Tutorial: Águas Claras

Adne Moretti Moreira

Nesse problema, queremos determinar, para cada prédio, quantos prédios à esquerda permanecem visíveis, isto é, quantos não são tampados por prédios mais altos ou de mesma altura entre eles, considerando também adições de andares já programadas em prédios construídos.

Uma abordagem ingênua seria, a cada operação de adição de andares, percorrer todos os prédios já construídos e atualizar suas alturas. No pior caso, isso levaria a uma complexidade $O(N^2)$, o que é inviável.

Para evitar esse custo, utilizamos a ideia de acumular as atualizações em uma variável global. Mantemos uma variável x que representa o total de andares adicionados até o momento. Assim, ao invés de atualizar explicitamente cada prédio, consideramos que sua altura real é dada por:

$h_{real} = h_{armazenada} + x$

Quando um novo prédio é construído com altura h, armazenamos o valor $h - x$. Dessa forma, conseguimos recuperar sua altura real a qualquer momento sem precisar modificar os valores anteriores.

Além disso, para calcular eficientemente quais prédios permanecem visíveis, uma abordagem é a utilização de uma pilha monótona decrescente.

A ideia central da pilha monótona é manter apenas os prédios que ainda podem ser visíveis no futuro. Sendo assim, ao inserir um novo prédio, removemos do topo da pilha todos os prédios com altura menor ou igual à dele, pois eles serão completamente "tampados" por esse novo prédio.

Com essa estrutura, o número de prédios visíveis à esquerda de um prédio i é simplesmente o tamanho atual da pilha no momento da inserção.

Dessa forma, combinando o uso de uma variável acumuladora para tratar atualizações globais com a manutenção de uma pilha monótona, conseguimos resolver o problema em tempo linear.

Tutorial: Biscoitos Mineiros

Edson Alves

Uma das formas de se resolver esse problema é relembrar do funcionamento das funcões $sen(x)$ e $cos(x)$.

A função $sen(x)$ começa em 0 e cresce até 1 quando $x = 90$ e a função $cos(x)$ é exatamente o contrário, ela começa em 1 e decresce até 0 quando $x = 90$. Assim, o único ponto de igualdade das duas é quando $x = 45$.

Portanto, podemos exibir 'Ambos' se a entrada for 45 e, caso contrário, exibir 'Costa' ou 'Saad' a depender se a entrada é menor ou maior que 45.

Outra forma de resolver o problema é realmente calcular o valor das funções no ponto dado e compará-las. Note, porém, que essa solução pode trazer problemas de precisão ao comparar dois números de ponto flutuante.

Tutorial: Consulta ao Oráculo

Daniel Saad Nogueira Nunes

Uma forma de resolver essa questão é construir um grafo direcionado em que cada vértice representa uma posição $i \mod n$, em que $n = |S|$. Se existir um i tal que T ocorra em S^∞ a partir de uma posição $i \mod n$, criamos uma aresta entre o nó $i \mod n$ e o nó $(i + m) \mod n$, com $m = |T|$, com peso w_i. Acrescentamos nesta construção dois nós especiais, s e t, e criamos uma aresta de s para todos os outros vértices do grafo, com peso 0, e uma aresta de todos os outros vértices para t, também com peso 0.

Se existe um caminho com k arestas no grafo construído, isso implica em uma ocorrência de T^k em S^∞. Assim, se existe um ciclo no grafo, isso implica que T^∞ ocorre em $S\infty$. Então, o grafo construído:

1. Possui ciclo, caso T^∞ ocorra em S^∞ a partir de alguma posição $i \mod n$.
2. Não possui ciclos.

No primeiro caso, a resposta é -1, pois T^∞ ocorre em S^∞. No segundo caso a resposta é o comprimento do caminho de maior custo entre s e t. Este problema pode ser resolvido em tempo linear em DAGs, utilizando uma ordenação topológica ou uma busca em profundidade, a partir de s, com memória para armazenar os resultados parciais.

Construção do grafo: Para construir o grafo, precisamos determinar, para cada posição $0 \leq i < n$, se T ocorre em S^∞ a partir das posições $i \mod n$. Isso pode ser feito utilizando o algoritmo Z, sobre a string $T\$X$ em que X é a concatenação de S consigo mesma até que $|X| \geq 2m$, dessa forma garantimos que, se existe uma ocorrência de T em S^∞, a partir de alguma posição $i \mod n$, ela será reportada.

Análise de complexidade: Construir o grafo leva tempo $\Theta(n + m)$ e fazer a busca para detectar o ciclo ou calcular o caminho de maior custo leva tempo $\Theta(n)$, já que o grau de saída de cada vértice é no máximo 1.

Observações: O grafo não necessariamente precisa ser construído, mas isso não muda a complexidade do algoritmo.

Tutorial: Dossiês de Brasília

Ruan Petrus

A operação definida implica que a distância entre duas strings depende apenas da parte após remover o maior prefixo comum. Ou seja, se $A = PA'$ e $B = PB'$, então:

$$\text{dist}(A, B) = \text{dist}(A', B').$$

Assim, sempre podemos remover o maior prefixo comum e assumir que:

- ou uma das strings é vazia;
- ou os primeiros caracteres são diferentes.

Em ambos os casos, vale:

$$\text{dist}(A, B) = |A| + |B|.$$

Logo, o problema se reduz a maximizar a soma dos tamanhos de duas substrings, após remover o maior prefixo comum.

Ideia

Para a string S, basta considerar os dois melhores caracteres distintos que geram as maiores substrings (isto é, os dois melhores pontos de início).

Para cada string T, para cada caractere c presente nela, queremos combinar com o melhor caractere de S que seja diferente de c:

- se o melhor de S for diferente de c, usamos ele;
- caso contrário, usamos o segundo melhor.

Também consideramos escolher uma substring vazia:

$$\max(|S|, |T|).$$

Complexidade

Pré-processamos S em $O(|S|)$ e cada query em $O(|T|)$.

A complexidade total é:

$$O(|S| + \sum |T_i|).$$

Tutorial: Engarrafamento na Estrutural

Edson Alves

O primeiro passo para a solução do problema consiste em obter a fatoração dos N inteiros a_i. Como $|a_i| \leq 10^7$, utilizar um algoritmo $O(\sqrt{|a_i|})$ leva a um veredito TLE.

O crivo linear permite determinar todos os primos menores ou iguais a 10^7 e também, para cada inteiro $m \leq 10^7$, o menor primo $\pi(m)$ que divide m. Esta informação permite obter a fatoração prima de m em $O(\log m)$.

Uma vez fatorados os inteiros, basta montar um histograma dos números primos que aparecem em qualquer uma das fatorações e montar um vetor v de pares (h_j, p_j), onde h_j é o número de ocorrências do primo p_j nesse histograma.

Por fim, basta ordenar v, em ordem decrescente, e aplicar uma abordagem gulosa para obter a pontuação máxima de Alberto. A complexidade desta solução é $O(A + N \log AN)$, onde

$$A = \max\{|a_1|, |a_2|, \ldots, |a_N|\}$$

Tutorial: Fifibonacci

Alberto Neto

Queremos escrever $f(n)$ e $f(m)$ como combinação linear de $f(0)$ e $f(1)$. Considere a matriz A dada por

$$\begin{vmatrix} \text{a} & 1 \\ \text{b} & 0 \end{vmatrix}$$

Seja $f(n) = c_k f(k) + c_{k-1} f(k-1)$. Substituindo $f(k)$ por $af(k-1) + bf(k-2)$, temos

$$f(n) = (a * c_k + c_{k-1}) f(k-1) + (b * c_k) f(k-2) \ .$$

Considerando apenas os coeficientes, ao multiplicar a matriz A pelo vetor $c = (c_k, c_{k-1})$ temos

$$A(c_k \quad c_{k-1}) = (a * c_k + c_{k-1} \quad b * c_k) \ .$$

Ou seja, A*c é o vetor cujos valores são os coeficientes de $f(n)$ como combinação linear de $f(k-1)$ e $f(k-2)$. Inicialmente temos $f(n) = 1 * f(n) + 0 * f(n-1)$, então

$$A^{n-1} * (1 \quad 0) = (c_1 \quad c_0) \ ,$$

onde $f(n) = c_1 * f(1) + c_0 * f(0)$. Fazendo o mesmo para $f(m)$, obtemos o sistema

$$f(n) = c_1 * f(1) + c_0 * f(0)$$

$$f(m) = d_1 * f(1) + d_0 * f(0)$$

onde $f(1)$ e $f(0)$ são incógnitas. Por fim basta resolver o sistema, que pode ser resolvido facilmente já que o anel dos inteiros módulo 998244353 é um corpo.

Tutorial: Gama F. C.

Guilherme Ramos

A solução pode ser implementada de forma simplificada acompanhando o enunciado. Primeiramente, leia a instrução e avalie se começa com `if` ou `while`. Para qualquer destes casos, basta "recortar" o trecho seguinte até os dois pontos terminais (exclusive), obtendo assim a expressão lógica que poderá ser apresentada conforme a instrução.

A instrução seguinte é, necessariamente, entrada ou saída de dados. No caso de entrada, basta separar o nome da variável do resto para compor a instrução. Caso contrário, basta "recortar" o trecho entre o primeiro parênteses aberto e o último fechado para compor a instrução.

Tutorial: Hanoi

José Leite

Podemos nos apoiar na solução do problema original. Para mover N discos, movemos $N-1$ para a torre auxiliar, depois movemos o maior disco para o destino e por fim movemos os $N-1$ para o destino.

```
hanoi(disco, origem, destino, auxiliar)
  se(disco > 0)
    hanoi(disco - 1, origem, auxiliar, destino)
    mover(disco, origem, destino)
    hanoi(disco - 1, auxiliar, destino, origem)
hanoi(n, A, B, C)
```

A primeira mudança é que não podemos sempre computar qual é a origem. Precisamos ler da entrada e sempre guardar qual é a torre atual de cada disco.

Depois disto, há uma chance de o disco já estar no seu destino. Caso isso aconteça, podemos ir direto para o passo de mover os outros $N-1$ para o destino, também.

```
hanoi(disco, destino)
  se(disco > 0)
    origem = torre_do_disco[disco]

    se(origem == destino)
      hanoi(disco - 1, destino)
    senao
      auxiliar = terceira torre que não é ‘origem‘ e nem ‘destino‘
      hanoi(disco - 1, auxiliar)
      mover(disco, origem, destino)
      hanoi(disco - 1, destino)
hanoi(n, B)
```

Tutorial: Índice Ecológico

Gustavo Leal

A ideia central do problema é transformar a condição de gcd exatamente igual a X em uma condição mais simples.

Se $\gcd(A_i, \ldots, A_j) = X$ então todos os elementos são divisíveis por X e $\gcd(A_i/X, \ldots, A_j/X) = 1$.

Portanto, transformamos o vetor da seguinte forma:

- se A_i não é divisível por X, marcamos a posição como inválida;
- caso contrário, substituímos A_i por A_i/X.

Agora, o problema passa a ser: contar subsegmentos com gcd igual a 1, sem atravessar posições inválidas.

Para decidir se o gcd de um conjunto de números é 1, não importa quantas vezes um primo aparece, apenas se ele aparece.

Assim, podemos substituir cada valor pelo produto de seus fatores primos distintos. Agora a ideia é dividir o vetor em segmentos (usando uma Segment Tree por exemplo) e manter, para cada segmento, informações suficientes para contar subsegmentos dentro dele e combinar com outros segmentos.

Para cada intervalo, precisamos manter:

- quantos subsegmentos internos têm gcd = 1;
- os gcds possíveis dos prefixos válidos;
- os gcds possíveis dos sufixos válidos.

Considere dois segmentos consecutivos L e R.

Todo subsegmento do intervalo unido está em L, em R ou começa em L e termina em R. Os dois primeiros já são conhecidos. Para os que cruzam o meio, qualquer subsegmento pode ser descrito como:

$$\text{sufixo de } L \ + \ \text{prefixo de } R.$$

Se o gcd do sufixo é g_1 e o do prefixo é g_2, então o gcd do subsegmento completo é $\gcd(g_1, g_2)$. Logo, basta considerar todos os pares possíveis de gcds de sufixos e prefixos e contar aqueles cujo gcd é 1.

Seja K o número máximo de gcds distintos armazenados em prefixos ou sufixos de um segmento. Cada combinação de dois segmentos custa $O(K^2)$ e cada operação envolve $O(\log N)$ combinações. Portanto, a complexidade por operação é $O(K^2 \log N)$.

Para 10^5 K é 7, pois $2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 = 510510 > 10^5$.

Tutorial: Jacaré F. C.

Edson Alves

Vamos aplicar um algoritmo guloso baseado nas seguintes afirmações:

1) 'Se é possível defender o chute atual, defenda': O objetivo é sempre aumentar o total de defesas, então sempre iremos tentar defender.

2) 'Se não é possível defender o chute atual, vá para o meio': A posição central é a melhor posição para se estar, visto que dela o goleiro consegue defender qualquer chute (possivelmente tendo que se movimentar para algum lado). Então, mesmo que o goleiro não consiga defender o chute atual, ir para o meio garante que ele defenderá o próximo.

Assim, basta utilizar os argumentos acima para obter uma solução construtiva em $O(N)$.

Além do guloso citado, também é possível resolver usando programação dinâmica. Nessa técnica podemos modelar o problema como $dp(i, j)$, onde i seria a posição do chute atual e j a posição onde o goleiro se encontra. Como há poucas posições distintas para o goleiro, essa solução também pode ser implementada em $O(N)$ em tempo e $O(N)$ ou $O(1)$ de mémoria.

Obs.: Os argumentos do algoritmo guloso só são válidos pois só há três posições para o goleiro. Se houvesse 4 ou mais, apenas a solução com programação dinâmica estaria correta.

Tutorial: Kubitschek Monumentos

Ruan Petrus

Visao geral

A floresta muda ao longo do tempo e, apos cada operacao, precisamos da soma dos diametros de todas as componentes.

Apesar de existir uma solucao online com Link-Cut Tree, resolveremos o problema offline, usando conectividade dinamica com *divide and conquer* no tempo:

`https://codeforces.com/blog/entry/15296`

Intervalos de atividade

Cada aresta fica ativa em um intervalo $[l, r)$.

Assim, processamos recursivamente intervalos de tempo. Em cada subproblema, as arestas ativas durante todo o intervalo sao tratadas como fixas, e apenas as demais continuam para os filhos da recursao.

Componentes descartaveis

Se uma componente nao sera mais tocada por nenhuma query futura naquele subproblema, entao ela nao mudara mais.

Nesse caso, calculamos seu diametro, somamos sua contribuicao a resposta e removemos essa componente da instancia.

Compressao da arvore

Para manter a complexidade sob controle, comprimimos cada componente relevante.

Nao basta preservar apenas os vertices que ainda aparecem nas queries futuras: tambem precisamos manter os endpoints do diametro, pois eles bastam para atualizar corretamente o diametro quando a componente for ligada a outra no futuro.

Alem deles, mantemos apenas os vertices de ramificacao necessarios para conectar tudo isso. Essa compressao e feita com a ideia de *virtual tree* ou *compressed tree*: mantemos apenas os vertices importantes e substituimos cada caminho entre dois vertices mantidos por uma unica aresta com peso igual a distancia original.

Assim, preservamos todas as distancias que ainda serao necessarias.

Corretude

A corretude segue de tres fatos:

- arestas ativas durante todo um intervalo podem ser tratadas como fixas nesse subproblema;
- componentes sem participacao em queries futuras podem ser finalizadas imediatamente;

- a compressao preserva as distancias entre todos os vertices importantes, entao os calculos futuros de diametro continuam corretos.

Complexidade

A profundidade da recursao e $O(\log Q)$.

A soma do tamanho das instancias em cada nivel e linear no numero de vertices e queries ainda relevantes. Logo, a complexidade total e

$$O((N + Q) \log Q).$$

Tutorial: Leis Contraditórias

Gustavo Leal

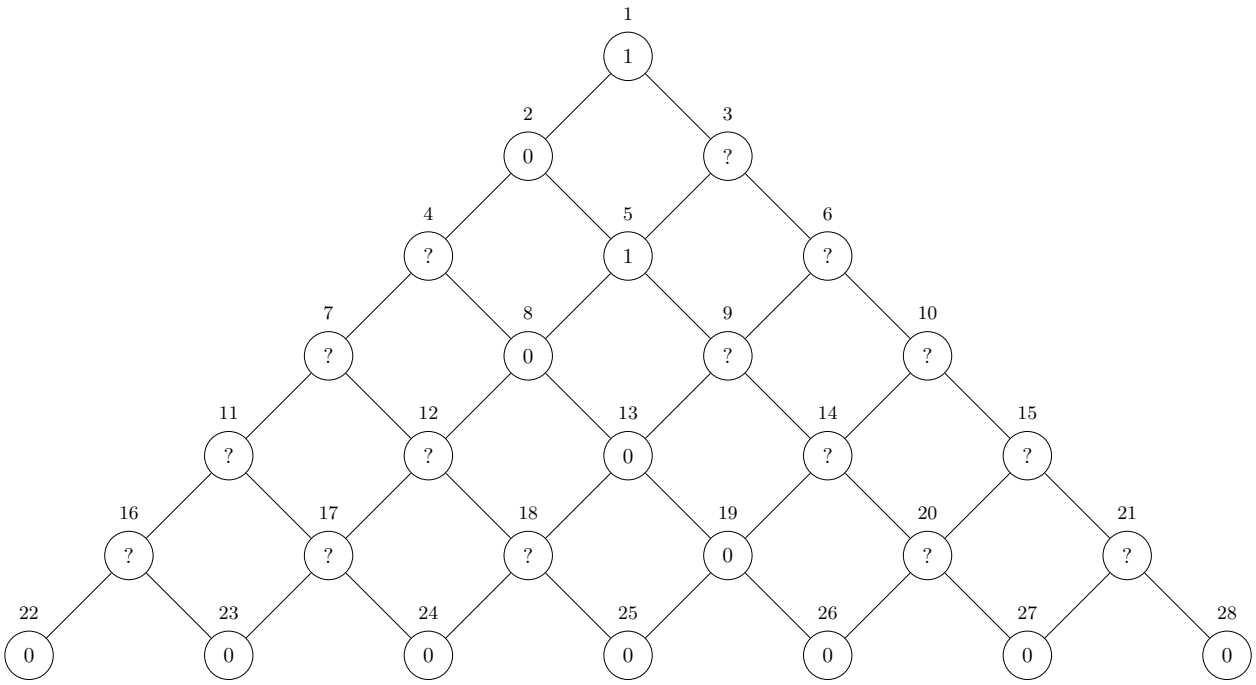
A primeira etapa para resolver o problema, é notar que em um caminho entre o topo e um vértice da base, com certeza existe uma contradição. Pois todos os vertices da base são 0 então é impossivel um 1 na camada logo acima possuir um 1 como um filho.

Uma possivel solução é começar as consultas na camada logo abaixo do topo, consulta-se os dois filhos; caso os dois sejam 0, o topo é uma contradição. Caso contrário, movemos a consulta para os filhos do filho do topo que é 1. Esta solução precisa de $2N$ consultas para encontrar a contradição, oque não é suficiente.

Para otimiza-la, consultamos apenas o filho mais a esquerda do topo. Temos duas possibilidades:

- Ele é um 1; nesse caso, movemos a consulta para o filho deste vértice.
- Ele é um 0; nesse caso, movemos a consulta para o filho do filho a direita do topo (o que não fizemos a consulta).

Após N consultas, teremos a arvore com uma estrutura parecida com essa:



Com certeza existe uma contradição no caminho entre o ultimo 1 que apareceu nas consultas, e a base. E existem no máximo N vértices nesse caminho. Podemos utilizar uma busca binaria para encontrar o 1 mais profundo nesse caminho, pois este será a contradição. Assim, a quantidade total de consultas será $N + log_2(N)$ oque é suficiente para a restrição.

Tutorial: Metrô Maluco

Alberto Neto

A solução modelo é **offline**. Existe uma solução online (pergunte ao Ruan Petrus).

Primeiro notemos que o caminho de a até b pode ser decomposto em um caminho para cima (a altura diminui) de a até $lca(a, b)$ e um caminho para baixo (a altura aumenta) de $\text{lca}(a, b)$ para b. No primeiro só há a possibilidade de subir pelas arestas da árvore, com custo $\text{h}(a) - \text{h}(\text{lca}(a, b))$. Pré-processamos esses valores e definimos $a = \text{lca}(a, b)$. para todas as consultas.

Seja u um vértice da árvore. Seja $\text{mx}(u)$ o maior índice na subárvore de u diferente de u ($\text{mx}(u) = -1$ se u for folha). Note que, utilizando o metrô, podemos ir de u até $\text{mx}(u)$ com custo 1 para qualquer vértice u, desde que $\text{mx}(u)$ exista. Esses valores podem ser calculados com uma simples dfs.

Faremos outra dfs a partir da raiz 1. Responderemos as consultas a b quando o vértice atual u na dfs for igual a b. Seja u o vértice atual da dfs e consideremos o caminho $c = (1, \text{mx}(1), \text{mx}(\text{mx}(1)), ..., \text{mx}^k(1) = v)$, onde $\text{mx}(v)$ está na subárvore de u e mx^k denota k vezes a composição de mx. Então o menor caminho de v até u é descer pelas arestas da árvore, ou ir para $\text{mx}(v)$, que está abaixo de u, e subir até u. Ou seja,

$$\text{custo}(v, u) = \min(\text{h}(u) - \text{h}(v), 1 + \text{h}(\text{mx}(v)) - \text{h}(u)) \ .$$

Considerando que o custo do caminho de 1 até v ja está calculado, temos

$$\text{custo}(1, u) = custo(1, v) + \text{custo}(v, u) \ .$$

Sejam $t_1, \ldots, t_k$ os filhos de u, e suponhamos (sem perda de generalidade) que $\text{mx}(u)$ está na subárvore de t_1. Ao chamar a dfs recursivamente para t_1, o custo de ir de v até $\text{mx}(u)$ é 2 se $\text{mx}(v) = u$ e 1 caso contrário (neste caso, $\text{mx}(v) = \text{mx}(u)$. Então $\text{custo}(v, t_1)$ e, posteriormente, $\text{custo}(v, \text{mx}(u))$ podem ser calculados facilmente. Ao chamar a dfs recursivamente para t_i com $i > 1$, porém, precisamos considerar o seguinte caminho:

$$v \to \text{mx}(u) \xrightarrow{\text{subindo}} u \to t_i \ ,$$

que pode ser menor do que descer pela árvore de v até t_i. Definiremos $\text{custo}(v, t_i)$ como o mínimo entre essas duas possibilidades.

Para considerar valores $\text{custo}(a, b)$ para $a \neq 1$, podemos manter um set de pares $\{\text{h}(x), x\}$ onde x é ancestral de u, e $x = \text{mx}^k(1)$ para algum k ou, ao longo da dfs, x não continha em sua subárvore o vértice $\text{mx}(p)$, como no caso de t_i no parágrafo acima. Os detalhes restantes são deixados como exercício.

Tutorial: Nibas e o contador

Adne Moretti Moreira

Neste problema, o objetivo é calcular o intervalo de tempo entre dois horários fornecidos. Dada a restrição de que ambos os horários estão entre 08:00 e 20:00, eliminamos a necessidade de tratar a virada do dia (meia-noite).

Uma estratégia para evitar problemas na implementação consiste em converter o tempo para uma unidade única: minutos totais desde o início do dia.

Seguindo essa abordagem, seja um horário F representado por $hh : mm$. O total de minutos M é calculado pela fórmula:

$M = (hh * 60) + mm$

Para dois horários, F e A, a diferença em minutos ΔM é obtida através da subtração dos valores em minutos:

$\Delta M = M_F - M_A$

Como os horários estão restritos ao intervalo entre 08:00 e 20:00, o valor de ΔM será sempre positivo, facilitando a conversão direta para o formato de saída.

$h_{res} = \Delta M/60$ (Divisão inteira)

$m_{res} = \Delta M \pmod{60}$ (Resto da divisão)

Formatação de Saída

A saída deve ser no formato $hh : mm$, o que exige preenchimento com zero à esquerda para valores menores que 10.