Aula 13 · Hashing e Jogos Combinatórios Imparciais

Desafios de Programação

Fernando Kiotheka Victor Alflen

UFPR

31/08/2022

Hashing

O que é *hashing*?

- *Hashing* pode ser traduzido como trituração.
- Uma função de *hashing* é uma função que tem como
  - **Entrada:** Uma sequência de bytes de tamanho variável.
  - **Saída:** uma sequência de bytes de tamanho fixo, o *hash*.
- O resultado pode ser chamado de resumo criptográfico, e é uma "impressão digital" da entrada.
- Existe um limite para a direcionalidade dessa função: Podemos "comprimir" sem perdas até 13 caracteres do alfabeto latino minúsculo por exemplo em um inteiro de 64 bits: $26^{13} < 2^{64}$.
- Uma propriedade boa para usos criptográficos é que seja praticamente impossível conseguir qualquer tipo de pista sobre os dados originais dado o *hash*.
- Porém, estamos preocupados apenas que tenhamos colisões mínimas para nossos algoritmos, a qualidade do *hash* é desprezível.

Mas pra que?

- Comparar inteiros é mais rápido que comparar vários bytes.
- Podemos ter alguma certeza de que $h(x) = h(y) \implies x = y$.
- Assim, podemos resolver alguns problemas:
  - Achar padrões.
  - Calcular o número de substrings diferentes.
  - Calcular o número de substrings palíndromes.
  - Entre outros.

Função de *hashing* polinomial móvel

$$\begin{aligned} h(s) &= s[0] + s[1]p + s[2]p^2 + \ldots + s[n-1]p^{n-1} \pmod{m} \\ &= \sum_{i=0}^{n-1} s[i]p^i \pmod{m}, \end{aligned}$$

onde p e m são números escolhidos.

- É razoável que p seja um número primo próximo ao tamanho do alfabeto. Para 26 caracteres, vamos usar $p = 31$.
- m deve ser grande pois a probabilidade de duas strings colidirem vai ser por volta de $\approx \frac{1}{m}$.
- Seria legal usar $m = 2^{64}$ pois com `unsigned long long`, você obtém esse módulo "de graça". Porém existe um método para gerar colisões para $m = 2^{64}$, então não é recomendado.
- Ao invés disso vamos usar $m = 10^9 + 9$ que permite fazer multiplicações em `long long`.

Implementação da função de *hashing* polinomial móvel

Código `hash.cpp`

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const int P = 31; const int M = 1e9 + 9;
ll polyhash(string const& s) {
    ll h = 0, p = 1;
    for (char c : s) {
        h += (c - 'a' + 1) * p; h %= M;
        p *= P; p %= M;
    }
    return h;
}
int main() {
    string s; while (cin >> s) {
        cout << polyhash(s) << "\n";
    }
}
```

Achando padrões: Algoritmo de Rabin-Karp

- Podemos usar a mesma ideia para achar correspondências usando *hashing*.
- Esse é o algoritmo de Rabin-Karp, e tem complexidade linear igual o KMP.
- A ideia é comparar a hash de cada prefixo no palheiro e comparar com a hash da agulha.

Implementação do Algoritmo de Rabin-Karp

Código `rabinkarp.h`

```
int search(string hay, string ne) {
    int n = hay.size(), m = ne.size();
    vector<ll> ps (max(n, m), 1);
    for (int i = 1; i < max(n, m); i++)
        ps[i] = (ps[i-1] * P) % M;
    vector<ll> h (n+1, 0);
    for (int i = 0; i < n; i++)
        h[i+1] = (h[i] + (hay[i]-'a'+1) * ps[i]) % M;
    ll hne = 0;
    for (int i = 0; i < m; i++)
        hne = (hne + (ne[i]-'a'+1) * ps[i]) % M;
    int c = 0;
    for (int i = 0; i < n - m + 1; i++)
        if ((h[i+m] - h[i] + M) % M == hne * ps[i] % M) {
            c++; cout << hay.substr(0, i) << "[" << ne << "]"
                 << hay.substr(i+m) << "\n";
        }
    return c;
}
```

Implementação do Algoritmo de Rabin-Karp

Código `rabinkarp.cpp`

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
const int P = 31; const int M = 1e9 + 9;
#include "rabinkarp.h"
int main() {
    string hay, ne; while (cin >> hay >> ne)
        cout << search(hay, ne) << "\n";
}
```

Entrada	Saída
banana a abbba bb aybabtu aybabtu	b[a]nana ban[a]na banan[a] 3 a[bb]ba ab[bb]a 2 [aybabtu] 1

Vamos quebrar o *hash*?

- Como falado anteriormente, existem técnicas *anti-hashing*.
- Geralmente isso é mais revelante em competições onde você pode hackear as soluções (que é criar uma entrada que sabota o algoritmo de outra pessoa).
- Uma das técnicas para se acharem colisões envolve o Algoritmo de Floyd: O Algoritmo da Tartaruga e Lebre.

Algoritmo da Tartaruga e Lebre

- Resolve o amplo problema de detecção de ciclos em grafos direcionados
- Consiste em manter dois ponteiros, um lento e outro rápido (tartaruga e lebre)
- Começando da raiz do grafo, a cada iteração a tartaruga dá um passo e a lebre dá dois (x e $2x$)
- Paramos este processo quando os dois ponteiros se encontram
- Finalmente, a tartaruga volta para a raiz e novamente movemos os dois ponteiros (dessa vez com a mesma velocidade x) até que se encontrem no primeiro vértice do ciclo

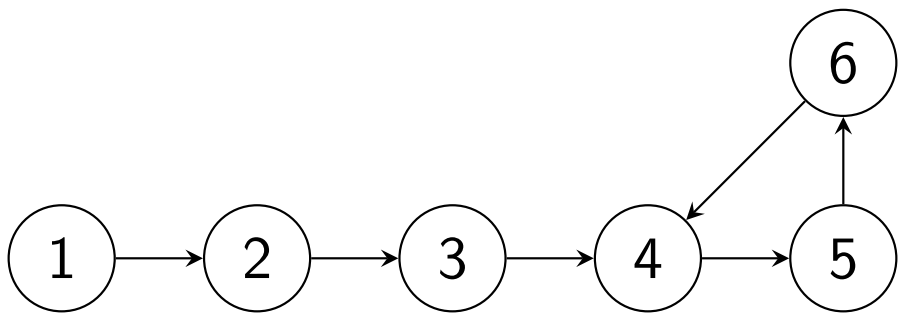
Ideia "pura"

Sejam X, T e L a raiz do grafo, a tartaruga e a lebre, respectivamente. Faremos o seguinte:

- $T \leftarrow \text{prox}(X)$
- $L \leftarrow \text{prox}(\text{prox}(X))$
- Enquanto $T \neq L$, $T \leftarrow \text{prox}(T)$ e $L \leftarrow \text{prox}(\text{prox}(L))$
- $T \leftarrow X$
- Enquanto $T \neq L$, $T \leftarrow \text{prox}(T)$ e $L \leftarrow \text{prox}(L)$

O valor final de T e L é o primeiro vértice no ciclo. Com isso também podemos calcular o tamanho deste ciclo.

Grafo de exemplo



Implementação do algoritmo de Floyd

Código `floyd.h`

```
struct cyc { ll prev, first, len; };
cyc find_cycle(ll start) {
    ll a = start, b = start;
    do {
        a = succ(a);
        b = succ(succ(b));
    } while (a != b);
    a = start;
    ll prev = -1;
    while (a != b) { prev = a; a = succ(a); b = succ(b); }
    ll len = 0;
    do { b = succ(b); len++; } while (a != b);
    return { .prev = prev, .first = a, .len = len };
}
```

Implementação do algoritmo de Floyd (continuado)

Código `floyd.cpp`

```
#include <bits/stdc++.h>
using namespace std; using ll = long long;
vector<int> nxt;
int succ(int x) { return nxt[x]; }
#include "floyd.h"
int main() {
    int n; while (cin >> n) {
        nxt.resize(n+1);
        for (int i = 1; i <= n; ++i) { cin >> nxt[i]; }
        cyc fl = find_cycle(1);
        cout << fl.first << " (" << fl.len << ")\n";
    }
}
```

Implementação do algoritmo de Floyd (continuado)

Entrada	Saída
6 2 3 4 5 6 4 3 2 3 1	4 (3) 1 (3)

Por que funciona?

Digamos que a cauda e o ciclo tenham A e C vértices, respectivamente.

- Rotulemos os vértices da cauda, em ordem, com os valores $-A$, $-A+1$, ..., -1
- Rotulemos os vértices do ciclo com 0, 1, ..., $C-1$
- Temos que $A = kC + r$ para algum $k \geq 0$ e algum $0 \leq r < C$
- Após A iterações, a tartaruga chega ao vértice 0 e a lebre ao vértice r ($2A$ passos, sendo os A primeiros os que a levaram ao vértice 0 e os últimos $A = kC + r = r \mod C$ dentro do ciclo)

Por que funciona? (continuando)

- Quando $r = 0$, a tartaruga e a lebre já estão no mesmo vértice
- Quando $r \neq 0$, fazemos mais $C - r$ iterações: a tartaruga chega ao vértice $C - r$ e a lebre chega ao vértice $r + 2(C - r) = C - r \mod C$ (mesma posição)
- Neste momento, a tartaruga volta para o início da cauda e ambas (tartaruga e lebre) dão $A = r \mod C$ passos
- A tartaruga estava em $-A$ (cauda) e deu A passos, chegando ao vértice 0
- A lebre estava em $C - r$ e deu $A = r \mod C$ passos, chegando ao vértice $(C - r) + r = 0 \mod C$ (mesma posição e início do ciclo)

Aplicando o Algoritmo de Floyd para Colisões

- Vamos escolher nossa função de próximo como sendo:
  - Aplicar um gerador de números pseudoaleatórios no valor dado (que gera sempre o mesmo valor dado uma semente).
  - Em seguida geramos uma string usando esse valor.
  - Fazemos hash usando a nossa função com essa string.
- Devido ao paradoxo do aniversário, a sequência vai eventualmente repetir, e isso acontece em tempo esperado $\mathcal{O}(\sqrt{N})$ (com 64 bits, $\mathcal{O}(\sqrt{2^{64}}) \approx 2^{32} \approx 40 \cdot 10^8$).
- Usando o Algoritmo de Floyd, a gente acha o primeiro ponto onde os valores são iguais, e geralmente os dois valores que levam ao ciclo são diferentes.

Implementação do Floyd para Colisões

Código `floydhash.h`

```
ll polyhash(string const& s) {
    ll h = 0, p = 1;
    for (char c : s) {
        h += (c-'a'+1) * p; h %= M;
        p *= P; p %= M; }
    return h;
}
string gen(ll u) {
    mt19937_64 rnd (u); u = rnd();
    string s;
    while (u) { s += ('a'+(u%26)); u /= 26; }
    return s;
}
ll succ(ll u) {
    return polyhash(gen(u));
}
```

Implementação do Floyd para Colisões (continuado)

Código `floydhash.cpp`

```
#include <bits/stdc++.h>
using namespace std;
using ll = unsigned long long;
const int P = 31; const int M = 1e9 + 9;
#include "floydhash.h"
#include "floyd.h"
int main() {
    ll start; while (cin >> start) {
        cyc c = find_cycle(start);
        if (c.prev == -1) { cout << "failed\n"; continue; }
        ll x = c.prev; ll y = x; ll len = c.len;
        while (len--) { y = succ(y); }
        string a = gen(x), b = gen(y);
        cout << "h(" << a << ")=" << polyhash(a) << "\n";
        cout << "h(" << b << ")=" << polyhash(b) << "\n";
    }
}
```

Implementação do Floyd para Colisões (continuado)

Entrada	Saída
0	h(mjnlbszkmgnfye)=42954140
1	h(tdtakvvbcvngcg)=42954140
2	h(mvrjrhlflkahzg)=125738538
3	h(bddfbuymdnisah)=125738538
4	h(mvrjrhlflkahzg)=125738538
5	h(bddfbuymdnisah)=125738538
	h(pmjpgiewvqkpfd)=372914348
	h(vtfxzjcxiximqc)=372914348
	h(mvrjrhlflkahzg)=125738538
	h(bddfbuymdnisah)=125738538
	h(qvwvskvziptrjd)=511774277
	h(qklaclhcjvtbfg)=511774277

Como remediar?

- Você pode gerar a base (p) aleatoriamente.
- Também as vezes é bom usar dois módulos (calcula um hash em um módulo e outro hash em outro módulo e utiliza os dois para fins de comparação).
- Em geral, é um problema impossível de resolver de maneira absoluta (por isso que geralmente se usam tamanhos maiores que 64 bits para chaves por exemplo, diminuindo as colisões possíveis).

Jogos Combinatórios Imparciais

Jogo imparcial

- Jogo finito.
- Jogo em turnos alternantes entre dois jogadores.
  - Jogo dos quadradinhos não é alternante: Se você fizer um quadrado você pode repetir sua jogada.
- Ações simétricas (a mesma pessoa ia ter as mesmas jogadas).
  - No Xadrez, um jogador só pode mexer as peças brancas e o outro só pode mexer as peças pretas.

Esvazie a pilha

Considere um jogo imparcial onde

- Existe uma pilha com N pedras.
- Os jogadores alternam jogadas entre si.
- Em cada jogada, o jogador da vez escolhe tirar 1, 2 ou 3 pedras da pilha (desde que a pilha tenha um número suficiente de pedras)
- Perde o jogador que não puder tirar pedra nenhuma (isto é, ganha aquele que tirar a última pedra)

Posições vencedoras e posições perdedoras

- Posição perdedora só pode mudar de estado para posição vencedora.
- Posição vencedora pode mudar o estado para alguma posição perdedora.
- No jogo anterior, posições com $X \pmod{3+1} = 0$ pedras são perdedoras e todas as outras são vencedoras.

O jogo de Nim

Consideremos agora o seguinte jogo:

- Há N pilhas de pedras de tamanho variável.
- Dois jogadores alternam jogadas entre si.
- Em cada jogada, o jogador da vez escolhe uma das pilhas e remove dela quantas pedras (no mínimo uma) quiser (desde que a pilha tenha o número de pedras a ser removido).
- Perde quem não tiver jogadas (ganha quem esvaziar a última pilha)

Chamamos este jogo de Jogo de Nim.

Olhando para as pilhas de Nim

Quais são as posições perdedoras e vencedoras no jogo de Nim?

- Posição perdedora: Xor dos tamanhos de pilhas vale 0
- Posição vencedora: Qualquer outra

Observação importante: todos os jogos são independentes (mudar uma pilha não muda nada no estado das outras).

Provando posições perdedoras

Posição é perdedora se e somente se o xor das pilhas é 0.

- Vamos provar que de uma posição perdedora $g = 0$, só podemos ir para uma posição vencedora $g' \neq 0$.
- Se o xor de todas as pilhas é 0,

$$g = p_1 \oplus p_2 \oplus p_3 \oplus \ldots \oplus p_n = 0$$
$$p_2 \oplus p_3 \oplus \ldots \oplus p_n = p_1$$

- Sem perda de generalidade, podemos pegar uma das pilhas, por exemplo a pilha p_1 e reduzir a unidades $(1 \leq a \leq p_1)$.

$$\begin{aligned} g' &= (p_1 - a) \oplus p_2 \oplus p_3 \oplus \ldots \oplus p_n \\ &= (p_1 - a) \oplus p_1 \end{aligned}$$

- Como em xor o único inverso de p_1 é p_1 e $p_1 \neq (p_1 - a)$, concluímos que $g' \neq 0$.

Provando posições vencedoras

Agora temos que provar que é possível ir de uma posição onde $g \neq 0$ para uma posição onde $g' = 0$.

- Seja p a posição do bit mais significativo em g.
- Existe um número ímpar de pilhas onde o bit p está ligado, escolhemos uma pilha u para jogar.
- Nossa jogada será $g' = g \oplus u \oplus t = 0$. Isto porque u é o inverso de u em xor, e t será o nosso novo u.
- Então $t = g \oplus u$, para cumprir a equação.
- Sabemos que $t < u$ pois vamos alterar apenas os bits menores ou iguais a p de u.

Então é possível partir de qualquer posição com soma xor $g \neq 0$ para uma com soma xor $g = 0$ (ou seja, ir de uma posição vencedora para uma perdedora).

Resolvendo o Nim

Código `nim.cpp`

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n; while (cin >> n) {
        int g = 0;
        vector<int> s (n);
        for (int i = 0; i < n; i++) {
            cin >> s[i]; g ^= s[i]; }
        if (g == 0) { cout << "-1\n"; continue; }
        int msb = (1<<__lg(g));
        for (int i = 0; i < n; i++) if (s[i]&msb) {
            int t = s[i]^g;
            cout << s[i]-t << " " << i+1 << "\n";
            break;
        }
    }
}
```

Exemplo de um jogo de Nim

Entrada	Saída
3 3 4 5 3 1 4 2 3 1 2 2 3 0 1 2 3 0 0 1 3 0 0 0	2 1 1 2 1 1 1 3 1 3 -1

Pilha 1	Pilha 2	Pilha 3	Movimento
3	4	5	Início do jogo
1 (-2)	4	5	Computador tira 2 da pilha 1
1	4	2 (-3)	Humano tira 3 da pilha 3
1	3 (-1)	2	Computador tira 1 da pilha 2
1	2 (-1)	2	Humano tira 1 da pilha 2
0 (-1)	2	2	Computador tira 1 da pilha 1
0	1 (-1)	2	Humano tira 1 da pilha 2
0	1	1 (-1)	Computador tira 1 da pilha 3
0	0 (-1)	1	Humano tira 1 da pilha 2
0	0	0 (-1)	Computador tira 1 da pilha 3 e ganha

Nim k-Slow

- Jogador escolhe uma pilha e tira $0 < P \leq K$ elementos dela.
- Posição é a soma xor dos tamanhos das pilhas $\pmod{k+1}$:

$$(a_1 \mod (k+1)) \oplus (a_2 \mod (k+1)) \oplus \ldots \oplus (a_n \mod (k+1))$$

Atenção que $(a_1 \oplus a_2 \oplus \ldots \oplus a_n) \pmod{k+1}$ **não é igual**.

$$\begin{aligned}
(6 \mod 5) \oplus (1 \mod 5) &= 1 \oplus 1 = 0 \\
(6 \oplus 1) &= 7 = 2 \pmod 5 \\
(6 \mod 7) \oplus (19 \mod 7) &= 6 \oplus 5 = 3 \\
(6 \oplus 19) &= 21 = 0 \pmod 7 \\
(6 \mod 5) \oplus (4 \mod 5) &= 1 \oplus 4 = 5 \\
(6 \oplus 4) &= 2 \pmod 5
\end{aligned}$$

Para alguns módulos específicos, o módulo pode ser comutativo com o xor. Para módulo $m = 2^n$, o módulo é equivalente a aplicação de máscara `& (m-1)`, que é comutativa com xor.

Nim monotônico

- Estado só é válido se a i-ésima pilha for menor ou tão alta quanto a $i+1$-ésima (sequência não decrescente).
- Fora isso podemos tirar um número arbitrário de pedras de uma pilha escolhida como no jogo original.
- Para um estado $A = (a_1, a_2, \ldots, a_{2k})$ (número par de pilhas), mapeamos A para um estado $B = (b_1, b_2, \ldots, b_k)$ onde $b_i = a_{2i} - a_{2i-1}$ (estado de diferença).
- Caso o estado A seja ímpar, colocamos mais uma posição 0 no seu início para torná-lo par.
- O estado A é uma posição perdedora no Nim monotônico se, e somente se, B é uma posição perdedora no Nim incrementado

Nim incrementado

- Agora vamos permitir adicionar pedras à pilha, e não só removê-las.
- É importante manter o jogo acíclico (finito), e portanto as regras (independente de quais sejam) devem garantir que os incrementos respeitem isso.
- Isso não muda a definição de posições vencedoras e perdedoras porque na estratégia vencedora é inútil adicionar pedras (se o primeiro jogador adiciona X pedras a uma pilha, o segundo pode remover X pedras da mesma pilha, desfazendo a jogada).

Função mex

- Mínimo excludente.
- Primeiro número natural que não pertence ao conjunto.

$$\text{mex}\{0, 1, 3, 5\} = 2.$$
$$\text{mex}\{1, 3, 5\} = 0.$$

Teorema de Sprague-Grundy

- Vamos encontrar para qualquer estado de um jogo imparcial um estado correspondente em um jogo de Nim incrementado.
- Para um estado e_i a partir do qual podem ser acessados os estados num conjunto E_i, atribuímos a e_i um jogo de Nim com pilha de tamanho X, onde X é o **número de Grundy** ou **nímero** de e_i ($G(e_i)$):

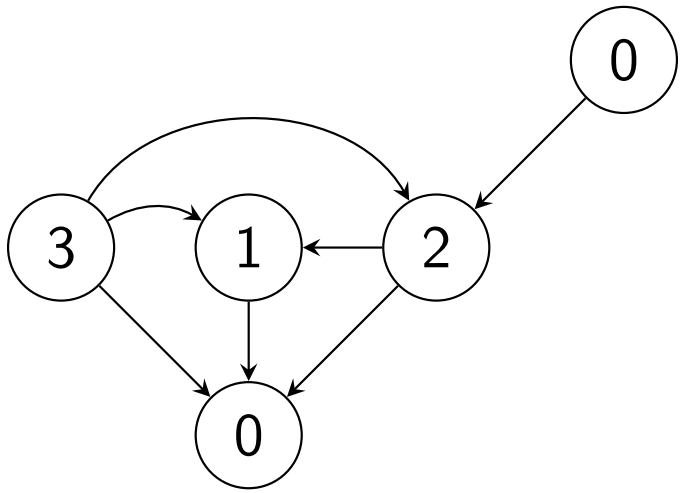
$$G(e_i) = \text{mex}(E_i)$$

- O nímero de um estado perdedor é 0, e o nímero de um estado vencedor é positivo.
- Por conta do uso da função **mex** que se fez necessário o uso do Nim incrementado.

Número de Grundy com subjogos

- Consideremos um jogo que consista de subjogos: a cada turno, o jogador da vez escolhe um subjogo e realiza uma jogada nele. O jogo acaba quando não há movimentos possíveis.
- Neste caso, o número de Grundy de um estado é o xor dos níneros dos subjogos.

DAG gerado de estados



Implementação do mex usando set

Código `mexset.h`

```
struct mex {
    set<int> tomex;

    void insert_mex(int x) {
        tomex.insert(x);
    }
    int get_reset_mex() {
        int m = 0;
        for (auto &j : tomex) {
            if (j > m) { tomex.clear(); return m; }
            m++;
        }
        tomex.clear();
        return m;
    }
};
```

Implementação do mex usando um vetor

Código `mex.h`

```
struct mex {
    vector<bool> tomex;
    int gt = 0;

    mex(int n) : tomex(n) {};

    void insert_mex(int x) {
        tomex[x] = 1;
        gt = max(gt, x);
    }
    int get_reset_mex() {
        int m;
        for (m = 0; tomex[m]; m++);
        fill(tomex.begin(), tomex.begin()+gt+1, 0);
        return m;
    }
};
```

O jogo de Grundy

- Existe uma pilha de n pedras e dois jogadores se movem alternadamente.
- Em cada jogada, um jogador escolhe uma pilha e divide ela em duas pilhas não vazias que tem um número diferente de moedas.
- O jogador que fizer a última jogada ganha o jogo.
- Esse jogo é um pouquinho mais complexo, e precisa que a gente use números de Grundy para resolver.

Implementação do jogo de Grundy

Código grundy.cpp

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e6+15;
#include "mexset.h"
vector<int> g (N, -1);
int main() {
    mex m (N);
    for (int n = 0; n <= 1226; n++) {
        if (n == 1 || n == 2) { g[n] = 0; continue; }
        for (int i = 1; i <= n/2; i++) if (i != n-i)
            m.insert_mex(g[i] ^ g[n-i]);
        g[n] = m.get_reset_mex();
    }
    int t; cin >> t; while (t--) {
        int n; cin >> n;
        if (n >= 1226)
            cout << "first\n";
        else
            cout << (g[n] != 0 ? "first" : "second") << "\n";
    }
}
```

Implementação do jogo de Grundy (continuado)

Entrada	Saída
3 6 7 8	first second first

Um exercício em procura de padrões

- Por que quando $N \geq 1226$, o primeiro jogador sempre ganha?
- Não sei, mas isso é verdade até pelo menos 10^6.
- É uma conjectura em aberto se isso é verdade sempre.
- Muitos jogos imparciais em geral chegam a padrões que se repetem, então as vezes tem como roubar com uma solução matemática (como já vimos).

Misère

- Quando o número de Grundy é zero, você está no estado de derrota.
- Tem alguns outros jogos onde isso é o contrário, o estado de zero é vitória. Isso é chamado de *misère*.
- Em alguns casos é só inverter o jogo, em outros, não é tão trivial.
- No Nim normal a estratégia é a mesma com apenas uma situação onde se muda a jogada: Quando a jogada iria deixar apenas pilhas de tamanho 1. Nesse caso, a estratégia certa é deixar um número ímpar de pilhas de tamanho 1.

Implementação do Misère

Código misere.cpp

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    cin.tie(0);
    ios_base::sync_with_stdio(0);
    int t; cin >> t;
    while (t--) {
        int n; cin >> n;
        bool all_ones = true;
        int res = 0;
        for (int i = 0; i < n; i++) {
            int x; cin >> x; res ^= x;
            if (x != 1) { all_ones = false; }
        }
        if (all_ones) { res = n % 2 == 0; }
        cout << (res != 0 ? "first" : "second") << "\n";
    }
}
```

Implementação do Misère (continuado)

Entrada	Saída
4 2 1 1 3 2 1 3 3 1 1 1 4 1 1 1 1	first second second first

Nim de escada

- Temos um vetor com N números.
- Podemos mover X de uma posição do vetor da direita para esquerda (soma X na esquerda, diminui X na direita).
- Como modelar isso como um jogo de Nim?

Nim de escada

- Podemos desconsiderar as posições ímpares porque o segundo jogador pode sempre "desfazer" a sua jogada.
- Então precisamos apenas das posição pares (que são as válidas).

Implementação do Nim de escada

Código `stair.cpp`

```
#include <bits/stdc++.h>
using namespace std;
int main() {
    int t; cin >> t;
    while (t--) {
        int n; cin >> n;
        int ans = 0;
        for (int i = 0; i < n; i++) {
            int p; cin >> p;
            if (i % 2) { ans ^= p; }
        }
        cout << (ans == 0 ? "second" : "first") << "\n";
    }
}
```

Implementação do Nim de escada (continuado)

Entrada	Saída
3 3 0 2 1 4 1 1 1 1 2 5 3	first second first

Nim de escada só que reescrito

Esses problemas acabam sendo só o problema do Nim de escada:

- Pegar um valor do filho para um pai em um árvore, altura par é válida, altura ímpar inválida.
- Mover pedras para a esquerda, pulando pedras que já estão colocadas no tabuleiro.

E isso é tudo!

- Todo o conteúdo que vocês vão precisar está aqui!
- Confira alguns links com leituras extras.
- Técnicas *anti-hashing*: `https://codeforces.com/blog/entry/60442`.
- Problemas com *hashing*: `https://codeforces.com/blog/entry/60445`.
- Mais variações de Nim: `https://www.youtube.com/watch?v=_99F4as2V6c`.
- Mais informações sobre k-Slow Nim e Nim monotônico: `https://arxiv.org/pdf/1705.06774.pdf`.
- Jogos em grafos arbitrários: `https://cp-algorithms.com/game_theory/games_on_graphs.html`.
- Bons estudos!