A. Angariando Padrões

Sejam A, B, C e D as quatro sequências de entrada. Em uma abordagem força-bruta para este problema consiste em montar todas as quádruplas (a, b, c, d) de elementos, com $a \in A$, $b \in B$, $c \in C$ e $d \in D$ de modo que o XOR bit a bit resultante seja 0. Esta abordagem possui custo de $O(N^4)$, que é inviável dado o tamanho máximo das sequências.

Podemos resolvê-lo de uma maneira mais eficiente usando a técnica de meet-in-the-middle. Primeiramente criamos dois vetores, X e Y: X possui o resultado do XOR bit a bit entre todas as duplas (a, b) enquanto Y é computado de forma similar, mas com as duplas (c, d). Após obter X e Y, ordenamos Y e para cada elemento x de X, precisamos identificar quantos elementos de Y são iguais a x. Ao ordenar Y, esta consulta pode ser respondida em tempo $O(\lg n)$ através de uma busca binária. O tempo total dispendido neste processo é $O(N^2 \lg n)$.

B. Buracos de Coruja

Podemos utilizar uma estratégia gulosa para resolver esse problema:

- Ordene os vetores das posições das corujas e das posições dos buracos;
- Considerando os vetores ordenados, atribua a i-ésima coruja ao i-ésimo buraco correspondente.
- Determine a diferença máxima entre um buraco e uma coruja.

C. Carma, Polícia!

A solução é simples, basta ler cada infração e contar sua frequência para cada procurado (em $O(n)$) e armazenar isso em uma estrutura tipo dicionário para busca posterior. A leitura da quantidade de maratonistas é simples, basta separar as palavras da frase nos espaços e encontrar a única informação que tem algum dígito (em $O(1)$). A seguir, lê-se os identificadores dos maratonistas e, caso um exista na lista de meliantes, basta ordenar os artigos em ordem reversa de ocorrências e direta de identificador do artigo, em $O(n \cdot log(n))$, e apresentar esta informação.

D. Dick Vigarista e a Corrida Maluca

O problema consiste em determinar as distâncias mínimas de Dick Vigarista para os vértices intermediários $P_2, P_3, \ldots, P_{K-1}$ da rota dos demais competidores, onde esta distância é medida em minutos.

Um primeiro ponto a ser observado é que Dick parte atrasado na corrida, de modo que o tempo de travessia de 1 a P_2 deve ser somado a todas as distâncias mínimas.

Para computar as distâncias pode-se utilizar um algoritmo de programação dinâmica baseado no algoritmo de Dijkstra. Seja $\text{dist}(u, t)$ a distância mínima de Dick ao vértice u, partindo de 1 e usando o turbo k vezes. O caso base é o início da corrida, lembrando que o atraso será somado posteriormente: $\text{dist}(1, 0) = 0$.

Assumindo que existe uma aresta entre os vértices u e v com peso w, há duas transições possíveis. Se Dick já utilizou o turbo para chegar em u, então $\text{dist}(v, 1) = \text{dist}(u, 1) + w$; caso o turbo ainda esteja disponível, então

$$\text{dist}(v, 0) = \min\{\ \text{dist}(u, 0), \text{dist}(u, 0) + w\ \}$$

e

$$\text{dist}(v, 1) = \min \left\{ \text{dist}(v, 1), \text{dist}(u, 0) + \frac{w}{2} \right\}$$

Duas observações sobre este algoritmo: em primeiro lugar, a ordem de processamento dos estados é dada pelo algoritmo de Dijkstra. Em segundo lugar, como esta divisão por dois decorrente do turbo acontece uma única vez, não é necessário armazenar as distâncias em ponto flutuante.

De posse destas distâncias mínimas, se $\text{dist}(P_i, 1) + d < \text{D}(P_i)$, onde $D(P)$ é a distância dos competidores até o ponto P, $d = D(P_2)$ é o atraso inicial de Dick e $i \in [2, P_{K-1}]$, então há um caminho que permite Dick chegar a P_i antes dos demais competidores.

Esta solução tem complexidade $O(E \log V)$.

E. Elocoin

Este problema pode ser facilmente resolvido utilizando Método de Horner, que é um algoritmo para converter um número escrito em uma base b para decimal.

Exemplo, considerando um inteiro B e $b = 2$, este pode ser reescrito em uma soma de pontências de 2 da seguinte forma:

$$B = b_{k-1} * 2^{k-1} + b_{k-2} * 2^{k-2} + \cdots + b_2 * 2^2 + b_1 * 2^1 + b_0 * 2^0$$

Em que b_i representa o dígito na posição i da representação binária de um número com k dígitos.

O Método de Horner é uma reescrita desse polinômio começando pela posição zero e então adicionando cada dígito multiplicado pela base. Fica o seguinte:

$$B = ((((((b_{k-1} * 2) + b_{k-2}) * 2 + \ldots) * 2) + b_2) * 2 + b_1) * 2 + b_0$$

A partir dessa construção, para realizar a divisão, o módulo N do número pode ser aplicado a cada operação de múltiplicação da representação.

$$B \,\%\, N = ((((((((((((b_{k-1} * 2)\%N) + b_{k-2}) * 2)\%N) + \ldots) * 2)\%N) + b_2) * 2)\%N) + b_1) * 2\%N) + b_0)\%N$$

Assim, a solução pode ser refletida no seguinte algoritmo:

```
resto = 0
for each bit in B:
    resto = (resto * 2) + bit
    resto = resto % N
if (resto != 0) {
    print("Phishing de criptomoeda.")
else {
    print("To the moon!")
}
```

F. Fermentação de Pães

O problema pode ser resolvido por meio de uma simulação simples utilizando a estrutura de dados **pilha**.

A esteira de descanso para a fermentação dos pães pode ser vista como uma pilha, uma vez que a colocação de pães recém-produzidos é feita somente pelo acesso frontal e os pães já inclusos são empurrados para trás.

Considere uma variável t para marcar a linha do tempo do processo de fermentação de pães, incluindo o momento em que são colocados e retirados da esteira. Inicializando-se $t = 1$, para cada pão $i = 1, \ldots, N$, faça:

1. Ler o tempo de fermentação do pão a_i;
2. $p_j \rightarrow$ o momento (minuto) em que o pão j que está no topo da pilha (no acesso frontal da esteira) terá seu processo de fermentação concluído;
3. Se $p == t$, retira o pão j da esteira. Se a fermentação atingiu o tempo ideal, contabilize na resposta. Repetem-se os passos 2 e 3 até que não existam mais pães prontos no tempo t.
4. Atualize $p \rightarrow$ o pão que está próximo ao acesso frontal;
5. Se $a_i + t <= p$, basta colocar o pão i na esteira, empilhando-se p_i+t;
6. Caso $a_i + t > p$, remova M pães da esteira, ordene esses pães e o pão a_i em ordem crescente do tempo em que a fermentação estará concluída e recoloque cada pão na esteira;
7. Incremente t.

Depois que todos os pães forem colocados na esteira, é hora de remover os pães da esteira. Para cada pão que está próximo do acesso frontal da esteira, verifique se ele atingiu sua fermentação ideal ou se passou do ponto. Faça isso levando em conta sempre o tempo t com os pães que são retirados da esteira.

G. Grupos de controle

O problema consiste na identificação do maior divisor ímpar $\iota(N)$ de N, o que pode ser feito por meio de recursão. Se N é ímpar, então $\iota(N) = N$; caso contrário, $\iota(N) = \iota(N/2)$.

A complexidade desta solução é $O(\log N)$.

H. Hagar.io

Este problema pode ser resolvido utilizando Conjuntos Disjuntos em uma estrutura do tipo *Union-Find* (DSU ou Disjoint Set Union), uma vez que a constituição dos grupos são modificadas dinamicamente ao longo do tempo. Essa implementação pode ser realizada da seguinte forma:

Estruturas globais:

```
grupos[N] // Inicializado cada grupo com seu próprio índice {1, 2, 3, .., N}
vida[N] // Inizializado todos os índices com 3
vitorias = 0
```

Todo grupo (g) irá possuir um pai (p) cujo a informaçao de força possuirá os pontos de vida atualizados. A função para encontrar o pai de um grupo é a seguinte:

```
find(g)
    if (g == grupos[g])
        return g
    return grupos[g] = find(grupos[g])
```

Quando dois grupos se encostam, uma função de união (*join*) é utilizada:

```
join(g1, g2)
    p1 = find(g1)
    p2 = find(g1)
    if (p1 == p2 || vida[p1] == 0 || vida[p2] == 0)
        return
    if (vida[p1] > vida[p2])
        vida[p1] += vida[p2]
        grupos[p2] = grupos[p1]
    else if (vida[p2] > vida[p1])
        vida[p2] += vida[p1]
        grupos[p1] = grupos[p2]
    else
        vida[p1] -= 1
        vida[p2] -= 1
```

Por útimo a função de ataque verifica se o jogador faz parte da equipe vencedora da ação:

```
attack(t, g1, g2)
    p1 = find(g1)
    p2 = find(g2)
    pj = find(J)
    if (vida[p1] != 0 && vida[p2] != 0 && vida[p1] != vida[p2])
        if (vida[p1] > vida[p2])
            vida[p2] = 0
            if (p1 == pj)
                vitorias++
        else if (vida[p2] > vida[p1])
            vida[p1] = 0
            if (p2 == pj)
                vitorias++
```

J. Já Apaguei

Este é um problema que envolve árvores. Podemos modelar cada núcleo i como sendo um vértice de uma árvore e contendo um consumo C_i e cada corredor como uma aresta que conecta dois vértices.

Para calcular a economia máxima obedecendo às restrições do problema, não podemos contabilizar consumos de nós adjacentes. Tome $X(i)$ como a maior economia da árvore com raiz em i enquanto ela não está ativa e $Y(i)$ definido identicamente, mas com a raiz ativa. Suponha também que $v_1, \ldots, v_k$ sejam os filhos de i. Assim, temos:

$$X(i) = C_i + \sum_{j=1}^{k}, Y(v_j)$$

$$Y(i) = \sum_{j=1}^{k} \max(X(v_j), Y(v_j))$$

Seja w um nó arbitrário, a solução, a qual pode ser computada recursivamente via busca em profundidade com *memoization*, está em $\max(X(w), Y(w))$.

Para recuperar a solução, basta percorrer a árvore em profundidade, verificando para cada nó v, se $x[v] > y[v]$ e se o seu pai, não foi incluído na solução. Em caso afirmativo, v deve estar na solução.

Tanto o preenchimento da tabela de programação dinâmica quanto a reconstrução da solução levam tempo $O(n)$, em que n é o número de nós da árvore.

K. Karen e os dados

Existem várias visualizações para este problema. Acredito que seja mais simples focar que a probabilidade é a fração $\frac{\textnormal{quantidade estados favoráveis}}{\textnormal{quantidade total de estados}}$. Assim, podemos considerar somente o problema de contagem e a soma de tudo será o total de estado.

Agora queremos encontrar a quantidade de estados possíveis com exatamente K valores distintos. Podemos contar a quantidade de formar de selecionar K valores dos M é $\binom{M}{K}$.

Falta contar a quantidade de jeitos de distribuir os valores para os dados. Como todos os dados são iguais, podemos pensar em outra representação e vamos buscar inspiração em histogramas. Podemos considerar um estado como uma tupla $(x_1, x_2, \ldots, x_k)$ onde x_i é a quantidade de dados com o i-ésimo valor. Queremos contar a quantidade de tuplas tal que $x_1 + x_2 + \ldots + x_k = N$ e $x_i \geq 1$.

Para uma restrição $x_i \geq 0$, podemos montar uma programação dinâmica simples onde escolhemos adicionar +1 no último da tupla ou não. Dessa forma 'f(sum, k) = f(sum - 1, k) + f(sum, k - 1)'. Por fim, para calcular a quantidade de estados com a restrição original $x_i \geq 1$ podemos só adicionar 1 em todos os elementos no início, usamos $f(n-k, k)$.

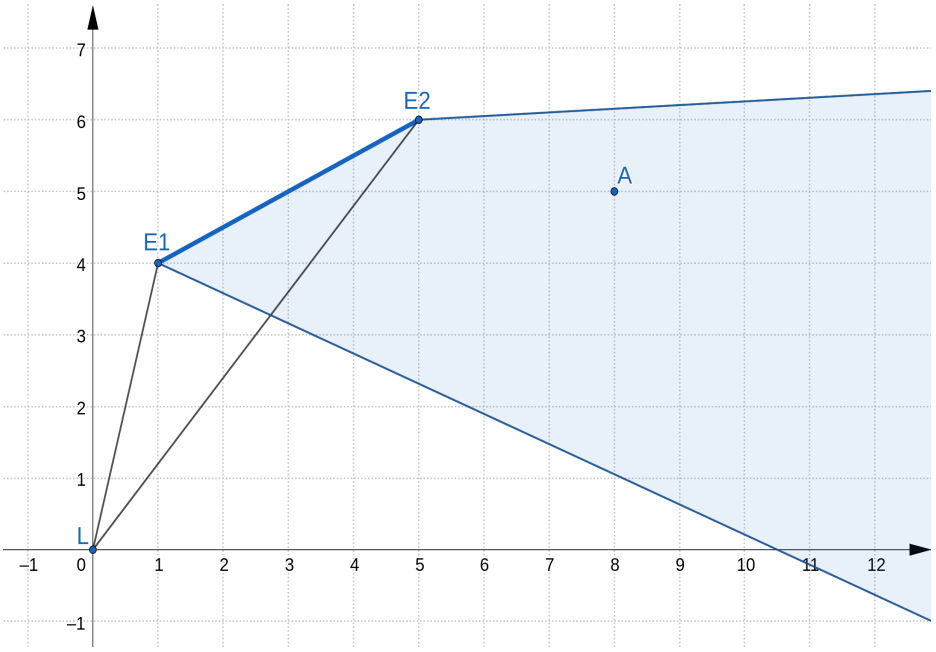
Caso o binômio seja implementado usando a relação de recorrência, a complexidade final é $O(N * M)$. Caso os fatoriais e seus inversos sejam pré-computados, a complexidade final é $O(M + N^2)$.

Outro resultado de combinatória é que $f(n, k) = \binom{n+k-1}{k-1}$. Por este fato, podemos construir uma solução em O(M).

Desafio: Você consegue resolver o problema para $M < 10^9$ e $N < 10^5$?

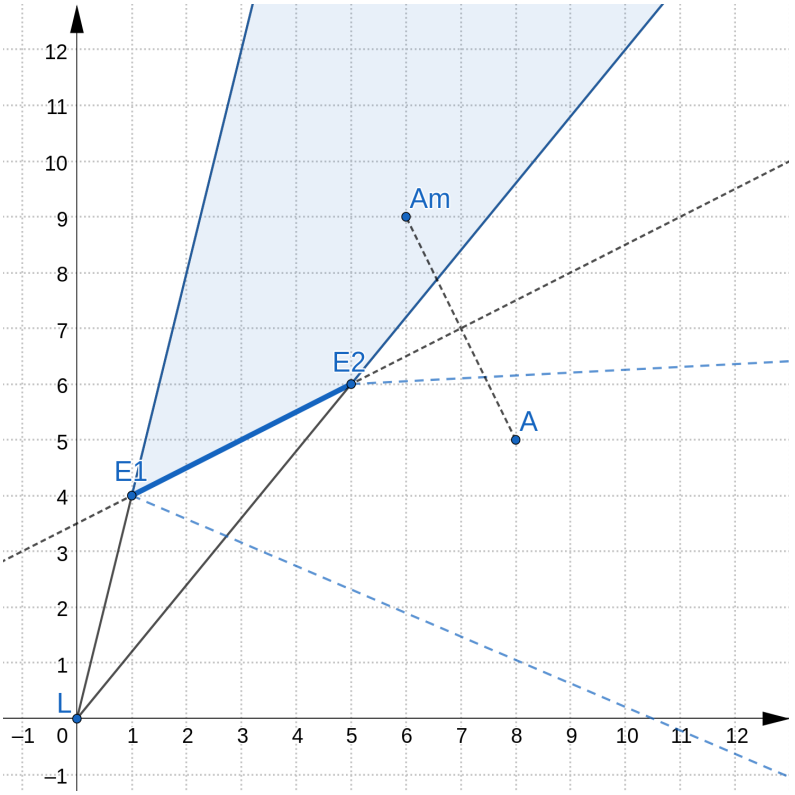
L. Laser do Léo

Existem dois limites até onde é possível acertar o laser: os pontos extremos do espelho E_1 e E_2. Todos os pontos dentro da região entre esses dois extremos podem ser alcançados pelo laser, como pode ser visto no exemplo abaixo, com a região destacada em azul:



Logo, a solução consiste em saber se o ponto A está dentro dessa região. O espelho respeita as leis da física, que regem a reflexão da trajetória do laser ao bater no espelho ($\theta_i = \theta_r$). Existem algumas formas de simular essa reflexão, é possível rotacionar a reta que representa a trajetória do laser, é possível rotacionar o espelho para que fique paralelo a um dos eixos coordenados, contudo são abordagens que exigem uma implementação mais complexa.

A solução descrita a seguir utiliza somente das operações vetoriais **Produto Vetorial** e **Produto Escalar**, juntamente com a ideia de **Imagem Virtual**: ao invés de refletir o laser, vamos gerar a imagem virtual A_m do ponto A para dentro do espelho, sendo agora possível considerar que o laser não reflete no espelho, apenas entra para dentro de sua imagem virtual, almejando agora acertar o ponto A_m.



Para gerar a imagem virtual A_m, basta espelhar/refletir o ponto A em relação a reta do espelho de forma simétrica, utilizando da operação produto escalar. A partir do produto escalar é possível encontrar a projeção do ponto A na reta do espelho, com isso, utilizamos de soma e subtração de vetores para refletir o ponto para o outro lado da reta, centrado na projeção encontrada.

Uma vez com o ponto A_m calculado, basta identificar se esse ponto está entre as retas LE_1 e LE_2, e para isso é possível utilizar da operação produto vetorial. A partir do sinal do produto vetorial é possível identificar se um ponto está para a direita, para a esquerda ou colinear com uma dada reta. Caso o ponto esteja para a esquerda/direita das duas retas (sinal dos produtos vetoriais sejam iguais), ele não está entre as duas retas e não pode ser alcançado, caso contrário (sinal dos produtos vetoriais sejam diferentes), o ponto se encontra entre as duas retas e é possível atingi-lo como o laser.

Caso queira aprender mais sobre o produto vetorial e escalar, assista as aulas da Turma Avançada disponíveis no link https://unb-cic.github.io/Maratona-Extensao/avancado/geometria/

M. Mariana e os cursos

Este problema pode interpretado como uma mochila binária: as durações dos cursos são os pesos, e os descontos são os valores. Assim, a solução tem complexidade $O(NT)$.

O. Obtendo Informações

Este problema consiste em determinar o tamanho do maior ciclo em uma permutação, o que pode ser feito em tempo $O(n)$. Para cada supervisor não marcado, simulamos o processo de consulta do estagiário até retornamos ao original enquanto uma variável acumuladora é atualizada. Sempre que um supervisor é visto ele é marcado e sempre guardamos a melhor resposta em relação à variável acumuladora.

Em código, temos:

```
int max_step = 1;
vector<bool> used(n, false);
for (int i = 0; i < n; i++) {
    if (!used[i]) {
        int step = 0;
        int next = i;
        do {
            used[next] = true;
            step++;
            next = pi[next];
        } while (next != i);
        max_step = max(max_step, step);
    }
}
cout « max_step « endl;
```