Problem A. A Viagem de Papai Noel

O problema considerado corresponde ao Problema do Caixeiro Viajante (`TSP`).

Gerar todas as permutações de cidades possíveis e analisar o custo de cada ciclo, levaria $\Theta(N!)$, o que não é viável para grafos de tamanho 20.

A solução clássica de programação dinâmica *top-down* para o `TSP` pode ser modelada da seguinte forma. Seja S um conjunto de vértices e $w(i, j)$ o custo da aresta que sai de i para j (distância euclidiana entre as cidades i e j). Temos que $M(i, S)$, com $i \in S$ nos fornece o custo mínimo para sair da cidade 0, passar por todos os pontos descritos por $S \backslash \{i\}$ e chegar ao nó i. Isto pode ser descrito pela seguinte relação de recorrência:

$$M(i,S) = \begin{cases} 0, & S = \{0\} \wedge i = 0 \\ 0, & S = \{0, \dots, N-1\} \\ \min\{w(i,k) + M(i, S \cup k) | k \notin S \wedge k \neq i\} \end{cases}$$

Claramente a solução pode ser obtida em $\min\{M(k, X) + w(k, 0) | 0 \leq k < N\}$, com $X = \{0, \dots, N-1\}$.

Como o número de cidades é menor ou igual a 20, S pode ser representado através de um vetor de bits, cuja operação de pertinência, inclusão ou exclusão de um elemento do conjunto levam tempo $\Theta(1)$.

Como $|\mathcal{P}(S)| = 2^N$, existem $N2^N$ subproblemas. Cada subproblema tem um fator linear envolvido na sua resolução, portanto a solução baseada em programação dinâmica leva $\Theta(N^2 2^N)$.

Problem B. Buracos de Minhoca

A função $\varphi(n)$ retorna o número de inteiros positivos x tais que $\gcd(x, n) = 1$. Ela é uma função multiplicativa e, para um primo p e um inteiro positivo k,

$$\varphi(p^k) = p^{k-1}(p-1)$$

Estas propriedades permitem computar $\varphi(n)$ para todos os valores de $n \in [1, 4 \times 10^5]$ em $O(n \log n)$ por meio de um crivo de Erastótenes modificado. Estes valores permitem computar o número de frações p/q, com $p \leq q$, tais que $\gcd(p, q) = 1$. Exceto pela fração $1/1$, todas as demais podem ser invertidas, gerando uma nova fração q/p que também está em S_1. Logo,

$$S_1 = 2 \times \left(\sum_{i=1}^{N} \varphi(i) \right) - 1$$

Para os demais setores S_k, com $k > 1$, basta observar que se $\gcd(a, b) = k$, então $x/y \in S_1$, com $a = x/k, b = y/k$. Assim, qualquer elemento p/q de S_1 tal que $pk \leq N$ e $qk \leq N$ será um elemento de S_k. Portanto,

$$S_k = 2 \times \left(\sum_{i=1}^{N/k} \varphi(i) \right) - 1$$

Se as somas parciais dos valores de φ forem pré-computados, esta solução terá complexidade $O(N \log N)$.

Problem C. Cupins vs Tamanduás: A Revanche

Simular o processo, procurando linearmente pelo primeiro tamanduá capaz de consumir o cupim, ou inserindo um novo tamanduá se necessário, tem complexidade $O(N^2)$ e leva ao TLE (nota: a versão original do problema permitia tal solução).

A dificuldade reside justamente em identificar, de forma eficiente, o tamanduá que deve consumir o cupim, caso exista. Para isto, é necessário coordenar três estruturas de dados:

1. $ms[i]$: capacidade de consumo do i-ésimo tamanduá;

2. $ts[m, is]$: um dicionário que associa os índices is de todos os tamanduás que podem consumir exatamente m gramas de cupins;

3. $st[m]$: uma árvore de segmentos que armazena, no índice m, o menor identificador de um tamanduá capaz de consumir exatamente m gramas de cupins.

A árvore de segmentos permite, com uma consulta do menor elemento no intervalo $[m, M]$, identificar o tamanduá que consumirá o cupim. Caso ele exista, as três estruturas devem ser atualizadas devidamente; caso não exista, deve ser inserido um novo tamanduá nas três estruturas.

Esta solução tem complexidade $O(N \log N)$.

Problem D. Durval e os Treinos Intensos

O problema pede para maximizarmos a quantidade de trechos em uma estrada de terra que Durval deverá correr em alta intensidade. Cada trecho i é determinado por um quilômetro inicial s_i e um quilômetro final f_i. Durval deve correr em velocidade reduzida entre os trechos.

Podemos verificar uma restrição do problema: os trechos escolhidos (selecionados) devem ser disjuntos, ou seja, dois trechos escolhidos para seu treino (s_j, f_j) e (s_{j+1}, f_{j+1}) devem respeitar a condição de que $f_j < s_{j+1}$. Uma maneira simples de resolver o problema é por meio de uma abordagem gulosa. Podemos identificar a seguinte sub-estrutura ótima que funciona para o problema:

Seja um conjunto T que maximiza os trechos escolhidos para o treino de Durval e um trecho (s_j, f_j), em que f_j é mínimo. Se $(s_j, f_j) \in T$, então $T - \{(s_j, f_j)\}$ também é máximo, uma vez que todos os trechos previamente considerados são disjuntos e maximizam T.

Portanto, podemos formular o seguinte algoritmo:

- Ordenar os trechos em ordem crescente do quilômetro final;
- $fimUltimo \leftarrow f_1$;
- $resposta \leftarrow 1$;
- Para $i \leftarrow 2, ..., N$, faça:
  1. $inicioAtual \leftarrow s_i$;
  2. $fimAtual \leftarrow f_i$;
  3. **se** $inicioAtual > fimUltimo$;
  4. $\quad fimUltimo = fimAtual$;
  5. $\quad resposta \leftarrow resposta + 1$;

A variável resposta armazena a quantidade máxima de trechos que Durval poderá correr em alta intensidade, tornando o seu treino mais difícil.

Problem E. Eds e a Prova Perfeita 2

A solução é simples, basta contar as ocorrências de cada tópico nas questões e identificar a frequência máxima. O passo seguinte é filtrar apenas os tópicos que não atingem o mínimo de 75% da frequência máxima e apresentá-los na ordem definida no enunciado.

Devido à ordenação, esta solução é $O(N \log(N))$.

Problem F. Farejando Arrays Especiais

Vamos definir um subarray que não possui pelo menos um elemento que aparece exatamente uma vez como um **contraexemplo**.

Um array que **não** é especial contém pelo menos um contraexemplo. Assim, para determinar se o array de entrada é ou não especial, basta tentarmos encontrar qualquer contraexemplo.

Vamos supor que o elemento A_i aparece apenas uma vez no array de entrada; nesse caso, qualquer subarray que contenha a posição i **não** será um contraexemplo. Assim, supondo que achamos esse índice i, basta analisarmos os subarrays $A[1..i-1]$ e $A[i+1..N]$ para tentarmos encontrar contraexemplos, porque nenhum dos subarrays presentes nesses subarrays contêm o índice i. Podemos aplicar este mesmo raciocínio para a análise dos subarrays $A[1..i-1]$ e $A[i+1..N]$ e assim por diante. Isto nos faz tender a usar uma estratégia recursiva para encontrar um contraexemplo.

Vamos criar uma função booleana recursiva $solve(L, R)$. Se $solve(L, R)$ retornar $true$, então existe um contraexemplo $A[i..j]$ de tal forma que $L \leq i, j \leq R$. Isto é, o contraexemplo $A[i..j]$ é um subarray do subarray $A[L..R]$. A resposta do problema seria basicamente a chamada $solve(1, N)$.

O algoritmo em alto nível para $solve(L, R)$ funciona assim: encontramos qualquer índice i tal que $L \leq i \leq R$ e que o valor A_i apareça apenas uma vez em $A[L..R]$. Se encontrarmos esse índice, então o retorno para a chamada $solve(L, R)$ é $solve(L, i-1)||solve(i+1, R)$. Caso contrário, o retorno é $true$, porque o próprio subarray $A[L..R]$ é um contraexemplo, dado que todos os elementos dentro dele aparecem mais de uma vez.

O caso base para a recursão de $solve(L, R)$ acontece quando $L \geq R$; nesse caso, retornamos $false$, porque um subarray de tamanho 1 ou um subarray vazio não são contraexemplos.

Resta agora pensarmos como resolver cada um dos seguintes dois problemas de forma eficiente:

1. Dado um subarray $A[L..R]$ e um índice k tal que $L \leq k \leq R$, como determinamos se A_k aparece em $A[L..R]$ apenas uma vez de forma eficiente?

2. Dada uma chamada de $solve(L, R)$, como encontramos o dito índice i de forma eficiente?

Problema 1: basta precomputarmos dois arrays auxiliares $prev[]$ e $next[]$ em $O(N \log N)$, com duas passadas (uma de trás pra frente e outra de frente pra trás) no array original.

- $prev[i]$ nos diz o índice j mais próximo possível à esquerda de i tal que $A_i = A_j$. Isto é, queremos um j tal que $j < i$, que $i - j$ seja minimizado e que $A_i = A_j$. Em caso de tal índice não existir, fazemos com que $prev[i]$ seja igual a 0.

- $next[i]$ nos diz o índice j mais próximo possível à direita de i tal que $A_i = A_j$. Isto é, queremos um j tal que $j > i$, que $j - i$ seja minimizado e que $A_i = A_j$. Em caso de tal índice não existir, fazemos com que $next[i]$ seja igual a $N + 1$.

Com esses valores precomputados, podemos criar a função booleana $unique(i, L, R)$ que vai nos dizer se A_i aparece apenas uma vez em $A[L..R]$. A função é simples, basta fazer com que ela retorne o resultado da expressão booleana $prev[i] < L$ && $next[i] > R$. É fácil ver que a complexidade da função $unique$ é $O(1)$.

Problema 2: intuitivamente, podemos tentar iterar por todos os índices i tal que $L \leq i \leq R$ e, usando a função $unique$, assim que encontrarmos o primeiro índice tal que A_i é único, então chamamos as devidas recursões. O problema é que essa forma de buscar i tem pior caso elevado: $O(N\hat{}2)$. O pior caso acontece quando $i = R$ (isto é, iteramos em todo o subarray $A[L..R]$, o que tem custo linear), o que faz com o que o subarray $A[L..i-1]$ seja ainda muito grande. Como esse processo pode acontecer N vezes (porque R sempre diminui em 1), e o custo para cada processo é linear no tamanho do subarray atual, a complexidade ficaria quadrática. Intuitivamente, poderíamos tentar iterar de trás pra frente em vez de de frente para trás, mas o problema é o mesmo, porque poderíamos apenas reverter o array de entrada do pior caso para criar um caso de teste pesado.

Acontece que a solução para achar i da melhor maneira possível combina ambas as abordagens: iterar por $A[L..R]$ tanto de trás pra frente quando de frente para trás, de forma "paralela". Isto é, em vez de iterar pelos índices $\{L, L+1, L+2, ..., R\}$ ou pelos índices $\{R, R-1, R-2, ..., L\}$, iteramos uma vez na frente, depois uma vez atrás, depois uma vez na frente e assim sucessivamente, com a ordem $\{L, R, L+1, R-1, L+2, R-2...\}$. Acontece que o simples fato de iterar pelos índices de forma “paralela” em vez de forma crescente ou decrescente faz com que a complexidade final do algoritmo seja $O(NlogN)$ em vez de $O(N\hat{}2)$.

Para entender o porquê, temos que pensar quantas vezes a função *unique* vai ser chamada para um índice i qualquer em todas as chamadas de *solve* que ele estiver presente. A soma de todas as vezes que essa função é chamada para todos os N índices do array de entrada determina a complexidade final do algoritmo, pois essa é a operação mais básica que acontece dentro de *solve*.

Quando encontramos um índice i para uma chamada de $solve(L, R)$, segmentamos o subarray $A[L..R]$ em dois: $A[L..i-1]$ e $A[i+1..R]$. O menor desses subarrays tem tamanho praticamente igual à metade da quantidade de índices que foram iterados para encontrarmos i. Por exemplo, se $L = 3$ e $R = 10$, iteraríamos usando a seguinte ordem: $\{3, 10, 4, 9, 5, 8, 6, 7\}$. Supondo que $i = 8$, então iteramos 3 vezes “de cada lado”, totalizando 6 iterações: iteramos por $\{3, 4, 5\}$ do lado esquerdo e por $\{10, 9, 8\}$ do lado direito. Os subarrays segmentados seriam $A[3..7]$ e $A[9..10]$, e acontece que o menor deles ($A[9..10]$) tem tamanho ≈ 3, porque o menor dos subarrays sempre vai conter só os índices que foram iterados do seu lado respectivo. Podemos classificar os índices $\{3, 4, 5\}$ como índices que foram iterados pelo lado maior e os índices $\{10, 9, 8\}$ como índices que foram iterados pelo lado menor (porque o subarray $A[9..10]$ é o menor entre os dois). Outra importante observação: o menor entre os dois subarrays sempre vai ter, no máximo, metade do tamanho do subarray original $A[L..R]$.

Levando o parágrafo acima em conta, acontece que um índice qualquer do array vai ser iterado pelo lado menor no máximo $O(logN)$ vezes. Isso acontece porque sempre que ele é “iterado pelo lado menor”, o subarray em que ele vai estar presente na próxima chamada de *solve* é pelo menos 2 vezes menor que o subarray da chamada *solve* pai. Como só podemos dividir N pela metade $O(logN)$ vezes, esse é o número de vezes que ele vai ser iterado pelo lado menor. Se a complexidade final do algoritmo fosse dada pela quantidade de vezes que um índice do array é iterado pelo lado menor, então poderíamos dizer que a complexidade final do algoritmo é $O(NlogN)$. Porém, ainda falta contabilizarmos a quantidade de vezes que um índice do array é iterado pelo lado maior. Acontece que a ordem de grandeza que todos os elementos são iterados pelo lado maior é igual à ordem de grandeze que todos os elementos são iterados pelo lado menor. A explicação é simples: como estamos iterando “paralelamente” (isto é, uma vez no lado menor e uma vez no lado maior), então toda iteração no lado maior tem uma iteração par no lado menor. Como vimos que o número total de iterações no lado menor é $O(NlogN)$, então o número total de iterações no lado maior também é $O(NlogN)$, fazendo com que o algoritmo tenha complexidade final $O(NlogN)$.

Problem G. Gole Perfeito

Para resolver o problema, podemos considerar cada a_i e d_i como um ponto no plano cartesiano. Como o copo final pode não estar totalmente cheio, i.e., $\sum_{i=1}^{n} p_i \leq 1$, é preciso acrescentar o ponto $(0, 0)$ também.

Se fizermos o convex hull dos pontos, qualquer ponto dentro do polígono formado pode ser alcançando com alguma combinação linear dos vetores que formam os vértices do polígono. Logo, após montarmos o convex hull, é necessário ver se o ponto (A, D) se encontra dentro do polígono final. Caso não esteja, já podemos printar 'N'.

Caso esteja, o problema agora é achar os pesos de cada suco. Para isso, podemos ver que se o ponto (A, D) está dentro do polígono, ele estará dentro de algum triângulo! Para acharmos esse triângulo, basta analisar os triângulos com os pontos de índice (índice da convex hull) $0, i$ e $i + 1$, com i de 1 até $n - 1$. Vamos chamar esses 3 pontos de $(x1, y1)$, $(x2, y2)$ e $(x3, y3)$. Com isso, podemos montar as 3 equações:

$p1 + p2 + p3 = 1$

$p1 * x1 + p2 * x2 + p3 * x3 = A$

$p1 * y1 + p2 * y2 + p3 * y3 = D$

Para resolver este sistema, podemos utilizar eliminação gaussiana.

É necessário cuidar separadamente do caso onde conseguimos alcançar o copo final com somente um copo de suco.

Problem H. Helena e os Cheques

Testar cada um dos 2^N subconjuntos tem complexidade $O(N2^N)$ leva a um veredito TLE.

O problema pode ser resolvido por meio de um algoritmo de programação dinâmica, que consiste em uma variante da mochila binária. Seja $dp(x,i)$ o número de maneiras de se obter x reais utilizando os i primeiros cheques. O caso-base acontece quando $i=0$: $dp(0,0)=1$ e $dp(x,0)=0$, se $x>0$. São duas transições possíveis:

$$dp(x,i) = dp(x,i-1),$$

se $x_i > x$, ou

$$dp(x,i) = dp(x,i-1) + dp(x-x_i,i-1),$$

caso contrário. Uma implementação *bottom-up* pode reduzir a memória utilizada para $O(B)$.

Esta solução tem complexidade $O(NB)$.

Problem I. IMC aproximado

A solução do problema consiste na extração do valor após a vírgula, o qual será a massa m no cálculo do IMC, sendo todo o valor da entrada igual a altura h. É possível verificar, por busca completa, que para as restrições da entrada a resposta "Sim" ocorre apenas se $h \in [1.37, 1.88]$.

Problem J. Jankenpon

A pontuação obtida pelo i-ésimo jogador pode ser computada em $O(N)$. Assim, uma solução que computa as pontuações para cada um dos jogadores tem complexidade $O(N^2)$, e como $N \leq 2\times 10^5$, ela terá veredito TLE.

Observe que a pontuação obtida por todos os jogadores que fizeram a mesma opção será a mesma. Assim, basta computar três pontuações $p(R), p(P), p(S)$, uma para cada opção. Seja $M = \max\{\ p(R), p(P), p(S)\ \}$ e $h(c)$ o número de jogadores que optaram por c. A solução s então será a dada por

$$s = \sum_{\substack{c\in\{R,P,S\} \\ p(c)=M}} h(c)$$

Contudo, é preciso tomar cuidado no cálculo de M: a pontuação $p(c)$ só deve ser considerada se $h(c) > 0$. Esta solução tem complexidade $O(N)$.

Problem K. Kaboom

Qualquer árvore geradora do grafo terá $N-1$ arestas. Assim, os jogadores tentarão remover as $M-(N-1)$ arestas restantes. O vencedor pode ser determinado por meio de um algoritmo de programação dinâmica.

Seja $dp(i)=1$ se Ana consegue vencer a partida tendo i fios que ainda podem ser cortados, e $dp(0)=0$, caso contrário. O caso base acontece com $i=0$, onde $dp(0)=0$. Como ambos jogarão de forma ótima, a transição será dada por

$$dp(i) = \max\{\min\{dp(i-2a), dp(i-a-b)\}, \min\{dp(i-b-a), dp(i-2b)\}\},$$

isto é, Ana escolherá o melhor cenário entre escolher a ou b, sabendo que Beto também analisará suas opções e escolherá aquela que levará Ana à derrota.

Esta solução tem complexidade $O(N)$.

Uma solução alternativa é definir um estado $f(i) = True$, caso o próximo jogador consegue vencer a partida tendo i fios que ainda podem ser cortados, $False$ caso contrário. Ambos os jogadores, se possível, jogarão para um estado onde o oponente (que se torna o próximo a jogar) perde, assim

$$f(i) = !f(i-a) \text{ or } !f(i-b)$$

Problem L. Laços Infinitos

Salve prof. Lamar!

Este é um problema de simulação, a estratégia é literalmente interpretar o que se pede cada linha mantendo o contador de programa atualizado de acordo com o fluxo do programa. Deve-se atentar para o fato de que um programa que leva exatamente 10^5 instruções para encontrar a instrução `EXIT` não deve fazer com que o interpretador imprima a mensagem "`laco infinito!`". Outro ponto que requer atenção é na execução da operação `MOD`, que deve retornar o resto de acordo com o sistema canônico de restos, isto é, a operação $a \mod b$ deve gerar um inteiro no intervalo $[0, |b| - 1]$.

Problem M. Média Móvel

A solução é simples, calcular a média para cada n elementos. A forma mais eficiente é considerar uma janela de n elementos na sequência, que desliza ao longo dos m elementos de modo que a variação é calculada por simplesmente retirar o valor do último elemento e acrescentar o valor do próximo. Assim, apenas 2 a cada n elementos são computados.

Essa solução é $O(n)$.

Problem N. Natal e árvores

O par (a, b) é um estado possível se, e somente se, todos os vértices no menor caminho entre a e b são menores que a e b. Prova é deixada ao leitor.

Podemos facilmente contar os pares (x, x) separadamente. Vamos contar os estados possíveis (a, b) onde $a < b$ e no final multiplicamos por dois para obter os estados (a, b) com $a \neq b$.

Vamos fixar o a e contar quantos pares (a, x) com o $a < x$ existem. Isto é, queremos contar caminhos que começam em a, terminam em $x > a$ e todos os outros vértices no caminho sejam menores que a.

Vamos processar os vértices na ordem 1, 2, ..., $n - 1$, n. Para 1, a resposta será o grau do vértice. Para 2 temos dois casos. Caso tenha não tenha aresta para 1, a resposta será o grau de 2. Caso tenha uma aresta $(1, 2)$, a resposta para 2 será a quantidade de arestas saindo de $\{1, 2\}$ para o resto da árvore.

Depois de processar o vértice i ele vai poder ser usado por outros vértices para alcancar vértices com identificadores maiores. Podemos agrupar, então, os vértices já processados conectados e a resposta para o vértice i é o grau de seu grupo. Para agrupar vértices podemos usar a estrutura DSU.

Complexidade final $O(n \cdot \alpha(n))$.

Problem O. Observação Estelar

Este é um problema de casamento de padrões bidimensional. Uma abordagem força-bruta levaria tempo $\Theta(NMKL)$, o que é inviável para o tamanho das entradas.

O truque aqui envolvido é transformar o padrão P, que possui 2 dimensões, em um padrão P' unidimensional. Como o padrão tem tamanho no máximo 60×60 e o alfabeto considerado é binário, qualquer coluna do padrão pode ser representada por um inteiro de 64-bits através da técnica de *fingerprinting* de Rabin-Karp. No caso, a função de *fingerprint* é perfeita, não há colisões. O tempo de pré-processamento do padrão P' é $\Theta(KL)$.

O texto também tem que ser pré-processado de acordo com este método. Ou seja, para cada coluna $T[i..i + K - 1][j]$, criaremos uma entrada no texto $T'[i][j]$ com a codificação da coluna em binário. É fácil ver que podemos obter $T'[i+1][j]$ a partir de $T'[i][j]$ e de $T[i+1..i+K][j]$ com simples operações bit a bit em tempo constante. Desta forma, o tempo de pré-processamento do texto é $\Theta(NM)$.

Agora, basta procurar o padrão no o texto usando o algoritmo KMP, que requer tempo de processamento proporcional ao tamanho de P' e tempo de busca proporcional ao tempo de T'. Juntando tudo, temos que a complexidade desta solução é de $\Theta(KL + NM + K + NM) = \Theta(NM)$.

Problem P. Pamonhas

O problema estabele que existem D, S e F pamonhas de doce, sal e fit e que devemos produzir kits de mesmo tamanho e que devem existir uma quantidade máxima de pamonhas de um único tipo em cada tipo.

Tal proposição está relacionada ao conceito de máximo divisor comum. Por isso, a solução consiste em calcular o máximo divisor comum entre D, S e F. Por isso, a complexidade computacional dessa solução é $O(\log max(D, S, F))$.