Análises dos Problemas do Dia 11/08

- Problema A – Mike and Cellphone

  Há várias formas de resolver esse problema, que requer apenas uma implementação cuidadosa. O que eu recomendo fazer é identificar quais os botões que, se não pressionados, permitem uma translação das posições a outras válidas. Por exemplo, se o número do telefone não tem nem o dígito 1, nem o 2, nem o 3, é sempre possível transladar as posições digitadas uma célula para cima. Os outros conjuntos mínimos de teclas que permitem uma translação se nenhum dos seus elementos forem usados são os seguintes: $\{1, 4, 7, 0\}$, $\{3, 6, 9, 0\}$ e $\{7, 0, 9\}$.

- Problema B – Balé

  O problema pede para que, dada uma permutação $P = (P_1, P_2, ..., P_N)$, contemos o número de inversões nessa permutação, isto é, quantos pares (i, j) distintos existem tais que $i < j$ e $P_i > P_j$. Note que, como N pode assumir valores até 10^5, algoritmos que façam da ordem de N^2 operações não serão rápidos o suficiente.

  Uma forma eficiente de resolver o problema é construindo uma árvore de segmentos que consulta a soma em um intervalo arbitrário dos elementos de um vetor $V = (V_1, V_2, ..., V_N)$. Inicialize V com zeros em todas as entradas. Leia sequencialmente os elementos da permutação e, para cada P_i lido, consulte a soma dos elementos de V_{P_i+1} a V_N, some esse resultado à sua resposta e então atualize o valor de V_{P_i} para 1. A ideia é que, assim, quando você lê um valor, você imediatamente conta todos os números maiores que ele que já apareceram antes na permutação e, fazendo isso para todos, você conta todas as inversões. Como são feitas N consultas e N atualizações na árvore de segmentos (que é construída sobre um vetor de tamanho N), a complexidade do algoritmo é $O(N \log N)$.

  Existem também outras maneiras de resolver esse exercício, como por exemplo usando uma implementação adaptada do *mergesort* ou com uma estrutura chamada *Binary Indexed Tree*.

- Problema C – Branch Assignment

  Se um determinado braço da empresa está em um grupo de tamanho S, ele terá que enviar $S - 1$ mensagens até a central e também terá que receber $S - 1$ mensagens de lá, vinda dos outros braços. Assim, fica claro que o

custo total dos envios de um grupo só depende do tamanho do grupo e do menor caminho de ida e volta de cada braço até a central. Então, a primeira coisa que devemos fazer é rodar um algoritmo de caminho mínimo – como o de Dijkstra, por exemplo – e calcular todas as distâncias a partir da central e até ela.

Agora que temos as distâncias calculadas, o problema se reduz a uma versão ligeiramente modificada do problema *Guardians of the Lunatics*, discutido no segundo dia da Escola. Aqui, podemos considerar o vetor de distâncias d_i de ida e volta de cada braço à central como sendo o vetor dos valores de periculosidade dos prisioneiros naquele problema. Então, só precisamos implementar um algoritmo de programação dinâmica que decide onde dividir cada grupo e depois otimizá-lo usando, por exemplo, a técnica de divisão e conquista. Note que a única diferença entre os dois problemas é que, neste, o fator que multiplica o custo de cada membro de um grupo é o tamanho do grupo menos um, enquanto no problema dos lunáticos esse fator era simplesmente o tamanho do grupo, mas isso não altera em nada a ideia da solução.

- Problema D – Myacm Triangles

  A primeira coisa a notar aqui é que o número N de pontos na entrada é muito pequeno, então, não precisamos de um algoritmo muito eficiente para nossa solução. O que vamos fazer é o seguinte: iterar sobre cada tripla de pontos para formar um triângulo e, então, checar se nenhum dos outros $N - 3$ pontos está dentro desse triângulo. Se existe um dos pontos da entrada dentro do triângulo, podemos ignorar esse triângulo, caso contrário, computamos sua área, pois ele é um candidato a ser o "power triangle". A complexidade total do algoritmo é $O(N^4)$.

  Uma dica interessante aqui é que podemos calcular o dobro da área dos triângulos ao invés de calcular a área, porque assim evitamos o fator $1/2$ e garantimos que só teremos que lidar com números inteiros.

- Problema E – Encolhendo Polígonos

  Primeiro, note que, como todos os vértices estão sobre coordenadas inteiras, o perímetro p do círculo deve ser um múltiplo do número de lados do polígono encolhido. Para cada divisor d de p, verifique se existem d vértices separados por distâncias de p/d unidades de comprimento de arco. Se existem, essa é uma forma de encolher o polígono. Agora basta escolher a forma que passa por mais vértices.

  Para marcar onde estão os vértices do polígono, podemos usar um vetor de booleanos, em que o i-ésimo elemento é verdadeiro se, e somente se, existe um vértice na posição de i unidades de comprimento de arco, medidos a partir de uma origem arbitrária. Assim, fazendo acessos a posições nesse vetor separadas por p/d, podemos verificar eficientemente a possibilidade de construção do polígono de d lados.

- Problema F – Runner Pawns

  Podemos interpretar cada estado do tabuleiro como um nó de um grafo e fazer uma busca em profundidade a partir do estado atual até encontrar um estado em que todos os peões tenham sido capturados pelo cavalo. Para representar o estado, é necessário marcar a posição de todos os peões que ainda não foram capturados e também a posição do cavalo. Isso pode ser feito usando máscaras de bits ou mesmo um vetor. Perceba que sua busca não precisa ir muito longe, porque em 7 passos é garantido que todos os peões tenham atravessado o tabuleiro.

  Para implementar os movimentos do cavalo, é conveniente usar um vetor que precalcula todas as variações possíveis nas coordenadas e, depois, basta iterar nos elementos desse vetor para visitar todas as casas vizinhas da casa atual. Para ficar mais claro, verifique os arrays `dx[]` e `dy[]` da solução fornecida e como eles são usados para calcular as células alcançáveis para o cavalo a partir da atual.

- Problema G – Aggressive Cows

  Dada uma distância d, é fácil checar em $O(N)$ se é possível acomodar todas as vacas a uma distância de pelo menos d uma da outra: vá iterando pelas posições x_i em ordem crescente e, tão logo for possível posicionar uma vaca, posicione-a lá; se ao fim do processo você não conseguiu pôr todas as N vacas, essa distância d é muito grande.

  Claramente, se é possível acomodar as vacas a uma distância mínima de d entre quaisquer duas delas, isso também deve ser possível para qualquer distância menor que d. Então, podemos fazer uma busca binária na resposta, isto é, vamos "chutando" valores para a maior distância possível. Se a distância que chutamos é pequena o suficiente, podemos tentar uma um pouco maior; se a distância já é grande demais, temos que diminuí-la. Vamos fazendo isso sempre reduzindo o espaço de busca pela metade para garantir a eficiência da busca. A complexidade final do algoritmo é de $O(N \log(\max\{x_i\}))$.

- Problema H – Restaurant Tables

  Só precisamos simular o que acontece no restaurante com a chegada de todos os clientes conforme o que é dado na entrada. Tome cuidado para que sua simulação siga exatamente as regras do enunciado, muitas pessoas implementaram um algoritmo que trata equivalentemente as mesas de dois lugares com um ocupado e as mesas de um lugar. Isso não é verdade! O enunciado explica que o restaurante, primeiro, tenta alocar os clientes em mesas completamente vagas, então, você tem que diferenciar esses três tipos de mesas: as de um lugar, as de dois lugares vagos e as de dois lugares com um ocupado.

- Problema I – Trapézio

  A princípio, não parece fácil decidir qual trapezista devemos pôr no topo: é vantajoso pôr aqueles com grande força porque eles são os que têm mais chance de aguentar o peso de todos os que estão por vir, mas também é bom pôr logo aqueles com grande peso, porque então menos trapezistas vão ter que suportar muito peso. Ainda que pareça difícil conciliar essas duas condições, é possível usar uma estratégia gulosa para resolver o problema. Escolha os trapezistas a partir daquele que tem a maior soma $P + F$ até aquele com a menor dessas somas. Se, seguindo essa ordem, algum deles tiver a força menor do que a soma do peso de todos os restantes, é impossível encontrar uma configuração válida.

  Para ajudar a ganhar intuição sobre esse problema, vamos provar que a solução localmente ótima é, de fato, escolher o trapezista com maior $P+F$. Para isso, vamos considerar dois trapezistas, i e j, tais que a soma do peso e força de i é maior que a de j, isto é,

  $$P_j + F_j < P_i + F_i. \tag{1}$$

  Agora, se o trapezista i não é capaz de sustentar os pesos de todos os outros trapezistas, temos:

  $$F_i < \sum_{k \neq i} P_k. \tag{2}$$

  Somando, membro a membro, as desigualdes (1) e (2):

  $$F_i + P_j + F_j < P_i + F_i + \sum_{k \neq i} P_k$$

  $$\Rightarrow \qquad F_j < \sum_{k \neq j} P_k, \tag{3}$$

  ou seja, o trapezista j, com soma de peso e força menor que a de i, também não é capaz de sustentar todos os outros pesos. Isso mostra que, se existe pelo menos um trapezista que pode ser posicionado no topo do trapézio, aquele com maior $P + F$ é um deles.

- Problema J – Buying a House

  Itere sobre todas as casas. Se o preço de uma delas for maior do que k, ignore-a; caso contrário, calcule a distância dessa casa até a casa da garota. Dentre todas as distâncias calculadas, a resposta final é a menor delas.

- Problema K – Homem, Elefante, Rato

  Se o jogo tivesse apenas duas posições – elefante e rato, por exemplo – poderíamos construir um vetor binário que marcasse quais os jogadores jogam a posição elefante e usar uma árvore de segmentos para consultar a soma em um intervalo, o que nos daria quantos elefantes há naquele intervalo. Entretanto, como temos 3 posições possíveis, devemos guardar mais

informações no nó da árvore. Podemos simplesmente fazer com que cada nó conte o número de homens, elefantes e ratos naquele intervalo, funcionando como uma árvore de segmentos de soma comum, mas operando com três informações independentes.

Como as atualizações podem ser pedidas em intervalos grandes, lembre-se de que temos que usar a técnica de "lazy propagation" para que o programa não exceda o tempo limite.

- Problema L – Find the Bone

  Guarde a posição de todos os buracos e simule o processo de troca dos copos, sempre acompanhando onde está o o osso em cada passo. Se o osso chegar a uma posição com um buraco, termine a simulação, porque ele não vai mais mudar a posição.