Seletivia individual 2019: Primeira prova

Nathan Benedetto Proença

3 de maio de 2019

A: Cats and Dogs

A ideia do problema é tratar os casos e entender, mesmo na historinha meio confusa, quando é possível montar uma resposta. Basicamente, precisava fazer observações simples sobre o problema até resolver.

Seja c,d e l respectivamente a quantidade de gatos, cachorros e patas contadas.

1. A quantidade de patas deve ser um múltiplo de 4. Caso contrário, é impossível.
2. Podemos então dividir a quantidade de patas por 4, e ter quantos animais estão em contato com o chão. Seja s esse número, ou seja, $s = l/4$.
3. Cachorros sempre tem os pés no chão. Logo, $d \leq s$.
4. Todo mundo com pé no chão tem que ser gato ou cachorro. Logo, $s \leq d+c$.
5. Cada cachorro pode no máximo carregar dois gatos nas costas. Logo, $c+d-s$, que é a quantidade de animais que não tem o pé no chão, precisa ser no máximo $2d$. Ou seja, $c + d - s \leq 2d$.

Considerando então que todas essas propriedades valem, ou seja, que l é múltiplo de 4 e que valem as restrições 3, 4, 5, é possível encontrar uma configuração de gatos montados em cachorros que seria uma resposta válida. Assim, essa caracterização é um se e somente se, e é o que devemos checar.

B: Anansi's Cobweb

Para resolver esse problema é necessário conhecer a estrutura de union-find. Se não a conhece, tem lição de casa:

1. Aula do Russo sobre union-find.
2. Aula de análise comentando

Em especial, e usando a terminologia do material de análise, entender por que tanto a *heurística dos tamanhos* quanto a *heurística da compressão de caminhos* por si só já tornariam o algoritmo $\lg n$ é interessante por te dar ideias que tu pode aplicar em outros problemas.

Enfim, sobre o union-find: Ele permite ir adicionando arestas em um grafo e checar se dois vértices estão conectados em tempo basicamente constante.

Podemos então resolver um problema parecido com o do enunciado. Suponha que você começa com um grafo vazio, e vai recebendo arestas para colocar, e precisa saber quantas componentes tem em cada iteração. Note que com o union-find, a contagem de componentes fica trivial: Começamos com n componentes, e pra cada aresta adicionada podemos decidir se ela diminui ou não a quantidade de componentes. Se estamos adicionando uma aresta entre vértices já conectados, a quantidade de componentes se mantém. Caso contrário, ela decresce em 1.

Assim, para resolver o problema da prova, basta processar as perguntas na ordem contrária. A operação de remover arestas se torna colocar, e ao invés de terminar no grafo vazio, começamos nele.

A ideia de guardar as perguntas que o problema dá e processar em uma ordem conveniente costuma ter várias aplicações, e simplificar diversos problemas. Costumamos chamar isso de resolver o problema *offline*. Escolhi esse problema por ser um caso simples onde a ideia funciona bem.

C: Capital Movement

Denote por n a quantidade de vértices do grafo, e m a de arestas. Esse problema é interessante por ter uma solução $O((n+m)\lg n)$ para qualquer grafo!

Vamos lá. Cada vértice tem pesos, e queremos, para cada vértice $i \in V$, encontrar o vértice de maior peso não é nem i nem seus vizinhos. A solução?

Ordene os vértices decrescentemente em peso em um vetor auxiliar, e para cada vértice guarde sua adjacência em um `set`. Para calcular a resposta de um vértice i, apenas percorra o vetor auxiliar até encontrar o primeiro vértice que não é i e nem está na adjacência de i. Cada pergunta tem custo $\lg n$, pelo uso do `set`.

Quanto tempo isso vai levar? Para cada vértice i, no pior caso precisamos olhar para ele e todos seus vizinhos. Logo, se tem d_i arestas incidentes em i, iremos olhar no set no máximo $d_i + 1$ vezes. Ou seja, a complexidade é no máximo

$$\sum_{i \in V} (d_i + 1) \lg n = \lg n \sum_{i \in V} (d_i + 1) = \lg n (2m + n).$$

onde vale que $\sum_{i \in V} d_i = 2m$ pelo fato de que cada aresta contribui exatamente duas vezes para a soma dos graus.

D: Assembly line

A parte interessante desse problema é que estamos lidando com uma operação que, em algebrês, não é associativa. Como o caso da folha mostra, neste caso ao se "juntar" uma sequência de elementos, pode-se obter resultados diferentes dependendo da ordem na qual juntamos eles dois a dois.

Compare, por exemplo, com o problema de multiplicação de matrizes. Em ambos os casos, temos uma sequência de coisas que queremos juntar em uma só, e temos uma função que junta duas em uma. Na multiplicação de matrizes, juntar é o produto matricial. Aqui, é aplicar a função dada na entrada.

Mas ambos os problemas tem uma diferença importante. No problema das matrizes, a matriz que vai ser obtida quando todos os produtos forem feitos é única. Logo, só precisamos encontrar a sequência que minimize o custo para chegar nela. Neste problema, sequências diferentes de operações não apenas tem custos diferentes, como também te levam a um resultado final diferente.

Mas, a comparação com o algoritmo de multiplicações de matrizes é bem útil, pois a solução aqui também é programação dinâmica. Mais ainda, é *a mesma programação dinâmica*, com a única diferença que você precisa calcular para todo valor possível, qual é a forma mais barata de construí-lo.

Há uma ressalva, e vários tomaram TLE por isso. Pense que o estado da sua PD é (i, j, l), onde i, j representam o intervalo $[i, j)$ e l representa o símbolo que tu quer montar. Considerando os subintervalos, você vai quebrar em um índice m e tentar combinar as soluções de $[i, m)$ com $[m, j)$. Mas, para isso, você precisa olhar todos os k valores possíveis de $[i, m)$, e todos os k valores possíveis de $[m, j)$, e checar se eles formam o símbolo l desejado.

Ou seja, a complexidade dessa abordagem é $\Theta(n^3k^3)$, pois existem $\Theta(n^2k)$ estados fazendo trabalho $\Theta(nk^2)$.

Contudo, muito trabalho está sendo repetido. Vamos considerar como estado apenas os intervalos. Estou no estado $[i, j)$, considerando combinar a partição $[i, m)$ com $[m, j)$. Cada lado tem k valores possíveis, e se eu iterar pelas k^2 combinações, eu já tenho todas as contagens para meu intervalo $[i, j)$.

Ou seja, a complexidade fica $\Theta(n^3k^2)$, pois temos $\Theta(n^2)$ estados fazendo trabalho $\Theta(nk^2)$.

Temos então mais uma coisa comum em programação competitiva. Muitas vezes, num problema de programação dinâmica, existem diversas formulações que obtém a mesma resposta. As vezes, algumas são mais eficientes que as outras. Logo, se você está tentando resolver um problema e chegou numa pd que iria estourar o tempo, não a descarte direto, mas procure formas equivalentes de montar a pd, ou formas mais eficientes de calcular as coisas.

E: Frontier

Ah, geometria. Este problema é interessante por não ser apenas o cálculo do fecho convexo.

Temos dois tipos de pontos no plano: torres e monumentos. Queremos encontrar o polígono de menor perímetro que pode ser formado pelas torres, com a restrição de que todos os monumentos precisam estar no interior do polígono.

Imagine que você é um guarda que percorre algum polígono válido no sentido horário. Por ser um polígono, termos os monumentos no seu interior, e você estar andando no sentido horário, temos que *durante toda a caminhada os monumentos estarão à sua direita*, e você irá terminar onde começou.

Por outro lado, dado qualquer caminho que comece e termine na mesma torre e, para cada aresta, os monumentos estão do lado direito, temos uma configuração válida.

Ou seja, para resolver o problema, basta montar um grafo *direcionado* cujos vértices são as torres, e um arco existe entre dois vértices se e somente se todos os monumentos estão á direita dessa aresta. Montar esse grafo de maneira trivial tem custo $O(n^2m)$, que cabe no tempo.

Então, basta encontrar o menor ciclo neste grafo, o que pode ser feito com vários Dijkstras, por exemplo. Como o grafo é potencialmente denso, pode-se ainda usar a implementação $O(n^2 + m)$ do Dijkstra (pegando o mínimo de maneira trivial entre os vértices não processados) e ter uma solução $O(n^3+nm)$.

F: Midnight Cowboy

Vou começar resolvendo um caso específico desse problema.

Suponha que temos um caminho com n vértices, onde o primeiro é o hotel caro, e o último é o hotel barato. Para cada vértice neste caminho, qual é a probabilidade do nosso bebum de chegar no hotel barato?

Seja H_i a probabilidade de chegar no hotel barato dado que ele está no vértice i. Trivialmente, se ele já está no caro, a probabilidade é zero. Logo, $H_0 = 0$. De maneira similar, se ele já está no barato, a probabilidade é um. Logo, $H_{n-1} = 1$. Para os demais, ele tem probabilidade meio de ir para cada lado. Logo, para qualquer i diferente de 0 ou de $n - 1$, temos que

$$H_i = \frac{H_{i-1} + H_{i+1}}{2}.$$

Podemos olhar uns casos pequenos para achar uma resposta simples. Se $n = 3$, temos que os valores de H são 0, 1/2, 1. Se $n = 4$, dá um pouco mais de trabalho, mas dá pra ver que os valores de H são $0, 1/3, 2/3$ e 1. Assim, podemos ter o palpite de tentar a solução

$$H_i = \frac{i}{n-1}.$$

Podemos ver que ela bate com os valores da borda – 0 e $n-1$ – e que respeita a igualdade acima. Ou seja, é uma solução válida para o problema.

Mas e daí? E o problema original?

A notícia boa é que toda parte "probabilística" já foi resolvida, e agora podemos voltar a ser maratonistas comuns, com medo de matemática. O grafo do problema sempre é uma árvore. Logo, podemos assumir que sempre tem uma folha. Precisamos considerar dois casos para essa folha. Se ela for qualquer um dos hotéis, já sabemos seu valor. Se não, caso nosso bebum esteja nela, sua única opção é ir para seu pai. Ou seja, se a folha não for nenhum dos hotéis, a resposta para ela é igual ao valor da resposta para o seu pai.

Podemos então aplicar essa ideia repetidas vezes, e ir removendo todas as folhas que pudermos do grafo. Como a única restrição é que não podemos remover os hotéis, quando esse processo terminar teremos exatamente o caminho entre os dois hotéis. Mas este caso sabemos calcular!

Algoritmicamente, pode-se fazer uma dfs a partir do hotel caro. Conforme formos "retirando" as folhas, o bebum irá parar exatamente no LCA entre sua posição inicial e o hotel barato. Tendo então a distância entre o hotel barato e o caro, e entre a posição que o bebum foi parar, podemos resolver o problema.

G: GCD Extreme

Primeiro note que pode-se precalcular as respostas para todo n e apenas imprimir conforme o necessário. Pensando assim, eu acho que existem dois requisitos para resolver esse problema em $O(n \lg n)$. Um é um pequeno resultado sobre a função ϕ de Euler, e o outro é sobre o crivo de Eratóstenes.

Temos então, um editorial em 3 atos:

Ato 1: Propriedades da ϕ de Euler

Vou começar com os resultados sobre a função ϕ. Dado que é algo de computeiro para computerios, irei fazer uma demonstração combinatória, com argumentos de contagem, para evitar sofrimento desnecessário.

Vamos lá. Podemos definir a função ϕ como

$$\phi(n) = |\{i \mid 1 \leq i \leq n, \quad \gcd(i,n) = 1.\}|.$$

Olhando para esta definição, podemos pensar em variações disso. Seja d um divisor de n. Nesse problema, é interessante calcular o tamanho do seguinte conjunto:

$$S_d = \{i \mid 1 \leq i \leq n, \quad \gcd(i,n) = d\}.$$

Claro, tudo bem se você ainda não vê pra que serve isso, irei explicar depois. Enfim, voltemos ao conjunto. Temos que, por definição, que $\phi(n) = |S_1|$. Irei afirmar que, em geral, $|S_d| = \phi(n/d)$. Ou seja, que S_d tem o mesmo tamanho que

$$\{i \mid 1 \leq i \leq n/d, \quad \gcd(i, n/d) = 1\}.$$

Para isto, basta encontrar uma bijeção entre ambos os conjuntos. Mas isto é fácil: Pegue a função que leva i em i/d. Para ver que isso vai funcionar, primeiro note que

$$\gcd(i,n) = d \iff \gcd(i/d, n/d) = 1.$$

Como, trivialmente, $1 \leq i \leq n$ vale se e somente se $1 \leq i/d \leq n/d$, temos que a função leva o primeiro conjunto no segundo. Pela mesma linha, a função que leva i em $i \cdot d$ é a inversa, e leva o segundo no primeiro.

Ou seja, $|S_d| = \phi(n/d)$. Tem uma outra ideia que vamos usar nesse problema, que aplicado com essa igualdade que acabamos de provar, prova um resultado "famoso" sobre a ϕ. Irei fazer esse pequeno detour.

Pegue todos os números até n. Note que, para cada um deles, o gcd entre ele e n precisa dividir n. Ou seja, podemos particionar esse conjunto de acordo com qual divisor vai dar o gcd entre o número e n. Em símbolos,

$$\{1, \ldots, n\} = \bigcup_{d|n} S_d.$$

Mas como acabamos de ver, temos que $|S_d| = \phi(n/d)$. Logo,

$$n = \sum_{d|n} \phi(n/d) = \sum_{d|n} \phi(d).$$

Essa fórmula seria uma maneira útil de checar se sua implementação de ϕ está correta, se apenas existisse uma forma eficiente de fazer essa soma nos divisores...

Ato 2: O Crivo de Eratóstenes

Não tem como não admirar o Eratóstenes. O cabra nasceu em 276 antes de Cristo, fincou um graveto no chão e estimou a circunferência da terra com um erro de 15%. Mas, para os maratonistas, o legal é que ele inventou um crivo, que todo mundo estuda, e é uma maneira rápida de encontrar números primos.

Agora, o procedimento que o Eratóstenes inventou é muito mais interessante. Pegue por exemplo, a função ϕ de Euler. Podemos modificar o crivo para calcular o valor de ϕ para todos os números menores ou iguais a n, em tempo $n \lg n$.

Pra isso, podemos utilizar a fórmula

$$\phi(n) = n \prod_{p|n} \left(\frac{p-1}{p} \right).$$

Dá pra provar também com ideias de contagem e combinatória, mas aqui só vou usar e pedir pra vocês olharem a wikipédia. Para calcular a $\phi(n)$ com essa fórmula, precisamos iterar em todos os primos que dividem n. Vamos olhar para uma implementação possível. Suponha que `N` é até onde queremos calcular, e que `ll` é um inteiro de 64 bits.

A ideia é ter um vetor `phi`, que começamos com o valor de n, e irmos corrigindo conforme encontramos os primos. Ou seja, podemos inicializar com

```
ll phi[N];
for (ll i = 0; i < N; i++) phi[i] = i;
```

E então, basta utilizar cada primo para corrigir os valores de seus múltiplos:

```
for (ll i = 2; i < N; i++)
  if (phi[i] == i) {
    for (ll m = i; m < N; m += i) {
      phi[m] /= i;
      phi[m] *= (i - 1ll);
    }
  }
```

Basicamente, o mesmo argumento que garante que o crivo irá encontrar os primos irá garantir que `phi[n]` é de fato $\phi(n)$. E essa ideia pode ser explorada de outras maneiras. Não precisamos (apesar de podermos) usar que o número é primo. Note que, no código abaixo, iremos eventualmente considerar todos os divisores de um número m, e sua complexidade total ainda vai ser $O(n \lg n)$.

```
for (ll d = 2; d < N; d++)
    for (ll m = d; m < N; m += d) {
                // d divide m
    }
  }
```

Ato 3: O problema GCDEX

Finalmente, estamos armados para enfrentar nosso rival. Defina

$$F_n = \sum_{i=1}^{n} \gcd(i, n).$$

No problema, você só soma até $n-1$, mas isso é relativamente fácil de resolver, e essa expressão é mais simples de trabalhar. Basicamente, estou afirmando que se conseguirmos calcular $\sum_{i=1}^{n} F_n$, resolvemos o problema.

Finalmente, para resolver o problema, você pode escrever F_n como uma soma nos divisores d de n, e entender qual a importância da discussão sobre o tamanho de S_d. Com isso, e usando o crivo como comentei, é possível resolver este problema em $O(n \lg n)$.