

Maratona de Programação da SBC 2016

Caderno de soluções

A - Andando no tempo

Para voltar para o exato ponto de saída, é necessário que ocorra alguma das alternativas: dois créditos são iguais ou um crédito é igual à soma dos outros dois. Com apenas alguns comandos if a solução fica com complexidade $O(1)$.

B - Batata quente

Observação 1: precisamos apenas da posição mais à frente da fila que a batata quente pode chegar caso a batata seja entregue a cada criança da fila (chamemos esse valor de M_i) - esse valor pode ser calculado com uma busca em profundidade.

Observação 2: Ao incrementar X , ou mantemos igual ou aumentamos as chances de vitória do professor. Isso nos permite utilizar busca binária para descobrir a posição ótima de X .

Observação 3: Podemos utilizar uma estrutura de range query mantendo apenas os M_i do intervalo atual. Isso nos permite responder, para um determinado X , quantas crianças implicariam num jogo ganho ou perdido para o professor dentro do intervalo.

Observação 4: Podemos responder às perguntas de modo off-line. Isso nos permite ordená-las de uma maneira conveniente: separando os inícios em intervalos de tamanho \sqrt{N} e mantendo os finais iguais. Isso nos permite diminuir o número de movimentos nos extremos dos intervalos da pesquisa para $O(Q\sqrt{N} + N\sqrt{N})$.

C - Containers

No pior caso, com oito valores distintos, há $8! = 40320$ configurações distintas. A partir de cada configuração há 10 movimentos possíveis. Com isso, o problema pode ser modelado diretamente como caminho mínimo em grafos. Cada configuração é um vértice e cada

movimento é uma aresta. Uma implementação razoável de Dijkstra usando map (ordenado) para indexar os vértices já é suficiente. É possível obter tempos bem mais rápidos com uma boa função de hash para indexar os vértices.

Opcionalmente, note que é possível computar com certa facilidade uma cota inferior relevante para o custo mínimo para ir de uma configuração C qualquer para uma outra configuração D qualquer. Por exemplo, se o valor 50 aparece apenas uma vez e a distância de Manhattan entre a posição dele em C e a posição dele em D é 3, então o caminho mínimo para ir de C para D é maior ou igual a 150. Aplicando algo assim para todos os valores da configuração podemos calcular uma boa cota inferior entre uma configuração C qualquer e a configuração final F, $h(C,F)$, o que vai ser uma heurística admissível para o algoritmo A* (uma generalização de Dijkstra), que nesse caso vai sempre computar corretamente o caminho mínimo e será ainda mais rápida do que Dijkstra com hash. A complexidade esperada é a mesma do Dijkstra.

D - Divisores

Como C é um múltiplo de N, basta procurar entre os divisores de C o menor número que satisfaz todas as restrições. Para fazer isso você testa apenas até a raiz de N, de forma que a solução tem complexidade $O(\sqrt{C})$.

E - Estatística hexa

Um $\text{total}(S,p)$ é composto por 16 somas que dependem da permutação p. Note porém, que o valor de cada soma, por exemplo, $S[4,9,5,a]$, depende apenas de quais dígitos já foram removidos, não da ordem em que foram removidos. Assim, $S[4,9,5,a] = S[5,a,9,4]$ por exemplo. Daí que, apesar de haver $16!$ permutações (um número não razoável), podemos ver que existem apenas 2^{16} (um número razoável) somas distintas, uma para cada possível subconjunto de dígitos removidos. Então, os $16!$ possíveis totais, sobre os quais queremos computar as respostas, são compostos de 16 somas entre 2^{16} possíveis somas.

O primeiro passo é pré-computar todas as 2^{16} possíveis somas, extraíndo os dígitos de cada número da sequência da entrada e acumulando. Digamos que $\text{soma}[k]$, para $0 \leq k \leq 2^{16}$, contenha agora as possíveis somas. k é uma máscara de bits que representa o conjunto de dígitos presentes na soma. Se o i-ésimo bit de k é 1, então o i-ésimo dígito de $[0,1,2,3,4,5,6,7,8,9,a,b,c,e,d,f]$ está presente na soma[k].

Os totais mínimo e máximo podem agora ser computados com programação dinâmica ou, de forma equivalente, por caminho mínimo e máximo em DAG. Vamos adotar a interpretação de grafos. Cada subconjunto de dígitos hexadecimais é um vértice. Existe uma aresta direcionada do vértice A para o vértice B, se B é exatamente igual a A menos um dígito. Por

exemplo, há uma aresta de $\{3,5,8,a,f\}$ para cada um dos cinco vértices: $\{5,8,a,f\}$, $\{3,8,a,f\}$, $\{3,5,a,f\}$, $\{3,5,8,f\}$ e $\{3,5,8,a\}$. O peso da aresta (A,B) é soma[B]. O total mínimo (máximo) é o caminho mínimo (máximo) entre $I=\{0,1,2,3,4,5,6,7,8,9,a,b,c,e,d,f\}$ e $F=\{f\}$.

Veja que cada uma das $16!$ permutações define um caminho distinto entre I e F. Para o somatório de todos os totais, basta aplicar alguma combinatória para sacar quantos caminhos passam por cada vértice e fazer o somatório.

Sendo B a base (16 neste caso) e N o tamanho da sequência, a solução esperada é $O(N \cdot 2^B)$.

F- Fundindo Árvores

Devido às estruturas das árvores destras e canhotas, os únicos vértices que podem ser superpostos são os vértices centrais. Vamos denominar Sequência de Descendentes Centrais (SDC) uma sequência de vértices adjacentes tal que cada vértice, menos o primeiro, é o filho central do vértice anterior na sequência.

Na árvore superposta, os vértices superpostos pertencem a uma SDC da árvore canhota e a uma SDC da árvore destra. Considere

$d1$ = comprimento da SDC que inicia na raiz da árvore destra

$e1$ = comprimento da SDC que inicia na raiz da árvore canhota

d = comprimento da mais longa SDC que inicia em qualquer vértice central da árvore destra

e = comprimento da mais longa SDC que inicia em qualquer vértice central da árvore canhota

A árvore superposta tem como raiz ou a raiz da árvore destra ou a raiz da árvore canhota, e portanto uma das SDCs deve iniciar na raiz de uma das árvores, destra ou canhota. O número máximo de vértices que podem ser superpostos é portanto

$$\text{vert_sup} = \max(\min(e1, d), \min(d1, e))$$

A solução para este problema envolve portanto percorrer as árvores para determinar os valores de d , e , $d1$ e $e1$. O número de vértices da árvore superposta com menos vértices é a soma dos vértices das árvores destra e canhota menos o valor de vert_sup . No fim a implementação deve ficar com complexidade $O(N)$.

G - Go--

Há várias formas de resolver o problema de maneira mais eficiente do que a força-bruta. Vamos descrever a ideia de duas soluções por programação dinâmica, uma $O(N^3)$, que já é suficiente, e outra $O(N^2)$.

A primeira ideia é contar diretamente cada sub-área, gastando um tempo constante para cada uma independente da dimensão. Seja $m[i][j][k]$ igual a: 0 se a sub-área de dimensão k , cujo canto superior esquerdo é a célula (i,j) , não contém nenhuma pedra; 1 se contém ao menos uma pedra preta e nenhuma branca; 2 se contém ao menos uma pedra branca e nenhuma preta; e 3 se contém pedras pretas e brancas. Podemos obter $m[i][j][k]$, em tempo constante, a partir de $m[i][j][k-1]$, $m[i][j+1][k-1]$, $m[i+1][j+1][k-1]$ e $m[i+1][j][k-1]$. Como ocorre com frequência em soluções assim, note que a terceira dimensão k não precisa ser alocada. Basta uma matriz N por N .

A segunda ideia é notar que, de fato, é possível contar, em tempo constante, quantas das sub-áreas cujo canto superior esquerdo é (i,j) são de cada jogador. Considere a sequência de sub-áreas, cujo canto superior esquerdo é (i,j) , de dimensões 1, 2, 3, Se há uma pedra preta na sub-área de dimensão k , todas as sub-áreas da sequência de dimensões maiores do que k também terão. Essa é a observação chave. Defina $\text{distP}(i,j)$ como o maior inteiro k tal que a sub-área com dimensão k , dessa sequência, não contém nenhuma pedra preta. De forma análoga, $\text{distB}(i,j)$ é o maior inteiro k tal que a sub-área com dimensão k , dessa sequência, não contém nenhuma pedra branca. Podemos interpretar esses valores como a distância da célula (i,j) até a primeira pedra preta (ou branca) quando seguimos a sequência referida. Fica como exercício ver que $\text{distP}(i,j)$ pode ser obtido, em tempo constante, a partir de $\text{distP}(i,j+1)$, $\text{distP}(i+1,j+1)$ e $\text{distP}(i+1,j)$; e que o número de sub-áreas, cujo canto superior esquerdo é (i,j) , pertencentes ao jogador que joga com as pedras pretas é $\max(0, \text{distB}(i,j) - \text{distP}(i,j))$. De forma simétrica, para as brancas vale $\max(0, \text{distP}(i,j) - \text{distB}(i,j))$.

H - huaauhahhuahau

A solução mais eficiente é guardar um ponteiro para o começo e para o fim da string e testar se as letras são iguais caso sejam válidas (ou seja, comparando apenas as vogais). Mas outras soluções menos eficientes também são possíveis, como por exemplo copiar a string para uma outra, eliminando as consoantes, e verificar se a nova string tem a propriedade desejada (palíndromo). A solução tem complexidade linear, porém os limites são baixos e qualquer solução com complexidade melhor que exponencial provavelmente passará.

I - Isósceles

Este problema pode ser resolvido em tempo $O(N)$, com duas passadas pelo vetor de alturas. A ideia é obter, independentemente, o maior lado esquerdo e o maior lado direito de um triângulo cujo centro está em cada posição i . Seja $h(i)$ a altura do muro na posição i . Suponha

que $esq(i)$ contenha a maior altura, na posição i , a partir da qual é possível descer na diagonal para a esquerda até a altura 1. Dá para ver que $esq(i+1)$ é igual a $\min(esq(i)+1, h(i+1))$. Vale $esq(1) = 1$. De maneira análoga, computamos $dir(i)$ para todas as posições: $esq(i)$ é computado com uma passada da esquerda para a direita no vetor, e $dir(i)$ com uma passada da direita para a esquerda. O maior triângulo isósceles, do tipo que eles querem, na posição i , é $\min(esq(i), dir(i))$, e a resposta final, claro, é a maximização desse valor para todas as posições.

J - Jogos olímpicos

A primeira coisa a se perceber é que para um atleta contar na resposta ele tem que ser o mais habilidoso e o menos fatigado. Vamos supor que esses dois parâmetros sejam nomeados H e F , e que consideremos que F seja a fadiga inversa (multiplicar os números na entrada por -1 , com isso podemos aplicar a mesma lógica para H e F). O atleta X é de ouro se em um determinado período $P \geq 0$ ele seja o com melhor H e F . Podemos computar os períodos em que cada atleta é o melhor em H e F independentemente, e ver se os períodos coincidem.

Agora consideremos apenas H . Um detalhe importante é que o período ou vai ser vazio ou vai ser apenas um período contínuo, pois ele melhora/piora linearmente, ou seja, se ele ficar pior que alguém ele nunca mais vai passar a ser melhor, e vice-versa. Ordenemos os atletas por melhor H . Com o passar do tempo os atletas podem mudar de posição nesta ordenação. Quando um atleta A fica melhor que um atleta B podemos descartar B , pois ele nunca mais será melhor que A novamente, e se o atleta C está na frente de B , passamos a considerar o tempo em que A passa C (se isso acontecer em algum momento). Só precisamos registrar quem está em primeiro, e por quanto tempo o atleta fica em primeiro. Calculamos os tempos em que os atletas mudarão de posição em relação ao atleta da frente, e processamos estes tempos em ordem.

Podemos fazer isso com uma árvore balanceada (set em C++ por exemplo). No pior caso, se temos N atletas teremos N trocas de posição, e nunca teremos mais do que N tempos para processar. A complexidade fica $O(N \log N)$. Depois de calcular os intervalos de cada atleta passamos em $O(N)$ por todos atletas descobrindo para cada atleta se eles ficaram em H e F em primeiro em algum período, e se seus períodos em H e F interceptam.

K - Kit de encolhimento de polígonos

A solução usa programação dinâmica. Se guardarmos como estado qual a operação escolhida dos últimos dois vértices, conseguimos testar para o vértice atual se aplicando as duas operações o polígono continua convexo. Quando escolhemos a operação temos um triângulo considerando o primeiro ponto e o ponto anterior, que podemos somar na área total.

A complexidade esperada fica $O(N \cdot K)$, onde K é uma constante considerando as orientações dos últimos pontos, ordem do polígono e as operações dos primeiros dois pontos (para testar se o polígono fechou de forma convexa).

L - Ladrilhos

Se preenchermos os buracos com uma cor igual a alguma das cores que já aparecem no mosaico as áreas existentes só podem aumentar de tamanho, sendo assim a resposta ótima sempre envolve pintar os buracos com uma cor que ainda não foi vista. Basta então considerar zero como sendo uma cor e usando uma busca em largura ou profundidade determinar o tamanho da menor área existente. Complexidade fica $O(\text{linhas} \cdot \text{colunas})$.