Tutorial: Areias da Anarquia

Daniel Saad Nogueira Nunes

Podemos utilizar uma técnica de programação dinâmica chamada binary-lifting para resolver esse problema. Esta técnica visa computar qual o ancestral a uma distância 2^i de um nó u qualquer. Seja u um nó e p o seu pai obtido durante uma busca em profundidade. Modelando como uma relação de recorrência temos:

$$up(u,i) = \begin{cases} p, & i = 0 \\ up(up(u,i-1),i-1), & i > 0 \end{cases}$$

Essa tabela pode ser preenchida durante uma busca em profundidade e, para cada nó u, gasta-se tempo $\Theta(\lg n)$ para isso.

Utilizaremos uma tabela extra para computar, além do ancestral a uma distância 2^i, a aresta de maior custo que liga o nó u ao seu ancestral:

$$up_w(u,i) = \begin{cases} w, & i = 0, \text{ em que } w \text{ é o custo da aresta que liga } u \text{ ao seu pai } p \\ up_w(u,i) = \max(up_w(u,i-1), up_w(up(u,i-1),i-1)), & i > 0 \end{cases}$$

Esta tabela também pode ser preenchida em tempo $\Theta(\lg n)$ para cada nó.

Com as tabelas up e up_w, é possível responder as consultas. Usamos as seguintes observações:

- Se a é ancestral de b: em outras palavras, a é o ancestral comum mais baixo de a e b. Precisamos só escalar a partir de b usando up e up_w até alcançar b.
- Se b é o ancestral comum mais baixo de a, podemos trocar a e b e o argumento é idêntico ao anterior.
- Se b e a possuem um ancestral comum c, escalamos a até chegar em c e depois escalamos b até chegar em c, guardando o custo da maior aresta.

Para identificar se um nó é ancestral de outro, basta examinar os tempos de entrada e saída da busca em profundidade. a é ancestral de b quando $t_{in}[a] \leq t_{in}[b]$ e $t_{out}[a] \geq t_{out}[b]$.

A pergunta que fica é: como escalar a árvore eficientemente? Como achar um ancestral comum rapidamente? A resposta é: damos o maior salto possível de tamanho 2^j sem que esse salto ultrapasse o ancestral comum. Em outras palavras: achamos o maior valor de j tal que $x = up(a,j)$ não seja um ancestral comum de a e b. Em seguida, movemos para esse x e continuamos o processo até que não seja mais possível. O algoritmo é correto, pois qualquer distância de escalada pode ser decomposta em uma soma de potências de dois. Sempre que usamos a tabela up, olhamos para up_w para atualizar a maior aresta do caminho se necessário. Como para cada nó a, olhamos para no máximo $\lg(n)$ entradas, então cada consulta pode ser respondida em tempo $\Theta(\lg n)$.

Custo total do algoritmo: $\Theta(n \lg n + q \lg n)$.

Tutorial: Bravo, bravo!

Guilherme Ramos

A solução é simples, basta ler cada mensagem fornecida como entrada e repeti-la na saída. Para as duas primeiras, após apresentar cada estrepolia, acrescente uma mensagem *"Bravo, bravo!"* (como 2^a e 4^a linhas da saída). Leia e apresente a manobra arriscada. Por fim, basta repetir a última mensagem (a reação da platéia).

Esta questão foi baseada na fantástica música *Piruetas de Chico Buarque.*

Tutorial: Cabo Carente

Daniel Porto

Uma solução é armazenar os comprimentos dos cabos em uma lista, ordenar a lista ir conectando os dois menores cabos existentes. Cada nova conexão gera um novo cabo, que substitui os dois cabos usados. Esse novo cabo deve ser inserido na posição correta de forma que a lista de cabos continue ordenada. Esse processo deve ser repetido até que só haja o "supercabo" final.

Tutorial: Divisores

Edson Alves

Se $D = 1$, a resposta será $n = 1$. Se $n > 1$, o Teorema Fundamental da Aritmética nos diz que n pode ser escrito como

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \dots p_r^{\alpha_r},$$

onde r é um inteiro positivo e p_i é primo, para $i = 1, 2, \dots, r$, com $p_i \neq p_j$ se $i \neq j$.

Como τ é uma função multiplicativa e $\tau(p^n) = (n + 1)$, para qualquer primo p, então

$$\tau(n) = (\alpha_1 + 1)(\alpha_2 + 1) \dots (\alpha_r + 1)$$

Escrevendo $D = d_1 d_2 \dots d_k$, com $d_i > 1$ para todo $i = 1, 2, \dots, k$, para determinar um positivo n tal que $\tau(n) = D$ basta fazer

$$n = p_1^{d_1 - 1} p_2^{d_2 - 1} \dots p_k^{d_k - 1}$$

Para minimizar o valor de n é preciso fazer duas coisas:

1. Tornar $p_1, p_2, \dots, p_k$ os k primeiros primos em ordem crescente; e
2. Avaliar todas as possíveis decomposições de D como $D = d_1 d_2 \dots d_k$, com $d_i > 1$ para todo $i = 1, 2, \dots, k$ e $d_i \geq d_{i+1}$ para $i = 1, 2, \dots, k - 1$.

Como $D \leq 10^9$, ele será composto $r < 30$ fatores primos, não necessariamente distintos, e as decomposições estão associadas às partições de r (a justificativa e a relação entre a decomposição e uma partição fica a cargo do leitor). Como 30 tem 5604 partições e cada partição pode ser avaliada em $O(r)$, a solução apresentada atende o limite de tempo do problema.

Tutorial: Estacionando na Terra

Jeremias Moreira Gomes

Para calcular a menor distância no pior caso possível, no caso trivial calcula-se o vetor de menores distâncias D a partir de todos os planetas para todos os outros planetas, e a resposta é dada por $max(1 \to D[i] + D[i] \to n)$, que indica o menor caminho a partir de Raxacoricofallapatorios até o planeta de pior caso possível $D[i]$ e a partir desse planeta até a Terra. Essa solução pode ser gerada a partir da execução de um floyd-warshall, porém não é suficiente para passar no tempo. Para otimizar, pode-se calcular um djikstra DR a partir da origem (Raxacoricofallapatorios) e outro djikstra DT a partir do final (Terra) e a resposta será dada por $max(DR[i] + DT[i])$.

Tutorial: Fitoplâncton

Edson Alves

O problema consiste em determinar uma string R tal que $S = R^M$, com o maior M possível. Primeiramente, observe que, para $M = 1$, $R = S$ é uma solução.

Suponha que exista uma string R, com $M > 1$, tal que $S = R^M$. Podemos escrever

$$S^2 = SS = (R^M)(R^M) = RR^M(R^{M-1}) = RSR^{M-1}$$

Neste caso, S é uma substring de S^2 que não é nem prefixo e nem sufixo. Assim, basta identificar, usando um algoritmo eficiente (por exemplo, *z-function*), o índice i da segunda ocorrência de S em S^2 (S ocorre, no mínimo, duas vezes em S^2, como prefixo e sufixo): se $i < |S|$ (isto é, S ocorre em uma posição diferente do sufixo), então a resposta será $M = i$ e $R = S[0..(i-1)]$, assumindo que os índices são contados a partir de zero.

Esta solução tem complexidade igual a do algoritmo utilizado (no caso da *z-function*, $O(|S|)$).

Tutorial: g3r3mias

Guilherme Ramos

Problema baseado no famoso "Estraga-Festa", do Prof. Daniel Saad na `IV Maratona de Programação do IFB`.

Basta descobrir qual equipe possui a maior quantidade de balões (b) e em caso de empate, o menor tempo acumulado (t). A resposta é "$b + 1$ t" caso $b < P$ ou "b $t - 1$" caso contrário.

Tutorial: Homem que Ordenava

João Henrique Souza Pereira

A solução é simples, basta contar a quantidade de árvores de cada altura e apresenta-las em ordem. Uma solução com *counting sort* pode ser aplicada em $O(n)$.

Tutorial: Império Quadradônico

José Leite

Em breve!

Tutorial: Jotinha e string

Alberto Neto

Em breve

Tutorial: Kátia e as árvores

Edson Alves

Como o grafo da entrada é não-direcionado e conectado, com N arestas, para obter uma árvore é preciso remover uma aresta sem perder a conectividade do grafo. Contudo, o grafo tem um ciclo (por isso não é árvore), então basta identificar o ciclo por meio de uma travessia (BFS ou DFS) e excluir qualquer uma das arestas que faz parte do ciclo.

A complexidade desta solução é $O(N)$.

Tutorial: Logística

Arthur Botelho

Para uma dada região, o melhor k a ser escolhido para minimizar o total de caixas é o máximo divisor comum (gcd) das quantidades de itens de cada pacote ($a_{i,j}$). Dessa forma, sendo $gcd(A, B, C, D)$ o gcd de todos os $a_{i,j}$ na região $((A, B), (C, D))$, e $soma(A, B, C, D)$ a soma desses mesmos $a_{i,j}$, o custo mínimo para uma região $((A, B), (C, D))$ é apenas $soma(A, B, C, D)/\gcd(A, B, C, D)$.

Se soubermos calcular rapidamente *soma* e gcd, podemos iterar todas as possibilidades de regiões possíveis, que são $O(|x|^2 \cdot |y|^2)$, e calcular a soma dos seus custos mínimos. Depois disso, dividimos o resultado (mod 998244353) pela quantidade de regiões possíveis e temos a média.

É possível calcular $soma(A, B, C, D)$ em $O(1)$ após um pré-processamento $O(n \cdot m)$ usando a técnica de Prefix Sum 2D. Além disso, com um pré-processamento $O(n \cdot m \cdot log(n) \cdot \log(m))$, é possível também calcular $\gcd(A, B, C, D)$ em $O(1)$ usando a estrutura Sparse Table 2D.

Assim, a complexidade de tempo dessa solução é $O(n \cdot m \cdot \log(n) \cdot \log(m) + |x|^2 \cdot |y|^2)$, e a memória usada é $O(n \cdot m \cdot \log(n) \cdot \log(m))$ devido à Sparse Table.

Como alternativa, também pode-se comprimir, em $O(n \cdot m)$, a matriz original em uma nova matriz menor que guarda dois valores simultaneamente, representando uma soma e um *gcd*. Podemos criar uma nova matriz b onde $b_{2i,2j} = (a_{x_i,y_j}, a_{x_i,y_j})$, ou seja, as posições de b com ambas as coordenadas pares guardam o valor de uma posição de a com coordenadas em x e y. Além disso, em $b_{2i+1,2j}$ $(1 \leq i \leq |x|, 1 \leq j \leq |y|)$ guardaremos a soma e o *gcd* entre todas as posições entre a_{x_i,y_j} e a_{x_{i+1},y_j} na matriz original (e de forma análoga para $b_{2i,2j+1}$). Por fim, em $b_{2i+1,2j+1}$, a soma e o *gcd* entre $a_{k,l}$ com $x_i \leq k \leq x_{i+1}$ e $y_j \leq l \leq y_{j+1}$.

Após essa compressão, podemos calcular tanto *soma* quanto gcd enquanto iteramos as possibilidade de regiões $((A, B), (C, D))$. Escolhemos (A, B), e iteramos todas as linhas e colunas abaixo da posição correspondente na matriz b, tomando pares crescentes de (C, D). Seja (K, L) a posição de (A, B) em b e (I, J) a posição de (C, D) em b. Ao iterar (I, J), pode-se obter tanto *soma* quanto gcd da submatriz $((K, L), (I, J))$ em $O(1)$ usando os valores calculados previamente para $((K, L), (I-1, J))$ e $((K, L), (I, J-1))$ (técnica de soma/*gcd* em prefixo 2D). Depois, se I e J são ímpares, usamos os valores atuais de *soma* e gcd para atualizar a soma dos custos mínimos.

Essa solução possui complexidade de tempo $O(n \cdot m + |x|^2 \cdot |y|^2)$, e usa apenas $O(n \cdot m)$ de memória.

Tutorial: Maturação de Queijos

Vinicius Borges

O problema pode ser resolvido por meio de uma abordagem baseada em programação dinâmica.

Primeiramente, deve-se ordenar a estrutura que armazena os instantes de tempo inicial e final dos queijos em ordem crescente ao relação ao tempo final. Seja `dp` uma tabela $N \times K$ responsável pela memoização, em que $dp[i][j]$ representa a quantidade máxima de queijos que podem ser feitos com os primeiros i queijos a um custo máximo j.

Para cada queijo i, deve-se verificar se ele pode ser incluído no hall dos queijos que serão maturados sem sobreposição e sem exceder os recursos financeiros j. Além disso, deve-se procurar pelo último queijo que pode ser feito antes do início do próximo queijo a ser analisado.

O seguinte algoritmo detalha as operações acima:

1. dp[0 ... N][0 ... K] $\leftarrow 0$
2. `PARA` $i \leftarrow 1, \ldots, N$
3. `PARA` $j \leftarrow 0, \ldots, K$
4. dp[i][j] $\leftarrow$ dp[i-1][j]
5. `SE` j $\geq$ queijos[i-1].custo
6. `PARA` $k \leftarrow i - 1, \ldots, 1$
7. `SE` dp[k-1][1] $\leq$ queijos[i-1].inicio:
8. dp[i][j] $\leftarrow \max(dp[i][j], dp[k][j - queijos[i-1].custo] + 1)$
9. resposta $\leftarrow dp[N][K]$

Primeiramente, é necessário ordenar os queijos em relação aos seus tempos de finalização do processo de maturação, resultando em uma complexidade $O(N \log N)$, em que N é a quantidade de queijos. A análise da parte de programação dinâmica mostra os loops aninhados executam $N \cdot K \cdot N$ iterações, totalizando uma complexidade de $O(N^2 \cdot K)$.

A tabela irá demandar um espaço de memória $(N + 1) \cdot (K + 1)$, o que resulta em uma complexidade de memória de $O(N \cdot K)$.

Tutorial: Nave Extraterrestre

Vinicius Borges

Pode-se verificar que o padrão fornecido deve ser armazenado em uma matriz bidimensional $N \times M$, pois devemos acessar elementos de posições adjacentes para cada posição analisada.

O problema deve ser resolvido verificando se cada caractere na posição (i, j) possui cor igual à posição $(i+1, j+1)$, e cores iguais $(i, j+1)$, $(i+1, j)$. Nesse caso, para percorrer a matriz, considere $i = 0, \ldots, N-1$ e $j = 0, \ldots, M-1$.

A complexidade deste código é $O(NM)$, devido à necessidade de se acessar todas as posição abaixo (ou acima) para cada posição do padrão.

Tutorial: O Jogo Aleatório

Arthur Botelho

Para um dado tamanho de tabuleiro k, a quantidade de fotos possíveis de serem tiradas corresponde à quantidade de configurações que $T(k)$ pode assumir ao final do jogo. Podemos abstrair o caráter aleatório do jogo e pensar em quantas disposições do tabuleiro podemos formar ao final movendo as peças qualquer número de posições para trás ou removendo-as do jogo. Vamos desconsiderar que as peças são numeradas e controladas por um jogador em específico, porque para o problema só importa a posição em que elas terminam.

Imagine que as peças removidas após se moverem na posição 1 são na verdade postas em uma posição imaginária 0, e que ela também inicia com uma peça. Assim, o tabuleiro ficaria com $k+1$ posições $T(k)_i$ com i de 0 a k. É possível ver que a maior quantidade de peças que podem terminar na posição $T(k)_i$ é $k-i+1$ (a quantidade de peças de i a k).

Agora, imagine a lista ordenada D tal que a frequência do número i é a quantidade de peças na posição $T(k)_i$. Isso é o mesmo de pegar as "distâncias" de todas às peças à posição 0 e ordená-las. Vamos ter como resultado uma lista crescente D que segue uma restrição específica: todo número na posição D_i $(0 \leq i \leq k)$ dessa lista está entre 0 e i — caso contrário, teríamos que ser capazes de mover peças para frente. Portanto, toda configuração final do tabuleiro é unicamente identificada por uma lista D desse tipo.

Além disso, toda lista D desse tipo pode ser obtida a partir de uma configuração final do tabuleiro. Basta colocar, na posição i do tabuleiro $T(k)$ (já incluindo a posição imaginária 0), uma quantidade de peças igual à frequência do número i em D. Como $0 \leq D_i \leq i$ para $0 \leq i \leq k$, isso é sempre possível, porque a frequência de um número x é no máximo $k-x+1$ de acordo com essa restrição (e estamos colocando exatamente $k+1$ peças). Podemos ver que isso está de acordo com as restrições para a disposição das peças ao final do jogo.

Dessa forma, precisamos saber quantas listas crescentes D de tamanho $k+1$, indexadas em 0, existem tais que $0 \leq D_i \leq i$. Isso é o mesmo que contar quantas listas D' de tamanho $k+1$, indexadas em 1, existem tais que $1 \leq D'_i \leq i$. Esse problema é um dos múltiplos problemas análogos que podem ser resolvidos pela sequência dos números de Catalan. Ou seja, para um dado k, a quantidade de fotos diferentes é igual ao $k+1$-ésimo número de Catalan. Vamos chamar de C_n o n-ésimo número de Catalan, que é expresso por $\binom{2n}{n} - \binom{2n}{n+1}$.

O problema pede quantas sequências de fotos S' podemos formar tal que a foto S'_i seja tirada após um jogo com $k = B_i$ e que as fotos são distintas par a par. Por definição, para que duas fotos S'_i e S'_j sejam iguais, é necessário que $B_i = B_j$ — e para para um dado tamanho k, sabemos quantas possibilidades de fotos distintas existem. Essa independência garante que a resposta do problema é apenas o produto das respostas para toda subsequência de B tal que todos os elementos dessa subsequência possuam o mesmo valor k, desde que subsequências diferentes possuam valores k diferentes. Ou seja, o problema é reduzido para: dado um k e um tamanho x (que é a frequência de k em B), saber quantas listas de tamanho x de fotos distintas par a par, todas tiradas após partidas em $T(k)$, podem ser formadas. Para quaisquer inteiros positivos N, X, a quantidade de listas de tamanho N distintas par a par em que cada elemento pode assumir X valores é apenas $arranjo(X, N) = X!/(X-N)! = \prod_{i=0}^{N-1}(X-i)$. O caso de $X < N$ é tratado por esta fórmula.

Seja F_k a frequência do valor k em B (que é equivalente ao tamanho da lista a ser formada na redução do problema). Como o maior valor de k possível, denotado por M_k, é 10^6, podemos calcular a resposta para o problema simplesmente como $\prod_{k=1}^{k=M_k} \prod_{i=0}^{F_k-1}(C_{k+1} - i)$. Todos os cálculos devem ser feitos mod 10^9+7.

Para otimizar o cálculo, podemos pré-computar os valores de $n!$ e $1/n!$ para n suficientemente grande (pouco mais que $2M_k$). Isso nos permite calcular $\binom{N}{K} = N!/((N-K)!K!)$ (e, por extensão, os números de Catalan) em $O(1)$. Depois disso, podemos calcular o produtório duplo descrito, que é $O(n + M_k)$.