Editorial - Maratona Mineira de Programação 2018

As letras estão dispostas de acordo com a ordem que foi colocada no contest do URI.

Resumo:
A. Teoria dos Números, BFS (?)
B. LCA, DFS, mochila
C. Força Bruta
D. Balão++
E. SegmentTree/BIT
F. Análise Combinatória
G.Ordenação
H.Programação Dinâmica
I. Força bruta/Histogramas (?)
J. AD-Hoc
K.DFS, gramáticas (?)

Os marcados com (?) ainda não resolvemos. Em breve...

B. Crise Hídrica

Neste exercício era nos dados um grafo, em que cada vértice era uma casa, e as arestas eram as ligações entre essas casas. **M** dessas casas tinham um preço de venda. Eram então dados **Q** caminhões, que sempre começavam em uma casa A e percorriam o caminho até a casa B, deixando **L** litros de água por todas as casas passadas nesse caminho. O cara tinha uma quantidade **D** de dinheiro e queria saber quais casas ele deveria comprar de forma a maximizar a quantidade de água que ele poderia obter.

Podemos perceber que se soubermos quanto cada casa terá de água após os Q caminhões passarem, a resposta se resumiria a uma mochila. O mais difícil era obter quanto cada casa recebeu de água de uma forma eficiente.

Pelo o enunciado do exercício podíamos ver que o grafo é uma árvore, e isso nos dá algumas possibilidades. Aqui vamos utilizar a técnica do LCA (Lowest Common Ancestor).

Para cada caminhão que percorre o caminho de X a Y deixando L de água, colocaremos L litros de água em X e Y, e colocaremos -L litros de água no LCA de X e Y e -L no pai desse LCA.

Após fazer isso, basta fazermos uma DFS percorrendo a árvore e espalhando a água dos vértices.

Se implementarmos o LCA de forma eficiente, conseguiremos obter uma complexidade de O**(QlogN)**. A DFS vai demorar apenas **O(N)**.

Após esses procedimentos, basta agora fazermos uma mochila, em que os objetos são as casas, onde cada uma tem um custo e um valor (água que cada uma recebeu), e o limite da mochila sendo o dinheiro do cara. Essa mochila tem por complexidade **O(D*N)**.

C. Dados

Aqui nos davam **N** dados clássicos, e qual a face deles estava sendo mostrada. O objetivo do exercício era ver qual o menor número de movimentos para conseguir deixar todos os dados com a mesma face sendo mostrada. Em cada movimento só poderíamos rotacionar o dado para um dos seus lados adjacentes. Podemos concluir então que para conseguir qualquer face era necessário 1 movimento, com exceção da face oposta, que são necessários 2 movimentos.

Podemos então, testar para cada número (1, 2, 3, 4, 5 e 6), qual a quantidade de movimentos para conseguir deixar a face dos **N** dados com esse número. A resposta é a menor delas.

Complexidade: o algoritmo tentará colocar todos os dados numa face. São 6 faces e N dados, então ele executará 6*N vezes. Portanto, a complexidade é **O(N)**, linear.

D. Feijão

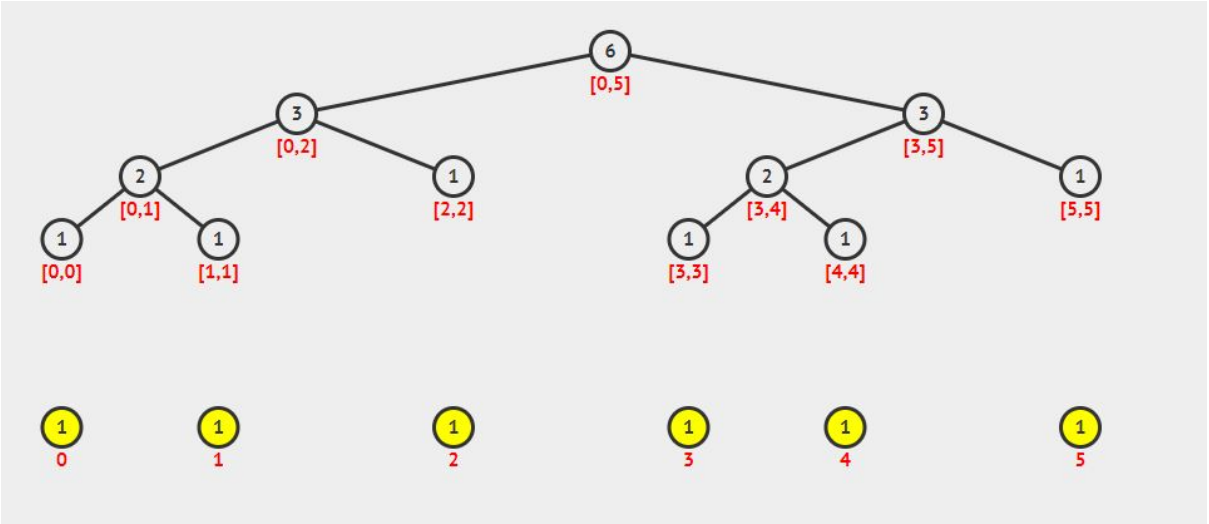
Nesse exercício, precisávamos mostrar em qual copo estava o feijão. A entrada contém 4 inteiros representando os copos, sendo 3 deles não tinham o feijão. Os copos eram representados pelo número 0, e o copo com feijão era indicado pelo número 1. A saída deverá ser o número que representa o copo que tem o feijão dentro.

E. Gonaldinho

Neste exercício precisávamos remover elementos de um array em uma ordem preestabelecida, e a medida que fossemos removendo, precisávamos indicar em qual índice do array o elemento removido estava.

Como o número de elementos podia ser de até **5*10^5**, precisávamos fazer a remoção e a consulta da posição de forma eficiente. O que nos ajuda nesse exercício, é que os valores estão sempre ordenados.

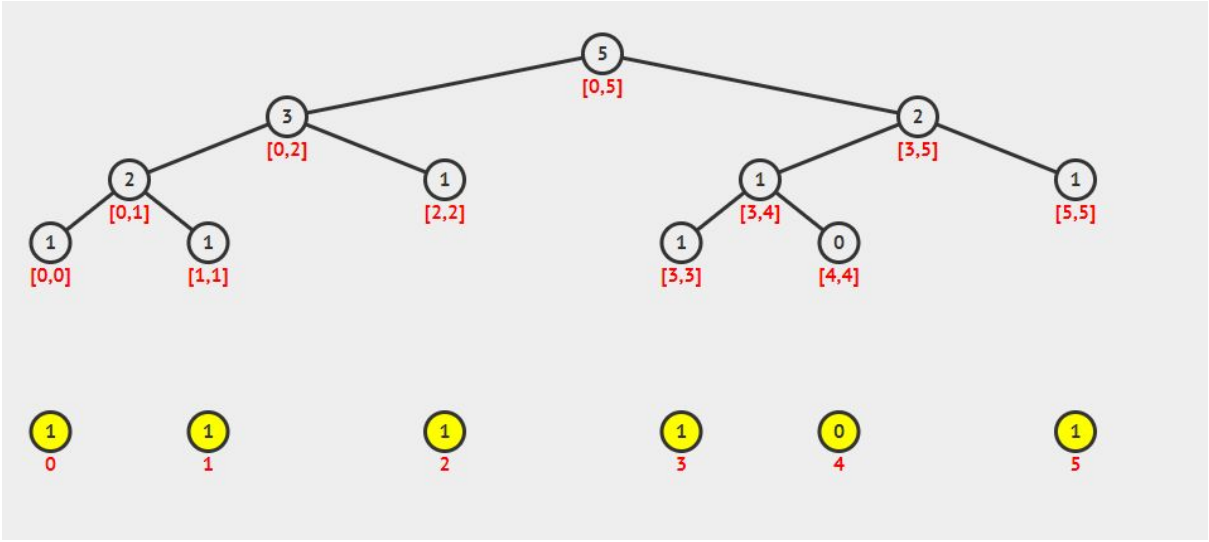
Podemos então, pensar nesse array como uma SegmentTree de soma. Criamos um vetor de N posições e em cada posição colocamos 1. Para N = 6, a árvore seria a seguinte:



Seja a ordem preestabelecida das remoções 5,4,1,2,3,6.

Eis como seria para descobrir a posição do 5 e removê-lo:

Fazemos uma query na SegmentTree da posição 0 até 5-1 (pois indexamos o array a partir do 0), para obtermos quantos termos temos à esquerda do 5 (incluindo ele). Isso resultaria em 5. Ou seja, o elemento 5 estava na posição 5 desse array. Após a query, fazemos um update na posição 5, e transformamos o antigo 1 que lá continha, em 0. Assim, em teoria, o elemento 5 não existe mais no array, e quando formos fazer uma query que o engloba , ele não será considerado. Eis a árvore após esse update:



Portanto, bastava repetir esse processo pra todas as remoções. Cada query e update (por ser point update) na SegmentTree leva O(logN) de tempo, então a complexidade total do algoritmo seria **O(NlogN).**

F. HM

Esse problema fala de Isabel, uma mulher que tem uma loja de roupas e é muito organizada e sempre tem a mesma quantidade de roupas masculinas e femininas em sua loja. Ela quer separar sua loja em categorias considerando esses dois critérios:

1. Todas as categorias possuirão o mesmo número de roupas masculinas e femininas.
2. As roupas já estão penduradas em uma grande arara e, para minimizar seu trabalho, ela apenas deseja colocar divisórias entre as categorias, sem trocar nenhuma roupa de lugar.

A questão é : de quantas formas ela poderia fazer isso?

A entrada é uma string composta pelas letras ‘M’ e ‘F’, representando roupa masculina e roupa feminina, respectivamente. E a saída é o número de maneiras possíveis que Isabel pode categorizar seu estoque (mod 10^9 + 7).

Primeira coisa a fazer era descobrir em quantos lugares tínhamos um pedaço válido (mesma quantidade de roupas masc. e fem.). Na tabela abaixo os índices em vermelho representam as posições em que temos estoques válidos (considerando as roupas à esquerda).

0123 MFMF >2 válidos	**012345** MFMFFM >3 válidos	**0123** MMFF >1 válido

Com isso descobrimos x: quantos trechos válidos existem. Assim existem x-1 locais para colocar divisórias.

Podemos considerar o problema como: se tenho n espaços, de quantas formas diferentes eu posso selecionar alguns deles?
Que seria a mesma coisa de : dado um conjunto com n elementos, quantos subconjuntos existem?

A fórmula para calcular os subconjuntos de um conjunto com n elementos é 2^n.

Assim, a solução do nosso problema seria 2^(x-1) % (10^9+7)

OBS: pelos exemplos de caso de teste nós deduzimos a fórmula, e ao lembrarmos da fórmula da quantidade de subconjuntos, nós nos convencemos que estava certo.

G. Montes Claros

Neste exercício eram dados **N** descrições de um monte, que eram um par de inteiros representando, respectivamente, sua distância da casa de Renato e sua tonalidade. Renato afirma que a medida que a distância dos montes vai aumentando em relação a sua casa, suas tonalidades vão diminuindo. Portanto, o exercício pedia pra gente verificar se isso era verdade. Podemos criar uma struct para representar um monte:

```
struct monte
{
	long long distancia,tom;

	bool operator < (const monte &q) const
	{
		return distancia < q.distancia;
	}

}montes[N];
```

Vamos **ordenar** esse vetor de montes pelas suas distâncias, em ordem crescente (por isso a sobrecarga do operador ali na struct). Depois basta a gente percorrer os N-1 montes, verificando se o monte da frente tem um tom maior (que indica que o Renato tá mentindo):

Se (montes[i]. tom < montes[i + 1].tom) // para todo i de 1 a n-1

O menino tá mentindo.

A complexidade do algoritmo vai tender a ordenação dos montes, pois a verificação é linear, portanto: **O(NlogN)**.

H. Panlíndromo

Esse exercício era bem legal, prato cheio pra quem curte pd.

Precisávamos encontrar o menor custo de transformar a string que nos era dada em um palíndromo. Para isso tínhamos uma operação que era trocar qualquer letra da string por qualquer letra do alfabeto, e para cada operação desse tipo o custo era a distância da letra original para a nova letra no alfabeto (tendo em vista que o alfabeto é circular, ou seja, o custo de transformar ‘a’ em ‘z’, é 1, e não 25). Outra opção que o exercício nos dava era o fato de podermos quebrar a string em K segmentos contíguos, e essa operação é sem custo.

Bom, o truque aqui é inicialmente resolvermos o problema pra quando K é 1, ou seja, quando verificamos o custo para string toda, sem quebras. Se utilizarmos a técnica da programação dinâmica, podemos construir todas as soluções ótimas para

todos os segmentos da string, ou seja, construindo a solução para o segmento da string toda, temos as soluções para todos os segmentos da string.

Vamos então construir essa solução.

Seja X a string e L e R os limitantes inferiores e superiores do atual estado da recorrência. A pd seria dada por:

Se L >= R **DP[L][R] = 0**
Se X[L] != X [R] **DP[L][R] = min(DP[L + 1][R - 1] + cust, DP[L + 1][R - 1])**
Senão **DP[L][R] = DP[L + 1][R - 1]**

obs: o **cust** é o custo de trocar a letra X[L] pela X[R]. Como podemos ver, sempre será ótimo trocar uma pela outra para transformarmos a string em palíndromo.

Após construirmos a DP acima, teremos todas as soluções ótimas para todos os seguimentos da string.

Agora, tentaremos quebrar a string em no máximo K segmentos, para diminuirmos o custo total. Caímos em outra recorrência dado pelo seguinte:

Seja **pos** e **kz,** a posição em que estou na string e quantos segmentos eu ainda posso quebrar, respectivamente, temos a seguinte pd:

```
int pd2(int pos, int kz)
{
	if(pos == n && kz >= 0) return 0;
	if(kz < 0) return 0x3f3f3f3f;
	if(dp2[pos][kz] != -1) return dp2[pos][kz];
	int ans = 0x3f3f3f3f;
	for(int i = pos ; i < n ; i++)
	{
		ans = min(ans, pd2(i+1, kz-1) + pd(pos,i));
	}
	return dp2[pos][kz] = ans;
}
```

Nela, estamos criando todas as possibilidades de quebrar a string em K segmentos contíguos, ou seja, em cada estado tentamos mandar a pd para até N (tamanho da string) posições a frente, e somamos o custo de transformar aquele pedaço em palíndromo (que é dado pela nossa primeira pd) a resposta.

Ao final temos como complexidade: primeira pd O(N^2), segunda pd O(N^2K). Portanto a complexidade total do algoritmo é: **O(N^2K)**.

J. Aplicando Prova

Nesse problema, temos que verificar se a professora pode aplicar a prova ou não. Existem 2 tipos de prova, o tipo 1 e o tipo 2.

São dados valores de **N, M** e **C**. **N** representa o valor de carteiras por fileira, **M** representa o número de fileiras e **C** representa o número de distância que tem que ter entre alunos com o mesmo tipo de prova, podendo ser carteira vazia ou aluno com outro tipo de prova. Cada fileira deve ter pelo menos uma fileira vazia a direita e a esquerda.

Primeiramente, precisamos verificar a condição de fileira vazia à esquerda e fileira vazia a direita.

```
for(int i=0;i<m;i++){
    if(coluna[i] != 0 && (coluna[i-1] != 0 || coluna[i+1] != 0)){
        erro++;
        goto final;
    }
}
```

Depois dessa condição ser verificada e não tiver erros, precisamos ver se a condição de distância entre alunos com tipos de provas iguais é maior ou igual a **C.**

```
for(int i=0;i<m;i++){//acessar por coluna
    int ult1 = -1, ult2 = -1; //nao apareceram
    for(int j=0;j<n;j++){
        int num = mat[j][i];
        if(num == 1){
          if(ult1 == -1)//nao tinha antes
            ult1 = j;
          else{//tinha antes
            if(j - ult1 <= c){//deu ruim
              erro ++;
              goto final;
            }
            else
              ult1 = j;
          }
        }
      if(num == 2){
          if(ult2 == -1)//nao tinha antes
            ult2 = j;
          else{//tinha antes
            if(j - ult2 <= c){//deu ruim
              erro ++;
              goto final;
            }
            else
              ult2 = j;
          }
        }
    }
}
```