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

VINICIUS KONCICOSKI

RESOLUÇÃO E ANÁLISE DE PROBLEMAS DA MARATONA DE PROGRAMAÇÃO DA SBC: UMA SELEÇÃO BASEADA NAS EDIÇÕES DE 2021 A 2024

CHAPECÓ
2025

VINICIUS KONCICOSKI

RESOLUÇÃO E ANÁLISE DE PROBLEMAS DA MARATONA DE PROGRAMAÇÃO DA SBC: UMA SELEÇÃO BASEADA NAS EDIÇÕES DE 2021 A 2024

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
2025

Koncicoski, Vinicius

RESOLUÇÃO E ANÁLISE DE PROBLEMAS DA MARATONA DE PROGRAMAÇÃO DA SBC: UMA SELEÇÃO BASEADA NAS EDIÇÕES DE 2021 A 2024 / Vinicius Koncicoski – 2025.

89 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, 2025.

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

VINICIUS KONCICOSKI

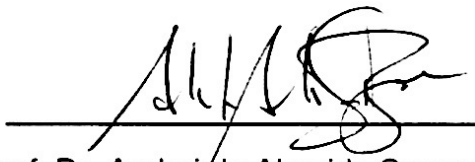
RESOLUÇÃO E ANÁLISE DE PROBLEMAS DA MARATONA DE PROGRAMAÇÃO DA SBC: UMA SELEÇÃO BASEADA NAS EDIÇÕES DE 2021 A 2024

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: 12/12/2025.

BANCA AVALIADORA



Prof. Dr. Andrei de Almeida Sampaio Braga - UFFS

Prof. Dr. Guilherme Dal Bianco - UFFS

Prof. Dr. Samuel da Silva Feitosa - UFFS

DEDICATÓRIA

Dedico este trabalho aos meus familiares, amigos e a todos que, de alguma forma, contribuíram para a minha trajetória, oferecendo apoio, incentivo e inspiração ao longo deste caminho.

"The art of programming is the art of organizing complexity."
(Edsger W. Dijkstra)

AGRADECIMENTOS

Agradeço ao meu professor e orientador, Andrei de Almeida Sampaio Braga, pelo acompanhamento ao longo da minha trajetória acadêmica, pela orientação durante o desenvolvimento deste trabalho e pela organização de todas as etapas realizadas. Sou também grato pelas orientações pessoais, profissionais e enquanto estudante, que foram fundamentais para o meu crescimento.

Agradeço aos professores Samuel da Silva Feitosa e Andrei de Almeida Sampaio Braga pela gestão do Clube de Programação, que ao longo dos anos tem contribuído para a formação e o aprendizado de diversos alunos em programação competitiva, incluindo a mim.

Registro também minha gratidão aos colegas que compuseram as equipes de competição das quais participei. Enfrentamos muitos desafios juntos, tanto nos estudos quanto nas provas, e cada experiência contribuiu para meu desenvolvimento acadêmico e pessoal.

Por fim, deixo um agradecimento especial a todos os professores que fizeram parte da minha vida acadêmica, desde o ensino fundamental até a conclusão da graduação. Cada um contribuiu de alguma forma para minha formação, e sou imensamente grato por todos os aprendizados ao longo dessa jornada.

RESUMO

As competições de programação, como a Maratona de Programação da SBC, são atividades relevantes para os estudantes e profissionais de ciência da computação, pois fomentam a resolução de problemas sob restrições de eficiência e a criação de materiais de estudo especializados. Este trabalho analisa e detalha as soluções de quatro problemas selecionados da Maratona de Programação da SBC, que abordam áreas fundamentais da computação. A metodologia envolveu a seleção de problemas por dificuldade, a implementação das soluções em C++, a validação em um juiz *online* e a análise de complexidade. Como resultado, foram desenvolvidas soluções eficientes aplicando o seguinte: a árvore de Fenwick para consultas em intervalos, o algoritmo de cadeias monotônicas para um problema de geometria computacional, o algoritmo de Dijkstra para uma variação do problema de caminhos mínimos e a técnica de programação dinâmica para um problema de partição. Este trabalho contribui como um recurso detalhado que demonstra a aplicação prática de algoritmos e estruturas de dados, servindo como guia para estudantes em preparação para competições de programação.

Palavras-Chave: Programação Competitiva, Estruturas de Dados, Grafos, Programação Dinâmica, Geometria Computacional.

ABSTRACT

Programming contests, such as the SBC Programming Marathon, are relevant activities for computer science students and professionals, as they foster problem solving under efficiency constraints and the creation of specialized study materials. This work analyzes and details the solutions to four selected problems from the SBC Marathon, which cover fundamental areas of computer science. The methodology involved the selection of problems by difficulty, implementation of the solutions in C++, validation in an online judge, and the complexity analysis. As a result, efficient solutions were developed applying the following: the Fenwick tree for range queries, the monotonic chain algorithm for a computational geometry problem, Dijkstra's algorithm for a variation of the shortest path problem, and the dynamic programming technique for a partition problem. This work contributes as a detailed resource that demonstrates the practical application of algorithms and data structures, serving as a guide for students preparing for programming contests.

Keywords: Competitive Programming, Data Structures, Graphs, Dynamic Programming, Computational Geometry.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

As competições de programação, popularmente conhecidas como maratonas, estabeleceram-se como uma atividade de relevância no cenário da ciência da computação. Elas proporcionam um ambiente onde estudantes e profissionais podem aprimorar suas habilidades na resolução de problemas complexos. A prática regular para esses eventos impulsiona um desenvolvimento que vai além da grade curricular tradicional, estimulando o raciocínio lógico, a criatividade e um domínio profundo sobre algoritmos e estruturas de dados.

O ambiente competitivo exige mais do que apenas uma solução funcional; ele demanda a solução mais eficiente. Com restrições de tempo e memória, os participantes são desafiados a analisar a complexidade de suas próprias ideias e a selecionar a abordagem algorítmica mais adequada para cada cenário. Esse processo de otimização reflete diretamente os desafios encontrados no desenvolvimento de *software* de alto desempenho na indústria e na pesquisa acadêmica. Por trás de cada problema resolvido em uma competição, existe uma base sólida de teoria da computação, que vai desde a análise assintótica até técnicas avançadas de projeto de algoritmos.

Este trabalho tem como objetivo principal analisar e detalhar as soluções para um conjunto de quatro problemas selecionados de edições da Maratona de Programação da Sociedade Brasileira de Computação (SBC). Os problemas foram escolhidos por abordarem áreas distintas e fundamentais da computação, como o processamento de consultas em intervalos com estruturas de dados eficientes, a aplicação de geometria computacional, a busca por caminhos mínimos em grafos e o uso de programação dinâmica. Ao decompor cada solução, desde a conceituação teórica e a escolha da estratégia até a implementação e a análise de complexidade, busca-se criar um material de estudo detalhado que sirva como um recurso de apoio para estudantes que desejam aprofundar seus conhecimentos e se preparar para futuras competições.

1.1 Metodologia

Nesta seção, são detalhados os procedimentos adotados para a elaboração deste trabalho, abrangendo os critérios para a seleção dos problemas analisados e a estratégia de implementação utilizada para desenvolver as soluções. O objetivo é fornecer uma visão sobre como o conteúdo foi escolhido e como as soluções foram desenvolvidas.

Revisão bibliográfica. Para embasar teoricamente as soluções desenvolvidas, foi realizada uma revisão bibliográfica focada em literatura clássica da ciência da computação e em guias de programação competitiva. A consulta a livros e artigos consolidados foi fundamental para garantir a correta aplicação dos algoritmos e das estruturas de dados. Esse processo permitiu verificar a adequação das técnicas e garantir que as implementações estivessem alinhadas com as melhores práticas e a teoria de complexidade de algoritmos.

Seleção de problemas. A seleção dos problemas foi guiada por dois critérios principais: a familiaridade com os problemas, pelo autor já os ter resolvido durante participações em competições, e a busca por problemas de um bom nível de dificuldade e que exigissem a aplicação de técnicas de programação. Especificamente, os problemas foram extraídos de edições da Maratona de Programação da SBC, incluindo as Fases Regionais de 2021, 2023 e 2024, e a Fase Nacional de 2023. Para avaliar a dificuldade de forma objetiva, foram utilizadas as estatísticas oficiais das provas, disponibilizadas pela SBC em seu histórico de resultados. Essas estatísticas, que incluem o número total de submissões e o número de submissões aceitas para cada problema, oferecem um indicador quantitativo do desafio que cada um representou para os competidores.

Com base nesses dados, foram escolhidos problemas que, além de apresentarem um nível de complexidade intermediário, abordassem algoritmos de grafos, estruturas de dados como a árvore de Fenwick, e outras técnicas relevantes no contexto da programação competitiva. O primeiro problema, por exemplo, teve 873 submissões com apenas 78 aceitas (taxa de sucesso de 9%). O segundo problema registrou 346 submissões e 69 aceitas (20% de sucesso). Já o terceiro problema, com 112 submissões e 49 aceitas, apresentou uma taxa de sucesso de 44%. Por fim, o quarto problema teve 814 submissões e 113 foram aceitas, resultando em uma taxa de sucesso de 14%. Esses percentuais indicam que os problemas selecionados não são triviais e exigem um conhecimento técnico para sua resolução.

Estratégia de implementação. Para a implementação das soluções, a linguagem de programação escolhida foi C++. Essa decisão se baseia em três fatores fundamentais. Primeiramente, por ser uma linguagem compilada, o C++ oferece um desempenho computacional muito alto, um requisito essencial em competições de programação onde o tempo de execução é um critério de avaliação. Em segundo lugar, sua robusta Biblioteca Padrão (STL, *Standard Template Library*) provê um vasto conjunto de estruturas de dados e algoritmos otimizados, como vetores, filas de prioridade e funções de ordenação, que são ferramentas indispensáveis na resolução de problemas complexos. Por fim, o C++ possui uma comunidade ativa no cenário da

programação competitiva, o que facilita o acesso a materiais de estudo e bibliotecas públicas, auxiliando no aprendizado e no desenvolvimento das soluções.

Adicionalmente, foram aplicadas práticas comuns em treinamentos para a Maratona de Programação, visando otimizar tanto o tempo de desenvolvimento quanto o de execução. É comum a utilização de um código base (*template*) que inclui a definição de macros para acelerar a escrita de código repetitivo e facilitar a depuração (isso pode ser visto no início dos códigos apresentados nos Apêndices A a D). Também são aplicadas configurações para otimizar as operações de entrada e saída de dados, como o uso de `ios_base::sync_with_stdio(0);` e `cin.tie(0);`. Esses comandos desvinculam os fluxos de entrada e saída do C++ das funções equivalentes da biblioteca C padrão, resultando em uma leitura e escrita de dados significativamente mais rápida, o que pode ser decisivo em problemas com grandes volumes de entrada.

Ferramenta de avaliação de soluções. A validação prática das implementações foi um passo crucial do desenvolvimento deste trabalho. Para este fim, foram utilizados juízes *online*, especificamente a plataforma Codeforces (MIRZAYANOV, 2025), que é amplamente reconhecida na comunidade de programação competitiva. Cada solução foi submetida à plataforma para ser avaliada com um conjunto de casos de teste. A obtenção do veredito *aceito* serviu como uma dupla validação: primeiramente, confirmou que a lógica do algoritmo estava correta e era capaz de lidar com diversos cenários, incluindo casos de borda; em segundo lugar, atestou que a eficiência da implementação e a complexidade do algoritmo eram adequadas para resolver o problema dentro dos limites de tempo e memória, comprovando a viabilidade da solução proposta.

Análise e discussão das soluções. Para cada problema abordado neste trabalho, foi destinada uma seção dedicada à análise e discussão da solução implementada, finalizando cada capítulo com uma avaliação detalhada. Para cada solução, foi realizada uma análise da complexidade computacional, tanto de tempo quanto de memória, justificando a adequação da abordagem aos requisitos de desempenho do problema. Adicionalmente, a seção de discussão explora as limitações de cada solução, examina possíveis variações e estabelece conexões com outros problemas, enriquecendo a compreensão do tema.

1.2 Organização do texto

O presente trabalho é organizado de forma a guiar o leitor através da análise detalhada de quatro problemas de programação competitiva, com os capítulos centrais seguindo uma estrutura padronizada para facilitar a compreensão.

Os Capítulos 2, 3, 4 e 5 são dedicados, cada um, a um problema específico. A estrutura de cada um desses capítulos é dividida em quatro seções principais:

- Conceitos: Apresenta a base teórica e os fundamentos da ciência da computação necessários para compreender a solução, como estruturas de dados e teorias de algoritmos.
- Algoritmo: Detalha o algoritmo central escolhido para a resolução, incluindo a motivação para sua escolha, uma explicação de seu funcionamento e os aspectos de sua implementação.
- Resolução: Descreve o enunciado do problema, a lógica da solução proposta e a aplicação passo a passo do algoritmo para resolvê-lo.
- Análise e discussão da solução: Finaliza o capítulo com uma análise da complexidade de tempo e memória da implementação, seguida por uma discussão sobre suas limitações, possíveis alternativas e comparações com outras técnicas.

O Capítulo 6 apresenta as conclusões do trabalho, consolidando os aprendizados obtidos e as observações sobre as soluções desenvolvidas. Ao final, são listados os apêndices, que contêm os códigos-fonte completos de cada problema abordado.

2. PROBLEMA 1: “NA TRAVE!”

2.1 Conceitos

Esta seção aborda os conceitos fundamentais para a resolução do problema. Partimos da técnica de soma de prefixos para introduzir a necessidade de uma estrutura de dados mais eficiente, a árvore de Fenwick, cujo funcionamento depende de operações bit a bit.

2.1.1 Soma de prefixos

A soma de prefixos é uma estrutura auxiliar baseada em prefixos de um vetor. Cada posição do vetor de soma de prefixos armazena a soma dos elementos do início até aquela posição. Esta técnica permite calcular rapidamente o somatório de intervalos consecutivos de elementos, utilizando a diferença entre dois prefixos (LAAKSONEN, 2020).

2.1.2 Complexidade assintótica

A complexidade assintótica descreve o comportamento do tempo de execução ou uso de memória de um algoritmo em função do tamanho da entrada. As notações mais comuns são O (pior caso), Ω (melhor caso) e Θ (caso médio). A análise assintótica permite prever o desempenho de um algoritmo para entradas muito grandes, desconsiderando constantes e termos de menor ordem (CORMEN et al., 2022).

2.1.3 Operações bit a bit

Operações bit a bit são operações realizadas diretamente sobre a representação binária dos números. No contexto deste trabalho, destaca-se a operação *AND* entre um número i e seu valor negativo $-i$ (denotada por $i \ \& \ -i$), que permite identificar a maior potência de 2 que divide i. Essas operações são fundamentais para

a eficiência da árvore de Fenwick, pois permitem calcular rapidamente os índices a serem atualizados ou consultados (HALIM; HALIM; EFFENDY, 2018).

2.2 Algoritmo

2.2.1 Motivação para a árvore de Fenwick

A árvore de Fenwick, que também é conhecida como *Binary Indexed Tree* (BIT), pode ser utilizada para resolver uma limitação de eficiência que existe na estrutura de soma de prefixos. Uma consulta em um vetor de soma de prefixos possui uma complexidade assintótica de $O(1)$ e a atualização possui uma complexidade de $O(n)$.

Ao adotar a árvore de Fenwick, tanto a consulta quanto a atualização passam a ter complexidade $O(\log n)$. Uma vantagem de utilizar a árvore de Fenwick em vez da estrutura similar árvore de segmentos convencional é a economia de memória: apesar das duas estruturas de dados terem a mesma complexidade assintótica de memória ($O(n)$), a árvore de Fenwick usa, em geral, 4 vezes menos memória que a árvore de segmentos convencional, conforme resumido na Tabela 2.1.

Tabela 2.1: Complexidade de operações e uso de memória

Estrutura	Tempo		Memória
	Busca	Atualização	
Soma de Prefixos	$O(1)$	$O(n)$	$O(n)$
Árvore de Fenwick	$O(\log n)$	$O(\log n)$	$O(n)$
Árvore de Segmentos	$O(\log n)$	$O(\log n)$	$O(n)$

Além das vantagens de tempo de execução e uso de memória, a árvore de Fenwick exige menos código do que a árvore de segmentos: basta um vetor e operações bit a bit para navegar pelos índices, sem precisar de recursão ou estruturas extras. Isso reduz a chance de erros e agiliza a implementação. Nas seções seguintes, cada passo desse procedimento será apresentado em detalhes (LAAKSONEN, 2020).

2.2.2 Funcionamento da árvore de Fenwick

A principal ideia que torna a árvore de Fenwick uma estrutura mais eficiente é a flexibilidade na atualização e na consulta, em comparação com o vetor de soma de prefixos. Em uma estrutura de soma de prefixos, cada posição do vetor armazena o somatório de todos os elementos até aquele índice.

Na prática, ao utilizar uma soma de prefixos, cada índice é responsável por toda a soma acumulada do início até ele. A árvore de Fenwick modifica essa abordagem: cada índice é responsável apenas por um subconjunto dos elementos, computando o somatório desses valores no momento da consulta, em vez de armazenar o resultado final como ocorre na soma de prefixos (HALIM; HALIM; EFFENDY, 2018).

Seja o vetor de valores V:

$$V = [1, 2, 8, 3, 5, 1, 4, 2].$$

O vetor de soma de prefixos correspondente é

$$P = [1, 3, 11, 14, 19, 20, 24, 26],$$

enquanto o vetor interno da árvore de Fenwick é

$$BIT = [1, 3, 8, 14, 5, 6, 4, 26].$$

A Figura 2.1 ilustra como cada posição do vetor da árvore de Fenwick (vetor BIT) é responsável pelo somatório de um intervalo específico de elementos do vetor V, organizando os dados de forma hierárquica e eficiente.

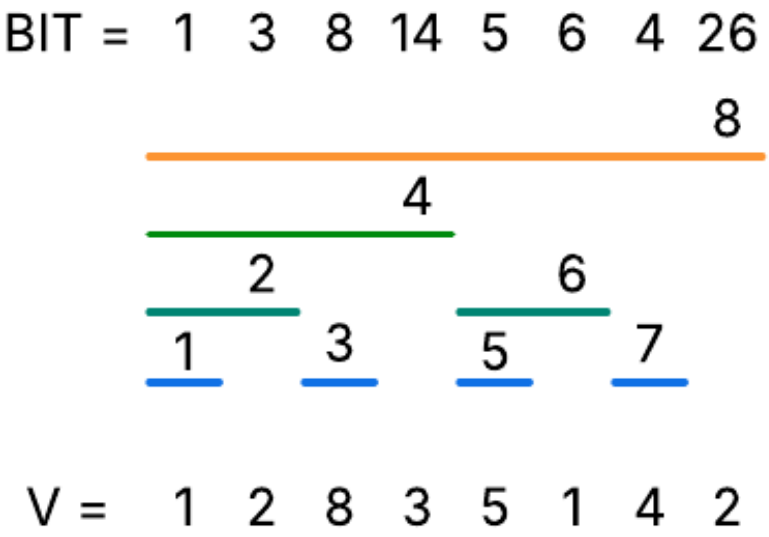


Figura 2.1: Intervalo de responsabilidade de cada índice do vetor *BIT* em relação aos elementos do vetor *V*

Agora será explorada a regra que define o conjunto de elementos do vetor *V* pelo qual é responsável um determinado índice *i* do vetor *BIT*. A árvore de Fenwick será considerada indexada no intervalo [1, *N*], o que facilita tanto a explicação quanto a implementação, permitindo o uso de operações de bits para calcular rapidamente os índices necessários nas operações de consulta e atualização.

A responsabilidade de cada índice pode ser determinada observando quantas vezes o número *i* é divisível por 2. De forma prática:

- Índices ímpares armazenam apenas o próprio valor.
- O índice 6, por exemplo, pode ser dividido uma vez por 2, indicando que é responsável por 2^1 elementos: os índices 5 e 6.
- O índice 4 é divisível duas vezes por 2, portanto é responsável por $2^2 = 4$ elementos: os índices: 1, 2, 3 e 4.
- Além disso, índices que são potências de 2 (como 2, 4, 8, 16, etc.) são responsáveis por todos os elementos desde o início do vetor até eles mesmos.

Para facilitar o cálculo da quantidade de elementos sob responsabilidade de um índice, utiliza-se uma operação de bits: o *AND* entre o índice *i* e seu valor negativo $-i$. Essa operação revela a maior potência de 2 que divide o índice (LAAKSONEN, 2020).

Por exemplo, para $i = 10$:

$$(10)_{10} = (1010)_2, \quad (-10)_{10} = (0110)_2$$

onde $(i)_{10}$ e $(i)_2$ indicam a representação do número i na base 10 e na base 2, respectivamente. Realizando a operação bit a bit:

$$(1010)_2 \ \& \ (0110)_2 = (0010)_2 = (2)_{10}$$

O resultado é o número 2, indicando que o índice 10 é responsável por 2 elementos.

Essa lógica é aplicada tanto nas operações de consulta quanto de atualização. Ambas as operações apresentam complexidade de $O(\log n)$, permitindo uma implementação compacta e eficiente, ideal para maratonas de programação.

2.2.3 Implementação da árvore de Fenwick

A seguir, será detalhada a implementação da árvore de Fenwick, abordando as operações de consulta e atualização. A apresentação será acompanhada de exemplos em linguagem C++ e referências diretas às linhas e variáveis utilizadas no código.

Consulta

O procedimento utilizado para realizar uma consulta e obter o somatório dos elementos no intervalo $[1, i]$ está ilustrado no Código 2.1. Nele, uma variável auxiliar *sum* (linha 2) atua como acumulador. A operação inicia no índice i e, enquanto $i > 0$, acumula o valor (linha 4) e, em seguida, atualiza i para $i - (i \ \& \ -i)$ (linha 5). Este processo se repete até que i se torne igual a zero.

Essa abordagem resulta em um código compacto e de fácil implementação em diversas linguagens de programação.

```
1 int query(int i) {
2     int sum = 0;
3     while (i > 0) {
4         sum += bit[i];
5         i -= (i & -i);
6     }
7     return sum;
```

```
8 }
```

Código 2.1: Função de consulta (`query`) em C++

Atualização

A operação de atualização está ilustrada no Código 2.2. Nela, considera-se o índice *i* que se deseja atualizar e o valor do incremento, denominado *delta* (segundo parâmetro indicado na linha 1). A cada iteração, adiciona-se *delta* ao índice atual (linha 3) e atualiza-se *i* para $i + (i \ \& \ -i)$ (linha 4). O processo continua enquanto *i* for menor ou igual ao tamanho *N* do vetor.

```
1 void update(int i, int delta) {
2     while (i <= N) {
3         bit[i] += delta;
4         i += (i & -i);
5     }
6 }
```

Código 2.2: Função de atualização (`update`) em C++

2.3 Resolução

2.3.1 Descrição do problema

O Problema “Na Trave!” apresenta uma competição chamada Internacional Competição de Pintores de Carros (ICPC). Nesta competição, os times podem participar uma quantidade limitada de vezes devido à intoxicação causada pela exposição prolongada à tinta. A etapa regional desta competição garante a vaga para a ICPC. Apenas os times que alcançam as melhores *x* colocações na etapa regional se classificam para a ICPC. O valor da quantidade de vagas disponíveis *x* pode variar a cada ano.

Os competidores de um time terão o sentimento de “má sorte” se, em sua última participação na competição, não conseguirem se classificar. Com base na colocação p_i que eles ficaram em sua última participação, eles vão continuar acompanhando f_i anos das próximas edições do evento. Se nessas próximas f_i edições a quantidade de vagas da edição em um determinado ano for maior que p_i, eles irão

se sentir com má sorte, pois, se eles estivessem participando naquele ano, teriam se classificado para a ICPC.

A resposta esperada para o problema é, dado uma lista com o número de vagas disponíveis por ano, saber quantos anos cada competidor se sentiu azarado.

2.3.2 Descrição da entrada do problema

A entrada do problema fornece as informações de cada caso, com base nas quais é possível desenvolver uma solução e apresentar uma resposta.

A primeira linha da entrada contém dois números inteiros Y e N ($1 \leq Y$, $N \leq 3 \times 10^5$), que representam, respectivamente, a quantidade de edições da competição e a quantidade de antigos competidores para os quais a resposta deve ser calculada. A linha seguinte apresenta uma lista de números inteiros x_1, x_2, ..., x_Y ($0 \leq x_i \leq 10^5$), indicando a quantidade de vagas disponíveis na etapa regional em cada ano.

Em seguida, cada uma das próximas N linhas contém três inteiros a_i, p_i e f_i, onde ($1 \leq a_i \leq Y$), ($1 \leq p_i \leq 10^5$) e ($0 \leq f_i \leq Y - a_i$). O valor a_i indica o ano da última participação de um competidor, p_i corresponde à colocação obtida por sua equipe nessa participação, e f_i informa por quantos anos o competidor continuou acompanhando os resultados após a última participação.

2.3.3 Descrição da saída do problema

Para a saída, deve-se imprimir N linhas, onde cada linha representa quantos anos aquele respectivo participante se sentiu azarado. Essa linha deve conter um número inteiro.

2.3.4 Solução ingênua para o problema

Inicialmente, é possível considerar uma solução ingênua para o problema, com o objetivo de compreender melhor o que deve ser calculado e como isso pode contribuir para o desenvolvimento de uma solução mais eficiente.

Após a leitura dos valores correspondentes à quantidade de edições da competição e à quantidade de competidores, são lidas as quantidades de vagas disponí-

veis na etapa regional em cada ano, armazenando-as em um vetor *vagas*. Em seguida, deve-se processar uma consulta para cada competidor, cujo objetivo é calcular o número de anos em que o competidor se sentiu azarado. É importante destacar que a_i representa o ano da última participação de um competidor, p_i indica a colocação obtida pelo time na última participação e f_i corresponde ao número de anos em que o competidor continuou acompanhando os resultados após a sua última participação (veja a Seção 2.3.2). Para cada consulta, realiza-se uma iteração sobre o vetor *vagas*, percorrendo cada posição $j = a_i + 1, ..., a_i + f_i$, contando-se as ocorrências em que p_i é menor ou igual a *vagas*[j], o número de vagas da etapa regional no ano j.

Um detalhe importante que não é explicitado no enunciado do problema é que, caso o competidor tenha se classificado em sua última participação no ano a_i, ele não se sentirá azarado em nenhuma circunstância. Dessa forma, essa verificação pode ser realizada no início de cada consulta, evitando-se, assim, o custo computacional de iterar sobre o vetor *vagas*.

Com base nessas informações, caso o competidor tenha se classificado em sua última participação, deve-se imprimir 0. Caso contrário, deve-se imprimir a quantidade de anos em que o competidor se sentiu azarado.

2.3.5 Análise e discussão da solução ingênua

Para analisar essa solução, é necessário saber dos limites de tempo de execução e uso de memória do problema, que são, respectivamente, 0,7 segundos e 1 GB. Além disso, vale lembrar que a quantidade de edições da competição e a quantidade de antigos competidores para os quais serão calculadas as respostas são no máximo 3×10^5 (informações dadas na Seção 2.3.2).

É possível observar que essa solução não apresenta eficiência suficiente para atender ao limite de tempo do problema, pois, no pior caso, seria necessário, para cada antigo competidor (3×10^5), percorrer toda a lista de edições (3×10^5), resultando em um total de 9×10^{10} operações, um número inviável para execução em 0,7 segundos.

A solução ingênua para o problema fornece uma informação importante para a construção de uma abordagem mais eficiente: o fato de que é necessário realizar repetidamente iterações em intervalos de uma lista para computar as respostas. Em outras palavras, o problema envolve múltiplas consultas em intervalos. Existem estruturas de dados mais adequadas para esse tipo de operação, como as árvores de segmentos e a árvore de Fenwick.

2.3.6 Solução eficiente utilizando árvore de Fenwick

Para analisar o funcionamento de uma consulta, observa-se que, para cada participante, é necessário consultar um intervalo de $a_i + 1$ até $a_i + f_i$. Para que a árvore de Fenwick retorne o resultado correto, o vetor auxiliar sobre o qual ela opera deve indicar, para cada ano, se o competidor se sentiu azarado ou não. Especificamente, caso a quantidade de vagas no ano seja maior ou igual à colocação p_i obtida pelo competidor em sua última participação, o ano é marcado com 1; caso contrário, com 0. A soma dos valores nesse intervalo indicará exatamente o número de vezes em que o competidor se sentiu azarado.

Observa-se que essa informação está diretamente relacionada ao valor de p_i. Assim, busca-se estruturar os dados de modo que a árvore de Fenwick seja capaz de, a qualquer momento, responder consultas sobre a quantidade de anos com vagas maior ou igual a p_i em um intervalo.

A entrada é dividida em duas partes: as inserções, correspondentes à quantidade de vagas em cada ano, e as consultas, que consistem nos N registros (a_i, p_i, f_i). O objetivo é responder a cada consulta de forma eficiente, utilizando uma árvore de Fenwick de soma.

Um ponto relevante é que, ao garantir que todos os anos com quantidade de vagas maior ou igual a p_i já estejam registrados na árvore de Fenwick, torna-se possível responder corretamente à consulta associada a esse p_i. Essa observação motiva uma estratégia baseada em ordenação e atualização incremental.

Para ilustrar a ideia, considere o exemplo a seguir. A competição ocorreu durante 6 anos, com as seguintes quantidades de vagas:

$$vagas = [1, 3, 4, 2, 8, 5]$$

Considere um competidor cuja última participação foi no ano $a_i = 2$, com colocação $p_i = 4$, e que acompanhou as próximas $f_i = 3$ edições. O intervalo de anos observados é dado por $[a_i + 1, a_i + f_i] = [3, 5]$. O objetivo é determinar, entre os anos 3 e 5, quantas vezes a quantidade de vagas foi maior ou igual a 4 (isto é, p_i).

Para isso, pode-se construir um vetor auxiliar que representa essa condição:

$$auxiliar = [0, 0, 1, 0, 1, 1]$$

Neste vetor, os valores 1 indicam os anos em que havia pelo menos 4 vagas disponíveis. Observa-se que nos anos 3 (valor 4), 5 (valor 8) e 6 (valor 5), essa

condição foi satisfeita. A soma no intervalo $[3, 5]$ resulta em $1 + 0 + 1 = 2$, valor correspondente ao resultado esperado da consulta.

Observa-se que todas as consultas cujo valor de p_i seja maior ou igual a 4 retornarão a resposta correta com base no vetor auxiliar construído. No entanto, se for necessário realizar uma consulta com o valor $p_i = 3$, por exemplo, o vetor auxiliar ainda não contém todos os valores adequadamente atualizados. Para resolver essa limitação, aplica-se uma estratégia baseada em ordenação. Após concluir todas as consultas com $p_i = 4$, a próxima inserção a ser processada será aquela correspondente ao próximo valor na ordem decrescente, neste caso, seria o valor 3, que representa o segundo ano, com base no exemplo considerado.

Com isso, o vetor auxiliar torna-se:

$$auxiliar = [0, 1, 1, 0, 1, 1]$$

A partir dessa atualização, as consultas com $p_i = 3$ passam a retornar o valor correto. Esse processo é repetido até que todas as consultas tenham sido processadas. Após o término, a saída final é ordenada para respeitar a ordem original de entrada.

O exemplo anterior ilustra que não é necessário preencher toda a árvore de Fenwick antecipadamente. Em vez disso, pode-se ordenar as inserções (anos com suas respectivas quantidades de vagas) em ordem decrescente de vagas, bem como as consultas em ordem decrescente de p_i. Em cada iteração, enquanto a quantidade de vagas de uma inserção for maior ou igual ao valor de p_i da próxima consulta, atualiza-se a árvore de Fenwick com essa inserção. A consulta é então realizada com base na árvore atualizada.

Essa abordagem permite processar todas as consultas de forma eficiente, garantindo que os dados relevantes para cada p_i estejam disponíveis no momento da consulta.

2.3.7 Implementação da solução eficiente

A seguir, são destacados os principais trechos do código da solução, explicando o papel de cada um na resolução. Essa versão do código foi desenvolvida visando uma melhor didática e compreensão do mesmo.

Existem três estruturas principais, apresentadas no Código 2.3:

- `consulta`: armazena os dados de cada consulta (ano da última participação do competidor, colocação na última participação, anos acompanhados, índice original e resposta);
- `insercao`: representa a quantidade de vagas em um determinado ano (quantidade de vagas e ano);
- `operacao`: estrutura auxiliar para processar inserções e consultas de forma ordenada, facilitando o processamento.

```
1  struct consulta {
2      int ultimo_ano;
3      int colocacao;
4      int anos_acompanhados;
5      int indice;
6      int resposta;
7  };
8
9  struct insercao {
10     int quantidade_vagas;
11     int ano;
12 };
13
14 struct operacao {
15     int valor_ordenacao;
16     int tipo_operacao;
17     struct consulta consulta;
18     struct insercao insercao;
19 };
```

Código 2.3: Estruturas auxiliares em C++

Existem também as funções `update` e `query`, que são responsáveis, respectivamente, pela atualização e pela consulta da árvore de Fenwick (mais detalhes sobre o funcionamento podem ser encontrados na Seção 2.2.3).

As inserções são lidas e reunidas em uma lista de operações, como mostrado no Código 2.4:

```
1  for (int ano = 1; ano <= y; ano++)
2  {
3      cin >> quantidade_vagas;
4      vagas.push_back(quantidade_vagas);
5
```

```
6       operacao op;
7       insercao ins;
8
9       ins.quantidade_vagas = quantidade_vagas;
10      ins.ano = ano;
11
12      op.valor_ordenacao = quantidade_vagas;
13      op.tipo_operacao = INSERCAO;
14      op.insercao = ins;
15
16      operacoes.push_back(op);
17  }
```

Código 2.4: Leitura das inserções para a criação da lista de operações em C++

Em seguida, as consultas são lidas e adicionadas à mesma lista de operações, como visto no Código 2.5:

```
1  for (int indice = 0; indice < n; indice++)
2  {
3       cin >> ultimo_ano >> colocacao >> anos_acompanhados;
4
5       operacao op;
6       consulta con;
7
8       con.ultimo_ano = ultimo_ano;
9       con.colocacao = colocacao;
10      con.anos_acompanhados = anos_acompanhados;
11      con.indice = indice;
12
13      op.valor_ordenacao = colocacao;
14      op.tipo_operacao = CONSULTA;
15      op.consulta = con;
16
17      operacoes.push_back(op);
18  }
```

Código 2.5: Leitura das consultas e extensão da lista de operações em C++

As operações são ordenadas de acordo com o valor relevante (quantidade de vagas para a operação de inserção ou colocação para a operação de consulta), em ordem decrescente. Em caso de igualdade do valor relevante, a operação de inserção virá antes na ordenação. Isso vai garantir que, ao processar uma consulta

com colocação p_i, todas as inserções com quantidade de vagas maior ou igual a p_i já tenham sido realizadas na árvore de Fenwick. Esta ordenação é descrita no Código 2.6:

```
bool ordenacao_operacao(operacao a, operacao b)
{
    if (a.valor_ordenacao == b.valor_ordenacao) {
        return a.tipo_operacao < b.tipo_operacao;
    }
    return a.valor_ordenacao > b.valor_ordenacao;
}

sort(operacoes.begin(), operacoes.end(), ordenacao_operacao);
```

Código 2.6: Ordenação da lista de operações em C++

Na sequência, como visto no Código 2.7, são processadas as operações ordenadas fazendo o seguinte:

- Para cada inserção: Atualiza a árvore de Fenwick.
- Para cada consulta: Verifica se o competidor se classificou em sua última participação. Se sim, a resposta é zero. Caso contrário, utiliza a árvore de Fenwick para contar rapidamente os anos em que ele se sentirá azarado.

```
for (auto operacao : operacoes)
{
    if (operacao.tipo_operacao == INSERCAO)
    {
        update(operacao.insercao.ano, 1);
    }
    else if (operacao.tipo_operacao == CONSULTA)
    {
        consulta con = operacao.consulta;

        if (con.colocacao <= vagas[con.ultimo_ano-1])
        {
            con.resposta = 0;
            resultados.push_back(con);
        }
        else
        {
```

```
18              int resposta = query(con.ultimo_ano + con.
                    anos_acompanhados) - query(con.ultimo_ano);
19              con.resposta = resposta;
20              resultados.push_back(con);
21          }
22      }
23  }
```

Código 2.7: Processamento das operações baseadas em seu tipo em C++

Por fim, como mostrado no Código 2.8, as respostas são ordenadas de volta seguindo a ordem original das consultas e depois impressas:

```
1  bool ordenacao_resposta(consulta a, consulta b)
2  {
3      return a.indice < b.indice;
4  }
5
6  sort(resultados.begin(), resultados.end(), ordenacao_resposta);
7
8  for (auto consulta : resultados) {
9      cout << consulta.resposta << endl;
10 }
```

Código 2.8: Impressão das respostas em C++

O código completo utilizado para resolver este problema pode ser encontrado no Apêndice A.

2.4 Análise e discussão da solução eficiente

2.4.1 Análise de complexidade da solução

Para avaliar a complexidade do algoritmo, consideram-se três partes principais do código: a ordenação do vetor de operações, o processamento das operações e a ordenação das respostas.

O vetor de operações possui tamanho $Y + N$. A ordenação desse vetor demanda tempo

$$O\big((Y+N)\log(Y+N)\big).$$

Em seguida, processam-se as operações por meio da árvore de Fenwick, que vai ser armazenada em um vetor de tamanho Y. Cada inserção apresenta complexidade $O(\log Y)$, totalizando

$$Y \times O(\log Y) = O(Y \log Y),$$

enquanto cada consulta utiliza duas vezes a função `query`, resultando em tempo

$$N \times 2 \times O(\log Y) = O(N \log Y).$$

Portanto, o processamento das operações leva tempo

$$O((Y + N) \log Y).$$

Por fim, o vetor de respostas tem tamanho N. A ordenação desse vetor é feita em tempo

$$O(N \log N).$$

Como $\log Y$ e $\log N$ situam-se na mesma ordem de grandeza de $\log(Y+N)$, o termo dominante continua sendo o da ordenação inicial. Assim, a complexidade total do algoritmo pode ser simplificada para

$$O((Y + N) \log(Y + N)).$$

Como $1 \leq Y, N \leq 3 \times 10^5$, isso resulta em um total aproximado de

$$O((6 \times 10^5) \log(6 \times 10^5)) \approx 1,15 \times 10^7$$

operações, um número viável para a execução em 0,7 segundos.

Em termos de uso de memória, alocam-se quatro vetores principais:

- O vetor de vagas, de tamanho Y;
- A lista de operações, de tamanho $Y + N$;
- O vetor interno da árvore de Fenwick, de tamanho Y;
- A lista de respostas, de tamanho N.

Consequentemente, o consumo total de memória é

$$O(Y) + O(Y + N) + O(Y) + O(N) \;=\; O(Y + N).$$

2.4.2 Discussão da solução

A árvore de Fenwick atende de forma eficiente consultas de soma e atualizações pontuais, mas tem limitações quando se trata de outros tipos de operação em intervalos. Por exemplo, para obter o valor máximo ou mínimo em um intervalo, a árvore de Fenwick não fornece esse recurso diretamente. Nesses casos, recomenda-se o uso de árvores de segmentos, que permitem consultas de máximo e mínimo em $O(\log n)$, sendo n o número de elementos da entrada.

Além disso, a árvore de Fenwick clássica suporta apenas atualizações em posições isoladas (por meio da função `update`). Quando for preciso modificar todos os elementos de um intervalo (por exemplo, incrementar cada valor em um segmento), essa estrutura não é adequada. Para tais situações, as árvores de segmentos oferecem a flexibilidade necessária.

3. PROBLEMA 2: “GRANDE TRATADO DA BYTELÂNDIA”

3.1 Conceitos

Esta seção detalha os conceitos de geometria computacional que fundamentam a solução. Abordamos como a divisão de territórios, baseada na Distância Euclidiana, forma um Diagrama de Voronoi, e como a identificação de regiões infinitas neste diagrama é simplificada pelo cálculo do Fecho Convexo.

3.1.1 Distância Euclidiana

A distância euclidiana é uma medida fundamental na geometria que calcula a separação direta entre dois pontos em um espaço *n*-dimensional. Essa métrica é obtida pela aplicação do teorema de Pitágoras, considerando as diferenças entre as coordenadas correspondentes dos pontos. No plano bidimensional, por exemplo, a distância entre os pontos $A = (x_1, y_1)$ e $B = (x_2, y_2)$ é dada por:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Essa medida é amplamente utilizada em algoritmos de geometria computacional, especialmente em problemas que envolvem proximidade, como a construção de diagramas de Voronoi e a determinação do par de pontos mais próximos em um conjunto. A distância euclidiana fornece uma base para definir métricas e realizar comparações espaciais em diversas aplicações computacionais (PREPARATA; SHAMOS, 1993).

3.1.2 Diagrama de Voronoi

O diagrama de Voronoi é uma estrutura que particiona o plano em regiões baseadas na proximidade a um conjunto específico de pontos chamados de sítios. Cada região, ou célula de Voronoi, contém todos os pontos do plano que estão mais próximos de um sítio específico do que de qualquer outro. As fronteiras entre essas células são formadas por segmentos de reta equidistantes entre pares de sítios (BERG et al., 2013).

3.1.3 Fecho Convexo

O fecho convexo de um conjunto de pontos no plano é o menor polígono convexo que contém todos esses pontos. Visualmente, pode ser imaginado como um elástico esticado ao redor dos pontos, envolvendo-os completamente. Esse conceito é essencial na geometria computacional, pois fornece uma forma de simplificar e analisar conjuntos de pontos (HALIM; HALIM; EFFENDY, 2020).

3.2 Algoritmo

3.2.1 Motivação para o algoritmo de cadeias monotônicas

Na resolução do problema, foi utilizado o algoritmo de cadeias monotônicas (HALIM; HALIM; EFFENDY, 2020) – também conhecido como algoritmo de Andrew – para construir o fecho convexo. Essa escolha se justifica pela necessidade de incluir os pontos colineares na borda do fecho.

Ao contrário do algoritmo de varredura de Graham (HALIM; HALIM; EFFENDY, 2020), cuja implementação padrão é mais difícil de adaptar para incluir pontos colineares na borda do fecho convexo, o algoritmo de cadeias monotônicas permite o tratamento adequado destes pontos de maneira mais fácil, garantindo que a fronteira seja representada corretamente. Ambos os algoritmos possuem complexidade $O(n \log n)$, mas o de cadeias monotônicas é mais simples de implementar e menos propenso a erros relacionados à colinearidade, sendo mais indicado para este tipo de problema.

3.2.2 Funcionamento do algoritmo de cadeias monotônicas

O algoritmo de cadeias monotônicas descrito a seguir considera a inclusão de pontos colineares na construção do fecho convexo. Isso garante que todos os pontos na fronteira do fecho sejam preservados.

Inicialmente, os pontos do plano são ordenados, primeiro pela coordenada x e, em caso de empate, pela coordenada y. Essa ordenação é essencial para a construção eficiente do fecho convexo.

A construção do fecho convexo é dividida em duas etapas, a construção do casco inferior (parte inferior do fecho convexo) e do casco superior (parte superior do fecho convexo). Após a construção de ambos os cascos, ocorre a junção dos cascos para o retorno do conjunto de pontos que representa o fecho convexo.

Para a construção do casco inferior, cria-se uma pilha vazia, que irá armazenar, em ordem, os pontos que formam o casco inferior até o ponto atual. Insere-se o primeiro ponto ordenado, ele será sempre parte do casco inferior, pois não há curva formada ainda.

Cada ponto *p* da lista ordenada é processado sequencialmente para verificar a manutenção da convexidade local. Enquanto houver pelo menos dois pontos na pilha, realiza-se o teste de orientação com os três últimos pontos (penúltimo, último e *p*). Se o resultado do teste indicar uma curva no sentido horário (valor negativo), o ponto no topo da pilha é removido, pois compromete a convexidade (Figura 3.2). Caso o resultado seja zero (colinearidade) ou positivo (curva no sentido anti-horário), o ponto *p* é adicionado ao topo da pilha (Figura 3.1).

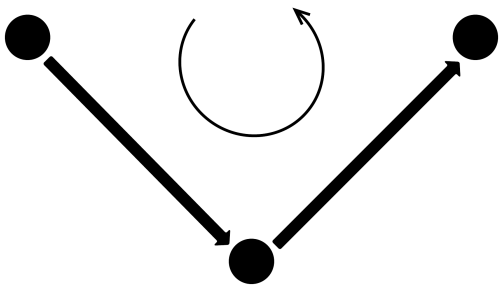


Figura 3.1: Teste de orientação: Sentido anti-horário

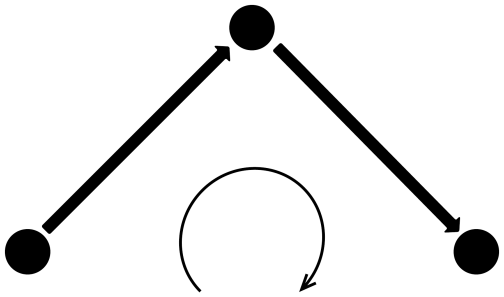


Figura 3.2: Teste de orientação: Sentido horário

O teste de orientação, comumente representado pela sigla `ccw` (do inglês *counter-clockwise*, que significa “sentido anti-horário”), é utilizado para determinar a orientação relativa entre três pontos consecutivos *A*, *B* e *C* no plano cartesiano. Esse teste é baseado no cálculo do produto vetorial entre os vetores $\overrightarrow{AB}$ e $\overrightarrow{AC}$, conforme a expressão abaixo:

$$ccw(A, B, C) = (B_x - A_x) \cdot (C_y - A_y) - (B_y - A_y) \cdot (C_x - A_x)$$

O valor resultante desse cálculo indica a orientação dos pontos: se for positivo, os pontos estão dispostos no sentido anti-horário; se negativo, no sentido horário; e se igual a zero, os pontos são colineares. Esse teste é fundamental para garantir que apenas os vértices que mantêm a convexidade local sejam mantidos durante a construção do fecho convexo.

Após processar todos os pontos para o casco inferior, procede-se de modo análogo (percorrendo os pontos em ordem reversa) para construir o casco superior. Por fim, concatenam-se as duas partes (omitindo os pontos extremos duplicados) para obter o fecho convexo completo.

3.2.3 Implementação do algoritmo de cadeias monotônicas

Na implementação do algoritmo de cadeias monotônicas, é utilizado o tipo `ii` como uma abreviação para `pair<int, int>`, representando um ponto no plano cartesiano. No Código 3.1, a função `ccw` emprega o tipo `ii` para representar os pontos `a`, `b` e `c`, realizando o teste de orientação conforme descrito na Seção 3.2.2.

O acesso às coordenadas de um ponto `p` do tipo `ii` é feito por meio de `p.first` para a coordenada *x* e `p.second` para a coordenada *y*. Com o objetivo de tornar o código mais conciso, é comum definir macros como `#define f first` e `#define s second`, permitindo o uso de `p.f` e `p.s`.

```
1 bool ccw(ii a, ii b, ii c) {
2     return (b.f - a.f) * (c.s - a.s) - (b.s - a.s) * (c.f - a.f) >=
          0;
3 }
```

Código 3.1: Função do teste de orientação em C++

A implementação do algoritmo, mostrada no Código 3.2, foi baseada na versão disponível no livro de Halim, Halim e Effendy (2020). Na linha 5, é realizada a ordenação do vetor de pontos, etapa fundamental para a construção eficiente do fecho convexo. Em seguida, na linha 6, inicia-se a iteração sobre os pontos ordenados, do início ao fim, com o objetivo de construir o casco inferior. Nas linhas 8 e 9, aplica-se a função `ccw` aos três últimos pontos da pilha para verificar se a orientação entre eles mantém a convexidade desejada. Enquanto a curva seja no sentido horário ou colinear, o ponto no topo da pilha é removido. Na sequência, o ponto atual é inserido no topo da pilha. A pilha, nesse caso, é representada de forma implícita por meio da manipulação do índice *k* sobre o vetor *H*.

A partir da linha 11, realiza-se a construção do casco superior, iterando-se agora os pontos em ordem reversa. O funcionamento é análogo ao do casco inferior, com as mesmas verificações de orientação nas linhas 12 e 13. A diferença está na direção da iteração, que percorre os pontos do final para o início do vetor ordenado.

Na linha 15, a função `resize` é utilizada para ajustar o tamanho do vetor *H*, removendo os pontos duplicados nas extremidades, uma vez que o primeiro e o

último ponto são adicionados duas vezes (uma em cada casco). Por fim, na linha 16, retorna-se o vetor *H*, que contém os pontos do fecho convexo final em ordem.

```
1  vector<ii> CH_Andrew(vector<ii> &v)
2  {
3      int n = v.size(), k = 0;
4      vector<ii> H(2*n);
5      sort(v.begin(), v.end());
6      for (int i = 0; i < n; ++i)
7      {
8          while ((k >= 2) && !ccw(H[k-2], H[k-1], v[i])) --k;
9          H[k++] = v[i];
10     }
11     for (int i = n-2, t = k+1; i >= 0; --i) {
12         while ((k >= t) && !ccw(H[k-2], H[k-1], v[i])) --k;
13         H[k++] = v[i];
14     }
15     H.resize(k);
16     return H;
17 }
```

Código 3.2: Algoritmo do fecho convexo em C++

3.3 Resolução

3.3.1 Descrição do problema

O Problema “Grande Tratado da Bytelândia” trata da divisão territorial entre os reinos de um continente fictício. Considera-se que o mundo de Bytelândia é representado por um plano infinito, onde cada reino possui uma única capital. O tratado estabelece que cada ponto do plano pertence ao reino cuja capital está mais próxima em linha reta (distância euclidiana). Em situações de empate, ou seja, quando um ponto está à mesma distância de duas ou mais capitais, esse ponto é considerado parte da fronteira entre os reinos envolvidos.

A partir dessa regra de divisão, observa-se que certos reinos podem ficar completamente cercados por outros, enquanto alguns podem possuir territórios ilimitados, estendendo-se infinitamente em alguma direção. Essa diferença tem gerado

insatisfação entre os monarcas, especialmente por parte daqueles cujos reinos ficaram confinados a áreas limitadas.

Diante disso, os governantes solicitaram uma análise baseada nas coordenadas das capitais dos reinos. O objetivo é identificar, com base nas regras do tratado, quais reinos possuem territórios infinitos.

3.3.2 Descrição da entrada do problema

A entrada consiste inicialmente em um único inteiro N ($2 \leq N \leq 10^5$), representando a quantidade de reinos. Em seguida, são fornecidas N linhas, cada uma contendo dois inteiros X e Y ($0 \leq X,\ Y \leq 10^4$), que representam as coordenadas da capital de um reino. As capitais são listadas em ordem crescente de identificador, de 1 até N, e cada uma ocupa uma posição única no plano, ou seja, não há duas capitais com as mesmas coordenadas. Considera-se ainda que o tamanho de cada capital é desprezível.

3.3.3 Descrição da saída do problema

A saída deve conter uma única linha composta por uma sequência de inteiros, separados por espaço, representando os identificadores dos reinos que possuem territórios infinitos de acordo com o tratado de divisão territorial. Os identificadores devem ser apresentados em ordem crescente. É garantido que sempre existirá ao menos um reino com território infinito.

3.3.4 Solução para o problema

A divisão territorial entre os reinos de Bytelândia, conforme descrita no enunciado do problema, corresponde geometricamente à construção de um diagrama de Voronoi no plano infinito. Nesse diagrama, cada ponto do plano pertence à célula de Voronoi da capital mais próxima, e a fronteira entre duas células é definida pelo conjunto de pontos equidistantes entre duas capitais.

A segunda entrada de exemplo do problema, utilizada para ilustrar a Figura 3.3, consiste em seis linhas, cada linha contendo um dos seguintes pares de coordenadas: (2, 1), (3, 3), (1, 4), (4, 5), (6, 3) e (4, 3). Cada par de coordenadas representa

a posição de uma capital no plano. A figura exibe o diagrama de Voronoi gerado a partir dessas capitais, ilustrando visualmente como os territórios são divididos segundo a menor distância euclidiana e permitindo observar quais regiões se expandem indefinidamente.

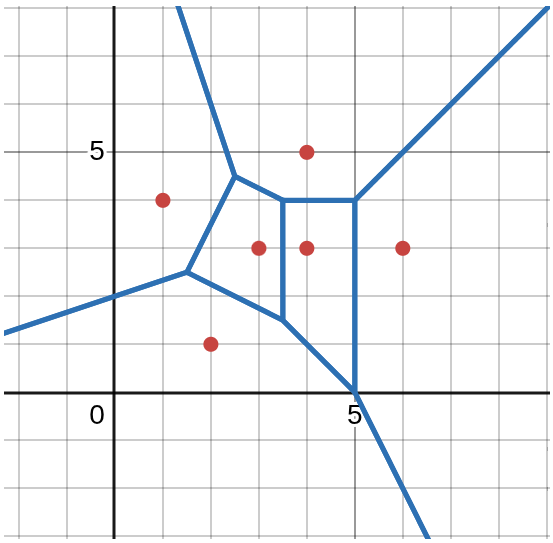


Figura 3.3: Diagrama de Voronoi do segundo exemplo de entrada para o problema

Ao observar um diagrama de Voronoi em um plano, nota-se que apenas os pontos que estão na “borda” do conjunto de capitais podem originar regiões infinitas. Isso ocorre porque as células internas são completamente cercadas por outras células, enquanto as células das capitais que estão nas extremidades se expandem indefinidamente em direção ao infinito. Dessa forma, a solução do problema reduz-se a identificar quais capitais se encontram na “borda” do conjunto de pontos, ou seja, na fronteira do fecho convexo das capitais.

3.3.5 Implementação da solução

A solução para o problema de identificar reinos com territórios infinitos baseia-se na propriedade de que apenas as capitais que compõem a fronteira do fecho convexo do conjunto de todas as capitais podem ter territórios infinitos no diagrama de Voronoi. Para isso, foi utilizado o algoritmo de cadeias monotônicas, conforme detalhado na Seção 3.2.2.

Inicialmente, os pontos das capitais são lidos da entrada e armazenados em um vetor (`pontos`). Para cada capital, além de armazenar suas coordenadas, é fundamental guardar seu identificador original, pois a saída do problema requer os identificadores dos reinos com territórios infinitos. Para isso, utiliza-se um mapa

(`mapa_identificadores`), que associa as coordenadas de um ponto ao seu identificador. É utilizado o tipo `ii` como uma abreviação para `pair<int, int>`, representando as coordenadas do ponto, e a macro `#define pb push_back` para se referir à operação de inserção (*push_back*) no vetor `pontos`, como pode ser visto no Código 3.3.

```
map<ii, int> mapa_identificadores;

for (int i = 0; i < n; i++)
{
    int x, y;
    cin >> x >> y;
    pontos.pb(ii(x, y));
    mapa_identificadores[ii(x, y)] = i + 1;
}
```

Código 3.3: Leitura da entrada e mapeamento de identificadores em C++

Em seguida, o algoritmo `CH_Andrew` é chamado para calcular o fecho convexo a partir do vetor de pontos (`pontos`). A função `CH_Andrew`, apresentada em detalhes no Código 3.2 e explicada na Seção 3.2.2, retorna um vetor contendo as coordenadas dos pontos que formam o fecho convexo, conforme ilustrado no Código 3.4.

```
vector<ii> fecho_convexo = CH_Andrew(pontos);
```

Código 3.4: Chamada da função CH_Andrew em C++

Após obter o fecho convexo, itera-se sobre os pontos resultantes. Para cada ponto no fecho, seu identificador original é recuperado usando o mapa de identificadores (`mapa_identificadores`) e adicionado a um conjunto (`resposta`). A utilização de um conjunto (a estrutura de dados `set` da biblioteca padrão do C++) garante que os identificadores sejam armazenados de forma única e em ordem crescente, atendendo aos requisitos de saída do problema. Esse processo está detalhado no Código 3.5.

```
set<int> resposta;
for (auto ponto : fecho_convexo)
{
    resposta.insert(mapa_identificadores[ponto]);
}
```

Código 3.5: Inclusão dos identificadores no conjunto resposta em C++

Finalmente, os identificadores dos reinos com territórios infinitos são impressos na tela, separados por espaços e em ordem crescente, como especificado na Seção 3.3.3. Essa impressão pode ser visualizada no Código 3.6.

```
bool fir = true;
```

```
2 for (auto valor : resposta)
3 {
4     cout << (fir == true ? "" : " ") << valor;
5     fir = false;
6 }
7 cout << endl;
```

Código 3.6: Impressão dos resultados em C++

Essa abordagem garante que todos os pontos que definem reinos com territórios infinitos sejam identificados de forma eficiente e correta.

3.4 Análise e discussão da solução

3.4.1 Análise de complexidade da solução

A complexidade computacional da solução para identificar os reinos com territórios infinitos é determinada fundamentalmente por três fases principais do algoritmo. A primeira fase consiste na leitura dos dados de entrada e no mapeamento dos identificadores originais das capitais. Ler as N coordenadas e armazená-las em um vetor consome tempo $O(N)$. Associar cada coordenada ao seu identificador original através de um mapa requer N operações de inserção. Dado que cada inserção em um mapa possui complexidade $O(\log N)$, esta etapa inicial tem uma complexidade de tempo de $O(N \log N)$ e uma complexidade de espaço de $O(N)$ para o armazenamento dos pontos e do mapa.

A segunda fase é o cálculo do fecho convexo, realizado pelo algoritmo de cadeias monotônicas. Este algoritmo inicia com a ordenação do vetor de N pontos, uma operação que tem uma complexidade de tempo de $O(N \log N)$. Subsequentemente, a construção dos cascos superior e inferior do fecho convexo envolve uma iteração linear sobre os pontos ordenados. Como cada ponto é empilhado e pode ser desempilhado no máximo uma vez, esta parte do processo tem uma complexidade de tempo linear, ou seja, $O(N)$. O vetor H, que armazena os pontos do fecho, pode ter um tamanho máximo de $2N$, implicando uma complexidade de espaço de $O(N)$. Assim, a etapa de ordenação domina a complexidade do algoritmo de cadeias monotônicas, resultando em $O(N \log N)$.

Por fim, a terceira fase envolve o processamento e a apresentação da saída. Após a obtenção dos h pontos que compõem o fecho convexo (onde $h \leq N$), seus

identificadores originais são recuperados do mapa. Cada uma dessas h buscas tem complexidade $O(\log N)$. Os identificadores recuperados são então inseridos em um conjunto (a estrutura de dados `set` da biblioteca padrão do C++), o que leva $O(h \log h)$ tempo para h inserções, já que cada inserção em um conjunto é $O(\log h)$. A impressão final dos h identificadores ordenados consome tempo $O(h)$. O espaço ocupado pelo conjunto é $O(h)$.

Considerando a soma das complexidades de cada fase, a complexidade de tempo geral da solução é ditada pelas operações de $O(N \log N)$ (leitura e mapeamento, ordenação no cálculo do fecho convexo e o processamento da saída, que no pior caso trata $h = N$ pontos), resultando em uma complexidade de tempo total de $O(N \log N)$. A complexidade de espaço geral é determinada pelo armazenamento das principais estruturas de dados, todas proporcionais a N, levando a uma complexidade de espaço de $O(N)$.

Estas complexidades indicam que a solução proposta é eficiente para o problema. Para o limite de $N \leq 10^5$, isso resulta em um total aproximado de $10^5 \times \log(10^5) \approx 10^5 \times 16.61 \approx 1.66 \times 10^6$ operações, um número viável para o limite de tempo do problema de 0,2 segundos.

3.4.2 Discussão da solução

O algoritmo de cadeias monotônicas é uma técnica eficiente e relativamente simples para calcular o fecho convexo de um conjunto de pontos em um plano bidimensional, operando com uma complexidade de tempo de $O(N \log N)$ devido à ordenação inicial dos pontos. Embora robusto para coordenadas inteiras (com o uso de tipos de dados adequados para evitar *overflow* nos produtos vetoriais), é suscetível a erros de precisão com números de ponto flutuante, o que pode exigir o uso de tolerâncias ou aritmética exata. As suas principais limitações incluem ser um algoritmo essencialmente bidimensional e não ser sensível à saída, o que significa que o seu desempenho não melhora significativamente quando o número de pontos no fecho (h) é muito pequeno.

Comparado a outros algoritmos, o de cadeias monotônicas oferece um bom equilíbrio entre desempenho e facilidade de implementação. É geralmente mais simples que o algoritmo de varredura de Graham (devido à ordenação) e o *Quickhull* (que é recursivo e tem pior caso $O(N^2)$), e mais consistente que o *Jarvis March* (que é $O(N \times h)$ e lento para h grande). Embora o algoritmo de Chan seja assintoticamente mais rápido para h pequeno $O(N \log h)$, é consideravelmente mais complexo.

4. PROBLEMA 3: “MEETING POINT”

4.1 Conceitos

Nesta seção, são apresentados os conceitos fundamentais da teoria dos grafos e algoritmos de caminho mínimo, essenciais para a compreensão da solução do problema proposto.

4.1.1 Grafos

Um grafo $G = (V, E)$ é uma estrutura matemática utilizada para representar relações entre objetos. Ele consiste em um conjunto V de vértices (ou nós) e um conjunto E de arestas (ou arcos), onde cada aresta conecta um par de vértices. As arestas podem ser direcionadas, indicando um fluxo ou relação com sentido único, ou não direcionadas, representando uma relação bidirecional (CORMEN et al., 2022).

Em muitos problemas, as arestas possuem um peso associado, que pode representar distância, custo, tempo ou capacidade. Um grafo com tais arestas é chamado de grafo ponderado. Um caminho em um grafo é uma sequência de vértices conectados por arestas. O comprimento ou custo de um caminho em um grafo ponderado é a soma dos pesos das arestas que o compõem (LAAKSONEN, 2020).

4.1.2 Problema do Caminho Mínimo

O problema do caminho mínimo é um problema clássico na teoria dos grafos e otimização. Dado um grafo ponderado, o objetivo é encontrar um caminho entre dois vértices (origem e destino) tal que a soma dos pesos das arestas nesse caminho seja minimizada (CORMEN et al., 2022). Este problema possui diversas variações, como o caminho mínimo de fonte única (encontrar os caminhos mínimos de um vértice de origem para todos os outros vértices) ou o caminho mínimo entre todos os pares de vértices. A solução para este tipo de problema é fundamental em aplicações como roteamento em redes, logística e planejamento de trajetórias.

4.1.3 Fila de Prioridade

Uma fila de prioridade é uma estrutura de dados abstrata semelhante a uma fila ou pilha, mas onde cada elemento possui uma “prioridade” associada. Elementos com maior prioridade são processados antes de elementos com menor prioridade. No contexto do algoritmo de Dijkstra, que é explicado na seção abaixo, uma fila de prioridade (geralmente implementada como um *heap*) é utilizada para armazenar os vértices que ainda não foram visitados, priorizando aqueles com a menor distância estimada da origem. Isso permite que o algoritmo selecione eficientemente o próximo vértice a ser processado, otimizando significativamente seu desempenho, especialmente em grafos esparsos (SEDGEWICK, 2001).

4.2 Algoritmo

4.2.1 Motivação para algoritmo de Djikstra

O algoritmo de Dijkstra, concebido por Edsger W. Dijkstra, é um algoritmo clássico e eficiente para resolver o problema do caminho mínimo de fonte única em grafos ponderados onde os pesos das arestas são não negativos. Ele opera de forma gulosa, construindo progressivamente um conjunto de vértices para os quais a distância mínima a partir da origem já foi determinada (CORMEN et al., 2022).

O problema abordado neste capítulo requer o cálculo de distâncias mínimas entre diferentes cruzamentos na cidade, que é modelada como um grafo onde os cruzamentos são vértices e as estradas são arestas com pesos correspondentes aos seus comprimentos. Como os comprimentos das estradas são sempre não negativos, o algoritmo de Dijkstra é uma escolha natural e eficiente para determinar essas distâncias mínimas. Sua capacidade de encontrar os caminhos mais curtos de um ponto de origem para todos os outros pontos no grafo é diretamente aplicável às necessidades do problema.

4.2.2 Funcionamento do algoritmo de Dijkstra

O algoritmo de Dijkstra funciona mantendo um conjunto de estimativas de distâncias mínimas do vértice de origem *s* para cada outro vértice no grafo. Inicialmente, a distância para *s* é definida como 0 e para todos os outros vértices como infinita. O algoritmo então procede iterativamente como descrito abaixo e ilustrado na Figura 4.1. (LAAKSONEN, 2020):

1. Marca o vértice de origem *s* como o vértice atual e o adiciona a um conjunto de vértices visitados (ou processados).
2. Para o vértice atual, examina todos os seus vizinhos não visitados. Para cada vizinho *v*, calcula a distância de *s* até *v* passando pelo vértice atual. Se essa nova distância for menor que a estimativa de distância atual para *v*, atualiza a estimativa de *v*. Essa etapa é conhecida como relaxação da aresta.
3. Seleciona o vértice não visitado com a menor estimativa de distância, marca-o como o novo vértice atual e o adiciona ao conjunto de visitados.
4. Repete os passos 2 e 3 até que todos os vértices alcançáveis tenham sido visitados, ou até que o vértice de destino (se houver um específico) seja visitado.

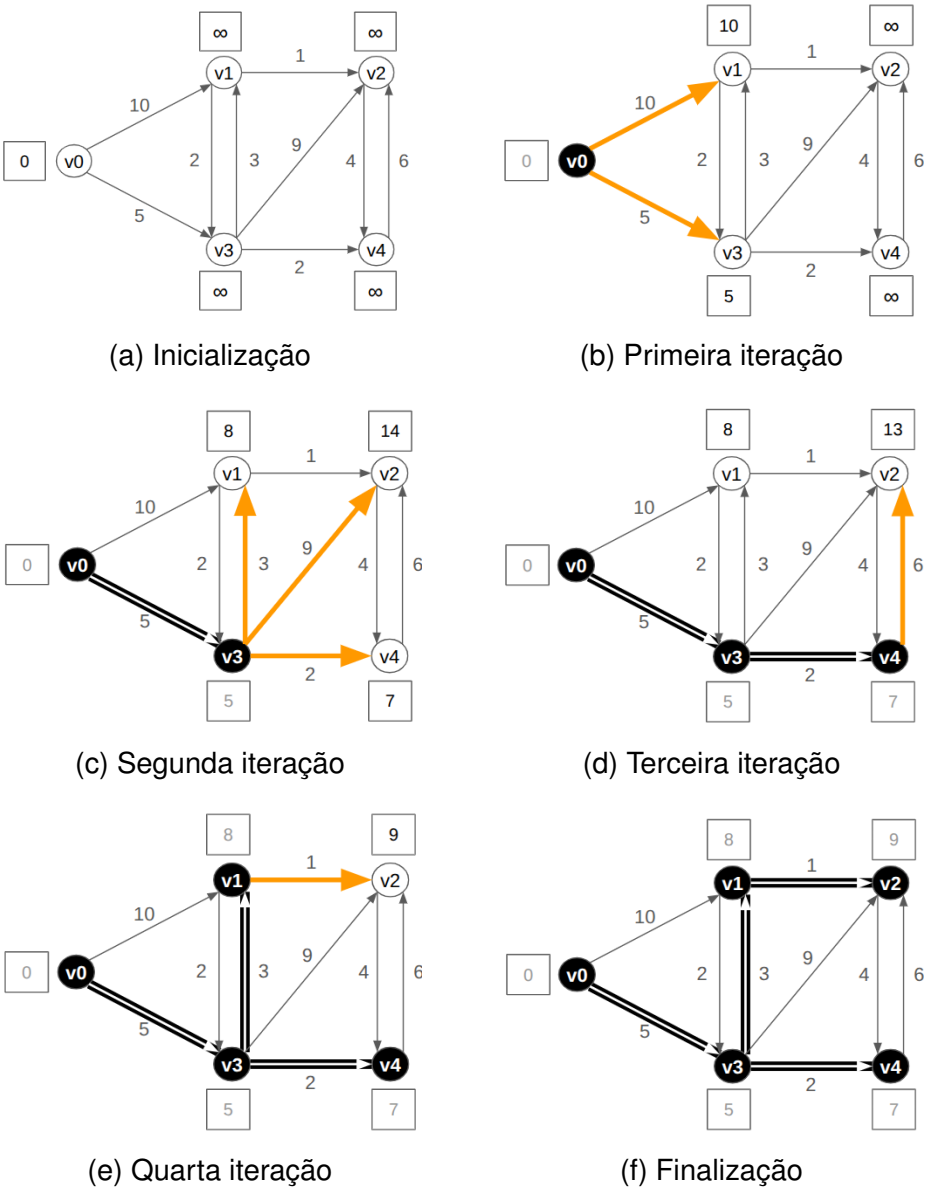


Figura 4.1: Exemplo de execução do algoritmo de Dijkstra passo a passo

Uma fila de prioridade é comumente usada para gerenciar eficientemente os vértices não visitados e suas estimativas de distância, permitindo a rápida seleção do vértice com a menor distância no passo 3. A garantia de que os pesos das arestas são não negativos é crucial para a corretude do algoritmo, pois assegura que, uma vez que um vértice é marcado como visitado, a distância calculada para ele é de fato a mínima (CORMEN et al., 2022).

4.2.3 Implementação do algoritmo de Dijkstra

A implementação do algoritmo de Dijkstra geralmente envolve listas de adjacência para representar o grafo, um vetor para armazenar as estimativas de distâncias da origem a cada vértice, um vetor booleano para marcar os vértices processados e uma fila de prioridade.

No Código 4.1, é apresentada uma função `dijkstra` em C++. Ela recebe o número de vértices `n`, o vértice de origem `s`, as listas de adjacência `adj`, o vetor de estimativas de distâncias `distancia` e o vetor de processados `processado`.

```
1  void dijkstra(int n, int s, vector<ii> adj[], vector<ll> &distancia
       , vector<bool> &processado)
2  {
3      for (int i = 1; i <= n; i++){
4          distancia[i] = INF;
5      }
6      priority_queue<ii> q;
7      distancia[s] = 0;
8      q.push({0, s});
9      while (!q.empty())
10     {
11         ll a = q.top().second;
12         q.pop();
13         if (processado[a]) {
14             continue;
15         }
16
17         processado[a] = true;
18         for (auto u : adj[a])
19         {
20             ll b = u.first, w = u.second;
21             if (distancia[a] + w < distancia[b])
22             {
23                 distancia[b] = distancia[a] + w;
24                 q.push({-distancia[b], b});
25             }
26         }
27     }
```

```
28 }
```

Código 4.1: Algoritmo de Dijkstra em C++

Na linha 4, as distâncias para todos os nós são inicializadas como `INF` (valor consideravelmente grande, que depende dos limites do problema). A distância para o nó de origem `s` é definida como 0 (linha 7). A fila de prioridade `q` armazena pares (o valor negativo da distância, vértice) para simular um *heap* mínimo, pois a fila de prioridade padrão do C++ é um *heap* máximo. O laço principal (linha 9) continua enquanto a fila não estiver vazia. Em cada iteração, o vértice `a` com a menor estimativa de distância é extraído (linha 11). Se `a` já foi processado, ele é ignorado (linha 14). Caso contrário, `a` é marcado como processado (linha 17). Em seguida, para cada vizinho `b` de `a` (linha 20), se um caminho mais curto via `a` é encontrado (linha 21), a estimativa da distância de `b` é atualizada (linha 23) e ele é adicionado à fila de prioridade (linha 24).

4.3 Resolução

4.3.1 Descrição do problema

O Problema “Meeting Point” (Ponto de Encontro) descreve uma cidade onde um amigo, Pedro, fica cansado exatamente no ponto médio de qualquer rota que ele toma para um ponto de encontro. A cidade possui N cruzamentos e M estradas bidirecionais com comprimentos definidos, e sempre existe um caminho entre quaisquer dois cruzamentos. A distância entre dois cruzamentos é o comprimento do caminho mínimo entre eles.

Pedro mora no cruzamento P e o grupo de amigos decidiu se encontrar no cruzamento G. Para garantir que Pedro chegue ao destino G sem se cansar antes, o plano é informá-lo de um ponto de encontro enganoso X, tal que, ao se dirigir de P para X (sempre pelo caminho mais curto), Pedro se canse exatamente em G. Para que o plano funcione, duas condições devem ser satisfeitas:

1. O cruzamento G deve pertencer a todo caminho mínimo possível que Pedro poderia tomar de P para X.
2. Para cada um desses caminhos mínimos de P para X, Pedro deve se cansar exatamente em G. Isso significa que G deve ser o ponto médio exato da rota P para X.

O objetivo é identificar todos os cruzamentos X que poderiam servir como esse ponto de encontro enganoso.

4.3.2 Descrição da entrada do problema

A primeira linha da entrada contém dois inteiros N ($2 \leq N \leq 10^5$) e M ($1 \leq M \leq 10^5$), representando, respectivamente, o número de cruzamentos e o número de estradas. Os cruzamentos são identificados por inteiros distintos de 1 a N. A segunda linha contém dois inteiros P e G ($1 \leq P, G \leq N$ e $P \neq G$), denotando o cruzamento onde Pedro mora e o ponto de encontro correto, respectivamente. Cada uma das M linhas seguintes descreve uma estrada com três inteiros U, V e D ($1 \leq U, V \leq N, U \neq V$ e $1 \leq D \leq 10^9$), indicando uma estrada bidirecional de comprimento D entre os cruzamentos U e V. É garantido que existe um caminho entre cada par de cruzamentos e que há no máximo uma estrada entre cada par de cruzamentos.

4.3.3 Descrição da saída do problema

A saída deve ser uma única linha contendo uma lista crescente de inteiros, representando os identificadores dos cruzamentos que funcionariam como o ponto de encontro enganoso. Se nenhum cruzamento funcionar, deve-se imprimir o caractere * (asterisco).

4.3.4 Solução para o problema

Para que um cruzamento X seja um ponto de encontro enganoso válido, as seguintes condições, que podem ser deduzidas a partir do enunciado, devem ser satisfeitas:

1. Pedro sempre toma um caminho de comprimento mínimo. Seja $d(A, B)$ a distância mínima entre os cruzamentos A e B.

2. Pedro se cansa no ponto médio da rota $P \to X$. Se ele se cansa em G, então G é o ponto médio. Isso implica que a distância de P a G deve ser igual à distância de G a X ao longo do caminho mínimo $P \to G \to X$. Portanto, $d(P, G) = d(G, X)$, e a distância total $d(P, X) = d(P, G) + d(G, X) = 2 \times d(P, G)$.

3. O cruzamento G deve pertencer a todo caminho mínimo possível de P para X.

A estratégia para encontrar os cruzamentos X válidos envolve duas execuções do algoritmo de Dijkstra:

- Primeira execução do algoritmo de Dijkstra: Calcula-se as distâncias mínimas de P para todos os outros cruzamentos no grafo original. Seja $dist_1[v]$ a distância mínima de P a v. A distância $dist_1[G]$ é a distância que Pedro percorre até se cansar (se G for o ponto de cansaço). Um cruzamento X satisfaz a condição do ponto médio se $dist_1[X] = 2 \times dist_1[G]$.
- Segunda execução do algoritmo de Dijkstra: Para verificar se G está em todo caminho mínimo de P para X, calcula-se novamente as distâncias mínimas de P para todos os outros cruzamentos, mas desta vez considerando que o cruzamento G não pode ser utilizado (exceto como origem ou destino final, o que não é o caso aqui, pois G é um ponto intermediário). Isso pode ser implementado "pulando" G durante a execução de Dijkstra (ou seja, tratando G como se já tivesse sido processado desde o início ou não o adicionando à fila de prioridade se ele for um vizinho). Seja $dist_2[v]$ a distância mínima de P a v sem passar por G.

Se $dist_2[X] > dist_1[X]$, significa que qualquer caminho mínimo de P para X que não utiliza G é mais longo do que o caminho mínimo que utiliza G. Isso garante que G deve estar em todo caminho mínimo de P para X. Esse resultado combinado com a condição $dist_1[X] = 2 \times dist_1[G]$ implica que G é um ponto médio em todo caminho mínimo de P para X. Por outro lado, se $dist_2[X] = dist_1[X]$, então existe pelo menos um caminho mínimo de P para X que não passa por G, invalidando X.

Portanto, um cruzamento X é uma solução válida se e somente se:

- $dist_1[G]$ é finito (o que é garantido pelo problema).
- $dist_1[X] = 2 \times dist_1[G]$.
- $dist_2[X] > dist_1[X]$.

Os cruzamentos X que satisfazem essas condições são coletados e impressos em ordem crescente.

4.3.5 Implementação da solução

A implementação da solução segue a estratégia descrita, utilizando a função `dijkstra` apresentada anteriormente na Seção 4.2.3.

Primeiramente, lê-se a entrada: N, M, P, G e as M estradas, construindo as listas de adjacência `adj` (Código 4.2).

```
int n, m, p, g, u, v, d;
cin >> n >> m >> p >> g;

vector<ii> adj[n + 1];
vector<ll> dist1(n + 1);
vector<bool> proc1(n + 1, false);
vector<ll> dist2(n + 1);
vector<bool> proc2(n + 1, false);

for (int i = 0; i < m; i++)
{
    cin >> u >> v >> d;
    adj[u].pb({v, d});
    adj[v].pb({u, d});
}
```

Código 4.2: Leitura da entrada e construção do grafo em C++

Dois vetores de distância, `dist1` e `dist2`, e dois vetores de processados, `proc1` e `proc2`, são inicializados.

Em seguida, realiza-se a primeira execução do algoritmo de Dijkstra a partir de P (Código 4.3).

```
dijkstra(n, p, adj, dist1, proc1);
```

Código 4.3: Primeira execução do algoritmo de Dijkstra C++

Isso preenche `dist1` com as distâncias mínimas de P a todos os outros nós.

Para a segunda execução do algoritmo de Dijkstra, o nó G deve ser ignorado. Isso é feito marcando `proc2[g]` como `true` antes de chamar a função (Código 4.4).

```
proc2[g] = true;
dijkstra(n, p, adj, dist2, proc2);
```

Código 4.4: Segunda execução do algoritmo de Dijkstra C++

Isso preenche `dist2` com as distâncias mínimas de *P* a todos os outros nós, sem passar por *G*.

Finalmente, itera-se por todos os cruzamentos *i* de 1 a *N* para verificar as condições citadas no fim da Seção 4.3.4. Os cruzamentos válidos são armazenados em um `set` para garantir a ordem crescente e unicidade, e depois impressos (Código 4.5).

```
set<int> resposta;
for (int i = 1; i <= n; i++)
{
    if (dist1[i] == 2 * dist1[g])
    {
        if (dist2[i] > dist1[i])
        {
            resposta.insert(i);
        }
    }
}

if (resposta.empty()) {
    cout << "*" << endl;
} else {
    bool fir = true;
    for (auto valor : resposta)
    {
        cout << (fir ? "" : " ") << valor;
        fir = false;
    }
    cout << endl;
}
```

Código 4.5: Verificação das condições e impressão da saída em C++

4.4 Análise e discussão da solução

4.4.1 Análise de complexidade da solução

A análise da complexidade computacional da solução para o problema inicia-se com a fase de leitura da entrada e construção do grafo. A leitura dos parâmetros N, M, P e G é realizada em tempo $O(1)$. Subsequentemente, as M estradas são processadas; para cada uma, a leitura e a adição às listas de adjacência (utilizando `vector` em C++, com inserções amortizadas em $O(1)$) consomem tempo $O(M)$. A inicialização de vetores auxiliares para N elementos demanda $O(N)$. Consequentemente, esta etapa inicial possui uma complexidade de $O(N + M)$.

A segunda fase envolve as duas execuções do algoritmo de Dijkstra. A solução aplica estas duas execuções a partir do vértice P. Cada execução, utilizando listas de adjacência e uma fila de prioridade baseada em *heap* binário, opera com uma complexidade de tempo de $O((N + M) \log N)$. Portanto, o tempo combinado para esta fase, que é a mais custosa computacionalmente, é de $2 \times O((N+M) \log N) = O((M+N) \log N)$. Para grafos conexos, onde $M \geq N - 1$, é comum simplificar esta expressão para $O(M \log N)$ se M for o termo dominante.

A fase final envolve a identificação dos pontos enganosos e a preparação da saída. O algoritmo itera sobre os N cruzamentos para verificar as condições. As comparações de distância são feitas em $O(1)$, e a inserção dos candidatos válidos em um conjunto (a estrutura de dados `set` da biblioteca padrão do C++) leva $O(\log N)$ por inserção. No pior cenário, esta etapa pode levar $O(N \log N)$. A impressão dos N resultados é feita em $O(N)$.

Considerando todas as fases, a complexidade de tempo total da solução é ditada predominantemente pelas execuções do algoritmo de Dijkstra. Assim, a complexidade de tempo assintótica é $O((M + N) \log N)$. Dados os limites do problema ($N, M \leq 10^5$), onde $\log 10^5 \approx 17$, o número de operações é da ordem de $(2 \times 10^5) \times 17 \approx 3{,}4 \times 10^6$, sendo viável para os limites de tempo do problema de 0,3 segundos em C++.

Em termos de memória, a complexidade de espaço é determinada pelo armazenamento da listas de adjacência para o grafo ($O(N + M)$), pelos vetores de distância e de nós processados ($O(N)$), pela fila de prioridade ($O(N)$ em seu pico de armazenamento) e pelo conjunto de respostas ($O(N)$). Consequentemente, a complexidade de espaço total é $O(N + M)$.

4.4.2 Discussão da solução

Embora o algoritmo de Dijkstra seja uma escolha eficiente e correta para o problema devido aos pesos não negativos das arestas, é importante reconhecer suas limitações e os cenários onde algoritmos alternativos seriam mais apropriados. A principal restrição de Dijkstra é sua incapacidade de lidar corretamente com arestas de peso negativo (CORMEN et al., 2022). Se o problema permitisse tais arestas, a estratégia gulosa de Dijkstra (finalizar a distância de um nó assim que ele é extraído da fila de prioridade) poderia falhar, pois um caminho descoberto posteriormente através de uma aresta negativa poderia oferecer uma rota mais curta para um nó já finalizado. Consequentemente, Dijkstra também não detecta ciclos de peso negativo, que tornariam o conceito de caminho mínimo indefinido. Adicionalmente, para grafos dinâmicos, onde a estrutura ou os pesos das arestas mudam frequentemente, reexecutar Dijkstra repetidamente pode ser computacionalmente custoso, e algoritmos especializados para caminhos mínimos dinâmicos seriam mais indicados.

Em cenários com arestas de peso negativo, o algoritmo de Bellman-Ford seria uma alternativa robusta (LAAKSONEN, 2020). Com uma complexidade de tempo de $O(|V||E|)$, onde V é o conjunto de vértices e E é o conjunto de arestas do grafo de entrada, ele é mais lento que Dijkstra em grafos com pesos não negativos, mas sua capacidade de processar pesos negativos e de detectar ciclos negativos o torna indispensável em contextos mais gerais. Se o problema envolvesse, por exemplo, “custos” que pudessem ser negativos, Bellman-Ford seria o algoritmo de escolha para o problema de encontrar caminhos mínimos de fonte única.

Para problemas que exigem o cálculo dos caminhos mínimos entre todos os pares de vértices, o algoritmo de Floyd-Warshall é uma solução clássica (LAAKSONEN, 2020). Ele também lida com pesos negativos (desde que não haja ciclos negativos) e possui uma complexidade de $O(|V|^3)$. Embora seja muito lento para o problema desse capítulo, que necessita apenas de caminhos a partir de uma fonte, Floyd-Warshall seria útil se o escopo do problema fosse expandido para analisar interações de caminhos entre múltiplos pontos de partida e destino simultaneamente, ou em grafos menores e densos onde a informação completa de todos os pares é o objetivo principal.

5. PROBLEMA 4: “KARAMELL”

5.1 Conceitos

Para a compreensão da solução proposta, esta seção detalha os conceitos essenciais. Aborda-se o problema da soma de subconjuntos como a base da estratégia, a programação dinâmica como a técnica para resolvê-lo eficientemente e, por fim, o método de *backtracking* para reconstruir a solução encontrada.

5.1.1 Problema da Soma de Subconjuntos

O problema da soma de subconjuntos é um problema clássico na ciência da computação. Dada uma coleção de números inteiros, o objetivo é determinar se existe um subconjunto desses números cuja soma seja igual a um valor alvo específico. Uma variação comum, relevante para este problema, é o problema da partição, que busca dividir um conjunto de números em dois subconjuntos com somas iguais. Embora seja um problema NP-completo em sua forma geral, pode ser resolvido eficientemente para certos casos com restrições nos valores dos números ou no tamanho do conjunto, geralmente utilizando programação dinâmica (CORMEN et al., 2022).

5.1.2 Programação Dinâmica e Memoização

Programação dinâmica é uma técnica de otimização utilizada para resolver problemas complexos dividindo-os em subproblemas menores e sobrepostos. A solução de cada subproblema é calculada uma única vez e armazenada (memoização) para evitar recálculos redundantes. Isso é particularmente útil em algoritmos recursivos onde a mesma chamada de função com os mesmos parâmetros pode ocorrer múltiplas vezes. A memoização transforma uma complexidade de tempo exponencial em uma complexidade polinomial (ou pseudo-polinomial, dependendo do problema), tornando a solução viável para entradas maiores (CORMEN et al., 2022).

5.1.3 *Backtracking* (Rastreamento Retroativo)

Após determinar que uma solução para a soma de subconjuntos existe, muitas vezes é necessário identificar quais elementos do conjunto original compõem essa soma. O *backtracking* é um processo algorítmico que pode ser usado para encontrar essa solução específica, “refazendo” as decisões tomadas durante o cálculo da programação dinâmica (SKIENA, 2020). Para a tabela de memoização do algoritmo da soma de subconjuntos, isso envolve começar da célula que indica a solução final e traçar um caminho de volta, identificando em cada etapa se um elemento foi incluído ou não para alcançar a soma parcial correspondente.

5.2 Algoritmo

5.2.1 Motivação para o algoritmo da soma de subconjuntos

A motivação central é usar o algoritmo da soma de subconjuntos para resolver o problema da partição: a tarefa de dividir um conjunto de números em dois subconjuntos de soma igual. A estratégia consiste em reduzir este problema a uma instância mais simples. Primeiro, calcula-se a soma total S dos elementos. Se for possível particionar o conjunto, cada subconjunto deverá somar exatamente $\frac{S}{2}$. Portanto, o problema se resume a encontrar um único subconjunto que atinja essa soma alvo. Se tal subconjunto for encontrado, o conjunto complementar terá, garantidamente, a mesma soma.

É aqui que o algoritmo da soma de subconjuntos se torna a ferramenta ideal. Implementado com programação dinâmica, ele pode verificar de forma eficiente a existência desse subconjunto alvo. Em seguida, uma vez confirmada a sua existência, aplicamos a técnica de *backtracking* sobre a estrutura de dados gerada para reconstruir os elementos que compõem a solução, resolvendo assim o problema da partição.

5.2.2 Funcionamento do algoritmo da soma de subconjuntos

O algoritmo da soma de subconjuntos pode ser implementado de várias maneiras. As duas principais abordagens existentes são: programação dinâmica recursiva ou iterativa. A abordagem detalhada a seguir será a recursiva.

Para compreender a implementação recursiva, é útil definir seus componentes principais. O algoritmo é implementado na função *soma_subconjunto*, responsável por determinar se uma soma alvo (representada pela variável *soma_alvo*) é possível, ou seja, o valor numérico que se deseja alcançar. Esta função opera sobre um vetor de valores (chamado de *valores*), que é o conjunto de números da entrada disponível. Durante sua execução, a função utiliza um índice para navegar pelo vetor e uma variável para rastrear a soma atual (*soma_atual*) do subconjunto que está sendo testado. Para otimizar o processo, uma tabela de memoização (*memo*) é usada para armazenar os resultados de subproblemas já resolvidos, evitando assim recálculos redundantes.

Soma de subconjuntos recursivo com memoização

A abordagem recursiva explora duas possibilidades para cada elemento do conjunto: ou o elemento é incluído no subconjunto para atingir a soma alvo, ou não é, como ilustrado na Figura 5.1. A função *soma_subconjunto*(*indice*, *soma_atual*) tenta alcançar a *soma_alvo*.

- **Casos Base:**
  - Se a soma atual for igual à soma alvo, uma solução foi encontrada, e neste caso retorna-se verdadeiro.
  - Se a soma atual for maior que a soma alvo ou o índice atual ultrapassar o tamanho do conjunto, este caminho não leva a uma solução, e neste caso retorna-se falso.
- **Passo Recursivo:** Para o elemento atual no vetor de valores (*valores*[*indice*]):
  - Tenta-se incluir *valores[indice]*: chamando *soma_subconjunto(indice + 1, soma_atual + valores[indice])*.
  - Tenta-se excluir *valores[indice]*: chamando *soma_subconjunto(indice + 1, soma_atual)*.

Se qualquer uma das chamadas retornar verdadeiro, então uma solução é possível.

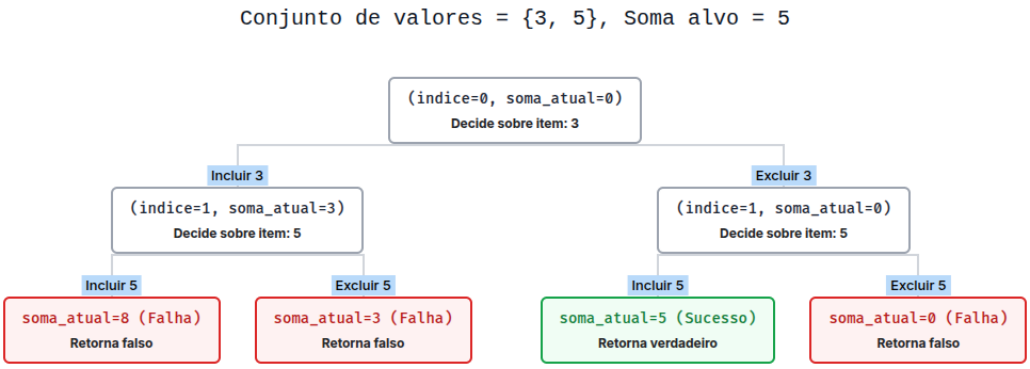


Figura 5.1: Árvore de decisão para o problema da soma de subconjuntos

Para evitar recálculos de subproblemas idênticos (mesmo *indice* e *soma_atual*), uma tabela de memoização (*memo*[*indice*][*soma_atual*]) é usada para armazenar os resultados. Antes de fazer o cálculo para um subproblema, verifica-se se o resultado já está armazenado na tabela. Caso esteja, não é necessário fazer o cálculo novamente. Caso contrário, o cálculo é feito e, em seguida, o resultado é armazenado na tabela.

Recuperação de uma solução em dois vetores

Uma vez que a chamada ao algoritmo de soma de subconjuntos confirma a existência de um subconjunto de soma igual a $\frac{S}{2}$ (sendo S a soma total do conjunto de valores), pode-se reconstruí-lo e, ao mesmo tempo, separar todos os valores originais em dois vetores distintos. A ideia geral é percorrer os valores em ordem reversa, usando a tabela de memoização para decidir a que vetor cada valor pertence:

Primeiro, define-se uma variável auxiliar *restante* que inicia em $\frac{S}{2}$. Em seguida, para cada valor v_i, começando do último índice e indo até o primeiro, é feito:

- Teste de inclusão: Verifica se é possível formar a soma *restante* incluindo v_i. Em termos da tabela de memoização, isto equivale a checar se

$$\textit{restante} \geq v_i \quad \text{e} \quad \textit{memo}[\, i-1 \,][\, \textit{restante} - v_i \,] = 1.$$

  Se ambas as condições forem verdadeiras, significa que v_i faz parte de um subconjunto cuja soma é $\frac{S}{2}$. Nesse caso, v_i é adicionado ao vetor A e *restante* é decrementado de v_i, para atualizar a soma restante a ser completada.

- Teste de exclusão: Caso contrário, conclui-se que v_i não foi usado naquele subconjunto e, então, é colocado no vetor *B*. O valor de *restante* permanece inalterado.

Repete-se esse procedimento até ter avaliado todos os elementos ou até que *restante* seja zerado. Ao final, o vetor *A* conterá exatamente os elementos cuja soma é $\frac{S}{2}$, e o vetor *B* reunirá todos os demais, também somando $\frac{S}{2}$. Dessa forma, é obtida a lógica de *backtracking* que transforma a informação armazenada na memoização em duas listas de valores que resolvem o problema de partição.

5.2.3 Implementação do algoritmo da soma de subconjuntos e reconstrução

A seguir apresenta-se a implementação em C++ do algoritmo de soma de subconjuntos (Código 5.1). A função `soma_subconjunto` (linha 1) verifica se é possível alcançar exatamente a soma alvo acumulando valores de `valores[indice]` em diante:

- Linha 3: Se soma atual é igual a soma alvo, um subconjunto válido foi encontrado, e a função retorna verdadeiro.
- Linha 7: Se a soma atual excede a soma alvo ou o índice atual está fora dos limites do vetor, a ramificação da busca é interrompida, retornando falso.
- Linha 11: Caso essa combinação de entrada já tenha sido processada, o valor previamente armazenado na memoização é retornado.
- Linha 15: É realizada uma chamada recursiva para avaliar a possibilidade de incluir o elemento `valores[indice]`.
- Linha 16: É realizada uma chamada recursiva para avaliar a possibilidade de excluir o elemento `valores[indice]`.
- Linha 18: O resultado da operação lógica OU (união) das duas chamadas recursivas anteriores é armazenado em `memo[indice][soma_atual]`.

```
1 bool soma_subconjunto(int indice, int soma_atual)
2 {
3     if (soma_atual == soma_alvo) {
4         return true;
5     }
6
```

```
7       if (soma_atual > soma_alvo || indice >= n) {
8           return false;
9       }
10
11      if (memo[indice][soma_atual] != -1) {
12          return memo[indice][soma_atual];
13      }
14
15      bool incluir = soma_subconjunto(indice + 1, soma_atual +
            valores[indice]);
16      bool excluir = soma_subconjunto(indice + 1, soma_atual);
17
18      return memo[indice][soma_atual] = ((incluir || excluir) ? 1 :
            0);
19  }
```

Código 5.1: Algoritmo de soma de subconjuntos em C++

O código responsável pela reconstrução em dois vetores está presente no Código 5.2. Após a função responsável pelo algoritmo da soma de subconjuntos retornar verdadeiro, utiliza-se a função `obter_subconjunto` (linha 1) para distribuir cada elemento em dois vetores genéricos `A` e `B`.

A variável `restante` é inicializada com a soma alvo (linha 3) e, em seguida, o *indice* é percorrido de $n - 1$ até 0 (linha 5):

- Linha 7: Se `restante` for maior ou igual ao valor atual presente em `valores[indice]` e `memo[indice][restante - valores[indice]] == 1` for verdadeiro, significa que o valor atual foi utilizado na construção da solução, então `valores[indice]` pertence ao vetor `A`. Atualiza-se `restante = restante - valores[indice]`.
- Linha 10: Caso contrário, `valores[indice]` pertence ao vetor `B`.

No fim, `A` contém um subconjunto com a soma alvo e `B` agrupa os demais elementos, somando o mesmo valor.

```
1   void obter_subconjunto()
2   {
3       int restante = soma_alvo;
4
5       for (int indice = n-1; indice >= 0; indice--)
6       {
7           if (restante >= valores[indice] && memo[indice][restante -
                valores[indice]] == 1) {
```

```
8               A.push_back(valores[indice]);
9               restante -= valores[indice];
10          } else {
11              B.push_back(valores[indice]);
12          }
13      }
14  }
```

Código 5.2: Reconstrução da solução em dois vetores em C++

5.3 Resolução

5.3.1 Descrição do problema

O Problema “Karamell” descreve a situação de Alice e Bob, irmãos gêmeos que recebem N sacolas de caramelos como presente de aniversário, onde a i-ésima sacola contém a_i caramelos. Eles decidem distribuir os caramelos de uma maneira específica: as sacolas são consideradas em uma ordem pré-determinada, e no i-ésimo passo, os a_i caramelos da i-ésima sacola são dados à pessoa que tiver menos caramelos naquele momento. Em caso de empate na quantidade de caramelos, Alice recebe a sacola.

A insatisfação dos irmãos surge do fato de que a quantidade final de caramelos que cada um recebe pode variar dependendo da ordem em que as sacolas são consideradas. O objetivo do problema é desenvolver um programa que determine uma maneira de ordenar as sacolas de forma que Alice e Bob recebam a mesma quantidade final de caramelos, se tal ordenação for possível.

5.3.2 Descrição da entrada do problema

A entrada do problema é composta por duas linhas. A primeira linha contém um único inteiro N ($1 \leq N \leq 100$), que representa o número de sacolas de caramelos. A segunda linha contém N inteiros a_1, ..., a_N ($1 \leq a_i \leq 100$), onde a_i indica a quantidade de caramelos na i-ésima sacola.

5.3.3 Descrição da saída do problema

A saída deve conter uma única linha. Se for impossível encontrar uma ordem das sacolas que resulte em uma divisão igual de caramelos entre Alice e Bob, o programa deve imprimir o valor -1. Caso contrário, a saída deve ser composta por N inteiros separados por espaço, representando uma ordenação válida dos valores a_i que garantam que os caramelos serão divididos igualmente entre os irmãos, seguindo a regra de distribuição descrita no problema.

5.3.4 Solução para o problema

A primeira etapa consiste em computar a soma total de todos os caramelos nas N sacolas, denotada por S. Se este valor for ímpar, não existe forma de dividi-los igualmente entre Alice e Bob e a resposta imediata é -1.

Quando S é par, define-se o valor alvo para cada irmão como

$$S_{alvo} = \frac{S}{2}.$$

Em seguida, aplica-se o algoritmo de soma de subconjuntos para verificar se há algum subconjunto de sacolas cuja soma seja exatamente S_{alvo}. Adota-se a implementação recursiva com memoização descrita anteriormente na Seção 5.2.2. Caso esse teste retorne falso, conclui-se novamente que a divisão igualitária não é possível e imprime-se -1.

Se, por outro lado, a função da soma de subconjuntos indicar que a soma S_{alvo} é atingível, a função de *backtracking* é usada para reconstruir a solução separando os N itens em dois vetores *alice* e *bob*. O vetor *alice* recebe exatamente aqueles elementos que compõem o subconjunto de soma S_{alvo}, e *bob* agrupa todas as sacolas restantes.

Para construir a ordenação final de saída, dois acumuladores (*soma_alice* e *soma_bob*) são mantidos, inicialmente zerados. O seguinte processo é repetido N vezes: se *soma_alice* $\leq$ *soma_bob*, uma sacola de *alice* é removida, adicionada ao fim da sequência de saída, e *soma_alice* é incrementado pelo valor dessa sacola; caso contrário, o mesmo procedimento é feito com *bob* e *soma_bob* é atualizado. Dado que $\sum_i alice[i] = \sum_i bob[i] = S_{alvo}$, ao final das N remoções, ambos acumuladores serão

iguais, e a sequência construída satisfará o critério de entrega alternada “quem tiver menos recebe” (com empate favorecendo Alice).

Por fim, essa sequência de saída contendo N valores é impressa. Se em qualquer fase anterior, a divisão igualitária mostrou-se impossível, a única saída impressa é -1.

5.3.5 Implementação da solução

A solução para o problema é implementada em C++ e se organiza em torno da função `main` (Código 5.3). Inicialmente, o programa lê a entrada, o número de sacolas N e as respectivas quantidades de caramelos a_i (armazenadas no vetor `valores`) e calcula a soma total dos caramelos (linhas 3-10). Uma verificação imediata é feita sobre a paridade desta soma total (linhas 12-15): se for ímpar, uma divisão igualitária é impossível, e o programa encerra imprimindo -1. Caso contrário, a `soma_alvo` para cada irmão é definida como metade da soma total (linha 17).

A tabela de memoização `memo` é então preparada (linha 18), sendo inicializada com -1 para indicar que nenhum subproblema da recursão foi calculado. O núcleo da lógica de decisão está na chamada à função `soma_subconjunto` (linha 20), detalhada na Seção 5.2.2. Esta função verifica recursivamente se é possível formar um subconjunto de sacolas que some exatamente a `soma_alvo` definida. Se `soma_subconjunto` retornar falso, indicando que tal subconjunto não existe, o programa também encerra com a saída -1 (linhas 20-23).

Com a confirmação da possibilidade de uma partição igualitária, a função `obter_subconjunto` (explicada na Seção 5.2.2) é chamada (linha 25). Esta função utiliza a tabela `memo`, agora preenchida pelos cálculos de `soma_subconjunto`, para realizar o *backtracking* e efetivamente separar as sacolas originais nos vetores `alice` e `bob`, representando os conjuntos de sacolas destinados a cada um.

A etapa seguinte é a construção da ordem final de entrega das sacolas (linhas 29-40), que deve respeitar a regra de distribuição do problema. O programa simula esta distribuição: em cada um dos N passos da iteração, verifica-se qual dos irmãos possui menos caramelos acumulados no momento (`soma_alice` vs. `soma_bob`). A sacola da vez é então retirada do final do vetor correspondente (`alice` ou `bob`, com Alice tendo prioridade em caso de empate) e adicionada ao vetor `resultado`. As somas parciais de caramelos de Alice e Bob são atualizadas a cada sacola distribuída.

Finalmente, o vetor `resultado`, que agora contém a sequência de sacolas que garante a divisão igualitária dos caramelos conforme as regras, é impresso na saída padrão (linhas 42-46).

```
1  int main(int argc, char const *argv)
2  {
3      cin >> n;
4      soma_alvo = 0;
5
6      for (int i = 0; i < n; i++)
7      {
8          cin >> valores[i];
9          soma_alvo += valores[i];
10     }
11
12     if ((soma_alvo % 2) == 1) {
13         cout << "-1\n";
14         return 0;
15     }
16
17     soma_alvo = soma_alvo / 2;
18     memset(memo, -1, sizeof(memo));
19
20     if (!soma_subconjunto(0, 0)) {
21         cout << "-1\n";
22         return 0;
23     }
24
25     obter_subconjunto();
26     int soma_alice = 0, soma_bob = 0;
27     vector<int> resultado;
28
29     for (int i = 0; i < n; i++)
30     {
31         if (soma_alice <= soma_bob) {
32             resultado.push_back(alice.back());
33             soma_alice += alice.back();
34             alice.pop_back();
35         } else {
36             resultado.push_back(bob.back());
37             soma_bob += bob.back();
```

```
38              bob.pop_back();
39          }
40      }
41
42      for (int i = 0; i < n; i++)
43      {
44          cout << (i == 0 ? "" : " ") << resultado[i];
45      }
46      cout << "\n";
47      return 0;
48  }
```

Código 5.3: Função principal para o Problema Karamell em C++

5.4 Análise e discussão

5.4.1 Análise de complexidade da solução

A eficiência da solução para o problema é determinada principalmente pela função `soma_subconjunto`. As operações iniciais, como leitura da entrada (N sacolas) e cálculo da soma total, são realizadas em tempo $O(N)$. De forma similar, a função `obter_subconjunto`, que reconstrói os subconjuntos, percorre os N elementos uma vez, resultando em complexidade $O(N)$. A construção final da ordem de saída também leva tempo $O(N)$, pois envolve N iterações com operações de tempo constante.

O componente dominante é a função `soma_subconjunto`. Implementada com recursão e memoização, ela explora um espaço de subproblemas definido pelo índice do item atual variando de 0 a $N-1$ e pela soma parcial acumulada variando de 0 até $\frac{S}{2}$ (onde S seria a soma total dos valores da entrada). O número total de subproblemas únicos é, portanto, $N \times (\frac{S}{2})$. Por meio da memoização, cada subproblema é calculado apenas uma vez. Assim, a complexidade de tempo da função `soma_subconjunto`, e por consequência da solução completa, é $O(N \times \frac{S}{2})$.

Considerando os limites do problema, onde $N \leq 100$ e cada quantidade de caramelos $a_i \leq 100$, a soma total S não excede $100 \times 100 = 10^4$. Portanto, $\frac{S}{2}$ é no máximo 5000. Isso implica uma complexidade de tempo de aproximadamente $100 \times 5000 = 5 \times 10^5$ operações, um volume computacional que se encaixa nos limites de tempo de 0,1 segundos do problema.

Quanto à complexidade de espaço, o maior requisito de memória vem da tabela de memoização, que precisa armazenar os resultados para cada subproblema, demandando $O(N \times \frac{S}{2})$ de espaço. Os vetores para armazenar os valores das sacolas, os subconjuntos de Alice e Bob, e o resultado final consomem espaço adicional proporcional a N, que é menor em comparação com a tabela. Para os limites do problema, é um uso de memória aceitável.

5.4.2 Discussão da solução

A abordagem utilizada, que reduz o problema da partição a uma instância do problema da soma de subconjuntos, é uma solução clássica. No entanto, é fundamental compreender que ambos são problemas NP-completos (CORMEN et al., 2022). Em termos simples, isso significa que não existe um algoritmo conhecido que possa encontrar a solução de forma rápida (em tempo polinomial) para todas as entradas possíveis. A solução com programação dinâmica, apesar de eficiente, não quebra essa barreira teórica. Sua complexidade de $O(N \times S)$ é considerada pseudo-polinomial, pois sua eficiência depende não apenas do número de elementos N, mas também da magnitude da soma S.

A viabilidade da solução está justamente nas restrições do problema. Como o número de sacolas N e a quantidade de caramelos em cada sacola são pequenos, a soma total S também é limitada. É essa característica que torna a abordagem de programação dinâmica eficaz aqui. Se as quantidades de caramelos pudessem ser números muito grandes (por exemplo, na casa dos bilhões), a tabela de memoização se tornaria impraticavelmente grande e o tempo de execução excessivo, mesmo com poucos itens.

É interessante notar que a estrutura do algoritmo da soma de subconjuntos é muito similar à de outro problema famoso: o Problema da Mochila (*Knapsack Problem*). No problema da mochila, o objetivo é maximizar o valor total de itens colocados em uma mochila com capacidade de peso limitada. O problema da soma de subconjuntos pode ser visto como uma variação do problema da mochila, onde o "peso" de cada item é igual ao seu "valor", e o objetivo é verificar se é possível encher a mochila perfeitamente com uma capacidade igual à soma alvo. Ambos são classicamente resolvidos com programação dinâmica, demonstrando um padrão recorrente na resolução de problemas combinatórios (CORMEN et al., 2022).

Por fim, caso a solução exata com programação dinâmica fosse inviável devido a valores muito grandes na entrada, seria necessário recorrer a outras estraté-

gias. Uma abordagem comum para o problema da partição é usar algoritmos de aproximação ou heurísticas. Por exemplo, um algoritmo guloso simples poderia ordenar os itens em ordem decrescente e, a cada passo, adicionar o próximo item ao subconjunto que tiver a menor soma no momento. Embora essa abordagem seja muito mais rápida (geralmente $O(N \log N)$) e muitas vezes encontre boas soluções, ela não oferece garantia de encontrar a partição perfeita, mesmo que uma exista. Essa é a troca clássica em problemas difíceis: sacrificar a garantia da solução ótima em troca de velocidade.

6. CONCLUSÃO

Este trabalho teve como objetivo analisar e detalhar soluções para um conjunto de problemas selecionados da Maratona de Programação da Sociedade Brasileira de Computação (SBC). O foco foi apresentar não apenas o código final, mas também explorar os conceitos teóricos, as escolhas algorítmicas e as análises de complexidade que fundamentam cada solução, de modo a criar um recurso de apoio para estudantes da área.

A metodologia empregada seguiu um processo estruturado. A seleção de problemas baseou-se em estatísticas oficiais de competições e na relevância dos algoritmos necessários. A implementação em C++ foi escolhida por seu desempenho e pelas funcionalidades de sua biblioteca padrão. Adicionalmente, cada solução foi validada em um juiz *online* para assegurar sua correção e desempenho, e foi complementada por uma revisão bibliográfica para alinhar as implementações com a teoria da área.

O primeiro problema, “Na Trave!”, lidava com a necessidade de responder a um grande volume de consultas em intervalos. Uma solução com abordagem ingênua seria inviável devido à sua complexidade quadrática. A solução eficiente utilizou uma técnica de processamento, ordenando as operações de entrada e aplicando uma árvore de Fenwick para obter as respostas, resolvendo o problema dentro dos limites de tempo.

A solução para o Problema “Grande Tratado da Bytelândia” envolveu uma abordagem de geometria computacional para determinar quais reinos possuíam territórios ilimitados. A solução consistiu em reduzir o problema ao cálculo do fecho convexo do conjunto de capitais. O algoritmo de cadeias monotônicas foi implementado para construir eficientemente esse fecho e, assim, identificar os reinos com territórios ilimitados.

No terceiro problema, “*Meeting Point*”, o desafio consistia em encontrar caminhos mínimos em um grafo que satisfizessem uma condição de ponto médio. A solução foi obtida por meio de uma estratégia que envolveu duas execuções do algoritmo de Dijkstra. A primeira calculava as distâncias padrão a partir da origem, enquanto a segunda calculava as distâncias em um grafo modificado, onde o ponto de encontro intermediário era evitado. A comparação dos resultados permitiu filtrar e identificar os cruzamentos que cumpriam todos os critérios.

Finalmente, o quarto problema, “*Karamell*”, foi modelado como uma instância do problema da partição. A solução demonstrou a aplicação da programação dinâ-

mica para resolver este problema NP-completo de forma eficaz, dadas as restrições da entrada. Utilizando uma abordagem recursiva com memoização, determinou-se a viabilidade de dividir caramelos em duas metades iguais. Em seguida, uma etapa de *backtracking* reconstruiu os conjuntos de sacolas para cada irmão, e uma simulação final gerou a ordem de distribuição que satisfazia as regras do problema.

Essa imersão em desafios de alta complexidade fomenta um desenvolvimento no estudante de ciência da computação. Além do domínio técnico de algoritmos e estruturas de dados, a prática constante cultiva a resiliência, a criatividade e o raciocínio lógico. Tais habilidades são universais e preparam o futuro profissional não apenas para ser aprovado em uma entrevista técnica, mas para contribuir de forma significativa na resolução de problemas reais e inovadores que encontrará ao longo de sua carreira.

REFERÊNCIAS BIBLIOGRÁFICAS

BERG, M. de et al. *Computational Geometry: Algorithms and Applications*. Heidelberg: Springer, 2013.

CORMEN, T. et al. *Introduction to Algorithms, fourth edition*. Cambridge: MIT Press, 2022.

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

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

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*. 2025. Acesso em: 26 de junho de 2025. Disponível em: <https://codeforces.com/>.

PREPARATA, F.; SHAMOS, M. *Computational Geometry: An Introduction*. New York: Springer, 1993. (Monographs in Computer Science).

SEDGEWICK, R. *Algorithms in C++, part 5: graph algorithms*. 3rd. ed. USA: Addison-Wesley, 2001.

SKIENA, S. S. *The Algorithm Design Manual*. 3rd. ed. Cham: Springer, 2020.

APÊNDICE A – CÓDIGO COMPLETO DO PROBLEMA 1

```cpp
#include<bits/stdc++.h>

using namespace std;

#define DBG(x) cout << "[" << #x << "]: " << x << endl
#define F(x) std::fixed <<std::setprecision(1)<<(x)
#define f first
#define s second
#define pb push_back
#define mp make_pair

typedef long long ll;
typedef vector<int> vi;
typedef pair<int, int> ii;
typedef pair<int, ii> iii;

const int INSERCAO = 1;
const int CONSULTA = 2;
const int MAX = 312345;

struct consulta {
    int ultimo_ano;
    int colocacao;
    int anos_acompanhados;
    int indice;
    int resposta;
};

struct insercao {
    int quantidade_vagas;
    int ano;
};

struct operacao {
    int valor_ordenacao;
    int tipo_operacao;
    struct consulta consulta;
```

```
38      struct insercao insercao;
39  };
40
41  int n, y, bit[MAX];
42
43  void update(int i, int delta)
44  {
45      while(i <= y) {
46          bit[i] += delta;
47          i += (i & -i);
48      }
49  }
50
51  int query(int i)
52  {
53      int sum = 0;
54      while(i > 0) {
55          sum += bit[i];
56          i -= (i & -i);
57      }
58      return sum;
59  }
60
61  bool ordenacao_operacao(operacao a, operacao b)
62  {
63      if (a.valor_ordenacao == b.valor_ordenacao) {
64          return a.tipo_operacao < b.tipo_operacao;
65      }
66      return a.valor_ordenacao > b.valor_ordenacao;
67  }
68
69  bool ordenacao_resposta(consulta a, consulta b)
70  {
71      return a.indice < b.indice;
72  }
73
74  int main(int argc, char const *argv[])
75  {
76      ios_base::sync_with_stdio(0);
77      cin.tie(0);
```

```
78      memset(bit, 0, sizeof(bit));
79
80      int quantidade_vagas, ultimo_ano, colocacao, anos_acompanhados;
81      vector<consulta> resultados;
82      vector<operacao> operacoes;
83      vector<int> vagas;
84
85      cin >> y >> n;
86
87      for (int ano = 1; ano <= y; ano++)
88      {
89          cin >> quantidade_vagas;
90          vagas.push_back(quantidade_vagas);
91
92          operacao op;
93          insercao ins;
94
95          ins.quantidade_vagas = quantidade_vagas;
96          ins.ano = ano;
97
98          op.valor_ordenacao = quantidade_vagas;
99          op.tipo_operacao = INSERCAO;
100         op.insercao = ins;
101
102         operacoes.push_back(op);
103     }
104
105     for (int indice = 0; indice < n; indice++)
106     {
107         cin >> ultimo_ano >> colocacao >> anos_acompanhados;
108
109         operacao op;
110         consulta con;
111
112         con.ultimo_ano = ultimo_ano;
113         con.colocacao = colocacao;
114         con.anos_acompanhados = anos_acompanhados;
115         con.indice = indice;
116
117         op.valor_ordenacao = colocacao;
```

```
118         op.tipo_operacao = CONSULTA;
119         op.consulta = con;
120 
121         operacoes.push_back(op);
122     }
123 
124     sort(operacoes.begin(), operacoes.end(), ordenacao_operacao);
125 
126     for (auto operacao : operacoes)
127     {
128         if (operacao.tipo_operacao == INSERCAO)
129         {
130             update(operacao.insercao.ano, 1);
131         }
132         else if (operacao.tipo_operacao == CONSULTA)
133         {
134             consulta con = operacao.consulta;
135 
136             if (con.colocacao <= vagas[con.ultimo_ano-1])
137             {
138                 con.resposta = 0;
139                 resultados.push_back(con);
140             }
141             else
142             {
143                 int resposta = query(con.ultimo_ano + con.
                        anos_acompanhados) - query(con.ultimo_ano);
144                 con.resposta = resposta;
145                 resultados.push_back(con);
146             }
147         }
148     }
149 
150     sort(resultados.begin(), resultados.end(), ordenacao_resposta);
151     for (auto consulta : resultados) {
152         cout << consulta.resposta << endl;
153     }
154 
155     return 0;
```

```
156 }
```

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 DBG(x) cout << "[" << #x << "]: " << x << endl
#define F(x) std::fixed <<std::setprecision(1)<<(x)
#define f first
#define s second
#define pb push_back
#define mp make_pair

typedef long long ll;
typedef vector<int> vi;
typedef pair<ll, ll> ii;
typedef pair<int, ii> iii;

bool ccw(ii a, ii b, ii c) {
    return (b.f - a.f) * (c.s - a.s) - (b.s - a.s) * (c.f - a.f) >=
        0;
}

vector<ii> CH_Andrew(vector<ii> &v)
{
    int n = v.size(), k = 0;
    vector<ii> H(2*n);
    sort(v.begin(), v.end());

    for (int i = 0; i < n; ++i)
    {
        while ((k >= 2) && !ccw(H[k-2], H[k-1], v[i])) --k;
        H[k++] = v[i];
    }

    for (int i = n-2, t = k+1; i >= 0; --i) {
        while ((k >= t) && !ccw(H[k-2], H[k-1], v[i])) --k;
        H[k++] = v[i];
    }
```

```
37
38      H.resize(k);
39      return H;
40  }
41
42  int main(int argc, char const *argv[])
43  {
44      ios_base::sync_with_stdio(0);
45      cin.tie(0);
46
47      int n;
48      cin >> n;
49      vector<ii> pontos;
50      map<ii, int> mapa_identificadores;
51
52      for (int i = 0; i < n; i++)
53      {
54          int x, y;
55          cin >> x >> y;
56          pontos.pb(ii(x, y));
57          mapa_identificadores[ii(x, y)] = i + 1;
58      }
59
60      vector<ii> fecho_convexo = CH_Andrew(pontos);
61      set<int> resposta;
62      for (auto ponto : fecho_convexo)
63      {
64          resposta.insert(mapa_identificadores[ponto]);
65      }
66
67      bool fir = true;
68      for (auto valor : resposta)
69      {
70          cout << (fir == true ? "" : " ") << valor;
71          fir = false;
72      }
73      cout << endl;
74
75      return 0;
```

```
76 }
```

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 DBG(x) cout << "[" << #x << "]: " << x << endl
#define F(x) std::fixed << std::setprecision(1) << (x)
#define f first
#define s second
#define pb push_back
#define mp make_pair

typedef long long ll;
typedef vector<int> vi;
typedef pair<ll, ll> ii;
typedef pair<int, ii> iii;
const ll INF = 1e9 + 7;

void dijkstra(int n, int s, vector<ii> adj[], vector<ll> &distancia
   , vector<bool> &processado)
{
    for (int i = 1; i <= n; i++){
        distancia[i] = INF;
    }
    priority_queue<ii> q;
    distancia[s] = 0;
    q.push({0, s});
    while (!q.empty())
    {
        ll a = q.top().second;
        q.pop();
        if (processado[a]) {
            continue;
        }

        processado[a] = true;
        for (auto u : adj[a])
        {
```

```
37              ll b = u.first, w = u.second;
38              if (distancia[a] + w < distancia[b])
39              {
40                  distancia[b] = distancia[a] + w;
41                  q.push({-distancia[b], b});
42              }
43          }
44      }
45  }
46
47  int main(int argc, char const *argv[])
48  {
49      ios_base::sync_with_stdio(0);
50      cin.tie(0);
51
52      int n, m, p, g, u, v, d;
53      cin >> n >> m >> p >> g;
54
55      vector<ii> adj[n + 1];
56      vector<ll> dist1(n + 1);
57      vector<bool> proc1(n + 1, false);
58      vector<ll> dist2(n + 1);
59      vector<bool> proc2(n + 1, false);
60
61      for (int i = 0; i < m; i++)
62      {
63          cin >> u >> v >> d;
64          adj[u].pb({v, d});
65          adj[v].pb({u, d});
66      }
67
68      dijkstra(n, p, adj, dist1, proc1);
69      proc2[g] = true;
70      dijkstra(n, p, adj, dist2, proc2);
71
72      set<int> resposta;
73      for (int i = 1; i <= n; i++)
74      {
75          if (dist1[i] == 2 * dist1[g])
76          {
```

```
77              if (dist2[i] > dist1[i])
78              {
79                  resposta.insert(i);
80              }
81          }
82      }
83
84      if (resposta.empty()) {
85          cout << "*" << endl;
86      } else {
87          bool fir = true;
88          for (auto valor : resposta)
89          {
90              cout << (fir ? "" : " ") << valor;
91              fir = false;
92          }
93          cout << endl;
94      }
95
96      return 0;
97  }
```

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 DBG(x) cout << "[" << #x << "]: " << x << endl
#define F(x) std::fixed << std::setprecision(1) << (x)
#define f first
#define s second
#define pb push_back
#define mp make_pair

typedef long long ll;
typedef vector<int> vi;
typedef pair<int, int> ii;
typedef pair<int, ii> iii;

const int MAXN = 110;
const int MAXSOMA = 10010;
int memo[MAXN][MAXSOMA];
int valores[MAXN];
int n, soma_alvo;
vector<int> alice, bob;


bool soma_subconjunto(int indice, int soma_atual)
{
    if (soma_atual == soma_alvo) {
        return true;
    }

    if (soma_atual > soma_alvo || indice >= n) {
        return false;
    }

    if (memo[indice][soma_atual] != -1) {
        return memo[indice][soma_atual];
    }
```

```
38
39      bool incluir = soma_subconjunto(indice + 1, soma_atual +
           valores[indice]);
40      bool excluir = soma_subconjunto(indice + 1, soma_atual);
41
42      return memo[indice][soma_atual] = ((incluir || excluir) ? 1 :
           0);
43  }
44
45  void obter_subconjunto()
46  {
47      int restante = soma_alvo;
48
49      for (int indice = n-1; indice >= 0; indice--)
50      {
51          if (restante >= valores[indice] && memo[indice][restante -
               valores[indice]] == 1) {
52              alice.push_back(valores[indice]);
53              restante -= valores[indice];
54          } else {
55              bob.push_back(valores[indice]);
56          }
57      }
58  }
59
60  int main(int argc, char const *argv[])
61  {
62      ios_base::sync_with_stdio(0);
63      cin.tie(0);
64
65      cin >> n;
66      soma_alvo = 0;
67
68      for (int i = 0; i < n; i++)
69      {
70          cin >> valores[i];
71          soma_alvo += valores[i];
72      }
73
74      if ((soma_alvo % 2) == 1) {
```

```
75          cout << "-1\n";
76          return 0;
77      }
78
79      soma_alvo = soma_alvo / 2;
80
81      memset(memo, -1, sizeof(memo));
82
83      if (!soma_subconjunto(0, 0)) {
84          cout << "-1\n";
85          return 0;
86      }
87
88      obter_subconjunto();
89      int soma_alice = 0, soma_bob = 0;
90      vector<int> resultado;
91
92      for (int i = 0; i < n; i++)
93      {
94          if (soma_alice <= soma_bob) {
95              resultado.push_back(alice.back());
96              soma_alice += alice.back();
97              alice.pop_back();
98          } else {
99              resultado.push_back(bob.back());
100             soma_bob += bob.back();
101             bob.pop_back();
102         }
103     }
104
105     for (int i = 0; i < n; i++)
106     {
107         cout << (i == 0 ? "" : " ") << resultado[i];
108     }
109     cout << "\n";
110
111     return 0;
112 }
```

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

ANEXO A – Certificado de Participação na Fase Regional na Maratona de Programação da SBC 2021

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

ANEXO C – Certificado de Participação na Fase Nacional na Maratona de Programação da SBC 2023

Certificate of Achievement

The 2023 ICPC South America/Brazil Finals - Maratona de Programação

Chapecó, 19 - 22 October 2023

Honorable Mention

Universidade Federal da Fronteira Sul

Vinicius Koncicoski

Heitor Machado

Pedro Henrique Piaia Dariva

Andrei Braga, Coach

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

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

Certificate of Achievement

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

31 August 2024

Universidade Federal da Fronteira Sul

Vinicius Koncicoski

Pedro Henrique Piaia Dariva

Pedro Spegiorin

Andrei Braga, Coach

Samuel Feitosa, Co-coach

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