be $O(n \cdot (500n) \cdot m) = O(500n^2m)$.

Problem Tutorial: "Euclid's Algorithm"

Let's solve the problem for each prime p independently. We want to find the minimal power of p in all the numbers $(a+d)^k - a^k$. First we can look at p which do not divide d. We can take $a = p$ and see that this p do not divide $(a+d)^k - a^k$.

$(a+d)^k - a^k = \sum_{i=1}^{k} \binom{k}{i} d^i a^{k-i}$. If the power of p is minimal for exactly one of these summands, then the power of p in the sum will be equal to this minimum.

Let's say that α is a power of p in d and γ is a power of p in k. Let's for now assume that p does not divide a, this will minimize the minimum if it is unique. Then the power of i-th (for $i < p^\gamma$) summand is $i\alpha + g(\gamma, i)$, where $g(\gamma, i)$ is the power of p in $\binom{k}{i}$, which can be calculated using Lucas theorem: $g(\gamma, i) = \gamma - f(i)$, where $f(i)$ is power of p in i.

We want to prove that 1st summand *almost always* has the unique minimal power. 1st summand has the power $\alpha + \gamma$. To prove it we want $f(i) < (i-1)\alpha$ and $\gamma < (p^\gamma - 1)\alpha$. Both of these inequalities are true when $\alpha \ge 2$ or $p \ge 3$. So, the only case we have to consider is $p = 2$, $\alpha = 1$.

Using similar ideas we can show that in this case only 1st and 2nd summands will have minimal power (equal to $\gamma + 1$). Since $p = 2$, it means that the sum has power at least $\gamma + 2$.

Answer for (d, k) should divide answer for $(2d, k)$, since $(a+2d)^k - a^k = ((a+2d)^k - (a+d)^k) + ((a+d)^k - a^k)$. That means that power of 2 cannot be greater than $\gamma + 2$.

Are we done? Unfortunately, there is a corner case inside this corner case. Remember that we have said that when p divides a it is worse for us? It turns out that it is not true for $\alpha = 1$, $p = 2$ **and** $k = 2$ since we only have two summands (both of them are minimal), and we can use a to increase power in one of them so that it is not minimal anymore.

One last problem: we cannot factorize number d. But we don't need to, we only have to multiply it by all the parts of k that are not coprime with d, it can be done with gcd (Euclid is with us). Total complexity is $O(\log^3 \max(d, k))$

Problem Tutorial: "Closest Pair Algorithm"

First, let's rewrite the code provided in statement in a more convenient way:

```
choose random angle phi
don't rotate the plane around the origin
let p be the array of points
let q be the array of indices of points
sort indices q by the (p[q[i]].x * cos(phi) - p[q[i]].y * sin(phi)) value in increasing order
ans = INF;
```

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        ANS[i][j] = -INF;

for (int i = 0; i < n; i++) {
    for (int j = i - 1; j >= 0; j--) {
        ANS[q[i]][q[j]] = ans;
        ans = min(ans, dist(p[q[i]], p[q[j]]));
    }
}

cnt = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if ((p[i].x * cos(phi) - p[i].y * sin(phi)) -
            (p[j].x * cos(phi) - p[j].y * sin(phi)) < ANS[i][j]) {
            cnt++;
        }
    }
}
```

Then the required number of calls to function "`dist`" will be stored in cnt.

Let's start increasing the angle phi from 0 to 2π and maintain correct arrays q and ANS. For some phi, equality $p_i.x * cos(phi) - p_i.y * sin(phi) = p_j.x * cos(phi) - p_j.y * sin(phi)$ will hold for some pairs (i, j). That means, that we should swap values i and j in array q. One can prove, that this values will always lie in consecutive positions in q and there will be $O(n^2)$ such events.

Let's understand what happens to array ANS when we swap two consecutive elements of array q. Value of ANS_{q_i,q_j} can be expressed as the minimum distance between some subset of pair of points. When we swap two elements q_k and q_{k+1} these subsets will only change for pair of points (q_i, q_j), where at least one of $q_i = q_k, q_i = q_{k+1}, q_j = q_k, q_j = q_{k+1}$ is true. For all of such pairs we can recalculate the value of ANS in $O(n)$ time in total. Note that, $ANS_{q_k,q_{k+1}}$ will become $-INF$, since this pair will not be considered in this order in original code. Also, ANS_{q_{k+1},q_k} will change from $-INF$ to some positive value.

It can be shown, that for a fixed pair (i, j) the value of $ANS_{i,j}$ can take $O(n)$ different values during the described procces (increasing phi from 0 to 2π). For each pair (i, j) and for each segment of angles, when the value of $ANS_{i,j}$ is constant we can solve the inequality $p_i.x * cos(phi) - p_i.y * sin(phi) < ANS_{i,j}$ and add the length of resulting subsegment of angles to the answer. After all we need to divide total obtained sum by the 2π.

The total complexity of the solution is $O(n^3)$.

Problem Tutorial: "Interactive Algorithm"

Possibly there are multiple solutions to this problem, the authors' idea was to allow solutions that make $O(n \log n)$ queries with different constants, so the query limit was loose enough.

Authors' solution sequentially finds pairs of adjacent elements, then glues them together and proceeds with these elements being adjacent in all future queries. Thus, at each moment we have some blocks of elements that are consecutive in the answer permutation, and we can subtract the number of glued pairs from any similarity value that we get in a query. This way the value we get is equal to the number of pairs of blocks that stand next to each other in the answer. Let this value be s.

One solution is to shuffle the blocks until we get $s = 1$. Then binary search to find the position of the match, each time shuffling half of the segment under consideration (and, possibly, everything outside of the segment) until we get $s = 0$. Don't forget to alternate which half to shuffle. When we have $s = 0$, we

can narrow down the segment to the part just shuffled. We can estimate the expected number of iterations required to get $s = 0$ or $s = 1$ (each of them) as roughly e when the number of blocks is large enough. On practice this approach makes around 19 000 queries for $n = 400$.

Another solution is to shuffle the blocks until we get $s > 0$. Then, again, binary search to find the position of the match, but this time we shuffle only one half of the segment under consideration and wait for the decrease in s. This solution makes around 7 000 queries on practice.

Important part in both solutions is that it can happen for small number of blocks (≤ 5) that we can't find $s = 0$ (in the first solution) or decrease in s (in the second solution). To fix that, just try all possible orderings of remaining blocks.

Problem Tutorial: "Not Our Problem"

First of all, we should check if we change all ? to 0 we will get good array. Otherwise the answer is 0. Then if we have ? surrounded by 0 or ? then we can change it to any number and answer is infinite. In all other cases all ? should be not greater than C, so answer is finite.

From the first part we know that there are no 3 consecutive ?, so now we have many independent problems of forms $(x\,?\,y)$ or $(x\,?\,?\,y)$. In both cases we can easily determine the limit on all ?, and problem of form $(x\,?\,y)$ ends here.

For the second part we can draw good points on a plane. There is a fixed figure for given C, and it can be represented as a union of square of size $C^{1/3}$ and two figures with one dimension of size $C^{1/3}$. We have to intersect this figure with a number of rectangles. It can be done by precalculating prefix sums and doing binary searches to find places where the figure changes.

Total complexity is $O((n + C^{1/3}) \log C)$.