

Problem A. Soma de ímpares

Provaremos por indução que $S_n = 1 + 3 + 5 \dots + 2n - 1 = n^2$.

Caso base: $n = 1, S_1 = 1 = 1^2$.

Suponha que $S_n = n^2$. $S_{n+1} = 1 + 3 + \dots + 2n + 1 = (1 + 3 + \dots + 2n - 1) + 2n + 1 = S_n + 2n + 1$.

Pela hipótese de indução, $S_{n+1} = n^2 + 2n + 1 = (n + 1)^2$. Qed.

Complexidade: $O(1)$.

Problem B. Potato War 1

No dia d , temos que Techno terá $X_1 + d \cdot Y_1$ batatas e Squid terá $X_2 + d \cdot Y_2$.

Dado que $X_1, Y_1, X_2, Y_2 \leq 1000$, é possível notar que ou Techno ganha em algum dia $d \leq 1001$ ou então ele nunca ganha, o que só acontece se $Y_1 \leq Y_2$ e $X_1 \leq X_2$.

Dessa forma, podemos resolver com um laço para checar para todo dia de 0 a 1001.

Outra opção é fazer uma conta simples para ver que queremos o primeiro t inteiro tal que:

$$t \cdot Y_1 + X_1 > t \cdot Y_2 + X_2 \implies t \cdot (Y_1 - Y_2) > X_2 - X_1 \implies t > \frac{X_2 - X_1}{Y_1 - Y_2} \quad (1)$$

Devemos tomar cuidado com os casos $X_1 > X_2$, em que Technoblade no tempo $t = 0$ e no caso em que $X_1 \leq X_2$ e $Y_1 \leq Y_2$ em que Technoblade nunca ganha.

Complexidade: $O(1)$.

Problem C. Vendas

Primeiro notamos que se $(v_i + v_{i+1} \dots + v_{i+k-1})/k$ é a menor média de k consecutivos, então $(v_i + v_{i+1} \dots + v_{i+k-1})$ tem a menor soma para k elementos consecutivos.

Dessa forma, ignoramos a média e focamos na soma de k consecutivos.

Podemos resolver esse problema com uma técnica chamada "janela deslizante". Primeiro computamos a soma que começa em $i = 1$. Para os próximos intervalos, $i = 2, i = 3 \dots$, notamos que de um intervalo para o outro há apenas a soma de um elemento no final e a subtração de um elemento no início.

Por exemplo, inicialmente temos a soma: $v_1 + v_2 + \dots + v_k = S_1$.

Para $i = 2$, a soma será $v_2 + v_3 + \dots + v_{k+1}$ que é $S_1 - v_1 + v_{k+1}$.

Tendo as somas, podemos ver qual a menor e a maior.

Complexidade: $O(N)$.

Problem D. Skywars

Primeiro, suponha que tivéssemos apenas Techno (T) e um oponente (*). Assim, o problema se reduz a achar o menor caminho em um grid, sendo que custa 1 para entrar uma célula vazia (.), e 0 caso contrário. Podemos resolver isso usando o Algoritmo de Dijkstra em $O(NM \log(NM))$ ou então com uma BFS 0-1 (tutorial disponível em <https://codeforces.com/blog/entry/22276>) em $O(NM)$.

Porém temos 2 inimigos, o que complica um pouco. Notamos primeiro que toda solução é um caminho de Techno para cada um dos inimigos, pois eles tem que estar conectados. O problema é que pode haver interseções. Note que se a união dos dois caminhos não for uma árvore, então há um ciclo, o que é sub-ótimo, pois não conecta nada a mais e não tem custo menor. Portanto, essa união forma uma árvore com no máximo 3 folhas. Toda árvore desse estilo, tem um nó pivotal; uma espécie de centro tal que se somarmos a distância desse nó às raízes teremos passado por toda aresta exatamente 1 vez.

Deste modo, a solução consiste em testar todo nó como pivo e somar a distância para cada um dos 3 jogadores: technoblade e os inimigos. Tome cuidado especial para o caso que o pivo é um ponto vazio, para não

contar repetido.

Para aqueles interessados em otimização combinatória, esse é um caso específico do problema "Steiner Tree um problema NP completo pro caso geral, e com algumas soluções interessantes para casos específicos. Uma solução geral está disponível aqui: <https://github.com/tdas0/lib/blob/master/library/misc/SteinerTree.cpp>, e para ler mais sobre o problema acesse https://en.wikipedia.org/wiki/Steiner_tree_problem.

Complexidade: $O(N \cdot M)$.

Problem E. Potato War 2

Tópicos: DP em dígito, Knapsack 0-1

Por enquanto assuma que não tem a restrição no número de pacotes da primeira loja.

Começamos simplificando a condição de que podemos pegar qualquer quantidade de pacotes por loja. Digamos que tem uma loja com b batatas por pacote.

Podemos pegar 0, b , $2b$... batatas. Agora considere que invés disso temos 60 lojas, a primeira vende pacotes de b batatas, a segunda de $2b$, a terceira $4b$... ou seja, a i -ésima vende em pacotes de $2^{i-1} \cdot b$ batatas por pacote. Mas nessas 60 lojas, você tem a opção de comprar 0 ou 1 pacotes. Note que para todo k ainda é possível pegar $k \cdot b$ batatas de exatamente 1 jeito, pois cada loja é basicamente um bit de k . Dessa forma, simplificamos nosso problema de knapsack com infinitos itens para uma knapsack 0-1, ou pega 0 pacotes ou 1 pacote.

Nesse ponto é importante que você saiba o que é Programação Dinâmica em Dígito: <https://codeforces.com/blog/entry/84928>

Vamos montar o problema como se fosse uma knapsack normal, com os seguintes estados, os quais vamos otimizar mais pra frente:

- Soma da quantidade de batatas pegadas até agora: S
- Loja atual: i
- Bit atual: bit . Esse é o estado mais complicado. Basicamente ele quer dizer que estamos na versão da loja i que tem $2^{bit} \cdot b_i$ batatas por pacote. Aqui temos a opção de pegar 0 ou 1 pacotes.

Dessa forma, temos:

$$dp(S, i, bit) = dp(S + 2^{bit} \cdot b_i, i + 1, bit) + dp(S, i + 1, bit) \text{ para } i \leq n \text{ e } dp(S, n + 1, bit) = dp(S, 1, bit + 1).$$

Os casos base são $dp(B, 1, 60) = 1$ e $dp(x, 1, 60) = 0$, para $x \neq B$, e a resposta é $dp(0, 1, 0)$.

OK, isso funciona pois estamos fazendo uma knapsack 0-1 passando por cada versão binarizada das lojas uma vez. O problema é que isso é muito lento pois S é MUITO grande.

Aqui vem a sacada legal. Primeiro, note que todo item da forma $2^{bit} \cdot b_i$ tem os últimos bit bits iguais a 0. Dessa forma, quando vamos de $dp(S, n + 1, bit)$ para $dp(S, 1, bit + 1)$ podemos checar se o bit 2^{bit} de S está correto, pois caso não esteja ele vai continuar errado e teremos que o valor dessa DP com certeza é 0. Isso corta um pouco do número de estados mas ainda podemos fazer melhor.

Ao ir de bit a $bit + 1$, podemos na verdade descartar o último bit de S , pois ele não dá nenhuma informação relevante mais pra frente, de forma que teremos:

$$dp(S, i, bit) = dp(S + b_i, i + 1, bit) + dp(S, i + 1, bit) \text{ para } i \leq n \text{ e } dp(S, n + 1, bit) = dp(\text{floor}(\frac{S}{2}), 1, bit + 1) \text{ se o bit 0 de } S \text{ estiver correto e 0 caso contrário.}$$

Note que agora não multiplicamos b_i por 2^{bit} , **invés disso dividimos S por 2!**

A última observação, é que $S \leq 2 \cdot \sum b_i \leq 1000$, pois no pior caso temos $S = 500 + 500/2 + 500/4 \dots \leq 1000$.

Para a restrição da primeira loja, basta colocar uma flag a mais como estado que diz se:

- A representação binária está maior que a de t
- A representação binária está igual a de t
- A representação binária está menor que a de t

Fica como exercício reformular as transições com essa nova flag.

Complexidade final: $O(N \cdot \log B \cdot \sum b_i)$.

Problem F. Férias

Tópicos: Árvore geradora mínima (também conhecida como MST)

Seja OPT o custo de uma férias ótima. É fácil ver que achar OPT é NP-completo, pois o problema do ciclo Hamiltoniano, pode se reduzir a nossa questão, no sentido que ao achar a resposta exata para essa questão em tempo polinomial, teríamos resolvido o problema do ciclo Hamiltoniano também.

Por sorte, queremos uma resposta com custo V tal que $V \leq 2 \cdot OPT$. É útil pensar em uma estimativa por baixo de OPT . Por exemplo, dado o custo da árvore geradora mínima (MST) $c(MST)$, é fácil ver que $c(MST) \leq OPT$, pois podemos obter uma árvore pegando as arestas do caminho de OPT e removendo redundâncias até ter uma árvore, a qual terá custo menor que OPT .

O que é ainda melhor, é que com uma MST é possível visitar todos os nós passando por cada aresta exatamente 2 vezes com a seguinte estratégia:

Primeiro escolhemos uma raiz para a árvore, digamos o vértice 1. Depois fazemos o seguinte procedimento de forma recursiva, chamado de *dfs* aqui:

- adicione v para a resposta
- para todo filho to de v chame *dfs*(to) e depois adicione v para a resposta. Isso é feito para manter a invariante que sempre estamos em v no final e em cada etapa desse loop

Dessa forma, toda aresta da MST é usada exatamente duas vezes: primeiro do pai para o filho e depois no sentido contrário.

Como um bônus, é possível fazer ainda melhor, com um custo $V \leq \frac{3 \cdot OPT}{2}$, mas em $O(N^3)$. Para mais detalhes vide https://en.wikipedia.org/wiki/Christofides_algorithm. A ideia é que ao duplicar a árvore estamos criando um grafo conexo e euleriano. Porém, uma estratégia melhor seria pegar a MST e fazer um emparelhamento de custo mínimo apenas dos vértices de grau ímpar, de modo que criamos um novo grafo conexo, euleriano e com peso menor. Temos que adaptar para o nosso contexto, pois o grafo não é completo, mas basta adicionar uma aresta de i para j com custo $dist_{i,j}$ para todo par.

Complexidade: $O(M \cdot \log M + N)$

Problem G. Beautiful Crown

Tópicos: Burn Side Lemma; Euler Totiente Function; Crivo de Eratóteles;

Pelo Lema de Burnside, temos que a resposta é:

$$\sum_{n=1}^N \sum_{k=0}^{n-1} \frac{m^{\gcd(k,n)}}{n}$$

Vamos simplificar o somatório intermediário: $S_n = \sum_{k=0}^{n-1} \frac{m^{\gcd(k,n)}}{n}$.

Para isso, vamos tentar agrupar o somatório vendo quantas vezes $\gcd(k,n) = d$ para todo d . Seja $f(n,d)$ esse valor o somatório seria:

$$S_n = \sum_{d=0}^{\infty} \frac{f(n,d)m^d}{n}.$$

Primeiro notamos que como $\gcd(k,n) = d$, então d deve ser divisor de n . Assim:

$$S_n = \sum_{d|n} \frac{f(n,d)m^d}{n}$$

Só falta achar $f(n, d)$. Tem alguns modos de fazer isso. Um deles é usar a função totiente de Euler.

$\phi(n)$ que conta quantos números inteiros de 1 a n são coprimos a n .

Note que $f(n, d) = \phi(n/d)$, pois $\gcd(k, n) = d \implies \gcd(\frac{k}{d}, \frac{n}{d}) = 1$.

Sendo assim, conseguimos achar S_n em $O(\sigma(n))$, com $\sigma(n)$ sendo o número de divisores de n . Porém, no total a complexidade é $O(N \log N)$ caso passemos pelos divisores ou fatorando o número (tendo o menor fator primo de todo número precalculado) ou com uma espécie de crivo. A prova fica como exercício pro leitor :)

Dica: Soma harmônica

Problem H. Divisão de Times

Essa era para ser uma das 3 questões mais difíceis. Créditos da questão vão à Arthur Lobo.

A primeira observação é que se $k > \sqrt{n}$ então a resposta é no máximo $\sqrt{n}$. Desse modo, a resposta só pode ocupar $2\sqrt{n}$ valores distintos, pois ou $k \leq \sqrt{n}$ e só há $\sqrt{n}$ modos ou então ele é maior e temos mais $\sqrt{n}$ formas.

A segunda observação é que conseguimos resolver para um k fixo em $O(n)$. Para fazer isso primeiro usamos dois ponteiros para achar para todo i qual o tamanho do maior intervalo que começa em i . Vamos chamar de $f(i)$ o maior índice tal que $[i, f(i))$ é válido, tomando cuidado para o fato do array ser cíclico. Além disso, se o array não fosse cíclico, temos que a resposta gulosa é a correta, então podemos achar o número de intervalos fazendo $1 \rightarrow f(1) \rightarrow f(f(1)) \dots$ até completar todo mundo.

Pegamos o i tal que esse intervalo $[i, f(i))$ é o menor possível e seja x esse tamanho. Vamos testar para todo l em $[i, f(i))$, agora incluso, como sendo o primeiro elemento do grupo, fingindo que o array não é mais cíclico. Note que a resposta é menor ou igual a $\frac{n}{x}$, pois x é o menor pulo possível. Dessa forma, vamos testar $x + 1$ inícios, cada um custando $\frac{n}{x}$, para um total de $O(n)$. Note que como $f(i) \leq f(i + 1)$, não é possível que ninguém em $[i, f(i))$ comece um novo intervalo. Por exemplo, se começarmos em $i - 1$, é garantido que $f(i - 1) \leq f(i)$, então o próximo intervalo começa em $f(i - 1) \in [i, f(i))$.

Uma solução alternativa para essa parte, e bem surpreendente, é:

Seja $f^k(i)$ o resultado de aplicar k vezes a função f : $f(f(f \dots f(i) \dots))$. Se testar como início $f^n(1)$ iremos obter a resposta! A prova disso fica para o leitor. Algumas observações que podem ajudar são:

- $f(i) \leq f(i + 1)$
- Se a resposta é OPT , então para qualquer início teremos OPT ou $OPT + 1$ grupos.
- Ao entrar em vértice no caminho ótimo, nunca saímos do valor ótimo.
- Fixe uma resposta ótima $v_1, v_2, v_3 \dots v_{OPT}$. Note que se $v_j < i < v_{j+1}$ então $f(v_j) \leq f_i \leq f(v_{j+1}) \implies v_{j+1} \leq f_i \leq v_{j+2}$.

Por fim, usamos um divide-and-conquer ou uma busca binária para conseguir encontrar os valores para todo k . Para a busca binária queremos achar o último índice r tal que a resposta para $k = r$ é igual a para $k = l$.

Para a busca binária é necessário algumas otimizações para diminuir a constante, como por exemplo usar memorização para não calcular o valor da resposta para um mesmo k mais de uma vez. Tive que ser um tanto rígido nesse quesito, para barrar solução em $O(n \log n)$ para k fixo usando binary lifting, apesar de que se a constante for muito boa talvez isso passe.

Complexidade: $O(n\sqrt{n} \log n)$.

Problem I. Username

Os limites e o tempo limite dão bastante liberdade para o usuário escolher como resolver

esse problema. Uma solução simples é usar busca binária somado com hash polinomial: <https://github.com/tdas0/lib/blob/master/library/string/ hashing.cpp>.

Primeiro notamos que se uma substring de tamanho k de D não aparece em G , $k < |D|$, então existe uma substring de tamanho $k + 1$ que também não ocorre em G . Assim, podemos usar a busca binária.

Para checar se toda substring de D de tamanho k ocorre em G , usaremos hash, guardando em um conjunto todos os valores desses hashes. Por fim, checamos se nenhum substring de G aparece nesse conjunto.

Seja $n = |D| + |G|$, onde $|D|$ é tamanho da string D .

Complexidade: $O(n \log^2 n)$

Bônus: É possível resolver com Suffix Array em $O(n)$. Concatenamos $D + \# + G$ e para todo sufixo de D procuramos o sufixo de G mais próximo no suffix array. Utilizando o LCP conseguimos encontrar para cada sufixo de D qual o menor tamanho desse sufixo que não ocorre em G .

Bônus 2, criando casos de teste: Criar casos de teste aleatórios é uma péssima ideia nesse problema, pois se não a resposta é muito pequena, em torno de 14 para os limites da questão. Uma ideia seria gerar D aleatoriamente e fazer $G = D[1, |D|] + D[2, |D|]$. Por exemplo, se $D = abc$, $G = abbc$, de modo que a menor substring é o próprio D ! É possível usar ideias similares para criar casos em que a resposta é um pouco menos que isso também.

Problem J. The Final Reckoning

Primeiro, note que "Technoblade never dies então já podemos imprimir isso de cara (de fato isso sempre ocorre, pois o 1 jogador pode pegar todas as moedas que estão em posições ímpares ou todas que estão em posições pares, e isso garante pelo menos metade das moedas no pior caso).

A solução é usar uma programação dinâmica. Seja $dp[l][r]$ o valor do maior ****lucro**** sendo que as moedas são m_l até m_r .

Para Techno, que joga de maneira ótima, $dp[l][r] = \max(m_l - dp[l+1][r], m_r - dp[l][r-1])$. Para Skeppy, $dp[l][r] = m_l - dp[l+1][r]$ se $m_l > m_r$ ou $dp[l][r] = m_r - dp[l][r-1]$ caso contrário. Note que isso de ir subtraindo a próxima dp coloca as moedas de techno com sinal positivo e as que Skeppy pegou com sinal negativo, conforme desejado.

Por fim, usaremos um guloso para conseguir a menor resposta lexicograficamente. Se a quantidade de moedas que temos nos permite pegar da esquerda sem perder pegamos dali. Usamos o valor de dp para checar se é possível continuar sempre ganhando.

Complexidade: $O(n^2)$

Problem K. Otimismo

Seja O_i o valor otimista da posição i e v_i o valor atual.

Vamos guardar simultaneamente dois valores: $D_i = O_i - v_i$ e v_i . Para suportar as mudanças em v_i de forma rápida podemos utilizar uma segment tree com Lazy, mas e para D_i ?

Note que se v_i vira $v_i + x$, D_i vira $O_i - (v_i + x) = (O_i - v_i) - x = D_i - x$, caso o valor de O_i não mude. Agora se O_i muda, é porque $D_i - x < 0$. Portanto, a mudança em D_i pode ser expressa por $D_i := \max(0, D_i - x)$.

Para suportar essa operação usaremos uma segmeent tree beats: <https://github.com/brunomaletta/Biblioteca/blob/master/Codigo/Estruturas/Segtree/segTreeBeats.cpp> para código e <https://codeforces.com/blog/entry/57319> para mais detalhes.

Complexidade: $O(n \log^2 n)$

Problem L. Experimento de F129

Vamos brutar o intervalo com o menor r_i que fara parte da nossa resposta. Para isso, ordenamos os intervalos por r_i . Agora temos aqueles intervalos com $l_j > r_i$ que são descartados bem como os intervalos com $r_j < r_i$, pois definimos que r_i é o menor r .

Todos os outros intervalos vão ter interseção, pois $l_j \leq r_i \leq r_j$. Portanto, vamos fazer para cada intervalo 2 eventos: em l_i adicionamos 1 para um contador e em r_i subtraímos 1. Com um line-sweep conseguimos manter quantos segmentos estão ativos. Similar a esse problema em que queremos colorir os intervalos com mínimo de cores:
<https://github.com/brunomaletta/Biblioteca/blob/master/Codigo/Problemas/intervalGraphColoring.cpp>

Complexidade: $O(n \log n)$