Problem A. Árvore Colorida

Esse é um clássico problema em que podemos resolver as queries offline. Primeiro, recebemos todas as queries, calculamos a resposta para cada nó e depois respondemos. Para calcular a resposta para cada nó, faremos uma DFS começando no vértice 1 e criaremos um map em que manteremos a frequência de cada cor no caminho da raiz até o nó atual na DFS. Quando entramos em um vértice novo, somamos 1 na frequência da cor do vértice atual no map, e quando saímos desse nó, subtraímos 1 da frequência dessa cor. Além disso, devemos garantir que todas as cores presentes no map possuem frequência positiva, removendo do map cores que atingirem frequência 0. Assim, a resposta para cada nó será apenas a quantidade de cores no map após processar esse nó na DFS.

Problem B. Beats

Basta implementar as operações exatamente da forma como foram descritas. Por exemplo, para a operação chmod(l, r, x), basta iterar por todos os índices i no intervalo $[l, r]$ e fazer a operação descrita. Assim, cada operação será realizada em $\mathcal{O}(n)$; logo, a complexidade final da solução será de $\mathcal{O}(nq)$. Como $1 \leq n, q \leq 1000$, isso é rápido o suficiente.

Problem C. Caça ao Tesouro

Como o tesouro é escolhido uniformemente ao acaso dentro do retângulo, a probabilidade de ele estar em uma determinada região é proporcional à área dessa região.

A equipe escava uma região em formato de hexágono regular de lado l, completamente contida no retângulo. Portanto, a probabilidade pedida é dada por: $\frac{\text{Área do hexágono}}{\text{Área do retângulo}}$.

A área do retângulo é: $A_r = b \cdot h$.

Um hexágono regular pode ser dividido em 6 triângulos equiláteros de lado l. A área de cada triângulo é: $\frac{\sqrt{3}}{4} \cdot l^2$.

Assim, a área do hexágono é: $A_h = 6 \cdot \frac{\sqrt{3}}{4} \cdot l^2 = \frac{3\sqrt{3}}{2} \cdot l^2$.

Logo, a resposta final é: $\frac{3\sqrt{3}}{2} \cdot \frac{l^2}{b \cdot h}$.

Problem D. Dominando Shurikens

O ponto principal desse problema é a seguinte informação: mesmo com $N = 10^{12}$, temos uma quantidade relativamente pequena de valores diferentes para $\lfloor N/i \rfloor$.

Podemos dividir os valores de i em dois grupos: $i \leq \sqrt{N}$ e $i > \sqrt{N}$. No primeiro grupo, temos $\sqrt{N}$ valores diferentes para i, logo temos no máximo $\sqrt{N}$ valores diferentes para $\lfloor N/i \rfloor$. No segundo grupo, temos que $N/i < \sqrt{N}$, logo temos apenas $\sqrt{N}$ opções de valores para N/i. Então, para um dado N, existem no máximo $2\sqrt{N}$ valores diferentes para $\lfloor N/i \rfloor$.

Ou seja, para $N = 10^{12}$, temos no máximo $2 \cdot 10^6$ valores para $\lfloor N/i \rfloor$.

Podemos criar uma árvore de segmentos com "Lazy Propagation", na qual cada posição se refere a um segmento de posições i com o mesmo valor para $\lfloor N/i \rfloor$. É necessário também guardar o comprimento real de cada segmento, para que possamos calcular o seu valor real para as queries de leitura da soma.

O único problema que resta é que os updates e queries podem definir l e r que não correspondem ao começo e final dos segmentos que definimos, de forma que aplicamos a soma a apenas alguns dos elementos. Para resolver esse problema, devemos dividir nossos segmentos também nessas posições do input. Isso adiciona no máximo $2 \cdot Q$ posições à nossa árvore de segmentos.

Dessa forma, o problema pode ser resolvido em $O(S \log S)$, sendo $S = 2 \cdot (\sqrt{N} + Q)$.

Problem E. EZ

Esse problema é de fato "EZ". A cobra pode sempre comer o rato mais pesado disponível, ou seja, os ratos

com peso N, $N-2$, $N-4$... A soma desses valores pode ser calculada com um loop simples, em $O(N)$, ou com um cálculo matemático, em $O(1)$.

Problem F. Formas de Formar Times

Para que $A(x) \cdot C(x) = B(x)$, é necessário que $C(x) = x^k$, para algum k inteiro não negativo. Ou seja, o conjunto P deve ser igual ao conjunto dos valores de S somados por k. Assim, a resposta do problema pode ser calculada iterando todo k de 0 a $N-1$ e somando a quantidade de formas de escolher S tal que $S' = P$, sendo S' o conjunto dos valores de S somados por k.

Para auxiliar nesse processo, podemos transformar o problema em escolher S tal que $S = P$, com S sendo um conjunto de valores da lista L', na qual $L'_i = L_i + k$. Isso é o mesmo que um conjunto de posições de L' que é o mesmo que o conjunto dos valores dessas posições. Chamaremos tais conjuntos S de válidos, e precisamos contar a sua quantidade.

Considere o grafo funcional induzido por L', onde i tem aresta para $min(N+1, L'_i)$ e $N+1$ tem uma aresta para si mesmo. A estrutura desse grafo é similar a um grafo de permutação, que tem apenas ciclos disjuntos, com a exceção de que há um único componente com uma estrutura de árvore enraizado no $N+1$.

Pela condição para um S ser válido, se i está em S, seria preciso que L'_i também estivesse. Por indução, podemos notar que, se i está em S, seria preciso que todo o componente de i no grafo funcional induzido estivesse em S. Mas $N+1$ não pode estar, por ser uma posição inválida da lista L'. Por isso, os conjuntos S válidos são apenas as uniões dos ciclos do grafo funcional induzido. Se há x ciclos, a quantidade de conjuntos válidos é $2^x - 1$.

A quantidade de ciclos pode ser calculada em $O(N)$ para somar na resposta total. A complexidade final da solução, pela iteração de k, é $O(N^2)$.

Problem G. Grande Maratona de Programação

Para cada equipe, são dados quatro inteiros (a_i, b_i, c_i, d_i). Observe que, após as duas etapas, o número de participantes restantes é $|a_i - b_i|$ e $|c_i - d_i|$. Defina um vetor para cada equipe:

$$p_i = (a_i - b_i,\ c_i - d_i).$$

Se escolhermos um subconjunto S, seja $(X, Y) = \sum_{i \in S} p_i$. Para uma consulta (k, l), a penalidade é:

$$k|X| + l|Y|.$$

Utilizamos a identidade:

$$k|x| + l|y| = \max_{s_1, s_2 \in \{-1,1\}} (s_1 k)x + (s_2 l)y.$$

Assim, podemos considerar quatro direções $d = (\pm k, \pm l)$ e calcular:

$$\max_S \left(\sum_{i \in S} p_i \right) \cdot d.$$

Dessa forma, queremos maximizar:

$$\sum_{i \in S} (p_i \cdot d).$$

Isso é maximizado quando escolhemos todos os pontos tais que:

$$p_i \cdot d > 0.$$

Portanto, o subconjunto de maior valor consiste exatamente em todos os pontos que estão no semiplano definido por d.

Simulamos esse processo inserindo todos os pontos e todas as direções das consultas (quatro por consulta), e ordenando-os por ângulo.

Após a ordenação, os pontos com produto escalar não negativo com uma direção d formam um segmento contíguo de tamanho no máximo 180°.

Processamos tudo em ordem circular usando dois ponteiros para manter uma janela que contém exatamente os pontos com $p_i \cdot d \geq 0$.

Para calcular a soma de forma eficiente, mantemos uma soma de prefixos dos pontos.

Complexidade: $O((n+q)\log(n+q))$.

Tome cuidado com o ponto $(0,0)$ **na ordenação angular.**

Problem H. Hora da Aula

Para que X divida A, o expoente de cada primo na fatoração de X deve ser menor ou igual ao seu expoente na fatoração de A. Quando multiplicamos A por X, somamos os expoentes da fatoração de X aos expoentes da fatoração de A. Quando elevamos A a X, multiplicamos por X cada expoente da fatoração de A.

Podemos guardar uma lista com os expoentes da fatoração de A (inicialmente composta apenas por 0's) e atualizá-la conforme as queries do tipo 1 e 2.

Queries do tipo 1 são processadas rapidamente se soubermos a fatoração de X, já que um número até 10^5 é divisível por no máximo 6 primos distintos e apenas esses terão seus expoentes modificados.

Similarmente, queries do tipo 3 são processadas rapidamente se apenas verificarmos os expoentes dos primos que dividem X.

Para queries do tipo 2, precisamos multiplicar os valores dos expoentes de todos os primos, o que resultaria em TLE para muitas queries desse tipo. Mas números até 10^5 assumem expoentes pequenos em suas fatorações (o expoente máximo para o fator 2, por exemplo, é 16). Podemos então manter um conjunto com apenas os primos cujos expoentes são diferentes de zero e menores que seus máximos, e atualizar apenas os expoentes desses primos. Como cada primo só pode ser processado por queries do tipo 2 até 4 vezes antes de ser removido desse conjunto, essas queries são processadas rapidamente.

Problem I. Inventando Comunicação

O enunciado garante que os casos de teste possuirão N e V gerados aleatoriamente. Com isso, para uma dada subsequência A do vetor, podemos considerar que seu hash $H(A)$ é também um resultado aleatório, assim como o hash $H(A^c)$ de seu complemento.

Seja $M = 998244353$ (o módulo). Existem M^2 valores possíveis para o par $(H(A), H(A^c))$, e em exatamente $\dfrac{M \cdot (M-1)}{2}$ deles $H(A)$ é menor do que $H(A^c)$, sendo estes os que resultam que A atende à condição. Considerando que todos esses pares são equiprováveis devido à aleatoriedade, temos uma probabilidade de $\dfrac{M \cdot (M-1)}{2 \cdot M^2}$ de uma subsequência qualquer A atender à condição. Como M é grande, isso é extremamente próximo de $\dfrac{1}{2}$, portanto, iremos considerar esta probabilidade em nossos cálculos a partir de agora.

Como a probabilidade de uma subsequência qualquer atender à condição é praticamente $\dfrac{1}{2}$, a probabilidade de não atender pode ser estimada também como $\dfrac{1}{2}$. Dessa forma, a probabilidade de a i-ésima menor subsequência lexicográfica ser a resposta a ser impressa é de $\dfrac{1}{2^i}$, porque isso exige que todas as anteriores não atendam e ela atenda à condição. Como essa probabilidade fica exponencialmente pequena com o crescimento de i, podemos enxergar que podemos testar manualmente poucas subsequências dentre as lexicograficamente menores até encontrar uma resposta.

O problema se torna então iterar as subsequências do vetor em ordem lexicográfica e testar cada uma manualmente. Para facilitar esse processo, podemos fazer uma suposição adicional de que a subsequência que é a resposta terá conterá poucos elementos — estimando que a probabilidade de conter exatamente k elementos é $\frac{1}{2^k}$ ou pior, já que existem pelo menos k subsequências menores lexicograficamente (seus prefixos). Dessa forma, podemos iterar as subsequências de tamanho k em ordem lexicográfica, para todo k até um valor pequeno (por exemplo, 20), até encontrar alguma que atenda à condição (o que deve acontecer rapidamente). Uma forma bastante conhecida de fazer essa iteração é usando uma função recursiva.

Por fim, dentre as subsequências de tamanho fixo que encontramos (as lexicograficamente menores entre seu tamanho), escolhemos a lexicograficamente menor como resposta. Assim, a probabilidade de a resposta não ser uma das que testamos é pior do que de $\frac{1}{2^K}$, onde K é o maior valor de k testado. Além disso, a probabilidade da iteração para um tamanho fixo testar mais do que i subsequências é próxima de $\frac{1}{2^i}$, o que mostra que o algoritmo deve rodar em tempo próximo de linear para cada k, devido principalmente ao custo computacional de calcular os hashes para a verificação da condição. Com isso, podemos ter confiança de que a solução irá se mostrar correta e eficiente o bastante para casos de teste gerados aleatoriamente.

Problem J. Jantar no RU???...

Arthur deve escolher como os valores de X todos os números primos até N para não jantar no RU em nenhum dos N dias.

É claro que todo número primo até N deve ser escolhido, porque números primos só são múltiplos de 1 e deles mesmos. Além disso, todo número não primo é múltiplo de algum número primo; então, ao fazer a escolha de todos os primos, Arthur de fato não irá jantar no RU em nenhum dos N dias. Usar qualquer número não primo como valor de X seria redundante, o que mostra que essa é a resposta mínima.

Para contar quantos números primos existem até N, pode-se usar o algoritmo do Crivo de Eratóstenes:

Inicialize um array P com N posições, inicialmente todas iguais a 1. Agora, iteramos todos os números i de 2 a N. Se $P_i = 0$, então i não é primo. Caso contrário, i é primo, e incrementa-se a resposta. Além disso, se for esse o caso, iteramos pelos múltiplos j de i até N marcando $P_j = 0$, para que nas próximas iterações não deixemos de considerá-los como não primos.

A complexidade desse algoritmo é $O(Nlog(log(N)))$. Uma explicação mais completa pode ser encontrada em: `https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html`

Problem K. Kalculadora de Dano

Este problema pode ser implementado com condicionais: podemos verificar se (T_a, T_d) caem no primeiro caso da definição de M ou não, e multiplicar o poder P por 2 ou $\frac{1}{2}$ de acordo. Em seguida, fazemos $X = max(0, H - P)$ (com P já multiplicado). Depois, se $X = 0$, imprimimos "Nocaute"; caso contrário, imprimimos a outra mensagem.

Problem L. Lady Gaga e o Coeficiente Ecoante

Seja L a lista que representa a letra da entrada, com L_i sendo a string que representa o i-ésimo verso. N é o tamanho de L. Seja K o tamanho máximo de uma string de L.

Podemos construir o Autômato de Sufixos dessa lista. Isso pode ser feito usando o algoritmo usual de construção do autômato, considerando a lista L como uma string cujo alfabeto são as palavras de até K letras, em $O(NK \log N)$. A partir do autômato, podemos usar a sua árvore de links para encontrar os Segmentos Ecoantes da letra.

Para isso, iremos iterar as classes de equivalência de L e processar o seu conjunto *endpos* (posições em que uma ocorrência de uma string da classe termina) completo usando a técnica de Small to Large. Isso

reduz o problema para: atualizar a resposta considerando uma posição do *endpos* desta classe com base no estado atual do conjunto (processo denominado query) e atualizar o estado do conjunto com base em uma nova posição *endpos* dessa classe (processo denominado update).

Considere uma classe que possui uma ocorrência terminando em $r2$ e outra em $r1$ ($r2 > r1$). Seja $s = r2 - r1$. Se s for menor ou igual ao tamanho máximo da classe, então existe uma sublista de L terminando em $r2$ de tamanho $2 \cdot s$ que é um Segmento Ecoante. Dessa forma, o valor desse Segmento Ecoante seria $(2 \cdot s) \cdot \sum_{k=r1-s+1}^{r2} |L_k| = (2 \cdot s) \cdot 2 \cdot \sum_{k=r1+1}^{r2} |L_k| = 4 \cdot (s) \cdot \sum_{k=r1+1}^{r2} |L_k|$.

Para auxiliar, considere o vetor P tal que $P_i = \sum_{k=1}^{i} |L_k|$. Então a fórmula pode ser escrita como $4 \cdot (s) \cdot (P_{r2} - P_{r1}) = 4 \cdot (r2 - r1) \cdot (P_{r2} - P_{r1})$. Uma forma ainda mais conveniente pode ser escrevê-la como $4 \cdot (r2 \cdot P_{r2} - r2 \cdot P_{r1} - r1 \cdot P_{r2} + r1 \cdot P_{r1})$. Isso permite visualizar que, para um certo $r2$, basta saber, para todo $r1$ presente no *endpos* de $r2$ a $r2 - M$ (sendo M o tamanho máximo da classe), a quantidade desses $r1$s, a sua soma, a soma de seus P_{r1} e a soma de $r1 \cdot P_{r1}$. A análise para $r1$ é análoga.

Dessa forma, iremos manter estruturas para calcular e atualizar essas somas de intervalo rapidamente — por exemplo, quatro instâncias de uma BIT / Fenwick Tree. Assim, o processo de update fica reduzido aos updates das BITs. Já na query, basta considerar a posição atual como $r1$ e $r2$ e verificar a sua contribuição para a resposta total. Como cada operação em uma BIT é $\log N$, a complexidade total desse processo fica $O(N \log^2 N)$.

A complexidade total da solução é $O(NK \log N + N \log^2 N)$, que passa confortavelmente no limite de tempo.

Problem M. Motorista Impaciente

A soma do tempo total gasto para selecionar uma vaga com a distância da vaga finalmente selecionada será chamada de **penalidade**.

Para cada $i \in [1, n]$, seja $f(i)$ o valor esperado da penalidade, dado que Henrique começou o processo a partir da vaga i (agindo de maneira ótima). O que queremos calcular é $f(1)$. Se $S(i)$ é a próxima vaga a partir de i (ou seja, $S(i) = i + 1$ se $i \neq n$ e $S(n) = 1$), temos

$$f(i) = p_i \min(1 + f(S(i)), d_i) + (1 - p_i)(1 + f(S(i)))$$

.

Definindo

$$s_i(x) = p_i \min(1 + x, d_i) + (1 - p_i)(1 + x),$$

temos $f(i) = s_i(f(S(i)))$, logo $f(1) = (s_1 \circ s_2 \circ \cdots \circ s_n)(f(1))$. Denotando a composição $s_1 \circ \cdots \circ s_n$ por s, temos $f(1) = s(f(1))$.

Analisaremos o comportamento da função $s = s_1 \circ s_2 \circ \cdots \circ s_n$ para demonstrar que ela possui exatamente um ponto fixo. Como já vimos que $f(1)$ é um ponto fixo de s (ou seja, $f(1) = s(f(1))$), bastará achar esse ponto fixo, o que pode ser feito com uma busca binária para encontrar um zero de $s(x) - x$, já que essa função é claramente contínua.

Note que, para $x \leq d_i - 1$, temos $s_i(x) = 1 + x$. Já para $x \geq d_i - 1$, temos $s_i(x) = p_i d_i + (1 - p_i)(1 + x)$, que é uma reta com inclinação $1 - p_i \in (0, 1)$. Assim, vemos que, para valores suficientemente pequenos de x, $s(x) = x + n$. Depois disso, $s(x)$ é linear por partes, formada por várias retas com inclinações no intervalo $(0, 1)$. Assim, a função dada por $s(x) - x$ é inicialmente constante e igual a n, e depois é dada por várias retas com inclinações no intervalo $(-1, 0)$. Ou seja, $s(x) - x$ é inicialmente constante e positiva, e depois é estritamente decrescente, eventualmente sendo igual a uma reta com inclinação negativa. Como ela é contínua, isso prova que ela tem exatamente uma raiz, logo s tem exatamente um ponto fixo, como queríamos.

Já que podemos calcular $s(x) - x$ de maneira ingênua em $\mathcal{O}(n)$, utilizando uma busca binária conseguimos achar o seu único 0 com precisão ϵ em $\mathcal{O}(n \log(1/\epsilon))$.

Problem N. Não Ter Medo de Cair

O problema pede, para todo i de 1 a M, a soma dos valores (ratings) maiores ou iguais a i.

Note que a soma dos maiores ou iguais a i é igual à soma dos maiores ou iguais a $i+1$ mais a soma dos iguais a i. Para resolver o problema, usaremos os arrays auxiliares f e ans. Iremos armazenar, em f_i, a quantidade de problemas com rating i, e usaremos isso para armazenar a resposta pedida pelo enunciado para i em ans_i.

Vamos iterar todo i de M até 1 (decrescendo i): a resposta para i é apenas a resposta para $i+1$ (já calculada anteriormente e armazenada em ans_{i+1}) somada de $i \cdot f_i$ (soma dos ratings dos problemas iguais a i — apenas o produto do rating pela quantidade de problemas correspondentes). Depois, basta iterar i de forma crescente e imprimir a resposta em ans_i.

A complexidade dessa solução é $O(N+M)$.