



Maratona PPCI de Programação SAET 2026

Caderno de Soluções

2026

Problema A. A Vaca e o Camaleão

Tempo limite: 1000 ms
Memória limite: 256 MiB
Autor: Ricardo Oliveira

Solução

Seja a um inteiro positivo menor ou igual a B . Note que:

- se $f(a, B) = f(A, B)$, então $A = a$;
- se $f(a, B) < f(A, B)$, então $A < a$;
- se $f(a, B) > f(A, B)$, então $A > a$.

Esta observação nos permite resolver o problema fazendo *busca binária* em a . Assim, o problema se torna calcular o valor de $f(a, B)$ para algum $a \leq B$ dado.

Seja $F(N)$ o número total de *bits* 1 em todos os inteiros menores ou iguais a N . Note que

- $f(a, B) = F(B) - F(a - 1)$

Assim, o problema agora se reduz a calcular $F(N)$ para algum inteiro N dado.

$F(N)$ pode ser computado com uma técnica de Programação Dinâmica conhecida como *digit dp*.

Seja n o número de *bits* de N , e seja $N_{n-1}N_{n-2}\dots N_1N_0$ a representação binária de N , do *bit* mais para o menos significativo.

Considere que você está escrevendo, em binário, os números menores ou iguais a N , do *bit* mais significativo ao menos significativo. Considere o estado $dp[b][l]$, onde b indica a posição do *bit* que você está escrevendo agora, e $l = 1$ (de “limitado”) se todos os *bits* que você já escreveu até o momento são exatamente iguais a $N_{n-1}N_{n-2}\dots N_{b+1}$, ou $l = 0$ caso contrário (a ideia aqui é que, se $l = 1$, você ainda precisa “tomar cuidado” para não escrever um número maior que N ; se $l = 0$, você já pode escrever qualquer número, pois ele certamente será menor ou igual a N . O ignorar do último caso da *dp*, dado abaixo, garante esta propriedade). O estado $dp[b][l]$ retorna o número de *bits* 1 que serão escritos ao todo sob estas condições, de forma que $F(N) = dp[n - 1][1]$.

O valor de $dp[b][l]$ é:

- considerando escrever o *bit* 0 na posição b :
→ $dp[b - 1][l']$, onde $l' = 1$ se e somente se $l = 1$ e $N_b = 0$;
- considerando escrever o *bit* 1 na posição b :
→ se $l = 0$, o *bit* 1 aparece, na posição b , em todos os 2^b números possíveis de serem escritos a partir desse estado; logo, somar $2^b + dp[b - 1][0]$;
→ se $l = 1$ e $N_b = 1$, o *bit* 1 aparece, na posição b , em todos os $N_{b-1}N_{b-2}\dots N_0 + 1$ números possíveis de serem escritos a partir desse estado; logo, somar $N_{b-1}N_{b-2}\dots N_0 + 1 + dp[b - 1][1]$;
→ se $l = 1$ e $N_b = 0$, não é possível escrever o *bit* 1 na posição b . Ignore este caso.

Uma possível referência para *digit dp* é <https://www.geeksforgeeks.org/dsa/digit-dp-introduction/>

Complexidade: $O(\lg B)$ (da busca binária) $\times O(\lg B)$ (do *digit dp*) = $O((\lg B)^2)$ por caso de teste.

Problema B. Barril de Bananas

Tempo limite: 1000 ms
Memória limite: 256 MiB
Autor: Henrique Farias

Solução

Verifique se o número de bananas é divisível pelo número de macacos. Caso seja, imprima B/N , caso o contrário, imprima -1 .

Complexidade: $O(1)$.

Problema C. Corredor Infinito

Tempo limite: 1000 ms
Memória limite: 256 MiB
Autor: Ricardo Oliveira

Solução

Mantenha uma *sequência ordenada de intervalos* de acertos seguidos. Por exemplo, para a string `ooxoxooxx`, a sequência de intervalos é $([1 - 2], [4 - 4], [6 - 7])$ (os intervalos de acertos seguidos). Já para a string `ooxooooxx`, a sequência se torna $([1 - 2], [4 - 7])$.

Além da sequência de todos os intervalos de acertos seguidos, mantenha também a *subsequência* ordenada contendo apenas os intervalos *válidos*, que são aqueles cujo tamanho é maior ou igual a K . Para a string `ooxooooxx` e $K = 3$, a subsequência de intervalos válidos é $([4 - 7])$.

Ao trocar um acerto por um erro, encontre na(s) sequência(s) o intervalo a qual pertence, e quebre-o em até outros dois intervalos. Como exemplo, se existe o intervalo $[6 - 8]$ e é trocado o resultado da tentativa 7, este intervalo deixa de existir e passam a existir outros dois: $[6 - 6]$ e $[8 - 8]$. Já se existe o intervalo $[6 - 8]$ e é trocado o resultado da tentativa 8, o intervalo passa a ser $[6 - 7]$; e se já existe o intervalo $[4 - 4]$ e é trocado o resultado da tentativa 4, o intervalo é removido da(s) sequência(s).

Ao trocar um erro por um acerto: dois intervalos podem ser “unidos”; um intervalo pode aumentar; ou um novo intervalo pode ser criado. Como exemplo, se existem os intervalos $[1 - 2]$ e $[4 - 5]$ e é trocado o resultado da tentativa 3, ambos os intervalos são “unidos” e trocados por $[1 - 5]$; se existe apenas o intervalo $[1 - 2]$, ele aumenta para $[1 - 3]$; se nenhum dos intervalos existem, um novo intervalo $[3 - 3]$ é criado.

A cada alteração na sequência de intervalos, lembre de atualizar, se necessário, também a subsequência de intervalos válidos. A resposta é sempre dada por $l + K$, sendo l o início do primeiro intervalo da subsequência de intervalos válidos, ou -1 se a subsequência está vazia.

Utilize estruturas de dados eficientes, como dois *set de pairs* do C++, de forma que cada operação (buscar, inserir e remover) é feita em $O(\lg N)$.

Complexidade total: $O(N \lg N + Q \lg N)$

Problema D. Dados Gêmeos

Tempo limite: 1000 ms
Memória limite: 256 MiB
Autor: Henrique Farias

Solução

Verifique qual valor é maior ou verifique a paridade em caso de empate.

Complexidade: $O(1)$

Problema E. Encanador

Tempo limite: 1000 ms
Memória limite: 256 MiB
Autor: Henrique Farias

Solução

Podemos implementar uma solução iterando um loop por todas letras da string em ordem e contando qual a estatura atual do Mario, se no estado atual ele está sobre efeito de uma estrela, e caso sim, contar também contabilizar a duração do efeito, neste caso durante os obstáculos sobre efeito da estrela ignoramos todos os danos.

Complexidade: $O(N)$ - Onde N é o tamanho da string.

Problema F. Fazendinha

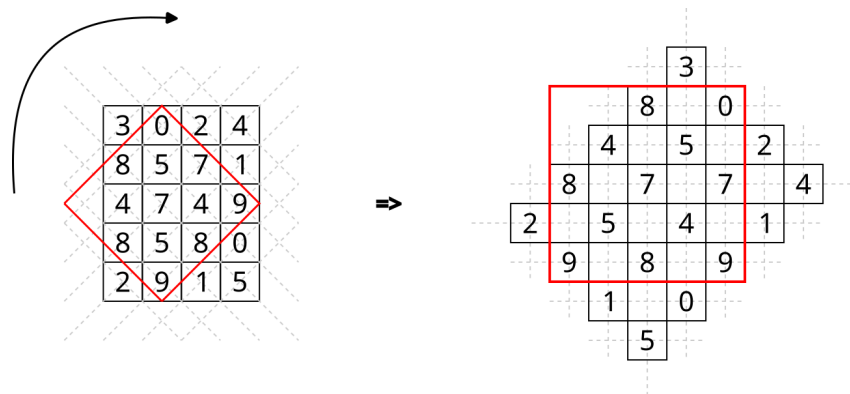
Tempo limite: 1500 ms
 Memória limite: 384 MiB
 Autor: Ricardo Oliveira

Solução

Note que o formato da área coletada é o de um *losango*. Primeiramente, rotacione a matriz de forma a tornar a área coletada um *retângulo*, paralelo aos eixos da matriz.

Esta rotação pode ser feita com a transformação $T : (i, j) \mapsto (i + j, j - i)$.

A figura abaixo exemplifica a matriz original e nova matriz rotacionada:



(Obs: para a implementação, as dimensões da matriz nova são dobradas; os “buracos” da matriz nova são preenchidos com zero; um deslocamento fixo (*offset*) é usado nos índices das colunas para não acessar colunas de índices negativos.)

Com a nova matriz, pré-compute sua *prefix-sum*, de forma a tornar possível obter a soma de qualquer submatriz (em uma região retangular) em $O(1)$. Você pode ler sobre *prefix-sum* em matrizes em <https://www.geeksforgeeks.org/dsa/prefix-sum-2d-array/>.

Para cada $1 \leq i \leq N$ e $1 \leq j \leq M$, determine o menor valor de D necessário para coletar ao menos T batatas se a colheitadeira for acionada em (i, j) . A resposta é o menor desses valores.

Dada uma posição (i, j) da matriz original, use a *busca binária* para determinar o menor valor de D necessário a partir dela: dada uma distância D , calcule a soma dos elementos no losango de vértices opostos $(i, j - D)$ e $(i, j + D)$ que na nova matriz transformada é o retângulo de vértices opostos $(i + j - D, j - D - i)$ e $(i + j + D, j + D - i)$. Esta soma pode ser obtida em $O(1)$ com a *prefix-sum* pré-calculada. Se esta soma for maior ou igual a T , diminua o valor de D ; caso contrário, aumente o valor de D (nos termos da busca binária).

Complexidade Total: $O(N \times M \times \lg(N + M))$

Problema G. Grande Dúvida

Tempo limite: 1000 ms
Memória limite: 256 MiB
Autor: Ricardo Oliveira

Solução

Percorra todas as posições da palavra P , testando se cada letra $P[i]$ é igual a C , e parando assim que a primeira posição de C for encontrada. Se a letra não foi encontrada após o final do laço, imprima -1 .

Complexidade: $O(|P|)$

Problema H. Horário

Tempo limite: 1000 ms
Memória limite: 256 MiB
Autor: Henrique Farais

Solução

Primeiramente observamos que para X movimentos a partir da hora 12h só vamos conseguir terminar em um horário com a mesma paridade de X .

Por exemplo para $X = 1$ podemos mover para 11h ou 1h, ambas horas ímpar, para $X = 2$ podemos mover das 11h para 10h ou 12h, e das 1h para 12h ou 2h, ambas pares.

Verificando se a paridade permite chegar na hora H verificamos em sequência se o valor de movimentos é maior ou igual a menor distância das 12h para H .

Complexidade: $O(1)$

Problema I. Invasores Espaciais

Tempo limite: 1000 ms
Memória limite: 256 MiB
Autor: Henrique Farias

Solução

Para resolver o problema, podemos inicialmente transformar o círculo de alienígenas em uma sequência linear duplicando seus elementos, permitindo considerar também as sequências que atravessam a fronteira entre o último e o primeiro alienígena. Em seguida, utilizamos a técnica de *Two Pointers* para manter uma sequência contínua e verificar quantos tipos distintos de escudo estão presentes nela. Sempre que a quantidade de tipos distintos ultrapassar K , diminuimos o tamanho da sequência pelo início até que ela volte a ser válida. Dessa forma, conseguimos encontrar a maior sequência contínua que possui no máximo K tipos distintos de escudo. Como uma sequência válida não pode possuir mais de N alienígenas, limitamos a resposta final a N .

Referência *Two Pointers*: <https://usaco.guide/silver/two-pointers?lang=cpp>

Complexidade: $O(N)$

Problema J. Jornada pelo Reino

Tempo limite: 1000 ms
Memória limite: 256 MiB
Autor: Ricardo Oliveira

Solução

Primeiramente, vamos descobrir a equação da parábola que representa a trajetória. Ela é dada uma função de segundo grau, na forma $f(x) = Ax^2 + Bx + C$. Precisamos descobrir os valores dos coeficientes A , B e C .

Tem-se que $f(0) = h_0$; logo, $C = h_0$. Para os outros coeficientes, temos o seguinte sistema de equações lineares:

$$\begin{cases} A(x_1)^2 + Bx_1 = h_1 - C \\ A(x_2)^2 + Bx_2 = h_2 - C \end{cases}$$

ou, na forma matricial,

$$\begin{bmatrix} (x_1)^2 & x_1 \\ (x_2)^2 & x_2 \end{bmatrix} \cdot \begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} h_1 - C \\ h_2 - C \end{bmatrix}.$$

Há diversos algoritmos para se resolver sistemas de equações lineares. Para o caso geral, veja https://cp-algorithms.com/linear_algebra/linear-system-gauss.html para um algoritmo cúbico. Mas, por se tratar de um sistema pequeno 2×2 , você também pode resolvê-lo por substituição direta em $O(1)$.

Com a função $f(x) = Ax^2 + Bx + C$ definida, podemos calcular a área utilizando Cálculo Integral. A área é dada por

$$\int_{d_1}^{d_2} Ax^2 + Bx + C \, dx = \frac{Ax^3}{3} + \frac{Bx^2}{2} + Cx \Big|_{d_1}^{d_2} = \frac{A(d_2)^3}{3} + \frac{B(d_2)^2}{2} + Cd_2 - \frac{A(d_1)^3}{3} - \frac{B(d_1)^2}{2} - Cd_1.$$

Complexidade: $O(1)$ por caso de teste.

Problema K. Kurt Manipulando Células

Tempo limite: 1000 ms
Memória limite: 256 MiB
Autor: Henrique Farias

Solução

Usamos o Teorema de Zeckendorf: todo inteiro K tem uma única decomposição em Fibonacci não consecutivos, e ela é justamente a decomposição com a menor quantidade de termos possível. Ela é obtida de forma gulosa, subtraindo sempre o maior Fibonacci $\leq K$.

Como $K \leq 10^{18}$, existem apenas 87 Fibonacci possíveis. Para cada K_i , decompomos e unimos utilizando a estrutura de Disjoint-Set Union (DSU) nos termos consecutivos da decomposição, marcando o tamanho do maior conjunto formado. Como os Fibonacci são nós fixos e compartilhados entre todas as decomposições, um valor repetido em dois K_i diferentes já une os dois conjuntos automaticamente.

Ao final, percorremos o conjunto de todos os 87 Fibonacci em ordem crescente e retornamos o primeiro (menor) cujo tamanho de componente é igual a o maior conjunto — isso já resolve o desempate pelo menor valor.

Referência para o Teorema de Zeckendorf: <https://planetmath.org/zeckendorfstheorem> Referência para o DSU: https://cp-algorithms.com/data_structures/disjoint_set_union.html

Complexidade: $O(N)$

Problema L. Lobo Faminto

Tempo limite: 1000 ms
Memória limite: 384 MiB
Autor: Ricardo Oliveira

Solução

Este problema envolve conceitos da Teoria dos Grafos.

Fazer uma busca (em profundidade ou em largura) para cada consulta não será rápido o bastante, dado que cada busca tem custo $O(N + M)$ e o custo total seria, portanto, $O(Q \times (N + M))$.

Entretanto, note que, em um grafo não direcionado, existe um caminho de l para m se e somente se l e m estão na mesma componente conexa.

Assim, pré-processe o grafo com uma única busca (em profundidade ou em largura) para identificar a componente conexa a qual cada vértice pertence. Então, para cada consulta, verifique se l e m estão na mesma componente, ou em componentes distintas.

Referência a busca em profundidade: <https://cp-algorithms.com/graph/depth-first-search.html>

Referência a busca em largura: <https://cp-algorithms.com/graph/breadth-first-search.html>

Como esta verificação pode ser feita em $O(1)$ após o pré-processamento, a complexidade total se torna $O(N + M + Q)$.

Problema M. Material de Consulta

Tempo limite: 1000 ms
 Memória limite: 384 MiB
 Autor: Ricardo Oliveira

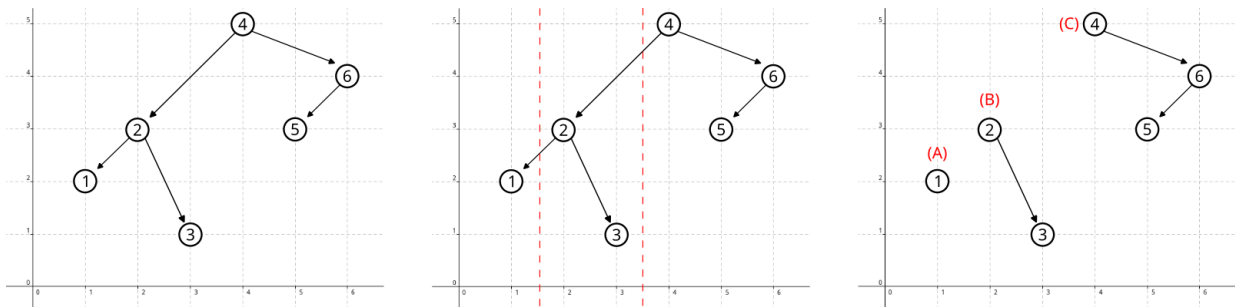
Solução

O problema pode ser resolvido com uma *árvore cartesiana*, também chamada de *treap*. A *treap* é uma árvore binária em que cada nó contém, além de sua chave em si, uma coordenada (x, y) . Em relação aos valores x , a árvore se comporta como uma árvore binária de busca (BST). Em relação aos valores y , a árvore se comporta como uma *heap*.

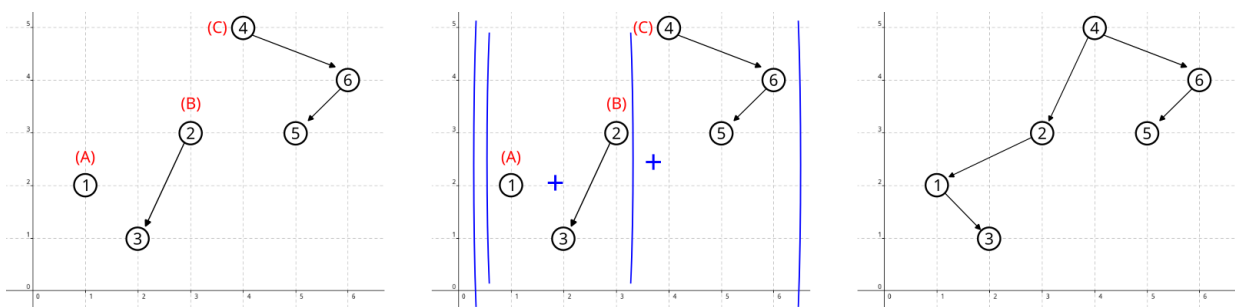
Neste problema, o valor de x será o índice de uma página, enquanto a chave em si é o algoritmo atualmente nesta página. Neste caso, o valor de x não é armazenado no nó, mas sim calculado implicitamente durante as operações na estrutura. Além disso, o valor de y pode ser um inteiro gerado aleatoriamente e uniformemente, o que garante, na prática, a altura logarítmica da árvore (pois a probabilidade de $h \neq O(\lg N)$, para valores grandes e relevantes de N , é virtualmente nula).

Você pode ler mais sobre a *treap* em https://cp-algorithms.com/data_structures/treap.html.

Para cada operação (L_i, R_i) , realize um *split* para $x < L_i$ e outro para $x \leq R_i$. Isto gera três árvores A , B e C , de forma que “isolamos”, na árvore B , os nós cujas posições x estão entre L_i e R_i . A figura abaixo mostra uma *treap*, os *splits* feitos para $L_i = 2$ e $R_i = 3$ e as três árvores resultantes.



Agora, precisamos inverter a ordem das chaves em toda a subárvore B , e fazer duas operações *merge* para reconstruir toda a árvore.



Para inverter a árvore B eficientemente, utilize a técnica de *lazy propagation*: mantenha em cada nó uma *flag* indicando se a subárvore enraizada nele deve ser invertida. Coloque a *flag* na raiz da árvore B . Então, sempre que um nó for visitado durante qualquer operação, verifique sua *flag* e, se necessário, troque os ponteiros para seus filhos e “empurre” a *flag* para ambos.

Por fim, faça um percurso *in-order* na árvore final para imprimir a resposta.

Complexidade total: $O((N + Q) \lg N)$ (com probabilidade virtualmente garantida).