Problem A. Troca de Bicicletas

Tópicos: implementação

Autor: Passinho

Problema:

Definimos a sequência $p_i^n = p_{p_i^{n-1}} \forall n \geq 1$ e $p_i^0 = i$. O problema consiste em, para cada i de 1 até n, imprimir uma linha com k inteiros separados em uma linha: $p_i^0, p_i^1 \dots p_i^k$. Onde k é o menor inteiro não-negativo com $p_i^{k+1} = i$.

Solução:

Para resolver o problema, basta guardar os valores de p em um vetor e, para cada valor de i, imprimir a sequência dele. Podemos fazer isso com um loop que calcula cada p_i^j em cada iteração a partir do valor da última iteração e terminar o loop quando o próximo valor for i.

Exemplo de implementação em Python:

```
t = int(input())

for _ in range(t):
    n = int(input())
    p = [0] + [int(x) for x in input().split()]

    for i in range(1, n + 1):
        ans = []

        ans.append(i)
        u = p[i]

        while u != i:
            ans.append(u)
            u = p[u]

        for x in ans:
            print(x, end = ' ')
        print()
```

Problem B. Cruzamento

Tópicos: guloso, ordenações

Autor: Lucca Nunes

Problema:

Dado uma sequência de alturas $H = \{H_1, H_2, ..., H_N\}$, N par, queremos organizar as alturas em $N/2$ pares (H_A, H_B) de forma a minimizar o maior valor de $\frac{1}{2}(H_A + H_B)$.

Solução:

Primeiramente, notemos que minimizar $\frac{1}{2}(X + Y)$ é equivalente a minimizar $X + Y$. Assim, podemos reduzir o problema a minimizar a maior soma das alturas em um par.

Considere agora a sequência de alturas $\{H_1, H_2, ..., H_N\}$ ordenada de forma **não-decrescente**, isto é, $H_i < H_{i+1}$ para todo i de 1 a $N - 1$.

Buscando minimizar a soma máxima, parece intuitivo parear H_1 com H_N, H_2 com $H_N - 1$, e assim por diante. Isso de fato nos fornece uma solução ótima, mas vejamos o porque.

Considere dois pares quaisquer (A, B) e (C, D) ($A \leq B$ e $C \leq D$) em uma solução ótima e suponha que eles não estejam pareados de acordo com a nossa ideia gulosa, ou seja, $A \leq C$ e $B \leq D$.

Vamos provar que trocar os pares para (A, D) e (B, C) não piora a resposta, isto é, não aumenta o valor da soma máxima em um par.

Para o primeiro pareamento, o valor da maior soma é $C + D$, pois supomos que $C \geq A$ e $D \geq B$.

Já para o novo pareamento, temos duas opções:

1. O par (A, D) possui a maior soma

2. O par (B, C) possui a maior soma

No primeiro caso, a maior soma vale $A + D$. Porém, note que, como $A \leq C$, $A + D \leq C + D$. Logo, a soma máxima não aumenta.

Similarmente, no segundo caso, a maior soma vale $B + C$. Porém, como $B \leq D$, $B + C \leq C + D$ e, novamente, a soma máxima não aumenta.

Concluímos, portanto, que o algoritmo guloso proposto sempre fornece uma resposta ótima.

Problem C. Zigue-Zague

Tópicos: matemática, geometria

Autor: Naim Santos

Organização: Daniel Hosomi

Problema:

Dado o lado D de um quadrado e uma distância máxima M, queremos encontrar a posição (X, Y) final após percorrer M unidades de medida alternando pelas diagonais do quadrado, partindo do canto inferior esquerdo de um quadrado em $(0, 0)$.

Solução:

Podemos considerar cada segmento da corda como a diagonal de um quadrado de lado D. Pelo Teorema de Pitágoras, sabemos que essa diagonal vale $L = D \cdot \sqrt{2}$.

Assim, podemos contar quantos por quantos quadrados completos Amy vai passar. Essa quantidade é dada por $Q = \lfloor \frac{M}{L} \rfloor$.

Logo, a coordenada final X vai ser $Q \cdot D$ somado à uma fração da diagonal do último quadrado.

Como Amy aguenta andar M metros e já contamos $Q \cdot L$, restam $M - Q \cdot L$. Aplicando novamente o Teorema de Pitágoras, sabemos que isso corresponde à diagonal de um quadrado de lado $K = (M - Q \cdot L)/\sqrt{2}$ e, portanto, $X = Q \cdot D + K$.

Para encontrar a coordenada Y, precisamos considerar dois casos:

1. Após passar pelos Q quadrados completos, Amy terminou no prédio "de cima" (Q é ímpar)

2. Após passar pelos Q quadrados completos, Amy terminou no prédio "de baixo" (Q é par)

No primeiro caso, ela está na altura D e desce K unidades de medida, terminando em $Y = D - K$.

No segundo caso, ela está na altura 0 e sobe K unidades de medida, terminando em $Y = K$.

Exemplo de implementação em Python:

```
d, m = [int(x) for x in input().split()]

sqr = 2**.5 # Raiz de 2

l = d*sqr

q = m // l

k = (m-q*l)/sqr

x = q*d + k
```

```
y = d-k if q % 2 else k
print(x, y)
```

Problem D. Constante Mística do Universo

Tópicos: matemática

Autor: Daniel Hosomi

Problema:

Dada a equação $F = \frac{G \cdot M_1 \cdot M_2}{D^2}$ e os valores das variáveis (exceto por G), queremos encontrar o valor de G.

Solução:

Podemos isolar G na fórmula dada para calcular seu valor diretamente:

$$F = \frac{G \cdot M_1 \cdot M_2}{D^2} \implies G = \frac{F \cdot D^2}{M_1 \cdot M_2}$$

Exemplo de implementação em Python:

```
m1, m2 = [float(x) for x in input().split()]
x1, x2 = [float(x) for x in input().split()]
f = float(input())
d = x1 - x2
g = (f * d*d) / (m1 * m2)
print(round(g, 10))
```

Problem E. Código

Tópicos: implementação, condicionais

Autor: Naim

Problema:

Você tem uma mensagem de 7 bits mais um bit de segurança, que é 1 se e somente se a quantidade de bits iguais a 1 na mensagem for ímpar.

Solução:

Podemos ver a contagem dos primeiros 7 bits e checar se o último bit faz sentido.

Note que podemos ainda dizer que se a quantidade total de bits for ímpar com certeza a mensagem foi corrompida!

Exemplo de implementação em Python:

```
a = [int(x) for x in input().split()]
print('S' if sum(a) % 2 != 0 else 'N?')
```

Problem F. Mensagem

Tópicos: bitmask, contagem

Autor: Passinho e Naim

Problema:

Dado uma string, queremos contar quantas substrings podem ter seus caracteres rearranjados de forma a se tornar um palíndromo.

Solução:

Pensemos primeiro em como dizer se uma string pode ter seus caracteres rearranjados de forma a se tornar um palíndromo.

Vamos pensar no que caracteriza um palíndromo. Como dito no enunciado, um palíndromo é uma palavra que se lê igual de traz pra frente. Ou seja, a primeira letra é igual à última, a segunda letra é igual à penúltima e assim por diante.

Pensando na estrutura de um palíndromo, podemos concluir que a frequência de cada caractere deve ser par, exceto, possivelmente, do caractere central, caso a string tenha tamanho ímpar.

Logo, podemos resumir o problema a contar quantas substrings possuem frequências pares de cada caractere, exceto no máximo um.

Isso já nos fornece um algoritmo lento: podemos iterar sobre todas as substrings e fazer a verificação das frequências de seus caracteres. No entanto, esse algoritmo teria complexidade $O(N^2)$, que não é rápido o suficiente para as restrições dadas.

Para otimizar essa solução, podemos guardar para cada posição da string a paridade da frequência de cada letra, usando uma **bitmask** para cada posição. Nessa bitmask, a i-ésima posição estará acesa se a i-ésima letra (de 'a' a 'h') tem paridade ímpar, e apagada do contrário. Além disso, guardamos também um vetor $freq$ com as frequências de cada bitmask, isto é, $freq[i]$ guarda quantas vezes a mask i apareceu até o momento (vamos construindo esse vetor gradualmente, de $i = 1$ até $i = N$.)

Assim, fixada uma posição j, precisamos contar quantas posições $1 \leq i < j$ atendem aos requisitos para que a substring do intervalo $[i + 1, j]$ seja palíndroma. Sabemos que essas posições são aquelas tais que $mask_j \oplus mask_i$ tem no máximo um bit aceso, sendo $\oplus$ a operação de XOR binária. Logo, podemos iterar sobre todas as máscaras (são no máximo 8, devido ao tamanho do alfabeto considerado) que atendem isso e somar à resposta a frequência de cada máscara.

Note que, como o número de substrings pode ser da ordem de N^2, precisamos utilizar long long em C++.

Problem G. Museu de flores

Tópicos: guloso, implementação

Autor: Bernardo Archegas, Lucca Nunes

Problema:

Dados alguns intervalos $[L_i, R_i]$, queremos maximizar a soma dos produtos das frequências dos caracteres em cada intervalo.

Solução:

Vamos provar que uma dessas duas strings binárias: 0101010..., 1010101... sempre gera uma resposta ótima.

Seja A o número de caracteres '0' no intervalo e B o número de caracteres '1'. Sem perda de generalidade, vamos assumir que $A \leq B$. Podemos mostrar que, para maximizarmos o produto AB, sendo N o tamanho do intervalo considerado, então $A = \left\lfloor \frac{N}{2} \right\rfloor$ e $B = \left\lceil \frac{N}{2} \right\rceil$.

Isso é verdade pois se $A < B$, então temos que $(A + 1)(B - 1) \geq AB$. Então, note que sempre que adicionamos uma unidade em A, subtraímos uma unidade de B e mantemos a restrição $A + 1 <= B - 1$, o produto sempre cresce. Portanto, é fácil notar que A deve ser igual a $\left\lfloor \frac{N}{2} \right\rfloor$ e B deve ser igual a $\left\lceil \frac{N}{2} \right\rceil$. Uma forma de manter essa igualdade para todos os intervalos ao mesmo tempo é usando uma das strings binárias definidas acima.

Problem H. Coleção de moedas

Tópicos: ordenações, matemática, implementação

Autor: Lucca Nunes

Problema:

Dado uma sequência $A = \{a_1, a_2, ..., a_n\}$, queremos encontrar uma permutação $P = \{p_1, p_2, ..., p_n\}$ de A tal que $p_i + p_{i+1} = S_i$ para $1 \leq i < n$

Solução:

Como S representa a soma dos valores adjacentes em P, temos $S_1 = p_1 + p_2$, logo, $p_2 = S_1 - p_1$. Similarmente, $p_3 = S_2 - p_2 = S_2 - (S_1 - p_1) = S_2 - S_1 + p_1$ e $p_4 = S_3 - p_3 = S_3 - S_2 + S_1 - p_1$.

De forma genérica, para $i > 1$, temos:

$$p_i = S_{i-1} - p_{i-1} = S_{i-1} - S_{i-2} + S_{i-3} - ... \pm p_1 = \sum_{k=1}^{i-1}[(-1)^{k+1} \cdot S_{i-k}] \pm p_1$$

, sendo que o sinal de p_1 depende da paridade de i:

$$p_i = \textstyle\sum_{k=1}^{i-1}[(-1)^{k+1} \cdot S_{i-k}] - p_1 \text{ se } i \text{ for par e}$$
$$p_i = \textstyle\sum_{k=1}^{i-1}[(-1)^{k+1} \cdot S_{i-k}] + p_1 \text{ se } i \text{ for ímpar.}$$

Para simplificar a notação, adotemos $c_i = \sum_{k=1}^{i-1}[(-1)^{k+1} \cdot S_{i-k}]$.

Ou seja, a sequência P é da forma $\{p_1, c_2 - p_1, c_3 + p_1, ..., c_n \pm p_1\}$, isto é, nas posições pares temos uma constante subtraída de p_1 e nas posições ímpares uma constante somada a p_1.

Dessa forma, é possível observar que a escolha de p_1 determina todo o resto da permutação, já que c_i depende apenas dos valores de S e não dos valores de P. Essa observação já nos fornece uma solução $\mathcal{O}(n^2)$: para cada a_i, testamos $p_1 := a_i$, construímos o resto da permutação em $\mathcal{O}(n)$ e verificamos se a sequência encontrada é de fato uma permutação de A.

Para otimizar essa solução, precisamos de mais uma observação. Seja x o índice do menor valor de P. Sabemos que $p_x = c_x \pm p_1$.

Além disso, dados dois índices pares distintos i e j, sabemos que $p_i = c_i - p_1$ e $p_j = c_j - p_1$. Logo, sabemos comparar p_i e p_j:

$$p_i < p_j \iff c_i < c_j$$

Assim, conseguimos ordenar crescentemente as posições pares com base nos valores de c_i: o menor valor de c_i corresponde ao menor valor de p_i. O mesmo raciocínio pode ser aplicado para as posições ímpares. Feito isso, basta observarmos que p_x deve ser o menor valor das posições pares ou o menor valor das posições ímpares. Podemos então construir as duas sequências em $\mathcal{O}(n)$ e imprimir aquela que gere a solução correta (é possível que as duas sejam válidas).

Complexidade final: $\mathcal{O}(n \cdot log(n))$ ou $\mathcal{O}(n)$, a depender da implementação.

Problem I. Complicações logísticas

Tópicos: matemática, sets, árvore de segmentos

Autor: Fernando Morato, Naim Santos

Problema:

De forma mais simples, dadas várias atualizações de casas sendo ocupadas ou desocupadas, queremos saber qual é a menor soma de distâncias entre casas ocupadas e algum ponto da reta.

Solução:

Primeiro, temos que perceber que o ponto que minimiza a soma das distâncias é a mediana dos pontos ocupados. Vamos provar.

Suponha que o ponto ótimo p não é a mediana. Vamos chamar de E a soma das distâncias de pontos menores que p e de D a soma das distâncias de pontos maiores que p. Portanto, a soma das distâncias para este caso é $E + D$. Suponha que o ponto p é menor que a mediana. Vamos analisar o que aconteceria se, neste caso, escolhêssemos o ponto $(p + 1)$ como local ótimo para posicionarmos nossa loja.

Se E' é a soma das distâncias dos pontos menores que $(p+1)$ e D' é a soma das distâncias dos pontos maiores que $(p+1)$ então, $E' = E + q_e + x$ e $D' = D - q_d$, onde q_e é a quantidade de pontos à esquerda de p, q_d é a quantidade de pontos à direita de p e x é a quantidade de pessoas que estão na casa da posição p. Note que x é ou 0 ou 1. Portanto, a resposta para este caso é $E' + D' = E + q_e + x + D - q_d$. Como p é menor que a mediana, então sabemos que $q_d > q_e$. Então podemos concluir que, se p for menor que a mediana, então $E' + D' <= E + D$.

O caso em que p é maior que a mediana é análogo. Logo, está provado que o ponto que minimiza a soma das distâncias é a mediana.

Com essa informação em mãos, basta encontrarmos um jeito eficiente de manter a mediana enquanto fazemos as atualizações. Para isso, podemos utilizar a estrutura de dados *set* do C++. A ideia aqui é manter dois sets A e B, onde a mediana é ou o maior elemento de A ou o menor elemento de B. Para fazermos isso, por simplicidade, vamos assumir que os sets possuem tamanho pelo menos 1. Ao receber uma atualização do tipo $+$, verificamos se o maior elemento do set A é maior que o número recebido. Se for, adicionamos esse valor em A e caso contrário, adicionamos em B. Ao recebermos uma atualização do tipo -, verificamos se o maior elemento de A é maior ou igual ao número recebido. Se for, removemos esse valor de A e caso contrário, removemos de B.

Note que, se a diferença dos tamanhos dos sets for maior que 1, precisamos consertá-los de modo a manter a propriedade descrita anteriormente. Podemos fazer isso da seguinte forma. Se o maior dos sets for o A, removemos o seu maior elemento e o adicionamos no set B. Se o maior dos sets for o B, removemos o seu menor elemento e o adicionamos no set A. Agora, garantimos que a diferença entre o tamanho dos sets é no máximo 1. Por isso, para acharmos a mediana basta imprimirmos o maior elemento de A, se este for maior ou igual a B, ou o menor elemento de B, caso contrário. Além disso, precisamos manter a soma dos elementos contidos em cada um dos sets.

Então, resposta fica $T_a * M - S_a + S_b - T_b * M$, onde T_a é o tamanho de A, T_b é o tamanho de B, S_a é a soma dos elementos contidos em A, S_b é a soma dos elementos contidos em B e M é a mediana dos pontos ativos. Por conta do uso de sets, a complexidade fica $\mathcal{O}(QlogQ)$.

OBS: Também podemos resolver esse problema utilizando Fenwick Trees ou Segment Trees.

Problem J. Chaves e cadeados

Tópicos: árvores, programação dinâmica, bitmasks, menor ancestral comum

Autor: Naim Shaikhzadeh

Problema:

Dada uma árvore com pesos nas arestas, $k \leq 16$ cadeados e suas respectivas chaves espalhados nos vértices, ache o custo do menor caminho de 1 para n que para todo vértice com cadeado sempre visita a chave antes.

Solução:

Primeiro, notamos que como temos uma árvore só existe um caminho de um vértice a outro (sem redundâncias). Vamos resolver o problema usando programação dinâmica com bitmasks. Para uma introdução, você pode ler esse artigo da USACO: `https://usaco.guide/gold/dp-bitmasks?lang=cpp`.

Vamos criar uma $dp(i, mask)$ = menor custo sendo que estou no vértice i e tenho as chaves da bitmask $mask$ ativas. As transições são duas:

- Pegar uma aresta: $dp(v, mask) = min(dp(to, mask) + w)$, para toda aresta de *to* para v com custo w. Isso só vale se v não tem cadeado ou se sua chave está contida em $mask$.

- Pegar um chave (se existe uma em v): $dp(v, mask + b_v) = dp(v, mask)$, com b_v sendo o bit da chave de v na bitmask. Podemos usar por exemplo: $b_{c_i} = 2^{i-1}$.

Fazendo essa programação dinâmica temos $N \cdot 2^k$ estamos e a transição depende apenas do número de arestas. De qualquer jeito, isso é demais.

Para otimizar, basta notar que se v não tiver chave, não for o início (1) nem o final (N), podemos tentar omitir ele, pois esse é apenas um ponto de transição, em que usaremos apenas as transições de pegar arestas.

Sendo assim, podemos com uma DFS obter todas os cadeados que estão no caminho de 1 a V. Chamamos esse conjunto de $need_V$. Note agora que só podemos ir da chave em c_i para a chave em c_j se $need_{c_j}$ estiver contida na $mask$ atual.

Podemos então fazer uma nova $dp(v, mask)$, mas com $v \in \{c_1, c_2, c_3..., c_k, 1, N\}$. Só temos que mudar a transição de pegar uma aresta para na verdade pular de um estado a qualquer outro, caso $need_v$ esteja em $mask$: $dp(to, mask) = min(dp(v, mask) + dist_{v,to})$, satisfazendo que toda chave de $need_{to}$ está em $mask$. $dist_{v,to}$ é a distância entre os nós e pode ser calculado via LCA em toda transição ou precomputado com uma DFS em $O(NK)$.

Complexidade final: $O(K^2 2^K + NK)$, para o caso de precomputar as distâncias.

Solução alternativa: Virtual Tree

Uma outra solução seria rodar o algoritmo de Dijkstra no grafo, com os vértices sendo o estado da dp (v,$mask$). Desse modo, temos $O(N2^k)$ vértices e arestas e o Dijkstra roda em $O(N2^k log(N2^k))$.

Para melhorar a complexidade, notamos que os únicos vértices que importam no Dijkstra são aqueles com início, final, chaves ou cadeados. Podemos computar a Virtual Tree disso e rodar o Dijkstra, agora com apenas $O(k2^k)$ vértices.

Para aprender sobre virtual tree é fortemente recomendado este vídeo: https://codeforces.com/blog/entry/76955