Contest D2 - 19/02/2016

Solution Sketches

Renzo Gonzalo Gómez Diaz

Disclaimer *Esta é uma analise não-oficial de alguns problemas de juízes "online". Qualquer erro no seguinte texto é minha culpa. Se você encontra algum erro ou sabe como resolver algum desses problemas de forma diferente, estarei muito feliz de saber :D. Pode enviar um e-mail a* gomez.renzo@gmail.com *ou meu blog. ANTES DE LER A SOLUÇÃO A CERTO PROBLEMA, TENTE-LO MAIS UMA VEZ.*

Tabela 1: *Problemas*

OJ ID	Nome
UVa 882	The Mailbox Manufacturers Problem
Live Archive 4367	Think I'll Buy Me a Football Team
SPOJ INTERVAL	Intervals
SPOJ ESJAIL	Escape from Jail
Live Archive 4704	Stringer
Live Archive 4970	Enter The Dragon

A. The Mailbox Manufactureres Problem

Denotemos por $dp[k][i][j]$ o número mínimo de foguetes necessários para encontrar, no pior caso, quanto pode aguentar uma caixa postal, se temos k caixas disponíveis, e sabemos que o número de foguetes está entre i e j. Então, pelo enunciado do problema sabemos que

$$dp[1][i][j] \quad = \quad \frac{j(j+1)}{2} - \frac{i(i-1)}{2}.$$

Notemos que, quando testamos um valor $i < \ell < j$ de foguetes, podem acontecer dois eventos: a caixa quebra ou não quebra. Se ela quebrar, então usamos $\ell + dp[k-1][i][\ell-1]$ foguetes. Caso contrário, usamos $\ell + dp[k][\ell+1][j]$ foguetes. Como estamos procurando esse número no pior caso, devemos considerar a maior dessas duas quantidades. Assim, chegamos à seguinte recorrência.

$$dp[k][i][j] = \begin{cases} i, & \text{se } i = j, \\ \min\{\max\{dp[k-1][i][\ell-1], dp[k][\ell+1][j]\} + \ell : i \leqslant \ell \leqslant j\}, & \text{caso contrário.} \end{cases}$$

Na expressão anterior, se $\ell = i$ consideramos que

$$\max\{dp[k-1][i][\ell-1], dp[k][\ell+1][j]\} = dp[k][\ell+1][j].$$

De forma similar, se $\ell = j$,

$$\max\{dp[k-1][i][\ell-1], dp[k][\ell+1][j]\} = dp[k-1][i][\ell-1].$$

Segue uma implementação dessa ideia no seguinte link.

B. Think I will Buy Me a Football Team

Seja V o conjunto de todos os bancos. Seja $f(u, v)$ o monto de dinheiro que deve pagar o banco u para o banco v. Então, para cada banco u, denotemos por

$$d(u) = \sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u).$$

Note que, se $d(u) > 0$, então o banco u deve pagar mais dinheiro do que recebe e, vice-versa quando $d(u) < 0$. Considere os conjuntos $A = \{v \in V : d(v) > 0\}$, $B = \{v \in V : d(v) < 0\}$ e $C = V \backslash (A \cup B)$. Pelas restrições do problema sabemos que

$$\sum_{v \in A} d(v) = \sum_{v \in B} d(v) = S.$$

Além disso, o monto mínimo de transferência entre os bancos não pode ser menor do que S. Agora, vamos mostrar que sempre podemos achar uma solução onde é transferido S de dinheiro. Para isso, criamos uma rede de fluxo com conjunto de vértices V, cada vértice com demanda $d(u)$, arcos entre todo par de vértice $u, v \in V$ de capacidade inferior $f(u, v)$ e superior $+\infty$. Como toda capacidade superior é $+\infty$ e toda capacidade inferior é menor do que $+\infty$ sabemos que esse problema tem um fluxo viável que satisfaz essas restrições (Teorema de Hoffman). Qualquer fluxo nesse grafo pode ser decomposto em caminhos aumentadores. Um caminho aumentador P nessa rede deve ser da forma $(a, \ldots, b)$, onde $a \in A$ e $b \in B$. Além disso, cada caminho nos indica como fazer cada transferência. Basicamente, se a capacidade de P é x, indica que devemos transferir x de a para b.

Então, é suficiente calcular $d(u)$ para cada banco e imprimir $\sum_{v \in A} d(v)$. A ideia deste problema era usar a intuição sem necessidade de mostrar que sempre existe essa solução mínima. Um exemplo de implementação desse algoritmo segue no seguinte link.

C. Intervals

Se consideramos $c_i = 1$ para $i = 1, 2, \ldots, n$, este problema é conhecido e admite a seguinte solução gulosa.

Algoritmo COVER-INTERVALS(I, n)

```
1 Seja I o vetor de intervalos I_i = [a_i, b_i].
2 Ordenar I de forma não-decrescente segundo b_i.
3 C ← ∅
4 para i ← 1 até n faça
5         se C ∩ I_i = ∅ então
6                 C ← C ∪ {b_i}
7 devolva C
```

Para mostrar que esse algoritmo devolve o número mínimo de elementos que são necessários para "cobrir" (intersectar) todos os intervalos, vamos fazer uma prova por contradição. Suponha que existe uma solução ótima que tem cardinalidade menor a C (solução gulosa). Dentre todas as soluções ótimas possíveis tome uma solução C^* com o maior prefixo em comum com C. Suponha que C e C^* estão ordenados de forma crescente.

Observação: Dados conjuntos $C = \{x_1, x_2, \ldots, x_k\}$ e $C^* = \{y_1, y_2, \ldots, y_m\}$, o **maior prefixo comum** é o máximo índice j, $0 \leqslant j \leqslant \min\{k, m\}$, tal que $x_i = y_i$ para $i = 1, \ldots, j$ e $x_{j+1} \neq y_{j+1}$ (se $j = 0$, então $x_1 \neq y_1$).

Seja j o tamanho do prefixo comum entre C e C^*. Seja I_r o primeiro intervalo que não é coberto pelo conjunto $\{x_1, \ldots, x_j\}$. Pela forma como é feita a escolha gulosa temos que $x_{j+1} = b_r$. Além disso, sabemos que y_{j+1} também deve cobrir I_r. Logo, $a_r \leqslant y_{j+1} < b_r$, já que $y_{j+1} \neq x_{j+1}$. Notamos que $C' = C^* - \{y_{j+1}\} + \{x_{j+1}\}$ também é uma solução ótima, já que todo intervalo que era coberto por y_{j+1} também é coberto por x_{j+1} em C^*. Porém, chegamos a uma contradição com a escolha de C^*, já que C' tem um prefixo comum com C maior do que C^*.

Agora, para resolver o problema inicial podemos usar um algoritmo similar ao caso anterior. A seguir, mostramos um algoritmo que resolve o problema.

Algoritmo COVER-INTERVALS2(I, c, n)

```
1 Seja I o vetor de intervalos I_i = [a_i, b_i] e c o vetor de restrições associado.
2 Ordenar I de forma não-decrescente segundo b_i.
3 C ← ∅
4 para i ← 1 até n faça
5       se |C ∩ I_i| < c_i então
6             k ← c_i − |C ∩ I_i|
7             Seja J_i ⊆ I_i o conjunto de inteiros "mais à direita" disjunto de C tal que
              |J| = k.
8             C ← C ∪ J_i
9 devolva C
```

Com um argumento semelhante ao caso anterior podemos mostrar que esse algoritmo está correto. Na parte da implementação, precisamos

1. Encontrar a cardinalidade da interseção de $C \cap I_i$.
2. Percorrer de direita para esquerda os elementos "livres" de I_i.

Para a parte 1 podemos usar uma BIT ou Segment Tree e para a parte 2 podemos usar union-find. Outra forma de implementar essa ideia é manter um vetor de intervalos "livres" e "usados" junto com um vetor de somas parciais (veja uma implementação dessa ideia no seguinte link).

D. Escape from Jail

Quando fazemos uma busca no grafo dado pelo enunciado, devemos ter cuidado de marcar que posições que precisam de uma chave (que ainda não temos) podemos atingir. Para dessa forma, recomeçar uma busca a partir dessas posições quando tenhamos a chave necessária. Se em algum momento chegamos à posição $N-1$, então o problema tem solução. Caso contrário, não tem solução. Veja uma implementação dessa ideia no seguinte link).

E. Stringer

Em primeiro lugar, seja f_i o número de ocorrências da i-esima letra do alfabeto na palavra que queremos construir. Além disso, seja $S = \sum_{i=1}^{N} f_i$, o tamanho dessa palavra. Notemos que o número de palavras totais que podemos gerar com $f_1, f_2, \ldots, f_N$ é (veja mais sobre o número multinomial)

$$m = \binom{S}{f_1, f_2, \ldots, f_N}.$$

Agora, façamos umas contas. Por exemplo, o número de palavras que começam com a é

$$x_1 = \binom{S-1}{(f_1-1), f_2, \ldots, f_N}.$$

Note que, se $x_1 > K$, então temos certeza que a palavra buscada começa com a. Neste caso, caímos numa instância menor do mesmo problema, queremos gerar a palavra na posição K com $S-1$ letras e frequências $f_1-1, f_2, \ldots, f_N$. No caso em que $x_1 \leqslant K$, temos certeza que a palavra que buscamos não começa com a. Além disso, caímos no problema de encontrar a palavra na posição $K - x_1$, que começa com uma letra maior do que a e com as frequências dadas. Isso já nos dá um algoritmo para gerar, letra por letra, a palavra na posição K

Agora, para calcular o número multinomial, podemos guardar numa matriz os expoentes de uma fatoração em primos do número $x!$ para $x = 1, \ldots, 50$. Uma implementação dessa ideia segue no seguinte link).

F. Enter The Dragon

Para cada $t_i \neq 0$, seja $f(i) = \max\{1 \leqslant j < i : t_j = t_i\}$. Então, o problema se reduz a achar para cada $t_i \neq 0$, um $t_k = 0$ tal que $f(i) < k < i$. Este problema pode ser resolvido por um algoritmo guloso que a cada passo escolhe um $t_k = 0$ tal que k é o menor possível. Logo, usando um `set` podemos achar um tal índice a cada iteração, caso não existir, o problema não tem solução. Também podemos fazer essa busca usando "union find". Uma implementação dessa ideia segue no seguinte link).