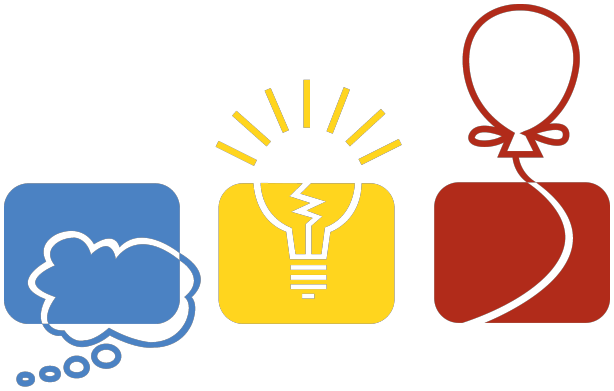


Maratona de Programação SAET 2025

Caderno de Soluções

2025

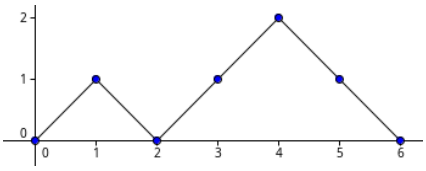
Problema A. Abrindo e Fechando Parênteses

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

Solução

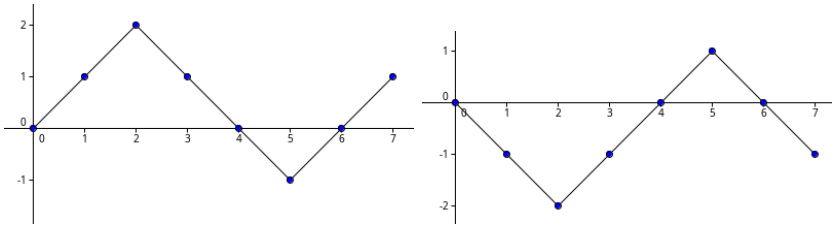
É bastante conhecido o algoritmo de verificação que usa uma pilha como estrutura auxiliar ou, neste caso em que há apenas um tipo de parênteses, um contador com o tamanho da pilha: incremente o contador ao encontrar (e decremente o contador ao encontrar). Para ser bem balanceada, o contador deve terminar em 0 (zero) e nunca pode ter assumido algum valor negativo (menor que zero) durante o algoritmo.

Esta ideia é equivalente a considerar (como +1 e) como -1, e, dado um intervalo, verificar se a soma de todos os seus valores é 0 e também se não existe nenhum prefixo do intervalo cuja soma é negativa. O gráfico abaixo exemplifica a soma dos prefixos para `()(())`, que é bem balanceada; note que termina em 0 e nunca fica abaixo de 0 no gráfico.



Apenas utilizar esse algoritmo a cada operação do tipo 2 faria a solução ter uma complexidade de $O(Q \times N)$, que é muito lento para os limites deste problema. Precisamos de uma estrutura de dados auxiliar que, dado um intervalo $[l..r]$, permita determinar a soma dos valores no intervalo, e também o valor da menor soma de um prefixo do intervalo (ambos devem ser iguais a 0 para a *substring* ser balanceada). Com uma **Árvore de Segmentos**, podemos determinar ambos os valores em $O(\lg N)$, o que é rápido o bastante.

Nos resta saber como atualizar a árvore a cada operação do tipo 1. Vamos notar o que acontece com as somas dos prefixos do intervalo quando ele é invertido; os gráficos abaixo exemplificam a soma dos prefixos para `(()))((` e seu inverso `))(((())`, que não são bem balanceados:



Note que o gráfico apenas se "espelha verticalmente", de forma que: a soma do intervalo tem seu sinal invertido; o novo valor mínimo do intervalo passa a ser o antigo valor máximo do intervalo, com sinal invertido; e o novo valor máximo do intervalo passa a ser o o antigo valor mínimo, também com sinal invertido. Assim, é possível processar cada operação em $O(\lg N)$ com a técnica de **Lazy Propagation**, mantendo em cada nodo da árvore a soma, o menor valor e o maior valor dos prefixos do segmento, e os atualizando conforme as propriedades de "espelhamento" citadas.

Uma referência para a Árvore de Segmentos e Lazy Propagation é https://cp-algorithms.com/data_structures/segment_tree.html.

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

Problema B. BugNote

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

Solução

A ideia da solução deste problema é que para cada nome escrito você deva comparar a string do nome com o set de nomes de alunos, caso o nome seja de um aluno existente, você faz uma segunda comparação com o tamanho do código, caso exceda, incremente em 1 uma váriavel relacionada ao índice daquele aluno, pois assim você também sempre deve verificar se essa variável para tal aluno não exceda 3, o que faria ele imune.

A complexidade da solução é $O(N \times Q)$,

Problema C. Cabeçada

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

Solução

Para este problema a primeira etapa da solução consiste em construir um grafo ponderado onde cada vértice representa um destino e cada aresta representa uma rua com distância D e altura da placa H. Apenas as arestas com $H = 0$ (sem placa) ou $H \geq 2.275$ (placa segura) são adicionadas ao grafo, pois as demais representam caminhos inseguros e devem ser completamente ignoradas. Em seguida, para cada par de origem e destino, calculamos a menor distância com o algoritmo de Dijkstra, já que o grafo possui pesos positivos e pode ter até 100 vértices e 200 arestas, o que garante eficiência dentro do limite de tempo. Depois de cada iteração se Henrique não consegue chegar ao destino i, ele deve continuar sua próxima tentativa a partir do local em que parou (ou seja, sua posição atual não muda).

A cada iteração, o algoritmo de Dijkstra é executado a partir da posição atual, retornando a menor distância ou `-1` caso o vértice de destino seja inalcançável.

Uma referência para o algoritmo de Dijkstra: https://cp-algorithms.com/graph/dijkstra.html

Complexidade total: $O(Q \times (M \log N))$.

Problema D. Disco de senha

Tempo limite: 2000 ms
Memória limite: 512 MiB
Autor: Ricardo Oliveira

Solução

Primeiramente, vamos remover a condição de que a string é circular concatenando a string com ela mesma. No primeiro exemplo de entrada, vamos transformar a string $S =$`fbcfbc` em $S =$`fbcfbcfbcfbc`. O problema se reduz agora a contar quantas substrings distintas de tamanho máximo K existem em S.

O *vetor de sufixos* (*Suffix Array*) de uma string S é o vetor de todos os sufixos de S, em ordem lexicografica crescente. Como exemplo, o vetor de sufixos de `fbcfbcfbcfbc` é:

```
 0:
 1: bc
 2: bcfbc
 3: bcfbcfbc
 4: bcfbcfbcfbc
 5: c
 6: cfbc
 7: cfbcfbc
 8: cfbcfbcfbc
 9: fbc
10: fbcfbc
11: fbcfbcfbc
12: fbcfbcfbcfbc
```

Não é necessário armazenar cópias da string em cada posição do vetor, mas sim apenas em qual posição em S o sufixo começa.

Note que toda substring de S é prefixo de algum de seu sufixo! Por exemplo, no sufixo `bcfbc` estão as substrings `b`, `bc`, `bcf`, `bcfb` e `bcfbc`.

Vamos construir a resposta de maneira incremental, percorrendo o vetor de sufixos em ordem; para cada sufixo processado, vamos incrementar na resposta a quantidade de **novas** substrings de tamanho máximo K contidas no sufixo (as que ainda não foram "vistas" anteriormente).

Para cada sufixo na posição i do vetor de sufixos, seja LCP_i o tamanho do maior prefixo comum (*Longest Commom Prefix*) de i com o sufixo $i-1$ no vetor de sufixos. Como exemplo, $LCP_2 = 2$ no exemplo dado, uma vez que o maior prefixo comum do sufixo na posição 2 (`bcfbc`) com o da posição 1 (`bc`) tem tamanho 2; Como exemplo exemplo, $LCP_6 = 1$; etc.

Para cada sufixo na posição i, note que todas as substrings nesse sufixo que tem tamanho até LCP_i já foram "vistas" em iterações anteriores; assim, há $min\{K, |\text{sufixo}|\} - LCP_i$ **novas** substrings de tamanho máximo K no sufixo i (ou nenhuma se $LCP_i \geq k$).

Uma referência para o algoritmo de construção do vetor de sufixos e cálculo de LCP é:

https://cp-algorithms.com/string/suffix-array.html

O problema pode ser resolvido em $O(N \lg N)$, mas a solução $O(N \lg^2 N)$ é rápida o bastante para os limites deste problema.

Problema E. Espaço na van

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

Solução

Para cada poltrona, verifique se sua largura é igual ou menor a L (isto é, se $l_i \leq L$). Se for, incremente um contador. Ao final da verificação, este contador terá o número de poltronas que podem ser usadas. A resposta é `SIM` se e somente se este número for maior ou igual a N.

Complexidade: $O(M)$

Problema F. Formação da dupla perfeita

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

Solução

Uma solução direta seria iterar em todos os $O(N^2)$ pares de discípulos e verificar, em $O(F)$, se cada par domina ao menos uma forma em, totalizando $O(N^2 \times F)$. Entretanto, esta solução não é rápida o bastante para os limites dados no problema.

Para reduzir a complexidade, note que é possível converter a string dada para cada discípulo em um *bitmask* de F bits: um inteiro onde o $i-$ésimo bit de sua representação binária é 1 se o $i-$ésimo caractere é `S`, ou 0 se é `N`. Como $F \leq 10$, este inteiro será no máximo $2^{10} - 1 = 1023$, que pode ser armazenado em uma variável `int`. Esta conversão é feita em $O(F)$ para cada discípulo, totalizando $O(NF)$.

Seja bm_i o *bitmask* do discípulo i. Para testar em $O(1)$ se os discípulos i e j podem ser uma dupla, basta verificar se $bm_i | bm_j = 2^F - 1$ (pois | é o operador *ou* bit-a-bit, e $2^F - 1$ é o *bitmask* com todos os bits em 1). Assim, a complexidade cai para $O(N^2)$. Entretanto, esta complexidade ainda não é rápida o bastante.

Note que há no máximo 2^F *bitmasks* possíveis, que é no máximo $2^{10} = 1024$ para os limites do problema!

Pré-compute $Q[bm]$, a quantidade de discípulos cuja *bitmask* é bm. Note que, para cada par de *bitmasks* bm_A e bm_B com $bm_A \neq bm_B$ onde $bm_A | bm_B = 2^F - 1$, há $Q[bm_A] \times Q[bm_B]$ duplas possíveis de serem formadas. Assim, itere entre os $O((2^F)^2)$ pares de *bitmasks* distintas e incremente a resposta em $Q(bm_A) \times Q(bm_B)$ para cada par possível.

Há um *corner case*, que é considerar quando bm_A e bm_B são a mesma *bitmask*. Note que o único caso em que é possível formar duplas com uma *bitmask* e ela mesma é a *bitmask* $2^F - 1$ (discípulos que dominam todas as técnicas). Para contar este caso, incremente a resposta em $(Q[2^F - 1] \times (Q[2^F - 1] - 1))/2$, o número de duplas que podem ser formadas entre eles.

A resposta pode não caber em um inteiro de 32 bits. Use *long long*.

Complexidade total: $O(NF + (2^F)^2)$.

Problema G. Genectorio

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

Solução

A solução do problema pode ser divida em 3 etapas: Primeiro, é necessário compreender que cada nucleotídeo (A, C, G, T) possui um par complementar fixo: A é pareado com T, T com A, C com G e G com C. Podemos representar essas letras por números inteiros ($A = 0, C = 1, G = 2, T = 3$) para facilitar o uso do operador `XOR`.

Em seguida, para cada posição i da string original S, o deslocamento utilizado na criptografia é determinado pelo `XOR` cumulativo entre X e os valores inteiros dos nucleotídeos já processados, isto é:

$$\text{deslocamento}_i = X \oplus S_1 \oplus S_2 \oplus \ldots \oplus S_i$$

onde $\oplus$ representa o operador `XOR` bit a bit.

Após calcular o deslocamento, ele é reduzido módulo 4 (pois o alfabeto tem apenas 4 letras), e o resultado indica o número de posições que devemos avançar no alfabeto circular $\{A, C, G, T\}$ a partir da letra complementar do nucleotídeo atual. Assim, a nova letra é obtida por:

$$\text{reposta} = (\text{complementar}(S_i) + \text{deslocamento}_i) \bmod 4$$

Complexidade total: $O(N)$.

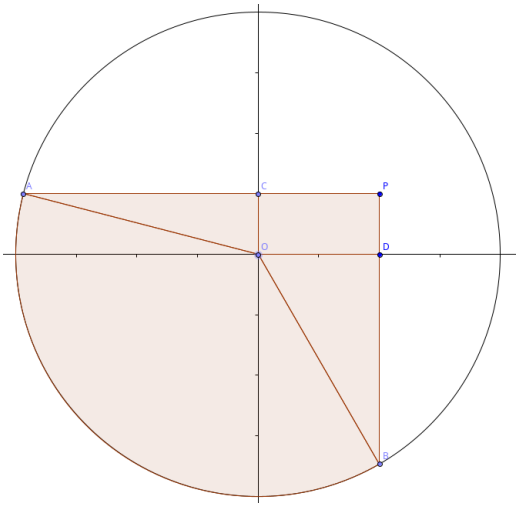
Problema H. Hora da Pizza

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

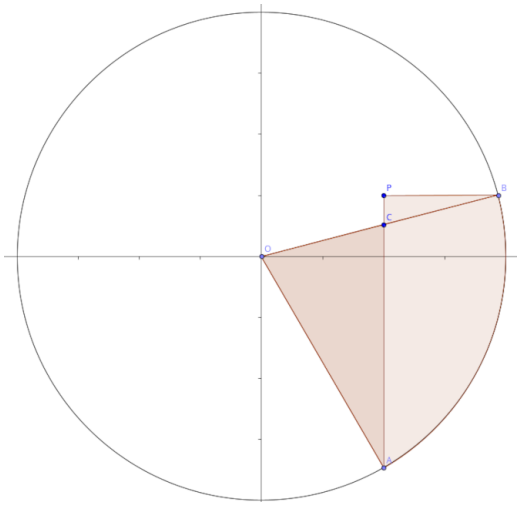
Solução

Primeiramente, faremos $X = |X|$ e $Y = |Y|$ de forma a colocar o ponto P no primeiro quadrante do círculo. Por simetria isto não altera a resposta, e nos permite considerar apenas o caso em que P está no primeiro quadrante.

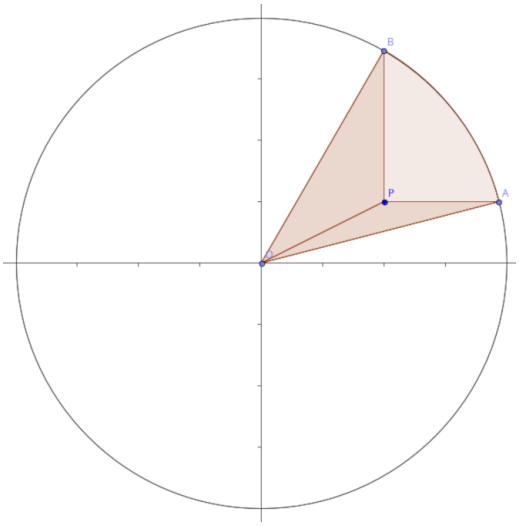
Cada área pode então ser calculada separadamente, de maneira analítica:



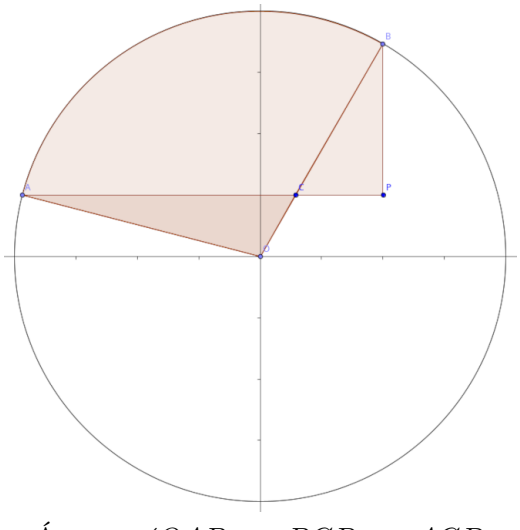
Área $= \angle OAB+ \triangle ACO+ \triangle OBD + \square OCPD$



Área $= \angle OAB+ \triangle BCP- \triangle OAC$



Área $= \angle OAB- \triangle OAP- \triangle OBC$



Área $= \angle OAB+ \triangle BCP- \triangle ACD$

Alternativamente, quando três das quatro áreas são calculadas, a quarta pode ser dada pela subtração da área total (πR^2) das demais áreas.

Lembre de sempre levar consigo uma boa implementação de funções da geometria!

Complexidade: $O(1)$

Problema I. Its Over

Tempo limite: 3000 ms
Memória limite: 512 MiB
Autor: Henrique Farias

Solução

Para resolver este problema uma abordagem seria pré-computar todos os número primos menores ou iguais que 2×10^7 usando o Crivo de Erasthotones. Com todos os primos já pré-computados podemos iterar por cada elemento do array guardando a soma daqueles nas posições primos numa váriavel e depois usar o crivo novamente para verificar se essa soma é primo também.

O valor 2×10^7 é suficiente porque existem menos de 200 primos até 1000, e portanto a soma dos valores nas posições primas não passa de 200×10^5.

Uma referência para o Crivo de Erasthotones: https://cp-algorithms.com/algebra/sieve-of-eratosthenes.html

Complexidade: $O(T \times loglogT + M)$ sendo $T < 2 \times 10^7$.

Problema J. Jogo

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

Solução

Partindo da segunda data podemos calcular a distância de tempo em relação a data anterior da seguinte forma: $(a_i - a_{i-1}) \times 360 + (m_i - m_{i-1}) \times 30 + (d_i - d_{i-1})$. Para cada iteração pode-se tirar o mínimo e máximo dos valores e no final imprimi-los como resposta correta.

Complexidade total: $O(N)$.

Problema K. Kurt, O camaleão que curte relógios

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

Solução

Primeiro, observamos que cada relógio opera em um ciclo de 12 horas, ou seja, há $12 \times 60 \times 60 = 43.200$ segundos distintos possíveis. É conveniente converter cada horário (h_i, m_i, s_i) em um único valor de segundos $t_i = h_i \times 3600 + m_i \times 60 + s_i$. Assim, cada relógio pode ser representado como um valor entre 0 e 43.199. A chave para resolver o problema é perceber que, se escolhermos um relógio alvo com tempo t_i, podemos calcular o custo (em cliques) necessário para ajustar todos os outros relógios para que coincidam com t_i. Quando pressionamos os botões dos demais relógios, cada relógio que está adiantado em relação a t_i precisa esperar até que o ciclo complete 43.200 segundos, enquanto os relógios atrasados exigem uma quantidade proporcional de cliques para alcançá-lo. Assim, o custo total para alinhar todos os relógios a um tempo t_i pode ser modelado em função das diferenças entre os tempos atuais e t_i. A solução esperada converte os horários para segundos e armazena em um vetor v. Em seguida, ordena o vetor e calcula a soma total dos tempos. Para cada relógio v_i, avalia-se o custo para torná-lo a referência, utilizando a fórmula:

$$\text{custo} = \text{soma} - n \cdot v_i + i \cdot MX,$$

onde $MX = 43.200$ representa o total de segundos em 12 horas, e o termo $i \cdot MX$ ajusta os relógios que ultrapassariam o ciclo completo de tempo. O menor custo encontrado entre todas as possibilidades é a resposta.

Complexidade total: $O(N \log N)$

Problema L. Laranja

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

Solução

Para a solução deste problema primeiro levamos em consideração que tanto as laranjas quanto as gavetas são distinguíveis, o problema equivale a contar o número de maneiras de particionar l elementos distintos em três subconjuntos ordenados (A, B, C) tais que $|A|, |B|, |C| \leq 4$ e $|A| + |B| + |C| = l$. Para cada possível par de tamanhos (j, k) das duas primeiras gavetas, o tamanho da terceira gaveta é $cur = l - j - k$. Se cur for válido (entre 0 e 4), o número de maneiras de escolher quais laranjas vão para cada gaveta é dado pelo número multinomial:

$$\frac{l!}{j!\, k!\, cur!}.$$

Assim, somando sobre todas as combinações válidas de (j, k), obtemos o número total de arranjos possíveis.

A solução esperada pré-calcula os fatoriais de 0! até 12! para permitir o cálculo eficiente das combinações multinomiais. Para cada dia, ele itera sobre todos os pares de tamanhos de gavetas (j, k) de 0 a 4 e soma os resultados válidos. Por fim, imprime o número total de maneiras correspondentes ou -1 se houver excesso de laranjas.

Complexidade total: $O(N)$

Problema M. Madrugada na Praia

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

Solução

Sobraram $B - V$ balões após a ventania. Assim, verifique (com `if`) se $B - V$ é divisível por N, testando se o resto da divisão é igual a 0 (isto é, se $(B - V)\%N = 0$). Se for, imprima $\frac{B-V}{N}$. Caso contrário, imprima -1.

Complexidade: $O(1)$