Editorial da Fase Zero da Maratona SBC de Programação 2025

A Ambíguo Gato de Schrödinger

Autores: João Victor Ayalla e Esrael Sousa

Se a caixa estiver fechada a resposta é "`vivo e morto`" independente do estado do gato, caso contrário, vai ser o estado do gato dado na entrada, podendo ser "`vivo`" ou "`morto`".

B Busca Periódica

Autor: Luiz H. B. Lago

Como uma adaptação de um problema clássico, que envolve achar o menor período de uma string, nesse caso temos que fazer isso considerando todas as strings que podem ser formadas sendo um caminho que sai da raiz de uma árvore até um determinado vértice.

Pensando no problema clássico sem ser em árvore (dado uma string de tamanho n, ache o seu menor período), podemos resolvê-lo utilizando o algoritmo KMP. Seja π_{n-1} o valor da função de prefixo considerando a string inteira, temos que:

Seja $k = n - \pi_{n-1}$. Se n for divisível por k, então, a resposta é igual a k, caso contrário, a resposta é igual a n - k. Essa solução é descrita muito bem em `https://cp-algorithms.com/string/prefix-function.html#compressing-a-string`.

Mas agora precisamos voltar ao problema original em árvores, e uma possível solução é rodar uma busca em profundidade a partir da raiz da árvore, e ir calculando a função de prefixo a medida em que vamos descendo na árvore, assim como podemos ir fazendo um rollback quando saímos no DFS como é mostrado no Algoritmo 1 e Algoritmo 2.

```
funcao dfs (int v) {
    for (cada filho u de v) {
        iteracao_kmp(caractere que tá na aresta de v para u);
        dfs(u);
        rollback();
    }
}
```

Algoritmo 1: Busca em profundidade fazendo iteração de KMP.

```
funcao iteracao_kmp (char ch) {
    // st é a string que representa o caminho feito na árvore
    s.append(ch);
    int i = st.size() - 1;
    int j = p[i - 1];
    while (j > 0 && st[i] != st[j])
      j = pi[j - 1];
    if (st[i] == st[j])
      j++;
    pi[i] = j;
}
```

Algoritmo 2: Implementação de uma iteração de KMP.

Mas em determinados casos, isso pode ser uma solução não tão eficiente para os limites da questão, já que em casos como o da Figura 1, é possível fazer com que esse while da função `iteracao_kmp` seja executado muitas vezes e a solução resulte em tempo limite excedido.

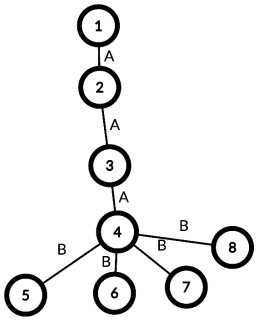

Figura 1: Caso ruim para iteração de KMP.

Assim, precisamos de um jeito diferente para calcular a função de prefixo que não seja o algoritmo KMP tradicional, e nisso, uma ideia clássica pode ser útil, a do autômato do KMP! Como no problema, o alfabeto tem tamanho igual a 26 (já que as arestas são compostas por letras minúsculas do alfabeto de 'a' até 'z'), podemos adaptar a função `iteracao_kmp` para computar um passo do autômato do KMP considerando o próximo caractere da string que representa o caminho feito na árvore como mostrado no Algoritmo 3.

```
funcao iteracao_kmp (char ch) {
    // st é a string que representa o caminho feito na árvore
    s.append(ch);
    int i = st.size() - 1;

    // aut -> matriz que vamos utilizar para representar o automato do kmp
    for (para cada caractere c de 'a' até 'z')
    {
      if (i > 0 && c != st[i])
        aut[i][c] = aut[p[i - 1]][c];
      else
        aut[i][c] = i + (c == st[i]);
    }
}
```

Algoritmo 3: Implementação de uma iteração de KMP usando autômato de KMP.

Um ótimo blog para entender sobre o autômato do KMP é o `https://cp-algorithms.com/string/prefix-function.html#building-an-automaton-according-to-the-prefix-function`.

Por fim, podemos calcular os valores de π_i como:

- $\pi_0 = 0$, pela definição da prefix function.
- $\pi_{i+1} = aut[\pi_i][c]$, sendo c o caractere que está sendo adicionado no fim da string que representa o caminho feito na árvore.

Complexidade final: $\mathcal{O}(26 \cdot N)$.

C Circuitos Lógicos Matriciais

Autor: Fernando Monteiro Kiotheka

Pelo enunciado, podemos resumir o problema em: Dadas várias matrizes CCNOT, devemos multiplicar todas elas na ordem dada na entrada e imprimir o resultado. Entretanto, se implementarmos essa ideia exatamente desse jeito, a solução teria complexidade $\mathcal{O}((2^n)^3 \cdot M)$ que resultaria em tempo limite excedido. Assim, vamos relembrar a definição de uma matriz CCNOT:

$$\text{CCNOT}(q_{c_1}, q_{c_2}, q_t)_{ij} = \begin{cases} 1 & \text{se } i \text{ tem os bits } c_1 \text{ e } c_2 \text{ ligados e } i \oplus 2^t = j \\ 0 & \text{senão se } i \text{ tem os bits } c_1 \text{ e } c_2 \text{ ligados e } i \oplus 2^t \neq j \\ 1 & \text{senão se } i = j \\ 0 & \text{caso contrário} \end{cases}$$

A grande sacada aqui é perceber que para todas as linhas i de uma matriz $CCNOT$, existe exatamente uma coluna j tal que o elemento $\text{CCNOT}(q_{c_1}, q_{c_2}, q_t)_{ij} = 1$. Para todas as demais colunas k, $\text{CCNOT}(q_{c_1}, q_{c_2}, q_t)_{ik} = 0$. A mesma coisa vale para as colunas, para todas as colunas da matriz, essa mesma afirmação vai ser verdadeira. Esta matriz é chamada de matriz de permutação.

Podemos então manter a permutação que a matriz representa ao invés da matriz completa. Um elemento P_i do vetor de permutação diz qual a coluna em que $\text{CCNOT}(q_{c_1}, q_{c_2}, q_t)_{ipos_i} = 1$ para linha i.

Assim, é possível simular a multiplicação de uma matriz CCNOT representada pelo vetor de permutação através da composição de permutação como mostra o Algoritmo 4.

```
for (int i = 0; i < (2^n); i++)
    ans[i] = pos[ans[i]];
```

Algoritmo 4: Combinação de permutação.

Complexidade final: $\mathcal{O}(2^N \cdot M)$.

D Decoerência Quântica

Autor: Esrael Sousa

Para resolver, contamos a diferença de "$*$" entre S e T e depois dividimos pelo número de "$*$" em S.

E Energização de Partículas

Autor: Vinícius Silva

Queremos encontrar onde a partícula de carga Y para após K procedimentos, começando de $X = 1$, e avançando $\gcd(X, Y)$ a cada passo. Caso o K fosse pequeno, poderíamos fazer força bruta simulando todos os procedimentos de energização da partícula. Porém, como $K \leq 10^9$, então é necessário buscar meios de otimizar isso. Podemos analisar alguns pontos a respeito do movimento da partícula:

1. Por definição, $\gcd(X, Y)$ sempre será um divisor de Y;
2. Cada passo dela é sempre múltiplo do passo anterior. Podemos deduzir assumindo que P é o passo atual da partícula saindo de X e que ela pode dar um número A qualquer de passos. Para tal, considere que queremos obter $\gcd(X + A \cdot P, Y)$. Como $P = \gcd(X, Y)$ temos que:

$$\begin{aligned}&\gcd(X + A \cdot P, Y)\\ &= \gcd(X + A \cdot \ gcd(X, Y), Y)\end{aligned}$$

   Usando $X = B \cdot \gcd(X, Y)$ e $Y = C \cdot \gcd(X, Y)$,

$$\begin{aligned}&\gcd(X + A \cdot \gcd(X, Y), Y)\\ &= \gcd(gcd(X, Y) \cdot (A + B), \gcd(X, Y) \cdot C)\\ &= \gcd(X, Y) \cdot \gcd(A + B, C)\\ &= P \cdot \gcd(A + B, C)\end{aligned}$$

   Como $\gcd(A + B, C) \geq 1$, então a condição acima está satisfeita, ou seja, o passo seguinte é sempre múltiplo do passo anterior. ■
3. Assumindo as duas observações acima, podemos então fatorar Y em $\mathcal{O}(\sqrt{N})$, obtendo os seus divisores em ordem crescente.

Com isso, se estamos no passo $P = \gcd(X, Y)$, precisamos saber se é possível obter algum múltiplo de P que também é múltiplo de algum divisor $D > P$ de Y para saber qual a próxima posição Z que a partícula irá mudar o seu passo. Caso isso não seja possível, então a partícula irá pular em passos de tamanho P até esgotar as K operações, portanto podemos apenas multiplicar P pela quantidade de procedimentos remanescentes.

Em resumo, precisamos encontrar a posição Z que:

- $Z > X$;
- Z é múltiplo de P;
- Z é múltiplo de um divisor D de Y, tal que $D > P$.

Para tal, podemos iterar por todos os divisores $D > P$, e pegar menor valor Z que satisfaça a relação acima, pois será a primeira posição que P mudará. Para um número ser múltiplo de D e P, então a condição mínima necessária é que o número também seja múltiplo de $lcm(P, D)$. Logo, com esse valor Z obtido, precisamos recalcular o número de procedimentos para chegar até esse número, logo esse será $T = \frac{Z-X}{P}$. Se $T \leq K$, então a resposta será $X + K \cdot P$. Caso contrário, subtraímos T de K e enviamos a partícula ao ponto $X = Z$, com novo passo $P = \gcd(Z, Y)$.

A complexidade final disso aparenta ser $O(\sqrt{Y} + \sigma(Y)^2)$, em que $\sigma(Y)$ equivale ao número de divisores de Y. No entanto, é $\mathcal{O}(\sqrt{Y} + \sigma(Y) \log Y)$.

O logaritmo surge da observação de que cada vez que o passo P muda, há a adição de pelo menos um primo ou o aumento do expoente de um dos primos que compõe a fatoração de P. Como o $\gcd(X, Y)$ para qualquer X é no máximo Y, então só temos no máximo $\mathcal{O}(\log Y)$ primos, então o número de mudanças fica limitado por essa complexidade.

Também é possível deduzir que todo valor de Z possível será um divisor de Y, tal que $Z = \gcd(X, Y) \cdot \rho_1(\frac{Y}{X})$, em que $\rho_k(N)$ é o k-ésimo menor primo da fatoração de N. Ou seja, o próximo passo será sempre multiplicado pelo menor primo que compõe a fatoração de $\frac{Y}{X}$, utilizando $\rho_1(\frac{Y}{X}) - 1$ procedimentos. É um bom exercício aos competidores chegarem a essa conclusão.

F Feynman Decorando Números

Autor: Esrael Sousa

Para resolver esse problema, podemos fazer programação dinâmica como mostra o Algoritmo 5. Como precisamos pegar 4 elementos que somem X, e temos várias consultas, podemos pré-calcular o número de combinações para todas as somas possíveis. Cada número da sequência pode ser o primeiro, segundo, terceiro ou quarto elemento da combinação, fazendo isso para cada soma possível.

```
std::vector<std::unordered_map<int, long long>> dp(5);
dp[0][0] = 1;
for (int x : array) {
    for (int k = 3; k >= 0; --k) {
        for (auto &[soma, qtd] : dp[k]) {
            dp[k + 1][soma + x] += qtd;
        }
    }
}
```

Algoritmo 5: Implementação da programação dinâmica do problema F.

Para cada consulta, mostre 0 se a chave q_i não está em $dp[4]$, caso contrário, $dp[4][q_i]$.

Também é possível fazer de forma recursiva e memorizando com uma matriz, tomando cuidado com números negativos.

G Grover e Seus Caminhos Especiais

Autor: João Victor Ayalla

Primeiro, podemos parar para pensar em uma solução para o problema quando $P = 0$. Nesse caso, o problema pode ser modelado como um problema de fluxo máximo como mostra a Figura 2.

Criamos um vértice source e um sink na rede de fluxo, e de um lado criar um vértice na rede para cada vértice da árvore. Além disso, devemos criar vértices para os valores possíveis de 1 até 5. Com isso, se o fluxo máximo for igual a N, achamos uma solução válida para o problema no qual $P = 0$.

Mas temos que lidar com as restrições dos caminhos, e para isso, algumas observações são importantes:

- Um caminho especial não pode ter tamanho maior do que 5, já que a sequência tem que ser crescente e só tem 5 valores possíveis para os vértices. Caso um dos caminhos especiais tenha tamanho maior do que 5, a resposta para esse caso de teste vai ser igual a -1.
- Para um caminho de tamanho X, ele pode ser preenchido de $\binom{5}{X}$ maneiras diferentes, já que quando fixamos uma certa combinação de números, só temos uma única maneira de arranjá-los de forma crescente. No pior caso, temos que $\binom{5}{X} = 10$ e com isso, como $1 \leq P \leq 5$, no pior dos casos temos 10^5 maneiras de preencher os vértices que aparecem em pelo menos um caminho; são poucas maneiras.

Sabendo disso, podemos pensar em percorrer todas essas possibilidades de preencher os vértices **que aparecem em pelo menos um caminho especial**, e para os demais vértices (que não aparecem em nenhum caminho especial), basta resolver o problema em que $P = 0$ que descrevemos anteriormente. Entretanto, rodar um algoritmo de fluxo 10^5 vezes (no pior caso) pode ser muito custoso e precisamos de algo melhor.

Para isso, podemos lembrar do Hall's Marriage Theorem: *Hall's theorem can be used to find out whether a bipartite graph has a matching that contains all left or right nodes. Assume that we want to find a matching that contains all left nodes. Let X be any set of left nodes and let $f(X)$ be the set of their neighbors. According to Hall's theorem, a matching that contains all left nodes exists exactly when for each X, the condition $|X| \leq |f(X)|$ holds.*

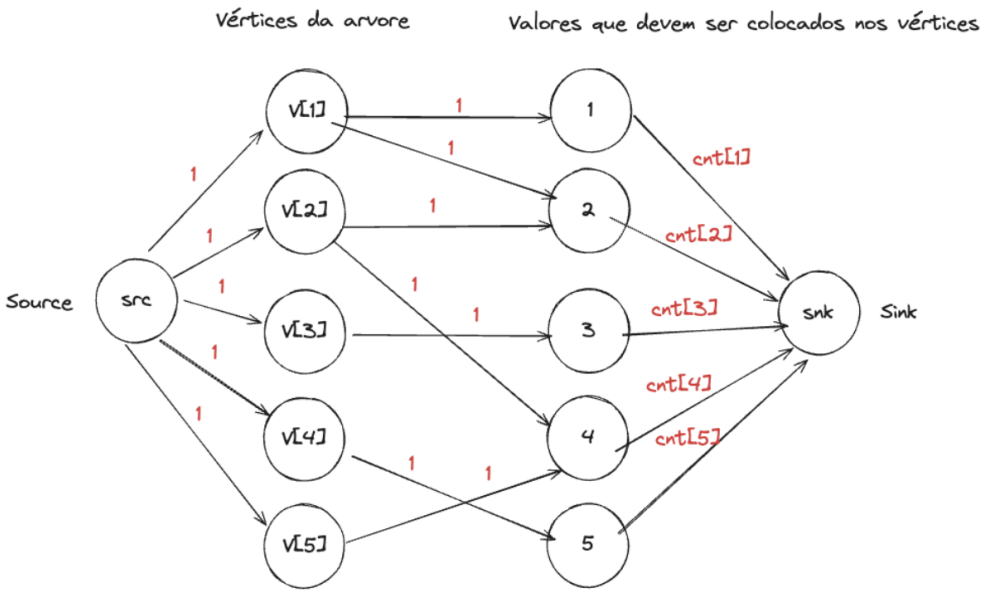

Figura 2: Modelagem de fluxo do problema G.

Assim, podemos verificar se o emparelhamento que queremos existe apenas fixando um subconjunto de valores. Como só existem valores de 1 até 5, existem no pior dos casos 31 subconjuntos não vazios de valores. E para cada um desses subconjuntos S devemos verificar se:

$$\sum_{x \in S} cnt_x \leq V$$

Sendo V a quantidade de vértices que pode assumir pelo menos um valor que está contido no conjunto S, ou em outras palavras, a quantidade de vértices i (que não aparecem em nenhum caminho especial) tal que a intersecção entre o conjunto S e o conjunto de valores que v_i pode assumir é não vazia.

Com isso, podemos fazer um backtracking para iterar por todas as possibilidades de preencher os valores dos vértices que aparecem em pelo menos um caminho especial e, para uma possibilidade de preencher esses valores, checamos se conseguimos preencher os valores dos vértices restantes usando o Hall's Marriage Theorem em $O(2^{\text{MAXVAL}} \cdot \text{MAXVAL})$, no qual $\text{MAXVAL} = 5$.

H Harmonia Palíndromica Binária

Autor: João Victor Ayalla

Vamos separar a solução em 3 casos:

- **Caso $X = 1$:**
  Se $X = 1$, logo a resposta é 1. Já que é $B(Y) = 1$ um palíndromo.
- **Caso $X \geq 1$, X é uma potência de 2:**
  Se X é uma potência de 2 e é maior do que 1, a resposta é igual a $X - 1$. Podemos ver que $B(X) = 1000000\ldots$ não é palíndromo, enquanto que $B(X - 1) = 1111111\ldots$ é um palíndromo por ser uma string composta somente por uns.
- **Caso X não é uma potência de 2:**
  Caso contrário, podemos concluir que $MSB(X) = MSB(Y)$ (no qual $MSB(X)$ representa o bit mais significativo de um número x), já que todo número K que tem a forma $K = 2^L + 1$ possui $B(K) = 1000\ldots0001$ que vai ser um palíndromo.

Nisso podemos concluir que se X não é uma potência de 2, então a resposta Y vai ser pelo menos igual a K ($Y \geq K$), sendo $K = 2^{MSB(X)} + 1$.

Logo, se sabemos a quantidade de bits de $B(Y)$, podemos ir de forma gulosa considerando do maior bit para o menor bit, e caso seja possível setar esse bit (não formar um número maior que X), nós setamos e seguimos para o próximo bit.

Note que, se setamos o K-ésimo bit em Y devemos também setar o bit na posição $MSB(X) - K$.

Complexidade final: $\mathcal{O}(\log X)$.

I Inspecionando o Emaranhamento

Autores: Thailsson Clementino e Rosiane de Freitas

Seja $f(i, j)$ a confiabilidade máxima utilizando o sensor i no tempo j. A resposta para o problema é $max_{1 \leq i \leq N}\{f(i, 1)\}$. Para calcular $f(i, j)$ podemos utilizar a relação de recorrência a seguir:

$$f(i,j) = \begin{cases} 0, & \text{se } j = T + 1, \\ \max\limits_{j+L \leq j' \leq j+U} \left\{ \sum\limits_{k=j}^{j'-1} c(i,k) + \max\limits_{ni \neq i}\{f(ni, j')\} \right\}, & \text{caso contrário.} \end{cases}$$

Para resolver, basta usar programação dinâmica com a complexidade $\mathcal{O}(N^2 \cdot T^2)$. Podemos otimizar um pouco, calculando $\sum\limits_{k=j}^{j'-1} c(i,k)$ em $O(1)$ utilizando soma de prefixos e $\max\limits_{ni \neq i}\{f(ni, j')\}$ em $\mathcal{O}(1)$ guardando as duas melhores respostas de $f(i, j)$ para cada j já computado. Com isso a complexidade te tempo cai para $\mathcal{O}(N \cdot T^2)$ que é o esperado como solução.

J Jornada das Partículas

Autor: Vinícius Silva

Primeiramente, busquemos solucionar o problema em uma linha, desconsiderando o fato que ele é circular. Dessa forma, ao fixar um filtro i, nós temos que X será A_i e a cada passo X aumentará em K. Logo, X é uma função linear em relação ao número de operações, sendo descrita por:

$$X = A_i + K \cdot T,$$

em que T é o número de operações.

Com isso, podemos simplificar a inequação para cada par de filtros (i, j), com $i \leq j$ para saber qual a condição necessária para que j filtre caso i comece. Sabemos que por ser em uma linha o número de operações será $j - i$. Nesse caso, a inequação é descrita por

$$A_i + K \cdot (j - i) > A_j.$$

Expandindo os termos em função de K e separando os termos que dependem de i em um lado e os de j em outro, a inequação resultante é

$$A_i - K \cdot i > A_j - K \cdot j.$$

Como os termos estão separados por índice, podemos considerar $Y_p = A_p - K \cdot p$, tornando a inequação final

$$Y_i > Y_j.$$

Então, nosso problema resume em encontrar o primeiro índice j no sufixo de cada i que satisfaz $Y_i > Y_j$.

Há várias maneiras de solucionar esse problema, porém uma maneira simples de resolver é utilizando uma estrutura de pilha. Podemos prosseguir da seguinte maneira a cada índice:

1. Verifique se a pilha está vazia. Se ela não estiver, siga para o passo 2, caso contrário vá ao passo 4.
2. Verifique se o Y do índice do topo da pilha é maior que o Y do índice atual. Caso seja, prossiga ao passo 3, caso contrário vá ao passo 4.
3. O índice do topo da fila terá o índice atual como posição na qual a partícula será filtrada. Logo, podemos fazer $B[indice\_topo] := indice\_atual$ e desempilhá-lo em seguida. Retorne ao passo 1.
4. Adicione $indice\_atual$ ao topo da pilha e termine.

Agora, considerando o problema como circular, temos um fato muito importante: para cada índice nunca haverá mais do que $N + 1$ operações. Isso ocorre devido ao fato que após N procedimentos, o filtro atual será o filtro que começou o processo. E como K é positivo, o X sempre será maior que o valor daquele filtro, logo a partícula sempre será filtrada. Com isso, nós só precisamos iterar no máximo $2 \cdot N$ vezes em "linha", usando o fato que ao chegar ao turno do filtro N, em vez de "voltar" ao filtro 1, podemos considerar como se ele fosse $N + 1$, do mesmo jeito do filtro 2 como $N + 2$ e assim por diante. Dessa forma, o cálculo da inequação não será alterado, mas só precisaria ter cuidado na hora de atualizar as respostas.

Complexidade final: $\mathcal{O}(N)$.

K K Elementos Perdidos

Autor: Vinícius Silva

Esse problema envolve encontrar as primeiras K subsequências crescentes de maior peso. Logo, de certa forma, é necessário saber como obter a maior subsequência de maior peso, e como generalizar isso para K elementos.

Para resolver esse problema, é possível construir o DAG (Directed Acyclic Graph) de pesos em relação às subsequências crescentes adicionando uma aresta para todo (u, v) tal que $A_u < A_v$ e $u < v$, com peso B_v. Com isso, podemos encontrar os caminhos mais pesados saindo de todo vértice u inicial e obter os K maiores. Simplificando isso, podemos criar um nó S e ligar a todo $1 \leq u \leq N$ com peso B_u, e um nó T que é ligado por $1 \leq u \leq N$ com peso 0.

Para simplificar a didática, considere a seguinte entrada:

- $N = 5$, K qualquer
- $A = \{5, 1, 2, 3, 4\}$
- $B = \{5, 1, 2, 3, 4\}$

O DAG inicial está representado na Figura 3. Já o DAG final é a união dos DAGs da Figura 3 e Figura 4. Foi feito a separação para reduzir a poluição visual.

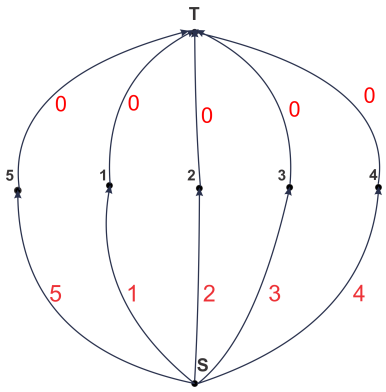

Figura 3: DAG do trecho $S \to T$. Em preto, estão os A_u, e em vermelho, os pesos de cada aresta.

Dessa forma, só precisamos encontrar os caminhos mais pesados de S à T, como o encontrado na Figura 5. No entanto, temos $\mathcal{O}(2^N)$ subsequências crescentes, e iterar por todas elas possui um custo alto. Porém,

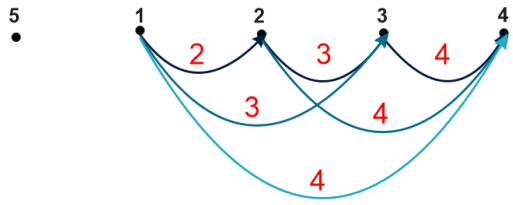

Figura 4: DAG do trecho $1 \leq u \leq N$.

é possível chegar a uma complexidade de $\mathcal{O}(N^2 K log(K))$ simplesmente podando subsequências subótimas, pois podemos manter até K caminhos ótimos por vértice u do grafo acima. Basicamente, podemos processar o grafo acima em ordem topológica, inicializando S com um caminho de tamanho 0, e para cada aresta $u \to v$ processada, pegamos os caminhos ótimos de u, adicionamos o peso dessa aresta e incorporamos aos caminhos ótimos que chegam em v. Quando a quantidade de caminhos ótimos em v for $> K$, podemos simplesmente remover o caminho de menor peso, pois este nunca agregará na resposta final. Com isso, a resposta final seria os caminhos que chegam em T em ordem não-crescente. Como há $\mathcal{O}(N^2)$ transições (arestas) e um custo de $\mathcal{O}(Klog(K))$ por transição, então chegamos à complexidade acima. No entanto, ainda é muito lento.

Para acelerar isso, poderíamos construir a solução calculando o caminho mais pesado e guardar potenciais candidatos através de estruturas persistentes, mantendo informações importantes do caminho sem utilizar muita memória. Juntando essas peças, é possível utilizar um algoritmo especializado para resolver esse tipo de problema: o algoritmo de Eppstein. Implementações padrões desse algoritmo resolve esse tipo de problema em $\mathcal{O}((V+E)logV+KlogK)$ usando o algoritmo de Dijkstra para calcular o caminho e os potenciais do grafo, além de árvores de esquerda e uma fila de prioridade para gerenciar os melhores caminhos. Vale lembrar que esse algoritmo encontra os K caminhos mais curtos, logo temos que multiplicar os pesos do grafo por -1 para encontrar os caminhos mais longos. Como o algoritmo de Dijkstra degenera para $\mathcal{O}(VElogV)$ para DAGs com pesos negativos, devemos trocar essa etapa por uma varredura em ordem topológica do grafo para deixar essa etapa em $\mathcal{O}(V+E)$. Além disso, como $V = \mathcal{O}(N)$ e $E = \mathcal{O}(N^2)$, então a solução inicial ainda é $\mathcal{O}(N^2 logN + KlogK)$, o que é muito lento.

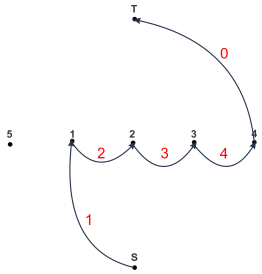

Figura 5: Caminho mais longo de $S \to T$.

Competidores que conhecem a solução do problema da LIS (Longest Increasing Subsequence) em complexidade subquadrática usando Dijkstra, BFS 0-1 ou até mesmo BFS com ordenação topológica já conseguem resolver esse problema aplicando o algoritmo diretamente. Caso contrário, podemos usar algumas técnicas para reduzir a complexidade.

Podemos decompor o vetor A em blocos de tamanho L, respeitando algumas características:

- Denote X como o bloco que o vértice u pertence. Então $X = \lceil \frac{u}{L} \rceil$.
- Só é possível ligar um vértice u aos vértices v com peso B_v tal que a condição de sequência é satisfeita e que estão no mesmo bloco X.
- É criado $\frac{N}{L}$ vértices auxiliares, ou checkpoints, de modo que cada checkpoint de um bloco X está ligado ao próximo ao próximo $(X \to X+1), 1 \leq X < \frac{N}{L}$, com peso 0.
- Os vértices do bloco X são ligados ao checkpoint do bloco X com peso 0.
- Os vértices do checkpoint X são ligados aos vértices v do bloco $X+1$ com peso B_v.

Adotando $L = 2$, a decomposição ficaria como na Figura 6.

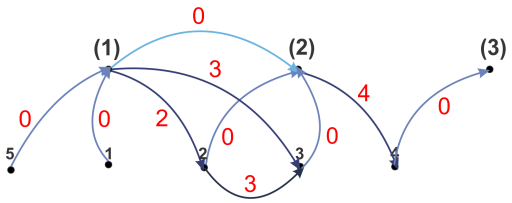

Figura 6: DAG do trecho $1 \leq u \leq N$ após a decomposição.

Com isso, tem-se $\mathcal{O}(L^2)$ arestas por bloco, cada vértice de "checkpoint" possui $\mathcal{O}(L)$ arestas, e o total de blocos é $\frac{N}{L}$, o que ficaria $\mathcal{O}(NL)$ no final. Obviamente, escolhendo $L = 1$, chegaríamos a $\mathcal{O}(N)$. No entanto, temos um grande problema dessa "decomposição":

- Como um checkpoint garante que toda transição que ele está fazendo a partir de um nó u de blocos anteriores para um nó v do bloco atual ou de blocos futuros está satisfazendo $A_u < A_v$?

Um exemplo de transição inválida é encontrada na Figura 7.

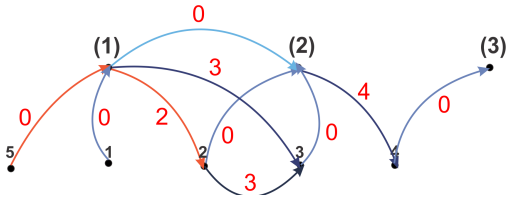

Figura 7: No caminho em laranja está representado uma transição inválida.

Pensando no problema de subsequências crescentes, há uma restrição em duas dimensões entre dois elementos: índice e valor. A divisão acima resolve o problema dos índices, portanto, é necessário também aplicar uma decomposição de blocos em termos de valor. Ou seja, teremos uma "decomposição 2D", onde cada bloco que restringe por índice, será subdividido em blocos de restrição por valor como mostra a Figura 8.

Iremos adotar como M o tamanho dos blocos com restrição de valor. Temos as seguintes características:

- Denote Y como o bloco de valor que o vértice u pertence. Então, $Y = \lceil \frac{A_u}{M} \rceil$.
- Cada par (u, A_u) está definido por um bloco (X, Y).
- Só é possível ligar um vértice u aos vértices v com peso B_v tal que a condição de sequência é satisfeita e que estão no mesmo bloco (X, Y).
- É criado $\frac{N^2}{LM}$ vértices auxiliares, ou checkpoints, de modo que cada checkpoint de um bloco (X, Y) está ligado tanto a $(X, Y + 1)$ quanto $(X + 1, Y)$, com peso 0, ou seja, uma matriz direcionada.
- Os vértices do bloco (X, Y) são ligados ao checkpoint do bloco (X, Y) com peso 0.
- Os vértices de cada checkpoint (X, Y) são ligados aos vértices v do bloco $(X + 1, Y + 1)$ com peso B_v.

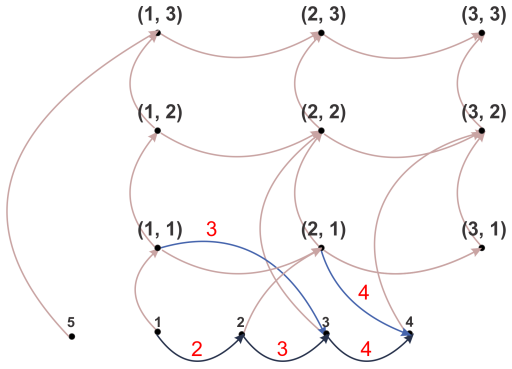

Figura 8: Decomposição 2D do exemplo. Arestas em bege possuem peso 0.

No entanto, temos outro problema:

- Para algum caminho de um checkpoint $(X, Y) \to (X', Y')$ com $X' > X, Y' > Y$, tem-se mais de um caminho de peso 0 que ligue esses blocos.

O problema é exemplificado na Figura 9.

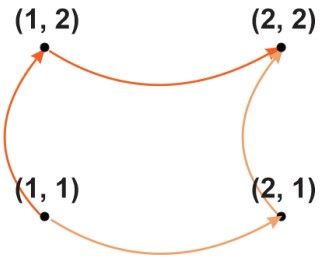

Figura 9: Exemplo de transição múltipla. $(1, 1) \to (2, 2)$ pode-se ir tanto por $(1, 2)$ quanto $(2, 1)$.

Para solucionar isso, podemos transformar a matriz em um tensor de ordem 3, criando uma terceira dimensão de tamanho 2 que restringe a matriz da seguinte maneira:

- O checkpoint $(0, X, Y)$ recebe todas as transições de vértices e/ou de $(0, X-1, Y)$ e só pode encaminhar para $(0, X+1, Y)$ ou $(1, X, Y)$.
- O checkpoint $(1, X, Y)$ recebe apenas as transições de $(0, X, Y)$ e/ou de $(1, X, Y-1)$ e só pode encaminhar para $(1, X, Y+1)$ ou aos vértices.

Essa ideia é semelhante aos conceitos empregados em fluxo para adotar demanda em vértice.

Com isso, cada entrada de checkpoint fará transições prosseguindo uma dimensão de cada vez. Analisando em duas dimensões, uma transição $(X, Y) \to (X', Y')$ fará $(X, Y) \to (X', Y)$ para depois fazer $(X', Y) \to (X', Y')$. Como cada transição só possui um caminho disponível, logo há exatamente um caminho de qualquer nó u a algum nó v tal que a condição de sequência crescente é satisfeita utilizando os checkpoints. Uma representação gráfica desse esquema pode ser encontrada na figura Figura 10.

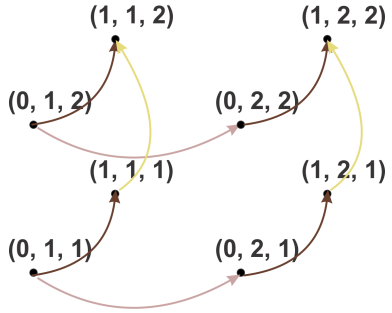

Figura 10: Transição com o tensor de ordem 3. Agora, $(1, 1) \to (2, 2)$ faz o percurso $(0, 1, 1) \to (0, 2, 1) \to (1, 2, 1) \to (1, 2, 2)$.

Dessa forma, temos $\mathcal{O}(\frac{N^2}{LM})$ checkpoints, e cada checkpoint (X, Y) recebe $\mathcal{O}(min(L, M))$ transições de vértices do array, porém a soma de transições no final será $\mathcal{O}(N)$, já que cada elemento só precisa ligar uma aresta para o checkpoint e vice-versa. Com essas informações, então temos $\mathcal{O}(\frac{N^2}{LM})$ vértices no grafo e $\frac{N^2}{LM} + \frac{N}{L} \cdot \mathcal{O}(L^2) + \frac{N}{M} \cdot \mathcal{O}(M^2) + \mathcal{O}(N)$ arestas $= \mathcal{O}(\frac{N^2}{LM} + NL + NM)$.

Amantes de decomposição SQRT poderiam aplicar $L = M = N^{\frac{1}{2}}$, deixando o grafo com $\mathcal{O}(N^{\frac{3}{2}})$ arestas e $\mathcal{O}(N)$ vértices. Olhando para o algoritmo, a complexidade fica $\mathcal{O}(N^{\frac{3}{2}} log N + K log K)$, o que é suficiente para resolver o problema.

No entanto, assintoticamente falando, a melhor decomposição vem com $L = M = N^{\frac{1}{3}}$, o que seria uma decomposição CBRT, que chega a uma complexidade de $\mathcal{O}(N^{\frac{4}{3}})$ arestas e vértices, o que resulta em $\mathcal{O}(N^{\frac{4}{3}}log(N) + KlogK)$.

Com o grafo montado e todos os problemas solucionados, podemos finalmente utilizar o algoritmo de Eppstein para obter os K caminhos mais pesados e chegar à solução final do problema, lembrando de multiplicar de volta por -1 o peso do caminho na resposta e imprimindo -1 caso o array final tenha tamanho $< K$.

Complexidade: $\mathcal{O}(N^{\frac{4}{3}}logN + KlogK)$.

L Linha de Produção de qPhones

Autor: Thailsson Clementino e Rosiane de Freitas

Para simular completamente a memória de um celular clássico com M megabytes (MB), é preciso garantir que os qubits do novo aparelho sejam capazes de representar ao menos M, ou seja, $M \cdot 8\,000\,000$ bits. Assim, é necessário encontrar o menor valor de k tal que $M \cdot 8000000 \leq 2^k$

Uma solução é iterar o k a partir de 1 e encontrar o primeiro k em que a desigualdade é satisfeita. Outra solução é calcular $\lceil \lg M \cdot 8000000 \rceil$.

M Movimentação Assustadora à Distância

Autores: Vinícius Silva e João Ayalla

A inspiração para a solução descrita aqui vem do blog `https://codeforces.com/blog/entry/119082` que é sobre entender sobre *multiple Möbius transform*. Um bom exemplo para começar pensando é em convoluções de gcd (máximo divisor comum) em $O(n \log \log n)$, como citado no blog.

Pensando primeiro em resolver o problema sem atualizações, apenas com consultas do tipo 1 em um vetor estático, podemos aplicar a seguinte ideia do Algoritmo 6.

```
for (int x = MAXN - 1; x >= 1; x--) {
// cnt[x] -> quantos índices i do array existem tal que a[i] é um múltiplo de x
ans[x] = pow(2, cnt[x]) - 1;
for (int y = x + x; y < MAXN; y += x)
    ans[x] -= ans[y];
}
```

Algoritmo 6: Resolvendo consultas do tipo 1 do jeito trivial.

Assim, ans_x seria a resposta de uma consulta do tipo 1 para um dado x. Agora, voltando para o contexto do blog, podemos calcular ans_x seguindo a ideia de *multiple Möbius transform* por meio do Algoritmo 7.

```
for (int x = 1; x < MAXN; x++) {
    ans[x] = pow(2, cnt[x]) - 1;
}

for (int p : prime_nums) {
    for (int x = 1; x * p < MAXN; x++)
        ans[x] -= ans[x * p];
}
```

Algoritmo 7: Resolvendo consultas do tipo 1 usando *multiple Möbius transform*.

No qual *prime_nums* é uma lista com todos os números primos que existem no intervalo $[1, 10^5]$. Assim, conseguimos resolver o problema sem atualizações pré-calculando todas as respostas em $O(n \log \log n)$.

Mas agora, voltamos ao problema em que devemos lidar com as atualizações e, assim, se estamos atualizando a_i (digamos, de um valor antigo x_{antigo} para um novo valor x_{novo}), devemos remover toda a contribuição da ocorrência de x_{antigo} e adicionar a contribuição da ocorrência de x_{novo}. Para que, assim, os valores de ans_k (para os diversos k) continuem corretos.

Pensando em atualizar a contribuição de um número x, precisamos diminuir ou aumentar em uma unidade o valor de cnt_d, para todo d tal que $d \mid x$.

Além disso, para todos os d, vamos repetir a estratégia de calcular ans_d em uma abordagem parecida com a que usamos anteriormente. Mas agora, só precisamos iterar por números que sejam relevantes para calcular o valor de ans_d. O código fica como no Algoritmo 8.

Para $1 \leq a_i \leq 10^5$, temos que o produto (`divisores_primos[`a_i`].size()` $\cdot$ `divisores[`a_i`].size()`) ≤ 640. Assim conseguimos resolver o problema mesmo com os $2 \cdot 10^5$ atualizações no pior caso, já que em uma operação do tipo 2 precisamos chamar $upd(x_{antigo}, false)$ e $upd(x_{novo}, true)$. Além disso, precisamos pré-calcular todos os valores iniciais de ans_x antes de começar a processar as consultas, podemos fazer esse pre-cálculo em $O(n \log \log n)$ como descrito acima.

```
funcao upd (int x, bool add) {
    for (int d : divisores[x]) {
        // se add == true, estamos adicionando a contribuição de uma ocorrência de x
        // se inicialmente ans[d] <= 2^(cnt[d]),
        // agora ans[d] <= 2^(cnt[d] + 1)
        // logo, podemos pensar em somar 2^(cnt[d]) + 2^(cnt[d])
        // para ficar com algo <= 2^(cnt[d] + 1).
        if (add) {
            to_upd[d] = pow(2, cnt[d]),;
            cnt[d]++;
        }
        // caso contrário, se estamos subtraindo a contribuição de uma ocorrência de x
        // se inicialmente ans[d] <= 2^(cnt[d]),
        // agora ans[d] <= 2^(cnt[d] - 1)
        // logo, podemos pensar em subtrair 2^(cnt[d] - 1)
        // de 2^(cnt[d]) para ficar com algo <= 2^(cnt[d] - 1).
        else {
            cnt[d]--;
            to_upd[d] = pow(2, cnt[d]) * -1;
        }
    }

    // precisamos calcular to_upd[d] de verdade
    // o valor que devemos somar para corrigir ans[d]
    // assim, devemos tomar cuidado para não atualizar com contribuições
    // que devem ser aplicadas a multiplos de d, já que não necessariamente
    // to_upd[d] vai ser a potencia de 2 que definimos acima
    // Assim podemos fazer algo similar ao que fizemos anteriormente,
    // juntando todos os pares (divisor primo, divisor)
    for (int p : divisores_primos[x]) {
        for (int d : divisores[x] em ordem crescente) {
            if (d * p > x)
                break;
            to_upd[d] -= to_upd[d * p];
        }
    }

    // note que o trecho de código acima, seria equivalente a algo como:
    // for (int i : divisores[x] em ordem decrescente) {
    //   for (int d : divisores[i]){
    //       if (i > d)
    //           to_upd[d] -= to_upd[i];
    //    }
    // }
    // só que em uma complexidade melhor

    // corrige ans[d]
    for (int d : divisores[x])
        ans[d] += to_upd[d];
}
```

Algoritmo 8: Resolvendo atualizações no problema M.