regionals | latin america region
icpc international collegiate programming contest
icpc.foundation

Maratona de Programação da SBC 2025

Este conjunto de problemas também foi utilizado simultaneamente nas seguintes competições:
The 2025 ICPC Gran Premio de Centroamerica,
The 2025 ICPC Gran Premio de Mexico e
Competencia Boliviana de Programación.

13 de setembro de 2025

Editorial

Os problemas e seus respectivos autores são:

- A - Alimentação Saudável - Francisco Arcos Filho
- B - Baralho Alho - Bruno Monteiro
- C - Collatz Polinomial - Francisco Arcos Filho
- D - Dominós - Luan Ícaro Pinto Arcanjo
- E - Expansão rodoviária - Maurício Collares
- F - Frangolino ali na mesa - Roberto Sales e Bruno Monteiro
- G - Gerador Universal - Vinicius dos Santos
- H - Habilidades especiais - Paulo Miranda
- I - Investigação Cósmica - Vinicius dos Santos
- J - João João - Vinicius dos Santos
- K - Knockout, suiço e outros formatos de torneio - Vinicius dos Santos
- L - LLMs - Vinicius dos Santos, Arthur Nascimento e Maurício Collares
- M - Muralhas reforçadas - Vinicius dos Santos

Além dos autores de problemas, colaboraram na elaboração da prova e deste editorial: Ricardo Anido, Paulo Cezar Pereira Costa, Rafael Grandsire, Fernando Kiotheka, Gabriel Pessoa, Marcos Kolodny e Juan Pablo Marin Rosas.

Promo:

SBC Sociedade Brasileira de Computação

v1.0

Problem A

Alimentação saudável

O problema pode ser modelado como uma tarefa de otimização: em cada turma, os alunos gostam de várias frutas, e as frutas podem ter interseções de preferência. A chave para minimizar o total de alunos é considerar, em cada turma, qual fruta tem o maior número de apreciadores, pois todos os demais gostos podem ser "explicados" por subconjuntos desses alunos.

Ou seja, na turma j, o número mínimo de alunos existentes é o máximo entre os números informados na coluna j. Repetindo esse raciocínio para cada turma e somando os máximos, obtemos o menor número possível de alunos.

Problem B

Baralho Alho

Temas necessários: permutações (e seus ciclos), string matching, teorema chinês do resto

Uma primeira observação é que a gente tem que quebrar a permutação nos seus ciclos, e resolver separado. Para cada ciclo, a gente quer saber quais rotações de um vetor dão igual ao outro. Uma forma de descobrir isso é duplicando o vetor e rodando um KMP (ou equivalente). Se algum vetor não tem nenhuma rotação que dá igual ao outro, `IMPOSSIVEL`.

Agora, a gente tem várias congruências da forma $k = a_i \pmod{m_i}$, e a gente precisa "combinar" elas. Isso é o que o teorema chinês do resto (CRT) resolve pra gente, mas o problema é que os números podem ficar grande demais, e a gente tomaria overflow. A solução pra isso é primeiro verificar se é possível ou não, e depois ir juntando equação uma por uma, até que o LCM fique grande demais.

Para verificar se tem solução, apresentamos 2 jeitos.

Jeito 1: note que, após deduplicar equações com mesmo mod, temos no máximo $\mathcal{O}(\sqrt{n})$ equações distintas, então a gente pode verificar se par-a-par as equações são satisfazíveis, e isso é verdade se e somente se o conjunto inteiro é satisfazível, aí a gente testa o `IMPOSSIVEL` assim. Referência: `https://math.stackexchange.com/q/1180796`.

Jeito 2:

A gente fatora m_i nos seus fatores primos distintos ($m_i = p_{i,1}^{e_{i,1}} \dots p_{i,x_i}^{e_{i,x_i}}$). Agora, vemos que

$$k = a_i \pmod{m_i}$$

se e somente se

$$\begin{aligned} k &= a_i \pmod{p_{i,1}^{e_{i,1}}} \\ &\dots \\ k &= a_i \pmod{mod p_{i,x_i}^{e_{i,x_i}}} \end{aligned}$$

Isso funciona já que o GCD desses mods que a gente criou é 1. Ou seja, a gente "quebrou" a congruência nos primos distintos.

Após fazer isso com todas as congruências, a gente vai ter que o conjunto inicial de congruências é equivalente ao conjunto de congruências:

$$\begin{aligned} k &= b_{0,0} \pmod{p_0^{e_{0,0}}} \\ k &= b_{0,1} \pmod{p_0^{e_{0,1}}} \\ &\dots \\ k &= b_{0,y_0} \pmod{p_0^{e_{0,y_0}}} \\ &\dots \\ k &= b_{t,0} \pmod{p_t^{e_{t,0}}} \\ k &= b_{t,1} \pmod{p_t^{e_{t,1}}} \\ &\dots \\ k &= b_{t,y_t} \pmod{p_t^{e_{t,y_t}}} \end{aligned}$$

onde p_j é primo. Vamos resolver para cada primo separado. Pensando em um só primo p:

$$\begin{aligned} k &= b_0 \pmod{p^{e_0}} \\ k &= b_1 \pmod{p^{e_1}} \\ &\dots \\ k &= b_{y_0} \pmod{p^{e_{y_0}}} \end{aligned}$$

Agora basta ver se não tem nenhuma "incompatibilidade" entre essas congruências. Se não há incompatibilidade dentro de nenhum primo, então sabemos que a resposta existe, pelo teorema chinês do resto.

Para agregar as congruências de um mesmo primo, se você tem equações incompatível, como no caso em que k tem que ser congruente a 0 mod 5 e a 1 mod 5, então não dá. Depois de fazer isso para toda potência do primo independentemente, a gente vai agregar de uma potência para a outra (agregar do 5 com $25 = 5^2$, com $125 = 5^3$, etc.) Isso é fácil: se você tem que $k = 1 \pmod 5$, então $k = 1 \pmod{25}$ também. Então é só ir fazendo isso "subindo" nas potências, e vendo se tem alguma inconsistência.

Após fazer isso (jeito 1 ou jeito 2): agora a gente sabe que é possível, e só queremos testar se é `DEMAIS` ou não. A gente vai juntando as equações uma por vez, até que o LCM $> 10^9$. Nesse caso, a resposta é $x =$ "o que a gente tem", ou $>= x + \text{LCM}$. Então a gente testa se x funciona. Se x funciona e $x <= 10^9$, imprime x. Senão, imprime `DEMAIS`.

Problem C

Collatz polinomial

Neste problema só é necessário fazer a simulação das operações, não sendo necessário fazer qualquer tipo de otimização adicional. Isso se deve ao fato da quantidade de iterações máxima ser baixa, sendo o caso de teste com maior resposta 101. Vetores podem ser utilizados, porém é comum a implementação utilizar inteiros, onde cada bit representa um coeficiente do polinômio.

Um polinômio pode ser representado por um vetor de inteiros que podem receber valores de 0 a 1, onde cada posição i do vetor é o valor do coeficiente de x^i. Por exemplo o polinômio $x+1$ é representado pelo vetor $[1, 1]$, e o polinômio x^2 por exemplo é o vetor $[0, 0, 1]$. O essencial é implementar as duas operações:

- Para multiplicar um polinômio por $(x + 1)$, primeiro multiple o polinômio por x . Isso pode ser feito movendo cada posição uma posição acima, por exemplo $[1, 0, 1]$, o polinômio $x^2 + 1$ vira $[0, 1, 0, 1]$, o polinômio $x^3 + x$. Aí some com o polinômio original em cada posição, ou seja $[0, 1, 0, 1] + [1, 0, 1] = [1, 1, 1, 1]$ representando o polinômio $x^3 + x^2 + x + 1$. Note que se na soma uma posição resulta em coeficiente 2, ela deverá se tornar 0, o que é idêntico a operação de xor, soma módulo 2.

- Para dividir o polinômio por x, basta deslocar todos os elementos em uma posição para baixo, por exemplo $[0, 1, 1]$ vira $[1, 1]$. Essa operação só é feita com o termo constante 0.

A simulação acaba quando o vetor for da forma [1] que representa o polinômio 1. Note que não é necessário que se reduza o tamanho do vetor em todas as operações, ele pode ser sempre do mesmo tamanho (por exemplo de tamanho 21).

Problem D

Dominós

Para resolver esse problema é necessário resolver os seguintes subproblemas:

- Dado um conjunto de peças de dominó, é possível vencer um jogo com este conjunto?
- Se cada subconjunto de um mesmo conjunto C possui um determinador valor, como calcular rapidamente a soma de todos os valores dos subconjuntos de um determinado subconjunto?

O primeiro problema pode ser resolvido formulando o jogo de dominó como um grafo não direcionado, onde cada vértice é uma peça de dominó e duas peças estão ligadas se elas possuem um número em comum. Se um grafo é semi-euleriano, ou seja, ele admite um caminho euleriano – um caminho que visita todas as arestas exatamente uma vez, mas que pode começar e terminar em vértices diferentes – então ele é um jogo de dominó onde é possível vencer. Um grafo é semi-euleriano se ele possui exatamente um componente conexo e existe exatamente zero ou dois vertices com grau ímpar. Isso é possível verificar fazendo uma busca em profundidade em $\mathcal{O}(N + M)$ onde N é a quantidade de peças de dominó e M é a quantidade de arestas que não excede 21.

O segundo problema pode ser resolvido utilizando uma técnica de programação dinâmica chamada "soma sobre subconjuntos" (*sum over subsets (SOS)*). Mais detalhes no seguinte blog do Codeforces: `https://codeforces.com/blog/entry/45223`. Desta forma a complexidade de se computar é de $\mathcal{O}(M \cdot 2^M)$.

Juntando os dois subproblemas, basta pré-computar para todo subconjunto de dominós se é possível vencer ou não o jogo com este subconjunto. Isto terá complexidade $\mathcal{O}(M \cdot 2^M)$. O valor então de cada subconjunto será 0 ou 1, e basta aplicar a técnica de programação dinâmica para saber qual a soma de todos os subconjuntos para cada subconjunto, também com complexidade $\mathcal{O}(M \cdot 2^M)$. Para resolver o problema, é necessário apenas descobrir qual o subconjunto de peças da consulta e imprimir o resultado que já foi calculado para aquele subconjunto.

Problem E
Expansão rodoviária

De forma resumida, o problema fornece como entrada um grafo G e busca determinar se existe uma árvore T tal que $T^2 = G$. Em caso afirmativo, o objetivo é retornar uma árvore que satisfaça essa condição.

Note primeiro que se G for um grafo completo, então sempre há resposta: Escolhendo qualquer vértice e conectando-o diretamente a todos os demais, obtemos uma árvore T cuja distância máxima é 2. Assim, T^2 é o grafo completo. Desse modo, vamos supor que $M \neq \frac{N(N-1)}{2}$ no que segue.

Existem algumas formas de se resolver este problema. Neste editorial, mostraremos duas abordagens, que serão expostas independentemente:

- A primeira abordagem usa *componentes biconexas* para achar candidatos para a vizinhança em T de um vértice. Mostra-se então que saber a vizinhança de um vértice qualquer em T é suficiente para calcular todas as demais.
- A segunda abordagem, que requer conhecimento de *Lex-BFS*, consiste em detectar as folhas da árvore original para, com isso, reduzir o grafo; em seguida, o processo é invertido para construir a árvore resultante.

Em ambas as soluções, vamos supor que o grafo G de entrada é, de fato, o quadrado de uma árvore (T^2), e nosso trabalho é apenas encontrar uma árvore T correspondente. Na prática, isso significa que certas situações que consideramos como garantidas na descrição abaixo, podem não acontecer. Nesse caso, a solução deve detectar tal falha e imprimir que não existe árvore válida.

Primeira abordagem: Componentes biconexas

Esta solução foi primariamente elaborada por Arthur Nascimento.

Para minimizar confusão, iremos escrever $N_T(v)$ para denotar a vizinhança de v na árvore T (que queremos computar) e $N_G(v)$ para denotar a vizinhança de v no grafo G da entrada, que supomos satisfazer $G = T^2$.

A solução se baseia no seguinte fato-chave: Suponha que, para algum vértice v, sabemos o conjunto $N_T(v)$. Então conseguimos reconstruir a árvore T toda. De fato, suponha que w é vizinho de v em T. Obviamente, v está na vizinhança de w. Além disso, os outros vizinhos de w em T são os elementos de $N_G(w)$ que estão a distância exatamente 2 de v em T. Em termos simbólicos,

$$N_T(w) = \{v\} \cup (N_G(w) \cap (N_G(v) \setminus N_T(v))).$$

Assim, se soubermos um $N_T(v)$, podemos fazer uma busca em profundidade que calcula as vizinhanças dos vizinhos do vértice atual e procede recursivamente.

Nosso objetivo é então achar $N_T(v)$ para *algum* vértice v. Suponha que sabemos encontrar um v não é folha de T. Considere o grafo $T' := T \setminus v$ obtido de T através da remoção do vértice v e das arestas incidentes a ele. Se $N_T(v) = \{w_1, \ldots, w_k\}$ ($k \geq 2$), então T' tem k componentes $C_1, \ldots, C_k$, e podemos numerá-las de modo que $w_i \in C_i$ para todo $1 \leq i \leq k$. Iremos também denotar por C'_i o conjunto $C_i \setminus \{w_i\}$.

Note que, para $i \neq j$, qualquer caminho em T entre C'_i e w_j necessariamente passa por w_i e logo depois por v. Afirmamos que isso implica que $N_T(v) = \{w_1, \ldots, w_k\}$ é componente biconexa do grafo $G' := G \setminus v$. De fato:

- Quaisquer dois vértices de $N_T(v)$ estão diretamente conectados em G', e portanto $N_T(v)$ é biconexo.

- A remoção de w_i desconecta C'_i do resto de $N_T(v)$ em G'. Como todo vértice fora de $N_T(v)$ está em algum C'_i, vemos que $N_T(v)$ é maximal biconexo em G'.

Intuitivamente, podemos explicar o segundo ponto do seguinte modo: um caminho entre C'_i e o resto de $N_T(v)$ em G' teria que "pular" os vértices consecutivos w_i e v do caminho em T, e as "arestas-atalho" (isto é, arestas de T^2 que não estão em T) só permitem pular um vértice por vez.

Além disso, se $|C_i| \geq 2$, pode-se verificar que C_i é também uma componente biconexa de G' (intuitivamente, podemos "pular" qualquer vértice removido de C_i usando as "arestas-atalho"). Em outras palavras, se $|C_i| \geq 2$, então w_i é vértice de corte, e uma das duas componentes biconexas de G' que contém w_i é exatamente $N_T(v)$.

É possível mostrar que esses são os únicos vértices de corte, e que nenhum vértice está em mais de duas componentes biconexas. Assim, se soubermos que v não é folha de T, basta então achar um vértice de corte c de $G \setminus v$. Para cada componente biconexa C que contém c, testamos se C pode ser $N_T(v)$ calculando as demais vizinhanças em T com o fato-chave do começo da explicação e verificando se a árvore resultante de fato satisfaz $T^2 = G$. Caso o vértice de corte esteja em mais de duas componentes biconexas, o problema não tem solução.

Na verdade, se começarmos com um vértice w qualquer e $G \setminus w$ não for biconexo, então w certamente é não-folha e podemos proceder como antes. Caso isso não ocorra, w é uma folha, mas podemos achar uma não-folha em $N_G(w)$ facilmente. De fato, se w for folha, um elemento $v \in N_G(w)$ é também folha se e só se $N_G(v) \cup \{v\} = N_G(w) \cup \{w\}$. Assim, conseguimos achar um $v \in N_G(w)$ que não é folha e usar v para fazer o procedimento anterior.

Desafio: O procedimento acima testa duas árvores no pior caso. Em que situações as duas árvores construídas são respostas válidas para o problema?
Desafio: Implemente essa abordagem em tempo $O(M)$.

Segunda abordagem: Lex-BFS

Esta solução é baseada no artigo "Algorithms for Square Roots of Graphs", de Yaw-Ling Lin e Steven S. Skiena.

Uma observação sutil, porém fundamental, é que um vértice é uma folha em T se, e somente se, sua vizinhança em G formar uma clique. O fato de que "se um vértice é uma folha, então sua vizinhança forma uma clique" é algo fácil de provar e bem intuitivo. Para a recíproca, podemos usar a contrapositiva: se um vértice c não é uma folha em T, então deve existir um caminho como $a - b - c - d$ na árvore original. Isso implica que a e d estão na vizinhança de c em T^2, mas como a distância $d_T(a, d) = 3$, não haverá a aresta $a - d$ em T^2. Portanto, a vizinhança de c em G não forma uma clique.

Para encontrar as folhas, poderíamos iterar sobre os vértices e checar se suas vizinhanças formam uma clique. Contudo, fazer isto de forma direta resulta em uma complexidade alta, o que pode não ser eficiente o suficiente. Em grafos da forma T^2, cada aresta está em no máximo dois cliques maximais (cliques que não são subconjuntos de uma clique maior), e portanto a soma dos tamanhos dos cliques maximais é $O(|E|)$. Além disso, se T é uma árvore, então T^2 é um *grafo cordal*. Existe um algoritmo eficiente para encontrar os cliques maximais utilizando um Lex-BFS para achar uma *ordenação de eliminação perfeita* de um grafo cordal, e a partir disso computar os cliques maximais. Para mais referências, acesse: `https://en.wikipedia.org/wiki/Chordal_graph`. Ao encontrar os cliques maximais, é fácil identificar as folhas, pois são os vértices que pertencem a apenas uma clique maximal. Com isto, temos um algoritmo para encontrar as folhas.

Uma observação seguinte para a resolução do problema é que, dada uma folha f, podemos analisar sua vizinhança em G, Separamos essa vizinhança em dois subconjuntos: F (vértices que também são folhas) e I (vértices que são internos, isto é, que não são folhas). Podemos observar que o conjunto F é formado pelos vértices adjacentes a f que têm mesmo grau que f em G. A partir do tamanho do conjunto I, podemos fazer as seguintes inferências:

- Se $|I| \leq 1$, então a árvore é uma estrela.
- Se $|I| \geq 3$, então o pai do conjunto de folhas $F \cup \{f\}$ na árvore original T é o centro de uma estrela formada pelos vértices em I.
- Se $|I| = 2$, então existe uma aresta em T conectando os dois vértices em I. Para determinar qual deles é o pai de $F \cup \{f\}$, temos as seguintes observações:
  - Se algum dos vértices em F já teve uma aresta inferida em um passo anterior do processo de reconstrução para um dos vértices em I, então essa conexão define o pai.
  - O vértice não conectado v é removido no mesmo momento que o vértice conectado u. Isso só poderia acontecer na última iteração, quando o grafo se torna uma árvore estrela e precisamos inferir quem é o centro e quem são as folhas. Porém, a penúltima operação (que não é a considerada no cenário atual, dado que a árvore após remover F tem diâmetro de pelo menos 4) seria necessariamente uma operação com $|I| = 2$, envolvendo v e um outro vértice w. Nesse caso, não seria possível considerar u como o centro da estrela final, e sim como uma das folhas, o que o levaria a ser removido primeiro.

Prova: Suponha falso. Sejam u o vértice de I conectado a F em T e v um vértice de I não conectado.

- **Caso 1:** O vértice não conectado v é removido antes do vértice conectado u. Isto significa que, no momento da remoção de v, o vértice u ainda não era uma folha e possuía outras conexões na árvore remanescente. Então $|I| \geq 3$, o que é uma contradição.
- **Caso 2:** O vértice u e o vértice v são removidos ao mesmo tempo. Isto só poderia acontecer se o grafo remanescente fosse uma estrela. Contudo, a premissa para esta análise é que a árvore após a remoção de F tem diâmetro de pelo menos 4, o que descarta a possibilidade de o grafo ser uma simples estrela nesse estágio.

Com isto, temos um algoritmo para reconstruir a árvore. Realizamos o processo de remoção de folhas e, em seguida, revertemos as operações. Essa reversão pode gerar até duas árvores candidatas considerando a aresta inferida na penúltima operação para ser o centro da estrela da última operação. Ao final, simplesmente checamos se alguma dessas duas opções T satisfaz a condição $T^2 = G$.

Problem F
Frangolino ali na mesa

Sejam $X_1, \ldots, X_Q$ as instruções que o robô recebeu. Vamos resolver o problema de trás pra frente, considerando adicionar ao problema uma instrução de cada vez. Mais especificamente, vamos inicialmente considerar que o robô começa na mesa X_Q e não executa nenhuma instrução. Depois, vamos considerar que o robô começa na mesa X_{Q-1} e que executará somente a instrução X_Q, e assim por diante.

De forma mais geral, vamos analisar como o valor esperado da quantidade das milanesas se comporta conforme consideramos as operações do nosso robô de trás pra frente. Mais formalmente, vamos definir $E(i, j)$ como o valor esperado de milanesas na mesa j se o robô começar na mesa X_i e processar as instruções $X_{i+1}, \ldots, X_Q$. Por conveniência, podemos estabelecer que $X_0 = 1$. Portanto, desejamos descobrir $E(0, j)$ para toda mesa j.

Vamos olhar mais de perto o que acontece com os nossos valores esperados quando consideramos uma instrução nova. Mais especificamente, vamos analisar a diferença entre $E(i, j)$ e $E(i + 1, j)$. Não é difícil perceber que $E(i, j) \neq E(i + 1, j)$ sse $X_i = j$. Portanto, somente uma mesa tem o seu valor esperado modificado conforme consideramos uma instrução nova.

Por último, podemos definir a seguinte recorrência para $E(i, j)$.

$$E(i,j) = \begin{cases} E(i+1,j) + \frac{X_{i+1}}{2} + \frac{X_{i+2}}{4} + \cdots + \frac{X_Q}{2^{Q-i}} & \text{se } X_i = j \\ E(i+1,j) & \text{se } X_i \neq j \end{cases}$$

Já que agora existe $\frac{1}{2}$ de chance de a mesa X_i receber X_{i+1} milanesas, $\frac{1}{4}$ de chance da mesa receber X_{i+2} milanesas, e assim por diante.

Apesar da recorrência a princípio parecer bidimensional, é notável que somente um valor de j é modificado a cada iteração. Isso permite que a resolvamos em $O(n)$, iterando de trás pra frente pelas operações e mantendo um acumulador com as somas de $\frac{X_{i+1}}{2} + \frac{X_{i+2}}{4} + \cdots + \frac{X_Q}{2^{Q-i}}$.

Problem G

Gerador universal

A observação principal para resolver o problema é que, para um B fixo e assumindo que não é possível aplicar operações com $i < 0$, existe uma única maneira de zerar os bits de C — ou nenhuma.

A intuição por trás desta afirmação baseia-se em um algoritmo guloso: ao varrer os bits da esquerda para a direita, sempre que um bit ativo é encontrado, aplica-se a operação alinhando o bit mais significativo de B com esse bit. Uma prova mais rigorosa de por que isso funciona é observar que as máscaras geradas por B, considerando apenas as operações com $i \geq 0$, formam um conjunto de vetores linearmente independentes em $\mathbb{Z}_2^N$.

Ao iterar sobre todas as máscaras B possíveis e as diferentes operações com $i < 0$, a aplicação do algoritmo guloso resulta em uma complexidade de $O(mn \cdot 2^{2m})$, onde $m = |B|$ e $n = |C|$. Para os limites do problema, essa complexidade é suficiente para que uma implementação com uma constante razoável seja aceita. No entanto, algumas observações podem otimizar a complexidade:

- Pode-se melhorar a constante do código por um fator de 2 ao perceber que a máscara ótima sempre terá o bit menos significativo ativo. Caso contrário, seria possível deslocar todos os bits da máscara para a direita, mantendo o mesmo número de operações.
- É possível remover o fator m da complexidade utilizando métodos mais eficientes para aplicar a operação, em vez de um simples laço com uma operação XOR.

Problem H

Habilidades especiais

Apresentaremos duas soluções para o problema, ambas com complexidade $O(K \cdot (N + M + 2^K))$, onde K é a quantidade de dígitos, N é o número de alunos e M é a quantidade de números especiais. A primeira solução utiliza convolução de OR com Fast Walsh–Hadamard Transform (FWHT), enquanto a segunda solução emprega a técnica de DP sobre subconjuntos (SOS DP).

Solução por Convolução: $O(K \cdot (N + M + 2^K))$

A ideia da solução é modelar o problema como uma Convolução de OR, usando Fast Walsh–Hadamard Transform.

Primeiramente, definimos os seguintes polinômios:

$$F(x) = \sum x^{H_i}$$

$$F_3(x) = conv_{or}(conv_{or}(F(x), F(x)), F(x))$$

Agora, consideramos o polinômio:

$$F_3(x) = \sum a_k \cdot x^k$$

O coeficiente a_k representa a quantidade de trios (com repetição de elementos permitida) cuja OR (bit a bit) seja k. No entanto, queremos apenas as equipes válidas, ou seja, formadas por **3 alunos distintos**. Portanto, é necessário ajustar essa contagem.

Para isso, usamos outros dois polinômios auxiliares:

$$F_2(x) = \sum b_k \cdot x^k$$

Aqui, o coeficiente b_k representa a quantidade de trios em que **pelo menos dois dos três elementos são iguais** e cuja OR (bit a bit) seja k.

$$F_1(x) = \sum c_k \cdot x^k$$

Neste caso, o coeficiente c_k corresponde à quantidade de trios onde **os três elementos são iguais** e cuja OR (bit a bit) seja k.

Correção das Contagens

Agora que temos as contagens para todos os casos, precisamos isolar as combinações com **3 elementos distintos**. Para isso, aplicamos o princípio da inclusão-exclusão.

- a_k inclui todas as combinações, inclusive aquelas com repetições de elementos.
- b_k conta as combinações com pelo menos 2 elementos iguais.
- c_k conta as combinações com todos os elementos iguais.

A quantidade de trios com 3 elementos distintos é então dada por:

$$resposta_k = \frac{a_k - 3 \cdot b_k + 2 \cdot c_k}{6}$$

A justificativa é a seguinte:

- Subtraímos $3 \cdot b_k$ porque um trio com dois elementos iguais pode ter o elemento distinto em qualquer uma das três posições.

- Porém, ao fazer isso, acabamos subtraindo excessivamente os casos com todos os elementos iguais (que aparecem em c_k), pois eles foram subtraídos 3 vezes e só deveriam ser subtraídos 1 vez. Por isso, somamos de volta $2 \cdot c_k$.
- Por fim, dividimos o resultado por 6 para corrigir as permutações dos elementos ($3! = 6$), já que estamos interessados apenas em conjuntos de alunos distintos, e não em suas ordenações.

Resumo

A estratégia computacional envolve:

1. Construir os polinômios descritos anteriormente.
2. Precomputar para todos os valores possíveis usando as convoluções, que podem ser feitas de forma eficiente com FWHT.
3. Acessar a resposta para cada número especial E_i.

Isso permite resolver o problema de maneira eficiente mesmo com restrições grandes.

Complexidade

A complexidade esperada da solução é:

- $O(K \cdot (N + M + 2^K))$, onde K é a quantidade de dígitos, N é o número de alunos e M é a quantidade de números especiais.

Solução por transformada: $O(K \cdot (N + M + 2^K))$

Notação

Vamos definir a terminologia para formalizar a solução:

- K: O número total de habilidades distintas.
- N: O número total de alunos.
- Uma **máscara** (mask) é um inteiro de K bits, que representa um subconjunto de habilidades.
- f[mask]: A quantidade de alunos cujo conjunto de habilidades é *exatamente* representado por `mask`.
- F[mask]: A quantidade de alunos cujo conjunto de habilidades é um *subconjunto* de `mask`. Matematicamente, $F[\text{mask}] = \sum_{\text{sub} \subseteq \text{mask}} f[\text{sub}]$.
- g[mask]: A resposta final. O número de equipes de 3 alunos distintos cuja união de habilidades é *exatamente* `mask`.
- G[mask]: O número de equipes de 3 alunos distintos cuja união de habilidades é um *subconjunto* de `mask`. Matematicamente, $G[\text{mask}] = \sum_{\text{sub} \subseteq \text{mask}} g[\text{sub}]$.

Passo 1: Contagem Inicial dos Alunos

Primeiro, processamos a entrada para popular o vetor f. Iteramos pelos N alunos e, para cada aluno com conjunto de habilidades H_i, incrementamos $f[H_i]$.

```
f = array de tamanho 2^K inicializado com zeros
Para cada aluno i de 1 a N:
    mask_aluno = converte a string H_i para inteiro
    f[mask_aluno]++
```

Passo 2: DP Sobre Subconjuntos (SOS DP) - Ida

O próximo passo é calcular $F[\text{mask}]$ para todas as 2^K máscaras possíveis. A definição é $F[\text{mask}] = \sum_{\text{sub}\subseteq\text{mask}} f[\text{sub}]$. Uma maneira ingênua seria iterar por todas as sub-máscaras para cada máscara, resultando em uma complexidade de $O(3^K)$, o que é inviável para $K = 20$.

Usamos a DP Sobre Subconjuntos, que calcula esses valores em $O(K \cdot 2^K)$. A ideia é iterar sobre cada bit i de 0 a $K-1$ e, para cada máscara, adicionar a contribuição das máscaras que diferem apenas no i-ésimo bit. Para mais detalhes sobre essa DP leia o blog do Codeforces: `https://codeforces.com/blog/entry/45223`.

```
// Inicialmente, F é uma cópia de f
F = f

// Itera sobre cada bit
Para i de 0 a K-1:
    // Itera sobre todas as máscaras
    Para mask de 0 a (2^K - 1):
        // Se o i-ésimo bit de mask está ligado
        Se (mask >> i) & 1:
            // Adiciona a contribuição da máscara sem o i-ésimo bit
            F[mask] += F[mask ^ (1 << i)]
```

Ao final deste processo, $F[\text{mask}]$ conterá o número total de alunos cujas habilidades são um subconjunto de `mask`.

Passo 3: Calculando Combinações de Equipes

Agora, vamos calcular $G[\text{mask}]$. Se a união das habilidades de uma equipe de 3 alunos é um subconjunto de `mask`, então cada um dos 3 membros deve, individualmente, possuir um conjunto de habilidades que é um subconjunto de `mask`.

O número de alunos que satisfazem essa condição é exatamente $F[\text{mask}]$. Portanto, o número de maneiras de formar uma equipe de 3 pessoas a partir deste grupo de alunos é dado pela combinação de $F[\text{mask}]$ escolhe 3.

$$G[\text{mask}] = \binom{F[\text{mask}]}{3} = \frac{F[\text{mask}] \cdot (F[\text{mask}] - 1) \cdot (F[\text{mask}] - 2)}{6}$$

Calculamos $G[\text{mask}]$ para todas as máscaras de 0 a $2^K - 1$.

Passo 4: Invertendo a Transformada (SOS DP - Volta)

Neste ponto, temos $G[\text{mask}] = \sum_{\text{sub}\subseteq\text{mask}} g[\text{sub}]$. Nosso objetivo é encontrar $g[\text{mask}]$. Para isso, aplicamos o Princípio da Inclusão-Exclusão, que pode ser implementado como a operação inversa da SOS DP.

A relação para obter g é:

$$g[\text{mask}] = G[\text{mask}] - \sum_{\text{sub}\subset\text{mask}} g[\text{sub}]$$

Isso pode ser implementado de forma eficiente com uma estrutura de loop semelhante à da ida, mas com subtração.

```
// Inicialmente, g é uma cópia de G
g = G

// Itera sobre cada bit
```

```
Para i de 0 a K-1:
    // Itera sobre todas as máscaras
    Para mask de 0 a (2^K - 1):
        // Se o i-ésimo bit de mask está ligado
        Se (mask >> i) & 1:
            // Remove a contribuição das sub-máscaras
            g[mask] -= g[mask ^ (1 << i)]
```

Após a execução deste algoritmo, o vetor g conterá a resposta para cada possível subconjunto especial.

Finalização

Para cada uma das M strings de consulta E_i, convertemos a string para sua representação inteira `mask_especial` e imprimimos o valor de g[mask_especial].

Complexidade Total

- **Passo 1 (Contagem):** $O(N \cdot K)$
- **Passo 2 (SOS DP - Ida):** $O(K \cdot 2^K)$
- **Passo 3 (Cálculo de G):** $O(2^K)$
- **Passo 4 (SOS DP - Volta):** $O(K \cdot 2^K)$
- **Consultas:** $O(M \cdot K)$

A complexidade dominante é $O(K \cdot (N+M+2^K))$. Com $N = 10^5, K = 20, M = 5 \cdot 10^4$, esta abordagem é eficiente o suficiente para ser aceita.

Bonus: FWHT

Além do método SOS-DP, a etapa de soma sobre subconjuntos também pode ser implementada de forma equivalente usando a Fast Walsh-Hadamard Transform (FWHT). A FWHT é uma transformada discreta que, quando aplicada com a operação de OR, computa exatamente a soma sobre subconjuntos. A transformada inversa da FWHT realiza a operação oposta, que é o que precisamos para o último passo. Ou seja, podemos resolver da mesma forma usando FWHT.

Problem I

Investigação Cósmica

O objetivo é encontrar o maior valor inteiro possível para o raio da órbita da primeira estrela, R_1, de forma que exista uma configuração válida de raios $R_1, R_2, \ldots, R_N$.

Análise das Restrições e Relações Fundamentais

Primeiramente, vamos formalizar as regras do problema. Temos N estrelas, e a i-ésima estrela está na posição (x_i, y_i).

Distância entre Estrelas

As estrelas i e $i+1$ estão sempre alinhadas horizontal ou verticalmente. Portanto, a distância euclidiana entre elas, que denotaremos por d_i, é simplesmente a diferença absoluta em uma das coordenadas, ou seja

$$d_i = \sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2} = |x_i - x_{i+1}| + |y_i - y_{i+1}|,$$

uma vez que um dos dois termos será sempre zero.

Condição de Transição de Órbita

A nave abandona a órbita da estrela i no ponto mais próximo da estrela $i+1$ e imediatamente começa a orbitar a estrela $i+1$. Para que essa transição seja contínua, o ponto de saída da órbita i deve ser o mesmo ponto de entrada na órbita $i+1$. Esses pontos estão no segmento de reta que conecta as estrelas i e $i+1$. Isso implica que as duas órbitas devem se "tocar" nesse ponto. Geometricamente, a soma de seus raios deve ser exatamente a distância entre seus centros:

$$R_i + R_{i+1} = d_i \quad \text{para } 1 \leq i < N.$$

Condições sobre os Raios

O problema estabelece duas condições para os raios:

1. Cada raio R_i deve ser um inteiro positivo: $R_i \geq 1$.
2. Para cada $1 \leq i < N$, o raio R_i deve ser estritamente menor que a distância para a próxima estrela: $R_i < d_i$.

Notemos que a segunda condição, $R_i < d_i$, é uma consequência natural da primeira e da condição de transição. Se $R_i + R_{i+1} = d_i$ e sabemos que $R_{i+1} \geq 1$, então:

$$R_i = d_i - R_{i+1} \leq d_i - 1.$$

Expressando todos os Raios em Função de R_1

Nosso objetivo é encontrar o maior R_1 possível. Para fazer isso, vamos expressar cada R_i em termos de R_1 e das distâncias d_i. A partir da relação $R_{i+1} = d_i - R_i$, podemos derivar as seguintes equações:

$$\begin{aligned}
R_2 &= d_1 - R_1 \\
R_3 &= d_2 - R_2 = d_2 - (d_1 - R_1) = (d_2 - d_1) + R_1 \\
R_4 &= d_3 - R_3 = d_3 - ((d_2 - d_1) + R_1) = (d_3 - d_2 + d_1) - R_1 \\
&\vdots
\end{aligned}$$

Podemos observar um padrão de soma alternada. Vamos definir uma constante C_i que representa a parte da expressão de R_i que depende apenas das distâncias:

$$R_i = C_i + (-1)^{i-1} R_1,$$

onde $C_1 = 0$ e, para $i > 1$, os C_i são as somas alternadas das distâncias:

$$C_i = \sum_{j=1}^{i-1} (-1)^{i-1-j} d_j = d_{i-1} - d_{i-2} + \cdots \pm d_1.$$

Uma forma mais simples e computacionalmente eficiente de calcular C_i é através da recorrência:

$$C_1 = 0,$$

$$C_i = d_{i-1} - C_{i-1} \quad \text{para } i > 1.$$

Estabelecendo os Limites para R_1

Agora, aplicamos a restrição $R_i \geq 1$ para cada i de 1 a N:

$$C_i + (-1)^{i-1} R_1 \geq 1.$$

Temos dois casos, dependendo da paridade de i:

- **Caso 1: i é ímpar.** Neste caso, $i - 1$ é par, e $(-1)^{i-1} = 1$. A inequação se torna:

  $$C_i + R_1 \geq 1 \implies R_1 \geq 1 - C_i.$$

  Isso nos dá um **limite inferior** para R_1.

- **Caso 2: i é par.** Neste caso, $i - 1$ é ímpar, e $(-1)^{i-1} = -1$. A inequação se torna:

  $$C_i - R_1 \geq 1 \implies C_i - 1 \geq R_1 \implies R_1 \leq C_i - 1.$$

  Isso nos dá um **limite superior** para R_1.

Para que uma solução válida exista, R_1 deve satisfazer todas essas N inequações simultaneamente. Portanto, R_1 deve estar em um intervalo $[L, R]$, onde L é o máximo de todos os limites inferiores e R é o mínimo de todos os limites superiores.

$$L = \max(\{1 - C_i \mid i \text{ é ímpar}\}),$$

$$R = \min(\{C_i - 1 \mid i \text{ é par}\}).$$

Solução Final

Uma configuração válida de órbitas existe se, e somente se, o intervalo de valores possíveis para R_1 não for vazio, ou seja, se $L \leq R$. Se essa condição for satisfeita, o maior valor inteiro possível para R_1 é, por definição, R. Se $L > R$, não há nenhum valor de R_1 que satisfaça todas as restrições, e a missão é impossível.

Algoritmo

1. Leia N e as coordenadas das N estrelas.
2. Calcule as distâncias $d_i = |x_i - x_{i+1}| + |y_i - y_{i+1}|$ para i de 1 a $N - 1$.
3. Inicialize as variáveis de limite: `long long L = 1`, `long long R = infinito` (um valor muito grande).
4. Inicialize `C = 0`.
5. Itere de $i = 1$ até N:
   (a) Se i é ímpar: $L = \max(L, 1 - C)$.
   (b) Se i é par: $R = \min(R, C - 1)$.
   (c) Se $i < N$, atualize C para o próximo passo: $C = d_i - C$.
6. Após o laço, se $L \leq R$, a resposta é R.
7. Caso contrário ($L > R$), a missão é impossível, e a resposta é `-1`.

Problem J

João João

Resumo do Problema

O objetivo do problema é determinar o número mínimo de novas tarefas que precisam ser criadas para que se tenha ao menos uma tarefa de cada nível de dificuldade, de 1 a 4.

A entrada consiste em uma lista de 10 números inteiros, cada um representando o nível de dificuldade de uma tarefa já existente.

Solução

A solução consiste em verificar quais níveis de dificuldade já estão presentes entre as 10 tarefas fornecidas e, em seguida, calcular quantos níveis ainda estão faltando.

Por exemplo, se as tarefas existentes cobrem os níveis 1, 2 e 4, apenas o nível 3 está faltando. Portanto, seria necessário criar apenas 1 nova tarefa. Se todos os quatro níveis já estiverem presentes, nenhuma nova tarefa precisa ser criada.

A implementação pode ser feita de forma eficiente utilizando uma estrutura de dados para armazenar os níveis de dificuldade únicos já existentes, como um vetor de booleanos de tamanho 4, um conjunto (`set`) ou mesmo 4 variáveis separadas. A solução mais simples utiliza um laço de repetição para a leitura da entrada, mas como o tamanho da entrada é constante, era também possível a solução sem este tipo de estrutura. A resposta é dada pela subtração entre 4 e a quantidade de níveis distintos já encontrados na entrada.

Problem K

Knockout, suíço e outros formatos de torneio

Para resolver o problema, precisamos descobrir o menor valor de X t.q. para toda rodada do torneio, a quantidade de jogadores com pontuação (i, j), $0 \leq i < A, 0 \leq j < B$, seja um número par.

Podemos deduzir a seguinte fórmula para a quantidade de jogadores que alcançam a pontuação (i, j) no torneio:

$$cnt(i,j) = \frac{X}{2^{i+j}} \binom{i+j}{i}$$

Uma explicação possível é que os jogadores que alcançam a pontuação (i, j) são os que vencem i partidas e perdem j partidas. Existem $\binom{i+j}{i}$ maneiras de escolher quais i partidas esses jogadores vencem e, por consequência, quais j partidas eles perdem. Chamemos esta sequência de vitórias e derrotas de *histórico* do jogador. Existem exatamente $\frac{X}{2^{i+j}}$ jogadores com cada um dos $\binom{i+j}{i}$ históricos possíveis.

Seja $f(x)$ uma função t.q. $2^{f(x)}$ é a maior potência de 2 que divide x. Temos que:

$$f(cnt(i,j)) = f(X) - (i+j) + f((i+j)!) - f(i!) - f(j!)$$

Como precisamos que $cnt(i, j)$ seja um número par positivo para todo i, j, temos que $f(cnt(i, j)) > 0$ para todo i, j. Assim, temos que:

$$\begin{gathered} f(X) - (i+j) + f((i+j)!) - f(i!) - f(j!) > 0 \\ f(X) > i + j - f((i+j)!) + f(i!) + f(j!) \end{gathered}$$

Uma das manifestações da Fórmula de Legendre nos diz que $f(x) = x - \text{popcount}(x)$, onde $\text{popcount}(x)$ é a quantidade de bits 1 em x. Assim, temos que:

$$\begin{aligned} f(X) > G(i,j) &= i + j - (i + j - \mathit{popcount}(i+j)) - (i - \mathit{popcount}(i)) - (j - \mathit{popcount}(j)) \\ &= i + j + \text{popcount}(i+j) - \text{popcount}(i) - \text{popcount}(j) \end{aligned}$$

Seja $Y = \max_{i,j} G(i, j)$. Não é dificil perceber que a resposta que procuramos é $Y + 1$. Contudo, computar o valor de Y passando por todos os valores de i, j possíveis é inviável dados os limites do problema.

Note que pode se dizer que as parcelas $\mathit{popcount}(i+j)$, $\mathit{popcount}(i)$ e $\mathit{popcount}(j)$ da fórmula acima assumem valores dentre 0 e $\log_2(A+B)$. Não é difícil perceber que $-\log_2(A+B) \leq \mathit{popcount}(i+j) - \mathit{popcount}(i) - \mathit{popcount}(j) \leq 0$.

Com isso, podemos deduzir duas desigualdades importantes:

$$\begin{gathered} G(A-1, B-1) \geq A - 1 + B - 1 - \log_2(A+B), \\ G(i,j) \leq i + j. \end{gathered}$$

Assim, se (i, j) está mais distante de $(A-1, B-1)$ que $\log_2(A+B)$, então $G(i, j)$ nunca será maior que $G(A-1, B-1)$.

Como sabemos que $\log_2(A+B) \leq 61$, podemos computar o valor de $G(i, j)$ somente para uma submatriz de tamanho 61×61 na parte inferior direita da matriz $(A-1) \times (B-1)$, que é mais

que suficiente para os valores de A, B dados na entrada, que nos dá uma solução com complexidade $O(\log^2(A+B))$.

Desafio 1: Resolva o problema em $O(\log(A+B))$.
Desafio 2: Resolva o problema em $O(\log\log(A+B))$.

Problem L
LLMs

Neste problema, simular os passos descritos é suficiente para passar sem algoritmos ou estruturas de dados arrojados. Uma consulta é composta de:

1. Criar um janela pegando as últimas K palavras da entrada. Desta forma, temos uma janela inicial de K palavras. Você já pode imprimir todas as palavras que são lidas da entrada para facilitar.

2. Em seguida fazemos um laço que para quando achamos a palavra ou quando a janela atual for vazia. Se a janela atual acabar com 0 palavras, significa que não encontramos a palavra no texto e a resposta é '$*$' para esta consulta.

3. Dentro do laço, iteramos i de 0 até $M-$ o tamanho da janela atual exclusive. Comparamos todas as palavras da base de conhecimento que começam em i com a janela atual. Se todas batem, adicionamos ao nosso vetor de candidatos a próxima palavra da base de conhecimento localizada em $i+$ o tamanho da janela atual.

4. Se o vetor de palavras candidatas é vazio, recomeçamos o laço retirando a primeira palavra da nossa janela.

5. Senão, iteramos sobre as palavras do nosso dicionário. Somamos todos os produtos internos dos vetores do dicionário com cada uma das palavras candidatas. Escolhemos a palavra do dicionário com maior produto interno, dando preferência a primeira que foi encontrada em caso de empate. Essa será a resposta.

Problem M

Muralhas reforçadas

O método mais fácil para resolução desse problema utiliza busca binária na resposta. Queremos buscar o menor mínimo que é possível ter como resposta. Pra testar se uma resposta candidata X é válida, basta achar o último valor no vetor que é menor que X e aplicar a operação naquela posição. Se o mínimo de todo o vetor ficar maior ou igual a X, é possível obter aquele o mínimo como resposta. Essa solução é $\mathcal{O}(N \log A)$ onde A é a resposta que pode variar de 0 a $2 \cdot 10^9$.

Pode ser resolvido também em $\mathcal{O}(N)$ através do uso de uma estrutura de fila mínima e vetores de prefixo e sufixo de mínimo. A fila mínima deve ser mantida com no máximo K elementos percorrendo pelo vetor. Para diminuir todos os elementos da fila em 1 (para replicar a operação do enunciado) basta manter uma variável de delta da fila mínima atual. Ao adicionar um elemento na fila mínima, o valor adicionado é subtraído com o delta. Ao obter um mínimo da fila mínima, adicione o delta novamente. Para decrementar todos os valores da fila, decremente o delta. Os vetores de prefixo e sufixo devem ser usados para calcular o valor da resposta na posição atual junto com o valor da fila mínima.