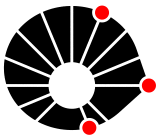
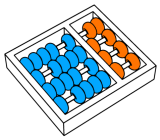




Tópicos em Programação Competitiva: Teoria dos Jogos, Combinatória, Grafos Direcionados e Árvores

Lucas Martiniano *Fábio Usberti*

Relatório Técnico - IC-PFG-26-26
Projeto Final de Graduação
2026 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Tópicos em Programação Competitiva: Teoria dos Jogos, Combinatória, Grafos Direcionados e Árvores

Lucas Martiniano Fábio Usberti

Resumo

Este projeto apresenta o Trabalho de Conclusão de Curso em Engenharia da Computação. Ele corresponde ao Volume II de uma série voltada à programação competitiva, servindo tanto de complemento aos tópicos já abordados quanto de introdução a novos temas para o leitor.

O trabalho reúne um conjunto de tópicos avançados de algoritmos, cobrindo teoria dos jogos, combinatória, probabilidade e algoritmos complementares sobre grafos direcionados e árvores.

1 Teoria dos Jogos

Teoria dos Jogos é uma área da matemática que estuda situações em que dois ou mais jogadores tomam decisões racionais visando maximizar seu próprio ganho. Em Programação Competitiva, a maioria dos problemas de teoria dos jogos envolve dois jogadores que se alternam em movimentos, e queremos determinar quem vence quando ambos os jogadores jogam de forma ótima.

Nesse capítulo vamos estudar jogos combinatórios, que são jogos em que não há elemento de sorte, ambos os jogadores têm informação completa sobre o estado do jogo e sempre existe um vencedor (ou seja, não há empate). Apesar de terem uma definição simples, esses jogos escondem uma estrutura matemática rica que nos permite determinar o resultado de configurações extremamente complexas de forma eficiente.

Os jogos combinatórios finitos considerados neste capítulo podem ser modelados como um grafo direcionado acíclico (DAG) em que cada nó representa um estado possível do jogo e cada aresta representa um movimento válido. O jogador que deve mover a partir de um estado sem arestas saindo perde. Embora esse DAG possa ser exponencialmente grande, as ferramentas que vamos desenvolver nesse capítulo permitem analisar o resultado do jogo sem precisar explorá-lo completamente.

1.1 Programação Dinâmica e Jogos

Para analisar jogos combinatórios, classificamos cada estado do jogo em duas categorias: estados *perdedores* (P-posições) e estados *ganhadores* (N-posições). Um estado é perdedor se o jogador da vez perde com jogo ótimo de ambos os lados, e um estado é ganhador se o jogador da vez vence com jogo ótimo.

A classificação segue três regras simples:

- Um estado terminal, em que nenhum movimento é possível, é uma P-posição: o jogador da vez não tem jogada e perde.
- Um estado é N-posição se existe pelo menos um movimento que leva a uma P-posição.
- Um estado é P-posição se *todos* os movimentos a partir dele levam a N-posições.

Como o jogo é acíclico, todo estado é classificado como exatamente um dos dois tipos: se tivesse um ciclo, seria possível voltar a um estado já visitado e o jogo nunca terminaria. Na prática, implementamos essa classificação como uma programação dinâmica sobre o DAG de estados do jogo.

Problema 1.1.1

(CSES - Stick Game) Dois jogadores se alternam em um jogo. Existe uma pilha com n palitos. Em cada turno, o jogador atual pode remover 1, 2 ou 3 palitos. O jogador que não puder realizar um movimento perde. Quem vence com jogo ótimo?

Os estados do jogo são os inteiros de 0 a n, representando o número de palitos restantes. Definindo $dp[i]$ como `true` se o jogador da vez com i palitos vence, o caso base é $dp[0] =$ `false`: o jogador da vez com 0 palitos não tem jogada e perde. Para os demais estados, o jogador atual vence se existe alguma jogada que leva o adversário a um estado perdedor:

$$dp[i] = \neg\, dp[i-1] \vee \neg\, dp[i-2] \vee \neg\, dp[i-3] \tag{1}$$

O DAG de estados desse jogo para $n = 8$ está representado na Figura 1. Cada nó corresponde a um número de palitos e cada aresta representa uma jogada possível. Perceba que o nó 0 (terminal) não tem arestas saindo e é classificado como P-posição. O nó 1 alcança apenas o nó 0 (P-posição), logo é N-posição. O nó 4 só alcança os nós 1, 2 e 3, todos N-posições, e portanto é P-posição.

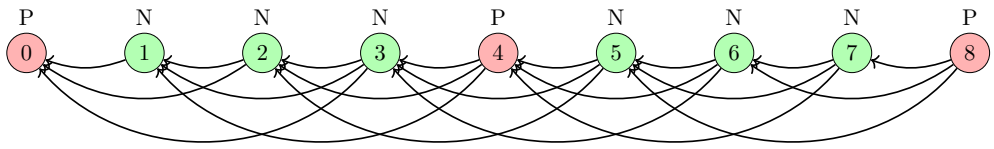


Figura 1: DAG dos estados do jogo de palitos com movimentos $\{1, 2, 3\}$ para $n = 8$. Nós verdes são N-posições (primeiro jogador vence) e nós vermelhos são P-posições (segundo jogador vence). Cada aresta representa um movimento válido.

Calculando os primeiros valores:

i	0	1	2	3	4	5	6	7	8	...
$dp[i]$	P	N	N	N	P	N	N	N	P	...

O padrão se repete: $dp[i] =$ P se e somente se $i \equiv 0 \pmod 4$. Isso faz sentido intuitivamente: se o número de palitos é múltiplo de 4, qualquer que seja a jogada do primeiro jogador (remover 1, 2 ou 3), o segundo jogador completa para o próximo múltiplo de 4, até chegar em 0.

Essa variante pertence à família dos jogos de *subtração*, em que o conjunto S de movimentos válidos é fixo. Para conjuntos S arbitrários, o padrão de P e N-posições pode ser mais complexo e calculado por força bruta:

```
vector<bool> dp(n + 1, false); // false = P-posição
for (int i = 1; i <= n; i++) {
    for (int s : S) { // S é o conjunto de movimentos
        if (i - s >= 0 && !dp[i - s]) {
```

```
            dp[i] = true; // existe jogada para P-posição
            break;
        }
    }
}
if (dp[n]) cout << "primeiro jogador vence\n";
else cout << "segundo jogador vence\n";
```

Código 1.1: Cálculo de P e N-posições para um jogo de subtração genérico.

Problema 1.1.2

(CSES - Nim Game I) Dois jogadores se alternam em um jogo com n pilhas de pedras. Em cada turno, o jogador escolhe uma pilha e remove pelo menos uma pedra dela. O jogador que não puder realizar um movimento perde. Quem vence com jogo ótimo?

Esse jogo é chamado de *Nim* e é um dos mais clássicos da teoria dos jogos. Uma abordagem ingênua seria enumerar todos os estados $(a_1, a_2, \ldots, a_n)$ e calcular a classificação com programação dinâmica. O problema é que o número de estados é $\prod_i (a_i + 1)$, o que pode ser exponencialmente grande.

Felizmente, existe um teorema notável que resolve o Nim em $O(n)$:

$$\text{O estado } (a_1, a_2, \ldots, a_n) \text{ é P-posição} \iff a_1 \oplus a_2 \oplus \ldots \oplus a_n = 0 \tag{2}$$

em que $\oplus$ denota o XOR bit a bit. O valor $a_1 \oplus a_2 \oplus \ldots \oplus a_n$ é chamado de *nim-sum*.

Para provar esse teorema, verificamos as três propriedades das P e N-posições:

1. O estado terminal $(0, 0, \ldots, 0)$ tem nim-sum 0, logo é corretamente classificado como P-posição.

2. A partir de qualquer estado com nim-sum $x \neq 0$, existe um movimento para um estado com nim-sum 0. Seja k o índice do bit mais significativo de x. Existe pelo menos uma pilha a_i com o bit k ativado. Defina $a_i' = a_i \oplus x$. Como o bit k de a_i era 1 e o de x também é 1, o bit k de a_i' é 0, o que garante $a_i' < a_i$, portanto a jogada é válida. O novo nim-sum após a jogada é $(a_1 \oplus \ldots \oplus a_i' \oplus \ldots \oplus a_n) = x \oplus a_i \oplus a_i' = x \oplus a_i \oplus (a_i \oplus x) = 0$.

3. A partir de qualquer estado com nim-sum 0, todo movimento leva a um estado com nim-sum $\neq 0$. Se reduzirmos uma pilha a_i para $a_i' < a_i$, o novo nim-sum é $(0) \oplus a_i \oplus a_i' = a_i \oplus a_i'$. Como $a_i \neq a_i'$, esse valor é não nulo.

A Figura 2 ilustra a classificação de todos os estados do Nim com duas pilhas de tamanho até 5. A diagonal principal (estados com $a = b$) é exatamente o conjunto de P-posições, confirmando que $a \oplus b = 0 \iff a = b$.

A implementação é trivial:

```
int xorSum = 0;
for (int i = 0; i < n; i++) {
    cin >> a[i];
```

```
    xorSum ^= a[i];
}
if (xorSum == 0) cout << "segundo jogador vence\n";
else cout << "primeiro jogador vence\n";
```

Código 1.2: Solução para o Nim clássico.

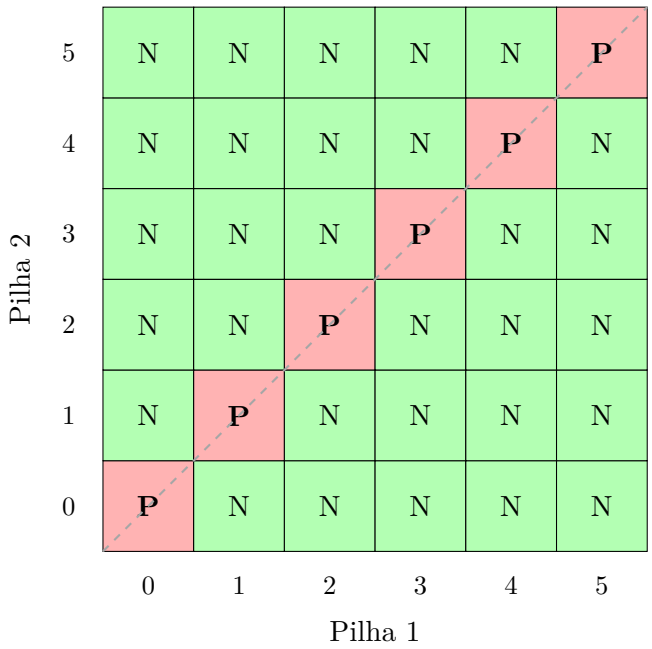


Figura 2: Classificação dos estados do Nim com duas pilhas de tamanho 0 a 5. Um estado (a, b) é P-posição (vermelho) se e somente se $a \oplus b = 0$, ou seja, $a = b$. A diagonal pontilhada corresponde exatamente ao conjunto de P-posições.

Além de determinar o vencedor, o teorema indica a jogada ótima: basta encontrar uma pilha a_i tal que $a_i \oplus x < a_i$, em que x é o nim-sum atual, e reduzi-la para $a_i \oplus x$. Considere o estado $(3, 5, 7)$: o nim-sum é $3 \oplus 5 \oplus 7 = 1$. O bit mais significativo de 1 é o bit 0, que está ativado em 3, 5 e 7. Tomando a pilha 3: $3 \oplus 1 = 2 < 3$. Após remover uma pedra dessa pilha o estado passa a $(2, 5, 7)$, com nim-sum $2 \oplus 5 \oplus 7 = 0$, uma P-posição para o adversário.

1.2 Sprague-Grundy

O resultado do Nim pode ser generalizado para jogos combinatórios imparciais finitos. O teorema de Sprague-Grundy afirma que qualquer jogo combinatório é equivalente a uma pilha do Nim com um determinado tamanho. Esse número é chamado de *valor de Grundy*, ou *nimber*, do estado.

Definição e cálculo do valor de Grundy

O valor de Grundy $G(s)$ de um estado s é o *mex* (*minimum excludant*) dos valores de Grundy dos estados alcançáveis a partir de s:

$$G(s) = \text{mex}\{G(s') \mid s' \text{ é alcançável a partir de } s\} \tag{3}$$

O $\text{mex}(S)$ é o menor inteiro não negativo que não pertence ao conjunto S:

$$\begin{array}{ll} \text{mex}(\emptyset) = 0 & \text{mex}(\{0,1,2\}) = 3 \\ \text{mex}(\{0\}) = 1 & \text{mex}(\{0,1,3\}) = 2 \\ \text{mex}(\{1,2\}) = 0 & \text{mex}(\{0,2,4\}) = 1 \end{array}$$

Note que $G(s) = 0$ para estados terminais (mex do conjunto vazio é 0), correspondendo a P-posições. De fato, $G(s) = 0$ se e somente se s é P-posição: os estados com Grundy 0 têm todos os seus sucessores com Grundy ≥ 1, e a partir de qualquer estado de Grundy ≥ 1 existe sempre um movimento para um estado de Grundy 0.

A Figura 3 mostra o mesmo DAG do jogo de palitos com os valores de Grundy anotados em cada nó. Compare com a Figura 1: os nós com $(G = 0)$ são exatamente as P-posições (vermelhos).

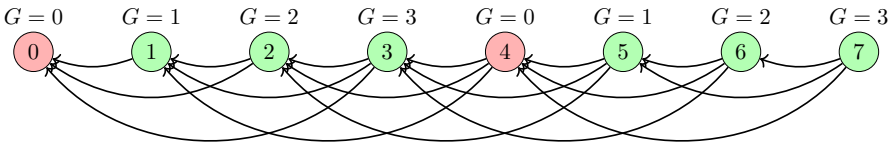


Figura 3: DAG dos estados do jogo de subtração com movimentos $\{1, 2, 3\}$, agora anotado com os valores de Grundy. O valor de Grundy de cada estado é o mex dos Grundy dos estados alcançáveis por uma aresta. Os nós com $G = 0$ (vermelho) correspondem às P-posições.

Para o jogo de subtração com $S = \{1, 2, 3\}$, os valores de Grundy calculados são:

Estado s	0	1	2	3	4	5	6	7	8	...
Alcançáveis	$\emptyset$	$\{0\}$	$\{0,1\}$	$\{0,1,2\}$	$\{1,2,3\}$	$\{2,3,4\}$	$\{3,4,5\}$	$\{4,5,6\}$	$\{5,6,7\}$	
$G(s)$	0	1	2	3	0	1	2	3	0	...

Soma de jogos e o teorema de Sprague-Grundy

A grande utilidade do valor de Grundy aparece quando o jogo é a *soma* de jogos independentes. Formalmente, se o jogo J é composto por subjogos independentes $J_1, J_2, \ldots, J_k$, em que em cada turno o jogador escolhe um dos subjogos e realiza uma jogada nele, então o valor de Grundy da posição combinada é:

$$G(J) = G(J_1) \oplus G(J_2) \oplus \ldots \oplus G(J_k) \tag{4}$$

Esse é o teorema de Sprague-Grundy. A prova segue verificando as mesmas três propriedades de P e N-posições usadas para o Nim: o estado com todos os subjogos em Grundy 0 é terminal; a partir de qualquer estado com XOR não nulo existe um movimento que zera o XOR; e a partir de qualquer estado com XOR zero todo movimento produz XOR não nulo. Os detalhes são análogos à prova do Nim.

O Nim clássico é um caso particular do teorema: cada pilha de tamanho a é um subjogo independente, e $G(\text{pilha de tamanho } a) = a$. Isso pode ser verificado diretamente pela definição, pois os estados alcançáveis de uma pilha de tamanho a são $\{0, 1, \ldots, a-1\}$, cujo mex é a.

Problema 1.2.1

(CSES - Nim Game II) Dois jogadores se alternam em um jogo com n pilhas de pedras. Em cada turno, o jogador escolhe uma pilha e remove 1, 2 ou 3 pedras dela. O jogador que não puder realizar um movimento perde. Quem vence com jogo ótimo?

Esse jogo é a soma de n jogos independentes, um por pilha. O valor de Grundy de uma pilha de tamanho a é $a \bmod 4$, como calculado na tabela acima. Pelo teorema de Sprague-Grundy, o primeiro jogador perde se e somente se

$$(a_1 \bmod 4) \oplus (a_2 \bmod 4) \oplus \ldots \oplus (a_n \bmod 4) = 0 \tag{5}$$

Problema 1.2.2

Dois jogadores jogam em um grafo direcionado acíclico com n vértices e m arestas. Existe uma ficha no vértice 1. Em cada turno, o jogador deve mover a ficha por uma aresta. Se um jogador não puder mover, ele perde. Quem vence com jogo ótimo?

Nesse problema, o próprio grafo dado é o DAG de estados do jogo: cada vértice é um estado e cada aresta é uma jogada. Esse tipo de problema é chamado de *jogo da ficha em DAG*. Para cada vértice v, o valor de Grundy é:

$$G(v) = \text{mex}\{G(u) \mid (v, u) \in E\} \tag{6}$$

Como o grafo é acíclico, calculamos os valores de Grundy em ordem topológica usando memoização. O código abaixo serve como template para os dois problemas desta seção:

```
vector<int> grundy(n + 1, -1);

function<int(int)> calc = [&](int v) -> int {
    if (grundy[v] != -1) return grundy[v];
    set<int> alcancaveis;
    for (int u : adj[v])
        alcancaveis.insert(calc(u));
    int mex = 0;
    while (alcancaveis.count(mex)) mex++;
    return grundy[v] = mex;
};
```

Código 1.3: Template: cálculo do valor de Grundy em um DAG com memoização.

Para o problema de uma única ficha, basta chamar `calc(1)` e verificar se `grundy[1] == 0`. A memoização garante que cada vértice é processado exatamente uma vez. Um vértice de grau de saída d insere d elementos em um conjunto e calcula o mex, que é no máximo d. Somando sobre todos os vértices, a complexidade total é $O((n+m)\log m)$.

Problema 1.2.3

Dois jogadores jogam com n pilhas de pedras. Em cada turno, o jogador escolhe uma pilha de tamanho $s > 1$ e a divide em duas pilhas de tamanhos a e b com $a + b = s$, $a \geq 1$ e $b \geq 1$. O jogador que não puder realizar um movimento perde (quando todas as pilhas têm tamanho 1). Quem vence?

O jogo é a soma de n subjogos independentes, um por pilha. Precisamos calcular $G(s)$ para cada tamanho de pilha s. Uma pilha de tamanho 1 é terminal: $G(1) = \text{mex}(\emptyset) = 0$. Para $s \geq 2$, as divisões alcançáveis são todos os pares (a, b) com $a+b = s$ e $a \leq b$. O estado resultante é a soma de dois subjogos com valor de Grundy $G(a) \oplus G(b)$:

```
grundy[1] = 0;
for (int s = 2; s <= maxN; s++) {
    set<int> alcancaveis;
    for (int a = 1; a <= s / 2; a++) {
        alcancaveis.insert(grundy[a] ^ grundy[s - a]);
    }
    int mex = 0;
    while (alcancaveis.count(mex)) mex++;
    grundy[s] = mex;
}
```

Código 1.4: Cálculo do valor de Grundy para o jogo de divisão de pilhas.

Os primeiros valores calculados são:

s	1	2	3	4	5	6	7	8	9	10
$G(s)$	0	1	0	1	0	1	0	1	0	1

Nesse caso o padrão é simples: $G(s) = (s-1) \bmod 2$, ou seja, pilhas de tamanho par têm Grundy 1 e pilhas de tamanho ímpar têm Grundy 0. Para entender por quê, note que ao dividir s em (a, b) com $a+b = s$ e $a \leq b$, se s é ímpar então a e b têm paridades opostas, logo $G(a) \oplus G(b) = 0 \oplus 1 = 1$ em todos os casos, e o mex de $\{1\}$ é 0. Se s é par então a e b têm a mesma paridade, logo $G(a) \oplus G(b) = 0$ em todos os casos, e o mex de $\{0\}$ é 1.

Para determinar o vencedor com n pilhas, basta calcular o XOR dos valores de Grundy de cada pilha. O primeiro jogador perde se e somente se o número de pilhas de tamanho par é par. Por exemplo, com pilhas de tamanho 3 e 4: $G(3) \oplus G(4) = 0 \oplus 1 = 1 \neq 0$, portanto o primeiro jogador vence. Com pilhas de tamanho 3 e 5: $G(3) \oplus G(5) = 0 \oplus 0 = 0$, portanto o segundo jogador vence.

1.3 Variações de Nim

Há diversas variantes do Nim, cada uma com uma análise própria. Nessa seção vamos estudar três variantes clássicas.

Nim Misère

Problema 1.3.1

(Nim Misère) Idêntico ao Nim clássico, mas agora o jogador que pegar a última pedra *perde*.

No Nim Misère a análise é quase idêntica à do Nim clássico. A única diferença relevante ocorre nos estados finais do jogo. O resultado completo é:

- Se todas as pilhas têm tamanho ≤ 1: o primeiro jogador perde se e somente se o número de pilhas de tamanho 1 é ímpar.
- Caso contrário (existe pelo menos uma pilha de tamanho ≥ 2): o primeiro jogador perde se e somente se o nim-sum é 0.

Enquanto houver alguma pilha de tamanho ≥ 2, o jogador com posição vencedora segue a estratégia do Nim clássico, mantendo o nim-sum zero após cada jogada. Ao chegar ao estado em que todas as pilhas têm tamanho 0 ou 1, esse jogador sempre pode controlar se o número de pilhas de tamanho 1 restantes será par ou ímpar, escolhendo esvaziar ou não a última pilha de tamanho maior que 1. Se o número final de pilhas de tamanho 1 for par, o adversário irá pegar a última pedra e perderá, pois os dois jogadores se alternam retirando exatamente uma pedra por turno.

```
int xorSum = 0, maxPilha = 0, contUm = 0;
for (int i = 0; i < n; i++) {
    xorSum ^= a[i];
    maxPilha = max(maxPilha, a[i]);
    if (a[i] == 1) contUm++;
}
bool perde;
if (maxPilha == 1)
    perde = (contUm % 2 == 1); // número ímpar de pilhas de 1 =
        perde
else
    perde = (xorSum == 0);
cout << (perde ? "segundo" : "primeiro") << " jogador vence\n";
```

Código 1.5: Solução para Nim Misère.

Staircase Nim

Problema 1.3.2

(CSES - Staircase Nim) Dois jogadores jogam em uma escada com n degraus numerados de 1 a n. Cada degrau i contém a_i pedras. Em cada turno, o jogador escolhe um degrau $i \geq 2$ e move qualquer quantidade de pedras (pelo menos 1) do degrau i para o degrau $i-1$. O degrau 0 é o chão e pedras que chegam lá são removidas do jogo. Quem não puder mover perde.

O resultado para o Staircase Nim é o seguinte: o primeiro jogador perde se e somente se o XOR do número de pedras nos degraus *ímpares* é zero:

$$a_1 \oplus a_3 \oplus a_5 \oplus \ldots = 0 \tag{7}$$

A Figura 4 ilustra um exemplo com 4 degraus.

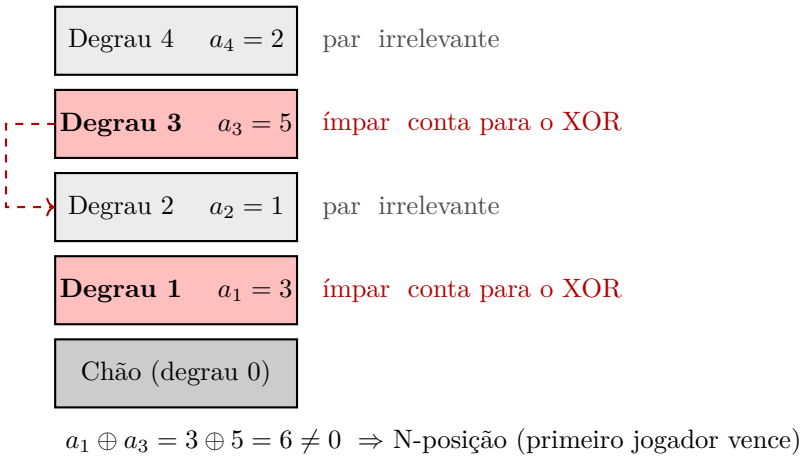


Figura 4: Ilustração do Staircase Nim com 4 degraus. Os degraus ímpares (vermelho) são os únicos relevantes para o nim-sum. Mover pedras de um degrau par para um ímpar pode ser desfeito pelo adversário na jogada seguinte, tornando pedras em degraus pares irrelevantes.

Para entender o porquê, considere o que acontece quando um jogador move pedras de um degrau *par* para o degrau ímpar logo abaixo. O adversário pode simplesmente mover essas mesmas pedras do degrau ímpar para o degrau par logo abaixo, neutralizando completamente o efeito sobre o XOR dos degraus ímpares. Portanto, qualquer jogada em degrau par pode ser desfeita pelo adversário em um único movimento. O jogo se reduz ao Nim clássico jogado apenas nos degraus ímpares.

```
int xorSum = 0;
for (int i = 1; i <= n; i += 2) // apenas degraus ímpares
    xorSum ^= a[i];
if (xorSum == 0) cout << "segundo jogador vence\n";
```

```
else cout << "primeiro jogador vence\n";
```

Código 1.6: Solução para Staircase Nim.

k-Nim

Problema 1.3.3

Dois jogadores jogam o Nim clássico com n pilhas, mas agora cada jogador pode remover pedras de no máximo k pilhas distintas por turno. Quem não puder mover perde.

Para o k-Nim, vale o seguinte teorema: o segundo jogador vence se e somente se, para todo bit de posição (b) em todas as pilhas, a soma desses bits é divisível por $k+1$.

Representando cada a_i em binário, a condição de derrota do primeiro jogador é:

$$\forall b \in \{0,1,2,\ldots\} : \quad \sum_{i=1}^{n} \left(\lfloor a_i / 2^b \rfloor \bmod 2 \right) \equiv 0 \pmod{k+1} \tag{8}$$

Para $k = 1$ essa condição se reduz exatamente ao nim-sum ser zero, recuperando o Nim clássico. A prova do caso geral segue o mesmo esquema de três itens: (1) o estado terminal satisfaz a condição; (2) a partir de qualquer posição que viola alguma condição, sempre existe um conjunto de no máximo k movimentos que restaura todas as condições, com a construção análoga à do Nim clássico trabalhando bit a bit; (3) a partir de qualquer posição que satisfaz todas as condições, qualquer conjunto de no máximo k movimentos viola pelo menos uma condição, pois cada movimento altera pelo menos um bit de alguma pilha e a soma de no máximo k alterações binárias não pode manter a divisibilidade por $k+1$ em todos os bits simultaneamente. A prova completa pode ser encontrada em [1].

```
bool perdendo = true;
for (int b = 0; b < 30 && perdendo; b++) {
    int soma = 0;
    for (int i = 0; i < n; i++)
        soma += (a[i] >> b) & 1;
    if (soma % (k + 1) != 0) perdendo = false;
}
cout << (perdendo ? "segundo" : "primeiro") << " jogador vence\n";
```

Código 1.7: Solução para k-Nim.

Resumo das condições de P-posição

A tabela abaixo resume as condições de P-posição para as variantes estudadas:

Jogo	**Condição de P-posição**	**Complexidade**
Nim clássico	$a_1 \oplus \ldots \oplus a_n = 0$	$O(n)$
Nim Misère	XOR $= 0$ (se existe pilha ≥ 2); ou #pilhas de tamanho 1 ímpar (se todas ≤ 1)	$O(n)$
Staircase Nim	$a_1 \oplus a_3 \oplus a_5 \oplus \ldots = 0$	$O(n)$
k-Nim	$\forall b : \sum_i a_{i,b} \equiv 0 \pmod{k+1}$	$O(n \log \max a_i)$

1.4 Aplicação: jogo de divisores

Nos exemplos anteriores, o DAG de estados era dado explicitamente pelo problema. Em vários problemas de Programação Competitiva, porém, a dificuldade está em identificar qual DAG subjacente representa o jogo e como calcular os valores de Grundy de forma eficiente para esse DAG. Vamos estudar um problema que ilustra esse processo.

Problema 1.4.1

Dois jogadores jogam em uma sequência de n inteiros $a_1, a_2, \ldots, a_n$. Em cada turno, o jogador escolhe um índice i e substitui a_i por qualquer divisor estrito de a_i (ou seja, qualquer divisor d com $1 \leq d < a_i$). O jogador que não puder realizar um movimento (quando todos os elementos forem 1) perde.
$1 \leq n \leq 10^5$, $1 \leq a_i \leq 10^6$.

Cada elemento a_i da sequência é um subjogo independente: o jogador escolhe um elemento da sequência e o substitui por um divisor estrito. O DAG de estados de cada subjogo tem os divisores de a_i como nós e arestas de x para cada divisor estrito de x. O jogo é a soma desses n subjogos, logo pelo teorema de Sprague-Grundy o primeiro jogador perde se e somente se $G(a_1) \oplus G(a_2) \oplus \ldots \oplus G(a_n) = 0$.

Resta calcular $G(v)$ para cada valor possível v. O estado terminal é $v = 1$, com $G(1) = 0$. Para $v > 1$, os estados alcançáveis são todos os divisores estritos de v. Usando a fatoração em primos $v = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, podemos notar que a sequência de movimentos em cada subjogo corresponde a reduzir os expoentes dos fatores primos.

Mais precisamente, pode-se mostrar que $G(v) = \Omega(v)$, onde $\Omega(v) = e_1 + e_2 + \ldots + e_r$ é o número de fatores primos de v contados com multiplicidade (também chamado de *função de Liouville aditiva*). A prova é por indução: os estados alcançáveis a partir de v são todos os divisores estritos d de v, que satisfazem $\Omega(d) < \Omega(v)$. Portanto, o conjunto de valores de Grundy dos sucessores é $\{0, 1, \ldots, \Omega(v) - 1\}$, e o mex é $\Omega(v)$.

```
// Crivo para calcular o menor fator primo de cada número
vector<int> spf(maxN, 0); // smallest prime factor
for (int i = 2; i < maxN; i++)
    if (spf[i] == 0)
        for (int j = i; j < maxN; j += i)
            if (spf[j] == 0) spf[j] = i;

// Calcular Omega(v) = número de fatores primos com multiplicidade
```

```cpp
auto omega = [&](int v) -> int {
    int cnt = 0;
    while (v > 1) { v /= spf[v]; cnt++; }
    return cnt;
};

int xorSum = 0;
for (int i = 0; i < n; i++) {
    int a; cin >> a;
    xorSum ^= omega(a);
}
cout << (xorSum == 0 ? "segundo" : "primeiro") << " jogador vence\n"
    ;
```

Código 1.8: Solução para o jogo de divisores.

Esse problema é um bom exemplo de como o teorema de Sprague-Grundy resolve jogos não triviais: identificamos os subjogos independentes, calculamos o valor de Grundy de cada um usando uma propriedade matemática do problema, e aplicamos o XOR para determinar o vencedor.

Exercícios

(5.1) [Project Euler - 400]

Uma árvore de Fibonacci é uma árvore binária definida recursivamente como: $T(0)$ é a árvore vazia; $T(1)$ é a árvore com apenas um nó; $T(k)$ é a árvore cujo nó raiz tem $T(k-1)$ como filho esquerdo e $T(k-2)$ como filho direito.

Dois jogadores jogam nessa árvore. Em cada turno, o jogador escolhe um nó e remove esse nó junto com toda a subárvore enraizada nele. O jogador que for forçado a remover a raiz da árvore inteira perde.

Seja $f(k)$ o número de jogadas vencedoras do primeiro jogador no primeiro turno ao jogar em $T(k)$. Por exemplo, $f(5) = 1$ e $f(10) = 17$. Encontre $f(10000)$ e imprima os últimos 18 dígitos da resposta.

(5.2) [CSES - Coin Piles]

Você tem duas pilhas de moedas com a e b moedas cada. Em cada turno, o jogador escolhe uma das duas opções: remover 2 moedas de uma pilha e 1 da outra, ou remover 1 moeda de cada pilha. O jogador que não puder realizar um movimento perde. Quem vence com jogo ótimo?

$1 \leq a, b \leq 10^9$.

(5.3) [CSES - Staircase Nim]

Dois jogadores jogam em uma escada com n degraus. Cada degrau i contém a_i pedras. Em cada turno, o jogador escolhe um degrau $i \geq 2$ e move qualquer

quantidade de pedras (pelo menos 1) do degrau i para o degrau $i-1$. Pedras no degrau 0 são removidas. Quem não puder mover perde?

$1 \leq n \leq 2 \cdot 10^5$, $0 \leq a_i \leq 10^9$.

(5.4) [CSES - Grundy's Game]

Dois jogadores alternam turnos em um jogo com uma pilha de n pedras. Em cada turno, o jogador escolhe uma pilha com pelo menos 2 pedras e a divide em duas pilhas de tamanhos *distintos*. O jogador que não puder realizar um movimento perde.

$1 \leq n \leq 10^6$.

(5.5) [Project Euler - 306: Paper-strip Game]

(5.6) [Codeforces - 1312F]

(5.7) [CodeChef - FDIVGAME]

2 Combinatória e Probabilidade

Nessa seção vamos explorar tópicos mais avançados de combinatória. Cobrimos os números de Catalan, contagem de estruturas em grafos via sequência de Prüfer, o Lema de Burnside para contar objetos com simetrias, e uma introdução a funções geradoras.

2.1 Números de Catalan

Os números de Catalan são uma sequência de inteiros positivos que aparecem surpreendentemente em muitos problemas de contagem combinatória. O n-ésimo número de Catalan é denotado por C_n e pode ser calculado pela fórmula

$$C_n = \frac{1}{n+1}\binom{2n}{n} \tag{9}$$

Os primeiros valores dessa sequência são $1, 1, 2, 5, 14, 42, 132, 429, \ldots$

Problema 2.1.1

(CSES - Bracket Sequences I) Quantas sequências de n pares de parênteses são válidas? Uma sequência é válida se cada parêntese de abertura possui um correspondente de fechamento e os pares não se cruzam. Por exemplo, para $n = 3$, as 5 sequências válidas são: `((()))`, `(()())`, `(())()`, `()(())`, `()()()`.

Esse é o exemplo clássico dos números de Catalan: a resposta para n pares de parênteses é exatamente C_n. Para entender o porquê, vamos pensar em como construir uma sequência válida. Uma sequência de n pares pode ser vista como um abre-parênteses inicial, depois uma sequência válida de k pares que está "dentro" desse parêntese, depois o fecha-parênteses correspondente, e por fim uma sequência válida com os $n-1-k$ pares restantes. Isso nos dá a recorrência

$$C_n = \sum_{k=0}^{n-1} C_k \cdot C_{n-1-k}, \quad C_0 = 1 \tag{10}$$

Essa recorrência também aparece no problema das árvores binárias: o número de árvores binárias ordenadas com n nós onde cada nó tem exatamente dois filhos ordenados (esquerdo e direito), cada um podendo ser uma subárvore não vazia ou uma folha externa vazia é C_n. Aqui contamos os nós internos (não-folha), e ao escolher a raiz, temos k nós internos na subárvore esquerda e $n-1-k$ na direita, para k variando de 0 a $n-1$.

Para provar que $C_n = \frac{1}{n+1}\binom{2n}{n}$, usamos uma bijeção elegante com caminhos em grade. Uma sequência de parênteses com n aberturas e n fechamentos pode ser vista como um

caminho em grade de $(0,0)$ a $(2n,0)$ com passos $+1$ (abre) e -1 (fecha). A sequência é válida se e somente se o caminho nunca fica abaixo de $y=0$.

O número total de caminhos de $(0,0)$ a $(2n,0)$ é $\binom{2n}{n}$. Para contar os inválidos, usamos o *Lema da Reflexão*, ilustrado na figura 5.

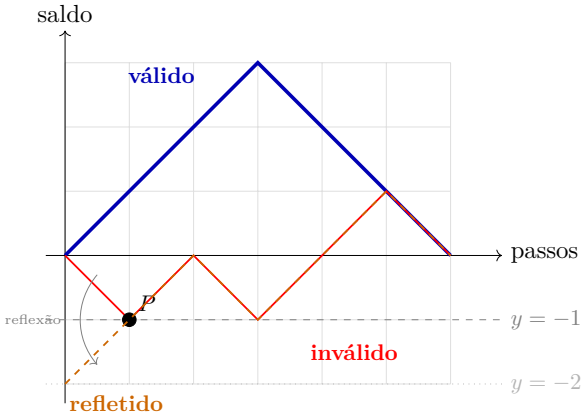


Figura 5: Lema da Reflexão para $n=3$. O caminho azul é válido (saldo sempre ≥ 0). O caminho vermelho é inválido: toca $y=-1$ pela primeira vez no ponto $P=(1,-1)$. O caminho laranja tracejado é o refletido: o prefixo até P é espelhado em torno de $y=-1$, mapeando $(0,0)$ para $(0,-2)$, e o sufixo a partir de P permanece inalterado. Existe uma bijeção entre caminhos inválidos de $(0,0)$ a $(2n,0)$ e todos os caminhos de $(0,-2)$ a $(2n,0)$.

Um caminho inválido necessariamente toca $y=-1$ em algum momento; seja P o primeiro ponto em que isso ocorre. Refletimos o *prefixo* do caminho — a parte de $(0,0)$ até P — em torno da reta $y=-1$. Como P está em $y=-1$, o ponto de partida $(0,0)$ é mapeado para $(0,-2)$, e o sufixo do caminho (de P até $(2n,0)$) permanece inalterado. Isso cria uma bijeção entre caminhos inválidos de $(0,0)$ a $(2n,0)$ e *todos* os caminhos de $(0,-2)$ a $(2n,0)$.

Para contar esses caminhos, note que um caminho de $(0,-2)$ a $(2n,0)$ tem comprimento $2n$ e deve subir 2 unidades no total. Chamando de U o número de passos $+1$ e D o número de passos -1, temos o sistema

$$\begin{aligned} U + D &= 2n \\ U - D &= 2 \end{aligned} \tag{11}$$

cuja solução é $U=n+1$ e $D=n-1$. Logo há exatamente $\binom{2n}{n+1}$ tais caminhos. Como $\binom{2n}{n+1}=\binom{2n}{n-1}$, podemos escrever o resultado de qualquer das duas formas. Portanto,

$$C_n = \binom{2n}{n} - \binom{2n}{n-1} = \frac{1}{n+1}\binom{2n}{n} \tag{12}$$

Outros problemas cujas respostas são os números de Catalan incluem: triangulações de um polígono convexo de $n+2$ lados, número de árvores binárias ordenadas com $n+1$ folhas, e número de caminhos em grade de $(0,0)$ a (n,n) que não cruzam a diagonal principal.

Problema 2.1.2

Dado n, calcule $C_n \bmod (10^9 + 7)$.
$1 \leq n \leq 10^6$.

A resposta é $C_n = \frac{1}{n+1}\binom{2n}{n}$. Para calculá-la de forma eficiente, basta usar o pré-cálculo de fatoriais e inversos que vimos na seção de fundamentos:

```
// Assumindo fat[] e inv[] pré-calculados até 2*MAXN
ll catalan(int n) {
    if (n < 0) return 0;
    // C_n = (2n)! / ((n+1)! * n!)
    return mul(fat[2*n], mul(inv[n+1], inv[n]));
}
```

Código 2.1: Cálculo de $C_n \bmod p$ em $O(1)$ após pré-computação.

Vale verificar os primeiros valores à mão. Para $n = 3$: $C_3 = \frac{1}{4}\binom{6}{3} = \frac{20}{4} = 5$, que bate com as 5 sequências válidas `((()))`, `(()())`, `(())()`, `()(())`, `()()()`, ilustradas como caminhos em grade na figura 6.

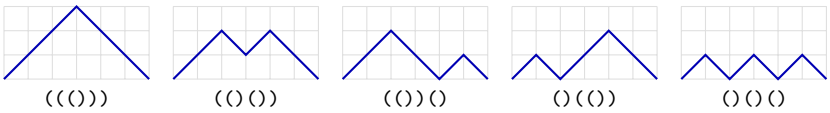


Figura 6: As $C_3 = 5$ sequências de parênteses válidas com 3 pares, representadas como caminhos em grade de $(0, 0)$ a $(6, 0)$. Cada passo para cima corresponde a "(" e cada passo para baixo a ")". Todos os caminhos permanecem em $y \geq 0$.

Problema 2.1.3

(CSES - Bracket Sequences II) Sua tarefa é calcular o número de sequências de parênteses válidas de comprimento n quando um prefixo da sequência é dado.
Imprima a resposta módulo $10^9 + 7$.
$1 \leq n \leq 10^6$.

Uma sequência total de comprimento n é válida se e somente se n é par e o saldo nunca fica negativo. Chamemos de s o saldo do prefixo dado (número de "(" menos número de ")") após processar os k caracteres. Se $s < 0$ em qualquer ponto do prefixo, ou se n for ímpar, a resposta é 0 imediatamente.

Caso contrário, queremos contar os sufixos de comprimento $m = n - k$ que (i) nunca fazem o saldo total cair abaixo de 0, e (ii) terminam com saldo total 0. Como o prefixo já contribui com saldo $s \geq 0$, precisamos de um sufixo que parta de saldo s, nunca desça abaixo de 0, e termine em saldo 0. Isso é equivalente a contar caminhos de $(0, s)$ a $(m, 0)$ em grade que nunca tocam $y = -1$.

Pelo mesmo argumento do Lema da Reflexão, ilustrado na figura 7, um caminho de $(0, s)$ a $(m, 0)$ usa U passos $+1$ e D passos -1 com $U + D = m$ e $U - D = -s$, logo $U = (m - s)/2$. O total de caminhos é $\binom{m}{(m-s)/2}$, e os inválidos bijetam com caminhos de $(0, -s - 2)$ a $(m, 0)$, que são $\binom{m}{(m-s)/2-1}$.

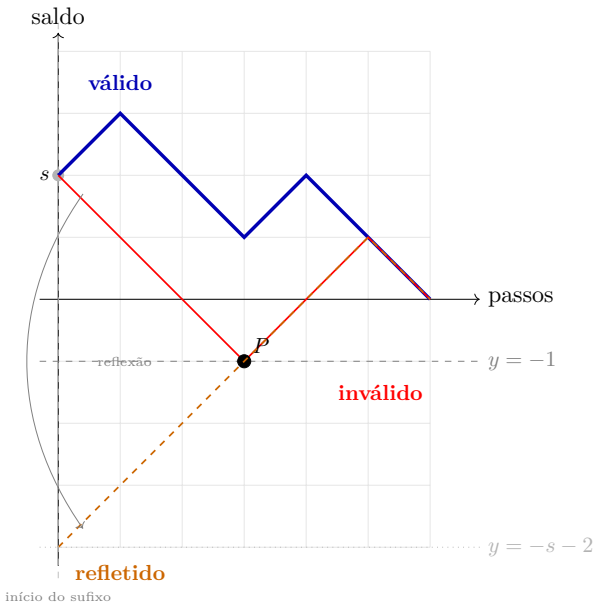


Figura 7: Lema da Reflexão para o Bracket Sequences II, com $s = 2$ e $m = 6$. Os caminhos partem de $(0, s)$ e devem chegar a $(m, 0)$ sem tocar $y = -1$. O caminho azul é válido. O caminho vermelho é inválido: toca $y = -1$ pela primeira vez em $P = (3, -1)$. O caminho laranja é o refletido: o prefixo até P é espelhado em torno de $y = -1$, mapeando $(0, s)$ para $(0, -s-2)$, e o sufixo a partir de P permanece igual. O número de caminhos inválidos de $(0, s)$ a $(m, 0)$ é portanto igual ao número total de caminhos de $(0, -s-2)$ a $(m, 0)$.

Portanto a resposta é

$$\binom{m}{(m-s)/2} - \binom{m}{(m-s)/2-1}, \qquad m = n - k \tag{13}$$

desde que $(m - s)$ seja par e não negativo; caso contrário a resposta é 0.

```
int n; cin >> n;
string prefix; cin >> prefix;
int k = prefix.size();
int saldo = 0;
bool invalido = false;
for (char c : prefix) {
    saldo += (c == '(' ? 1 : -1);
    if (saldo < 0) { invalido = true; break; }
}
```

```cpp
int m = n - k;
if (invalido || (m - saldo) < 0 || (m - saldo) % 2 != 0) {
    cout << 0 << "\n";
} else {
    int U = (m - saldo) / 2;
    ll ans = (nck(m, U) - nck(m, U - 1) + MOD) % MOD;
    cout << ans << "\n";
}
```

Código 2.2: Solução em $O(n)$ para Bracket Sequences II usando o Lema da Reflexão.

Essa solução roda em $O(n)$ após o pré-cálculo de fatoriais. Note que para um prefixo vazio ($k = 0$), o saldo é 0 e a fórmula se reduz a $\binom{n}{n/2} - \binom{n}{n/2-1} = C_{n/2}$, recuperando o número de Catalan como esperado.

2.2 Variáveis Aleatórias Indicadoras e Esperança

Nessa seção exploramos uma técnica central em probabilidade discreta: decompor uma variável aleatória complexa em uma soma de *variáveis indicadoras* variáveis que valem 1 se um certo evento ocorre e 0 caso contrário. Pela linearidade da esperança, o valor esperado da soma é a soma dos valores esperados individuais, e cada indicador tem esperança igual à probabilidade do evento correspondente. Essa decomposição frequentemente transforma um problema difícil em vários subproblemas simples.

Problema 2.2.1

(Clássico) Em um torneio de eliminação simples com $n = 2^k$ jogadores, dois jogadores específicos A e B são sorteados aleatoriamente no chaveamento. Qual a probabilidade de A e B se encontrarem na final?
Assuma que qualquer jogador vence qualquer partida com probabilidade $\frac{1}{2}$.

Para resolver esse problema, calculamos a probabilidade usando probabilidade condicional em duas etapas: primeiro a chance de A e B caírem em lados opostos do chaveamento, e depois a chance de ambos chegarem à final dado que estão em lados opostos.

Etapa 1: probabilidade de ficarem em lados opostos. O chaveamento divide os n jogadores em dois grupos de $\frac{n}{2}$. Fixado A em alguma posição, há $n-1$ posições restantes para B, das quais $\frac{n}{2}$ estão no lado oposto ao de A. Portanto,

$$P(\text{lados opostos}) = \frac{n/2}{n-1} \tag{14}$$

Etapa 2: probabilidade de ambos chegarem à final dado que estão em lados opostos. Para chegar à final, cada jogador deve vencer $k - 1 = \log_2(n) - 1$ partidas consecutivas, cada uma com probabilidade $\frac{1}{2}$. Como as trajetórias de A e B são independentes (estão em lados opostos),

$$P(\text{ambos chegam à final} \mid \text{lados opostos}) = \frac{1}{2^{k-1}} \cdot \frac{1}{2^{k-1}} = \frac{4}{n^2} \tag{15}$$

Combinando as duas etapas:

$$P(\text{se encontram na final}) = \frac{n/2}{n-1} \cdot \frac{4}{n^2} = \frac{2}{n(n-1)} \tag{16}$$

Para $n = 8$, a probabilidade é $\frac{2}{8 \cdot 7} = \frac{1}{28} \approx 3.6\%$, bem menor do que a intuição inicial poderia sugerir.

Problema 2.2.2

(Clássico - Problema do Colecionador de Cupons) Um álbum tem n figurinhas distintas. Cada vez que você compra um pacote, recebe uma figurinha escolhida aleatoriamente e uniformemente entre as n. Qual o número esperado de pacotes que você precisa comprar para completar o álbum?

Dividimos o processo em n fases. Na fase k (após ter $k-1$ figurinhas distintas), precisamos comprar pacotes até conseguir uma figurinha inédita. Nesse momento, há exatamente $n - k + 1$ figurinhas que ainda não temos, então a probabilidade de qualquer pacote ser inédito é $p = \frac{n-k+1}{n}$. O número de pacotes nessa fase segue uma distribuição geométrica de parâmetro p, cujo valor esperado é $\frac{1}{p} = \frac{n}{n-k+1}$.

Pela linearidade da esperança, o número total esperado de pacotes é a soma das esperanças de cada fase:

$$E[T] = \sum_{k=1}^{n} \frac{n}{n-k+1} = n \sum_{j=1}^{n} \frac{1}{j} = n \cdot H_n \tag{17}$$

onde $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ é o n-ésimo número harmônico. Para $n = 100$, são necessários em média $100 \cdot H_{100} \approx 519$ pacotes.

Esse resultado tem uma interpretação computacional importante: o número esperado de amostras aleatórias com repetição necessárias para cobrir um universo de n elementos é $\Theta(n \log n)$.

Problema 2.2.3

Você joga um dado de 6 faces repetidamente. Qual o valor esperado de jogadas até que você tenha visto todos os 6 valores pelo menos uma vez?

Pelo resultado anterior com $n = 6$:

$$E[T] = 6\left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6}\right) = 6 \cdot \frac{49}{20} = \frac{147}{10} = 14.7 \tag{18}$$

Problema 2.2.4

Vasya tem n cartas com números de 1 a n em ordem aleatória. Ele as vira uma por uma. Dizemos que a i-ésima carta é *notável* se ela for maior que todas as cartas anteriores. A primeira carta é sempre notável.
Qual o valor esperado do número de cartas notáveis?

Definimos X_i como a variável aleatória indicadora de que a i-ésima carta é notável. A carta i é notável se ela for o máximo entre as i primeiras cartas, o que acontece com probabilidade $\frac{1}{i}$ (pois qualquer das i cartas pode ser o máximo).

Pela linearidade da esperança:

$$E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \frac{1}{i} = H_n \tag{19}$$

Novamente o número harmônico! Para $n = 1000$, o número esperado de cartas notáveis é $H_{1000} \approx 7.49$.

Problema 2.2.5

(Clássico - Passeio Aleatório) Uma formiga começa no vértice 0 de uma reta e a cada segundo se move uma posição para a esquerda ou direita com igual probabilidade. Qual a probabilidade de a formiga estar na origem após exatamente $2n$ passos?

Para retornar à origem em $2n$ passos, a formiga deve dar exatamente n passos para a direita e n para a esquerda. O número de sequências possíveis de $2n$ passos é 2^{2n}, e as que resultam em retorno à origem são $\binom{2n}{n}$. Portanto,

$$P(\text{retorno em } 2n \text{ passos}) = \frac{\binom{2n}{n}}{4^n} \sim \frac{1}{\sqrt{\pi n}} \tag{20}$$

onde a aproximação segue da fórmula de Stirling. Um resultado famoso do passeio aleatório é que em uma dimensão, a formiga retorna à origem com probabilidade 1 (o passeio é recorrente).

Vale notar a conexão com os números de Catalan que vimos anteriormente. Se contarmos apenas os caminhos de comprimento $2n$ que retornam à origem *sem nunca ter saldo negativo* ou seja, a formiga nunca vai para $y < 0$ esses caminhos correspondem exatamente às sequências de parênteses válidas com n aberturas e n fechamentos. Pelo Lema da Reflexão, o número de tais caminhos é $C_n = \binom{2n}{n} - \binom{2n}{n-1}$. A probabilidade de um passeio aleatório de comprimento $2n$ retornar à origem permanecendo sempre não-negativo é portanto

$$\frac{C_n}{4^n} = \frac{1}{n+1} \cdot \frac{\binom{2n}{n}}{4^n} \tag{21}$$

que para n grande se aproxima de $\frac{1}{(n+1)\sqrt{\pi n}}$, decaindo muito mais rápido do que a probabilidade de simples retorno à origem.

2.3 Contagem em Grafos

Nessa seção vamos estudar como contar estruturas em grafos, com foco em árvores rotuladas e a poderosa ferramenta da sequência de Prüfer.

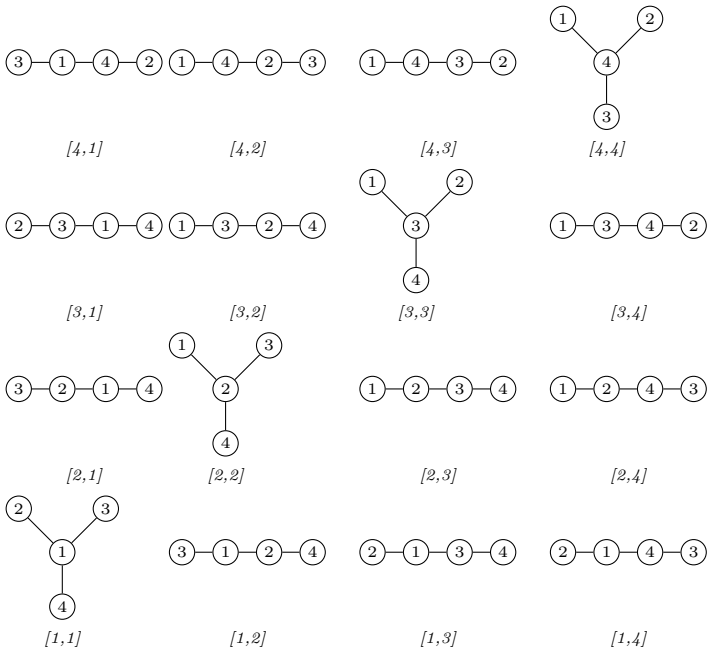
2.3.1 Fórmula de Cayley e Sequência de Prüfer

Já vimos na seção anterior a fórmula de Cayley para florestas. O caso especial mais famoso é o seguinte resultado:

Teorema de Cayley: O número de árvores rotuladas com vértices $\{1, 2, \ldots, n\}$ é n^{n-2}.

A prova desse resultado usa a *sequência de Prüfer*, que construiremos a seguir. A ideia é mostrar uma bijeção entre árvores rotuladas com n vértices e sequências de comprimento $n-2$ com elementos em $\{1, \ldots, n\}$: como há exatamente n^{n-2} tais sequências, o teorema segue diretamente.

Por exemplo, para $n = 4$, temos $4^2 = 16$ árvores distintas. Elas estão todas ilustradas abaixo, com sua respectiva sequência de Prüfer:



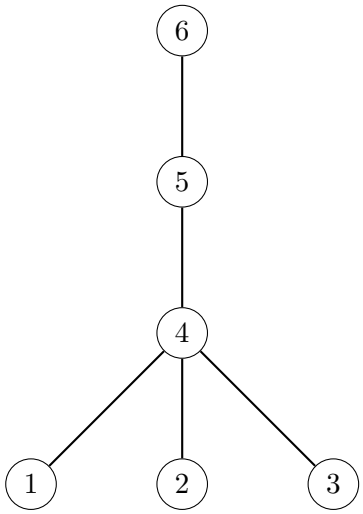
Construindo a sequência de Prüfer a partir de uma árvore:

Dada uma árvore com n vértices, repetimos o seguinte processo $n-2$ vezes: encontre a folha (vértice de grau 1) de menor número, adicione o número de seu único vizinho à sequência, e remova essa folha da árvore. A sequência resultante tem comprimento $n-2$ e é a sequência de Prüfer da árvore. A figura 8 mostra um exemplo com 6 vértices.

```cpp
vector<int> prufer_encode(vector<vector<int>>& adj, int n) {
    vector<int> deg(n + 1, 0), seq;
    for (int i = 1; i <= n; i++) deg[i] = adj[i].size();
    priority_queue<int, vector<int>, greater<int>> leaves;
    for (int i = 1; i <= n; i++) if (deg[i] == 1) leaves.push(i);
    for (int step = 0; step < n - 2; step++) {
        int leaf = leaves.top(); leaves.pop();
        int parent = -1;
```

```
        for (int x : adj[leaf]) if (deg[x] > 0) { parent = x; break; }
        seq.push_back(parent);
        deg[leaf] = 0;
        deg[parent]--;
        if (deg[parent] == 1) leaves.push(parent);
    }
    return seq;
}
```

Código 2.3: Construção da sequência de Prüfer a partir de uma árvore.



Sequência de Prüfer: [4, 4, 4, 5]

Figura 8: Árvore com 6 vértices e sequência de Prüfer $[4, 4, 4, 5]$.

Uma propriedade fundamental da sequência de Prüfer é que os vértices de grau 1 (folhas) nunca aparecem nela. Isso porque uma folha só é removida quando é a menor folha disponível, e nesse momento seu único vizinho que é adicionado à sequência. A única exceção seria quando vértice n (o maior vértice) tem grau 1, mas ele nunca aparece na sequência.

Dessa observação segue uma propriedade mais forte: o grau de cada vértice v na árvore é exatamente $1 + a$, onde a é o número de vezes que v aparece na sequência de Prüfer. Isso porque v é removido da árvore em um único momento (quando se torna folha) e cada vez que aparece na sequência corresponde a uma aresta com um filho que foi removido antes dele. Na figura 8, por exemplo, o vértice 4 aparece 3 vezes na sequência $[4, 4, 4, 5]$, logo $\deg(4) = 1 + 3 = 4$; o vértice 5 aparece 1 vez, logo $\deg(5) = 1 + 1 = 2$; e os vértices 1, 2, 3 e 6 não aparecem, confirmando que são folhas com grau 1. Assim, a sequência de Prüfer carrega toda a informação de grau da árvore, o que garante que a decodificação é única: dada uma sequência, existe exatamente uma árvore que a gera.

Reconstruindo a árvore a partir da sequência de Prüfer:

Dada uma sequência de Prüfer de comprimento $n-2$, reconstruímos a árvore de forma análoga: mantemos o conjunto de candidatos a folha (vértices que não aparecem mais na sequência restante). A cada passo, o menor candidato a folha é conectado ao próximo elemento da sequência, e ambos são removidos de seus respectivos conjuntos.

```cpp
vector<pair<int,int>> prufer_decode(vector<int>& seq, int n) {
    vector<int> deg(n + 1, 1);
    for (int x : seq) deg[x]++;
    priority_queue<int, vector<int>, greater<int>> leaves;
    for (int i = 1; i <= n; i++) if (deg[i] == 1) leaves.push(i);
    vector<pair<int,int>> edges;
    for (int x : seq) {
        int leaf = leaves.top(); leaves.pop();
        edges.push_back({leaf, x});
        deg[x]--;
        if (deg[x] == 1) leaves.push(x);
    }
    // última aresta conecta os dois vértices restantes
    int u = leaves.top(); leaves.pop();
    int v = leaves.top(); leaves.pop();
    edges.push_back({u, v});
    return edges;
}
```

Código 2.4: Reconstrução da árvore a partir da sequência de Prüfer.

Como a codificação e a decodificação são inversas uma da outra, temos uma bijeção entre o conjunto de árvores rotuladas com n vértices e o conjunto de sequências de comprimento $n-2$ sobre $\{1,\ldots,n\}$. Como há exatamente n^{n-2} tais sequências, o Teorema de Cayley está provado.

Problema 2.3.1

Quantas árvores rotuladas com n vértices existem tal que os vértices $v_1, v_2, \ldots, v_k$ têm graus $d_1, d_2, \ldots, d_k$ respectivamente?

Pela propriedade da sequência de Prüfer, o vértice v_i deve aparecer exatamente $d_i - 1$ vezes na sequência de comprimento $n-2$. Sendo $S = \sum_{i=1}^{k}(d_i-1)$, isso equivale ao total de posições que os k vértices ocupam na sequência. Assim, queremos contar as sequências em que cada v_i ocupa exatamente $d_i - 1$ posições e as $n-2-S$ posições restantes são preenchidas livremente.

Para contar essas sequências, primeiro escolhemos as S posições e depois permutamos os k vértices nelas (com repetição):

$$\binom{n-2}{S} \cdot \frac{S!}{\prod_{i=1}^{k}(d_i-1)!} = \frac{(n-2)!}{\prod_{i=1}^{k}(d_i-1)!\,(n-2-S)!} \tag{22}$$

Cada uma das $n-2-S$ posições livres pode ser preenchida por qualquer dos $n-k$

vértices sem grau fixado, de forma independente, dando $(n-k)^{n-2-S}$ possibilidades. O número total de árvores é portanto

$$\frac{(n-2)!}{\prod_{i=1}^{k}(d_i-1)!\,(n-2-S)!}\cdot(n-k)^{n-2-S}, \qquad S=\sum_{i=1}^{k}(d_i-1) \tag{23}$$

desde que $S \le n-2$ e todos os $d_i \ge 1$. Como verificação, tomemos $n=5$, $k=1$, $d_1=2$: temos $S=1$ e a fórmula dá $\frac{3!}{1!\cdot 2!}\cdot 4^2 = 3\cdot 16 = 48$.

2.3.2 Contagem de Árvores Enraizadas

Uma árvore enraizada é uma árvore em que um vértice especial é designado como raiz. Qualquer árvore não enraizada com n vértices pode ser enraizada de n formas diferentes (uma para cada escolha de raiz). Portanto, o número de árvores enraizadas rotuladas com n vértices é $n \cdot n^{n-2} = n^{n-1}$.

Problema 2.3.2

(Clássico) Quantas árvores rotuladas com n vértices existem tal que o vértice 1 e o vértice 2 são adjacentes e $\deg(1) = 1$?

Queremos árvores onde 1 é folha e seu único vizinho é 2. A ideia é remover o vértice 1 e a aresta $(1,2)$: o que sobra é uma árvore arbitrária sobre os $n-1$ vértices $\{2,3,\ldots,n\}$, onde 2 pode ter qualquer grau. Essa operação é uma bijeção: dado qualquer árvore sobre $\{2,\ldots,n\}$, basta pendurar 1 em 2 para obter uma árvore sobre $\{1,\ldots,n\}$ com $\deg(1)=1$ e 2 como vizinho de 1. Pelo Teorema de Cayley, o número de árvores rotuladas com $n-1$ vértices é $(n-1)^{n-3}$.

Para verificar: com $n=4$, a fórmula dá $(4-1)^{4-3} = 3^1 = 3$. As três árvores rotuladas sobre $\{2,3,4\}$ são distinguidas pelo vértice central: (i) 2 é central, com arestas $\{2,3\}$ e $\{2,4\}$; (ii) 3 é central, com arestas $\{2,3\}$ e $\{3,4\}$; (iii) 4 é central, com arestas $\{2,4\}$ e $\{3,4\}$. Pendurando 1 em 2 em cada caso, obtemos exatamente 3 árvores sobre $\{1,2,3,4\}$ com $\deg(1)=1$ e $(1,2)$ como aresta, confirmando o resultado.

Problema 2.3.3

(Clássico) Você tem um grafo completo com n vértices. Quantas florestas enraizadas rotuladas existem com exatamente k componentes conexas?

Uma floresta com n vértices e k componentes tem necessariamente $n-k$ arestas, propriedade análoga ao fato de que uma árvore com n vértices tem $n-1$ arestas. Queremos contar essas florestas usando a generalização da fórmula de Cayley.

Pela fórmula de Cayley generalizada para florestas, o número de florestas com n vértices e k raízes fixadas é $k \cdot n^{n-k-1}$. No entanto, se não fixamos quais são as k raízes, precisamos escolhê-las: há $\binom{n}{k}$ formas de escolher os k vértices que serão raízes. Portanto, a resposta é

$$\binom{n}{k}\cdot k\cdot n^{n-k-1} \tag{24}$$

Note que para $k = 1$ a fórmula dá $\binom{n}{1} \cdot 1 \cdot n^{n-2} = n^{n-1}$, que é o número de árvores enraizadas rotuladas com n vértices, consistente com o fato de que uma floresta de componente única é uma árvore, e cada uma das n^{n-2} árvores não enraizadas pode ser enraizada em n vértices distintos.

2.4 Lema de Burnside e Teoria de Pólya

Em vários problemas de contagem, dois objetos são considerados iguais se um pode ser obtido do outro por alguma simetria.

2.4.1 Lema de Burnside

Seja G um grupo de simetrias que age sobre um conjunto X de objetos. O Lema de Burnside diz que o número de órbitas (classes de equivalência) de X sob a ação de G é

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g| \tag{25}$$

onde $X^g = \{x \in X : g(x) = x\}$ é o conjunto de elementos de X fixados pela simetria g, ou seja, que não mudam quando aplicamos g.

Em termos práticos: para contar objetos distintos módulo um grupo de simetrias, somamos, para cada simetria g do grupo, quantos objetos são fixados por g, e dividimos pelo tamanho do grupo.

Problema 2.4.1

(CSES - Counting Necklaces) Quantos colares distintos existem com n contas, onde cada conta pode ter k cores possíveis? Dois colares são o mesmo se um pode ser obtido do outro por rotação.

O conjunto X são todos os colares com cores fixas, ou seja, $|X| = k^n$. O grupo de simetrias G são as n rotações possíveis (rotação de $0, 1, 2, \ldots, n-1$ posições).

Para a rotação de r posições, um colar é fixado se e somente se ele é periódico com período $\gcd(r, n)$. Ou seja, as contas se repetem em blocos de tamanho $\gcd(r, n)$, e há $k^{\gcd(r,n)}$ colares fixados por essa rotação.

Pelo Lema de Burnside, o número de colares distintos é

$$\frac{1}{n} \sum_{r=0}^{n-1} k^{\gcd(r,n)} \tag{26}$$

```
ll burnside_necklace(ll n, ll k) {
    ll ans = 0;
    for (ll r = 0; r < n; r++) {
        ans = (ans + modpow(k, __gcd(r, n))) % MOD;
    }
    return ans * modpow(n, MOD - 2) % MOD;
```

```
}
```

Código 2.5: Contagem de colares distintos com n contas e k cores usando Burnside.

Por exemplo, para $n = 4$ e $k = 3$, a rotação 0 fixa $3^4 = 81$ colares, as rotações 1 e 3 fixam $3^1 = 3$ cada, e a rotação 2 fixa $3^2 = 9$.

Total: $(81 + 3 + 9 + 3)/4 = 96/4 = 24$ colares distintos.

Problema 2.4.2

Quantos colares distintos existem com n contas, onde cada conta pode ter k cores possíveis? Dois colares são o mesmo se um pode ser obtido do outro por rotação *ou reflexão*.

Agora o grupo de simetrias inclui tanto as n rotações quanto as n reflexões (o grupo diedral D_n de ordem $2n$).

Para as rotações, o cálculo é idêntico ao anterior: a rotação de r posições fixa $k^{\gcd(r,n)}$ colares.

Para as reflexões, o cálculo depende da paridade de n:

- Se n é ímpar: todas as n reflexões passam por um vértice e o meio da aresta oposta. Cada reflexão fixa uma conta e emparelha as demais, resultando em $k^{(n+1)/2}$ colares fixados por cada reflexão. A contribuição total das reflexões é $n \cdot k^{(n+1)/2}$.
- Se n é par: $n/2$ reflexões passam por dois vértices opostos, fixando $k^{n/2+1}$ colares cada, e $n/2$ reflexões passam por dois pontos médios de arestas opostos, fixando $k^{n/2}$ colares cada. A contribuição total das reflexões é $\frac{n}{2} \cdot k^{n/2+1} + \frac{n}{2} \cdot k^{n/2}$.

Aplicando o Lema de Burnside com o grupo diedral, o número de colares distintos é

$$\frac{1}{2n}\left(\sum_{r=0}^{n-1} k^{\gcd(r,n)} + \text{contribuição das reflexões}\right) \tag{27}$$

Para fixar a ideia, vamos trabalhar o exemplo $n = 4$ e $k = 2$ (cores preta e branca) com a conta completa. O grupo diedral D_4 tem 8 simetrias: 4 rotações e 4 reflexões.

Para as rotações: a rotação 0 fixa $2^4 = 16$ colares, as rotações 90 e 270 fixam $2^1 = 2$ cada, e a rotação 180 fixa $2^2 = 4$ colares.

Para as reflexões, como $n = 4$ é par, as 2 reflexões por eixo que passa por 2 vértices opostos fixam $2^3 = 8$ colares cada, e as 2 reflexões por eixo que passa pelos meios de arestas opostos fixam $2^2 = 4$ colares cada.

Total pelo Lema de Burnside:

$$\frac{1}{8}(16 + 2 + 4 + 2 + 8 + 8 + 4 + 4) = \frac{48}{8} = 6 \tag{28}$$

Os 6 colares distintos com 4 contas pretas (`P`) e brancas (`B`), obtidos pelo Lema de Burnside com o grupo diedral D_4, estão ilustrados na figura 9. Contas em preto = `P` e contas em branco = `B`.

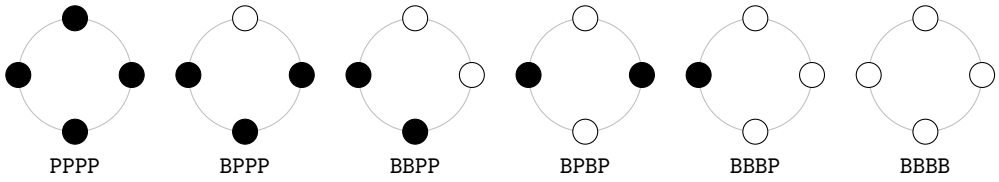


Figura 9: Os 6 colares distintos estão ilustrados acima.

Além disso, vale mencionar que a solução apresentada para o CSES - Counting Necklaces roda em $O(n)$, o que é suficiente para os limites do problema. No entanto, é interessante notar que podemos otimizar ainda mais agrupando os valores de r por valor de $\gcd(r, n)$, o que se torna essencial quando n é muito grande. Para cada divisor d de n, o número de valores de $r \in [0, n-1]$ com $\gcd(r, n) = d$ é $\phi(n/d)$, onde ϕ é a função totiente de Euler. A fórmula se torna

$$\frac{1}{n} \sum_{d|n} \phi(n/d) \cdot k^d \tag{29}$$

Isso reduz o número de termos de n para o número de divisores de n, tornando a solução muito mais eficiente para n grande.

```
ll phi(ll n) {
    ll result = n;
    for (ll p = 2; p * p <= n; p++) {
        if (n % p == 0) {
            while (n % p == 0) n /= p;
            result -= result / p;
        }
    }
    if (n > 1) result -= result / n;
    return result;
}

ll burnside_fast(ll n, ll k) {
    ll ans = 0;
    for (ll d = 1; d * d <= n; d++) {
        if (n % d == 0) {
            ans = (ans + phi(n / d) % MOD * modpow(k, d)) % MOD;
            if (d != n / d)
                ans = (ans + phi(d) % MOD * modpow(k, n / d)) % MOD;
        }
    }
    return ans % MOD * modpow(n, MOD - 2) % MOD;
}
```

Código 2.6: Contagem de colares usando Burnside com agrupamento por gcd.

2.4.2 Teoria de Pólya

A Teoria de Pólya é uma extensão do Lema de Burnside que vai além da simples contagem: ela produz um *polinômio gerador* que descreve como os objetos se distribuem de acordo com suas composições. Isso permite, por exemplo, responder não apenas quantos colares distintos existem com k cores, mas quantos existem com *exatamente* a_1 contas da cor 1, a_2 da cor 2, e assim por diante.

A ferramenta central é o *enumerador cíclico de índices* (cycle index) de um grupo G agindo sobre n posições. Para cada simetria $g \in G$, decompomos as n posições em ciclos disjuntos: se g tem $c_1(g)$ ciclos de comprimento 1, $c_2(g)$ ciclos de comprimento 2, e assim por diante, o cycle index é

$$Z(G; s_1, s_2, \ldots, s_n) = \frac{1}{|G|} \sum_{g \in G} s_1^{c_1(g)} s_2^{c_2(g)} \cdots s_n^{c_n(g)} \tag{30}$$

O **Teorema de Pólya** diz que, se substituirmos cada s_j por $\sum_i w_i^j$, onde w_i é o peso da cor i, o resultado é a função geradora que conta os objetos distintos por composição de cores. Em particular, substituindo $s_j = k$ para todo j (cores intercambiáveis), recuperamos a fórmula de Burnside: o número total de objetos distintos com k cores é $Z(G; k, k, \ldots, k)$.

Para o grupo cíclico C_n das rotações de um colar de n contas, os ciclos da rotação de r posições têm todos o mesmo comprimento $n/\gcd(r, n)$, com $\gcd(r, n)$ ciclos no total. Portanto o cycle index é

$$Z(C_n; s_1, \ldots, s_n) = \frac{1}{n} \sum_{r=0}^{n-1} s_{n/\gcd(r,n)}^{\gcd(r,n)} \tag{31}$$

Problema 2.4.3

Quantos colares distintos com 6 contas existem usando exatamente 2 contas vermelhas, 2 azuis e 2 verdes, considerando apenas rotações?

Pelo Teorema de Pólya, substituímos $s_j = r^j + b^j + g^j$ no cycle index de C_6, onde r, b, g são variáveis formais representando as três cores. A resposta será o coeficiente de $r^2b^2g^2$ no polinômio resultante.

O cycle index de C_6 é calculado agrupando as 6 rotações por tipo de ciclo. A rotação 0 não move nada, gerando 6 ciclos de comprimento 1. As rotações 1 e 5 giram todas as 6 contas num único ciclo. As rotações 2 e 4 criam 2 ciclos de comprimento 3, e a rotação 3 cria 3 ciclos de comprimento 2. Portanto:

$$Z(C_6) = \frac{1}{6}\left(s_1^6 + 2s_6 + 2s_3^2 + s_2^3\right) \tag{32}$$

Agora substituímos $s_j = r^j + b^j + g^j$ em cada termo e extraímos o coeficiente de $r^2b^2g^2$. O único termo que contribui é $s_1^6 = (r+b+g)^6$, cujo coeficiente é o multinomial $\frac{6!}{2!2!2!} = 90$, e $s_2^3 = (r^2+b^2+g^2)^3$, cujo coeficiente é $\frac{3!}{1!1!1!} = 6$. Os termos s_6 e s_3^2 não produzem o monômio $r^2b^2g^2$ pois suas potências são maiores que 2. O resultado é

$$\frac{1}{6}(90+0+0+6) = \frac{96}{6} = 16 \tag{33}$$

Para comparação, a fórmula de Burnside simples com $k = 3$ cores contaria todos os colares independentemente da composição, dando $\frac{729+3+9+27+9+3}{6} = 130$ colares distintos no total. A Teoria de Pólya nos permite extrair a contagem para uma composição específica, o que Burnside sozinho não consegue fazer.

2.5 Introdução a Funções Geradoras

Funções geradoras são uma ferramenta que transforma problemas de contagem em problemas de álgebra de polinômios. A ideia central é representar uma sequência de inteiros $a_0, a_1, a_2, \ldots$ como os coeficientes de uma série de potências formal:

$$A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots = \sum_{n \geq 0} a_n x^n \tag{34}$$

O x aqui é apenas um marcador formal: não nos preocupamos com convergência, e x nunca recebe um valor numérico. O que importa é que operações algébricas sobre funções geradoras correspondem a operações combinatórias sobre as sequências que elas codificam.

A operação mais importante é o produto. Se $A(x)$ conta de quantas formas escolher k objetos de um tipo A, e $B(x)$ conta de quantas formas escolher j objetos de um tipo B, então o coeficiente de x^n no produto $A(x) \cdot B(x)$ conta de quantas formas escolher n objetos no total, somando sobre todas as formas de dividir n entre os dois tipos. Formalmente:

$$[x^n]\, A(x) \cdot B(x) = \sum_{k=0}^{n} a_k \cdot b_{n-k} \tag{35}$$

Essa operação é chamada de *convolução* e ela aparece naturalmente sempre que combinamos escolhas independentes.

2.5.1 Funções Geradoras Ordinárias

As *funções geradoras ordinárias* (OGF) são a forma mais direta: $A(x) = \sum a_n x^n$, onde a_n conta objetos indistinguíveis de tamanho n. O exemplo mais simples é a série geométrica: se podemos usar $0, 1, 2, \ldots$ objetos de um certo tipo, a função geradora correspondente é

$$1 + x + x^2 + \cdots = \frac{1}{1-x} \tag{36}$$

e se cada objeto tem tamanho c, a série vira $\frac{1}{1-x^c}$.

Problema 2.5.1

(CSES: Coin Combinations I) Você tem um sistema monetário com t moedas, cada uma com um valor inteiro positivo. Calcule o número de formas distintas ordenadas de produzir a soma n usando as moedas disponíveis. Duas formas que usam as mesmas

moedas em quantidades iguais, mas em ordens diferentes, são consideradas distintas. Imprima a resposta módulo $10^9 + 7$.

Quando a ordem importa, cada sequência de comprimento ℓ de moedas que soma n é contada. A função geradora correspondente não é um produto independente por moeda: cada posição da sequência pode ser qualquer moeda, e a função geradora de uma única posição é $M(x) = x^{c_1} + x^{c_2} + \cdots + x^{c_t}$. Uma sequência de exatamente ℓ posições tem função geradora $M(x)^\ell$. Somando sobre todos os comprimentos possíveis, chegamos em

$$\sum_{\ell \geq 0} M(x)^\ell = \frac{1}{1 - M(x)} \tag{37}$$

e a resposta é o coeficiente de x^n. Chamemos essa função geradora de $F(x) = \frac{1}{1-M(x)}$. Podemos reescrevê-la como

$$F(x) = 1 + F(x) \cdot M(x) \tag{38}$$

Chamando de $dp[n]$ o coeficiente de x^n em $F(x)$, essa equação nos diz que para formar o termo x^n em $F(x)$, pegamos o termo x^{n-c_i} de $F(x)$ e multiplicamos por x^{c_i}, para cada moeda c_i. O caso base é o 1, que diz que $dp[0] = 1$ (há exatamente uma forma de formar a soma 0: não usar nenhuma moeda). Isso nos dá a recorrência

$$dp[n] = \sum_{i=1}^{t} dp[n - c_i] \tag{39}$$

que se traduz diretamente na DP $dp[j]$ += $dp[j - c_i]$ para cada c_i, iterando j no loop externo e as moedas i no loop interno. Isso garante que todas as ordenações sejam contadas, pois para cada valor de j consideramos todas as moedas possíveis como último elemento da sequência.

Problema 2.5.2

(CSES - Coin Combinations II) Você tem um sistema monetário com t moedas, cada uma com um valor inteiro positivo. Calcule o número de formas distintas não ordenadas de produzir a soma n usando as moedas disponíveis. Duas formas que usam as mesmas moedas em quantidades iguais, mas em ordens diferentes, são consideradas iguais. Imprima a resposta módulo $10^9 + 7$.

A função geradora para as moedas de valor c_i é $\frac{1}{1-x^{c_i}}$, pois podemos usar $0, 1, 2, \ldots$ moedas desse tipo, gerando as somas $0, c_i, 2c_i, 3c_i, \ldots$ Como as escolhas de cada denominação são independentes entre si, a função geradora total é o produto sobre todos os tipos:

$$F(x) = \prod_{i=1}^{t} \frac{1}{1 - x^{c_i}} \tag{40}$$

Para extrair a recorrência, não expandimos esse produto de uma vez. Adicionamos uma moeda por vez. Suponha que $F_{\text{velho}}(x)$ é a função geradora usando as primeiras $k-1$ moedas. Ao incluir a k-ésima moeda c_k, obtemos

$$F_{\text{novo}}(x) = F_{\text{velho}}(x) \cdot \frac{1}{1 - x^{c_k}} \tag{41}$$

$$\Rightarrow F_{\text{novo}}(x) = F_{\text{velho}}(x) + F_{\text{novo}}(x) \cdot x^{c_k} \tag{42}$$

Chamando $dp[n]$ o coeficiente de x^n em $F_{\text{novo}}(x)$:

$$dp_{\text{novo}}[n] = dp_{\text{velho}}[n] + dp_{\text{novo}}[n - c_k] \tag{43}$$

O número de formas de formar a soma n usando as k primeiras moedas é igual ao número de formas sem usar a moeda c_k mais o número de formas de chegar em $n - c_k$ já tendo c_k disponível. Na prática, usamos um único vetor dp e atualizamos in-place, o que exige processar uma moeda inteira por vez: a DP é $dp[j] \mathrel{+}= dp[j - c_i]$ para cada c_i, iterando as moedas no loop externo e j no loop interno. O caso base é $dp[0] = 1$.

2.5.2 Funções Geradoras Exponenciais

Quando os objetos são *distinguíveis*, como elementos de uma permutação onde importa qual elemento está em qual posição, a convolução ordinária não captura o problema corretamente, pois ela não leva em conta que estamos escolhendo quais objetos vão para cada parte. A solução é usar *funções geradoras exponenciais* (EGF), que inserem um fator $n!$ no denominador:

$$\hat{A}(x) = \sum_{n \geq 0} a_n \frac{x^n}{n!} \tag{44}$$

O produto de duas EGFs tem uma propriedade diferente da OGF: o coeficiente de $x^n/n!$ em $\hat{A}(x) \cdot \hat{B}(x)$ é $\sum_{k=0}^{n} \binom{n}{k} a_k b_{n-k}$. O fator $\binom{n}{k}$ aparece naturalmente porque, ao combinar k objetos do tipo A com $n-k$ do tipo B, precisamos escolher *quais* dos n objetos vão para cada parte, o que é exatamente a convolução binomial.

A EGF mais famosa é $e^x = \sum_{n \geq 0} x^n/n!$, que corresponde à sequência $a_n = 1$ e tem a interpretação de *conjunto*: há exatamente 1 conjunto com n elementos distinguíveis quaisquer, independentemente de n.

Problema 2.5.3

Quantas permutações de n elementos existem com exatamente k pontos fixos?

Um ponto fixo é um elemento i tal que $p(i) = i$. Vimos anteriormente que o número de permutações sem nenhum ponto fixo são os *desarranjos* $D_n = n! \sum_{j=0}^{n} (-1)^j/j!$. A EGF dos desarranjos é

$$\hat{D}(x) = \sum_{n\geq 0} D_n \frac{x^n}{n!} = \frac{e^{-x}}{1-x} \tag{45}$$

Para construir uma permutação com exatamente k pontos fixos, escolhemos quais k elementos serão fixos e organizamos os $n-k$ restantes como um desarranjo. A EGF da parte dos pontos fixos é $\frac{x^k}{k!}$ (exatamente 1 forma de fixar k elementos distinguíveis, e o $k!$ vem da normalização). Pelo produto de EGFs, a função geradora de permutações com exatamente k pontos fixos é

$$\frac{x^k}{k!} \cdot \hat{D}(x) \tag{46}$$

e o coeficiente de $x^n/n!$ nesse produto é $\binom{n}{k} D_{n-k}$: escolhemos k dos n elementos para serem pontos fixos e desarranjamos os restantes. Para $k = 0$, recuperamos D_n, consistente com a definição.

2.5.3 Multiplicação de Polinômios com NTT

Em muitos problemas de funções geradoras precisamos calcular produto de dois polinômios de grau $O(n)$. A multiplicação ingênua leva $O(n^2)$, que pode ser ruim. A solução é a Transformada Rápida de Fourier (FFT), que computa o produto em $O(n \log n)$.

A ideia da FFT é explorar que multiplicação de polinômios no domínio dos *coeficientes* equivale a multiplicação ponto a ponto no domínio dos *valores*. Avaliamos $A(x)$ e $B(x)$ em $N = 2^{\lceil \log_2(n+m+1) \rceil}$ pontos escolhidos como raízes da unidade, multiplicamos ponto a ponto em $O(N)$, e depois interpolamos o resultado de volta para os coeficientes, tudo isso em $O(N \log N)$.

Aqui, para evitar problemas de precisão com ponto flutuante, usa-se a *Transformada Teórica de Números* (NTT), que opera inteiramente sobre $\mathbb{Z}_p$ para um primo especial p. O primo 998244353 é o mais comum.

Problema 2.5.4

(CSES - Apples and Bananas) Dados dois polinômios $A(x) = \sum_{i=0}^{n} a_i x^i$ e $B(x) = \sum_{j=0}^{m} b_j x^j$, calcule $C(x) = A(x) \cdot B(x)$.
$n, m \leq 2 \cdot 10^5$.

```
const int MOD = 998244353, g = 3;

ll modpow(ll a, ll b, ll mod) {
    ll res = 1; a %= mod;
    for (; b > 0; b >>= 1) {
        if (b & 1) res = res * a % mod;
        a = a * a % mod;
    }
    return res;
}
```

```
11
12 void ntt(vector<ll>& a, bool inv) {
13     int n = a.size();
14     for (int i = 1, j = 0; i < n; i++) {
15         int bit = n >> 1;
16         for (; j & bit; bit >>= 1) j ^= bit;
17         j ^= bit;
18         if (i < j) swap(a[i], a[j]);
19     }
20     for (int len = 2; len <= n; len <<= 1) {
21         ll w = inv ? modpow(g, MOD-1-(MOD-1)/len, MOD)
22                    : modpow(g, (MOD-1)/len, MOD);
23         for (int i = 0; i < n; i += len) {
24             ll wn = 1;
25             for (int j = 0; j < len/2; j++, wn = wn*w%MOD) {
26                 ll u = a[i+j], v = a[i+j+len/2]*wn%MOD;
27                 a[i+j] = (u+v)%MOD;
28                 a[i+j+len/2] = (u-v+MOD)%MOD;
29             }
30         }
31     }
32     if (inv) {
33         ll n_inv = modpow(n, MOD-2, MOD);
34         for (auto& x : a) x = x * n_inv % MOD;
35     }
36 }
37
38 vector<ll> multiply(vector<ll> a, vector<ll> b) {
39     int result_size = a.size() + b.size() - 1;
40     int n = 1; while (n < result_size) n <<= 1;
41     a.resize(n); b.resize(n);
42     ntt(a, false); ntt(b, false);
43     for (int i = 0; i < n; i++) a[i] = a[i] * b[i] % MOD;
44     ntt(a, true);
45     a.resize(result_size);
46     return a;
47 }
```

Código 2.7: NTT para multiplicação de polinômios módulo um primo.

Exercícios

(3.1) [CSES - Bracket Sequences I]

Quantas sequências de n pares de parênteses são válidas?

Imprima a resposta módulo $10^9 + 7$.

(3.2) [CSES - Bracket Sequences II]

Você recebe uma string que representa o início (prefixo) de uma sequência de parênteses. De quantas formas existem de terminar de preencher essa string para que ela se torne uma sequência de parênteses válida?

Imprima a resposta módulo $10^9 + 7$.

(3.3) [CSES - Counting Necklaces]

Quantos colares distintos existem com n contas e k cores, onde dois colares são o mesmo se um pode ser obtido do outro por rotação?

Imprima a resposta módulo $10^9 + 7$.

(3.4) [CSES - Counting Grids]

Quantas grades $n \times n$ distintas existem onde cada célula pode ser preta ou branca, onde duas grades são iguais se uma pode ser obtida da outra por rotação de 0, 90, 180 ou 270?

Imprima a resposta módulo $10^9 + 7$.

(3.5) [CSES - Counting Tilings]

Quantas formas existem de cobrir um grid $n \times m$ com dominós (1×2 ou 2×1)?

Imprima a resposta módulo $10^9 + 7$.

(3.6) [CSES - Apples and Bananas]

Dados dois polinômios $A(x) = \sum_{i=0}^{n} a_i x^i$ e $B(x) = \sum_{j=0}^{m} b_j x^j$, calcule $C(x) = A(x) \cdot B(x)$.

(3.7) [Exercício]

Quantos colares distintos existem com n contas e k cores, considerando que dois colares são iguais se um pode ser obtido do outro por rotação ou reflexão?

Imprima a resposta módulo $10^9 + 7$.

(3.8) [Exercício]

Um grafo de n vértices e m arestas é dado. Você deve atribuir cores aos vértices tal que, para todo par de vértices u e v conectados por uma aresta, $|cor(u) - cor(v)| = 1$. Quantas formas distintas existem de fazer isso módulo $10^9 + 7$?

(3.9) [CSES - Graph Paths I]

Você tem um grafo direcionado com n vértices e m arestas. Quantos caminhos de comprimento k existem do vértice 1 ao vértice n?

Imprima a resposta módulo $10^9 + 7$.

(3.10) [Exercício]

Você tem uma sequência de n inteiros. Você quer saber quantos subarrays (subconjuntos contíguos de elementos consecutivos) dessa sequência têm a soma de seus elementos divisível por k.

Imprima a resposta módulo $10^9 + 7$.

3 Grafos Direcionados

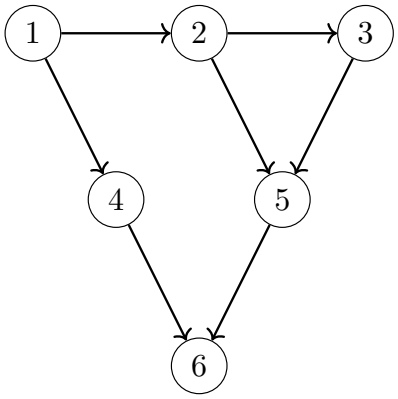
Um grafo direcionado é um grafo em que cada aresta possui uma direção.

3.1 Ordenação Topológica

A ordenação topológica de um DAG é uma listagem de seus vértices tal que, para toda aresta (u, v), o vértice u aparece antes do vértice v na listagem. Ela é definida apenas para DAGs, pois a presença de um ciclo tornaria impossível ordenar os vértices de forma consistente com todas as arestas.

Problema 3.1.1

(CSES - Course Schedule) Você tem n disciplinas e m pré-requisitos. Cada pré-requisito é da forma (a, b), indicando que a disciplina a deve ser cursada antes da disciplina b. Encontre uma ordem para cursar todas as disciplinas respeitando todos os pré-requisitos, ou diga que é impossível.



Uma ordenação topológica: $1 \to 2 \to 3 \to 4 \to 5 \to 6$

Figura 10: Exemplo de DAG com uma ordenação topológica. O vértice 1 possui grau de entrada 0 e é o ponto de partida natural.

Esse problema é um exemplo clássico de ordenação topológica. Modelamos as disciplinas como vértices e os pré-requisitos como arestas direcionadas: se a deve ser cursada antes de b, adicionamos a aresta (a, b). Uma ordenação topológica desse grafo, como a ilustrada na Figura 10, fornece a ordem desejada. Se o grafo contém um ciclo, é impossível satisfazer

todos os pré-requisitos.

Um algoritmo simples para encontrar a ordenação topológica é baseado no conceito de *grau de entrada* de um vértice, que é o número de arestas que chegam nele. Observe que todo DAG possui pelo menos um vértice com grau de entrada 0. Esse vértice pode ser colocado primeiro na ordenação, pois não depende de nenhum outro. Ao removê-lo do grafo, o grafo resultante ainda é um DAG, e podemos repetir o processo.

Esse algoritmo é conhecido como *algoritmo de Kahn* e pode ser implementado de forma eficiente usando uma fila:

```
vector<int> in(n + 1, 0);
vector<vector<int>> adj(n + 1);
for (int i = 0; i < m; i++) {
    int a, b; cin >> a >> b;
    adj[a].push_back(b);
    in[b]++;
}

queue<int> fila;
for (int i = 1; i <= n; i++)
    if (in[i] == 0) fila.push(i);

vector<int> ordem;
while (!fila.empty()) {
    int u = fila.front(); fila.pop();
    ordem.push_back(u);
    for (int v : adj[u]) {
        in[v]--;
        if (in[v] == 0) fila.push(v);
    }
}

if ((int)ordem.size() < n)
    cout << "IMPOSSIBLE\n";
else
    for (int x : ordem) cout << x << "\n";
```

Código 3.1: Ordenação topológica pelo algoritmo de Kahn.

Se ao final do algoritmo todos os n vértices foram processados, a ordem encontrada é uma ordenação topológica válida. Caso contrário, o grafo contém um ciclo e a tarefa é impossível. O algoritmo roda em $O(n + m)$.

Uma alternativa igualmente popular é obter a ordenação topológica a partir de uma DFS: ao terminar de processar um vértice (ou seja, quando a DFS retorna dele), o inserimos no início de uma lista. Ao final, essa lista contém os vértices em ordem topológica. A detecção de ciclos é feita checando se alguma aresta leva a um vértice que ainda está sendo processado (vértice cinza na DFS).

Problema 3.1.2

(CSES - Longest Flight Route) Você tem n cidades e m voos direcionados. Encontre o maior número de cidades em um caminho do vértice 1 ao vértice n, ou diga que tal caminho não existe.

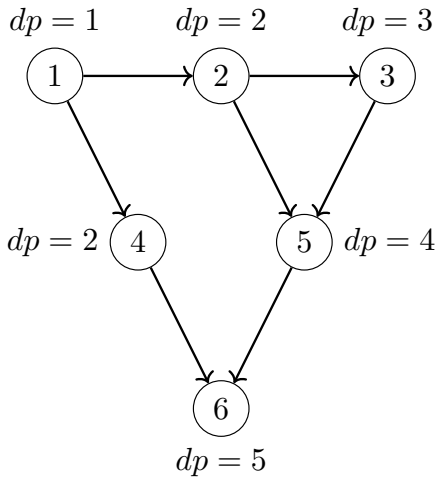


Figura 11: Valores da DP de caminho máximo do vértice 1 ao 6. A resposta é $dp[6] = 5$, correspondendo ao caminho $1 \to 2 \to 3 \to 5 \to 6$.

O grafo dos voos é um DAG (voos formam rotas sem ciclos entre cidades). Definimos $dp[v]$ como o maior número de cidades em um caminho de 1 até v, cujos valores estão ilustrados na Figura 11. O caso base é $dp[1] = 1$, e para os demais vértices:

$$dp[v] = 1 + \max_{(u,v)\in E} dp[u] \tag{47}$$

Essa recorrência só pode ser calculada corretamente se, ao processar v, todos os vértices u com aresta (u, v) já foram processados, o que é exatamente o que a ordenação topológica garante. Ao final, a resposta é $dp[n]$, se n foi alcançado, ou "IMPOSSIBLE" caso contrário.

```
vector<int> dp(n+1, -1);
vector<int> par(n+1, -1);
dp[1] = 1;
for (int u : ordem) { //ordenacao topologica
    if (dp[u] == -1) continue;
    for (int v : adj[u]) {
        if (dp[u] + 1 > dp[v]) {
            dp[v] = dp[u] + 1;
            par[v] = u;
        }
    }
```

```
}
if (dp[n] == -1)
    cout << "IMPOSSIBLE\n";
else {
    cout << dp[n] << "\n";
    vector<int> caminho;
    for (int v = n; v != -1; v = par[v])
        caminho.push_back(v);
    reverse(caminho.begin(), caminho.end());
    for (int x : caminho) cout << x << " ";
}
```

Código 3.2: DP de caminho máximo em DAG usando ordenação topológica.

3.2 Componentes Fortemente Conexas

Em um grafo não direcionado, uma componente conexa é um conjunto maximal de vértices em que todo par de vértices é conectado por algum caminho. Em grafos direcionados, esse conceito se desdobra em dois: componentes fracamente conexas (ignorando as direções das arestas) e componentes *fortemente* conexas.

Uma *Componente Fortemente Conexa* (SCC) é um conjunto maximal de vértices C tal que, para todo par de vértices $u, v \in C$, existe um caminho direcionado de u para v e um caminho direcionado de v para u.

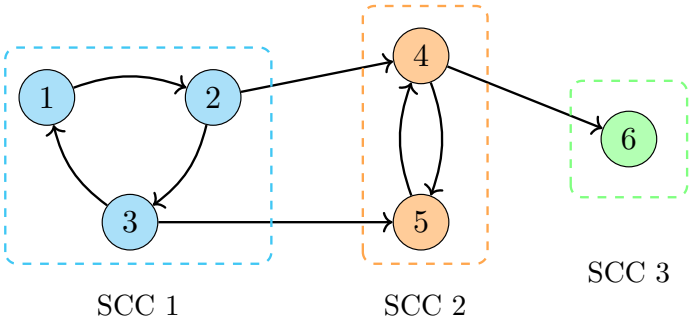


Figura 12: Grafo direcionado com três componentes fortemente conexas destacadas. As arestas entre SCCs distintas formam o grafo condensado, que é sempre um DAG.

No grafo da Figura 12, as SCCs são $\{1, 2, 3\}$, $\{4, 5\}$ e $\{6\}$. Observe que, dentro de cada componente, é possível chegar de qualquer vértice a qualquer outro seguindo as direções das arestas. Já entre componentes distintas, se existe uma aresta de uma componente para outra, não existe caminho de volta.

Uma propriedade fundamental é que, se condensarmos cada SCC em um único super-vértice e adicionarmos arestas entre super-vértices onde havia arestas entre componentes

distintas, o grafo resultante é sempre um DAG. Esse grafo é chamado de *grafo condensado* e é extremamente útil para reduzir problemas em grafos gerais a problemas em DAGs.

Problema 3.2.1

(CSES - Planets and Kingdoms) Você tem n planetas e m rotas espaciais direcionadas. Dois planetas pertencem ao mesmo reino se e somente se existe uma rota do primeiro para o segundo **e** uma rota do segundo para o primeiro (não necessariamente diretas). Atribua a cada planeta um número de reino de 1 a k, em que k é o número total de reinos.

Esse problema é exatamente encontrar as SCCs do grafo. O algoritmo de Kosaraju resolve esse problema em $O(n+m)$ com duas passagens de DFS.

A ideia central do Kosaraju é explorar a observação de que se executarmos uma DFS e registrarmos os vértices em ordem de término de visita, o *último* vértice a terminar pertence a uma SCC que não tem arestas de entrada vindas de outras SCCs no grafo condensado (ou seja, está em uma componente "raiz"). Ao invertermos todas as arestas do grafo, essa componente raiz se torna uma componente "folha", alcançável apenas a partir de si mesma. Assim, ao rodar uma DFS no grafo invertido a partir desse vértice, visitamos exatamente os vértices dessa SCC, e nenhum outro.

O algoritmo executa uma DFS no grafo original registrando os vértices em uma pilha na ordem de término, constrói o grafo transposto G^T com todas as arestas invertidas, e então processa os vértices na ordem inversa à da pilha executando uma DFS em G^T. Cada DFS completa a partir de um vértice não visitado encontra exatamente uma SCC.

```
int n, num_scc = 0;
vector<vector<int>> adj, radj;
vector<bool> visited;
vector<int> order, comp;

void dfs1(int u) {
    visited[u] = true;
    for (int v : adj[u])
        if (!visited[v]) dfs1(v);
    order.push_back(u);
}

void dfs2(int u, int c) {
    comp[u] = c;
    for (int v : radj[u])
        if (comp[v] == -1) dfs2(v, c);
}

int main() {
    // leitura do grafo ...
    adj.assign(n+1, {}); radj.assign(n+1, {});
    visited.assign(n+1, false); comp.assign(n+1, -1);

```

```
    for (int i = 1; i <= n; i++)
        if (!visited[i]) dfs1(i);

    for (int i = (int)order.size()-1; i >= 0; i--)
        if (comp[order[i]] == -1) dfs2(order[i], num_scc++);
}
```

Código 3.3: Algoritmo de Kosaraju para encontrar SCCs.

Ao final, o vetor *comp* atribui a cada vértice o índice de sua SCC. O número total de SCCs é *num_scc*.

Problema 3.2.2

(CSES - Coin Collector) Você tem n salas e m túneis direcionados. Cada sala tem c_i moedas. Você pode começar em qualquer sala e terminar em qualquer sala. Qual o máximo de moedas que você consegue coletar?

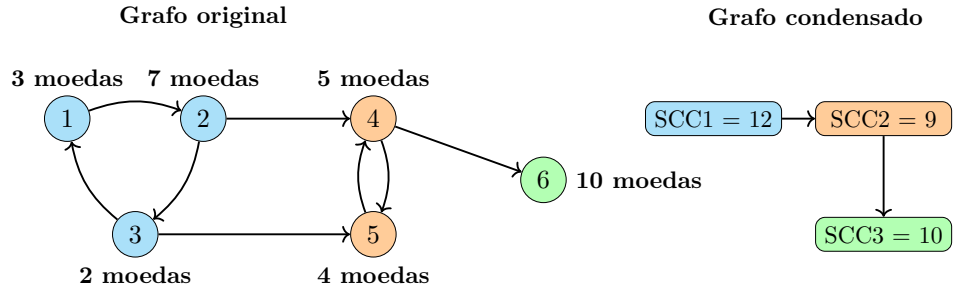


Figura 13: À esquerda, o grafo original com o número de moedas de cada vértice. À direita, o grafo condensado: cada SCC vira um super-vértice com a soma das moedas. A resposta é o caminho de maior soma no DAG condensado.

Note que, se você entrar em uma SCC, pode coletar todas as moedas dela antes de sair, pois dentro de uma SCC é possível alcançar qualquer vértice a partir de qualquer outro. Portanto, o valor de uma SCC é a soma das moedas de todos os seus vértices.

Com isso, condensamos o grafo e cada SCC vira um super-vértice com valor igual à soma das moedas de seus membros. O problema se reduz a encontrar o caminho de maior soma no DAG condensado, o que pode ser resolvido com DP em ordem topológica, como fizemos na seção anterior.

3.3 2-SAT

O problema de satisfatibilidade booleana (SAT) pergunta se existe uma atribuição de valores verdadeiro/falso a variáveis booleanas que satisfaça uma fórmula lógica. O caso geral (3-

SAT) é NP-completo, mas a versão com duas variáveis por cláusula, o *2-SAT*, pode ser resolvida em tempo linear usando SCCs.

Uma instância de 2-SAT tem n variáveis booleanas $x_1, x_2, \ldots, x_n$ e m cláusulas, em que cada cláusula é a disjunção de exatamente dois literais. Um literal é uma variável x_i ou sua negação $\neg x_i$. Por exemplo:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3) \tag{48}$$

Queremos encontrar valores para x_1, x_2, x_3 que tornem toda cláusula verdadeira, ou determinar que isso é impossível.

A redução ao problema de SCCs parte da observação de que uma cláusula $(a \vee b)$ é equivalente a duas implicações: $(\neg a \Rightarrow b)$ e $(\neg b \Rightarrow a)$. Se a for falso, b obrigatoriamente deve ser verdadeiro, e vice-versa.

Construímos então um grafo de implicações: cada variável x_i gera dois nós, um para x_i e outro para $\neg x_i$. Para cada cláusula $(a \vee b)$, adicionamos as arestas $\neg a \to b$ e $\neg b \to a$.

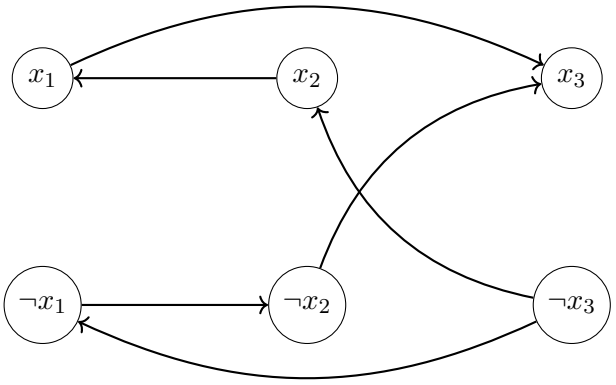


Figura 14: Grafo de implicações para a fórmula $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee x_3)$. Cada cláusula $(a \vee b)$ gera as arestas $\neg a \to b$ e $\neg b \to a$.

A instância é satisfatível se e somente se nenhuma variável x_i está na mesma SCC que $\neg x_i$. Se x_i e $\neg x_i$ estivessem na mesma SCC, a implicação $x_i \Rightarrow \neg x_i$ e $\neg x_i \Rightarrow x_i$ seriam ambas verdadeiras, o que é uma contradição.

Para construir a solução, usamos o grafo condensado. Como ele é um DAG, podemos numerar as SCCs em ordem topológica. Atribuímos x_i = verdadeiro se a SCC de x_i vem depois da SCC de $\neg x_i$ nessa ordem, e x_i = falso caso contrário.

A ideia é que, no grafo condensado, arestas vão de componentes anteriores para posteriores na ordem topológica. Se a SCC de x_i aparece depois da de $\neg x_i$, temos $\neg x_i \Rightarrow x_i$ no DAG condensado (mas não o contrário), então podemos escolher x_i = verdadeiro de forma consistente.

O Kosaraju numera as SCCs de forma que componentes sem predecessores no DAG condensado recebem os menores índices. Portanto, a SCC de x_i aparece "depois" (tem índice

maior) quando recebe mais implicações convergindo para ela, e basta checar se comp$[x_i] >$ comp$[\neg x_i]$.

```cpp
// variaveis de 1 a n
// literal x_i  -> indice i
// literal ~x_i -> indice i + n
int neg_of(int a) { return a > n ? a - n : a + n; }

void add_clause(int a, int b) {
    adj[neg_of(a)].push_back(b);
    radj[b].push_back(neg_of(a));
    adj[neg_of(b)].push_back(a);
    radj[a].push_back(neg_of(b));
}

// apos rodar Kosaraju, verificar e atribuir valores
bool satisfativel = true;
vector<bool> val(n + 1);
for (int i = 1; i <= n; i++) {
    if (comp[i] == comp[i + n]) {
        satisfativel = false;
        break;
    }
    val[i] = comp[i] > comp[i + n];
}
```

Código 3.4: 2-SAT: construção do grafo de implicações e resolução via Kosaraju.

A função `neg_of` retorna o índice da negação de um literal: se o literal é x_i (índice $i \leq n$), retorna $i+n$; se é $\neg x_i$ (índice $i > n$), retorna $i-n$. Para adicionar a cláusula $(a \vee b)$, chamamos `add_clause(i, j+n)` para a cláusula $(x_i \vee \neg x_j)$, por exemplo.

Exercícios

(3.1) [CSES - Game Routes]

Você tem um DAG com n vértices e m arestas. Quantos caminhos existem do vértice 1 ao vértice n? Imprima a resposta módulo $10^9 + 7$.

$1 \leq n \leq 10^5$, $1 \leq m \leq 2 \cdot 10^5$.

(3.3) [CSES - Giant Pizza]

Uma família de n pessoas quer montar uma pizza gigante com m ingredientes. Cada pessoa tem dois pedidos: ela quer que um determinado ingrediente seja incluído ou excluído da pizza. A família precisa atender pelo menos um pedido de cada pessoa.

Determine se existe uma escolha de ingredientes que satisfaça essa condição e, se sim, exiba uma solução.

$1 \leq n \leq 10^5$, $1 \leq m \leq 10^5$.

4 Árvores

4.1 Diâmetro de Árvore

O diâmetro de uma árvore é o maior caminho entre dois vértices quaisquer da árvore. Formalmente, o diâmetro é $\max_{u,v} dist(u, v)$, em que $dist(u, v)$ é o número de arestas no caminho entre u e v.

Problema 4.1.1

(CSES - Tree Diameter) Dada uma árvore de n vértices, encontre o comprimento do seu diâmetro.
$1 \leq n \leq 2 \cdot 10^5$.

Uma observação central para resolver esse problema é o seguinte lema: se fixarmos um vértice qualquer s e encontrarmos o vértice mais distante de s, chamado de u, então u é uma extremidade do diâmetro da árvore.

Para provar esse lema por contradição, suponha que o diâmetro da árvore seja o caminho entre os vértices a e b, com comprimento D, e que u não seja nem a nem b. Sabemos que $dist(s, u) \geq dist(s, a)$ e $dist(s, u) \geq dist(s, b)$, pois u é o mais distante de s.

Como a árvore não tem ciclos, todos os caminhos são únicos. Seja x o vértice onde o caminho de s encontra o diâmetro a-b, e seja j o vértice onde o caminho de u encontra o diâmetro a-b. O vértice j divide o diâmetro em dois lados: um em direção a a e outro em direção a b. O vértice x está obrigatoriamente em um desses lados.

Sem perda de generalidade, assuma que x está entre j e b, ou seja, a ordem no diâmetro é $a \cdots j \cdots x \cdots b$. Nesse caso, o caminho de s até a passa obrigatoriamente por x e por j, de modo que $dist(s, a) = dist(s, x) + dist(x, j) + dist(j, a)$ e portanto $dist(s, j) = dist(s, x) + dist(x, j)$, o que nos dá $dist(s, a) = dist(s, j) + dist(j, a)$. Vamos mostrar que $dist(u, b) \geq D$, contradizendo a maximalidade do diâmetro. Nessa topologia, o caminho de s até u passa obrigatoriamente por j, portanto $dist(s, u) = dist(s, j) + dist(j, u)$, o que nos dá a igualdade exata $dist(u, j) = dist(s, u) - dist(s, j)$. Como também $dist(u, b) = dist(u, j) + dist(j, b)$, temos

$$\begin{aligned} dist(u, b) &= dist(u, j) + dist(j, b) \\ &= dist(s, u) - dist(s, j) + dist(j, b) \\ &\geq dist(s, a) - dist(s, j) + dist(j, b) \\ &= dist(j, a) + dist(j, b) = D \end{aligned} \tag{49}$$

uma contradição. Se x estivesse entre a e j, o argumento seria simétrico, mostrando que

$dist(u, a) \geq D$, o que também contradiz a maximalidade do diâmetro.

Com esse lema, o algoritmo para encontrar o diâmetro fica simples: escolhemos qualquer vértice s, rodamos uma BFS ou DFS para encontrar o vértice u mais distante de s, e então rodamos uma segunda BFS ou DFS a partir de u para encontrar o vértice v mais distante de u. O diâmetro é $dist(u, v)$.

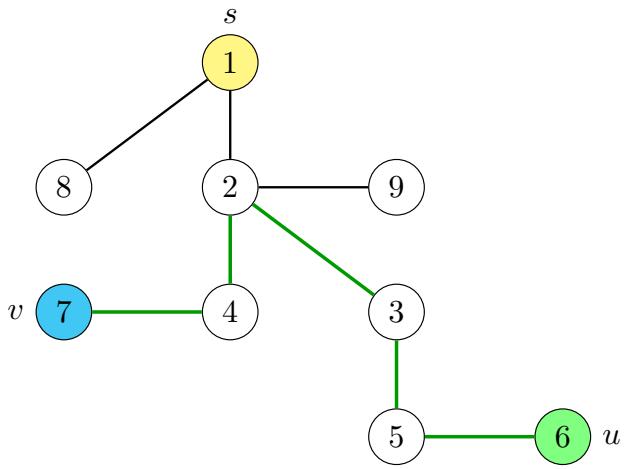


Figura 15: Árvore cujo diâmetro, de comprimento 5, está destacado em verde (caminho de $v = 7$ a $u = 6$).

A figura 15 ilustra esse processo. Começando de $s = 1$, o vértice mais distante é $u = 6$, à distância 4. Rodando novamente a partir de u, o vértice mais distante é $v = 7$, com distância 5. De fato, o caminho $6 \to 5 \to 3 \to 2 \to 4 \to 7$ é o diâmetro da árvore.

```
int farthest(int start, vector<int>& dist, vector<vector<int>>& adj)
    {
    fill(dist.begin(), dist.end(), -1);
    queue<int> q;
    dist[start] = 0;
    q.push(start);
    int node = start;
    while (!q.empty()) {
        int at = q.front(); q.pop();
        for (int x : adj[at]) {
            if (dist[x] == -1) {
                dist[x] = dist[at] + 1;
                q.push(x);
                if (dist[x] > dist[node]) node = x;
            }
        }
    }
    return node;
}
```

```cpp
int main() {
    // ...leitura do grafo e inits
    int u = farthest(1, dist, adj);
    int v = farthest(u, dist, adj);
    cout << dist[v] << "\n"; // diâmetro
    return 0;
}
```

Código 4.1: Algoritmo para encontrar o diâmetro de uma árvore em $O(n)$.

Problema 4.1.2

(CSES - Tree Distances I) Dada uma árvore de n vértices, para cada vértice v encontre $maxdist(v)$, ou seja, a maior distância de v para qualquer outro vértice.
$1 \leq n \leq 2 \cdot 10^5$.

Uma solução ingênua seria rodar uma BFS a partir de cada vértice, o que resulta em complexidade $O(n^2)$. Podemos fazer melhor usando o diâmetro.

Primeiro encontramos as duas extremidades do diâmetro u e v, usando o algoritmo anterior. Em seguida, rodamos uma BFS a partir de u e outra a partir de v, obtendo os vetores du e dv de distâncias. A observação chave é que, para qualquer vértice x, o vértice mais distante de x é sempre u ou v. Portanto:

$$maxdist(x) = \max(du[x], dv[x]) \tag{50}$$

Para provar isso, suponha que exista um vértice w tal que $dist(x, w) > \max(dist(x, u), dist(x, v))$. Pelo mesmo argumento do lema anterior aplicado ao par (x, w), chegaríamos em uma contradição com o fato de u e v serem extremidades do diâmetro.

4.2 Mínimo Ancestral Comum (LCA)

O *Lowest Common Ancestor* (LCA) de dois vértices u e v em uma árvore enraizada é o único vértice w no caminho entre u e v que tem a menor profundidade.

Uma aplicação imediata do LCA é calcular a distância entre dois vértices em uma árvore. Se conhecemos a profundidade de cada vértice e sabemos calcular o LCA, a distância entre u e v é simplesmente

$$dist(u, v) = depth[u] + depth[v] - 2 \cdot depth[lca(u, v)] \tag{51}$$

Isso ocorre pois o caminho de u a v necessariamente passa pelo LCA, e a distância de u ao LCA é $depth[u] - depth[lca(u, v)]$, e analogamente para v.

Problema 4.2.1

(CSES - Company Queries II) Você tem uma árvore de n vértices com raiz no vértice 1. Para cada query (u, v), encontre o LCA de u e v.
$1 \leq n, q \leq 2 \cdot 10^5$.

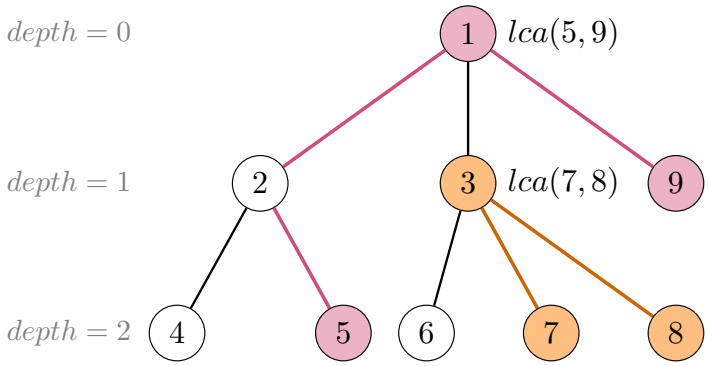


Figura 16: Exemplo de LCA em uma árvore enraizada no vértice 1. O LCA dos vértices 7 e 8 é o vértice 3 (destacado em laranja), enquanto o LCA dos vértices 5 e 9 é o vértice 1, a própria raiz (destacado em roxo).

Uma abordagem ingênua seria subir pelos ancestrais de u e v alternadamente até que se encontrem, o que no pior caso roda em $O(n)$ por query. Para q queries, isso resulta em $O(nq)$, que é inviável para os limites dados.

A solução eficiente usa *binary lifting*. A ideia é precalcular $anc[v][k]$, o 2^k-ésimo ancestral de v. Ou seja, $anc[v][0]$ é o pai de v (1 ancestral), $anc[v][1]$ é o avô de v (2 ancestral), $anc[v][2]$ é o tataravô (4 ancestral), e assim por diante. A relação de recorrência é simplesmente

$$anc[v][k] = anc[anc[v][k-1]][k-1] \tag{52}$$

ou seja, o 2^k-ésimo ancestral de v é o 2^{k-1}-ésimo ancestral do 2^{k-1}-ésimo ancestral de v.

O precálculo é feito com uma DFS em $O(n \log n)$:

```
const int LOG = 18;
int anc[MAXN][LOG], depth[MAXN];
void dfs(int node, int parent, int d) {
    anc[node][0] = parent;
    depth[node] = d;
    for (int k = 1; k < LOG; k++)
        anc[node][k] = anc[anc[node][k-1]][k-1];
    for (int x : v[node])
        if (x != parent) dfs(x, node, d + 1);
}
```

Código 4.2: Precálculo do binary lifting para LCA em $O(n \log n)$.

Note que definimos $anc[raiz][0] = raiz$, o que garante que os saltos além da raiz não saiam dos limites do vetor.

Para responder uma query (u, v), primeiro nivelamos u e v para a mesma profundidade subindo o vértice mais profundo com saltos binários. Depois, subimos os dois simultaneamente usando os saltos precalculados até encontrar o LCA:

```cpp
int lca(int u, int v) {
    if (depth[u] < depth[v]) swap(u, v);
    int diff = depth[u] - depth[v];
    for (int k = 0; k < LOG; k++)
        if ((diff >> k) & 1) u = anc[u][k];
    if (u == v) return u;
    for (int k = LOG - 1; k >= 0; k--)
        if (anc[u][k] != anc[v][k]) {
            u = anc[u][k];
            v = anc[v][k];
        }
    return anc[u][0];
}
```

Código 4.3: Consulta de LCA em $O(\log n)$.

No código 4.3, primeiro igualamos as profundidades de u e v subindo o vértice mais profundo pela representação binária de $diff$. Em seguida, se $u = v$ após o nivelamento, esse vértice é o próprio LCA. Caso contrário, subimos os dois simultaneamente pelos maiores saltos possíveis que ainda não os fazem coincidir, garantindo que ao final ambos sejam filhos do LCA. A complexidade total é $O(n \log n)$ para o precálculo e $O(\log n)$ por query.

Problema 4.2.2

(CSES - Distance Queries) Você tem uma árvore de n vértices e q queries. Cada query é um par (u, v) e você deve responder a distância entre u e v na árvore.
$1 \leq n, q \leq 2 \cdot 10^5$.

Com o binary lifting já implementado, esse problema se resolve diretamente. Calculamos $depth[v]$ durante a DFS de precálculo, e para cada query basta combinar as distâncias à raiz:

```cpp
int dist(int u, int v) {
    return depth[u] + depth[v] - 2 * depth[lca(u, v)];
}
```

Código 4.4: Distância entre dois vértices usando LCA.

O custo de cada query é $O(\log n)$ e o custo total é $O((n + q) \log n)$.

Referências

[1] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for your Mathematical Plays*. A K Peters, 2 edition, 2001.

[2] Daryl Casarotto. Prüfer sequences. VIGRE REU, University of Chicago, 2006. Disponível em: `https://www.math.uchicago.edu/~may/VIGRE/VIGRE2006/PAPERS/Casarotto.pdf`.