Contest D1 - 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 10202	Pairsumonious Numbers
SPOJ INTERVAL	Intervals
Live Archive 4970	Enter The Dragon
Live Archive 3132	Minimax Triangulation
Live Archive 4291	Sculpture
Live Archive 4390	Counting Heaps

A. Pairsumonious Numbers

Seja $S = (s_1, s_2, \ldots, s_k)$ a lista de somas dadas na entrada tal que $k = N(N-1)/2$. Além disso, suponha que $s_i \leqslant s_{i+1}$ para $i = 1, 2, \ldots, k-1$. Sejam $a_1, a_2, \ldots, a_N$ os números que geram S ordenados de forma não-decrescente. Agora, suponha que você conhece a_1. Como $s_1 = a_1 + a_2$, podemos saber o valor de a_2. Da mesma forma, $s_2 = a_1 + a_3$ e, portanto, podemos descobrir a_3. Agora, $s_3 = a_1 + a_4$ ou $s_3 = a_2 + a_3$. Como conhecemos a_1, a_2 e a_3 podemos obter o valor de a_4. Assim, podemos continuar até descobrir o valor de todos os elementos. Claro, todo depende de saber o valor de a_1. Notemos que

$$a_1 = \frac{(a_1 + a_2) + (a_2 + a_3) - (a_2 + a_3)}{2}.$$

Como $s_1 = a_1 + a_2$, $s_2 = a_1 + a_3$ e $a_2 + a_3$ é algum dos elementos de S, podemos fixar $s_j = a_2 + a_3$, $j \geqslant 3$, e testar se essa escolha gera todo o conjunto S. Para testar se uma certa escolha de $a_2 + a_3$ gera o conjunto S podemos começar com uma cópia do conjunto S, digamos T. Primeiro, descobrimos os valores de a_1, a_2 e a_3. Depois, eliminamos $a_1 + a_2$, $a_1 + a_3$ e $a_2 + a_3$ de T. Agora, sabemos que o menor elemento de T é $a_1 + a_4$. Então, descobrimos o valor de a_4, e eliminamos de T as somas $a_i + a_4$, para $i = 1, \ldots, 3$. De novo sabemos que o menor elemento de T deve ser da forma $a_1 + a_5$, e repetimos o processo até

descobrir todos os números, ou até que o conjunto T não tenha um elemento $a_i + a_j$ que desejamos eliminar. Neste último caso, sabemos que a escolha inicial está errada. Veja no seguinte link uma implementação desse algoritmo.

B. 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).

C. 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).

D. Minimax Triangulation

Seja P o polígono dado pelo enunciado. Denotamos por $P[i \dots j]$ o polígono que resulta de considerar os pontos $P_i, \dots P_{i+1}, \dots, P_j$ (em sentido anti-horário) de P. Para resolver este problema podemos processa subpolígonos em ordem não-decrescente segundo o tamanho. Em primeiro lugar, suponha que temos calculado todas as diagonais do polígono. Neste caso, vamos considerar que as arestas do polígono também são diagonais. Uma forma fácil de fazer isso, leva tempo $O(n^3)$ (veja [] Cap. 1). Seja $D(P)$ o conjunto de diagonais de P. Considere

$$dp[i][j] = \text{máxima área de uma triangulação mínima de } P[i \dots j].$$

Então, $dp[i][i] = dp[i][i+1] = 0$ para $i = 0, 1, \dots, n-1$. Suponha que $k > i+1$. Se P_iP_k não é uma diagonal, então consideramos que $dp[i][k] = +\infty$. Caso contrário, consideramos que

$$dp[i][k] = \min\{\max\{dp[i][j], dp[j][k], \text{área}(P_i, P_j, P_k) : i < j < k, P_iP_j \text{ e } P_jP_k \in D(P)\}\}.$$

A resposta que queremos é $\min\{dp[i][i+n-1] : 0 \leqslant i < n\}$, onde $i+n-1$ é tomado modulo n. Para evitar o modulo podemos duplicar o vetor. Essa recorrência pode ser avaliada em $O(n^3)$. Segue uma implementação dessa ideia no seguinte link. Além disso, veja no seguinte link uma solução que não calcula as diagonais.

E. Sculpture

Em primeiro lugar, notemos que $n \leqslant 50$. Logo, em cada eixo temos no máximo 100 pontos distintos. Então, começamos fazendo compressão das coordenadas das dimensões de cada caixa. Para obter um algoritmo $O(n^4)$, primeiro, marcamos as posições correspondentes às

caixas e depois fazemos um "flood-fill" desde a parte externa para determinar as posições que ficam fora da escultura. Finalmente, usando essa informação podemos determinar a área e o volume pedido. Além disso, devemos ter cuidado ao calcular a área e o volume usando as coordenadas reais de cada posição. Para melhorar a complexidade da solução podemos, ao invés de marcar todas as posições correspondentes a cada caixa, só marcar as "bordas" de cada caixa. Com isso obtemos um algoritmo $O(n^3)$. Veja no seguinte seguinte link uma implementação do algoritmo $O(n^4)$.

F. Counting heaps

Uma forma de resolver este problema é achar uma recorrência que represente a contagem pedida no enunciado. Seja T uma árvore de n vértices. Denotemos por $H(T, n)$ o número de formas de rotular os vértices de T usando n números distintos, de forma a satisfazer a "heap property". Notemos que para satisfazer essa propriedade a raiz de T deve ser rotulada sempre com o menor elemento do conjunto. Seja r a raiz de T. Notamos que, depois de rotular o vértice r, devemos rotular as subárvores cujas raízes são os filhos de r em T. Seja $f : V(T) \to \mathbb{N}$ uma função que, dado um vértice v de T, devolve o tamanho da subárvore com raiz em v. Após rotular r e repartir os números entre cada uma das subárvores, caímos numa instância menor do problema inicial. Sejam $v_1, v_2, \ldots, v_k$ os filhos de r. Notemos que existem

$$\binom{n-1}{f(v_1), f(v_2), \ldots, f(v_k)}$$

formas de particionar um conjunto de $n-1$ elementos, em k conjuntos de cardinalidades $f(v_1), f(v_2), \ldots, f(v_k)$, respectivamente (veja o seguinte link para mais informação). Além disso, cada uma dessas partições gera soluções diferentes. Finalmente, obtemos a seguinte recorrência

$$H(T, n) = \binom{n-1}{f(v_1), f(v_2), \ldots, f(v_k)} \prod_{i=1}^{k} H(T_i, f(v_i)),$$

onde $T_1, T_2, \ldots, T_k$, são as subárvores com $v_1, v_2, \ldots, v_k$, como raiz, respectivamente. Se abrimos a recorrência e cancelamos termos comuns, chegamos à seguinte formula.

$$H(T, n) = \frac{(n-1)!}{\prod_{v \in V(T) \setminus \{r\}} f(v)}.$$

Para avaliar essa formula podemos fazer o seguinte. Calcular os expoentes da factorização prima de $(n-1)!$ num vetor. Para fazer a factoração, pre-calculamos os primos até 50000. Depois, fatoramos cada número até $n-1$ ou usamos a formula de Polignac. Finalmente, subtraímos a esse vetor os expoentes da fatoração de cada $f(v)$ no denominador. O algoritmo obtido consome tempo $O(n \lg n)$. Veja no seguinte link uma implementação desse algoritmo.