UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

MARCO ANTONIO BALESTRIN

ANÁLISE E SOLUÇÃO DE PROBLEMAS PROPOSTOS NA MARATONA DE PROGRAMAÇÃO SBC DOS ANOS 2023, 2024 E 2025

CHAPECÓ
2026

MARCO ANTONIO BALESTRIN

ANÁLISE E SOLUÇÃO DE PROBLEMAS PROPOSTOS NA MARATONA DE PROGRAMAÇÃO SBC DOS ANOS 2023, 2024 E 2025

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação da Universidade Federal da Fronteira Sul (UFFS), como requisito para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Andrei de Almeida Sampaio Braga

CHAPECÓ
2026

Balestrin, Marco Antonio

ANÁLISE E SOLUÇÃO DE PROBLEMAS PROPOSTOS NA MARATONA DE PROGRAMAÇÃO SBC DOS ANOS 2023, 2024 E 2025 / Marco Antonio Balestrin – 2026.

66 f.

Orientador: Prof. Dr. Andrei de Almeida Sampaio Braga

Trabalho de Conclusão de Curso (Graduacão) – Universidade Federal da Fronteira Sul, Curso de Ciência da Computação, Chapecó, SC, 2026.

1. Programação Competitiva 2. Maratonas de Programação 3. Estruturas de Dados 4. Grafos 5. Exponenciação Modular I. Braga, Prof. Dr. Andrei de Almeida Sampaio, orient. II. Universidade Federal da Fronteira Sul. III. Título.

MARCO ANTONIO BALESTRIN

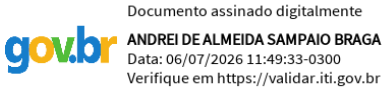
ANÁLISE E SOLUÇÃO DE PROBLEMAS PROPOSTOS NA MARATONA DE PROGRAMAÇÃO SBC DOS ANOS 2023, 2024 E 2025

Trabalho de Conclusão de Curso apresentado ao Curso de Ciência da Computação da Universidade Federal da Fronteira Sul (UFFS), como requisito para obtenção do título de Bacharel em Ciência da Computação.

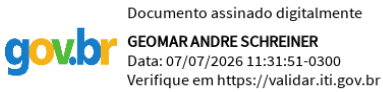
Orientador: Prof. Dr. Andrei de Almeida Sampaio Braga

Este Trabalho de Conclusão de Curso foi avaliado e aprovado pela banca avaliadora em: 26/06/2026.

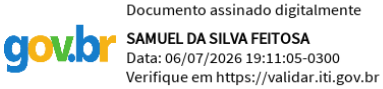
BANCA AVALIADORA



Prof. Dr. Andrei de Almeida Sampaio Braga - UFFS



Prof. Dr. Geomar André Schreiner - UFFS



Prof. Dr. Samuel da Silva Feitosa - UFFS

AGRADECIMENTOS

Agradeço primeiramente à minha família, pelo apoio incondicional, paciência e incentivo em todos os momentos desta jornada, sendo minha base e motivação para seguir em frente.

Aos meus amigos de equipe da maratona de programação, agradeço pela parceria, pelo aprendizado compartilhado e por todas as horas dedicadas juntos aos treinos e competições, que contribuíram imensamente para meu crescimento pessoal e acadêmico.

Aos professores do clube de programação e aos demais professores do curso, agradeço por todo o ensinamento, dedicação e conhecimentos transmitidos ao longo desses anos, essenciais para minha formação.

Por fim, expresso minha sincera gratidão ao meu orientador, professor Andrei de Almeida Sampaio Braga, pela disponibilidade e pelas valiosas contribuições que foram fundamentais para a realização e conclusão deste trabalho.

RESUMO

O presente trabalho tem como objetivo promover uma análise e discussão acerca da solução de problemas de diferentes edições da Maratona de Programação SBC. Para tanto, foi realizada uma seleção de problemas, considerando a dificuldade e o tema de cada um. Em seguida, para cada problema, foi implementada uma solução em C++, sendo verificada a complexidade de tempo e de memória, além de realizada a avaliação por meio de um juiz *online*. Os resultados obtidos refletem a eficácia das soluções, baseadas em conceitos como os seguintes: vetor de frequências para processamento em bloco, operações bit a bit para manipulação de números, busca em profundidade para achar ciclos específicos em um grafo e exponenciação modular para calcular inversos multiplicativos modulares. Conclui-se que, ao apresentar soluções detalhadas para diferentes tópicos da computação, além de explicar as estruturas e algoritmos utilizados, este trabalho contribui de maneira significativa para a disseminação de material de estudo destinado a futuros competidores de maratonas de programação, bem como a estudantes interessados em programação competitiva.

Palavras-Chave: Programação Competitiva, Maratonas de Programação, Estruturas de Dados, Grafos, Exponenciação Modular.

ABSTRACT

The present work aims to promote an analysis and discussion of problem solutions from different editions of the SBC Programming Marathon. To this end, a selection of problems was made, taking into account each problem's difficulty and theme. Then, for each problem, a C++ solution was implemented, with its time and memory complexity verified, in addition to being evaluated through an online judge. The results obtained reflect the effectiveness of the solutions, based on the following concepts: frequency arrays for block processing, bitwise operations for number manipulation, depth-first search to identify specific cycles in a graph, and modular exponentiation to compute modular multiplicative inverses. In conclusion, by presenting detailed solutions for different computing topics and explaining the structures and algorithms used, this work significantly contributes to the dissemination of study material aimed for future programming marathon competitors, as well as students interested in competitive programming.

Keywords: Competitive Programming, Programming Contests, Data Structures, Graphs, Modular Exponentiation.

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

A programação competitiva é uma prática que se encaixa na categoria de esportes mentais, focando no desenvolvimento de habilidades intelectuais, como estratégia e raciocínio lógico. Os problemas propostos abrangem uma variedade de situações, desde cenários totalmente hipotéticos até questões baseadas em situações reais do cotidiano. Cada desafio tem requisitos específicos de tempo e memória, e a solução é apresentada por meio de um código de programação (algoritmo). Além disso, para cada problema, são fornecidos detalhes contextuais, explicações sobre as entradas e saídas esperadas, e exemplos ou casos de teste para auxiliar na compreensão (CRIȘAN; STĂNICĂ; ALTAR-SAMUEL, 2020).

Dois formatos se destacam nas competições de programação, popularmente chamadas de maratonas de programação: o formato da Olimpíada Internacional de Informática (IOI, *International Olympiad in Informatics*), direcionado a estudantes do ensino médio e do primeiro ano do ensino superior, e o formato do *International Collegiate Programming Contest* (ICPC), voltado exclusivamente para estudantes do ensino superior (KOSTKA, 2021). Participar dessas competições resulta no aprimoramento das habilidades de resolução de problemas e maior familiaridade com algoritmos e estruturas de dados. Além disso, a participação em maratonas estimula o trabalho em equipe, fortalece a tomada de decisões sob pressão, além de proporcionar oportunidades de contato e interação profissional e maior visibilidade no mercado de trabalho.

O objetivo deste trabalho é promover uma análise e uma solução detalhada para um conjunto de quatro problemas selecionados de diferentes edições da Maratona de Programação da Sociedade Brasileira de Computação (SBC). Para uma contribuição mais significativa, foram selecionados problemas que abrangem diversos tópicos fundamentais da computação, como estratégias de processamento em bloco, operações bit a bit, grafos, recorrências e exponenciação modular. Cada solução segue um fluxo pré-determinado, iniciando na conceituação teórica e na decisão do plano a ser executado, e finalizando na implementação, na análise e na discussão da estratégia adotada. Dessa forma, busca-se elaborar um material abrangente que sirva como um recurso valioso para estudantes que buscam intensificar seus conhecimentos e se preparar de forma eficaz para competições futuras.

1.1 Metodologia

Nesta seção, são descritos os passos seguidos para a elaboração deste trabalho, detalhando o processo de seleção dos problemas, bem como a estratégia e as ferramentas utilizadas na sua resolução. O objetivo é fornecer uma compreensão clara dos critérios empregados na escolha dos problemas e da abordagem adotada para a implementação das soluções, evidenciando o raciocínio e as justificativas para as decisões tomadas ao longo de cada etapa.

Revisão bibliográfica. Para que cada solução tenha embasamento teórico, foi realizada a revisão bibliográfica com foco na literatura de ciência da computação voltada para tópicos de programação competitiva. A escolha de livros já consolidados permitiu uma consulta mais direta, o que resultou em abordagens mais sólidas, inteligentes e alinhadas com as melhores práticas.

Seleção dos problemas. Para a escolha dos problemas, dois critérios foram seguidos: o contato prévio do autor com os problemas, uma vez que participou das competições para as quais foram elaborados, e a preferência por problemas que envolvessem a aplicação de técnicas variadas e apresentassem um nível adequado de dificuldade. Os problemas foram extraídos das edições regionais e nacionais da Maratona de Programação da SBC, especificamente das Fases Regionais de 2023, 2024 e 2025 e da Fase Nacional de 2024.

Além dos temas presentes em cada problema, um índice quantitativo foi selecionado para determinar quais problemas ofereciam um desafio maior. Esse índice foi obtido através das estatísticas fornecidas pela SBC para cada problema de cada edição, as quais apontam o número total de submissões e o número de submissões aceitas e, consequentemente, a porcentagem de sucesso. Para o primeiro problema escolhido, da edição de 2023, houve 43 submissões aceitas de um total de 1.193, ou seja, uma porcentagem de sucesso de 4%. Já para o segundo problema, de 2024, 288 submissões aceitas de um total de 1.349, ou seja, uma taxa de êxito de 21%. No terceiro problema, também de 2024, nenhuma submissão aceita de um total de 9, ou seja, uma proporção de 0%. Por fim, para o quarto problema, de 2025, 83 submissões aceitas de um total de 174, com uma taxa de sucesso de 48%.

Estratégia de implementação e otimização. Para a implementação das soluções, uma estratégia foi utilizar a linguagem de programação C++. Alguns fatores foram decisivos para essa escolha, entre eles o fato de a linguagem ser compilada, o que significa que o código gerado é otimizado para execução rápida, condição essa que é crucial em competições, onde o tempo de execução do algoritmo pode ser

um fator determinante. Outro fator é a linguagem dispor de uma Biblioteca Padrão (STL, *Standard Template Library*), que oferece implementações otimizadas de diversas estruturas de dados, o que economiza tempo e esforço ao resolver problemas complexos.

Além disso, C++ permite um controle preciso sobre a alocação e desalocação de memória, o que é essencial para otimizar o uso de recursos e evitar desperdícios, possibilitando uma eficiência tanto em termos de tempo quanto de memória. Somado a isso, o C++ conta com uma comunidade ativa de desenvolvedores, especialmente no contexto da programação competitiva, o que facilita o acesso a recursos e bibliotecas que aceleram o desenvolvimento.

Uma estratégia de otimização amplamente utilizada no ambiente de programação competitiva é o emprego de um código base, que reúne definições e estruturas reutilizáveis para diferentes problemas (*template*). No caso específico do C++, também são incluídas definições otimizadas, como o `ios_base::sync_with_stdio(0)`, que desativa a sincronização entre as funções de entrada e saída, e o `cin.tie(0)`, que impede a limpeza automática da saída sempre que uma operação de entrada é realizada. Essas otimizações proporcionam um aumento significativo na velocidade de execução, especialmente em cenários que envolvem grandes volumes de dados.

Análise, avaliação e discussão das soluções. Para cada um dos problemas abordados no trabalho, foi dedicada uma seção à análise da complexidade de tempo e de memória, a fim de justificar a abordagem adotada. Em seguida, para avaliar a solução, foi utilizado um juiz *online*, a plataforma Codeforces (MIRZAYANOV, 2026), que se destaca há vários anos na comunidade de programação competitiva. Isso permitiu a validação tanto da lógica empregada quanto da eficiência da implementação em relação aos limites de tempo e memória estabelecidos. Por fim, para cada problema, uma seção foi destinada à discussão da solução, evidenciando o motivo do funcionamento, explorando possíveis limitações e destacando prováveis alternativas.

1.2 Organização do texto

A organização deste trabalho visa conduzir o leitor de maneira detalhada pelas soluções dos quatro problemas abordados, sendo que cada um deles é tratado em um capítulo diferente. Os Capítulos 2, 3, 4 e 5 são dedicados a explorar um problema específico em cada um deles, e cada capítulo segue uma estrutura padrão, com quatro seções principais, como visto a seguir:

- Conceitos: Apresenta os fundamentos teóricos e os conceitos essenciais para entender a solução.
- Algoritmo: Expõe o algoritmo principal selecionado para compor a solução, detalhando a motivação da sua utilização, o seu funcionamento e exemplos de implementação.
- Resolução: Descreve a entrada e saída do problema, como é esperado o resultado e como aplicar o algoritmo escolhido levando em conta os requisitos do problema descrito.
- Análise e discussão: Evidencia uma análise da complexidade de tempo e de memória da implementação, assim como apresenta uma discussão sobre limitações e possíveis alternativas.

Em seguida, o Capítulo 6 reúne as principais conclusões do trabalho, destacando os conhecimentos adquiridos ao longo do desenvolvimento e as considerações feitas sobre as estratégias implementadas. Por fim, são apresentados os apêndices contendo os códigos das soluções de forma integral e os anexos contendo os certificados que comprovam a participação nas edições escolhidas da Maratona de Programação da SBC.

2. PROBLEMA 1: “EXTRAINDO PÓLEN”

O Problema “Extraindo Pólen” detalha uma situação na qual em um jardim existem *N* flores que floresceram, cada uma com uma quantidade de pólen a ser coletado pelas abelhas. As abelhas devem formar uma fila e esperar sua vez. Quando a vez de uma abelha chega, esta deve se deslocar para a flor que apresente o maior total de pólen. Ao chegar à flor, a abelha coleta a soma dos dígitos do total, por exemplo: a abelha chega à flor com total de pólen de 234, então deve coletar o equivalente a $2 + 3 + 4 = 9$, deixando a flor atual com $234 - 9 = 225$ grãos de pólen. Se a flor, por acaso, tiver 0 grãos, a abelha coletará 0 grãos em sua vez.

A dificuldade surge quando Gertrude, uma abelha trabalhadora, acredita que esse processo todo seria muito confuso. Assim, o objetivo do problema é, dadas *N* flores e uma posição *K* na fila de coleta, determinar qual será a quantia de grãos coletados pela abelha Gertrude quando chegar sua vez na fila. Uma descrição completa do problema está disponível em <https://codeforces.com/gym/104555/problem/E>.

A ideia central da solução é usar um vetor para registrar as frequências e processar todas as flores que possuem a mesma quantidade de pólen de uma só vez. Em vez de analisar flor por flor individualmente, o algoritmo as agrupa por quantidade de pólen, verificando a cada novo grupo se a abelha alvo já pode ser atendida ou se outras flores precisam ser coletadas primeiro. A resolução seguiu a linha de raciocínio proposta pela equipe do canal oficial do Clube de Programação da UTFPR, cujas explicações e lógica detalhadas deste problema podem ser consultadas em <https://www.youtube.com/watch?v=JBq6quUw8-I>.

A seguir, são detalhados os fundamentos que sustentam a solução apresentada. Inicialmente, expõem-se os conceitos teóricos necessários, seguidos pela estruturação do algoritmo utilizado. Logo após, demonstra-se a resolução do problema para, por fim, realizar uma análise e discussão sobre os resultados obtidos.

2.1 Conceitos

Esta seção tem como objetivo esclarecer os conceitos fundamentais para a compreensão da solução proposta. Destaca-se a simulação de estados como o desafio central do problema e o vetor de frequências, estrutura do algoritmo de ordenação linear *counting sort*, como a estrutura de dados chave para a solução.

2.1.1 Simulação de estados

A simulação de estados é uma técnica computacional que visa modelar o comportamento de um sistema dinâmico ao longo do tempo. O sistema é definido por um conjunto de estados e regras que ditam como ele transita de um estado para outro a cada evento (CORMEN et al., 2022).

2.1.2 Vetor de frequências

O vetor de frequências é uma estrutura de dados fundamental, implementada com um simples vetor (*array*), onde o índice corresponde a um valor e o conteúdo armazenado nesse índice representa o número de ocorrências daquele valor. Essa técnica, por exemplo, é o passo inicial do algoritmo de ordenação linear *counting sort*, no qual é utilizada para contar os elementos de entrada antes de posicioná-los (HALIM; HALIM; EFFENDY, 2018).

2.2 Algoritmo

2.2.1 Motivação para o vetor de frequências

A motivação principal de se usar o vetor de frequências é obter uma forma eficiente de processar elementos quando estes pertencem a um intervalo limitado de valores inteiros. O vetor mantém uma contagem direta de cada elemento, isso possibilita que sejam feitas operações em bloco, o que, dependendo do problema, pode ser essencial para evitar passos adicionais desnecessários.

Essa abordagem se destaca devido à frequência de qualquer elemento desejado poder ser consultada em tempo constante, além de contornar a necessidade de algoritmos de busca ou de estruturas de dados mais complexas, que apenas dificultariam a solução.

2.2.2 Funcionamento do vetor de frequências do *counting sort*

Diferentemente de outros algoritmos de ordenação, que são baseados somente em comparação entre os elementos inseridos, o *counting sort* utiliza-se de outras operações para realizar a ordenação. O conjunto de técnicas que abrangem esse algoritmo possibilita sua classificação como um algoritmo executado em tempo linear (CORMEN et al., 2022).

O *counting sort* pode ser dividido em três partes: vetor de frequências, soma de prefixos e processo de posicionamento. Para entender o fluxo de funcionamento, a Figura 2.1 exemplifica os passos do algoritmo. Primeiro, na fase de frequência, o algoritmo cria um vetor auxiliar *C* para contar as ocorrências de cada elemento do vetor de entrada *A*. Em seguida, na fase de soma de prefixos, o vetor *C* é modificado: cada posição passa a armazenar a soma de seu valor original mais o valor da posição anterior. Com isso, C_s indica as posições dos elementos de *A* de maneira ordenada.

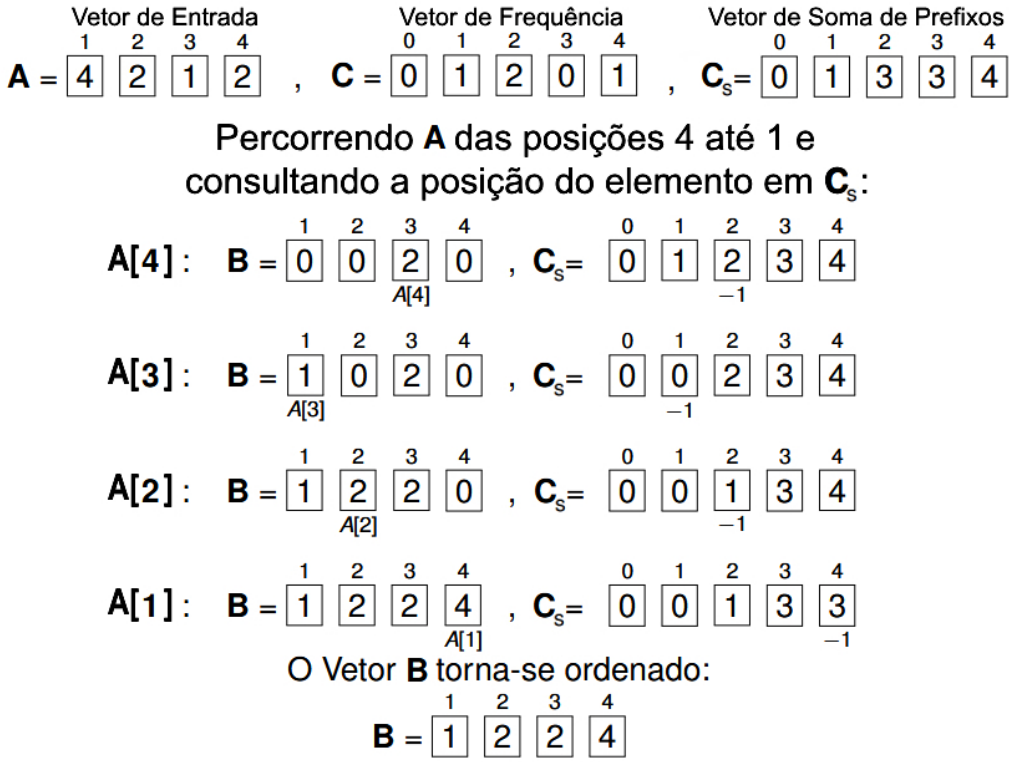


Figura 2.1: Funcionamento do algoritmo de *counting sort*

Por último, na fase de posicionamento, o vetor de saída *B* é construído. Percorrendo o vetor de entrada *A* do final para o início, cada elemento é colocado na posição indicada pelo vetor C_s. Após posicionar um elemento, o valor correspondente

em C_s é decrementado para garantir que o próximo elemento idêntico seja colocado na posição correta, uma antes. O vetor *B* ordenado é mostrado no fim da Figura 2.1.

2.2.3 Implementação do vetor de frequências do *counting sort*

A seguir, apresenta-se uma implementação em C++ da técnica do vetor de frequências do algoritmo de *counting sort* (Código 2.1). Considerando *N* o número de elementos do vetor de entrada *A* e *MAX_VALOR_A* o valor máximo de um elemento de *A*, é realizada a contagem das ocorrências de um valor *A*[*i*] em *C*, ou seja, o vetor *C* na posição *A*[*i*] é acrescido de uma unidade a cada vez que esse valor aparece.

```
1 // N: Número de elementos de A
2 // MAX_VALOR_A: Valor máximo de um elemento de A
3
4 vector<int> C(MAX_VALOR_A + 1);
5
6 for(int i = 0; i < N; i++){
7     C[A[i]] += 1;
8 }
```

Código 2.1: Código em C++ para vetor de frequências

2.3 Resolução

Conforme descrito no início do capítulo, o objetivo do problema é determinar a quantidade total de pólen coletada por uma abelha a partir de um conjunto de flores e a posição da abelha Gertrude em uma fila. A coleta segue uma restrição específica: a abelha deve selecionar sempre a flor com a maior carga de pólen, contabilizando apenas a soma dos dígitos desse valor. Para a solução (descrita abaixo), busca-se processar simultaneamente todas as flores que possuem a mesma quantidade de pólen, utilizando um vetor de frequências para agilizar o cálculo.

2.3.1 Descrição da entrada do problema

A entrada é composta por duas linhas. A primeira linha contém dois inteiros N ($1 \leq N \leq 10^6$) e K ($1 \leq K \leq 10^9$), que representam, respectivamente, o número de flores e a posição de Gertrude na fila. A segunda linha contém N inteiros F_i ($1 \leq F_i \leq 10^6$ com $1 \leq i \leq N$), nos quais o i-ésimo inteiro é a quantidade inicial de pólen da i-ésima flor.

2.3.2 Descrição da saída do problema

A saída deve conter apenas uma única linha com um inteiro Q que representa a quantidade de pólen que a abelha Gertrude coletará quando chegar seu turno na fila de coleta do jardim.

2.3.3 Solução para o problema

A seguir, é descrita a solução para o problema. A Figura 2.2 mostra os passos executados na solução quando, por exemplo, há 4 flores, uma com 10, outra com 11 e duas com 12 grãos de pólen, e Gertrude é a terceira na fila de coleta.

Primeiramente, analisa-se a distribuição das flores conforme suas quantidades de pólen. Em seguida, partindo das flores com a maior quantidade (12), inicia-se a coleta. Como ainda não é a vez de Gertrude, Fulana e Beltrana avançam e coletam 1 + 2 = 3 grãos de pólen, reduzindo o total daquelas flores de 12 para 9. Por fim, chega a vez de Gertrude e a flor a ser coletada é a que possui 11 grãos, logo ela retira 1 + 1 = 2 grãos de pólen.

Traduzindo a dinâmica da Figura 2.2 para um passo a passo mais estruturado, a primeira etapa consiste em criar um vetor de frequências F com o maior valor possível de grãos (10^6) onde cada posição de F está preenchida com 0. Em seguida, para cada valor de grãos da entrada, deve-se atualizar F, incrementando o valor em 1 de acordo com cada ocorrência. O passo seguinte é iterar sobre F da sua última posição até sua primeira e verificar em cada posição i se K (posição da abelha Gertrude na fila de coleta) se qualifica para uma das duas possibilidades a seguir:

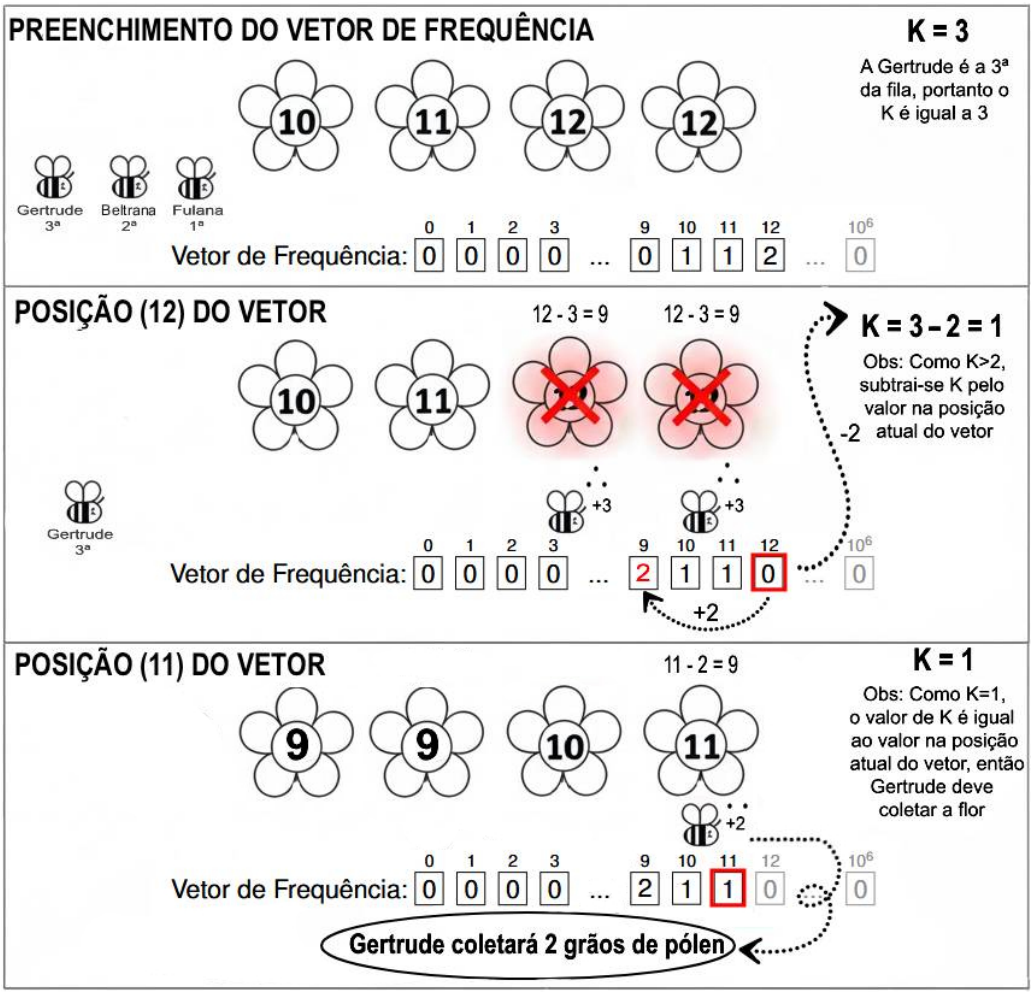


Figura 2.2: Exemplo do processo de extração de pólen

- Se $K > F[i]$, então Gertrude não poderá coletar a flor i, pois essa condição indica que ainda restam, no mínimo, $F[i]$ abelhas posicionadas à frente dela na fila;
- Se $K \leq F[i]$, então Gertrude poderá coletar a flor i, visto que essa condição indica que ela se encontra entre as próximas $F[i]$ posições na fila.

Caso Gertrude colete a flor, basta somente imprimir o valor da soma dos dígitos em $F[i]$. Caso contrário, três operações são feitas: o valor de K é diminuído de $F[i]$, a nova posição em F, dada por i subtraído da soma dos seus dígitos, é acrescida de $F[i]$ e o valor de $F[i]$ é atualizado. Se ao fim de todas as iterações nenhuma flor for coletada, 0 é impresso na tela.

2.3.4 Implementação da solução

A solução do problema é implementada em duas funções principais: a função `soma_digitos` e a função `main` (mostradas nos Códigos 2.2 e 2.3). Na `soma_digitos`, a função recebe um inteiro `valor_graos`, que será o número a ter seus dígitos somados, e inicializa a variável `soma` em zero (linhas 1-2). Logo abaixo, um laço de repetição começa e, enquanto `valor_graos` não for zero, é adicionado o dígito da unidade de `valor_graos` à variável `soma`, através do resto da divisão por 10, e, posteriormente, é retirado esse mesmo dígito de `valor_graos`, por meio da divisão por 10 (linhas 3-6). Por fim, retorna-se a soma completa de todos os dígitos de `valor_graos` (linha 7).

```
1 int soma_digitos(int valor_graos){
2     int soma = 0;
3     while(valor_graos){
4         soma += valor_graos % 10;
5         valor_graos /= 10;
6     }
7     return soma;
8 }
```

Código 2.2: Código em C++ para a função `soma_digitos`

Já na função `main`, são declarados os inteiros N, K e Q (correspondentes aos citados nas Seções 2.3.1 e 2.3.2) e um vetor de inteiros com índices de 0 ao valor máximo de N, ou seja, 10^6 (linhas 10-11). Em seguida, são lidos N e K e os N elementos (F_i), preenchendo o vetor de frequências F (linhas 13-19). Posteriormente (linhas 20-28), é feita uma iteração variando i de 10^6 até 0, comparando a cada iteração o valor de K e $F[i]$. Se $K \leq F[i]$, atribui-se o valor de Q. Caso contrário, adiciona-se $F[i]$ à posição da nova flor, decrementa-se K e atualiza-se $F[i]$. O último passo é imprimir Q, seja ele 0 ou não, e finalizar o programa (linhas 29-32).

```
int main() {
    int N, K, Q = 0;
    vector<int> F(1000001);

    cin >> N >> K;

    for(int i = 0; i < N; i++){
        int Fi;
        cin >> Fi;
        F[Fi] += 1;
    }
    for(int i = 1000000; i >= 0; i--){
        if(K <= F[i]){
            Q = soma_digitos(i);
            break;
        }
        F[i-soma_digitos(i)] += F[i];
        K -= F[i];
        F[i] = 0;
    }
    cout << Q << '\n';

    return 0;
}
```

Código 2.3: Código em C++ para a função `main`

2.4 Análise e discussão

2.4.1 Análise de complexidade da solução

Através de uma análise da eficiência da solução do problema, é possível constatar que, para o primeiro laço do algoritmo, no registro das frequências, a complexidade é de $O(N)$. Em seguida, no segundo laço, a complexidade se dá pelo valor máximo de grãos de pólen (P_{max}) multiplicado pela complexidade da função

`soma_digitos`, totalizando $O(P_{max} \times \log P_{max})$. Portanto, a solução, em sua integridade, tem complexidade de $O(N + P_{max} \times \log P_{max})$.

Segundo as especificações do problema, $1 \leq N \leq 10^6$ e $1 \leq F_i \leq 10^6$. Assim, caso $N = 10^6$ e o (*maior*) $F_i = P_{max} = 10^6$, então a complexidade de tempo será de aproximadamente $10^6 + 10^6 \times 6$ operações, ou seja, uma carga computacional executada dentro do limite estipulado de 0,6 segundos do problema.

Quanto à complexidade de espaço, é possível uma análise mais direta, já que independe das variáveis N, K *ou* F_i, somente de P_{max}, o qual é declarado em um vetor F de tamanho $P_{max} + 1 = 10^6 + 1$. Isso resulta em um uso de memória mais do que aceitável.

2.4.2 Discussão da solução

O método empregado para resolver o problema, que transforma uma simulação direta e lenta em um processo agrupado e eficiente, é basicamente uma otimização baseada em restrições de valor. O ponto principal da solução não é a utilização de um algoritmo complexo, mas sim o uso de uma estrutura de dados simples: o vetor de frequências.

É fundamental destacar que, embora a solução utilize a mesma técnica de contagem que serve de base para o *counting sort*, o problema em si não envolve ordenação, mas sim a simulação de um processo de estados. A complexidade final de $O(N + P_{max} \times \log P_{max})$, onde N é o número de flores e P_{max} é a quantidade máxima de grãos de pólen, é linear em N. Entretanto, sua eficiência está atrelada não apenas ao número de elementos de entrada (N), mas também à magnitude máxima de seus valores (P_{max}).

A viabilidade da solução advém justamente das restrições do problema. Como N e a quantidade de grãos de pólen F_i são limitados a 10^6, a memória necessária para o vetor de frequências e o número de iterações do laço principal são de fácil gerenciamento. Se a quantidade de pólen pudesse ser um número muito grande, por exemplo 10^{12}, o vetor de frequências se tornaria inviável para ser armazenado na memória, e o tempo de execução do laço seria excessivo, tornando esta solução ineficaz.

Caso a solução com vetor de frequências fosse inviável, seria necessário recorrer a outras estratégias. Uma abordagem alternativa seria usar uma estrutura de dados que mantivesse apenas as flores existentes, como um `map` em C++ ou um `max-heap`. Um `max-heap` (fila de prioridade) permitiria encontrar a flor mais cheia em tempo $O(\log N)$ a cada passo. Essa abordagem ainda seria lenta demais, dado que

K pode chegar a 10^9. Portanto, caso seja possível, utilizar mais memória para evitar operações repetitivas e alcançar uma complexidade de tempo superior é uma escolha justa e necessária.

3. PROBLEMA 2: “LEXICOGRAFICAMENTE MÁXIMO”

O Problema “Lexicograficamente Máximo” expõe uma circunstância na qual uma lista de *N* inteiros se encontra armazenada na memória de um dispositivo eletrônico. Este é capaz de realizar a seguinte operação: trocar bits entre os elementos da lista, gerando números que possivelmente nem sequer estavam listados.

O obstáculo surge quando, dada uma lista de números, é necessário fazer diversas trocas entre os bits, e o resultado deve ser lexicograficamente máximo. Em outras palavras, o primeiro elemento da lista, após as trocas, deve ser o maior possível, seguido do segundo maior e assim por diante.

Por exemplo, se os valores de entrada fossem 8 (1000), 5 (0101) e 1 (0001), os dois bits ativos do 5 seriam deslocados para o 8, enquanto o único bit ativo do 1 seria deslocado para o 5, resultando nos binários (1101), (0001) e (0000). Nota-se que o primeiro número da nova sequência contém os bits ativos mais significativos possíveis, o segundo apenas um bit ativo e o último nenhum bit ativo. Essa sequência garante que o resultado seja o maior possível em comparação lexicográfica. Mais detalhes sobre o problema estão disponíveis em <https://codeforces.com/gym/105327/problem/L>.

A essência da solução reside em registrar os bits ativos em cada posição de cada número para, posteriormente, reconstruí-los. Para o registro, utilizam-se operações bit a bit na manipulação das representações binárias. Já na etapa de reconstrução, aplica-se um algoritmo guloso para alcançar a resposta ótima esperada. A solução baseou-se em uma estratégia própria, elaborada pela equipe durante a competição.

Abaixo, são apresentados os pontos principais que explicam a solução. Primeiramente, abordam-se os conceitos teóricos necessários, seguindo para a elaboração do algoritmo utilizado. Na sequência, mostra-se a resolução do problema para, por fim, chegar à análise e discussão dos resultados encontrados.

3.1 Conceitos

Esta seção busca a compreensão da solução proposta, detalhando os conceitos principais. Destacam-se as operações bit a bit como a base central do problema e o método guloso (*greedy*) como o ponto-chave para construir a solução esperada.

3.1.1 Operações bit a bit

As operações bit a bit atuam diretamente nos dígitos binários (bits) de um número. Entre as principais, destacam-se a operação *AND*, que pode ser utilizada para verificar se um bit específico está ativo (igual a 1), a operação *OR*, que pode ser usada para ativar um bit específico, e a operação de *shift*, que desloca todos os bits de um número para a esquerda ou para a direita, efetivamente multiplicando ou dividindo o valor por potências de 2 (HALIM; HALIM; EFFENDY, 2018).

3.1.2 Algoritmo guloso

O algoritmo guloso é uma abordagem usada para resolver problemas de otimização. A principal estratégia é fazer a escolha que se mostra a melhor naquele exato momento, ignorando possíveis consequências futuras dessa escolha. A cada etapa do processo, o algoritmo toma uma decisão ótima de maneira local com a esperança de que a sequência de decisões leve a uma solução ótima de maneira global. (CORMEN et al., 2022).

3.2 Algoritmo

3.2.1 Motivação para operações bit a bit

A motivação central é utilizar o conceito de operações bit a bit para resolver problemas em que cálculos entre números podem ser feitos utilizando propriedades binárias. A estratégia consiste em, ao invés de manipular diretamente um número na representação decimal, analisar a distribuição dos bits que o compõem na forma binária. Em vez de observar cada número de maneira isolada, considera-se o conjunto total de bits em cada posição e contabiliza-se quantos deles estão ativos, ou seja, com valor 1, em um vetor de contagem.

Ao empregar essa forma de operação com números, torna-se viável trabalhar com quantidades fixas de bits, o que possibilita a construção de novos valores de maneira controlada. A partir das operações aliadas às contagens, pode-se garantir que

otimizações necessárias sejam feitas, como a maximização de valores ou ordenações específicas.

3.2.2 Funcionamento de operações bit a bit

A estratégia é operar sobre a forma binária de um número, processando cada posição de bit de maneira independente. O princípio fundamental consiste, para cada número, em verificar quais posições de bit estão com valor igual a 1 e registrar em um contador de cada posição.

A verificação é realizada através de duas operações: operações de deslocamento (*shift*) e operações lógicas *AND*. A primeira é usada para criar máscaras que isolam cada posição de bit. Por exemplo, como mostrado no início da Figura 3.1, ao deslocar o valor 1 sucessivas vezes para a esquerda, é possível obter as potências de 2 correspondentes a cada posição. Ou seja, as máscaras funcionam como moldes binários, em que apenas o bit da posição desejada está ativo (igual a 1) e todos os demais permanecem zerados.

Já a segunda operação é aplicada entre a máscara e o número analisado. Assim como é demonstrado na continuação da Figura 3.1, caso o resultado da operação não seja zero, o bit naquela posição está ativo. Por fim, cada vez que um bit ativo é encontrado, incrementa-se o contador associado àquela posição em uma unidade.

3.2.3 Implementação de operações bit a bit

A seguir, apresenta-se uma implementação do algoritmo de contagem de bits (Código 3.1). Para cada valor K lido (linha 1) é possível realizar as seguintes operações:

- Linha 3: Iterar sobre todos os bits da forma binária de K.
- Linha 4: Através do deslocamento, cria-se uma máscara de bit que contém apenas o j-ésimo bit ativo, isto é, somente a posição do bit de interesse é igual a 1, enquanto todas as demais são 0. Em seguida, realiza-se a operação lógica *AND* para verificar se esse bit específico está ativo no número analisado.
- Linha 5: Incrementar a contagem associada àquela posição.

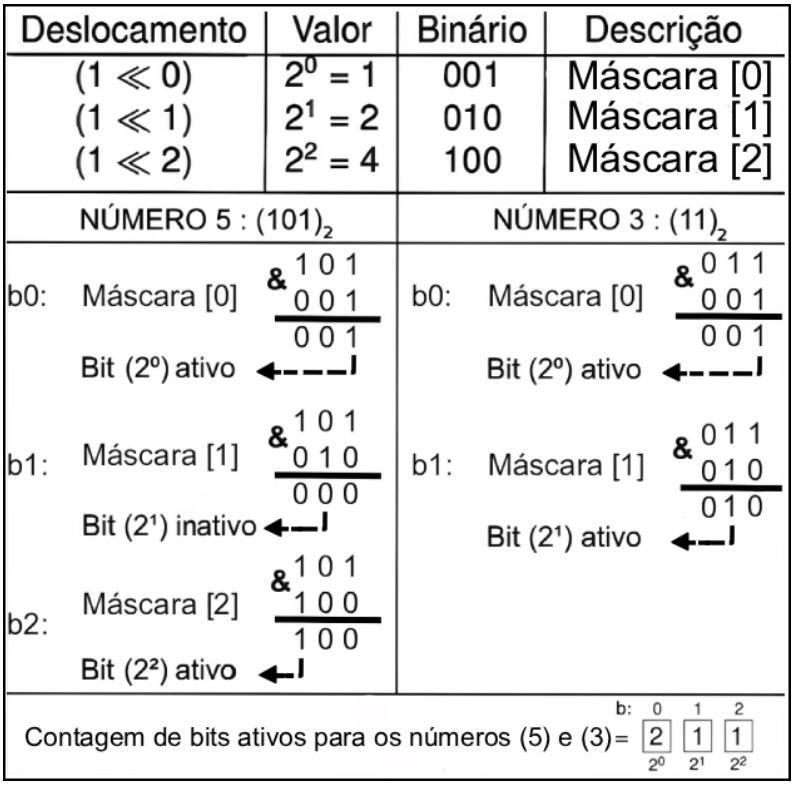


Figura 3.1: Exemplo de operações e contagem dos bits

```
cin >> K;

for(int j = 0; (1 << j) <= K; j++){
    if((1 << j) & K)
        contagem[j]++;
}
```

Código 3.1: Código em C++ para as operações bit a bit

3.3 Resolução

Como apresentado no início do capítulo, o objetivo do problema é realizar a troca dos bits de um conjunto de números dado, de forma que cada número seja, em sequência, lexicograficamente máximo. Para a solução detalhada a seguir, visa-se registrar as ocorrências de bits em cada valor da entrada para, com essa informação, construir novos valores com o maior número possível de bits mais significativos ativos.

3.3.1 Descrição da entrada do problema

A entrada é composta por duas linhas. A primeira linha contém um único inteiro N ($1 \leq N \leq 10^5$) e a segunda linha contém N inteiros $a_1, \ldots, a_N$ ($0 \leq a_i \leq 10^9$), separados por espaços.

3.3.2 Descrição da saída do problema

A saída deve conter apenas uma única linha com N inteiros, separados por espaços, representando a sequência de elementos lexicograficamente máximos.

3.3.3 Solução para o problema

A primeira etapa consiste em realizar a contagem dos bits em cada posição para os N números. Para isso, cada valor é analisado individualmente, verificando-se quais posições possuem bits ativos. Cada vez que um bit está ativo, incrementa-se 1 em sua posição correspondente. Ao fim, o vetor de contagem representa a quantidade total de bits para cada posição binária de todos os números da entrada. Revisitando o exemplo do começo do capítulo, para os números 8, 5 e 1, o vetor de contagem seria o seguinte: 2 para o primeiro bit menos significativo, 0 para o segundo, 1 para o terceiro e 1 para o quarto bit menos significativo.

Na segunda etapa, o vetor de contagens é usado para construir novos valores, começando do bit menos significativo para o mais significativo. Realiza-se o deslocamento à esquerda sobre o valor 1 para criar uma máscara binária com o bit correspondente ativo, e essa é combinada ao valor em construção por meio de operações *OR* bit a bit, adicionando o novo bit sem alterar os já configurados.

Para todos os N números gerados na segunda etapa, são impressos na tela os números que, através das operações realizadas, podem ser chamados de lexicograficamente máximos.

3.3.4 Implementação da solução

A solução do problema é implementada diretamente na função `main` (mostrada no Código 3.2). Inicialmente, é considerado o máximo número de bits de um número da entrada, que são 30 bits, equivalente a 10^9 (linha 1). No próximo passo, é dividido o problema em duas partes, representadas pelos dois laços de repetição presentes (linhas 9-15 e 19-28).

No primeiro laço (linhas 9-15), para cada número lido da entrada, são percorridos seus bits e é verificado se cada um está ativo, através do operador *AND* (&) entre o número e a máscara ($1 << j$), e, em seguida, é realizada a contagem dessa posição de bit. No segundo laço (linhas 19-28), com uma estratégia gulosa, são reconstruídos N números percorrendo-se a contagem de todos os bits existentes e atribuindo ao `novo_valor` um bit de cada posição contabilizada, por meio do operador *OR* (|) entre o valor sendo construído e a máscara ($1 << j$). Para cada valor reconstruído, são impressos os valores em si e os espaços que os separam (linhas 27-29). Relembrando o exemplo do início do capítulo, para os números 8, 5 e 1, os valores reconstruídos seriam 13 (1101), 1 (0001) e 0 (0000).

3.4 Análise e discussão

3.4.1 Análise de complexidade da solução

Por meio de uma avaliação da eficiência da solução do problema, concluise que, para o primeiro laço do algoritmo, na contagem dos bits, a complexidade se dá pelo número de elementos (N) e pelo máximo número de bits de um número da entrada (N_BITS), portanto de $O(N \times N\_BITS)$. Logo após, no segundo laço, a complexidade é avaliada, sendo idêntica à do primeiro laço, $O(N \times N\_BITS)$, pois percorre N números e para cada número N_BITS posições para formar o novo valor.

Considerando as restrições do problema ($1 \leq N \leq 10^5$ e $0 \leq a_i \leq 10^9$, e que $10^9 \approx 2^{30}$), o algoritmo realiza, no pior caso, cerca de $(2 \times 10^5 \times 30) = 6 \times 10^6$ operações simples. Portanto para o limite estipulado pelo problema de 1 segundo o algoritmo mostra-se eficiente.

Quanto à complexidade de espaço, esta pode ser calculada simplesmente analisando o vetor de contagem, que tem um tamanho fixo de número de bits, no caso

```cpp
#define N_BITS 30
int contagem[N_BITS];

int main(){

    int N, K;
    cin >> N;

    for(int i = 0; i < N; i++){
        cin >> K;
        for(int j = 0; (1 << j) <= K; j++){
            if((1 << j) & K)
                contagem[j]++;
        }
    }

    int novo_valor;

    for(int i = 0; i < N; i++){
        novo_valor = 0;
        for(int j = 0; j < N_BITS; j++){
            if(contagem[j] > 0){
                novo_valor = (1 << j) | novo_valor;
                contagem[j]--;
            }
        }
        cout << novo_valor << ' ';
    }
    cout << '\n';
    return 0;
}
```

Código 3.2: Código em C++ para o Problema “Lexicograficamente Máximo”

30. Assim, o uso de memória é de $O(1)$, ou seja, constante em relação à quantidade de números da entrada. Isso evidencia um consumo de memória extremamente baixo, muito inferior ao máximo permitido.

3.4.2 Discussão da solução

A abordagem utilizada para resolver o problema se destaca em contextos onde os dados podem ser descritos em termos binários, o que elimina o uso de es-

truturas mais complexas. Além disso, possibilita à implementação aproveitar a simplicidade das operações bit a bit, que são executadas de forma eficiente em nível de hardware, tornando o algoritmo direto e de fácil execução.

Quanto à eficiência, cada número é analisado até o bit ativo mais significativo (no máximo 30 posições). Portanto, o custo constante associado às operações de deslocamento e comparação com *AND*, é mínimo quando comparado a abordagens aritméticas mais elaboradas. A solução é viável justamente devido às restrições do problema, no qual é possível utilizar apenas um vetor fixo de contagem de tamanho reduzido, independentemente da entrada. Entretanto, como o método apenas contabiliza a presença de bits, não são preservadas as informações sobre a posição original dos números, nem é possível retornar à configuração inicial. Ou seja, trata-se de uma abordagem eficiente para reconstrução ou redistribuição de valores, mas não totalmente adequada quando é necessário manter uma estrutura lógica ou até mesmo uma ordenação dos dados de entrada. Porém, dentro do escopo do problema, essa característica não é um obstáculo para a eficiência nem para os resultados obtidos, embora possa ser uma limitação em outros casos.

Existem outras abordagens para o problema, que seriam consideravelmente menos eficientes. Uma possibilidade, mesmo que ingênua, seria simular as operações de troca entre os bits da entrada. Por exemplo, para maximizar a_1, seria preciso iterar por todos os outros $N - 1$ números e por todos os seus bits, realizando trocas. Esse processo teria que ser repetido para o próximo valor da entrada e para os valores seguintes, resultando em uma complexidade polinomial mais alta, provavelmente na ordem de $O(N^2)$, o que excederia o limite de tempo.

4. PROBLEMA 3: “BIKETOPIA’S CYCLIC TRACK”

O Problema “Biketopia’s Cyclic Track” apresenta uma situação na qual, em um determinado país fictício chamado Biketopia, há *N* cidades e *M* estradas bidirecionais. Cada estrada conecta duas cidades, e cada cidade é alcançada por pelo menos três estradas, além de que é sempre possível viajar de uma cidade para outra usando uma ou mais estradas.

A dificuldade surge quando o torneio anual de corrida de bicicleta do país está prestes a acontecer. Diferentemente dos anos anteriores, neste caso, o circuito fechado do torneio, que deve passar por pelo menos três cidades, precisa ser planejado de forma que não prejudique o acesso das pessoas a essas cidades. Em outras palavras, é essencial que, em cada cidade, exista pelo menos uma estrada livre e não conectada ao circuito, garantindo que o acesso seja mantido para todos os habitantes e visitantes. Uma descrição detalhada do problema está disponível em <https://codeforces.com/gym/105505/problem/B>.

A estratégia principal da solução consiste em utilizar busca em profundidade em grafos e propriedades de árvores para encontrar um ciclo (circuito fechado) adequado. Devido às restrições do problema, a existência desse circuito é garantida, cabendo à solução apenas localizá-lo de forma eficiente. O caminho adotado para a solução foi guiado pelas recomendações e dicas deixadas pelos próprios autores dos problemas, conforme disponibilizado em <https://codeforces.com/gym/105505/attachments/download/28286/icpcla24_solution_sketches.pdf>.

As seções abaixo explicam o percurso da solução proposta. Inicia-se pelos conceitos teóricos, avançando para a construção do algoritmo adotado. Posteriormente, apresenta-se o passo a passo da resolução para, por fim, conduzir a análise e a discussão dos resultados.

4.1 Conceitos

Esta seção tem como objetivo esclarecer os conceitos fundamentais necessários para a compreensão da solução proposta. Três elementos principais sustentam a abordagem: o uso de grafos como estrutura base do problema, um método recursivo que gera uma árvore como estratégia principal e a identificação de arestas de retorno como a essência para se localizar um ciclo adequado.

4.1.1 Grafos

Um grafo é uma estrutura composta por nós (ou vértices) conectados por arestas. Vértices que se conectam são chamados de vizinhos ou adjacentes. Um caminho é uma sequência de nós ligados por arestas, e o seu comprimento é determinado pelo número de arestas que o compõem. Já um ciclo é um tipo especial de caminho, onde o nó inicial é o mesmo que o nó final. Um grafo é considerado conexo quando existe um caminho entre qualquer par de nós. As arestas podem ser direcionadas, indicando uma relação unidirecional, ou não direcionadas, representando uma relação bidirecional (LAAKSONEN, 2020).

4.1.2 Árvores

Árvores são um tipo especial de grafo, em que o número de arestas é igual ao número de vértices menos uma unidade. Elas não contêm ciclos, ou seja, não há caminhos fechados, e são conexas. Além disso, em uma árvore, há sempre um único caminho possível entre dois vértices (HALIM; HALIM; EFFENDY, 2018).

4.1.3 Recursão

A recursão é um conceito onde uma função é definida em termos de si mesma. Uma equação de recorrência descreve uma função, utilizando a própria função em sua definição. Essa equação geralmente possui dois tipos de casos: o caso base, que não envolve chamadas recursivas e fornece a solução direta, e o caso recursivo, onde a função é chamada novamente com entradas menores ou diferentes (CORMEN et al., 2022).

4.2 Algoritmo

4.2.1 Motivação para a busca em profundidade

A motivação fundamental de se utilizar a busca em profundidade (DFS, do inglês *Depth First Search*) em um grafo é devido ao seu funcionamento. Diferentemente da busca em largura (BFS, do inglês *Breadth First Search*), que explora o grafo por níveis de distância, a DFS prioriza a profundidade. Os dois algoritmos são ideais para problemas como identificação de componentes conexos ou detecção de ciclos. Ao realizar buscas em profundidade em um grafo, a estrutura que corresponde à sequência das chamadas recursivas realizadas durante a execução do algoritmo (na qual cada vértice aponta para o vértice que o chamou) forma um conjunto de árvores (CORMEN et al., 2022). Esses pontos tornam a DFS uma técnica útil em cenários onde a exploração completa de caminhos e a organização recursiva são essenciais.

4.2.2 Funcionamento da busca em profundidade

Como o próprio nome aponta, o seu cerne é a profundidade. A partir de um nó inicial, a DFS segue um único caminho, descobrindo novos nós até que não seja mais possível avançar. Então, a busca retorna aos nós anteriores que ainda têm caminhos não explorados. A DFS pode ser implementada de duas formas: iterativa com uma estrutura de pilha ou recursiva. Optando pela forma recursiva, a Figura 4.1 ilustra o funcionamento da busca em profundidade em um grafo com 6 vértices de *A* a *F*, através dos seguintes passos:

- A busca inicia no vértice *A*.
    1. Visita-se **A**.
- A partir de *A*, o primeiro vizinho não visitado é *B*.
    2. Visita-se **B**.
- De *B*, o próximo vértice não visitado é *E*.
    3. Visita-se **E**.
- A partir de *E*, não há mais vértices não visitados. Retorna-se para *B*.

- De *B*, também não há mais vizinhos não visitados, então retorna-se para *A*.
- De volta a *A*, o próximo vértice não visitado é *C*.

4. Visita-se **C**.

- Após *C*, o próximo vértice não visitado é *F*.

5. Visita-se **F**.

- Após *F*, o próximo vértice não visitado é *D*.

6. Visita-se **D**.

- Como *D* não possui mais vizinhos não visitados, retorna-se para *F*, *C* e *A* e a busca é encerrada.

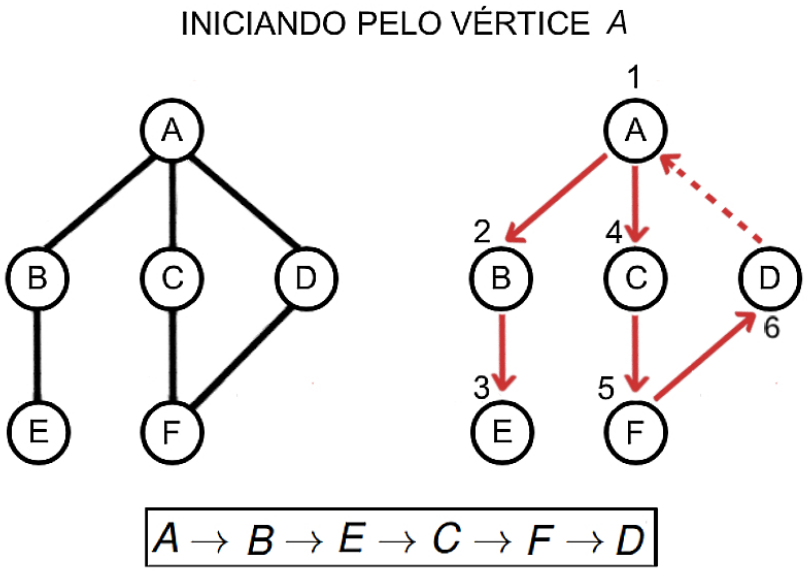


Figura 4.1: Exemplo de busca em profundidade (para o grafo à esquerda)

Ainda na Figura 4.1, é importante destacar que as arestas sólidas vermelhas fazem parte da árvore de busca gerada pela DFS, enquanto a aresta pontilhada vermelha não pertence a essa árvore. Esta última é conhecida como aresta de retorno, pois conecta um vértice a um ancestral já visitado durante a busca.

4.2.3 Implementação da busca em profundidade

A seguir, apresenta-se uma implementação recursiva do algoritmo de busca em profundidade (Código 4.1). Considerando declarados a `lista_adjacencias`, que

armazena, para cada vértice, uma lista com seus vértices adjacentes, e o vetor `visitado`, responsável por registrar os vértices já visitados, os passos seguintes são executados:

- Linhas 1-3: É iniciada a busca em profundidade a partir do vértice `v` e o vértice atual é marcado e impresso.
- Linhas 4-8: Para cada vizinho `u` de `v`, verifica-se se o mesmo já foi visitado. Caso não, é feita uma chamada recursiva, passando-o como parâmetro.

```
1 void dfs(int v) {
2     visitado[v] = true;
3     cout << v << ' ';
4     for (int u : lista_adjacencias[v]) {
5         if (!visitado[u])
6             dfs(u);
7     }
8 }
```

Código 4.1: Código em C++ para a busca em profundidade (DFS)

4.3 Resolução

Como exposto no início do capítulo, o objetivo do problema é encontrar um ciclo que passe por no mínimo três cidades, devendo existir pelo menos uma estrada livre não conectada a esse ciclo. Para a solução exposta adiante, procura-se apenas encontrar, observando a árvore gerada pela busca em profundidade sobre o grafo, o circuito desejado. Devido às particularidades do problema, esse circuito está sempre presente.

4.3.1 Descrição da entrada do problema

A primeira linha da entrada contém dois inteiros N ($4 \leq N \leq 2 \times 10^5$) e M ($6 \leq M \leq 3 \times 10^5$), que representam, respectivamente, o número de cidades e o número de estradas. Além disso, cada cidade apresenta um número distinto de 1 a N e cada estrada, um identificador distinto de 1 a M.

Em seguida, na *i*-ésima linha das *M* estradas, são apresentados dois inteiros *U* e *V* ($1 \leq U, V \leq N$ e $U \neq V$) representando as cidades que a estrada conecta. É garantido que existe no máximo uma estrada conectando cada par de cidades.

4.3.2 Descrição da saída do problema

Caso exista um trajeto possível, a saída deve consistir de duas linhas. A primeira linha deve indicar o número de estradas no trajeto, e a segunda linha deve listar os identificadores das estradas, na ordem em que aparecem no trajeto. Se não houver nenhum trajeto possível, a saída deve conter um único caractere asterisco ('*').

4.3.3 Solução para o problema

A ideia central da solução é, a partir do primeiro vértice do grafo, executar uma DFS, classificando as arestas entre arestas de árvore (aquelas usadas pela DFS) e arestas de retorno (as que conectam um vértice a um ancestral já visitado) – um exemplo é mostrado na Figura 4.1. Como o grafo é conexo (é sempre possível viajar de uma cidade para outra usando uma ou mais estradas) e cada vértice possui grau pelo menos 3 como dito no começo do capítulo, sempre existirão arestas de retorno. A partir daí, verifica-se, a cada aresta de retorno, a profundidade do ancestral, que representa a profundidade do vértice na árvore da DFS.

Em seguida, seleciona-se a aresta de retorno que conecta o vértice mais profundo possível a um ancestral o mais distante possível da raiz. Por exemplo, se a aresta de retorno fosse (u, v), com u sendo ancestral de v, o ciclo esperado seria formado pela aresta e pelo caminho da árvore que liga u até v.

Ao escolher a aresta de retorno, é garantido que, mesmo que houvesse a remoção das arestas do ciclo, o grafo permaneceria conexo. Isso ocorre porque cada vértice do ciclo ainda possui pelo menos uma aresta que não pertence a ele, consequência da restrição do grau mínimo 3 imposto pelo problema. Ademais, a estrutura da busca em profundidade assegura que sempre existe um caminho alternativo até a raiz que independe das arestas removidas.

4.3.4 Implementação da solução

A solução do problema é dividida em duas partes principais: a função `dfs` e a função `main`, mostradas, respectivamente, nos Códigos 4.2 e 4.3. Na função `dfs`, inicia-se a busca em profundidade verificando se um vértice já foi visitado, utilizando a distância do vértice inicial. Ou seja, se a distância de um vértice for zero, significa que ele ainda não foi visitado (linhas 1–7).

```
1  void dfs(int v, int d){
2      dist[v] = d;
3      for(auto [u, id]: lista_adjacencia[v]){
4          if(dist[u] == 0){
5              ancestral[u] = make_pair(v, id);
6              dfs(u, d + 1);
7          }
8          else if(dist[u] < dist[v]
9                  && u != ancestral[v].first
10                 && dist[u] > dist[maior_id]){
11             maior_id = u;
12             aresta_maior = make_pair(v, id);
13         }
14     }
15 }
```

Código 4.2: Código em C++ para a função `dfs`

Na sequência (linhas 8–15), para cada vértice já visitado, é verificado se o vértice *u* é um ancestral de *v*, se a aresta não faz parte da árvore de DFS, e se o ancestral *u* está o mais profundo possível na árvore. Caso essas condições sejam satisfeitas, a aresta é considerada uma aresta de retorno e, portanto, candidata a formar um ciclo. Se a profundidade do ancestral *u* for a maior observada até o momento, as variáveis `maior_id` e `aresta_maior` são atualizadas.

```
16 int main(){
17     int N, M;
18     cin >> N >> M;
19 
20     for(int i = 1; i <= M; i++){
21         int U, V;
22         cin >> U >> V;
23 
24         lista_adjacencia[U].push_back(make_pair(V, i));
25         lista_adjacencia[V].push_back(make_pair(U, i));
26     }
27     maior_id = 0;
28     dfs(1, 1);
29 
30     int tam = dist[aresta_maior.first] - dist[maior_id] + 1;
31     cout << tam << '\n';
32     cout << aresta_maior.second;
33 
34     for(int i = aresta_maior.first; i != maior_id;
35         i = ancestral[i].first){
36 
37         cout << ' ' << ancestral[i].second;
38     }
39     cout << '\n';
40 
41     return 0;
42 }
```

Código 4.3: Código em C++ para a função `main`

Na função `main`, são lidos o número de cidades (N) e de estradas (M) (linhas 16–18), bem como as M estradas, que são armazenadas em uma lista de adjacência juntamente com os identificadores das arestas (linhas 19–26). Após a leitura dos dados, a função `dfs` é chamada a partir do vértice 1, com a profundidade inicial igual a 1 (linhas 27–28). Por fim, o programa reconstrói o ciclo percorrendo a sequência de ancestrais desde o vértice mais profundo até o ancestral identificado, imprimindo o tamanho do ciclo e os identificadores das arestas que o compõem (linhas 30–39).

Assim, o algoritmo encontra e exibe um ciclo que pode ser removido sem desconectar o grafo, atendendo às condições estabelecidas no problema.

4.4 Análise e discussão

4.4.1 Análise de complexidade da solução

A eficiência do algoritmo baseia-se na execução de uma única busca em profundidade (DFS) sobre o grafo, o que permite identificar tanto a estrutura da árvore de exploração quanto a aresta de retorno responsável pela formação do ciclo. A complexidade de tempo, portanto, é determinada pelo número total de vértices N e arestas M, uma vez que cada vértice é visitado exatamente uma vez e cada aresta é processada no máximo duas vezes. Assim, a complexidade de tempo da solução é O(N + M).

Após a DFS, a reconstrução do ciclo é feita percorrendo a cadeia de ancestrais entre dois vértices, o que no pior caso exigiria menos que $O(N)$ operações, mesmo assim permanece linear em relação ao tamanho do grafo. Avaliando os limites do problema N ($4 \leq N \leq 2 \times 10^5$) e M ($6 \leq M \leq 3 \times 10^5$), o algoritmo executa, no pior dos casos, 5×10^5 operações, valor que respeita o limite de 2 segundos imposto pelo problema.

A complexidade espacial também é linear em relação ao tamanho do grafo, $O(N + M)$, já que o programa utiliza estruturas proporcionais ao número de vértices e arestas: a lista de adjacência para armazenar o grafo, o vetor de distâncias e o vetor de ancestrais. Esses vetores consomem memória de acordo com o tamanho máximo da entrada, o que é adequado e inferior aos limites estabelecidos.

4.4.2 Discussão da solução

A abordagem adotada combina propriedades teóricas de árvores de DFS com algumas garantias estruturais do problema. A ideia central é que, em um grafo conexo onde todos os vértices possuem grau pelo menos 3, sempre existe um ciclo cuja remoção não desconecta o grafo. A DFS é particularmente conveniente nesse contexto, pois permite distinguir de forma clara entre arestas de árvore e arestas de retorno, possibilitando a escolha correta de um ciclo seguro.

Esse método é direto e eficiente, pois elimina a necessidade de verificações adicionais de conectividade. A utilização da aresta de retorno, que conecta o vértice mais profundo ao ancestral mais distante, garante que o ciclo obtido seja suficientemente profundo na estrutura da árvore, minimizando assim o risco de desconexão após a formação do ciclo desejado.

Embora o algoritmo não explore técnicas mais complexas, como pontes, componentes biconexas ou verificação explícita de conectividade, ele se mantém matematicamente sólido. Entretanto, se o problema impusesse restrições diferentes, como a presença de grafos muito densos, arestas direcionadas ou pesos complexos, a simplicidade desta solução não seria suficiente. Em casos assim, seria necessário empregar algoritmos ou estruturas de dados mais aprimoradas, como *disjoint sets*.

5. PROBLEMA 4: “FRANGOLINO ALI NA MESA”

O Problema “Frangolino ali na Mesa” relata um cenário no qual Washington, um chef de um restaurante, resolve integrar a tecnologia da inteligência artificial com sua paixão que é a culinária. Então, ele fabrica um robô-garçom, chamado Frangolino. Para testar sua criação, Washington pensou em preparar uma noite especial entre amigos. Para isso, *N* mesas serão separadas e numeradas de 1 a *N*, e somente um prato será servido: frango à milanesa.

O impasse surge quando Washington resolve que seria engraçado, devido ao único prato servido, fazer um jogo com as palavras. Assim, definiu que Frangolino iniciaria a noite na mesa 1 e executaria as seguintes instruções: instrução “Ali na mesa *X*”, para Frangolino ir à mesa *X*, e a instrução “À milanesa *X*”, para registrar no sistema um pedido de *X* frangos à milanesa para a mesa na qual Frangolino se encontra. Porém, devido a uma limitação do robô-garçom, que não consegue determinar qual instrução deve seguir, a cada comando, há 50% de chance do Frangolino executar a primeira ou a segunda instrução. Por exemplo, se o primeiro comando recebido tiver valor igual a 3, Frangolino pode interpretar como “Ali na mesa 3” e se dirigir à mesa 3, ou ele pode interpretar como “À milanesa 3” e registrar um pedido de 3 frangos para a mesa 1 (mesa inicial). Portanto, a tarefa principal é, dado o histórico de comandos recebidos pelo robô-garçom, descobrir, para cada mesa, qual o número esperado de frangos à milanesa que serão servidos. O relato completo do problema está disponível em <https://codeforces.com/gym/106073/problem/F>.

O cerne da solução está em analisar as probabilidades e as restrições do problema e, em seguida, para cada mesa existente, processar o impacto de cada instrução disponível. Para esse processo, utilizam-se fórmulas de recorrência combinadas com um algoritmo de exponenciação modular e o cálculo do inverso multiplicativo, o que permite encontrar a quantidade esperada de frangos à milanesa por mesa. A abordagem para a solução deste problema seguiu as observações contidas em um documento elaborado por um ex-competidor, que pode ser acessado em <https://codeforces.com/gym/106073/attachments/download/33204/editorial.pdf>.

Na sequência, o desenvolvimento da solução é apresentado em etapas. Primeiramente, estabelece-se o embasamento com os conceitos teóricos. A etapa seguinte consiste na formulação do algoritmo necessário. Por fim, realiza-se a exposição da resolução, seguida pela análise e discussão dos resultados.

5.1 Conceitos

Esta seção pretende expor os conceitos fundamentais para o entendimento da solução proposta. Para isso, evidencia-se a utilização da exponenciação modular e do inverso multiplicativo modular, indispensáveis para lidar com as especificações impostas pelo problema, bem como o emprego da recorrência que permite estruturar o cálculo da solução desejada.

5.1.1 Exponenciação modular

A exponenciação modular é uma operação frequentemente utilizada em cálculos de teoria dos números, pois permite calcular de maneira mais eficiente grandes potências, enquanto cálculo direto de uma potência pode se tornar inviável devido ao seu crescimento exponencial. A exponenciação modular, inclusive, é fundamental em muitos algoritmos de teste de primalidade e em sistemas de criptografia RSA (CORMEN et al., 2022).

5.1.2 Recorrência

Uma recorrência é uma função em que um valor depende de seus valores anteriores, calculados com base em uma fórmula recursiva que envolve uma soma ponderada de termos anteriores, cada um multiplicado por coeficientes constantes. Uma forma comum de calcular esses valores é através de programação dinâmica, onde todos os valores são computados sequencialmente (LAAKSONEN, 2020).

5.1.3 Inverso multiplicativo modular

O inverso multiplicativo modular consiste em converter uma divisão ($a \,/\, b \bmod m$) em uma multiplicação ($a \times b^{-1} \bmod m$), o que permite tratar b como um fator multiplicativo desde que b e m sejam primos entre si (coprimos). Para determinar esse inverso de maneira eficiente, especificamente quando m é um número primo, aplica-se o Pequeno Teorema de Fermat, que possibilita encontrar o valor necessário através

de exponenciação modular em vez de realizar a divisão tradicional (HALIM; HALIM; EFFENDY, 2020).

5.2 Algoritmo

5.2.1 Motivação para a exponenciação modular

A motivação central para o uso da exponenciação modular é permitir o cálculo eficiente de potências elevadas, especialmente quando essas operações precisam ser realizadas sob um módulo. Assim, permitindo que a exponenciação seja realizada em tempo razoável mesmo para expoentes muito elevados.

Além disso, a técnica serve de base para o cálculo de inversos multiplicativos modulares, pois quando a é coprimo de m e m é primo, o inverso de a é dado por $a^{m-2} \bmod m$ (LAAKSONEN, 2020). Assim, a exponenciação modular torna possível computar inversos utilizando apenas multiplicações e as reduções modulares.

5.2.2 Funcionamento da exponenciação modular

O princípio fundamental da técnica baseia-se na definição recursiva da potenciação, permitindo reduzir a complexidade do cálculo através da divisão do expoente. Para um módulo m, a função é definida pela seguinte estrutura condicional:

$$a^n \equiv \begin{cases} 1 & \text{se } n = 0 \\ \left(a^{n/2} \times a^{n/2}\right) \mod m & \text{se } n \text{ é par} \\ \left(a^{n-1} \times a\right) \mod m & \text{se } n \text{ é ímpar} \end{cases}$$

Nesta abordagem, quando n é par, o problema é reduzido à metade do tamanho original, ou seja, $(n/2)$ multiplicado por $(n/2)$. Quando n é ímpar, o problema é reduzido para $(n-1)$ e multiplicado pela base. Por exemplo, se $a = 3$ e o expoente $n = 3$, sob o módulo $m = 7$, a resolução ocorre através das seguintes chamadas recursivas:

1. Caso $n = 3$ **(ímpar):**

$$3^3 \equiv 3^2 \times 3 \pmod 7$$

É necessário calcular 3^2.

2. Caso $n = 2$ **(par):**

$$3^2 \equiv 3^1 \times 3^1 \pmod 7$$

É necessário calcular 3^1.

3. Caso $n = 1$ **(ímpar):**

$$3^1 \equiv 3^0 \times 3 \pmod 7$$

É necessário calcular 3^0.

4. Caso $n = 0$ **(base):**

$$3^0 \equiv 1 \pmod 7$$

5. Retorno da recursão:

   - $n = 1$:

$$3^1 \equiv 1 \times 3 = 3 \pmod 7$$

   - $n = 2$:

$$3^2 \equiv 3 \times 3 = 9 \equiv 2 \pmod 7$$

   - $n = 3$:

$$3^3 \equiv 2 \times 3 = 6 \pmod 7$$

5.2.3 Implementação da exponenciação modular

Abaixo, apresenta-se uma implementação do algoritmo de exponenciação modular (Código 5.1). Para um módulo m pré-definido, a execução da função `exp_modular` apresenta os seguintes passos:

- Linhas 2-3: Aplica-se o módulo à `base` (a) para garantir que o valor inicial pertença ao conjunto dos restos módulo `MOD` (m).
- Linha 5: Verifica-se se o expoente `exp` (n) é igual a zero. Trata-se do caso base da recursão, no qual se retorna o valor 1 (correspondendo à definição de a^0).
- Linhas 7-10: Verifica-se a paridade de `exp`. Caso seja par, realiza-se uma chamada recursiva para `exp/2`, obtém-se o valor parcial e ele é multiplicado por si mesmo, com o resultado sob o módulo `MOD` (correspondendo à operação $a^n \equiv a^{n/2} \times a^{n/2} \pmod m$).

- Linhas 11-13: Caso `exp` seja ímpar, realiza-se uma chamada recursiva para `exp - 1`, multiplicando o resultado pela `base` e aplicando o módulo `MOD` (correspondendo à operação $a^n \equiv a^{n-1} \times a \pmod{m}$).

```
1  long long int exp_modular(long long int base, long long int exp) {
2      long long int parte;
3      base %= MOD;
4
5      if (exp == 0) return 1;
6
7      if (exp % 2 == 0) {
8          parte = exp_modular(base, exp / 2);
9          return (parte * parte) % MOD;
10     }
11     else {
12         return (base * exp_modular(base, exp - 1)) % MOD;
13     }
14 }
```

Código 5.1: Código em C++ para a exponenciação modular

5.3 Resolução

Como relatado no início do capítulo, o objetivo do problema é, a partir de uma lista de comandos, descobrir a quantia esperada de frango à milanesa para cada mesa que um robô-garçom atende. Há a restrição de que, ao receber um comando, o robô-garçom com 50% de chance ou se move para a mesa do comando, ou registra um pedido de frangos para a mesa em que ele está. Para a solução desenvolvida subsequentemente, propõe-se que, por meio de fórmulas de recorrência baseadas em conceitos probabilísticos, encontre-se a quantidade final esperada de frangos para cada mesa analisada.

5.3.1 Descrição da entrada do problema

A entrada é composta por duas linhas. A primeira linha contém dois inteiros N e Q ($1 \leq N, Q \leq 10^5$), representando respectivamente o número de mesas e o número de comandos do robô-garçom. A segunda linha contém Q inteiros $X_1, \ldots, X_Q$ ($1 \leq X_i \leq N$), separados por espaços. Cada X_i descreve o valor do i-ésimo comando que o robô-garçom recebeu, na ordem em que ocorreram.

5.3.2 Descrição da saída do problema

A saída deve conter N linhas. A i-ésima linha deve conter o valor esperado do número de frangos à milanesa que serão servidos na mesa i, conforme descrito a seguir. O valor esperado pode ser um número racional, que será representado como uma fração irredutível $\frac{p}{q}$. Como p e q podem ser valores grandes, deve-se imprimir o resultado de $p \times q^{-1} \bmod (10^9 + 7)$, onde q^{-1} é o inverso multiplicativo de q módulo $(10^9 + 7)$.

5.3.3 Solução para o problema

A primeira etapa consiste em pré-calcular a potência de $1/2$, utilizando a exponenciação modular e o inverso multiplicativo modular ($a^{m-2} \bmod m$). Para isso, obtêm-se a potência como:

$$2^{-k} \equiv (2^{-1})^k \pmod{10^9 + 7},$$

onde:

$$2^{-1} \equiv 2^{(10^9+7)-2} \equiv 500000004 \pmod{10^9 + 7}$$

é o inverso multiplicativo modular de 2.

Em seguida a solução utiliza uma soma auxiliar recursiva $h(i)$ calculada de trás para frente, que armazena todas as somas ponderadas necessárias. Define-se $h(i)$ por:

$$h(Q + 1) = 0, \qquad h(i) = \frac{h(i + 1) + X_i}{2},$$

sendo Q o número de comandos recebidos por Frangolino. Essa recorrência representa a expectativa acumulada para o comando i a partir da expectativa acumulada do

comando seguinte $i+1$, ajustada pela probabilidade do Frangolino executar a instrução "À milanesa X_i" (50%).

Na próxima etapa, obtém-se a contribuição efetiva de cada comando. Com $h(i)$ calculado, define-se

$$g(i) = \frac{h(i+1)}{2},$$

que corresponde à contribuição esperada do comando i consistir na instrução "Ali na mesa X_i". Em seguida, agrupa-se as contribuições por mesa:

$$f(j) = \sum_{i:X_i=j} g(i).$$

O caso da mesa 1 recebe um tratamento adicional, pois o robô-garçom inicia nela: deve-se somar $h(1)$ a $f(1)$, contabilizando todas as contribuições sobre a mesa inicial, mesmo se nenhum comando tiver direcionado o robô-garçom para a mesa 1. Por fim, para cada uma das N mesas, os valores de f calculados compõem o resultado final.

Por exemplo, para um $N = 2$, um $Q = 3$ e $X_1 = 2$, $X_2 = 1$, $X_3 = 2$ os seguintes passos seriam realizados:

- Cálculo de h (para cada comando $i = 1, \ldots, Q+1$ de trás para a frente):

$$h(4) = 0$$

$$h(3) = (0+2)/2 = 1$$

$$h(2) = (1+1)/2 = 1$$

$$h(1) = (1+2)/2 = 3/2$$

- Cálculo de g (para cada comando $i = 1, \ldots, Q$):

$$g(1) = h(2)/2 = 1/2$$

$$g(2) = h(3)/2 = 1/2$$

$$g(3) = h(4)/2 = 0$$

- Cálculo de f (para cada mesa $j = 1, \ldots, N$, com uma soma adicional para $f(1)$):

$$f(1) = g(2) = 1/2 \;\Rightarrow\; f(1) + h(1) = 1/2 + 3/2 = 2$$

$$f(2) = g(1) + g(3) = 1/2 + 0 = 1/2$$

Devido à necessidade de precisão aritmética e ao crescimento das frações, todas as operações devem ser realizadas sob o módulo $10^9 + 7$. O resultado final impresso para cada mesa representa o número esperado de frangos à milanesa servidos naquela mesa.

5.3.4 Implementação da solução

A solução (Código 5.2) se concentra em duas partes: o cálculo do inverso multiplicativo modular de 2 e o cálculo da recorrência. Primeiramente, é utilizada a função `exp_modular` que realiza a exponenciação modular, e por propriedade, o inverso multiplicativo modular também, para atribuir à variável `inv2` o resultado da operação (linha 1). Em seguida (linhas 3-7) são lidos os parâmetros do problema, ou seja, o número de mesas *N*, o número de comandos *Q* e, em seguida, a sequência $X_1, \ldots, X_Q$.

Para o cálculo da recorrência, primeiramente, são construídas, nas linhas 9-14, as somas ponderadas e guardados os valores no vetor *h*. Acumula-se a informação do passo seguinte (`h[i+1]`) somada ao valor do comando atual (`X[i]`), e multiplica-se tudo por `inv2`. Na continuação do algoritmo (linhas 16–21), utiliza-se o vetor *h* para alimentar o vetor final de respostas `f`, inicialmente zerado. Para cada *i* de 1 a *Q*, calcula-se um valor intermediário `g_i`, que representa a contribuição associada ao comando naquela posição. Esse valor é somado diretamente à mesa indicada por `X[i]`, ou seja, `f[X[i]]`.

Em seguida (linha 23), trata-se a contribuição inicial, específica da mesa 1, adicionando `h[1]` ao valor `f[1]`. Essa etapa é necessária porque todos os comandos futuros afetam a mesa inicial diretamente. Por fim, nas linhas 25–26, é impresso o vetor *f* de 1 até *N*, exibindo, para cada mesa, o valor acumulado de frangos à milanesa esperados.

```cpp
    long long int inv2 = exp_modular(2, MOD - 2);

    int N, Q;
    cin >> N >> Q;

    vector<long long int> X(Q + 1);
    for (int i = 1; i <= Q; i++) cin >> X[i];

    vector<long long int> h(Q + 2, 0);

    for (int i = Q; i >= 1; i--) {
        h[i] = (h[i + 1] + X[i]) % MOD;
        h[i] = h[i] * inv2 % MOD;
    }

    vector<long long int> f(N + 1, 0);

    for (int i = 1; i <= Q; i++) {
        long long int g_i = h[i + 1] * inv2 % MOD;
        f[X[i]] = (f[X[i]] + g_i) % MOD;
    }

    f[1] = (f[1] + h[1]) % MOD;

    for (int i = 1; i <= N; i++)
        cout << f[i] % MOD << '\n';
```

Código 5.2: Código em C++ para o Problema “Frangolino ali na mesa”

5.4 Análise e discussão

5.4.1 Análise de complexidade da solução

A eficiência do algoritmo é analisada pela forma como a recorrência é aplicada e como o resultado é formado. Como a recorrência é resolvida de maneira iterativa, percorrendo a lista de comandos de 1 a Q para g e f (ou de Q a 1 para h),

a complexidade de tempo é linear no número de comandos, ou seja, $O(Q)$. Já para a impressão dos resultados, são percorridas as mesas de 1 a N, resultando em uma complexidade de $O(N)$.

Um detalhe importante a ser destacado é o cálculo do inverso multiplicativo modular de 2. Ele é obtido pela exponenciação modular com custo de $O(\log MOD)$. Como, no pior caso, $MOD = 10^9+7$, $\log MOD$ corresponde a cerca de 30 multiplicações e representa, então, um custo fixo, independente de N ou de Q, portanto considerado como tempo constante na análise global. Assim, a complexidade de tempo final resulta em $O(N+Q)$, a qual, para $(1 \leq N, Q \leq 10^5)$, totaliza um máximo de 2×10^5 operações, um valor aceitável para o limite de $0,5$ segundos.

O algoritmo utiliza estruturas de tamanho proporcionais ao número de mesas (N) e comandos (Q): vetores f, g, e h. Assim, a complexidade de espaço também é linear, $O(N + Q)$. Essa complexidade indica que o uso de memória está abaixo dos limites estabelecidos.

5.4.2 Discussão da solução

A solução proposta usa uma abordagem probabilística que traduz o comportamento do sistema em termos de expectativas ao longo do tempo. O algoritmo foca em como cada comando afeta o sistema com o passar do tempo, considerando incertezas. Essa forma de ver o problema, que analisa as dependências entre os eventos, permite resolver o cálculo de forma mais eficiente.

O ponto principal é que a dinâmica dos comandos não é tratada de forma isolada, mas integra um processo cuja influência de um comando sob o outro diminui progressivamente. Além disso, o uso de aritmética modular garante estabilidade numérica, mantendo os valores controlados e precisos, mesmo quando eles crescem rapidamente.

Embora a solução não utilize métodos mais sofisticados como manipulação explícita de probabilidades ou estruturas de dados avançadas, ela se apoia em um raciocínio matemático, aproveitando a interpretação probabilística e uma recorrência simples. Caso o processo tivesse dependências mais complexas, probabilidades variáveis ou relações não lineares entre comandos, essa abordagem simplificada não seria suficiente, exigindo técnicas mais elaboradas.

6. CONCLUSÃO

Este trabalho teve como objetivo a construção e análise de soluções para os problemas selecionados de diferentes edições da Maratona de Programação da Sociedade Brasileira de Computação (SBC). A proposta foi elaborar um material de apoio para futuros competidores, que não apenas fornecesse códigos prontos, mas também incluísse explicações sobre as técnicas necessárias, além de uma discussão que abordasse o funcionamento e as limitações de cada implementação.

A estrutura padrão definida possibilitou a aplicação consistente da metodologia ao longo de todo o texto. Os dados estatísticos e os critérios de seleção dos problemas garantiram uma dificuldade adequada para cada um deles. Com o apoio da revisão bibliográfica, foi possível validar os conceitos apresentados, e a escolha da linguagem C++ permitiu o uso de estruturas e otimizações relevantes para a implementação. Para verificar a validade das soluções, foi utilizado um juiz *online*, nos quais todas as soluções obtiveram êxito.

No primeiro problema, “Extraindo Pólen”, eram necessários cálculos em blocos para encontrar um valor em uma determinada ordem. Simular cada passo seria inviável devido ao tempo de execução estipulado. A solução consistiu no uso de um vetor de frequências, permitindo que múltiplos valores fossem analisados simultaneamente. Em seguida, o problema “Lexicograficamente Máximo” lidava com a manipulação de números através de operações bit a bit. A solução baseou-se em contar as ocorrências de bit em cada número e, após isto, construir novos valores lexicograficamente máximos de forma gulosa.

Já no terceiro problema, “Biketopia’s Cyclic Track”, foi necessário encontrar um ciclo em um grafo que atendesse às exigências do enunciado. Para isso, a solução utilizou uma busca em profundidade, o que possibilitou identificar as arestas de retorno, essenciais para definir a resposta esperada. Por fim, no último problema, “Frangolino ali na Mesa”, o desafio era descobrir uma quantia esperada para uma determinada mesa. A solução encontrada foi usar recorrências para calcular cada valor, aliada a uma abordagem probabilística, que foi expressa por meio do inverso multiplicativo modular, calculado através da exponenciação modular.

A contribuição deste material para a prática de programação competitiva permite ao estudante de Ciência da Computação aprimorar tanto habilidades técnicas quanto competências pessoais. Um estudante que constantemente exercita sua mente resolvendo problemas de programação competitiva desenvolve, na prática, as habilidades necessárias para o mercado de trabalho e, ao aplicar diversos conceitos teóricos, enriquece ainda mais sua formação acadêmica.

REFERÊNCIAS BIBLIOGRÁFICAS

CORMEN, T. et al. *Introduction to Algorithms*. 4th. ed. Cambridge: MIT Press, 2022.

CRIȘAN, D. A.; STĂNICĂ, J. L.; ALTAR-SAMUEL, A. N. Competitive programming. An Analysis of the Performance in Romania. *Journal of Information Systems & Operations Management*, v. 14, n. 1, p. 43–59, 2020. Acesso em: 25 de maio de 2026. Disponível em: <https://ideas.repec.org/a/rau/jisomg/v14y2020i1p43-59.html>.

HALIM, S.; HALIM, F.; EFFENDY, S. *Competitive Programming 4 - Book 1: The Lower Bound of Programming Contests in the 2020s*. USA: Lulu Press, 2018. (Competitive Programming 4: The New Lower Bound of Programming Contests in the 2020s, bk. 1).

HALIM, S.; HALIM, F.; EFFENDY, S. *Competitive Programming 4 - Book 2: The Lower Bound of Programming Contests in the 2020s*. USA: Lulu Press, 2020.

KOSTKA, B. *Sports Programming in Practice*. Dissertação (Mestrado) — Faculty of Mathematics and Computer Science, University of Wrocław, 2021. Acesso em: 25 de maio de 2026. Disponível em: <https://ii.uni.wroc.pl/media/uploads/2021/12/06/bartosz-kostka.pdf>.

LAAKSONEN, A. *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*. Cham: Springer, 2020. (Undergraduate Topics in Computer Science).

MIRZAYANOV, M. *Codeforces: programming contests platform*. 2026. Acesso em: 25 de maio de 2026. Disponível em: <https://codeforces.com/>.

APÊNDICE A – CÓDIGO COMPLETO DO PROBLEMA 1

```cpp
#include <bits/stdc++.h>
using namespace std;

int soma_digitos(int valor_graos){
    int soma = 0;
    while(valor_graos){
        soma += valor_graos % 10;
        valor_graos /= 10;
    }
    return soma;
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int N, K, Q = 0;
    vector<int> F(1000001);

    cin >> N >> K;

    for(int i = 0; i < N; i++){
        int Fi;
        cin >> Fi;
        F[Fi] += 1;
    }

    for(int i = 1000000; i >= 0; i--){
        if(K <= F[i]){
            Q = soma_digitos(i);
            break;
        }
        F[i-soma_digitos(i)] += F[i];
        K -= F[i];
        F[i] = 0;
```

```
36      }
37      cout << Q << '\n';
38
39      return 0;
40  }
```

Código A.1: Código completo do Problema 1 em C++

APÊNDICE B – CÓDIGO COMPLETO DO PROBLEMA 2

```cpp
#include <bits/stdc++.h>
using namespace std;

#define N_BITS 30
int contagem[N_BITS];

int main(){
    ios::sync_with_stdio(0);
    cin.tie(0);

    int N, K;
    cin >> N;

    for(int i = 0; i < N; i++){
        cin >> K;
        for(int j = 0; (1 << j) <= K; j++){
            if((1 << j) & K)
                contagem[j]++;
        }
    }

    int novo_valor;

    for(int i = 0; i < N; i++){
        novo_valor = 0;
        for(int j = 0; j < N_BITS; j++){
            if(contagem[j] > 0){
                novo_valor = (1 << j) | novo_valor;
                contagem[j]--;
            }
        }
        cout << novo_valor << ' ';
    }
    cout << '\n';
    return 0;
```

```
36 }
```

Código B.1: Código completo do Problema 2 em C++

APÊNDICE C – CÓDIGO COMPLETO DO PROBLEMA 3

```cpp
#include <bits/stdc++.h>
using namespace std;

#define MAXN 200001

vector<pair<int, int>> lista_adjacencia[MAXN];
int dist[MAXN], maior_id;
pair<int, int> ancestral[MAXN], aresta_maior;

void dfs(int v, int d){
    dist[v] = d;
    for(auto [u, id]: lista_adjacencia[v]){
        if(dist[u] == 0){
            ancestral[u] = make_pair(v, id);
            dfs(u.first, d + 1);
        }
        else if(dist[u] < dist[v]
                && u != ancestral[v].first
                && dist[u] > dist[maior_id]){
            maior_id = u;
            aresta_maior = make_pair(v, id);
        }
    }
}

int main(){
    ios::sync_with_stdio(0);
    cin.tie(0);

    int N, M;
    cin >> N >> M;

    for(int i = 1; i <= M; i++){
        int U, V;

```

```
36          cin >> U >> V;
37
38          lista_adjacencia[U].push_back(make_pair(V, i));
39          lista_adjacencia[V].push_back(make_pair(U, i));
40      }
41      maior_id = 0;
42      dfs(1, 1);
43
44
45      int tam = dist[aresta_maior.first] - dist[maior_id] + 1;
46      cout << tam << '\n';
47      cout << aresta_maior.second;
48
49      for(int i = aresta_maior.first; i != maior_id;
50          i = ancestral[i].first){
51
52          cout << ' ' << ancestral[i].second;
53      }
54      cout << '\n';
55
56      return 0;
57  }
```

Código C.1: Código completo do Problema 3 em C++

APÊNDICE D – CÓDIGO COMPLETO DO PROBLEMA 4

```cpp
#include <bits/stdc++.h>
using namespace std;

#define MOD 1000000007

long long int exp_modular(long long int base, long long int exp) {
    long long int parte;
    base %= MOD;

    if (exp == 0) return 1;

    if (exp % 2 == 0) {
        parte = exp_modular(base, exp / 2);
        return (parte * parte) % MOD;
    } else {
        return (base * exp_modular(base, exp - 1)) % MOD;
    }
}

int main(){
    ios::sync_with_stdio(0);
    cin.tie(0);

    int N, Q;
    cin >> N >> Q;

    vector<long long int> X(Q + 1);
    for (int i = 1; i <= Q; i++) cin >> X[i];

    vector<long long int> h(Q + 2, 0);
    long long int inv2 = exp_modular(2, MOD - 2);

    for (int i = Q; i >= 1; i--) {
        h[i] = (h[i + 1] + X[i]) % MOD;
        h[i] = h[i] * inv2 % MOD;
```

```
    }

    vector<long long int> f(N + 1, 0);

    for (int i = 1; i <= Q; i++) {
        long long int g_i = h[i + 1] * inv2 % MOD;
        f[X[i]] = (f[X[i]] + g_i) % MOD;
    }

    f[1] = (f[1] + h[1]) % MOD;

    for (int i = 1; i <= N; i++)
        cout << f[i] % MOD << '\n';

    return 0;
}
```

Código D.1: Código completo do Problema 4 em C++

ANEXO A – Certificado de Participação na Fase Regional da Maratona de Programação SBC 2023

Certificate of Achievement

The 2023 ICPC South America/Brazil First Phase

02 September 2023

icpc International Collegiate Programming Contest

icpc.foundation

Universidade Federal da Fronteira Sul

João Henrique Alves dos Santos

Marco Balestrin

Bernardo Facchi

Andrei Braga, Coach

William B. Poucher

William B. Poucher, Ph. D.
ICPC Executive Director

ANEXO B – Certificado de Participação na Fase Regional da Maratona de Programação SBC 2024

Certificate of Achievement

The 2024 ICPC South America/Brazil First Phase - Maratona SBC de Programação

31 August 2024

Ninety-fifth Place

Universidade Federal da Fronteira Sul

João Henrique Alves dos Santos

Marco Balestrin

Ana Paula Hartmann

Samuel Feitosa, Coach

Andrei Braga, Co-coach

William B. Poucher, Ph. D.
ICPC Executive Director

ANEXO C – Certificado de Participação na Fase Nacional da Maratona de Programação SBC 2024

Certificate of Achievement

The 2024 ICPC South America/Brazil Finals

07 - 09 November 2024

High Honor

Universidade Federal da Fronteira Sul

João Henrique Alves dos Santos

Marco Balestrin

Ana Paula Hartmann

Samuel Feitosa, Coach

Andrei Braga, Co-coach

William B. Poucher, Ph. D.
ICPC Executive Director

ANEXO D – Certificado de Participação na Fase Regional da Maratona de Programação SBC 2025