

Problem A. Acessando montanhas

O problema é feito para enganar e parecer ser um problema de grafo, porém a restrição em que é sempre dado uma montanha de nível menor ser necessária para uma de nível maior facilita bastante a solução, com isso podemos ver que é sempre possível pegar a montanha de nível 1 antes da nível 2 e assim por diante.

Formalmente falando, é sempre possível pegar uma montanha i antes de uma montanha j se $i < j$. Dessa forma a menor ordem lexicográfica são os números de 1 a N ordenados.

É sempre possível ter resposta então a possibilidade de printar -1 também foi colocada para desviar a atenção dos competidores.

Problem B. Baralho Infinito

Os números nas cartas do baralho são representados por potências do número n , assim, o valor de uma jogada é a soma de diferentes potências de n . Como cada potência só pode aparecer uma vez na jogada, temos que o valor da jogada, quando representado na base n , consiste apenas de 0 e 1.

$$s_k = a_0n^0 + a_1n^1 + a_2n^2 + \dots + a_in^i + \dots$$

Onde s_k é o valor do k -ésimo termo na sequência e a_i só pode assumir os valores 0 ou 1.

Dessa forma, podemos observar a correspondência entre a representação de s_k na base n e a representação do número k na base binária. Assim, observamos que o termo a_i será igual a 1 somente se o i -ésimo bit na representação binária do número k for igual a 1.

Logo, basta iterarmos sobre os bits do número k e adicionarmos n^i à resposta caso o i -ésimo bit esteja ativo. A complexidade fica $O(\log(k))$.

Problem C. Collatz-Estrela da Dor e do Sofrimento

A observação chave é: quando $B_{t,i,j} = 1$, $A_{t,i,j} = 1$ e $\text{Prob}(S, t_l, t_r)$, para qualquer S contendo (i, j) e $t_l \leq t \leq t_r$, é $\frac{1}{1}$, que tem valor 1 mod M — não maior que $\lfloor \frac{M}{2} \rfloor$ para qualquer $M > 2$. Podemos verificar que para $B_{0,i,j}$ de 1 a 10^5 , existe um X tal que $B_{X,i,j} = 1$. Mais do que isso, isso vale para $X \leq T_{max}$ pequeno, sendo $T_{max} = 244$. Assim, podemos precalcular todos os $A_{t,i,j}$, $B_{t,i,j}$, $A'_{t,i,j}$ e $B'_{t,i,j}$ com t até T_{max} em custo $O(M \log(M) + HWT_{max} \log(M))$. Isso porque, para achar $NCP(X, M)$ com $X < M$, guardamos os coprimos a M em um vetor e buscamos o resultado da função com busca binária.

Assim, podemos, para cada query, iterar os subgrids de K_i em $O(H^2W^2)$ e, para cada um deles, verificar se atendem à condição imposta pelo enunciado. Se $t_l > t_r$ ou $t_r > T_{max}$, já podemos descartar o subgrid. Caso contrário, precisamos calcular a probabilidade mod M . Para isso podemos usar diretamente os valores das probabilidades $\frac{A'_{t,i,j}}{B'_{t,i,j}} \bmod M$ e uní-las usando a fórmula de união de probabilidade.

No final, nós queremos efetuar a união de probabilidade em um intervalo multidimensional no tempo t e coordenadas i e j (união de $\text{Prob}((i, j, i, j), t, t)$ com $t_l \leq t \leq t_r$, $x_1 \leq i \leq x_2$, $y_1 \leq j \leq y_2$). Podemos observar que a operação de união com as probabilidade mod M forma um monoide. Assim, usaremos uma Disjoint Sparse Table (DiST) multidimensional para efetuar essa operação em intervalo de dimensão D em $O(2^D)$ (nesse caso, $D = 3$). O preprocessamento para isso tem custo $O(H \log(H) W \log(W) T_{max} \log(T_{max}))$.

Ao final, nossa complexidade é $O((M \log(M) + HWT_{max} \log(M)) + (H \log(H) W \log(W) T_{max} \log(T_{max}) + (QH^2W^22^D)))$, sendo a primeira fase o preprocessamento de A e B , a segunda fase o preprocessamento da DiST e a terceira fase o cálculo das queries. Podemos verificar que isso passa no limite com as restrições do input.

Problem D. Delírio na Unballoon

O volume do cone é dado por $\frac{\pi r_C^2 h_C}{3}$ e o volume da semi-esfera por $\frac{2\pi r_C^3}{3}$. Assim, o volume V_C de sorvete suportado pela casquinha é $\frac{\pi}{3}(r_C^2 h_C + 2r_C^3)$.

O volume da garrafa de Lucas Sala é $V_L = \pi r_L^2 h_L$; logo, devemos imprimir “Injusto” se e somente se $V_L > V_C$, que é equivalente a $\pi r_L^2 h_L > \frac{\pi}{3}(r_C^2 h_C + 2r_C^3)$. Simplificando, vemos que isso acontece exatamente quando $3r_L^2 h_L > r_C^2 h_C + 2r_C^3$. Essa última desigualdade pode ser verificada diretamente.

Problem E. Escola Mágica de Palavras

Pela simetria de permutação, os únicos estados que importam são N , quantas letras precisamos adicionar, e K , quantas letras distintas já utilizamos; dessa forma, podemos construir a seguinte DP.

$$DP[N][K] = K * DP[N-1][K] + DP[N-1][K+1]$$

Onde na transição podemos ou adicionar uma letra que já apareceu, ou adicionar uma letra que ainda não apareceu.

Como o máximo de letras distintas que podemos adicionar é 26, a quantidade de estados da DP é $O(N*26)$ e a transição é $O(1)$.

Problem F. Fibonacci

Note que a sequência de Fibonacci módulo M é periódica. De fato, existem apenas M^2 possíveis pares (a, b) de números no intervalo $[0, M-1]$ que podem ser assumidos por termos consecutivos dessa sequência, e um tal par determina unicamente o restante da sequência. Além disso, não é difícil ver que a primeira tal repetição é o par $(0, 1)$. A estratégia para a resolução do problema será encontrar um múltiplo k desse período, que será menor que $5 \cdot 10^{18}$. Denotando por r o resto da divisão de a^b por k , bastará calcular f_r módulo M , utilizando, por exemplo, exponenciação de matrizes.

Denotaremos por $\pi(k)$ o período da sequência de Fibonacci módulo k , em que k é um inteiro positivo qualquer. Perceba que essa função tem a seguinte propriedade: se a e b são inteiros positivos coprimos, então $\pi(ab) = \text{lcm}(\pi(a), \pi(b))$. Assim, basta determinar os valores de π em potências de primos.

Primeiro analisamos o comportamento da função em primos. Fixe então um primo p diferente de 5 (o valor de $\pi(5)$, que é 20, pode ser encontrado de forma ingênua). Assumindo um pouco de familiaridade com teoria dos corpos, demonstraremos que $p^2 - 1$ é um múltiplo de $\pi(p)$. Um cálculo simples demonstra que $\pi(2) = 3$, logo a afirmação vale quando $p = 2$ e podemos assumir que $p \neq 2$.

Suponha que ϕ_1, ϕ_2 são soluções da equação $x^2 = x + 1$ em algum corpo de característica p . Segue imediatamente por indução que

$$\phi_1^n = f_n \phi_1 + f_{n-1}$$

$$\phi_2^n = f_n \phi_2 + f_{n-1}$$

para todo inteiro $n \geq 1$. Subtraindo as equações, obtemos $\phi_1^n - \phi_2^n = f_n(\phi_1 - \phi_2)$. Portanto, se $\phi_1 \neq \phi_2$, temos a fórmula de Binet:

$$f_n = \frac{\phi_1^n - \phi_2^n}{\phi_1 - \phi_2}.$$

Como $p \neq 2, 5$, a fórmula de Bhaskara mostra que podemos tomar $\phi_1, \phi_2 = \frac{1 \pm \sqrt{5}}{2}$ como soluções distintas para $x^2 = x + 1$. Note que ϕ_1, ϕ_2 pertencem a uma extensão K de F_p de grau no máximo 2. Assim, a ordem do grupo multiplicativo de K (denotada por $|K^*|$) é $p-1$ ou $p^2-1 = (p-1)(p+1)$. O teorema de Lagrange e a fórmula de Binet implicam que $\pi(p)$ divide $|K^*|$, que por sua parte divide p^2-1 , como queríamos demonstrar.

Por fim, vale que $\pi(p^k)$ divide $\pi(p) \cdot p^{k-1}$. Isso segue imediatamente de [1, Th. 5].

[1] Wall, D. D. (1960). Fibonacci Series Modulo m. The American Mathematical Monthly, 67(6), 525–532. <https://doi.org/10.2307/2309169>

Problem G. Grafo do Amor

O problema pede que calculemos o matching máximo considerando as arestas do grafo como bidirecionais e desconsiderando arestas de um vértice para si mesmo.

Podemos verificar que o grafo de Iasmim é um grafo funcional. Grafos funcionais são sempre formados por vários componentes, onde em cada componente existe exatamente um ciclo. Neste problema, vamos desconsiderar ciclos de um único vértice. O que temos, em cada componente, é uma estrutura que é quase uma árvore, mas pode conter uma aresta a mais (formando um ciclo). O problema pode ser resolvido de forma independente para cada componente, somando-se a resposta de cada um ao final.

Para resolver o problema em um componente, vamos considerar que saberíamos resolver caso o componente fosse uma árvore (procure por Tree Matching — há uma questão no CSES com esse nome), em complexidade suficiente para os limites deste problema (por exemplo, $O(N)$ ou $O(N\sqrt{N})$). Agora, podemos pensar o seguinte caso haja um ciclo: retirar uma aresta dele ou os vértices dela transforma o componente em uma árvore.

Então, pegue uma aresta qualquer do ciclo. Resolva o problema desconsiderando essa aresta na árvore que sobra — temos o maior matching sem essa aresta. Agora, retire os vértices dessa aresta e resolva novamente o problema na árvore restante — somando 1 ao resultado, temos o maior matching com a aresta. Então o maior matching no componente é simplesmente o maior entre o maior matching sem e com essa aresta qualquer do ciclo.

Problem H. Hora do Impacto

Podemos afirmar que dois triângulos se intersectam se, e somente se, pelo menos uma aresta do primeiro triângulo intersecta uma aresta do segundo triângulo.

Além disso, já que a velocidade é constante, podemos também afirmar que, se duas arestas não se intersectam em um tempo t , mas já se intersectaram em algum tempo anterior a t , então, para qualquer instante posterior a t , essas duas arestas não se intersectarão. Isso ocorre porque a distância entre dois segmentos que se movem com velocidade constante ao longo do tempo descreve uma função convexa.

Assim, podemos concluir que o instante de tempo em que ocorre a colisão entre os triângulos corresponde ao menor tempo em que há interseção entre uma aresta do primeiro triângulo e uma aresta do segundo triângulo. Dessa forma, podemos iterar sobre todos os pares de arestas, uma de cada triângulo, e determinar o menor tempo em que as arestas se intersectam. O problema, então, se reduz a: encontrar o menor tempo em que dois segmentos de retas, que se movem com velocidades constantes, se intersectam.

Para isso, consideramos a função $\vec{l}_1(\lambda_1)$ que parametriza um segmento de reta cujas extremidades são os pontos A e B :

$$\vec{l}_1(\lambda_1) = (B - A)\lambda_1 + A, \quad 0 \leq \lambda_1 \leq 1.$$

Aqui, $\vec{l}_1$ representa um vetor de duas dimensões com as coordenadas de um ponto ao longo do segmento. Sabendo que os pontos se movem com velocidade constante $\vec{v}_1$, definimos a função $\vec{s}_1(\lambda_1, t)$, que descreve o movimento do segmento ao longo do tempo:

$$\vec{s}_1(\lambda_1, t) = \vec{l}_1(\lambda_1) + t\vec{v}_1, \quad 0 \leq \lambda_1 \leq 1, \quad t \geq 0.$$

Fazemos o mesmo para o segundo segmento de reta, considerando os pontos C e D como suas extremidades:

$$\begin{aligned} \vec{l}_2(\lambda_2) &= (D - C)\lambda_2 + C, \quad 0 \leq \lambda_2 \leq 1. \\ \vec{s}_2(\lambda_2, t) &= \vec{l}_2(\lambda_2) + t\vec{v}_2, \quad 0 \leq \lambda_2 \leq 1, \quad t \geq 0. \end{aligned}$$

Os dois segmentos se intersectam em um determinado instante se existir um ponto que pertença simultaneamente a $\vec{s}_1(\lambda_1, t)$ e $\vec{s}_2(\lambda_2, t)$. Dessa forma, a condição para a existência de interseção é:

$$\vec{s}_1(\lambda_1, t) = \vec{s}_2(\lambda_2, t), \quad \text{com } 0 \leq \lambda_1, \lambda_2 \leq 1, \quad t \geq 0.$$

Essa equação representa um sistema de duas equações (uma para cada coordenada) com três variáveis.

Por fim, podemos observar que, quando dois segmentos começam a se intersectar, a colisão inicial sempre contém uma extremidade de um dos segmentos. Assim, basta verificarmos quatro casos: quando $\lambda_1 = 0$, $\lambda_1 = 1$, $\lambda_2 = 0$ ou $\lambda_2 = 1$. Para cada caso, a equação $\vec{s}_1(\lambda_1, t) = \vec{s}_2(\lambda_2, t)$ torna-se um sistema de duas equações com duas variáveis. Portanto, para cada caso, resolvemos o sistema e identificamos o menor valor de t em que ocorre a interseção. O mínimo entre os quatro casos será o menor tempo em que os dois segmentos se intersectam.

Assim, com a resposta para cada par de arestas, podemos determinar o instante em que ocorre a colisão entre os triângulos.

Problem I. Inventando Nomes

Para resolver o problema, basta iterar os $N+1$ lexicograficamente menores nomes possíveis pelas restrições do enunciado e escolher o lexicograficamente menor que não aparece na lista do input. Podemos fazer isso recursivamente: inicia-se com a string vazia, e vamos decidir qual será o próximo caractere a ser adicionado a ela, em ordem de a a z . Se a string depois de adicionar o caractere não está na lista, ela é a resposta, caso contrário, chamamos a recursão considerando a string atual. Se a recursão achou um nome fora da lista, encontramos a resposta, caso contrário, passamos para o próximo caractere e continuamos a busca.

Esse processo é $O(N \cdot X)$, onde X é a complexidade para verificar se uma string está na lista do input. Se usarmos uma Trie contendo os nomes do input, poderemos manter o nó da Trie em que estamos enquanto fazemos a recursão e verificar em $O(1)$ se a string atual está na lista.

Apesar disso, pelas restrições no tamanho total do input, podemos usar um simples set de strings e chamar o método count do set (complexidade total $O(N \cdot K \cdot \log(N))$). Isso funciona porque, pelas restrições das strings do input serem distintas e a soma de seus tamanhos ser limitada, não é possível gerar um caso de teste que faça essa solução passar do limite de tempo. Você pode checar isso gerando pra cada valor de K as N lexicograficamente menores strings até a soma de seus tamanhos não passar do limite.

Problem J. Juntando Xegos

Para uma única consulta, a resposta pode ser resolvida com uma **base-xor** em $O(N \log(\max(A_i)))$.

Sabendo que os elementos A_i podem ser interpretados como vetores em um espaço vetorial sobre $\mathbb{Z}_2$, o subproblema de uma única consulta pode ser resolvido iterando o intervalo $[L, R]$ e verificando se cada elemento (vetor) será ou não adicionado à base desse espaço vetorial.

Por um lado, se um novo elemento for *linearmente independente* de todos os vetores já presentes na base, então esse novo elemento não pode ser derivado de nenhuma combinação linear dos elementos da base e, portanto, precisa ser incluído na **base** desse espaço vetorial.

Por outro lado, se um novo elemento *não for linearmente independente* dos vetores já presentes na base, isso significa que esse elemento pode ser formado por uma combinação linear dos vetores da base e, portanto, é redundante e será incluído no **núcleo** (kernel) desse espaço vetorial.

Com a base-xor de $[L, R]$ computada, responder se um elemento X pode ser formado por um subconjunto desse intervalo é fácil: basta verificar se ele é linearmente independente da base. Se for linearmente independente, então não pode ser formado. Caso contrário, pode. Além disso, pode-se provar que o número total de subconjuntos distintos que podem formar X é dado por $2^{||\text{kernel}||}$, se X não for linearmente independente da base.

A base terá no máximo $\log(\max(A_i))$ elementos em qualquer instante, e para verificar se um novo elemento pode ser adicionado a essa base, basta compará-lo com os vetores já existentes na base. Isso gera a complexidade total de $O(N \log(\max(A_i)))$ para o subproblema de uma única consulta.

Agora, para resolver problemas com múltiplas consultas, podemos usar uma heurística para resolvê-las offline. Ao iterar de $[1, N]$ com o ponteiro r , podemos computar a base-xor para todos os subintervalos que começam em um l (com $l \leq r$) e terminam em r .

Ao iterar um novo elemento r e verificar se ele é linearmente independente da base de todos os subintervalos, podemos começar tentando adicioná-lo ao menor subintervalo primeiro. Se ele for

linearmente independente, adicionamos, seguimos para o próximo menor subintervalo, e continuamos o processo.

Mas, quando esse novo elemento não for linearmente independente do subintervalo considerado, podemos interromper o loop dos subintervalos, porque os próximos subintervalos maiores (que contêm o subintervalo atual) também não precisarão que esse elemento seja adicionado às suas bases.

Com essa heurística, a complexidade final de tempo é $O((N + Q) \log(\max(A_i)))$ e a complexidade total de memória é $O(N \log(\max(A_i)))$.

Observação: sabendo que toda resposta será 0 ou uma potência de 2, é possível pré-computar os valores de potência de 2 módulo MOD com antecedência.

Problem K. Koga precisa de sua ajuda

Se a linha tiver o pokémon Torterra, imprima "Staraptor, eu escolho voce!"; se tiver Staraptor, imprima "Luxray, eu escolho voce!"; se tiver Luxray, imprima "Torterra, eu escolho voce!".

Problem L. Legado de Lagrange

O problema é simples, basta fazer exatamente o que está pedido no enunciado, não há algum truque ou técnica específica necessária aqui.

Problem M. Menor Menor Caminho

Resumo da primeira solução: No DAG de menores caminhos, encontre com programação dinâmica o caminho que minimiza a maior aresta.

Resumo da segunda solução: Encontre o comprimento w do menor caminho e faça busca binária para encontrar o caminho que minimiza a maior aresta.

Detalhamento da primeira solução:

Primeiro montamos um grafo chamado DAG de menores caminhos a partir do grafo G dado na entrada. Esse grafo possui os mesmos vértices que G e contém uma aresta (a_i, b_i, w_i) se e somente se G contém essa aresta e ela está presente em algum menor caminho de 1 até N . Para conferir se está segunda condição vale, fazemos o seguinte: execute um Dijkstra a partir de 1 no grafo G e outro Dijkstra a partir de N no grafo reverso de G (o grafo obtido revertendo a direção das arestas de G). Com isso, obtemos para cada vértice a a distância de 1 até a e a distância de a até N . Logo, para conferir se a aresta (a_i, b_i, w_i) de G pertence ao DAG de menores caminhos, basta verificar se $(\text{distância de 1 até } a_i) + w_i + (\text{distância de } b_i \text{ até } N)$ é igual a distância de 1 até N .

Não é difícil demonstrar que o grafo construído é de fato acíclico e possui a seguinte propriedade: qualquer caminho nesse grafo que começa em 1 e termina em um vértice v é um menor caminho de 1 até v no grafo original. Além disso, todo menor caminho no grafo original existe no DAG de menores caminhos. Dessa forma, basta achar um caminho qualquer que minimize a maior aresta nesse DAG. Para isso, para cada vértice a denote por $f(a)$ o valor mínimo de uma maior aresta no caminho de a até N no DAG de menores caminhos. Definimos também que $f(N) = -\infty$. Se $N(a)$ é o conjunto de vizinhos de a , vale que

$$f(a) = \min_{b \in N(a)} (f(b)).$$

Podemos utilizar essa equação para calcular f recursivamente. Para garantir uma boa complexidade, é necessário memoizar os valores de f . Essa etapa possui complexidade $O(N + M)$. Como precisamos rodar dois Dijkstras, a complexidade final é $O(M + N \log(N))$.

Detalhamento da segunda solução:

Primeiro, encontre o comprimento w do menor caminho de 1 até N com o algoritmo de Dijkstra. Defina o valor de um caminho como sendo o comprimento de sua maior aresta. Utilizaremos uma busca binária para encontrar o menor valor de um menor caminho de 1 até N . Para isso, basta saber conferir para cada c se existe um menor caminho de 1 até N com valor menor ou igual a c . Isso é simples: basta remover as arestas de G com comprimento maior que c e rodar um Dijkstra a partir de 1. Se (e somente se) o valor encontrado para a distância de 1 até N nesse novo grafo também for w , então existe um menor caminho de 1 até N com valor menor ou igual a c .

Problem N. Na zdrowie

Iremos préprocessar todos os intervalos que são permutações. Para cada $1 \leq i \leq n$, denotaremos por $p(i)$ a posição em que i ocorre na lista $a_1, \dots, a_n$. Note que essa posição existe e é única, já que a lista dada é uma permutação.

Inicializamos um inteiro r com o valor $p(1)$, e observamos que o intervalo $[p(1), r]$ é uma permutação. Em seguida, incrementamos ou decrementamos r repetidas vezes até que r atinja o valor $p(2)$. Nesse processo, computamos o maior valor M atingido por a_r , o menor valor L atingido por r e o maior valor R atingido por r . Note que a sublista de b entre L e R possui exatamente $R - L + 1$ inteiros, seu menor elemento é 1 e seu maior elemento é M . Assim, para que essa sublista seja uma permutação, é necessário e suficiente que $M = R - L + 1$. Se esse for o caso, guardamos o fato de que o intervalo determinado por (L, R) é uma permutação.

Após esse préprocessamento, para responder uma query da forma l_i, r_i basta verificar se o intervalo l_i, r_i é uma permutação. Para isso, basta que, na etapa anterior, toda vez que verificamos que o intervalo determinado por L, R é uma permutação, inserimos o par (L, R) em um conjunto. Ao receber a query, verificamos se o par (l_i, r_i) pertence ao conjunto.

A complexidade do préprocessamento é $O(n \log(n))$ devido às inserções no conjunto, e a complexidade da resposta de cada query é $O(\log(n))$, para verificar se um elemento pertence a um conjunto. No total, temos a complexidade $O((n + q) \log(n))$.

Observamos que é possível resolver o problema na complexidade $O(n + q \log(n))$. Para isso, ao invés de utilizar um conjunto, inserimos os pares (L, R) em uma lista e ao final do pré-processamento ordenamos essa lista utilizando o counting sort. Essa ordenação pode ser feita em $O(n)$, devido ao fato de que $1 \leq L, R \leq n$.