Problem Tutorial: "Abstract Circular Cover"

Let's firstly solve the problem for one given k.

In fact, our problem is to find k points at which we split our cycle onto circular segments. This problem would be much simpler if we had a line instead of a circle. So, let's guess one of the points at random. If we guess one point correctly, it would suffice to run dynamic programming on a line in $O(n^2k)$. The answer contains k distinct points, so we guess one of them with probability $\frac{k}{n}$. That's it, in order to get error probability $2^{-\varepsilon}$, we have to repeat the process $O(\frac{n}{k}\log\frac{1}{\varepsilon})$ times, achieving the total complexity of $O(n^3\log\frac{1}{\varepsilon})$.

Now back to our problem. Note that dp for k segments also compute answers for every number of segments smaller than k. Then, in order to achieve probability $2^{-\varepsilon}\cdot n$ (note additional n factor), we have to repeat the process only $O((\frac{n}{k}-\frac{n}{k+1})\log\frac{1}{\varepsilon})$ times for each k.

$$\sum_k\left(\frac{n}{k}-\frac{n}{k+1}\right)n^2k\log\frac{1}{\varepsilon}=\sum_k\frac{n}{k+1}n^2\log\frac{1}{\varepsilon}=O\left(n^3\log n\log\frac{1}{\varepsilon}\right)$$

Therefore, the total complexity is $O(n^3\log^2 n\log\frac{1}{\varepsilon})$.

Note that it is only the upper bound, and in practice, the probability of error is smaller giving the total complexity about $O(n^3\log n\log\frac{1}{\varepsilon})$.

Problem Tutorial: "Biggest Set Ever"

The thing we need to calculate is the *rem*-th coefficient of $(1+x^0)(1+x^1)\dots(1+x^{T-1}) \bmod (x^n-1)$. Multiples are periodic with period n, so, if we denote $P(x)=(1+x^0)(1+x^1)\dots(1+x^{n-1})$, then we need to calculate $P(x)^{T/n}(1+x^0)\dots(1+x^{(T-1) \bmod n}) \bmod (x^n-1)$. Also, we can multiply by $(1+x^a)$ in $O(n)$ time, so, if we calculate $G(x)=(P(x)^{T/n}) \bmod (x^n-1)$, we will do the rest in $O(n^2)$.

Consider $e_1, e_2, \dots e_n$: the n-th roots of unity. It turns out that $P(e_i)$ is integer for every e_i. Indeed, $P(x)$ is very symmetric, and if $\gcd(n,i)=\gcd(n,j)$, then $P(e_i)=P(e_j)$. Then, it can be proved by induction on d, divisors on n, that $P(e_{n/d})$ is rational (and in fact integer), using calculations like in FFT. So every $P(e_i)$ is integer. Also, $G(e_i)=P(e_i)^{T/n}$. Now you can implement calculations like in this induction to get $P(e_i)$, and reverse calculations to restore G from $G(e_i)$. Another way is to notice that it means that $(P(x)^t \bmod 998\,244\,353)$ is periodic with period $998\,244\,352$, and you can use standard binary exponentiation with FFT to calculate G.

The complexity is $O(n^2+\log T+d(n)\log MOD)$ or $O(n^2+\log T+n\log n\log MOD)$ depending on solution, where $d(n)$ is the number of divisors of n, and $MOD=998\,244\,353$.

Bonus: how fast can you do the $O(n^2)$ part? Authors can do it in $O(n^{1.5}\,\mathrm{polylog}(n))$, maybe you can do faster?

Problem Tutorial: "Convex Sets On Graph"

Consider arbitrary convex set C. For any two distinct points $x, y \in C$, if the graph has a block B containing both x and y, then $B \subseteq C$. Build a block-cut tree and run dynamic programming on it.

Problem Tutorial: "Delete Two Vertices Again"

Consider some DFS tree of the graph. Each edge has a lower vertex (let's call it v) and a higher vertex (let's call it r). By DFS tree properties, r is an ancestor of v (possibly, r is the parent of v). Let's handle all the edges with the same v at the same time.

When we delete the edge $v \to r$, the DFS tree splits into several components: H (the component containing vertices above r (and some other vertices)), M (the component containing the vertices strictly between v and r (and some other vertices)) and the components L_i of children of v. Then, to know whether the resulting graph is connected or not, we need only to know whether H and M, H and L_is, M and L_is are

connected (L_i and L_j can't be connected because of DFS tree properties). $H = \varnothing$ or $M = \varnothing$ are corner cases that can be handled almost naively. To deal with the rest, precalculate some binary liftings to check whether H and M are connected and maintain whether L_is are connected to M and H during the scanline over r (it can be done with a segment tree). Total time and space complexities are $O((n+m)\log n)$.

Problem Tutorial: "Easy One"

Let's solve this hard problem step by step. We will call operations 1 and 4 *increasing*, and call operations 2 and 3 *decreasing*, because they increase or decrease the sum of numbers by one, respectively. Also let's denote answer to the problem as $ANS(a, b, turns)$.

- $turns \bmod 2 = 0$, $\frac{turns}{2} \geq |a - b|$, otherwise the answer is 0.
- Suppose $a = 0$ and $turns = 2b$. Then the answer is $(2b)!!$. Why? Let's look how the rightmost "2" appeared. Only way is that we wrote "1" at some step, and transformed it into "2" at the next step. And we can insert this pair of operations in any of $(2n-1)$ places, so the rightmost "2" multiplies the answer by $(2n-1)$.
- But we can look at it in a bit different way (we are still in case $a = 0$ and $turns = 2b$). Lets's draw opening bracket in i th position, if we draw "1" in this step, and closing, if if transform "1" into "2" in this step.
- Then this bracket sequence is correct, and pairs of brackets, which correspond to one position, are paired in correct brackets order. Then we can look at this sequence as in DFS walk on tree (opening brackets means turn up, closing - turn down), and every vertex corresponds to some position in final string, let's call this position as **number in vertex**.
- Then the numbers in vertices forms a topological sorting of tree. Moreover, it is a bijection between ways to obtain b symbols "2" form zero symbols in $2b$ steps and pairs (Tree with order on childs of size b, topsort on it).
- Suppose now that a is nonzero, but still $turns = 2(b-a)$. Then in previous bijection size of tree is still $b - a$, but numbers in tree are from 1 to b, so $ANS(a, a + 2turns, turns) = ANS(0, 2turns, turns)\binom{a}{a+2turns}$
- Now we need to handle decreasing steps somehow. Suppose we have $a = 0$. Way to look at them is the following: we make an increasing step, such that last digit becomes "2", then delete this last digit. In terms of trees this is the following: we have two types of vertices - alive (and they have number in it) and deleted (without numbers in it), and every deleted vertex has in correspondence oriented edge (every edge can be in correspondence with no more than one deleted vertex), which is after we leave subtree of this vertex in DFS order (that means that we delete symbol "2", which is corresponded to this vertex, in the step, which is corresponded to the edge). Moreover, alive vertex can't be the child of deleted one, and if some deleted vertex is a child of another deleted, then it's edge is before father's in DFS order.
- Let's fix form of the tree T of size n (with child's order). Let $f(k, n, T)$ be number of ways to build a structure form previous paragraph with k deleted vertices, that is, topsort on alive and corresponding edge to every deleted vertex, with properties from previous paragraph. That $\frac{f(k,n,T)}{f(0,n,T)}$ doesn't depend or T and has simple formula. It can be proved by induction on tree's size.
- So $ANS(0, b, turns)$ has easy formula too. Now let's pass to the general case. Initially we have a symbols "2". Suppose none of them died. Then everything is like in the case with $a = 0$, but numbers in topsort are from 1 to b, and number of alive vertices is $b - a$, so number of such ways is $\binom{b}{a} ANS(0, b-a, turns)$.

- Now let's suppose that exactly one initial "2" was deleted. That is, we have tree with $b - a + 1$ alive vertex and $\frac{turns}{2} - (b - a + 1)$ deleted vertices, and we should mark one extra edge, which correspondes to initial deleted vertex. But it can be arbitrary unoccupied edge! So the number of ways is $ANS(0, b-a+1, turns)$, multiplied by the number of unoccupied edges, which is the same for every tree and easily computable.
- This logic works for arbitrary number of deleted initial vertices. So the answer is easily computable sum of length $O(n)$.

Complexity is $O(n)$.

Problem Tutorial: "Football"

Firstly, make a such rotation that the line $y = 0$ would be a bisector of the segment between players. Suppose that the ball is closer to the opponent than to the teammate. We win if and only if there is such a point on $y = 0$ that the ball could reach earlier than the opponent (and draw if simultaneously).

Let F be a function that computes time that we need to go to point $(x, 0)$ from point p with speed s.

$$F_{p,s}(x) := \frac{\sqrt{(x - p.x)^2 + p.y^2}}{s}$$

Instead of comparing $F(p_1, s_1)$ and $F(p_2, s_2)$, we compare their squares. Let p_1, s_1 be the opponent position and speed and p_2, s_2 be the ball position and speed.

$$\begin{gathered} F^2_{p_1,s_1} - F^2_{p_2,s_2} \\ ((x - x_1)^2 + y_1^2)s_2^2 - ((x - x_2)^2 + y_2^2)s_1^2 \\ (s_2^2 - s_1^2)x^2 - 2(x_1 s_2^2 - x_2 s_1^2)x + (x_1^2 + y_1^2)s_2^2 - (x_2^2 + y_2^2)s_1^2 \end{gathered}$$

Then, it is a polynomial of degree 2, and its discriminant is

$$D = (x_1 s_2^2 - x_2 s_1^2)^2 - (s_2^2 - s_1^2) \cdot ((x_1^2 + y_1^2)s_2^2 - (x_2^2 + y_2^2)s_1^2).$$

Problem Tutorial: "Game On Board"

Consider the bipartite graph, where vertices are rows and columns, and edge is black cell. Then painting fourth corner of some rectangle didn't change the connected components of graph, so initially the graph should be connected. And conversely, every connected graph can be transformed into a complete graph by this operations. So the thing we need to calculate is the number of spanning trees in complete bipartite graph with parts of sizes n and m. This can be done using Kirchhoff theorem: the number of spanning subtrees is equal to the determinant of some matrix. Matrix can be diagonalized using elementary transformations, and finally the determinant is $m^{n-1}n^{m-1}$.

The complexity is $O(\log n + \log m)$.

Problem Tutorial: "Hardcore String Counting 2"

The original table (see `https://oeis.org/A006156`) from the OEIS (with $n = 110$) was generated by exhaustive enumeration of all ternary square-free words on a supercomputer. Of course, we can't beat a supercomputer without better algorithms.

Let Good(ℓ) be the number of square-free words of length ℓ and Mini(ℓ) be the number of *minimal squares* of half-length ℓ. A *minimal square* is a square that does not contain smaller squares as substrings.

Clearly, any word of length ℓ that *is not* square-free contains a minimal square of length at most $\ell/2$. Hence, we have the following solution in time $O(\text{Good}(n/2)\cdot n+\text{Mini}(n/2)\cdot n^2)$ and space $O(\text{Mini}(n/2)\cdot n)$: build an Aho-Corasick automaton for all minimal squares of length at most $n/2$ and compute the number of square-free strings of length at most n by dynamic programming over said automaton. Now, the time complexity is tractable for a precalculation, but the space usage is way too large (we need to use much more RAM compared to what is usually available).

It may be possible to fit the above solution into the RAM of your computer by doing several constant optimizations, such as exploiting the internal symmetries of the Aho-Corasick automaton structure and computing the answers modulo several small primes and restoring them with Chinese remainder theorem. I decided to not cut off such tricks, because the problem is already pretty difficult, but the intended solution is much cooler.

The key observation in the intended solution can be seen as an incomplete application of inclusion-exclusion principle. As it turns out, it is easy to describe words that are not square-free but don't contain small minimal squares (with half-lengths at most $\ell/3$). Then, we can independently find the number of words that do not contain "small" squares as substrings (with half-lengths at most $\ell/3$) and the number of words with only mid-sized (with half-lengths from $\ell/3$ to $\ell/2$) squares as substrings.

More precisely, a word of length ℓ that doesn't contain squares of lengths at most $\ell/3$ can't contain two squares with different half-lengths as substrings. More formally, each such word has a unique largest substring of type wwp, where p is some prefix of w and all minimal squares in the word are substrings of wwp with length $2|w|$ (it is easy to see that every such substring is a minimal square, if any of them is). The proof of this statement *for strings of length at most* 120 is an exercise to the reader, their programming skills and their computer. There is a proof that works for all n and it is surprisingly easy to understand, but the margins are too narrow...

Hence, we can compute the answer for n in $O(\text{Good}(n/2)\cdot n+\text{Mini}(n/2)\cdot n^2)$ time, but only $O(\text{Mini}(n/3)n^2)$ memory: compute the number of words without minimal squares of half-length at most $n/3$ and explicitly subtract words that contain only large minimal squares.

Problem Tutorial: "Inv"

A permutation and its inverse have the same number of inversions. Additionally, a permutation is an involution if and only if it is the same as its inverse. So the parity of the number of involutions with given number of inversions is equal to the parity of the number of all permutations with given number of inversions: all non-involutions can be paired up. And the latter number can be calculated by dynamic programming.

The complexity is $O(n^3)$.

Problem Tutorial: "Justice For Everyone"

Firstly, if the answer is nonzero, a_i should have same order with b_i. Without loss of generality, $a_1 < a_2 < \ldots < a_n$ and $b_1 < b_2 < \ldots < b_n$, and $a_i < b_i$ $\forall i$. Also, denote the number of steps by t ($t = \frac{\sum b_i - \sum a_i}{2}$). Let's solve a simpler problem now.

1. Let's forget that numbers should be different all the time. This means that only differences $d_i = b_i - a_i$ matter now. Now the problem can be solved with inclusion-exclusion principle. Let's call the step bad if we increase the same number in it. Then, the answer is $\sum(-1)^s \binom{t}{x_1,\ldots,x_n}\binom{2(t-s)}{d_1-2x_1,\ldots,d_n-2x_n}$, where $s = x_1 + \ldots + x_n$, $x_1 \le \frac{d_1}{2}, \ldots, x_n \le \frac{d_n}{2}$. It means the following: we take firstly all ways to obtain b_i from a_i, even with increasing the same index in one step. Now steps doesn't matter at all, and this number is just $\binom{2t}{x_1,\ldots,x_n}$. Then we decrease it by ways with a given bad step. This means we choose a position of this pair and index which we increase and then again calculate the binomial coefficient. And so on. If you look at it, you can notice that it is a linear combination of coefficients of product of some polynomials $\prod_i P_{n,a_i-b_i}(x)$.

2. Now let's remember the condition about different numbers. Let $f(x_1 \to y_1, \ldots, x_n \to y_n)$ be the number of ways to obtain $(y_1, \ldots, y_n)$ from $(x_1, \ldots, x_n)$ without conditions about different numbers. Then, the answer is $\sum_\sigma \text{sign}(\sigma) f(a_1 \to b_{\sigma(1)}, \ldots, a_n \to b_{\sigma(n)})$, where the sum is taken by all permutations. If we remember that f is a linear combination of product of polynomials, we obtain that the answer is some linear combination of coefficients of determinant of matrix with $P_{n,a_i-b_j}(x)$ in cell (i, j). Now we can calculate the determinant in nM points (where M is the maximal number among a_i, b_i) and interpolate it to obtain the polynomial, and then take the linear combination to get the answer.

The complexity is $O(Mn^4 + M^2n^2)$.

Problem Tutorial: "Keep It Cool"

Suppose we have some permutation of pairs (denote its i-th pair as (x_i, y_i)). Consider the following process: initially we have permutation $1, 2, \ldots, n$, then in i-th step we are swapping numbers x_i and y_i.

It turns out that permutation is balanced if and only if in every step we swap adjacent elements, and finally obtain permutation $n, n-1, \ldots, 1$. Now, the desired number of balanced permutations is equal to the number of ways to obtain permutation $n, n-1, \ldots, 1$ from $1, 2, \ldots, n$ in $\frac{n(n-1)}{2}$ steps using swaps of adjacent elements, in which some pairs of numbers are swapped before others (it corresponds to restrictions). This can be calculated by a simple dynamic programming in $O(n!(n+m))$.

Problem Tutorial: "Lower Algorithmics"

There are multiple possible solutions, but the intended one uses BFS. Let's solve the problem for $\ell = r$ first. Then, the possible sums of ℓ summands from A are exactly the integers we can reach from $\ell \cdot a_1$ in at most ℓ steps, each corresponding to choosing i and adding $a_i - a_1$ to the current sum.

To deal with different number of summands, we need to find out the numbers that can be reached from $m \cdot a_1$ in at most m such steps for each $m \in [\ell, r]$. To do so, we run a BFS up to distance r, but with initial distance to $m \cdot a_1$ being $r - m$. Then, the numbers that we have reached are exactly the required numbers. We can speed up the BFS with bit optimizations. The total complexity is $O(\text{maxa} \cdot \text{maxr} \cdot n/w)$, where w is the size of the machine word.

Note that this solution does not require that the range of possible number of summands is actually consecutive, it could have been arbitrary.

There is another interesting solution, using Fourier transform. Basically, the problem asks us to compute the number of non-zero coefficients in $T^\ell + T^{\ell+1} + \ldots + T^r$, where T is a given polynomial with coefficients 0 and 1. We can choose a FFT-friendly modulo (say, $p = 998\,244\,353$) and compute $T^\ell + T^{\ell+1} + \ldots + T^r$ modulo p. By basic properties of Fourier transform (linearity and conservation of product), $\mathcal{F}(T^\ell + T^{\ell+1} + \ldots + T^r) = \mathcal{F}(T)^\ell + \ldots + \mathcal{F}(T)^r$ (here, $\mathcal{F}$ stands for the Fourier transform of order 2^{21}). To compute the last quantity, we need to simply compute $x^\ell + \ldots + x^r$ separately for all Fourier coefficients of T. Then, we can use inverse Fourier transform to compute $T^\ell + \ldots + T^r$. The only problem is that we computed the number of ways to get some sum modulo p. Hence, we can have false negatives, because the number of ways to get the sum can be divisible by p. To avoid this issue, replace all 1-s in the T by random integers from 0 to $p-1$. Then, we can use Schwartz-Zippel lemma to bound the error probability per target sum by $O(\text{maxr}/p)$. To make the total error probability negligible, compute the answer twice for different choices of random coefficients. Time complexity is $O(\text{maxa} \cdot \text{maxr} \cdot \log(\text{maxa} \cdot \text{maxr}))$.