



Maratona Interna - Soluções

Instituto de Computação UNICAMP

16 de agosto de 2025

Responsáveis pelo desenvolvimento da prova:

- Fábio Usberti
- Naim Shaikhzadeh Santos
- Henrique Campos Padula
- Miguel Alessandro Mini Huambachano
- Renan Fernando Franco da Silva
- Víctor Racsó Galván Oyola

Conteúdo

Problema 1: Salvador	2
Problema 2: Astana	3
Problema 3: João Pessoa	4
Problema 4: Luxor	5
Problema 5: Guadalajara	6
Problema 6: Chapecó	7
Problema 7: Dhaka	8
Problema 8: Campo Grande	10
Problema 9: Moscou	11
Problema 10: Gramado	12
Problema 11: Online	13
Problema 12: Campina Grande	14

Solução do problema: “Salvador”

Autor(es): Naim Santos

Editorial por: Racsó Galván

Para resolver este problema, basta validar se $2 \cdot (v_i + N) > \sum v_i + N$ para cada v_i .

A complexidade será de $O(1)$.

Solução do problema: “Astana”

Autor(es): Naim Santos

Editorial por: Racsó Galván

Para resolver este problema, basta manipular uma sequência maximal de posições consecutivas com valores consecutivos. Isso pode ser feito em $O(N)$ enquanto se lêem os valores ou em $O(d_N)$ marcando os dias e depois iterando sobre os dias entre 1 e d_N .

Solução do problema: “João Pessoa”

Autor(es): Naim Santos

Editorial por: Racsó Galván

Para resolver este problema, podem-se usar duas abordagens:

Abordagem 1

Se considerarmos o balanço dos prefixos dos parênteses abertos e fechados, é natural pensar que a primeira posição em que esse balanço atinge o seu mínimo deveria ser o ponto de referência para o corte; pois, se for 0, então a string original será uma expressão corretamente balanceada, e se for menor que 0, então haverá alguns parênteses fechados que ainda precisam ser abertos, de modo que estes naturalmente terão que se mover para o lado direito no shift cíclico.

A complexidade dessa abordagem será $O(N)$.

Abordagem 2

Se considerarmos o problema geral de balanceamento de parênteses, é bem sabido que se podem manipular pares (x, y) onde a expressão atual tem x parênteses abertos pendentes de fechar e y parênteses fechados pendentes de abrir. Dessa forma, define-se a operação de concatenação como

$$(x, y) + (a, b) = (x + a - \min\{x, b\}, y + b - \min\{x, b\})$$

Onde subtraímos $\min\{x, b\}$ porque essa é a quantidade de parênteses que acabam se emparelhando.

Se calculamos as concatenações dos prefixos e sufixos da string, podemos encontrar um ponto de corte válido em $O(N)$.

Solução do problema: “Luxor”

Autor(es): Naim Santos

Editorial por: Racsó Galván

Para resolver este problema, é simples notar que, se a soma de todos os valores não estiver dentro do intervalo válido, a resposta é que é impossível.

Agora, se essa soma estiver no intervalo válido, isso significa que podemos garantir que não sairemos do intervalo se

- Quando a soma atual for negativa, somarmos um número não negativo (se existir)
- Quando a soma atual for não negativa, somarmos um número negativo (se existir)

No final, somamos o restante dos valores pendentes, o que garante que não sairemos do intervalo válido (já que todos têm o mesmo sinal e, se não fosse assim, a soma total estaria fora do intervalo válido).

A complexidade será $O(N)$.

Solução do problema: “Guadalajara”

Autor(es): Naim Santos

Editorial por: Racsó Galván

Para resolver este problema, podemos considerar que os H são 1 e os T são 0, de forma que, se considerarmos as strings invertidas, teremos números binários.

Segundo o procedimento proposto, sempre se escolhe algum H para trocá-lo por T e, em seguida, pode-se atribuir qualquer valor às posições à esquerda. O equivalente disso na representação binária é ir de um número x para outro número $y < x$.

O que foi dito acima nos dá a ideia de que sempre podemos imprimir todos os números binários desde a string original até o 0, devendo apenas manter o comprimento das strings e atribuir adequadamente os H e T.

A complexidade será $O(|s| \cdot 2^{|s|})$, onde s é a string da entrada.

Solução do problema: “Chapecó”

Autor(es): Henrique Padula
Editorial por: Racsó Galván

Para resolver este problema, devemos fazer as seguintes observações:

Observação 6.1. *A posição de corte pode ser delimitada de acordo com os seguintes cenários:*

- *Um par de posições $i < j$ com $V_i = V_j$ não estabelece nenhuma restrição sobre a posição de corte.*
- *Um par de posições $i < j$ com $V_i > V_j$ estabelece uma restrição sobre a posição de corte como $P_i \leq C$.*
- *Um par de posições $i < j$ com $V_i < V_j$ estabelece uma restrição sobre a posição de corte como $C \leq P_j - 2$.*

Como a função do problema tem natureza decrescente e depois crescente, podemos ignorar o fato de que $i < j$ e simplesmente usar $j = i + 1$, pois isso é suficiente para calcular o intervalo correto.

É possível manter a informação dos pontos com um `std::set` para que, ao inserir (p, v) , seja possível buscar o máximo j tal que $P_j < p$ e o mínimo k tal que $p < P_k$ em tempo $O(\log M)$.

A complexidade será $O(M \log M)$.

Solução do problema: “Dhaka”

Autor(es): Naim Santos
Editorial por: Racsó Galván

Para resolver este problema, podem-se usar duas abordagens:

Abordagem 1 - Ordenação Topológica

Podemos modelar um grafo direcionado no qual será estabelecida uma aresta entre cada par de nós u e v tais que o valor associado a u deve ser menor que o de v .

O grafo inicialmente terá N nós (que representam cada tuk-tuk) com arestas entre os nós $(i + 1, i)$. Para adicionar uma mudança de posição do tuk-tuk P_i para a posição X_i , realizaremos os seguintes passos:

- Criar um novo nó, chamemos de v .
- Identificar o nó que representa P_i , chamemos de u . Adicionar a aresta (u, v) .
- Se existir, identificar o nó que representa o tuk-tuk que está uma posição à frente de P_i , chamemos de k . Adicionar a aresta (v, k) .
- Se existir, identificar o nó que representa o tuk-tuk que está uma posição atrás de P_i , chamemos de w . Adicionar a aresta (w, v) .

Podemos gerar uma ordenação topológica do grafo e atribuir ao i -ésimo nó o valor i para saber o histórico de valores de cada tuk-tuk ao longo de todo o processo. Depois, é possível reconstruir os valores iniciais e as mudanças de valor a partir do histórico em tempo $O(N + Q)$.

A complexidade será $O(NQ)$ com $O(N + Q)$ de memória.

Abordagem 2 - Compressão e Fenwick/Segment Tree

É possível visualizar a sequência de posições como indo de N a 1 da esquerda para a direita e ir estendendo-a inserindo os tuk-tuks na nova posição em que estarão. Após criar toda a nova sequência com no máximo $N + Q$ elementos, podemos atribuir-lhes a pontuação correspondente de acordo com sua posição na sequência.

Por exemplo, para o primeiro caso de teste, temos a seguinte simulação do processo (os elementos destacados são os “ativos” no momento):

$$L_0 = \{\mathbf{3}, \mathbf{2}, \mathbf{1}\}$$

Após a primeira modificação, o tuk-tuk 2 deve ir como primeiro, então inserimos um 2 à direita do 1:

$$L_1 = \{\mathbf{3}, \cancel{2}, \mathbf{1}, \mathbf{2}\}$$

Após a segunda modificação, o tuk-tuk 3 deve ir como segundo, então inserimos um 3 à direita do 1:

$$L_2 = \{\cancel{3}, \cancel{2}, \mathbf{1}, \mathbf{3}, \mathbf{2}\}$$

Assim, os valores podem ser deduzidos a partir de L_2 :

Tuk-tuk	1	2	3
Valores iniciais	3	2	1
Modificação 1	3	5	1
Modificação 2	3	5	4

Tabela 1: Descrição dos valores em cada consulta

No entanto, essa ideia sozinha leva $O(Q^2)$, o que causaria TLE.

É possível otimizar essa ideia observando o seguinte:

Observação 7.1. *As sequências L_i podem ser expressas como sequências L'_i de tamanho $2n+1$ da forma:*

$$L'_i = \{z_1, a_1, z_2, a_2, \dots, z_{n-1}, a_n, z_n\}$$

Onde z_j são posições dummy que representam z_j posições inativas em L_i e a_j são as posições ativas dos tuk-tuks de L_i .

O que foi dito acima nos permite analisar os L_i em suas versões comprimidas e, a partir delas, podemos reconstruir a lista original usando Fenwick/Segment Tree para encontrar o k -ésimo 1 de uma sequência de tamanho no máximo $N + Q$.

A complexidade dessa solução será $O(NQ + (N + Q) \log(N + Q))$ e a memória requerida é $O(N + Q)$.

Solução do problema: “Campo Grande”

Autor(es): Naim Santos
Editorial por: Racsó Galván

Para resolver este problema, devemos notar que sempre faremos perguntas binárias, o que significa que, para encontrar um peixe, dividiremos um conjunto de tamanho $n > 1$ em dois conjuntos de tamanho x e $n - x$.

A escolha de x não é arbitrária: para minimizar a quantidade esperada de perguntas, devemos balancear o tamanho dos dois subconjuntos. Intuitivamente, se um lado for muito maior que o outro, a profundidade média das folhas da árvore de decisões aumentará. De fato, é possível mostrar formalmente que a função de custo associada é convexa discreta, de modo que a soma $f(x) + f(n - x)$ é minimizada quando x está o mais próximo possível de $\frac{n}{2}$.

Outra maneira de ver isso é notar que o problema é equivalente a construir uma árvore binária ótima com n folhas de peso igual, exatamente o caso tratado pelo algoritmo de Huffman. Nesse cenário, o ótimo é que todas as folhas tenham profundidade $\lfloor \log_2 n \rfloor$ ou $\lceil \log_2 n \rceil$, o que só acontece se a raiz divide o conjunto de maneira quase balanceada. Assim, a escolha ótima é

$$x \in \left\{ \left\lfloor \frac{n}{2} \right\rfloor, \left\lceil \frac{n}{2} \right\rceil \right\},$$

e em particular $x = \frac{n}{2}$ quando n é par.

Dessa forma, podemos calcular a quantidade esperada de perguntas com uma função $f(n)$ definida da seguinte forma:

$$f(n) = \begin{cases} 1 & n = 1, \\ 1 + f\left(\frac{n}{2}\right) & n = 2k, \\ 1 + \frac{n-1}{2n} f\left(\frac{n-1}{2}\right) + \frac{n+1}{2n} f\left(\frac{n+1}{2}\right) & n = 2k + 1. \end{cases}$$

Onde $x = \left\lfloor \frac{n}{2} \right\rfloor$.

Teorema 8.1. *O número de estados distintos visitados ao calcular $f(n)$ é $O(\log n)$.*

Demonstração. Considere o conjunto $S(n)$ dos tamanhos de subproblemas que aparecem ao avaliar $f(n)$ quando sempre dividimos por $x \in \{\lfloor n/2 \rfloor, \lceil n/2 \rceil\}$. Defina, para $k \geq 0$,

$$L_k := \left\lfloor \frac{n}{2^k} \right\rfloor, \quad U_k := \left\lceil \frac{n}{2^k} \right\rceil.$$

Afirmamos que

$$S(n) \subseteq \{L_k, U_k : 0 \leq k \leq \lfloor \log_2 n \rfloor\}.$$

De fato, por definição $n \in \{L_0, U_0\}$. Agora, os filhos de qualquer estado m são $\lfloor m/2 \rfloor$ e $\lceil m/2 \rceil$. Se $m \in \{L_k, U_k\}$, então

$$\left\lfloor \frac{m}{2} \right\rfloor, \left\lceil \frac{m}{2} \right\rceil \in \left\{ \left\lfloor \frac{n}{2^{k+1}} \right\rfloor, \left\lceil \frac{n}{2^{k+1}} \right\rceil \right\} = \{L_{k+1}, U_{k+1}\}.$$

Logo, por indução sobre a profundidade, todos os estados gerados pertencem ao conjunto indicado. Além disso, para $k > \lfloor \log_2 n \rfloor$, temos $n/2^k < 1$, de modo que $L_k = U_k = 0$ (ou não há estados válidos > 0), e a recursão termina em 1. Portanto,

$$|S(n)| \leq 2(\lfloor \log_2 n \rfloor + 1) = O(\log n).$$

□

A complexidade para calcular essa recorrência é $O(N)$ se computarmos todos os valores até N . Contudo, com memoization partindo de um n fixo e usando a divisão ótima, apenas $O(\log n)$ estados distintos são realmente visitados, o que dá tempo e memória $O(\log n)$ para avaliar $f(n)$.

Solução do problema: “Moscou”

Autor(es): Naim Santos
Editorial por: Racsó Galván

Para resolver este problema, podemos usar busca binária sobre a resposta. Calcularemos quantos subarrays têm uma força menor ou igual a X e validaremos se são menores que K ou maiores ou iguais a K . O objetivo é encontrar o menor X tal que essa contagem seja maior ou igual a K .

Pré-processaremos dois arrays L e R de tamanho N , que armazenarão as seguintes informações:

- L_i é o menor índice $j \leq i$ tal que T_i é o mínimo estrito entre todas as posições $[j, i]$.
- R_i é o maior índice $i \leq j$ tal que T_i é o mínimo (podem haver múltiplos mínimos) entre todas as posições $[i, j]$.

Tanto L quanto R podem ser calculados em $O(N)$ usando uma pilha.

Para contar os subarrays com força menor ou igual a X que tenham como ponto mínimo a posição i , devemos considerar o seguinte:

$$T_i \cdot (r - l + 1)^2 \leq X \rightarrow (r - l + 1) \leq \sqrt{\frac{X}{T_i}}$$

Assim, devemos escolher posições $a \in [L_i, i]$ e $b \in [i, R_i]$ tais que sua distância não exceda $C = \left\lfloor \sqrt{\frac{X}{T_i}} \right\rfloor$.

Suponhamos que iteramos sobre quantos elementos pegamos do lado esquerdo ($l = i - a + 1, r = b - i + 1$), então a contribuição se baseará no fato de que:

$$l + r - 1 \leq C \rightarrow r \leq C - l + 1$$

E, combinando com a característica de que $r \leq R_i - i + 1$, a contribuição se torna

$$\text{Contribuição} = \sum_{l=1}^{i-L_i+1} \min\{C - l + 1, R_i - i + 1\}$$

Considerando as posições em que

$$C - l + 1 \leq R_i - i + 1$$

$$C + 1 - (R_i - i + 1) \leq l$$

Podemos contar quantos l contribuem com $R_i - i + 1$ e quais contribuem com $C - l + 1$, de forma que poderemos calcular o total em $O(N)$ usando algumas fórmulas conhecidas para somas.

A complexidade total será $O(N \log(N^2 \cdot MAX))$, onde $MAX = 10^7$.

Solução do problema: “Gramado”

Autor(es): Naim Santos

Editorial por: Miguel Miní

Sem perda de generalidade, assumamos que $b_i \leq b_{i+1}$ para todo $1 \leq i \leq n - 1$. Seja a função $f(k, i) :=$ melhor resposta ao tomar k elementos dos primeiros i .

Abordagem 1 - Segment Tree persistente

Teorema 10.1. *A função $f(k, i)$ é totalmente monótona.*

Demonstração. Para provar a monotonicidade total devemos verificar que ela se cumpre para duas linhas consecutivas $k, k + 1$ e colunas i, j com $i < j$:

- Se $f(k, i) = f(k, j)$, então $f(k + 1, i) \leq f(k + 1, j)$.

Se a primeira igualdade se cumpre, temos que tomando a mesma resposta não se pode piorar, portanto é trivialmente verdadeiro.

- Se $f(k, i) < f(k, j)$, então $f(k + 1, i) < f(k + 1, j)$.

Seja s qualquer subsequência de $k + 1$ elementos ótima nos primeiros i elementos, e seja v um índice que não pertence a alguma subsequência ótima s' de k elementos que termina em i . Então, trocamos essa sequência pela sequência que produz $f(k, j)$, a qual melhora nossa solução, assim a desigualdade se cumpre.

□

Provado isso, podemos proceder de duas formas: usando *SMAWK*, já que só se precisa da monotonicidade total, ou usando dp com divide and conquer, já que monotonicidade total implica monotonicidade simples. Assim, o problema se resume em encontrar $f(x, y)$ eficientemente.

Podemos resolver com persistência, de modo que para cada posição e o respectivo k , encontremos a soma dos k maiores, o que pode ser feito em $O(\log n)$ por consulta.

Abordagem 2 - convolução ($\max, +$)

Uma abordagem alternativa pode ser obtida usando dividir e conquistar. Dado um intervalo $[l, r]$ queremos encontrar a melhor solução para $0 \leq k \leq r - l + 1$.

- Se $l = r$, o resultado é trivial.
- Caso contrário, suponhamos que temos a resposta para os intervalos $[l, m]$ e $(m, r]$. Então nossa resposta pode estar incluída aí, ou usando elementos de ambos os intervalos. Como todos os b_j são maiores ou iguais no lado direito, nosso máximo b encontra-se nesse lado, assim:

$$c_k = \max_{i+j=k} (x_i + y_j)$$

Onde y_j indica a resposta para j elementos no lado direito — já que tomamos o b ali —, e x_i indica a soma dos i maiores a no lado esquerdo. Notemos que esta última é côncava, então podemos aplicar a convolução ($\max, +$).

A complexidade total será $O(N \log N)$ o $O(N \log^2 N)$.

Solução do problema: “Online”

Autor(es): Naim Santos

Editorial por: Miguel Miní

A ideia principal consiste em poder decidir avançar sem me equivocar, já que não devo avançar quando me devolve a minha posição, a minha posição mais um ou a anterior; em outro caso sempre poderia avançar pelo menos uma posição. Portanto, devo consultar ao menos 3 posições à frente; no pior dos cenários, apenas acertando as 3 seguintes é possível avançar. Se randomizarmos quais serão as 3 seguintes, encontramos nossa resposta a cada 4.5 passos.

No entanto, cada vez que acerto, se randomizo os elementos seguintes, posso aumentar k elementos com probabilidade $\frac{1}{2^k}$, o que, em média, me permite avançar $1 + \sum_{i \geq 1} \frac{1}{2^i} = 2$ passos em cada acerto. Isso me dá 2.25 passos esperados para acertar um elemento.

Outra abordagem seria analisar o que acontece se apenas randomizo a parte final — seria o mesmo? Embora pareça a mesma análise, apenas randomizar me dá 4 passos em expectativa, e diferenciam-se porque a primeira abordagem tem memória das tentativas falhadas, o que reduz a nossa quantidade de consultas.

A complexidade total será $O(NE)$ onde $E = 2.25$ é a quantidade de tentativas esperadas para avançar um passo.

Solução do problema: “Campina Grande”

Autor(es): Naim Santos

Editorial por: Miguel Miní

O problema pode ser traduzido a uma multiavaliação em polinômio, notemos que no ano 0 a resposta é $A_0 = \prod_{i=1}^n c_i!$, no ano 1 é $A_1 = \prod_{i=1}^n (c_i + 1)! = A_0 \prod_{i=1}^n (c_i + 1)$, e em geral no ano j é $A_j = A_0 \prod_{l=1}^j (\prod_{i=1}^n (c_i + l))$, se denotamos

$$f(x) = \prod_{i=1}^n (c_i + x)$$

temos

$$A_k = A_0 \prod_{x=1}^k f(x)$$

Portanto, devemos avaliar em f os pontos $1, 2, \dots, k$. Para obter f podemos multiplicar como se fosse uma segment tree, ou escolhendo os dois vetores com menor comprimento cada vez, o que resulta ótimo segundo o algoritmo de Huffman.

A complexidade total será $O(N \log N \times (\log N + \log K))$.