Problema A. ?Adreuqse arap atierid

Este problema foi inspirado na definição real de algoritmos bilaterais do Unicode. Mais especificamente pela palestra Wait, ?tahW: The Twisted Road to Right-to-Left Language Support de Moriel Schottlender (`https://www.youtube.com/watch?v=OCQd02hORJQ`).

A solução do juiz utiliza de três passadas no texto e alguns vetores auxiliares:

- Um vetor é criado que indica qual a direção do caractere naquela posição, podendo ser $+1$ ou -1. Visto que o primeiro caractere sempre tem direção definida, é só questão de avaliar se o caractere atual é uma letra latina minúscula e colocar $+1$, ou se for uma letra latina maiúscula, -1, ou senão, copiar a direção do caractere anterior.
- Em seguida, é caminhado no texto criando um intervalo $[l, r)$. l é inicialmente 0, e o r é iterado. O último caractere neutro é guardado em uma variável chamada nnr. Assim que a direção do texto for alterada, o l é colocado em uma nova posição. Se a troca da direção é feita de -1 para $+1$, invertemos aquela parte do texto, desde l até nnr. A inversão leva em conta os caracteres invertidos como `‘(’` para `‘)’`.
- No final, é feito um processo parecido em busca dos conjuntos de caracteres fracos que foram invertidos e devem ser agora desinvertidos. Usando de um vetor que diz se uma seção foi invertida, é procurado por um um segmento $[l, r)$ de caracteres fracos invertidos, que são invertidos novamente. É necessário prestar atenção para que não seja desinvertido uma parte do segmento que anteriormente não havia sido invertido.

A complexidade desta solução é efetivamente $\mathcal{O}(n)$, usando técnicas de *two pointers*. Isto porque cada segmento $[l, r)$ em cada passo invertido é disjunto de todos os outros segmentos, e os laços iteram apenas em uma direção até o fim do texto.

Soluções usando apenas um laço com vários laços aninhados fazendo as diversas inversões de forma mais direta são possíveis.

Problema B. Biblioteca Alfabetizada

Este problema foi inspirado pelo jogo Ex Libris (`https://boardgamegeek.com/boardgame/201825/ex-libris`).

O *grau de desorganização* descrito pelo problema também pode ser chamado de número de inversões. Como a ordenação dos livros é feita de forma lexicográfica, é questão de criar um vetor de pares associando o título de cada livro com o seu índice, ordenar este vetor, e extrair os índices, obtendo então uma permutação em que se podem contar as inversões. Pode-se dizer que fazendo esse processo estamos “comprimindo” os títulos de livros originais em números comparáveis.

Existem algumas maneiras diferentes de resolver o problema de contar inversões, incluindo o uso de árvores de Fenwick, árvore de segmentos ou MergeSort. Cuidado apenas para não tentar “reiniciar” as árvores por completo (até o limite de 10^5 livros) em cada nova biblioteca, já que esse processamento vai passar do tempo limite (10^{10} operações), restrinja-se a zerar apenas o intervalo que será utilizado. Todas elas rodam em tempo $\mathcal{O}(n \lg n)$, que não é mais do que já foi usado para ordenar cada biblioteca.

Em seguida, obtido a contagem de inversões de cada biblioteca, vem a parte talvez mais desafiadora que é entender como que os K bibliotecários são distribuídos. O enunciado quer que você os distribua de tal maneira:

- Todas as bibliotecas desorganizadas recebem um bibliotecário inicialmente.
- Os bibliotecários restantes devem ser divididos (arredondado para baixo) conforme a proporção: número de inversão da biblioteca i dividido pela soma de todos os números de inversão.
- Em seguida, sobrará, no máximo, n bibliotecários. Cada um destes deve ser encaminhado a uma biblioteca diferente, seguindo a ordem daquelas que tem maior *grau de desorganização*, e como critério de desempate, o índice menor.

Fazendo essas contas usando `long long`, obtemos um valor para cada biblioteca que soma K. Lembre-se que se todas as bibliotecas estiverem organizadas, é só imprimir n zeros, se você tentar dividir por 0, vai receber um erro em tempo de execução.

Problema C. Cordéis Incompletos

Este problema foi inspirado pelos cordéis que contam a história de Lampião.

Uma garantia importante do enunciado é que a folha da capa sempre estará presente e que os números de página sempre são consistentes. Então, apenas rejeitamos cordéis que tenham as dimensões erradas (lembrando que cada face de uma folha de cordel tem duas páginas).

Para descobrir os cordéis faltantes, podemos notar que a capa tem uma propriedade importante: Ela sempre possui os números de página 1, 2, $(4g-1)$ e $4g$, onde g é o número de folhas do cordel. Também é importante que cada folha pode ser unicamente identificada pelo menor número de página. A página de capa por exemplo é a primeira folha, e uma folha que tem números 3, 4, $(4g-3)$ e $(4g-2)$ é a segunda folha, etc.

Então, o segredo é descobrir o número g que pode ser encontrado na folha de capa (que é a que tem o maior número de página entre todas as folhas, $4g$), e descobrir quais folhas estão presentes identificando elas pelo menor número de página presente naquela folha.

Então, é só imprimir as que não estão presentes, que sempre tem números de página $(2i-1)$, $2i$, $(4g-2i-3)$ e $(4g-2i-2)$ para $1 \leq i \leq g$, ou alternativamente, $(2i+1)$, $(2i+2)$, $(4g-2i-1)$ e $(4g-2i)$ para $0 \leq i < g$.

A solução é $\mathcal{O}(n+g)$ que é $\mathcal{O}(n)$.

Problema D. Dígitos Verificadores

Este problema relata uma história real, a história de Kushim que pode ser encontrada em um vídeo do Matt Parker (`https://www.youtube.com/watch?v=MZVs6wF7nC4`).

É necessário uma estrutura de dados que possa fazer soma em intervalo e sobrescrever o valor em uma posição de forma rápida. Pelo menos duas podem ser usadas para este papel: Árvore de segmentos e árvore de Fenwick (que é usada na solução do juiz), que permitem que estas duas operações sejam feitas em $\mathcal{O}(\lg n)$. Em cada posição da estrutura de dados, inicialmente coloca-se cada dígito da fita, $\mathcal{O}(n \lg n)$.

Com as propriedades enunciadas, para verificar uma soma ou multiplicação, basta somar os dígitos de cada operando do intervalo $\pmod 9$, em seguida realizar a operação com os dois resultados, e por fim, comparar com o dígito de verificação dado. Para colocar um dígito na posição em uma árvore de Fenwick, é necessário manter o vetor original e computar um Δd para mudar o valor naquela posição.

A solução então possui complexidade $O(n \lg n + q \lg n)$.

Problema E. Estrelas Constelares

Este problema é basicamente uma aplicação do fecho convexo (*convex hull*). A solução do juiz utilizada de uma implementação de fecho convexo *monotone chain*. É fácil perceber que as linhas das constelações não importam, apenas seus pontos que são utilizados para extrair um fecho convexo. Para eliminar elementos duplicados, a solução utiliza-se de `set`s.

Como o fecho convexo tem complexidade $\mathcal{O}(n \lg n)$ para n pontos, a complexidade da solução é de $\mathcal{O}(cl \lg l)$.

Problema F. Fofoca Corre Solta

Prefácio: Provavelmente existe uma solução mais rápida para este problema. Podemos considerar que os limites aplicados aqui tornam o problema "fácil".

Este problema foi inspirado pela matéria de Sistemas Distribuídos.

A solução do juiz consiste em criar um grafo como mostrado na figura a partir do formato desorganizado de entrada. Cada vértice do grafo representa um produto cartesiano entre as pessoas u e seus respectivos índices de acontecimento i, obtendo assim um grafo com nm vértices.

Primeiramente, as arestas correspondentes a ações subsequentes da mesma pessoa são colocadas logo no processamento da entrada.

Em seguida, é feito um processamento que consiste em uma busca em profundidade utilizando de um vetor de espera de recebimento `wait_recv`, um vetor de espera de envio `wait_send` e um vetor de posição `pos`. O funcionamento consiste em iterar até o final utilizando o vetor `pos` (de forma que se o processo aparecer de novo para ser processado, ele continua de onde parou), fazendo a seguinte rotina:

- Assim que um envio é encontrado, é visto se o destinatário já está esperando uma mensagem do nosso processo. Se sim, "quitamos" o envio e o recebimento ao mesmo tempo e fazemos essa ligação no grafo. Senão, sinalizamos que estamos esperando um envio e enfileiramos o processo para qual esperamos enviar e nós mesmos, e abortamos a iteração para processar a fila.
- Assim que um recebimento é encontrado, é visto se o remetente já enviou uma mensagem para o nosso processo. Se sim, "quitamos" o envio e o recebimento ao mesmo tempo e fazemos essa ligação no grafo. Senão, sinalizamos que estamos esperando um recebimento e enfileiramos o processo para qual esperamos receber e nós mesmos, e abortamos a iteração para processar a fila. Isto é uma forma simétrica do que acontece acima.

Para realizar as consultas, fazemos duas operações de busca em profundidade no grafo direcionado construído acima. Se é possível atingir o vértice (j, f_j) por meio de (i, f_i), significa que i aconteceu antes. Se é possível atingir o vértice (i, f_i) por meio de (j, f_j), significa que j aconteceu antes. Se é impossível de atingir os vértices nesses dois casos, não podemos concluir nada sobre a ordem dos acontecimentos.

Visto que fazemos essas buscas em profundidade a cada consulta, nossa solução tem complexidade $O(qnm)$.

Problema G. Glob Só Que Louco

Uma solução simples para este problema é o uso de bibliotecas de `regex`. Elas podem ser encontradas nativamente em Python, ou em C++17. A entrada pode ser adaptada para uma expressão regular que faz exatamente o trabalho esperado pelo problema.

A solução alternativa (ou esperada), é o uso de programação dinâmica. No artigo Wildcard Pattern Matching do GeeksForGeeks (`https://www.geeksforgeeks.org/wildcard-pattern-matching/`), é possível encontrar exatamente o algoritmo utilizado pelo juiz. As adições são em lidar com os conjuntos de caracteres, que são representados por `set`s. A solução portanto é $\mathcal{O}(qnm)$.

Problema H. Horas Nlogônicas

Podemos perceber que o número de "ponteiros iguais" é igual ao maior divisor comum entre os três números. Para computar os "ponteiros iguais", é só preciso iterar sobre todas as horas, e verificar se existe um minuto e um segundo que divide exatamente (não deixa resto) relativo a aquela hora. A solução é $\mathcal{O}(h)$.

Problema I. Índice de Arquivos

O enunciado exige que seja mantida a propriedade temporal dos arquivos. Isso significa que só podemos mover arquivos que estão no final da fila de arquivos no topo. Isso permite criar uma solução com um algoritmo ganancioso.

Primeiramente, lemos os arquivos e calculamos o espaço entre o último e o atual. Lembre-se de usar `long long` para representar números da ordem de 10^{10}. Em seguida, executamos o algoritmo testando mover todos os n arquivos, partindo do final. Quando o tempo necessário exceder o necessário, imprimimos o espaço que conseguimos liberar com aquele tempo.

Esta solução é $O(n)$. É possível implementar o segundo laço em $O(\lg n)$, fazendo busca binária no espaço necessário, porém, é desnecessário já que já foi necessário um laço $O(n)$ para ler a entrada e complica a implementação.