Tutorial: Primos Arrojados

Arthur Luís Komatsu Aroeira

Para verificar se um número é um primo arrojado, basta realizar a seguinte rotina enquanto o número tiver mais de 1 dígito:

- verificar se ele é um número primo. Se sim, continuar a rotina. Se não, esse número já não é mais arrojado;
- retirar seu último algarismo (pode ser feito fazendo a divisão inteira por 10).

Verificar se o número é primo pelo método da força bruta até a raiz quadrada resulta em TLE, já que para cada query, no pior caso, rodamos o algoritmo $O(\sqrt{n} \log n)$ vezes ($O(\log n)$ vem da quantidade de verificações para cada número, ou seja, sua quantidade de algarismos). Logo, a complexidade total neste caso será $O(T\sqrt{n} \log n)$, o que é muito lento pelos limites do problema.

Um jeito mais rápido de verificar se os números são primos é rodar o algoritmo do crivo de Erastótenes antes em $O(n \log \log n)$ para verificar se um dado número é primo em $O(1)$ e responder cada query em $O(\log n)$. A complexidade total neste caso será $O(n \log \log n + T \log n)$, o que é suficiente pelos limites do problema.

Outra solução é utilizando a chamada "solução offline". Como existem apenas 78 primos arrojados menores que 10^7, é possível fazer um programa mais lento que gere todos eles previamente e incluir esta informação na solução final.

Tutorial: Listas Ordenadas

Daniel Saad Nogueira Nunes

A ideia da solução é a seguinte.

Crie dois vetores C e D, tais que C armazena todos os intervalos $[i, j]$ de sublistas crescentes e D faz o mesmo mas para listas decrescentes. Este processo pode ser feito em tempo $\Theta(n)$ ao realizar uma varredura linear no vetor de valores (técnica two-pointers).

Naturalmente, ambos os vetores estarão ordenados.

Para cada consulta sobre um intervalo $[a, b]$:

- Se $a = b$, a lista $L[a, b]$ tem tamanho unitário, e portanto não pode ser crescente e nem decrescente.
- Se $a < b$, deve ser realizada uma busca binária sobre C em busca do intervalo $[i_c, j_c]$ com menor j_c possível que seja maior ou igual a b. Isso pode ser feito usando `lower_bound` da `STL`. O mesmo é feito com o vetor D, obtendo o intervalo $[i_d, j_d]$.
- Se $[i_c, j_c]$ existe, então basta verificar se $i_c \leq a$. Se for o caso, deverá ser impresso `''crescente''`.
- Caso contrário, se $[i_d, j_c]$ existe, basta verificar se $i_d \leq a$. Se for o caso, deverá ser impresso `''decrescente''`.
- Senão: o programa deve imprimir `''nenhum''`.

A complexidade total desta solução é $\Theta(N + Q \lg N)$.

Uma solução alternativa utiliza da técnica de soma de prefixos, sobre dois vetores, A e B. A é preenchido com 1 no intervalo $[i, j]$ se este é um intervalo crescente e com 0 caso contrário. Analogamente, B é preenchido com 1 no intervalo $[i, j]$ se este é um intervalo decrescente e com 0 caso contrário. Após este preenchimento computa-se a soma de prefixos, tanto em A, quanto em B.

Dada uma consulta $[l, r]$, com $l < r$ temos:

- Se $A[r] - A[l] = r - l$, então $[l, r]$ é um intervalo crescente.
- Se $B[r] - B[l] = r - l$, então $[l, r]$ é um intervalo decrescente.
- Caso não se enquadre em nenhuma das duas alternativas anteriores, o intervalo $[l, r]$ não possui uma ordenação.

Se $l = r$, a resposta só pode ser `''nenhum''`.

A complexidade total desta solução é $\Theta(N + Q)$, uma vez qaue cada consulta é respondida em tempo constante.

Tutorial: Quebra-Cabeça

Edson Alves da Costa Júnior

Para formar um palíndromo, a primeira letra deve ser igual a última, a segunda igual a penúltima, e assim por diante. Desta forma, basta que cada letra apareça um número par de vezes.

A única exceção é o caso onde o tamanho da string S é ímpar: neste caso, uma das letras tem que aparecer uma quantidade ímpar de vezes.

A solução tem complexidade $O(N)$.

Tutorial: Sequência Leia e Fale

Daniel Saad Nogueira Nunes

A partir de um número da sequência leia-e-fale, podemos computar o próximo utilizando uma técnica chamada Run-Length-Encoding. Ela consiste em contar o número de símbolos iguais consecutivos e substituí-los por por dois símbolos: um inteiro com a quantidade de repetições e o símbolo que se repete.

Para evitar computações redundantes, é possível pré-computar todos os números da sequência leia-e-fale uma única vez e guardá-los em uma tabela. Desta forma, só é necessário ler o índice e recuperar o número ao inspecionar a entrada da tabela correspondente.

Tutorial: Self-Service

Edson Alves da Costa Júnior

Como a massa G é dado em gramas e o desconto D em porcentagem, temos a seguinte relação, onde V é o valor que o restaurante cobra por quilo e R o valor final da conta:

$$R = V \times \left(\frac{G}{1000}\right) \times \left(1 - \frac{D}{100}\right)$$

Logo basta isolar V na expressão acima para obter uma solução com complexidade $O(1)$.

Tutorial: Canibais

Vinicius Ruela Pereira Borges

Este problema é semelhante ao problema da Mochila Booleana, sendo resolvido por um algoritmo baseado em programação dinâmica. A ideia é maximizar o valor de energia das pessoas que serão devoradas pelos canibais, respeitando o custo máximo que os canibais estão dispostos a gastar para capturá-las. A solução consiste em criar uma matriz $\mathrm{P}[1..N][1..M]$ e de maneira iterativa, resolver cada sub-problema possível dado pelo par (i, j), em que determina-se o valor máximo de energia que os canibais conseguem obter caso devorem i pessoas a um custo máximo de captura j. A solução final estará armazenada na posição $\mathrm{P}[N][M]$ e seu custo computacional é $O(NM)$.

Tutorial: Canetta

Lucas Vasconcelos Mattioli

Vamos inicialmente definir uma solução ingênua para o problema: suponha que um vetor booleano in seja mantido durante todas as Q operações. Sempre que a pessoa i entrar no laboratório, fazemos com que in_i seja 'true' e sempre que i sair, fazemos com que in_i seja 'false'. Além disso, mantemos um vetor de inteiros ans, onde ans_i representa quantos apertos de mão a pessoa i realizou no total. Com essas estruturas definidas, é simples resolver o problema de maneira correta mas lenta: sempre que x entrar no laboratório, passamos em todas as pessoas y de sua lista L_x e, se in_y for 'true', incrementamos ans_x e ans_y em 1, separadamente. Essa solução tem complexidade $O(Q \cdot \sum_{i=1}^{N} S_i)$, a qual não deveria passar nos limites impostos.

Suponha que escolhemos uma constante K antes de resolver o problema (seu valor será definido mais para frente). Com isso em mente, vamos classificar as pessoas em 2 categorias: **small** e **big**. Se o tamanho da lista de amizades L_i de uma pessoa i for menor ou igual à K, então i é considerada **small**; caso contrário, i é considerada **big**.

A solução final utiliza a ideia da solução ingênua: nela, mantemos também as estruturas in e ans. Sempre que uma pessoa **small** entrar no laboratório, fazemos exatamente o que faríamos na solução ingênua: passamos por todos seus amigos e atualizamos as posições de ans respectivas corretamente. Podemos fazer isso porque, por ser **small**, a pessoa tem no máximo K amigos, fazendo com que o número de operações não seja tão grande.

O problema acontece quando uma pessoa i, que é **big**, entra no laboratório. Nesse caso, podemos passar em todas as outras pessoas **big** e verificar quais estão lá dentro e são amigas de i e atualizar ans corretamente. Podemos fazer isso porque existem no máximo $O(\frac{\sum_{i=1}^{N} S_i}{K})$ pessoas **big**. Falta, então, encontrarmos uma maneira de contabilizar quantos amigos **small** de i existem dentro do laboratório naquele momento. Para isso, podemos manter, para cada pessoa **small** x, uma lista B_x de pessoas **big** que têm x como amigo. Além disso, mantemos um vetor de inteiros $bigcache$, pra cada pessoa **big** y, onde $bigcache_y$ indica quantos amigos **small** de y estão dentro do laboratório naquele momento. Assim, sempre que uma pessoa **small** f entrar no laboratório, passamos por todos as pessoas k em B_f e incrementamos $bigcache_k$ em 1. Analogamente, sempre que uma pessoa **small** f sair do laboratório, passamos por todos as pessoas k em B_f e decrementamos $bigcache_k$ em 1. Finalmente, para fazer a contabilização $big - small$ que faltava: quando uma pessoa **big** y entrar no laboratório, basta somar $bigcache_y$ à variável ans_y.

A complexidade final da solução é $O(Q \cdot (K + \frac{\sum_{i=1}^{N} S_i}{K}))$, pois em cada uma das Q operações fazemos no máximo $O(K + \frac{\sum_{i=1}^{N} S_i}{K})$ operações. Para encontrar o K ótimo, temos que minimizar a expressão $K + \frac{\sum_{i=1}^{N} S_i}{K}$. Com alguma manipulação algébrica, pode-se ver que o K que minimiza a equação é $\sqrt{\sum_{i=1}^{N} S_i}$. Assim, dados os limites do problema, uma boa escolha de K seria 316. Caso tenha interesse em estudar mais sobre essa técnica, procure sobre *sqrt decomposition*.

Tutorial: Inventário

Edson Alves da Costa Júnior

O problema consiste em resolver a equação diofantina

$$Mx + Cy = V,$$

com $x, y \in N$. Observe que se o maior divisor comum $d = (M, C)$ de M e C não dividir V, não há solução. Assim, o problema se reduz à equação

$$ax + by = c,$$

com $a = M/d, b = C/d, c = V/d$ e $(a, b) = 1$.

Seja (x_0, y_0) uma solução particular do problema $ax+by = 1$, a qual pode ser encontrada com o algoritmo estendido de Euclides. Seja $x' = cx_0, y' = cy_0$. Assim, $ax' + by' = c$.

Se $x' = qb + r$, com $0 < r \leq b$, temos que

$$c = ax' + by' = a(qb + r) + by' = ar + b(y' + qa),$$

isto é, $x'' = r, y'' = (c - ar)/b$ é solução com x positivo mínimo, caso $y'' > 0$; caso contrário, não há solução. Esta abordagem tem complexidade $O(\log(M + C))$.

Observação: é preciso tomar cuidado com possíveis erros de *overflow* (não é preciso computar o valor de x' diretamente), e atentar ao fato de que r deve ser maior que zero (basta somar b se $r \leq 0$).

Tutorial: AC ou WA?

Edson Alves da Costa Júnior

É possível identificar o i-ésimo caractere de Fibonacci (ou das strings S_j de Tarcísio) sem construir tais strings explicitamente. Para as strings de Fibonacci o i-ésimo caractere é dado por:

```
char nth_char_F(int n, int i)
{
    if (i == 1)
        return 'B';

    if (i == 2)
        return 'A';

    auto a = Fs[i - 1];

    return n <= a ? nth_char_F(n, i - 1) : nth_char_F(n - a, i - 2);
}
```

A rotina para as strings S_j é praticamente idêntica, mudando apenas a ordem de concatenação:

```
char nth_char_S(int n, int i)
{
    if (i == 1)
        return 'B';

    if (i == 2)
        return 'A';

    auto a = Fs[i - 2];

    return n <= a ? nth_char_S(n, i - 2) : nth_char_S(n - a, i - 1);
}
```

Em ambas rotinas, `Fs` é um vetor com os número de Fibonacci pré-computados, que são utilizados para reduzir o problema de computar o i-ésimo caractere de F_k para o problema de computar o i-ésimo caractere de F_{k-1} (se $i \leq$ a) ou o $(i - a)$-ésimo caractere de F_{k-2}, caso contrário, onde a é o $(k - 1)$-ésimo número de Fibonacci. Vale a mesma interpretação para as strings S_j.

Para o problema, basta pré-computar até o 88º número de Fibonacci, que é o primeiro que excede o valor 10^{18}. A primeira chamada de ambas rotinas tem que começar com parâmetro i igual a k, o qual pode ser determinado através da função `lower_bound()` da STL.

Tutorial: Vendedor de Alpiste Neurótico

Jeremias Moreira Gomes

Inicialmente, podemos definir a seguinte função de *custo* para saber o peso necessário, que recebe como entrada uma lista (tab) com os pesos nos n potes, o valor n com a quantidade de potes e um valor v em que potes com valor menor que v serão completadas e potes com valor maior que v terão esses pesos sobressalentes removidos.

```
custo(tab, n, v) {
    ans = 0
    for (i = 0; i < n; i++) {
        if (tab[i] > v) {
            ans = ans + tab[i] - v
        } else {
            ans = ans + v - tab[i]
        }
    }
    return ans
}
```

Essa função *custo* possui característica unimodal. Ou seja, se $[A, B]$ é o intervalo que abrange as quantidades mínimas e máximas de variações de pesos, $\exists x$ $(A \leq x \leq B)$ tal que $\forall y$ $(A \leq y \leq B)$, $custo(x) \leq custo(y)$.

Assim, é possível aplicar uma busca ternária pois existe um valor K tal que:

- $\forall a, b$ com $A \leq a < b \leq K, f(a) > f(b)$
- $\forall a, b$ com $K \leq a < b \leq B, f(a) < f(b)$

O algoritmo fica da seguinte forma:

```
int buscaTernaria(tab, n) {
    precisao = 0.001;
    l = 0
    r = max(tab) // valor máximo de tab
    while ((r - l) / 2.0 > precisao) {
        p1 = (l * 2.0 + r) / 3.0;
        p2 = (l + 2.0 * r) / 3.0;
        v1 = custo(tab,p1, n);
        v2 = custo(tab,p2, n);
        if (v1 < v2) {
            r = p2;
        } else {
            l = p1;
        }
    }
    return custo(tab, r, n);
}
```

Como a função custo tem tempo de avaliação $\Theta(n)$, o custo total do algoritmo é $\Theta(n \log C)$, onde C é o maior valor possível para c_i.

Tutorial: Campo Minado

Daniel Saad Nogueira Nunes

Dado um vetor $V[1, n]$, uma Fenwick Tree é uma estrutura de dados capaz de responder as operações de

1. Atualização: adiciona um valor δ a um elemento $V[i]$.
2. Soma de intervalos: para um intervalo $[i, j]$, computa $\sum_{k=i}^{j} V[k]$

Ambas as operações levam tempo $\Theta(\lg n)$ e podem ser implementadas da seguinte maneira:

```
// Realiza a soma do prefixo V[1,i]
int sum(int i) {
    int sum = 0;
    while (i > 0) {
        sum += ft[i];
        i -= (i & -i);
    }
    return sum;
}

// Obtém a soma do intervalo V[i,j] através da diferença
// de duas somas de prefixo
int sum(int i, int j) { return sum(j) - sum(i - 1); }


// Atualiza o elemento V[i] adicionando o valor
// val a ele e propaga esta modificação aos
// elementos adequados da estrutura de dados
void update(int i, int val) {
    while (i < n) {
        ft[i] += val;
        i += (i & -i);
    }
}
```

Um bom guia para esta estrutura de dados está disponível na plataforma TopCoder (https://www.topcoder.com/comm programming/tutorials/binary-indexed-trees/).

É possível generalizar esta estrutura de dados para um espaço bidimensional, como o apresentado neste problema. Então, para uma matriz $N \times M$, utiliza-se N Fenwick-Trees, uma para cada linha da matriz $Matrix[i][1, M]$.

Para responder a consulta de soma sobre o retângulo com canto superior esquerdo $(x1, y1)$ e canto inferior direito $(x2, y2)$, são utilizadas consultas sobre retângulos menores. Tome os 4 retângulos (A,B, C e D), todos com canto superior esquerdo $(1, 1)$ e cantos inferiores direitos $(x2, y2)$, $(x1-1, y2)$, $(x2, y1-1)$ e $(x1-1, y1-1)$ respectivamente. A soma do retângulo $(x1, y1)(x2, y2)$ pode ser obtida através de $A-B-C+D$. Esta última equação possui um abuso de notação: cada retângulo corresponde à soma dos elementos que se encontram nele.

A atualização sobre um elemento da matriz utilizando a Fenwick-Tree bidimensional de um modo análogo a da estrutura unidimensinal.

Um esboço das implementação é apresentado em seguida:

```
// Realiza a soma do retângulo com canto superior esquerdo
// (1,1) e canto inferior direito (x,y)
int sum(int x, int y) {
    int s = 0;
    for (; y > 0; y -= (y & -y)) {
        s += ft2d[y].sum(x);
    }
    return s;
}
// Realiza a soma do retângulo com canto superior esquerdo
// (x1,y1) e canto inferior direito (x2,y2)
int sum(int x1, int y1, int x2, int y2) {
    int a = sum(x2, y2);
    int b = sum(x2, y1 - 1);
    int c = sum(x1 - 1, y2);
    int d = sum(x1 - 1, y1 - 1);
    return a - b - c + d;
}

// Atualiza o elemento (x,y) somando a ele o valor val
// propaga esta alteração sobre os elementos adequados
// da estrutura de dados
void update(int x, int y, int val) {
    for (; y <= n; y += (y & -y)) {
        ft2d[y].update(x, val);
    }
}
```

Qualquer uma dessas operações pode ser respondida em tempo $\Theta(\lg(NM))$.

Tutorial: Senha

Daniel Saad Nogueira Nunes

Este problema pode ser resolvido utilizando a simplificação de um algoritmo elaborado por Donald Knuth em 1977. O algoritmo em si é mais complexo e garante um número máximo de 5 tentativas para acertar a senha, no entanto, a simplificação é suficiente para este problema e será descrita a seguir:

A ideia é a seguinte:

1. Construa um conjunto com todas as possíveis 1296 tentativas (6^4).
2. Faça $x = 1122$ e retire 1122 do conjunto.
3. Use x como sua tentativa.
4. Receba o par (p, q).
5. Se $(p, q) = (4, 0)$ ou o número de tentativas se esgotou, encerre.
6. Para cada elemento s do conjunto, teste x contra s de acordo com as regras do jogo:
   - Se o resultado de x tomando s como a provável senha é diferente de (p, q), s pode ser eliminado do conjunto.
   - Caso contrário, s é uma potencial solução, deixe-a no conjunto.
7. Faça x receber o menor valor que sobrou no conjunto e continue do passo 3.

Tutorial: Quadratura do Círculo

Edson Alves

Sabendo que a área do círculo de raio R é igual a $A_C = \pi R^2$ e que a área do quadrado de lado L é igual a $A_Q = L^2$, a igualdade $A_C = A_Q$ implica que

$$L = \sqrt{\pi R^2}$$

Outra maneira de se determinar o valor de L é observar que $0 \leq L \leq 2R$, e usar a busca binária até que a condição de parada descrita na saída seja atingida.

Tutorial: Estraga-Festa

Daniel Saad Nogueira Nunes

O problema é simples. Basta computar qual equipe possui a maior quantidade de balões (b) e em caso de empate, o menor tempo acumulado (t).

A resposta é `'N'` caso:

1. $M < b$.
2. $M = b \; t < 300 \cdot M$.

Caso contrário a resposta é `'S'`.

Tutorial: Construindo Genomas

Daniel Saad Nogueira Nunes

Este é um problema clássico da Ciência da Computação denominado *Shortest Common Superstring (SCS)* cujo objetivo é, dado uma sequência de *strings* $S = (s_0, s_1, \ldots, s_{n-1})$, determinar a **menor** *string* que contenha cada uma das *strings*.

Para resolvê-lo, utilizamos o seguinte algoritmo:

1. Retire todas as *strings* de S que estão contidas em outra *string* de S, transformando S efetivamente em um conjunto $S = \{s_{0'}, s_{1'}, \ldots, s_{n'-1}\}$.

2. Compute $overlap[i][j]$ que corresponde ao tamanho do maior prefixo de s_j que é um sufixo de s_i em que $s_i, s_j \in S$.

3. Inicialize SCSLen $\leftarrow \infty$ e SCS $= \perp$.

4. Para cada permutação de $\pi(0, 1, \ldots, n' - 1) = (i_0, i_1, \ldots i_{n'-1})$ faça:

   (a) Compute SCSLen $= \displaystyle\sum_{k=0}^{n'-1} |s_{\pi[k]}| - \sum_{k=0}^{n'-1} overlap[\pi[k]][\pi[k+1]]$.

   (b) Caso SCSLen seja o menor valor encontrado até o momento, atualize SCS ao copiar os caracteres de acordo com a permutação, lembrando de descontar os caracteres duplicados, indicados pela matriz *overlap*.

5. Forneça como saída SCSLen e a SCS propriamente dita.

A complexidade total do algoritmo é $\Theta(nn!)$.

Outra forma de encarar esse problema é reduzi-lo ao problema do caixeiro viajante. Nesta redução, cada *string* é um vértice e quaisquer dois vértices u e v estão conectados com o custo determinado pelo $overlap[u][v]$. Este problema admite uma solução bem conhecida através de Programação Dinâmica com complexidade $\Theta(n2^n) \subsetneq \Theta(nn!)$.