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

JOÃO HENRIQUE ALVES DOS SANTOS

ANÁLISE E RESOLUÇÃO DE PROBLEMAS DA MARATONA DE PROGRAMAÇÃO DAS EDIÇÕES DE 2023, 2024 E 2025

CHAPECÓ
2026

JOÃO HENRIQUE ALVES DOS SANTOS

ANÁLISE E RESOLUÇÃO DE PROBLEMAS DA MARATONA DE PROGRAMAÇÃO DAS EDIÇÕES DE 2023, 2024 E 2025

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

Orientador: Prof. Dr. Andrei de Almeida Sampaio Braga

CHAPECÓ
2026

Santos, João Henrique Alves dos

ANÁLISE E RESOLUÇÃO DE PROBLEMAS DA MARATONA DE PROGRAMAÇÃO DAS EDIÇÕES DE 2023, 2024 E 2025 / João Henrique Alves dos Santos – 2026.

77 f.

Orientador: Prof. Dr. Andrei de Almeida Sampaio Braga

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

1. Programação Competitiva 2. Algoritmo Guloso 3. Busca Completa 4. Grafos 5. Busca Binária I. Braga, Dr. Andrei de Almeida Sampaio, orient. II. Universidade Federal da Fronteira Sul. III. Título.

JOÃO HENRIQUE ALVES DOS SANTOS

ANÁLISE E RESOLUÇÃO DE PROBLEMAS DA MARATONA DE PROGRAMAÇÃO DAS EDIÇÕES DE 2023, 2024 E 2025

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

Orientador: Prof. Dr. Andrei de Almeida Sampaio Braga

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

BANCA AVALIADORA

gov.br Documento assinado digitalmente
ANDREI DE ALMEIDA SAMPAIO BRAGA
Data: 06/07/2026 14:22:50-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Andrei de Almeida Sampaio Braga - UFFS

gov.br Documento assinado digitalmente
GEOMAR ANDRE SCHREINER
Data: 07/07/2026 11:31:51-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Geomar André Schreiner - UFFS

gov.br Documento assinado digitalmente
SAMUEL DA SILVA FEITOSA
Data: 06/07/2026 19:11:05-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Samuel da Silva Feitosa - UFFS

AGRADECIMENTOS

Em primeiro lugar, agradeço eternamente aos meus pais pelo apoio e amor incondicional, sem os quais nada seria possível para mim.

Agradeço imensamente ao meu orientador pelo incansável esforço em não somente guiar a construção deste trabalho, mas também por tornar possível a minha jornada pela programação competitiva.

Também agradeço muito aos meus amigos e companheiros de equipe das competições, que dividiram comigo momentos felizes e experiências marcantes durante a graduação.

RESUMO

O desenvolvimento do pensamento computacional e a capacidade de projetar soluções corretas e eficientes são competências fundamentais para estudantes e profissionais da área da computação. Nesse contexto, as competições de programação, como a Maratona de Programação da SBC, desempenham um papel crucial no aprimoramento dessas habilidades. Este trabalho tem como objetivo contribuir para o treinamento nessas competições por meio da análise detalhada de quatro problemas de dificuldade média e elevada, oriundos de edições recentes da Maratona de Programação da SBC. A metodologia adotada consiste na introdução de conceitos teóricos, na descrição dos problemas e de suas soluções, na implementação das abordagens em C++, na análise de complexidade de tempo e de espaço e na discussão sobre limitações e alternativas possíveis. A corretude das soluções foi validada por meio de submissões na plataforma *Codeforces*. O conjunto de problemas que compõe este trabalho envolve algoritmos gulosos, busca completa, aritmética modular, grafos e busca binária. O resultado é um material de apoio que conecta teoria e prática, auxiliando na preparação de estudantes para competições de programação.

Palavras-Chave: Programação Competitiva, Algoritmo Guloso, Busca Completa, Grafos, Busca Binária.

ABSTRACT

The development of computational thinking and the ability to design correct and efficient solutions are fundamental competencies for students and professionals in the field of computing. In this context, programming contests, such as the SBC Programming Marathon, play a crucial role in strengthening these skills. This work aims to contribute to training for such competitions through a detailed analysis of four medium and high difficulty problems drawn from recent editions of the SBC Programming Marathon. The methodology adopted consists of the introduction of theoretical concepts, the description of the problems and their solutions, the implementation of the approaches in C++, the analysis of time and space complexity, and the discussion of limitations and possible alternatives. The correctness of the solutions was validated through submissions on the *Codeforces* platform. The selected problems involve greedy algorithms, complete search, modular arithmetic, graphs, and binary search. The result is a supporting study resource that connects theory and practice, assisting students in preparing for programming contests.

Keywords: Competitive Programming, Greedy Algorithm, Complete Search, Graphs, Binary Search.

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

O desenvolvimento de habilidades lógicas, do pensamento computacional e da capacidade de abstração é imprescindível para estudantes da área da computação, sejam aqueles que desejam seguir uma carreira acadêmica ou ingressar no âmbito profissional (WING, 2006). Nesse contexto, as competições de programação destacam-se como atividades que promovem grande aprendizado em projeto e análise de algoritmos, além do aprofundamento em estruturas de dados e técnicas de otimização. Esses conhecimentos transcendem o ambiente competitivo e aplicam-se às mais diversas áreas da computação, constituindo também competências valorizadas no mercado (SBC, 2026).

Os desafios propostos por essas competições abrangem uma vasta quantidade de conceitos estudados pelo campo teórico da computação. Suas aplicações na resolução de problemas exige perspicácia e senso analítico, impulsionando os competidores a projetarem soluções que são não apenas corretas, mas também eficientes. Embora uma solução ineficiente para um respectivo problema possa ser concebida até por iniciantes, construir um algoritmo eficiente para o mesmo problema pode representar um desafio significativo mesmo para competidores mais experientes (LAAKSONEN, 2020).

Para o fim de contribuir com o treinamento para essas competições, este trabalho tem como objetivo abordar conceitos teóricos recorrentes nessas competições, descrever problemas e suas soluções, relacionando-os com os conteúdos discutidos. Além disso, foram incluídas implementações seguidas de análises de complexidade de tempo e espaço e, por fim, uma discussão sobre cada problema. Para isso, foram escolhidos quatro problemas de dificuldade média e elevada, oriundos da Maratona de Programação da Sociedade Brasileira de Computação (SBC). Os problemas escolhidos contemplam diversas técnicas e algoritmos primordiais na área da computação.

1.1 Metodologia

Nesta seção, são apresentados os critérios adotados para a escolha das referências bibliográficas e o processo de definição do conjunto de problemas analisados. Em seguida, são apresentadas as estratégias de implementação utilizadas, bem como os procedimentos de teste e avaliação empregados para verificar a corretude das so-

luções. Por fim, detalha-se como é conduzida a análise e a discussão das abordagens propostas ao longo do trabalho.

1.1.1 Revisão bibliográfica

Neste trabalho, foram utilizados como referências livros fundamentais da área de algoritmos, assim como outros mais específicos de programação competitiva (HALIM; HALIM; EFFENDY, 2018; LAAKSONEN, 2020; CORMEN et al., 2022; SKIENA, 2020), como base para a construção das apresentações dos conceitos e das técnicas empregadas na resolução dos problemas abordados. Também foram consultados outros materiais de apoio que visam fortalecer o entendimento de técnicas recorrentes na programação competitiva. Essas referências forneceram o embasamento teórico necessário para apresentar conceitos e algoritmos, associando-os de forma adequada aos problemas discutidos no presente trabalho.

1.1.2 Seleção de problemas

A composição do conjunto de problemas analisados neste trabalho partiu de edições recentes da Maratona de Programação da SBC, abrangendo as Fases Regionais de 2023, 2024 e 2025 e a Fase Nacional de 2024. A seleção considerou o entendimento do autor sobre as técnicas envolvidas assim como a intenção de incluir problemas de dificuldade média e elevada, capazes de sustentar análises relevantes para a área. Para apoiar esse processo de escolha, foram consultadas as estatísticas oficiais disponibilizadas pela SBC, utilizadas como apoio na identificação de problemas com um nível de desafio compatível com os objetivos deste trabalho. Com base nesses critérios, foram definidos os quatro problemas que compõem o escopo deste trabalho, apresentados na tabela abaixo:

1. *Kool Strings* (Fase Nacional 2024)
    - Total de submissões: 159
    - Submissões aceitas: 49 (31%)
2. *Harmônicos Interferentes* (Fase Regional 2024)
    - Total de submissões: 613
    - Submissões aceitas: 83 (14%)

3. *Desvio* (Fase Regional 2023)
    - Total de submissões: 102
    - Submissões aceitas: 6 (6%)
4. *Muralhas Reforçadas* (Fase Regional 2025)
    - Total de submissões: 2858
    - Submissões aceitas: 224 (8%)

1.1.3 Estratégia de implementação

A estratégia de implementação adotada neste trabalho baseou-se na utilização da linguagem C++, escolhida principalmente por seu excelente desempenho e por ser amplamente consolidada no contexto da programação competitiva. Além da eficiência, a linguagem oferece uma biblioteca padrão robusta, que disponibiliza estruturas e funções essenciais para a resolução de problemas, como vetores, tuplas e estruturas de *heap*. Por também ser a linguagem mais utilizada nesse meio, grande parte dos materiais de apoio, como editoriais e tutoriais relevantes da área, se encontram naturalmente exemplificados em C++, o que favorece a comparação com implementações consolidadas (LAAKSONEN, 2020).

No que diz respeito às escolhas específicas de implementação, foram utilizados mecanismos de otimização de entrada e saída, como `ios::sync_with_stdio(0)` e `cin.tie(0)`, que reduzem o custo de entrada/saída ao dessincronizar as bibliotecas do C e C++ e ao evitar *flush* automático entre leituras e escritas. Da mesma forma, a impressão de quebras de linha foi feita com `'\n'` ao invés de `std::endl`, evitando o custo adicional do *flush* do *buffer* a cada uso (LAAKSONEN, 2020). Também foram empregadas macros para definição de constantes e algumas estruturas e variáveis foram declaradas globalmente com o objetivo de simplificar a passagem de parâmetros e reduzir o volume de código. Além disso, essas estruturas foram dimensionadas considerando o valor máximo de entrada previsto nos enunciados, com uma margem adicional. Isso permitiu prevenir problemas relacionados a tamanho insuficiente, tornando as implementações mais seguras e estáveis dentro dos limites estabelecidos pelos problemas estudados.

1.1.4 Ferramenta de avaliação de soluções

A avaliação das soluções propostas foi apoiada por uma ferramenta de verificação prática, utilizando a plataforma *Codeforces* (MIRZAYANOV, 2026) como ambiente de teste. Por meio da submissão das implementações e da análise dos vereditos obtidos, foi possível confrontar as abordagens desenvolvidas com os casos de teste oficiais e com as restrições de tempo e memória dos problemas. Esse procedimento contribuiu para atestar a corretude das soluções apresentadas e reforçar a confiabilidade dos resultados discutidos ao longo do trabalho.

1.1.5 Análise e discussão das soluções

Em cada problema, as soluções são apresentadas acompanhadas de uma análise de complexidade de tempo e de espaço, de modo a demonstrar o custo computacional das abordagens adotadas. A partir dessa análise, evidencia-se que o desempenho de cada solução é compatível com os limites estabelecidos pelos problemas, indicando que as implementações são suficientes para serem aceitas em ambientes de competições. Além disso, há uma discussão complementar para cada solução, com observações sobre eventuais limitações e menções a alternativas possíveis.

1.2 Organização do texto

Este trabalho é organizado de maneira a apresentar quatro problemas da Maratona de Programação da SBC nos Capítulos 2, 3, 4 e 5, cada um destinado a tratar de um problema particular. Estes capítulos contam com as seguintes seções:

- **Conceitos**: introduz os fundamentos teóricos necessários para a solução de cada problema;
- **Algoritmo**: apresenta a motivação do algoritmo adotado, descreve seu funcionamento e expõe uma implementação;
- **Resolução**: descreve a estrutura de entrada e de saída do problema, desenvolve a ideia da solução proposta e explica detalhes relevantes da implementação;
- **Análise e discussão**: analisa as complexidades de tempo e de espaço da solução e discute suas limitações, bem como alternativas possíveis.

2. PROBLEMA 1: KOOL STRINGS

O problema *Kool Strings* descreve que a professora Kardashi é obcecada por *strings* binárias (formadas somente por 0's e 1's) e considera uma *string* como *kool* quando ela contém menos do que um número K de caracteres consecutivos iguais. Para testar as habilidades dos competidores, ela propõe o seguinte desafio: dado um número K, uma *string* S e uma operação de modificação que consiste em escolher um índice i e inverter o caractere S_i de 0 para 1 ou vice-versa (esta alteração pode ser executada qualquer número de vezes), informar qual a menor quantidade de modificações exigidas para que a *string* se torne *kool*, mostrando também qualquer *string* válida como resultado da aplicação desta quantidade de operações sobre S. Uma descrição completa do problema pode ser acessada aqui: https://codeforces.com/gym/105505/problem/K.

A solução para o problema envolve a utilização de um algoritmo guloso, cuja justificativa parte de algumas observações a respeito da estrutura das *strings* binárias consideradas *kool*. Adicionalmente, é necessário tratar de forma especial o caso em que $K = 2$, onde uma nova abordagem torna-se necessária. A solução apresentada neste capítulo foi desenvolvida pelo time do autor durante a competição, e sua explicação baseia-se nos comentários dos autores do problema, presentes no editorial oficial disponibilizado pela Sociedade Brasileira de Computação (SBC).

Nas seções a seguir, serão apresentados os conceitos envolvidos na solução desse problema e as etapas de sua resolução. Essas etapas incluem o algoritmo proposto, o detalhamento da solução e, ao final, a análise e a discussão a seu respeito.

2.1 Conceitos

Nesta seção, serão apresentados os conceitos de ótimo local, ótimo global, propriedade da escolha gulosa, subestrutura ótima e algoritmo guloso, fundamentos essenciais para a construção de uma solução eficiente para o problema abordado neste capítulo.

2.1.1 Ótimo local, ótimo global e propriedade da escolha gulosa

Ótimo local é um ponto em uma função $f(X)$ tal que, considerando a sua vizinhança imediata, é a alternativa que oferece o melhor resultado. Ótimo global é a melhor opção possível entre todas as soluções viáveis. A diferença entre um ótimo local e um ótimo global pode ser observada na Figura 2.1, onde o objetivo é encontrar um valor de X de forma que $f(X)$ tenha o maior valor possível. Nos chamados algoritmos gulosos, descritos logo abaixo, é possível combinar os resultados provenientes de ótimos locais de forma que a solução final coincida com o ótimo global. Quando isso vale, é dito que o problema possui a propriedade da escolha gulosa (CORMEN et al., 2022).

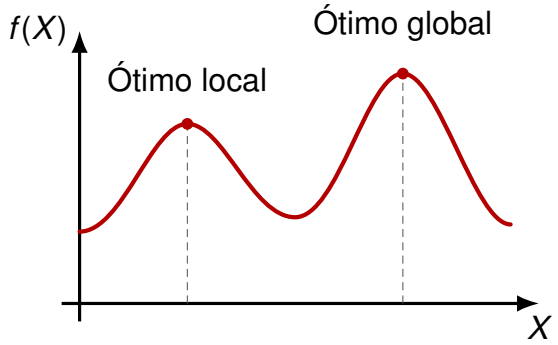


Figura 2.1: Um ótimo local e um ótimo global para uma função $f(X)$

2.1.2 Subestrutura ótima

A subestrutura ótima significa que uma solução ótima do problema contém soluções ótimas dos subproblemas que a compõem. Essa característica permite decompor o problema em partes menores, resolver cada parte de modo independente e, em seguida, recompor o resultado final. Em determinados problemas, é possível provar que a combinação de uma escolha gulosa com as soluções ótimas dos subproblemas resultantes dessa escolha leva a uma solução ótima global (CORMEN et al., 2022).

2.1.3 Algoritmo guloso

Um algoritmo guloso é composto por uma sequência de etapas em que, a cada passo, é feita uma escolha localmente ótima, i.e., a decisão que parece mais vantajosa naquele momento (HALIM; HALIM; EFFENDY, 2018). Essa técnica é tipicamente eficiente, visto que a solução final é construída diretamente com base nas decisões tomadas em cada passo, que nunca são reconsideradas. Quando um problema possui a propriedade da escolha gulosa, combinada com a subestrutura ótima, um algoritmo guloso leva, de forma garantida, a uma solução ótima (LAAKSONEN, 2020).

2.2 Algoritmo

2.2.1 Motivação para o algoritmo guloso

A utilização de um algoritmo guloso na resolução de um problema é, quando aplicável, preferível a outras técnicas. A solução costuma ser eficiente, simples e de fácil codificação, já que não é necessário implementar mecanismos complexos de arrependimento de decisões tomadas durante a execução. Para fortalecer o entendimento do funcionamento de um algoritmo guloso, na Seção 2.2.2 será apresentada uma solução para o problema da mochila fracionária, que servirá de base para o entendimento do conceito envolvido na solução proposta para o problema abordado neste capítulo.

2.2.2 Funcionamento do algoritmo guloso

O problema da mochila fracionária é uma variação do clássico problema da mochila, no qual é permitido fracionar os itens. Dado um conjunto de N itens, cada um com um valor v_i e um peso w_i, e uma mochila com capacidade máxima M, o objetivo é maximizar o valor total dos itens colocados na mochila sem ultrapassar o limite de peso. Ao ser permitido inserir apenas uma fração de um item na mochila, isso admite que o problema seja resolvido de forma exata e eficiente por meio de uma estratégia gulosa.

A solução consiste em calcular, para cada item, a razão valor/custo, representada por $r_i = \frac{v_i}{w_i}$, que indica o valor obtido por unidade de peso. Em seguida, os itens são ordenados em ordem decrescente dessa razão, priorizando aqueles com maior valor por peso. O algoritmo percorre a lista nessa ordem, inserindo cada item completamente enquanto houver capacidade disponível. Se o próximo item não couber totalmente, o item é fracionado para que seja inserido somente a capacidade restante da mochila. Esse processo garante que cada decisão local contribua para maximizar o valor total final da mochila, resultando em uma solução ótima em tempo $O(N \log N)$ devido à ordenação inicial.

2.3 Resolução

Conforme exposto no início do capítulo, o objetivo do problema *Kool Strings* é identificar qual o menor número de inversões de *bits* em uma *string* binária que são necessárias para que ela não contenha K ou mais caracteres consecutivos iguais. Como resposta, o programa deve informar essa quantidade juntamente de uma *string* válida resultante dessas inversões. A solução para o problema será explicada abaixo.

2.3.1 Descrição da entrada do problema

A entrada consiste em uma linha contendo um número K e uma *string* S ($2 \leq K \leq |S| \leq 10^5$), onde K é o número que limita a quantidade de caracteres consecutivos iguais e S é a *string* na qual as operações de troca (possivelmente nenhuma) serão realizadas.

2.3.2 Descrição da saída do problema

A saída deve consistir em uma linha com um número X, que representa a quantidade mínima de modificações necessárias para que a *string* se torne *kool* e uma *string* T válida, obtida a partir de S, aplicando-se X operações de troca.

2.3.3 Solução

A solução para este problema é fundamentada em algumas observações, das quais um algoritmo guloso pode ser derivado:

1. A cada bloco de tamanho K de caracteres consecutivos iguais, é suficiente modificar não mais do que um deles para satisfazer às restrições do problema;

2. Para qualquer índice i tal que $K \leq i \leq |S|$ em que o intervalo $[S_{i-k+1}, S_i]$ é composto de caracteres iguais, se $i = |S|$ ou se $i < |S|$ e $S_i = S_{i+1}$, a melhor escolha de operação de troca a ser realizada no intervalo $[S_{i-k+1}, S_i]$ é em S_i, para assegurar que o restante da *string* a partir do índice $i + 1$ permaneça independente de todos os caracteres anteriores. Caso $i < |S|$ e $S_i \neq S_{i+1}$, a melhor escolha de modificação é em S_{i-1}, para prevenir que o caractere S_i estenda um intervalo de caracteres iguais a partir de $i + 1$, o que poderia acarretar em mais modificações do que o necessário para toda a *string* (Figura 2.2);

3. Para um intervalo A de caracteres iguais consecutivos, onde $1 \leq |A| \leq |S|$, seguindo as escolhas ótimas descritas na observação 2 baseadas na observação 1, o mínimo de caracteres a serem modificados dentro desse intervalo é $\left\lfloor \frac{|A|}{K} \right\rfloor$;

4. A resposta ótima se dá pelo somatório da quantidade de modificações dentro de cada um dos N intervalos de caracteres iguais consecutivos presentes na *string*:

$$\sum_{i=1}^{N} \left\lfloor \frac{|A_i|}{K} \right\rfloor .$$

Adotando como exemplo a entrada $K = 3$ e $S = 11111111100$ e seguindo as observações descritas anteriormente, os índices a serem modificados são indicados na Figura 2.2. Neste caso é possível notar que, não trocar o último caractere do bloco 1, fará com que este índice junte-se aos índices 4 e 5 (caso eles também não sejam modificados) e forme um bloco com 3 caracteres consecutivos iguais. Além disso, é observável que mudar o caractere da posição 9 do bloco 3 fará com que o mesmo forme, juntamente com os índices 10 e 11 do bloco seguinte, uma sequência de K caracteres iguais.

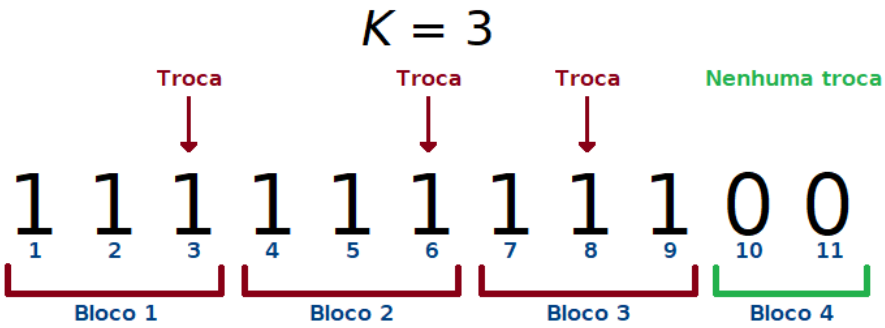


Figura 2.2: Decomposição da *string* *S* em blocos de *K* caracteres consecutivos iguais

Após as 3 trocas corretas serem executadas, é obtida a *string* mostrada na Figura 2.3. Esta nova *string* não possui *K* ou mais caracteres consecutivos iguais tornando-se *kool*, atendendo aos requisitos do problema.

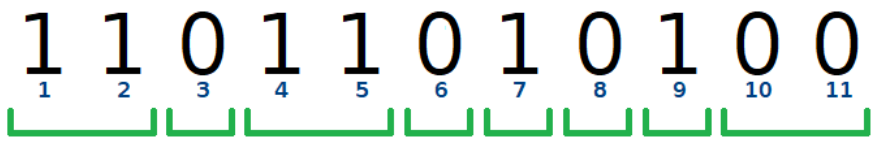


Figura 2.3: A *string* resultante das alterações previstas na Figura 2.2

As observações expostas anteriormente funcionam para qualquer *string* *S* com um $K > 2$, mas não funcionam para o caso $K = 2$. Por exemplo, com a entrada $K = 2$ e $S = 1001$, a operação de modificação determinada pelo algoritmo atual não levará a uma resposta correta ao final do programa.

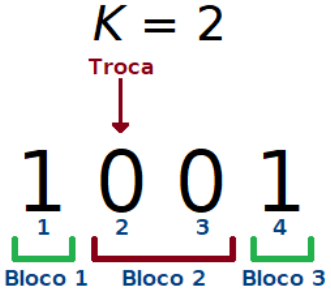


Figura 2.4: Caso problemático de mudança de caractere

Como é possível observar na Figura 2.4, a operação de alteração indicada pela observação 1 resulta em uma extensão de caracteres consecutivos com o bloco 1, necessitando mais modificações que não são previstas usando as observações descritas anteriormente. Neste caso particular, controlar quais caracteres devem ser

modificados pode ficar consideravelmente mais difícil, e um tratamento especial faz-se necessário. Neste cenário onde $K = 2$, é possível realizar uma nova observação que permitirá resolver o problema de forma simplificada, separando este caso dos demais:

5. Se $K = 2$, só existem dois tipos de *strings kool* possíveis: S' = "010101..." ou S'' = "101010...", cada uma com o mesmo tamanho de S.

Seguindo esta observação, é possível gerar estas duas novas *strings* e a resposta final para o problema será qual destas duas alternativas possui o menor número de diferenças para a *string* original. Utilizando o exemplo de entrada anterior, teremos a comparação da *string* original S com as duas opções S' e S'', conforme pode ser visualizado na Figura 2.5. Como S' e S'' empatam na quantidade de caracteres diferentes com relação à S, qualquer uma pode ser escolhida como resposta válida.

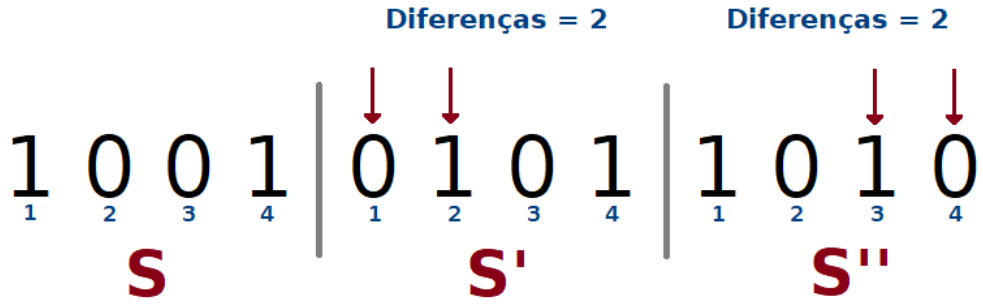


Figura 2.5: As duas respostas possíveis e suas diferenças para S

2.3.4 Implementação da solução

Primeiro, é necessário direcionar a solução conforme o valor de K. Para o caso $K > 2$, é feita uma iteração sobre todos os caracteres da *string* `str` lida da entrada, utilizando as seguintes variáveis auxiliares:

- `answer`, que representa qual a quantidade mínima de operações de modificação que tiveram que ser executadas para que a *string* atenda às restrições impostas pelo problema;
- `current_char`, que será utilizado para identificar qual o último caractere lido da *string* (esta variável é inicializada com o caractere `#`, servindo como um valor padrão);

```
1  n = (int) str.size();
2  if(k > 2) {
3      current_char = '#';
4      for(int i = 0; i < n; i++) {
5          if(str[i] == current_char) {
6              counter++;
7          } else {
8              current_char = str[i];
9              counter = 1;
10         }
11         if(counter == k) {
12             if(i < n - 1 && str[i] != str[i + 1]) {
13                 str[i - 1] = str[i] == '1' ? '0' : '1';
14             } else {
15                 str[i] = str[i] == '1' ? '0' : '1';
16             }
17             current_char = str[i];
18             counter = 1;
19             answer++;
20         }
21     }
22 }
```

Código 2.1: Resolução para o caso geral $K > 2$

- `counter`, que armazena quantos caracteres iguais consecutivos foram contados até o momento.

Conforme mostrado no Código 2.1, sempre que `counter` atingir o valor K (linha 11) em uma posição `i`, são analisadas duas possibilidades de mudança: se a posição `i` atual não for a última e se `str[i]` for diferente de `str[i + 1]` (linha 12), a alteração é feita em `str[i - 1]`; caso contrário, o próprio `str[i]` é modificado, conforme expresso na observação 2 do início da Seção 2.3.3. Em seguida, a variável `counter` retorna ao valor 1, `current_char` armazena o atual caractere da posição `i` e `answer` é incrementado (linhas 17-19).

Para o caso $K = 2$, utilizam-se duas novas *strings*, `str_op1` e `str_op2`, ambas do mesmo tamanho de `str`: a primeira no formato "010101...", e a segunda "101010...", como descrito na observação 5 do fim da Seção 2.3.3. Para construí-las, como pode ser visto no Código 2.2, utiliza-se uma variável auxiliar `f` que começa em 0 e, a cada posição, alterna o seu valor entre 0 e 1 (linha 25). Assim, os caracteres das respectivas posições de `str_op1[i]` e `str_op2[i]` são determinados de acordo com o `f` atual, onde `str_op1[i]` segue o valor de `f` e `str_op2[i]` recebe o valor oposto (linhas 26-27). Após a construção destas duas strings, é feita a comparação de ambas com a string original `str`. Para cada posição `i`, os caracteres `str_op1[i]`

```
23  else { // k == 2
24      string str_op1, str_op2;
25      for(int i = 0, f = 0; i < n; i++, f ^= 1) {
26          str_op1.push_back(f ? '1' : '0');
27          str_op2.push_back(f ? '0' : '1');
28      }
29      for(int i = 0; i < n; i++) {
30          if(str_op1[i] != str[i]) op1++;
31          if(str_op2[i] != str[i]) op2++;
32      }
33      if(op1 <= op2) {
34          answer = op1;
35          str = str_op1;
36      } else {
37          answer = op2;
38          str = str_op2;
39      }
40  }
```

Código 2.2: Resolução para o caso particular $K = 2$

e `str_op2[i]` são comparados com `str[i]`: se `str[i]` for diferente de `str_op1[i]`, um contador `op1` é incrementado; se `str[i]` for diferente de `str_op2[i]`, um contador `op2` é incrementado (linhas 30-31). Após essas ações serem realizadas para todas as posições de `str`, define-se `answer` como o mínimo entre `op1` e `op2`, e a string `str` original se torna `str_op1` se `op1` $\leq$ `op2`, ou `str_op2` caso contrário (linhas 33-39).

Ao final, tanto para o caso $K > 2$ quanto para $K = 2$ a variável `answer` terá o valor da resposta ótima e a *string* `str` conterá essa resposta. Tudo o que resta agora é imprimir os resultados (Código 2.3).

```
41      cout << answer << ' ' << str << '\n';
```

Código 2.3: Impressão dos resultados obtidos

2.4 Análise e discussão

2.4.1 Análise da complexidade da solução

O algoritmo proposto apresenta uma única fase de processamento, na qual é realizada uma iteração sobre todos os caracteres da *string* de entrada S. Tanto

para qualquer caso em que $K > 2$ quanto para o caso particular $K = 2$, cada posição da *string* é analisada uma única vez, executando operações simples de custo constante. Desta forma, a complexidade de tempo da solução é $O(N)$, onde N representa o comprimento de S.

Para o caso $K = 2$, são utilizadas duas *strings* auxiliares proporcionais ao tamanho da *string* S. Ainda assim, o consumo de memória permanece linear em relação ao tamanho da entrada, uma vez que o número total de caracteres armazenados é proporcional a N. Portanto, a complexidade de espaço da solução é $O(N)$.

Dado que $N \leq 10^5$, o algoritmo proposto resolve o problema com folga para o tempo limite estabelecido de 0,5 segundos.

2.4.2 Discussão da solução

O algoritmo proposto é de natureza gulosa e resolve o problema de forma eficiente, realizando em cada passo escolhas localmente ótimas que não precisam ser reconsideradas conforme a execução prossegue, conduzindo a uma resposta final ótima. Entretanto, em uma possível variação deste problema no qual esta propriedade não se mantém, uma estratégia gulosa pode não ser suficiente, exigindo o uso de outras técnicas, como a programação dinâmica. Nesse cenário, pode ser necessário modelar uma solução que conta com múltiplos estados, o que aumentaria consideravelmente o custo computacional, fazendo com que o limite de $K \leq N \leq 10^5$ tivesse que ser drasticamente reduzido para que uma solução esperada permanecesse dentro do tempo de execução permitido.

3. PROBLEMA 2: HARMÔNICOS INTERFERENTES

O problema *Harmônicos Interferentes* inicia revelando que duas pessoas, Arthur e Bruna, estão testando um novo protocolo de envio de mensagens. Esse protocolo consiste no envio de uma mensagem *M* e uma sequência de controle *N* de Arthur para Bruna, onde ambas consistem em uma sequência de *bits*. O protocolo garante que o número inteiro representado pelos *bits* de *M* é um múltiplo do inteiro representado por *N*.

Para cada *bit*, tanto da mensagem *M* quanto da sequência de controle *N*, se corretamente transmitido, tem o seu valor correspondente (0 ou 1) armazenado no dispositivo receptor. No caso de uma interferência, a informação original do *bit* é perdida e o caractere '$*$' é armazenado no lugar. O resultado da transmissão é então identificado por M' e N'.

Se *bits* demais foram perdidos durante uma transmissão, a mensagem é simplesmente descartada. Contudo, se no máximo 16 *bits* foram perdidos, Bruna ainda gostaria de tentar recuperar a mensagem original. A tarefa é: dadas as sequências M' e N' recebidas por Bruna, determinar uma possível mensagem *M* que possa ter sido enviada por Arthur. Uma descrição completa do problema pode ser acessada aqui: https://codeforces.com/gym/105327/problem/H.

A estratégia utilizada na resolução do problema consiste na aplicação de uma busca completa para identificar todas as possíveis atribuições de valores aos *bits* perdidos. Além disso, emprega-se a aritmética modular para evitar que os valores calculados excedam os limites de representação dos tipos numéricos padrão da linguagem C++. A solução discutida neste capítulo foi elaborada pelo autor deste trabalho.

Nas seções a seguir, serão apresentados os conceitos que fundamentam a solução proposta, bem como as etapas de sua resolução. Entre essas etapas, estão a apresentação do algoritmo escolhido, o detalhamento da solução e, ao final, a análise da sua complexidade, com a menção a abordagens alternativas para a resolução do problema.

3.1 Conceitos

Nesta seção, serão introduzidos os conceitos de busca completa e *backtracking*, bem como o de aritmética modular. Tais conceitos são fundamentais para a

programação competitiva e serão utilizados para a construção da solução para o problema tratado neste capítulo.

3.1.1 Busca completa e *backtracking*

A busca completa (ou força bruta) é um método exaustivo que percorre todo espaço de soluções de um determinado problema e verifica, para cada uma dessas soluções, se atende ao objetivo estabelecido. Costuma ser empregada em conjunto com o *backtracking*, uma técnica sistemática de geração de soluções na qual cada uma delas é construída uma única vez, evitando repetições. Seu funcionamento consiste em estender, passo a passo, uma solução parcial de forma recursiva, até que uma resposta completa seja alcançada (SKIENA, 2020). Em alguns casos, é possível interromper a construção de algumas soluções parciais se elas já não podem mais levar a uma solução válida ou se a melhor solução encontrada até esse ponto supera qualquer possibilidade que possa ser gerada a partir desse ramo da busca.

3.1.2 Aritmética modular

A aritmética modular é amplamente utilizada na programação competitiva para assegurar que cálculos envolvendo números grandes não ultrapassem os limites de representação dos tipos padrão da maioria das linguagens de programação (HALIM; HALIM; EFFENDY, 2018). Quando os cálculos são realizados sob módulo m (com m geralmente sendo um número primo), os resultados de cada operação matemática ficam restritos ao conjunto de inteiros de 0 a m - 1. Para a aritmética modular, valem as seguintes propriedades (LAAKSONEN, 2020):

$$(x + y) \bmod m \equiv (x \bmod m + y \bmod m) \bmod m$$
$$(x - y) \bmod m \equiv (x \bmod m - y \bmod m) \bmod m$$
$$(x \cdot y) \bmod m \equiv (x \bmod m) \cdot (y \bmod m) \bmod m$$
$$x^n \bmod m \equiv (x \bmod m)^n \bmod m$$

3.2 Algoritmo

3.2.1 Motivação para a busca completa

Quando o espaço de busca é suficientemente pequeno, a busca completa torna-se um algoritmo extremamente útil na resolução de um problema, dada sua simplicidade, facilidade de compreensão e a sua garantia de encontrar uma solução ótima. Além disso, a busca completa pode ser utilizada em instâncias pequenas de problemas mais complexos que demandam técnicas mais eficientes, a fim de analisar a corretude dessas soluções por meio da comparação dos resultados obtidos (HALIM; HALIM; EFFENDY, 2018).

3.2.2 Funcionamento da busca completa

O funcionamento de uma busca completa recursiva costuma se dividir em dois momentos: (1) enquanto a solução está sendo construída, onde cada passo gera uma possibilidade de extensão da solução parcial atual e prossegue com a busca; (2) quando uma solução completa foi alcançada e ela é então avaliada. Os parâmetros de uma função dessa natureza são, geralmente, compostos por variáveis que indicam o estado em que a solução parcial se encontra, assim como um indicativo de até que ponto uma solução pode ser estendida.

Para exemplificar tal procedimento, o seguinte problema será abordado: gerar todas as *strings* binárias de tamanho K. Para a resolução desse problema, a decisão tomada em cada passo é a seguinte: para cada posição da *string*, consideram-se as possibilidades de incluir 0 ou 1 nessa posição. A busca inicia-se com uma solução vazia e prossegue recursivamente para cada possibilidade até que K posições tenham sido preenchidas, quando a solução é avaliada. Esses passos podem ser visualizados na Figura 3.1, que demonstra a busca gerando todas as cadeias binárias de tamanho $K = 3$.

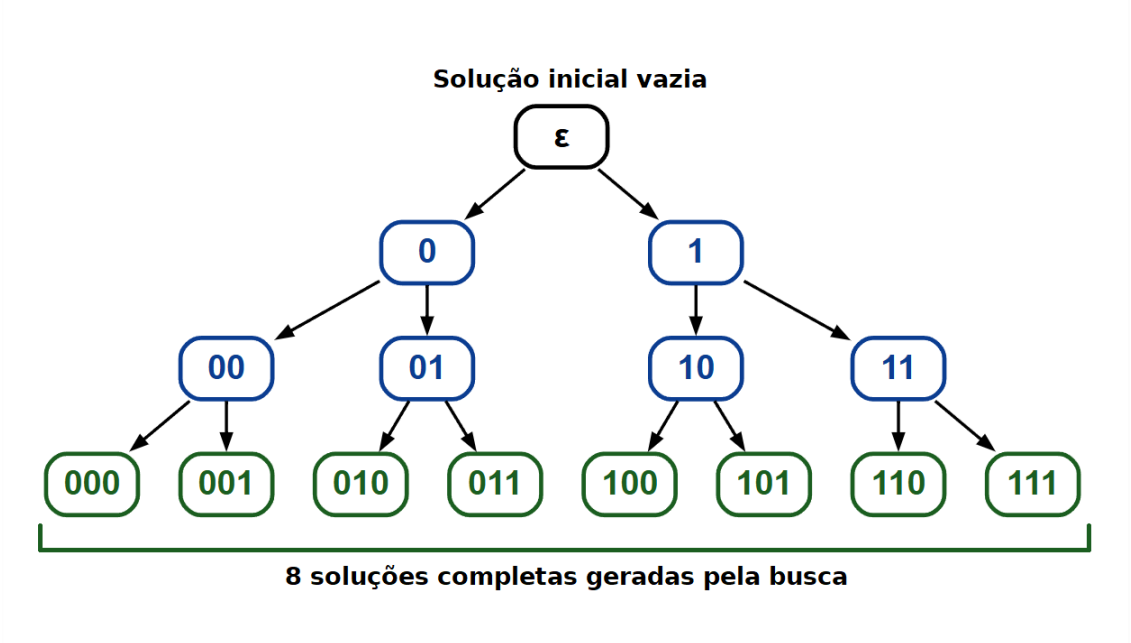


Figura 3.1: Árvore de busca gerando todas as *strings* binárias de tamanho 3

3.2.3 Implementação da busca completa

A função recursiva utilizada na resolução do problema da geração de todas as strings binárias de tamanho K, descrito na Seção 3.2.2, tem a sua implementação demonstrada no Código 3.1. Esta função é composta pelos seguintes parâmetros (linha 1):

- `string &str`, que contém a solução parcial construída até o momento (supõe-se que a *string* foi alocada anteriormente com tamanho `k`);
- `int i`, representando a posição atual da *string*, que deve ser preenchida com 0 ou 1;
- `int k`, que indica o tamanho alvo da *string*.

O caso base consiste em verificar a condição `i == k` (linha 2) que, se verdadeira, denota que uma solução completa foi alcançada, devendo então ser avaliada (neste caso, simplesmente exibida). Caso contrário, as possibilidades de extensão da solução são verificadas, preenchendo a posição `i` com 0 ou 1 e seguindo recursivamente para a próxima posição (linhas 5-8).

```
void search(string &str, int i, int k) {
    if(i == k) {
        cout << str << endl;
    } else {
        str[i] = '0';
        search(str, i + 1, k);
        str[i] = '1';
        search(str, i + 1, k);
    }
}
```

Código 3.1: Função recursiva que gera todas as *strings* binárias de tamanho *K*

3.3 Resolução

Como observado no início do capítulo, o objetivo do problema *Harmônicos Interferentes* é determinar uma possível atribuição de valores aos *bits* perdidos no envio das *strings M* e *N*, de modo que o inteiro representado por *M* seja divisível pelo inteiro representado por *N*. A resposta impressa pelo programa deve ser uma *string M* que satisfaça essa condição de divisibilidade. Nas seções a seguir, será apresentada a solução completa para o problema.

3.3.1 Descrição da entrada do problema

A entrada do problema consiste em duas linhas, que são as informações recebidas por Bruna. A primeira, representando M', onde $(1 \leq |M'| \leq 500)$. A segunda linha representa N', onde $(1 \leq |N'| \leq 16)$. Todos os caracteres de M' e N' são 0, 1 ou '$*$'. É garantido que N' possui ao menos um *bit* 1 e que não existem mais do que 16 caracteres '$*$' na entrada.

3.3.2 Descrição da saída do problema

A saída consiste em uma linha, representando uma possível mensagem *M* válida obtida a partir das informações recebidas por Bruna.

3.3.3 Solução

A primeira característica a ser notada sobre o problema é que, na entrada, nunca há mais do que 16 *bits* substituídos por caracteres '$*$'. Isso significa que uma solução que analisa todas as possibilidades de atribuição de valores para cada um desses *bits* é aceitável (no máximo um número da ordem de $6{,}6 \cdot 10^4$, como explicado mais detalhadamente na Seção 3.4.1). Essas possibilidades podem ser geradas e testadas de um modo recursivo, utilizando a técnica de *backtracking*.

As funções recursivas utilizadas na resolução desse problema são de natureza similar àquela utilizada para a geração de todas as *strings* binárias de tamanho K, descrita na Seção 3.2.3, com algumas modificações necessárias para adequar-se à resolução do problema. Uma dessas diferenças é que ao invés da busca ramificar em duas possibilidades (utilizar 0 ou 1) para todas as posições da *string*, essa ramificação acontece somente nas posições que sofreram interferência no recebimento do respectivo *bit*, ou seja, se a *string* contiver um '$*$' naquela posição. Para as posições que não sofreram interferência, basta apenas seguir o valor já existente na *string*, prosseguindo recursivamente até ser gerada uma solução completa. Esse mesmo procedimento é realizado para as duas *strings* M' e N' da entrada. Ao gerar completamente uma possibilidade para cada *string*, os inteiros que representam as cadeias de *bits* M e N, identificados por X e Y, respectivamente, são então avaliados: se X for divisível por Y, o programa deve exibir a *string* M construída pela busca e terminar a sua execução; se não, outras possibilidades devem ser analisadas até ser encontrada uma solução válida.

A segunda característica do problema a ser notada é que é necessário um procedimento que constrói, dígito a dígito, os inteiros que representam cada possibilidade de cadeia de *bits* gerada a partir de M' e N'. Tal procedimento consiste em utilizar um inteiro Z, inicialmente com valor 0, e na execução dos seguintes passos para cada posição da *string* de *bits*, denotada de forma genérica por S:

1. Multiplica-se Z pela base numérica em que a *string* S está escrita, nesse caso, a base 2;
2. Soma-se a Z o valor do dígito contido na respectiva posição, nesse caso, 0 ou 1.

As posições da *string* S serão avaliadas na ordem do *bit* mais significativo para o menos significativo.

Ao final da execução desse procedimento, a variável Z conterá o valor, na base decimal, do número representado pela cadeia de *bits* S. Esses passos de cons-

trução podem ser visualizados abaixo, onde é desenvolvido esse processo para uma *string* $S = 110$, sendo 6 o inteiro que ela representa:

Valor inicial: $Z = 0$.

- Posição $i = 0$, dígito $d = 1$:
  1. $Z \leftarrow 2 \cdot Z = 2 \cdot 0 = 0$
  2. $Z \leftarrow Z + d = 0 + 1 = 1$
- Posição $i = 1$, dígito $d = 1$:
  1. $Z \leftarrow 2 \cdot Z = 2 \cdot 1 = 2$
  2. $Z \leftarrow Z + d = 2 + 1 = 3$
- Posição $i = 2$, dígito $d = 0$:
  1. $Z \leftarrow 2 \cdot Z = 2 \cdot 3 = 6$
  2. $Z \leftarrow Z + d = 6 + 0 = 6$

Resultado: $Z = 6$.

O procedimento mencionado acima deve ser realizado para cada possibilidade de cadeia de bits gerada pelas duas *strings* M' e N' lidas da entrada, onde os passos mencionados são executados dentro das chamadas recursivas, para cada posição das *strings*. Qualquer inteiro gerado a partir de uma possibilidade de cadeia oriunda de N' não excede o limite máximo de um tipo numérico inteiro comum, visto que N' contém no máximo 16 *bits*. Contudo, há um problema: M' pode conter até 500 *bits*. Um número que contém 500 *bits* excede, por muito, o valor máximo que qualquer tipo numérico padrão pode representar em uma linguagem como *C++*. Entretanto, este problema pode ser contornado ao desfrutar da aritmética modular. Ao utilizar um número Y, que é o próprio número gerado a partir de uma possibilidade de cadeia para N', é possível fazer com que as operações matemáticas de construção do número X sejam feitas sob módulo Y. Isso assegura que o resultado final de X esteja restrito ao conjunto de inteiros de 0 a $Y - 1$, conforme explicado anteriormente na Seção 3.1.2. Essa operação é concretizada com a inclusão de um novo passo além dos passos 1 e 2 acima, que será utilizado somente na geração de X:

3. Realiza-se a operação de X mod Y (resto da divisão de X por Y), restringindo X ao intervalo de inteiros $[0, Y - 1]$.

Ao final da execução desses passos para todas as posições de uma cadeia gerada a partir de M', o caso base da busca recursiva é avaliado: se o valor final de X é 0, significa que ele deixa um resto 0 ao ser dividido por Y. Isso expressa que as

possibilidades das duas cadeias geradas a partir de M' e N' formam uma configuração válida de resposta, indicando que o número X é divisível por Y. Alcançando tal resultado, basta apenas exibir a *string* de *bits* gerada a partir de M' e encerrar a execução da busca.

3.3.4 Implementação da solução

Conforme discutido na Seção 3.3.3, a ideia principal da resolução do problema é, primeiro, executar uma busca para gerar uma possibilidade de número para Y, formado a partir de N'. Essa busca pode ser visualizada no Código 3.2. Ela recebe os seguintes parâmetros:

- `string &m`: representa a *string* M' lida da entrada;
- `string &n`: representa a *string* N' lida da entrada;
- `int y`: é o número gerado, passo a passo, a partir de uma possibilidade de atribuição de bits para `n`;
- `int i`: armazena o índice atual da *string* `n`.

```
1  void search_y(string &m, string &n, int y, int i) {
2      if(i == (int) n.size()) {
3          search_x(m, 0, y, 0);
4      } else {
5          if(n[i] != '*') {
6              y *= 2;
7              y += n[i] - '0';
8              search_y(m, n, y, i + 1);
9          } else {
10             y *= 2;
11             search_y(m, n, y, i + 1);
12             search_y(m, n, y + 1, i + 1);
13         }
14     }
15 }
```

Código 3.2: Função recursiva que gera todas as possibilidades para o número Y

Para cada posição da *string* `n`, as operações 1 e 2 vistas na Seção 3.3.3 são aplicadas. Para o caso em que o caractere de uma posição `i` não é '$*$', o dígito existente naquela posição é somado a `y` e a busca segue para a próxima posição (linhas

5-8). Se o caractere daquela posição for '∗', a busca se ramifica em duas possibilidades, escolhendo atribuir 0 ou 1 para esse dígito, prosseguindo recursivamente (linhas 9-12). O caso base da recursão é avaliado quando a variável `i` for igual ao tamanho da *string* `n`. Isso significa que uma possibilidade para o número `y` foi gerada por completo, e a partir de agora uma nova função de busca deve ser chamada para gerar um número a partir da *string* `m`. O resultado de `y` será passado como parâmetro para essa busca, onde será utilizado para a realização das operações modulares na construção do número `x`.

A função que gera um número a partir da *string* `m`, que pode ser observada no Código 3.3, é semelhante à função anterior, com algumas diferenças. Uma delas é a inclusão da operação de módulo, identificada pelo passo 3 da Seção 3.3.3. Para o caso em que o caractere da posição `i` não é '∗', o tratamento é similar ao da função anterior (linhas 8-12). Contudo, quando esse caractere é '∗' (linhas 13-20), além da inclusão do passo extra da operação de módulo, a *string* `m` tem o caractere da posição `i` modificado para 0 e 1, para cada possibilidade, e depois modificado de volta para '∗'. A razão disso é que ao final da busca, o resultado a ser exibido deve ser a *string* `m` gerada. Em cada passo da busca, as posições que contiverem caracteres '∗' são substituídas por uma das duas opções possíveis e, ao final, a *string* `m` será concordante com o número `x` construído ao longo da busca. Esse processo permite que nenhuma *string* adicional seja necessária para guardar o resultado final obtido. O caso base da função é avaliado quando a variável `i` atinge o tamanho da *string* `m` (linhas 2-6). O que acontece agora é a verificação da condição `x == 0` que, se verdadeira, significa que essa possibilidade para o número `x` gerado a partir de `m` é congruente a 0 módulo `y`, denotando que `x` é divisível por `y`. Neste caso, basta imprimir a *string* `m` na saída e encerrar a execução do programa. Caso a condição seja verificada como falsa, a busca prossegue até ser encontrada uma solução viável (é garantido que ao menos uma exista).

3.4 Análise e discussão

3.4.1 Análise da complexidade da solução

Visto que como para cada *bit* existem duas opções possíveis (0 e 1), a quantidade de combinações é determinada por 2^K, tal que K é a quantidade total de caracteres '∗' existentes. Como $K \leq 16$, isso se traduz em um número da ordem de $6{,}6 \cdot 10^4$ possíveis soluções a serem consideradas. Para cada uma dessas combinações, to-

```
void search_x(string &m, int x, int y, int i) {
    if(i == (int) m.size()) {
        if(x == 0) {
            cout << m << '\n';
            exit(0);
        }
    } else {
        if(m[i] != '*') {
            x *= 2;
            x += m[i] - '0';
            x %= y;
            search_x(m, x, y, i + 1);
        } else {
            x *= 2;
            m[i] = '0';
            search_x(m, x % y, y, i + 1);
            m[i] = '1';
            search_x(m, (x + 1) % y, y, i + 1);
            m[i] = '*';
        }
    }
}
```

Código 3.3: Função recursiva que gera todas as possibilidades para o número X

das as posições das duas strings M' e N' da entrada são avaliadas, sendo que para cada uma são aplicadas operações de custo constante. A complexidade de tempo da solução é então definida como $O(2^K \cdot (|M'| + |N'|))$, configurando-se na execução de aproximadamente $3{,}3 \cdot 10^7$ operações. Essa solução é aceitável dentro do tempo limite estabelecido pelo problema de 0,3 segundos.

3.4.2 Discussão da solução

Conforme explicado anteriormente na Seção 3.3.3, a existência de não mais do que 16 caracteres '$*$' na entrada permite o emprego de uma das mais elementares técnicas de resolução de problemas, que é a busca completa (força bruta). Tal técnica se encaixa muito bem para uma grande variedade de problemas em que a sua utilização é viável, dada a sua simplicidade de implementação e de entendimento, além da sua garantia de retorno de uma solução válida, caso ao menos uma exista. Uma busca completa recursiva resolve o presente problema de uma forma muito natural e direta, embora seja possível implementar a solução sem recursão com a utilização de *bitmasks*.

Uma dificuldade importante do problema abordado nesse capítulo é a necessidade de lidar com um número inteiro de até 500 *bits*, como mencionado na Seção 3.3.3. Isso torna o conhecimento de aritmética modular essencial ao implementar uma solução em linguagens que não tem suporte a inteiros de precisão arbitrária, como C++. Outras abordagens possíveis para a resolução desse problema, sem o uso da aritmética modular, seriam utilizar a linguagem *Java* e sua classe `BigInteger`, ou mesmo o próprio `int` padrão da linguagem *Python*.

4. PROBLEMA 3: DESVIO

O problema *Desvio* inicia revelando que o prefeito de uma cidade chamada *Nlogonia* pretende repavimentar alguns trechos de ruas. Isso significa que, enquanto um trecho estiver em obras, o mesmo não poderá ser utilizado, sendo necessário realizar um desvio. Cada trecho de rua tem um comprimento positivo W e interliga duas esquinas U e V da cidade, que pode ser transitado de U para V e vice-versa.

Um desvio é definido como uma rota alternativa que inicia em uma esquina U e termina em V, sem utilizar o trecho que conecta diretamente essas duas esquinas, enquanto ele estiver interditado para a sua repavimentação.

O objetivo do problema é determinar, para cada trecho de rua da cidade, qual o desvio de menor comprimento possível que interliga as duas esquinas associadas, enquanto o trecho estiver interditado. O enunciado do problema pode ser acessado em: https://codeforces.com/gym/104555/problem/D.

Interpretando a estrutura formada por esquinas e ruas como um grafo ponderado, simples e não dirigido, o problema pode ser resolvido utilizando o algoritmo de Dijkstra. A solução explora propriedades da árvore de caminhos mínimos produzida pelo algoritmo para calcular, de forma eficiente, os desvios de comprimento mínimo. Além disso, é necessário tratar separadamente um caso de exceção que não é contemplado diretamente pelo cálculo padrão. A solução apresentada neste capítulo foi desenvolvida pelo autor deste trabalho.

Na sequência, serão introduzidos os conceitos necessários para a compreensão da solução, mais especificamente os de grafo, algoritmo de Dijkstra e árvore de caminhos mínimos. A partir desses conceitos, será exposta a solução completa para o problema. Por fim, serão realizadas a análise de complexidade da abordagem adotada e a discussão de outras possibilidades de solução, apontando as vantagens e limitações de cada uma.

4.1 Conceitos

Nesta seção, os conceitos de grafo, caminhos mínimos de fonte única e árvore de caminhos mínimos serão introduzidos, que formarão a base para a construção da ideia e da solução proposta para o problema abordado neste capítulo.

4.1.1 Grafo

Um grafo é uma estrutura matemática usada para representar relações entre elementos. Formalmente, pode ser definido como um par $G = (V, E)$, em que V é um conjunto de vértices e E é um conjunto de arestas que conectam pares de vértices. Essas conexões podem ser de mão dupla ou de sentido único. Além disso, as arestas podem ainda possuir um custo ou capacidade associada, caso este em que o grafo é denominado ponderado (CORMEN et al., 2022).

4.1.2 Caminhos mínimos de fonte única

O problema dos caminhos mínimos de fonte única é um tema recorrente na programação competitiva, aparecendo tanto de forma direta quanto em diversas variações que impõem restrições adicionais ao caminho ou ao grafo. Dado um grafo ponderado $G = (V, E)$ e um vértice de origem $S \in V$, o objetivo é determinar, para cada vértice $U \in V$, o custo mínimo de um caminho que parte de S e chega até U, custo esse que é composto pela soma dos custos de todas as arestas utilizadas nesse caminho (HALIM; HALIM; EFFENDY, 2018).

4.1.3 Árvore de caminhos mínimos

A árvore de caminhos mínimos é uma árvore enraizada em um vértice S de um grafo $G = (V, E)$ que é composta de caminhos mínimos de uma origem S para todos os vértices U do grafo alcançáveis a partir de S. Uma propriedade central associada a essa árvore é a subestrutura ótima dos caminhos mínimos: se um caminho de menor custo vai de S até U, então todos os subcaminhos desse percurso, desde a origem até os vértices intermediários, também são mínimos (CORMEN et al., 2022). A árvore é produzida por algoritmos que resolvem o problema dos caminhos mínimos de fonte única, como o algoritmo de Dijkstra, que será apresentado na Seção 4.2. A Figura 4.1 ilustra um exemplo de grafo e sua respectiva árvore de caminhos mínimos enraizada no vértice 1.

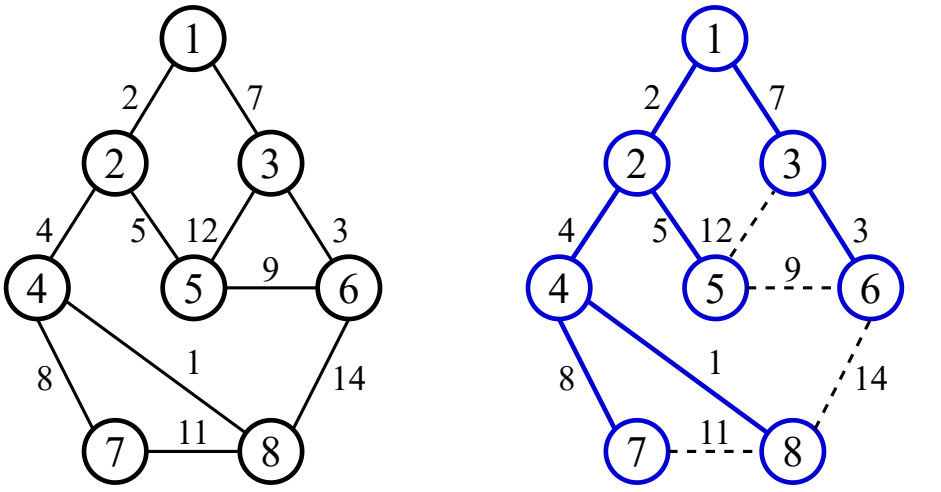


Figura 4.1: Árvore de caminhos mínimos enraizada no vértice 1

4.2 Algoritmo

4.2.1 Motivação para o algoritmo de Dijkstra

O algoritmo de Dijkstra é uma ferramenta fundamental na programação competitiva para resolver, de forma eficiente, o problema de caminhos mínimos de fonte única (*single-source shortest paths*) em grafos cujas arestas possuem custos não negativos. Em termos gerais, ele mantém um vetor de estimativas de distâncias a partir de uma origem S que vão sendo atualizadas durante a execução, até que as distâncias corretas sejam determinadas para todos os vértices alcançáveis (LAAKSONEN, 2020).

Para o problema discutido neste capítulo, o algoritmo de Dijkstra é uma escolha natural, já que permite obter eficientemente os caminhos mínimos a partir de uma origem, com uma complexidade de tempo de $O((M+N)\ \log\ N)$, em que M representa a quantidade de arestas e N o número de vértices do grafo. Através desse algoritmo, também é possível controlar com facilidade informações vinculadas à árvore de caminhos mínimos, que moldarão a ideia central da solução proposta.

4.2.2 Funcionamento do algoritmo de Dijkstra

O algoritmo utiliza uma fila de prioridade (isto é, um *heap mínimo*) para monitorar qual o vértice ainda não finalizado que possui a menor distância estimada a partir da origem. Inicialmente, a fila contém apenas o vértice de origem, com distância zero. Além disso, é mantido um vetor de distâncias, no qual são registradas as melhores estimativas encontradas até o momento para cada vértice. No início, para cada vértice do grafo exceto a origem, esse vetor contém um número que representa uma distância infinita.

O passo a seguir é repetido enquanto a fila não estiver vazia: retira-se da fila o vértice cuja estimativa é atualmente a menor. Em seguida, verifica-se se essa estimativa coincide com o valor registrado no vetor de distâncias; caso seja maior, ela é descartada por estar desatualizada. Caso contrário, o vértice é processado, tentando-se melhorar as estimativas de distância de seus vizinhos. Sempre que um vizinho tem sua distância reduzida, o vetor de distâncias é atualizado e esse vizinho é inserido novamente na fila com a nova estimativa.

Ao final da execução, o vetor de distâncias contém os custos mínimos de caminhos da origem para todos os vértices alcançáveis do grafo. Se para algum vértice a distância permanecer com o valor que representa infinito, isso indica que ele é inalcançável a partir do vértice inicial da busca. A Figura 4.2 ilustra um exemplo do passo a passo do algoritmo, com uma busca por caminhos mínimos originada no vértice 1.

Uma característica importante do algoritmo de Dijkstra, é que, sempre que um vértice é retirado da fila de prioridade pela primeira vez, sua estimativa de distância já corresponde à distância real da origem até esse vértice. Entretanto, essa propriedade vale apenas para grafos cujas arestas possuem custos apenas não negativos. A implementação desse algoritmo, que será apresentada a seguir na Seção 4.2.3, também pode ser aplicada a grafos que contêm arestas de custo negativo, desde que não existam ciclos de custo negativo. No entanto, nesse cenário, a complexidade de tempo deixa de ser $O((M + N) \log N)$ e pode crescer de forma exponencial, pois já não há mais garantia de que a primeira remoção de um vértice da fila de prioridade corresponda à sua distância final correta (HALIM; HALIM; EFFENDY, 2018).

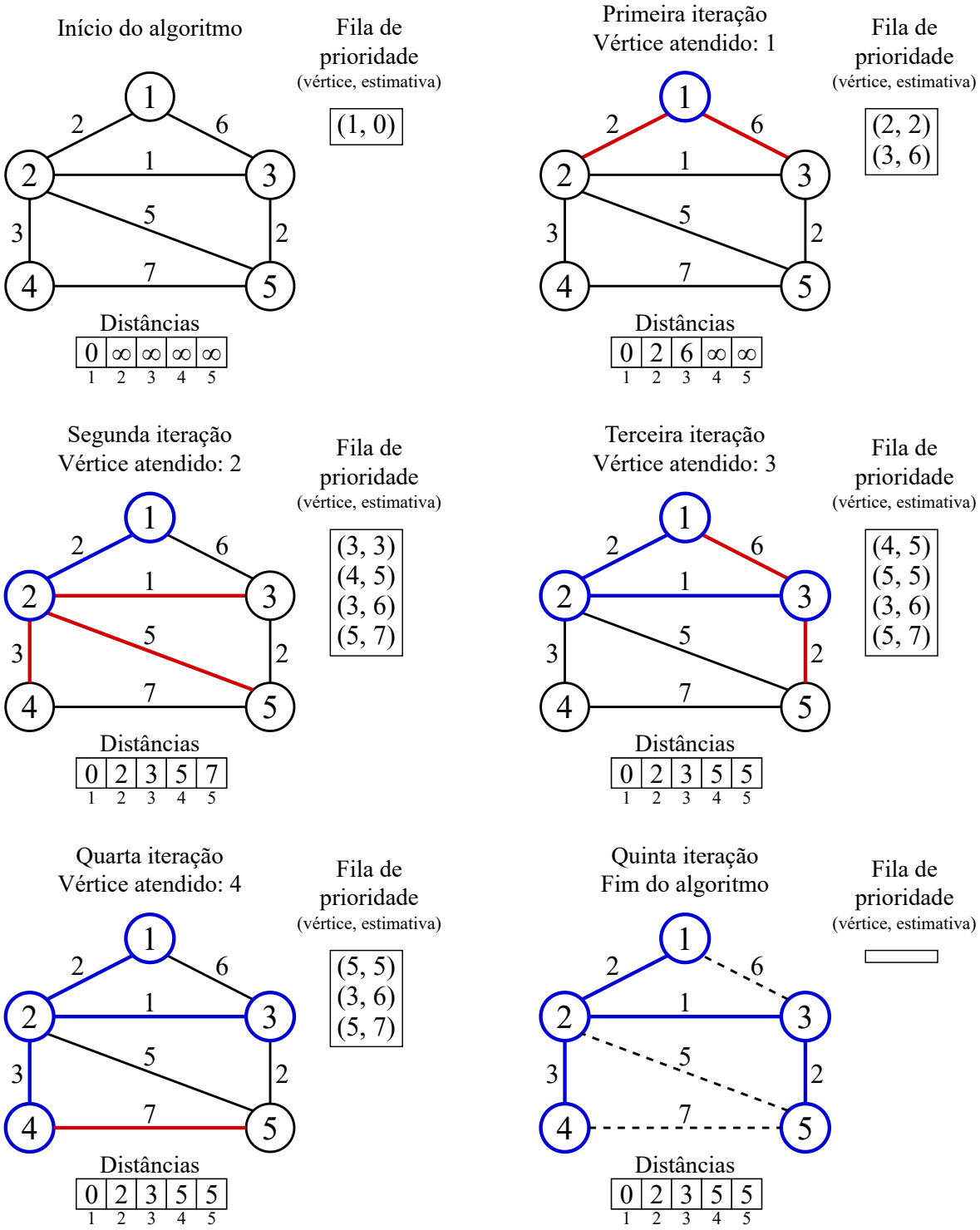


Figura 4.2: Iterações do algoritmo de Dijkstra para a origem 1

4.2.3 Implementação do algoritmo de Dijkstra

A implementação do algoritmo de Dijkstra é apresentada no Código 4.1, e leva como parâmetro um vértice `source` do grafo, que representa a origem dos caminhos (linha 7). Como parte do algoritmo, as seguintes constantes, variáveis e estruturas de dados são declaradas globalmente (linhas 1-5), e cada uma dessas informações tem seus significados explicados a seguir:

- `MAX`: constante que representa o limite máximo de vértices do grafo;
- `INF`: constante que representa infinito;
- `int n`: variável que armazena o número de vértices do grafo;
- `int m`: representa a quantidade de arestas do grafo;
- `int dist[MAX]`: vetor que guardará as distâncias do vértice de origem para todos os vértices;
- `vector<vector<pair<int, int>>> graph(MAX)`: estrutura onde está armazenado o grafo, onde para cada posição `graph[u]`, está contida uma lista de pares `{v, w}`, onde existe uma aresta entre `u` e `v` e `w` é o custo associado.

A primeira etapa da execução do algoritmo consiste na declaração da fila de prioridade `pq` como um *heap mínimo*, no preenchimento das distâncias para todos os vértices do grafo com `INF` (exceto pela origem `source`) e na inserção de `source` na fila com distância 0 (linhas 8-12).

A seguir, enquanto a fila `pq` não estiver vazia (linha 13), o vértice `u` com menor estimativa de distância no momento é extraído da fila (linhas 14-15). Se essa estimativa estiver incorreta, significando que o vértice retirado já foi processado, ele é ignorado (linha 16). Caso contrário, todos os vizinhos `v` desse vértice são processados: se a distância de `u` mais o custo `w` da aresta `(u, v)` resulta em uma estimativa menor do que a atual para `v`, o vértice `v` é inserido na fila de prioridade juntamente com sua nova estimativa, além da atualização do valor contido em `dist` (linhas 17-22).

```cpp
#define MAX 312
#define INF 1123456789

int n, m, dist[MAX];
vector<vector<pair<int, int>>> graph(MAX);

void dijkstra(int source) {
    priority_queue<pair<int, int>, vector<pair<int, int>>,
        greater<pair<int, int>>> pq;
    fill(dist, dist + n, INF);
    dist[source] = 0;
    pq.emplace(0, source);
    while(!pq.empty()) {
        auto [du, u] = pq.top();
        pq.pop();
        if(du > dist[u]) continue;
        for(auto &[v, w] : graph[u]) {
            if(dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                pq.emplace(dist[u] + w, v);
            }
        }
    }
}
```

Código 4.1: Função que implementa o algoritmo de Dijkstra

4.3 Resolução

De acordo com o que foi apresentado no início do capítulo, o propósito do problema *Desvio* é determinar, para cada trecho de rua da cidade, o valor do caminho de comprimento mínimo entre as esquinas a ele associadas, sem utilizá-lo. Na sequência, a solução será apresentada em detalhes.

4.3.1 Descrição da entrada do problema

A entrada do problema consiste em, primeiramente, uma linha contendo dois inteiros N e M ($1 \leq N \leq 300$). Eles significam, respectivamente, o número de esquinas

e de trechos existentes na cidade. Cada uma das M linhas a seguir descreve um trecho de rua, que é composto por três inteiros U, V e W ($1 \leq U \leq N, 1 \leq V \leq N, U \neq V, 1 \leq W \leq 10^6$), representando que há um trecho de mão dupla de comprimento W que interliga as esquinas U e V. É garantido que não existe mais do que um trecho de rua conectando o mesmo par de esquinas.

4.3.2 Descrição da saída do problema

A saída do problema deve consistir em M linhas, cada uma contendo um inteiro que representa qual a distância do desvio mais curto para cada trecho de rua da cidade, na mesma ordem que eles foram lidos da entrada. Se para o respectivo trecho não existir um desvio possível, o programa deve responder -1.

4.3.3 Solução

A partir do enunciado, pode-se concluir que a entrada descreve um grafo simples, ponderado e não dirigido, onde as esquinas representam os vértices e os trechos de ruas representam arestas com determinados custos. O objetivo do problema é, para cada aresta (U, V), determinar o caminho de menor custo entre os vértices U e V sem utilizar a própria aresta que os conecta diretamente. Para essa tarefa, é possível utilizar o algoritmo de Dijkstra, descrito anteriormente na Seção 4.2.

Uma forma direta e simples de resolver o problema seria executar o algoritmo de Dijkstra para cada aresta (U, V) do grafo. Fixando U como vértice de origem e V como destino, pode-se utilizar uma implementação padrão do algoritmo com apenas uma pequena modificação: ignorar a aresta que conecta diretamente esses dois vértices no cálculo do caminho mínimo entre eles. No entanto, essa abordagem é ineficiente, pois apresenta um custo de tempo muito elevado, como será discutido posteriormente na Seção 4.4.2. Dessa forma, torna-se necessária a adoção de uma estratégia mais eficiente.

Uma solução mais eficiente consiste em executar o algoritmo de Dijkstra para cada vértice do grafo (ao invés de para cada aresta). A ideia é, ao executar o algoritmo para um vértice de origem S, aproveitar para já calcular os desvios mínimos entre S e todos seus vizinhos. É importante notar que, embora para uma origem S e um vértice vizinho U a aresta que os conecta não possa ser utilizada no cálculo do desvio mínimo

entre S e U, essa mesma aresta pode ser considerada ao calcular o desvio mínimo entre S e outro vizinho V.

A chave para a resolução desse problema é observar, em um grafo qualquer, quais os desvios ótimos envolvendo uma origem U para todos os seus vizinhos. Uma ilustração pode ser visualizada na Figura 4.3, onde é mostrado um exemplo de grafo em que o vértice 1 é a origem.

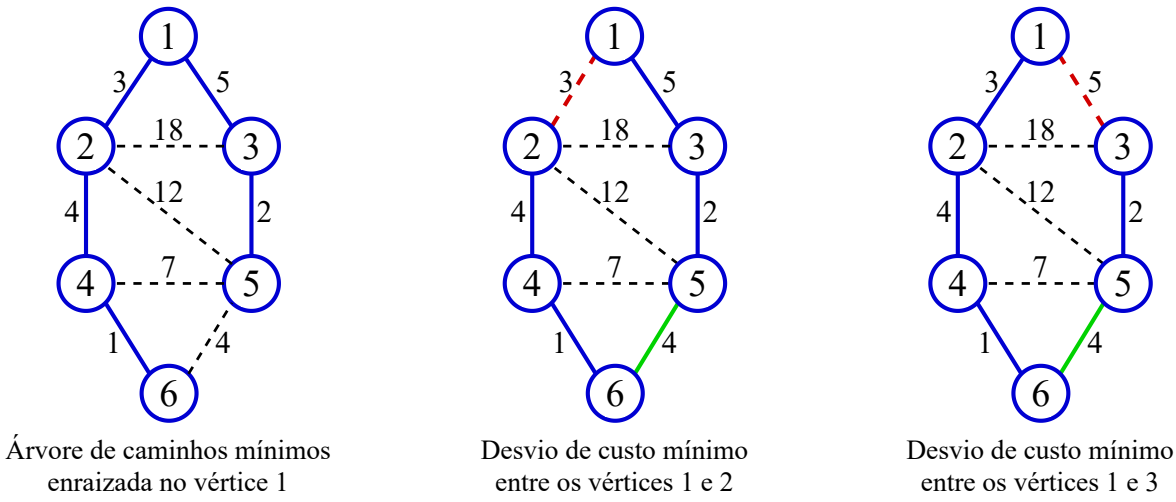


Figura 4.3: Semelhanças entre a árvore de caminhos mínimos a partir de 1 e os desvios ótimos de 1 a 2 e de 1 a 3

Nesse caso, é possível notar que os desvios mínimos envolvendo os vértices 2 e 3 têm muito em comum com a árvore de caminhos mínimos gerada a partir do vértice de origem 1 pelo algoritmo de Dijkstra. Na figura, a aresta destacada em verde é uma aresta que não havia sido incluída na árvore de caminhos mínimos e que foi levada em conta para o cálculo dos desvios. Já as arestas destacadas em vermelho fazem parte da árvore de caminhos mínimos e tiveram que ser ignoradas ao calcular os desvios de custo mínimo.

Com base nessas semelhanças, a solução para o problema consiste em executar o algoritmo a partir de cada possível origem S da seguinte maneira. Considere o momento em que, o algoritmo processa um vértice W tal que W é vizinho de um vértice T e T e W possuem, respectivamente, ancestrais distintos U e V na árvore de caminhos mínimos que são vizinhos de S (é possível que $T = U$ e $W = V$). Nesse passo serão calculadas as seguintes possibilidades de respostas de desvios de custo mínimo entre os vértices S e U e os vértices S e V:

$$\text{desvio}(S, U) = \text{distância}(S, T) + \text{distância}(S, W) + \text{aresta}(T, W) - \text{aresta}(S, U)$$

$$\text{desvio}(S, V) = \text{distância}(S, T) + \text{distância}(S, W) + \text{aresta}(T, W) - \text{aresta}(S, V)$$

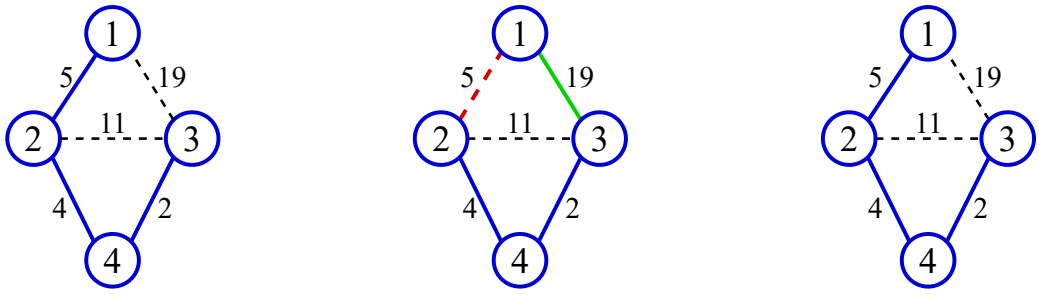
Para auxiliar nesse cálculo, durante a execução do algoritmo de Dijkstra com origem em S, é mantido um vetor *first*. *first[T]* indica, para cada vértice T do grafo, qual é o vizinho U de S que aparece como primeiro passo em um caminho mínimo de S até T. Em outras palavras, *first[T]* armazena o vértice vizinho de S que é o ancestral de T na árvore de caminhos mínimos enraizada em S. Essa informação será atualizada em cada momento que uma estimativa melhor de distância para cada vértice T é encontrada.

Utilizando como referência a Figura 4.3, é possível associar a origem S ao vértice 1. O momento em que os cálculos supracitados identificam os valores corretos dos desvios de custo mínimo entre 1 e seus vizinhos $U = 3$ e $V = 2$ é quando o vértice 5 é retirado da fila de prioridade durante a execução do algoritmo de Dijkstra. Nessa iteração, os vértices T e W correspondem a 5 e 6, respectivamente, e possuem ancestrais vizinhos de 1 na árvore de caminhos mínimos diferentes entre si ($U = 3$ e $V = 2$). Portanto, o valor do desvio de custo mínimo entre 1 e 2 será corretamente calculado como 16, e o desvio entre 1 e 3 calculado como 14.

Os cálculos dos desvios serão executados e armazenados em uma tabela *resposta* de tamanho $N \times N$. Posteriormente, na impressão das respostas, o valor do desvio de custo mínimo para uma aresta (U, V) é recuperado simplesmente consultando a célula *resposta*$[U][V]$. Como a situação descrita acima para os vértices U e V pode acontecer para diferentes vértices T e W, sempre que uma nova possibilidade de valor de desvio para os vértices S e U e os vértices S e V for calculada, deve-se armazenar na tabela o mínimo entre o valor já existente e o novo valor obtido.

A execução desses passos é suficiente para calcular o valor ótimo da maioria dos desvios requisitados, mas há alguns casos em que é preciso ter atenção extra. Um exemplo desse caso pode ser visualizado Figura 4.4. Esse cenário de exceção ocorre quando um vizinho U de uma origem S já possui, por si só, um caminho mínimo que não utiliza a aresta (S, U). Nesse contexto, as condições necessárias para aplicar diretamente o cálculo descrito anteriormente podem não ser satisfeitas, fazendo com que as respostas impressas ao final do programa tenham valores incorretos.

Para compreender melhor esse caso de exceção, considere a situação ilustrada na Figura 4.4, no momento em que o vértice 2 é retirado da fila de prioridade durante a execução do algoritmo de Dijkstra, tendo o vértice 1 como origem dos caminhos. Nesse ponto, são calculadas possibilidades de desvios de custo mínimo entre 1 e 2 e entre 1 e 3 (pois 2 encontra 3 na busca e, nesse instante, 2 e 3 possuem ancestrais vizinhos de 1 distintos entre si). No entanto, esses valores ainda são provisórios e não correspondem aos valores ótimos dos respectivos desvios. Além disso, a partir do vértice 2, as estimativas de distância para os vértices 3 e 4 são melhoradas, e as



Árvore de caminhos mínimos enraizada no vértice 1

Desvio de custo mínimo entre os vértices 1 e 2

Desvio de custo mínimo entre os vértices 1 e 3

Figura 4.4: Caso de exceção para o cálculo dos desvios de custo mínimo com vértice de origem 1

informações do vetor *first* são atualizadas. Assim, embora 3 seja vizinho de 1 no grafo original, passará a ficar registrado em *first[3]* o vértice 2. Mais adiante na execução do algoritmo, quando o vértice 4 é retirado da fila de prioridade, ele encontrará o vértice 3 como vizinho. Nesse momento, 3 e 4 têm o mesmo ancestral vizinho de 1 na árvore de caminhos mínimos (o vértice 2) e, portanto, a condição necessária para o cálculo dos desvios de custo mínimo entre 1 e 2 e entre 1 e 3 deixa de ser satisfeita, fazendo com que o algoritmo não identifique as respostas corretas.

O problema descrito acima pode ser contornado com o uso de um vetor auxiliar *is_neighbor*, em que *is_neighbor[U]* é definido como *verdadeiro* se U é vizinho da origem S e *false* caso contrário. Sempre que um vértice U é retirado da fila de prioridade, verifica-se se ele é vizinho de S e se o valor armazenado em *first[U]* é diferente de U. Esse é exatamente o caso que ocorre com o vértice 3 no grafo da Figura 4.4. Quando essas condições são verdadeiras, são calculadas novas possibilidades de desvios de custo mínimo entre os vértices S e U e entre S e V, em que V = *first[U]*:

$$\text{desvio}(S, U) = \text{distância}(S, U)$$
$$\text{desvio}(S, V) = \text{distância}(S, U) + \text{aresta}(S, U) - \text{aresta}(S, V)$$

4.3.4 Implementação da solução

Para auxiliar na resolução desse problema, alguns vetores são declarados globalmente, como podem ser vistos no Código 4.2. Essas estruturas juntam-se àquelas já existentes na implementação padrão do algoritmo de Dijkstra explicada na Seção 4.2.3. Os significados desses novos vetores são dados a seguir, e são definidos para cada origem S para a qual o algoritmo de Dijkstra é executado:

- `int first[MAX]`: para cada vértice, armazena qual é o vizinho da origem pelo qual se inicia o caminho mínimo a partir dessa origem;
- `int start_edge[MAX]`: para os vértices vizinhos da origem, armazena qual o custo da aresta que conecta o respectivo vértice à origem;
- `int is_neighbor[MAX]`: contém 0 (falso) ou 1 (verdadeiro) para cada vértice do grafo se ele é vizinho da origem;
- `int answer[MAX][MAX]`: matriz que registra as respostas requeridas pelo problema.

```
1  #define MAX 312
2  #define INF 1123456789
3
4  int n, m, dist[MAX], first[MAX], start_edge[MAX], is_neighbor[MAX],
5      answer[MAX][MAX];
6  vector<vector<pair<int, int>>> graph(MAX);
```

Código 4.2: Constantes, variáveis e estruturas globais necessárias para a resolução do problema

O primeiro passo a ser realizado, é a leitura e armazenamento do grafo dado pela entrada (linhas 1-10), visualizada no Código 4.3. Além disso, é feita a criação de mais uma estrutura auxiliar `vector<pair<int, int>> edges(m)` (linha 3) servindo para memorizar a ordem de entrada das arestas, que deverá ser respeitada ao realizar a impressão dos resultados obtidos pelas execuções do algoritmo de Dijkstra.

```
1  int u, v, w;
2  cin >> n >> m;
3  vector<pair<int, int>> edges(m);
4  for(int i = 0; i < m; i++) {
5      cin >> u >> v >> w;
6      u--; v--;
7      graph[u].emplace_back(v, w);
8      graph[v].emplace_back(u, w);
9      edges[i] = {u, v};
10 }
```

Código 4.3: Leitura da entrada e criação do vetor auxiliar `edges`

Os próximos passos, observados no Código 4.4, consistem em, primeiramente, inicializar a matriz `answer` com todos os valores de suas células definidos

como `INF`, representando que, inicialmente, nenhuma resposta válida foi obtida (linhas 11-15). A seguir, para cada vértice do grafo definido como origem, uma rodada do algoritmo de Dijkstra é executada (linhas 16-18).

```
11 for(int i = 0; i < n; i++) {
12     for(int j = 0; j < n; j++) {
13         answer[i][j] = INF;
14     }
15 }
16 for(int u = 0; u < n; u++) {
17     dijkstra(u);
18 }
```

Código 4.4: Inicialização da matriz `answer` e execução do algoritmo de Dijkstra para todos os vértices do grafo

A seguir, no Código 4.5, são apresentadas as adaptações feitas na inicialização do algoritmo de Dijkstra, onde informações importantes a respeito dos vizinhos da origem `source` são preenchidas. Além da fila de prioridade e do vetor de distâncias já mencionados anteriormente na Seção 4.2.3, o vetor `first` tem todas as suas posições inicializadas com -1, representando que, inicialmente, nenhum vértice do grafo foi alcançado por algum vizinho de `source` (linha 5). Já o vetor `is_neighbor` é preenchido com 0, significando que nenhum vértice foi identificado ainda como vizinho de `source` (linha 6).

```
1  void dijkstra(int source) {
2      priority_queue<pair<int, int>, vector<pair<int, int>>,
3          greater<pair<int, int>>> pq;
4      fill(dist, dist + n, INF);
5      fill(first, first + n, -1);
6      fill(is_neighbor, is_neighbor + n, 0);
7      for(auto &[v, w] : graph[source]) {
8          dist[v] = w;
9          first[v] = v;
10         start_edge[v] = w;
11         is_neighbor[v] = 1;
12         pq.emplace(w, v);
13     }
```

Código 4.5: Inicialização das informações sobre os vértices no algoritmo de Dijkstra

O próximo passo consiste no seguinte: como é necessário preencher as informações corretamente para os vizinhos do vértice inicial `source` (e não para o próprio `source`), a primeira iteração do algoritmo é feita à parte. Sendo assim, ao invés da fila de prioridade inicialmente conter o vértice de origem, os vizinhos dele são colocados,

e as seguintes informações, para cada um desses vizinhos, são preenchidas (linhas 7-11):

- `dist[v] = w`: a distância inicial de `v` é o custo da aresta `(source, v)`, ou seja, `w`;
- `first[v] = v`: o vértice vizinho de `source` que é o primeiro passo no caminho mínimo de `source` a `v` é, inicialmente, o próprio `v`;
- `start_edge[v] = w`: registra o custo `w` da aresta `(source, v)`;
- `is_neighbor[v] = 1`: o vértice é marcado como vizinho de `source`.

Após o preenchimento dessas informações, o vizinho `v` é adicionado à fila de prioridade com estimativa inicial de `w` (linha 12).

A seguir, será demonstrado o processo modificado de retirada e processamento dos vértices da fila de prioridade, juntamente com os cálculos dos desvios de custo mínimo de todos os vizinhos do vértice `source` da vez. Primeiramente, no Código 4.6, pode ser visualizado o tratamento do caso de exceção, descrito na Seção 4.3.3.

A identificação desse caso (linha 17) consiste em verificar se o vértice `u` da vez é vizinho de `source` e o caminho mínimo de `source` a `u` passa por outro vértice vizinho de `source` antes (o que corresponde a `first[u] != u`). Se verificado verdadeiro, os cálculos corretos dos desvios de custo mínimo do caso de exceção são aplicados.

```
13      while(!pq.empty()) {
14          auto [du, u] = pq.top();
15          pq.pop();
16          if(du > dist[u]) continue;
17          if(is_neighbor[u] == 1 && first[u] != u) {
18              answer[source][u] = dist[u];
19              int detour = dist[u] + start_edge[u] - start_edge[first[u]];
20              answer[source][first[u]] = min(answer[source][first[u]], detour);
21          }
```

Código 4.6: Tratamento do caso de exceção

Após essa etapa, o algoritmo prossegue com o processamento dos vizinhos de `u`, como pode ser observado no Código 4.7. Primeiramente, se um vértice `v` vizinho de `u` é a própria `source`, ele é ignorado, para evitar cálculos incorretos (linha 23). A seguir, a condição para o cálculo do desvio de custo mínimo para o caso padrão, detalhado na Seção 4.3.3, é avaliada (linha 24). Se verificada verdadeira, isso significa que `v` já foi descoberto anteriormente (`first[v] != -1`) e que os vértices `u` e `v` tem os

seus caminhos mínimos iniciados a partir de diferentes vértices vizinhos de `source`. Satisfeitas essas condições, são aplicados os cálculos dos desvios de custo mínimo entre `source` e `first[u]` e entre `source` e `first[v]` (linhas 25-28). Outra informação importante a ser registrada é, se o caminho mínimo calculado para `u` mais o custo `w` da aresta `(u, v)` reduz a estimativa de distância de `v`, `first[v]` deve receber o valor de `first[u]`. Isso significa que os caminhos mínimos de `source` para `u` e `v` partem do mesmo vizinho de `source`.

```
22          for(auto &[v, w] : graph[u]) {
23              if(v == source) continue;
24              if(first[v] != first[u] && first[v] != -1) {
25                  int detour_su = dist[v] + dist[u] + w - start_edge[first[u]];
26                  answer[source][first[u]] = min(answer[source][first[u]], detour_su);
27                  int detour_sv = dist[v] + dist[u] + w - start_edge[first[v]];
28                  answer[source][first[v]] = min(answer[source][first[v]], detour_sv);
29              }
30              if(dist[v] > dist[u] + w) {
31                  pq.emplace(dist[u] + w, v);
32                  dist[v] = dist[u] + w;
33                  first[v] = first[u];
34              }
35          }
```

Código 4.7: Tratamento do caso padrão e processamento dos vizinhos do vértice `u`

Após a execução do algoritmo de Dijkstra para todos os vértices do grafo, resta apenas recuperar e imprimir as respostas para todas as arestas, na ordem que são dadas pela entrada. Essa etapa é exposta no Código 4.8. Se uma célula `answer[u][v]` contiver o valor `INF`, significa que não existe nenhuma possibilidade de desvio de `u` para `v` sem utilizar a aresta que conecta esses dois vértices, onde a resposta a ser impressa deve ser -1. Se não, basta imprimir o valor lá existente.

```
36  for(auto &[u, v] : edges) {
37      cout << (answer[u][v] != INF ? answer[u][v] : -1) << '\n';
38  }
```

Código 4.8: Impressão das respostas para todas as arestas do grafo

4.4 Análise e discussão

4.4.1 Análise da complexidade da solução

A solução para o problema *Desvio* pode ser dividida em duas partes principais. A primeira consiste na leitura do grafo e na inicialização da matriz de respostas. A leitura exige $O(M)$ de tempo, sendo M o número de arestas do grafo de entrada, enquanto a inicialização da matriz `answer`, de dimensão $N \times N$, demanda $O(N^2)$, sendo N o número de vértices. Assim, o custo combinado dessa etapa inicial é $O(M + N^2)$.

A segunda parte corresponde à execução do algoritmo de Dijkstra a partir de cada vértice do grafo. Cada rodada do algoritmo tem complexidade $O((M+N)\ \log\ N)$. Na implementação apresentada, os cálculos que identificam os desvios de custo mínimo são operações de custo constante $O(1)$, realizadas durante a retirada dos vértices da fila de prioridade e durante o processamento das arestas, não impactando na complexidade do algoritmo. Executando essa função para todas as N possíveis origens, obtém-se um custo total de $O(N\ (M+N)\ \log\ N)$, que domina a complexidade de tempo final da solução.

É importante notar que o grafo de entrada pode ser completo, de modo que o número de arestas M pode chegar a aproximadamente N^2. Mesmo nesse pior cenário, com $N \leq 300$, o número total de operações fica na ordem de 10^8, o que é compatível com o limite de tempo estabelecido de 4,5 segundos. A complexidade de espaço da solução fica em $O(M + N^2)$, por conta das estruturas que armazenam o grafo, a lista de arestas e a matriz de respostas.

4.4.2 Discussão da solução

O problema é de alto nível de dificuldade pois exige um grande entendimento do algoritmo de Dijkstra e da estrutura da árvore de caminhos mínimos produzida por ele. Para a resolução do problema, é necessário observar propriedades não triviais a respeito do grafo, como o fato de que caminhos mínimos com origens comuns formam ramos que podem se “encontrar” durante a execução do algoritmo, além de ser preciso prestar atenção a casos particulares que fogem do padrão. Reconhecer esses encontros e interpretá-los como candidatos a desvios mínimos é a principal ideia por trás do algoritmo proposto.

Em oposição à dificuldade da solução aqui proposta, existe uma abordagem muito mais trivial, conforme mencionado na Seção 4.3.3. Tal abordagem consiste em executar o algoritmo de Dijkstra para cada aresta do grafo, ignorando-a ao calcular o caminho mínimo entre os dois vértices associados a ela. Como o grafo pode ser completo, essa estratégia exige a execução de $M \approx N^2$ rodadas do algoritmo. Assim, sua complexidade total é $O(N^2\ (M + N)\ \log\ N) \approx O(N^4 \log N)$, resultando em cerca de $6,6 \cdot 10^{10}$ operações, uma quantidade impraticável para os limites de tempo estabelecidos.

Uma abordagem alternativa para a resolução do problema utiliza o algoritmo de Floyd-Warshall, combinado com uma técnica de divisão e conquista para tratar o cálculo dos desvios de custo mínimo, atingindo uma complexidade de $O(N^3 \log N)$. Outra opção seria adotar um Dijkstra quadrático (sem fila de prioridade). Essa versão elimina o fator $\log N$ da complexidade, transformando-a em $O(N^3)$. Entretanto, essa versão só se comporta melhor na prática para grafos extremamente densos.

Por fim, vale destacar que a solução apresentada depende do fato de o grafo ser não dirigido e de todas as arestas terem custo não negativo. No caso de um grafo dirigido, a lógica usada para identificar desvios a partir do encontro de ramos durante o algoritmo de Dijkstra não seria mais válida, pois esse encontro não garante a existência de um caminho alternativo no sentido correto. Além disso, se houvesse arestas de custo negativo, o algoritmo de Dijkstra não poderia ser aplicado, sendo necessário recorrer ao algoritmo de Bellman-Ford. Como esse algoritmo possui complexidade de tempo de $O(NM)$, executar uma busca a partir de cada origem levaria a um número inviável de operações realizadas para $N \leq 300$, tornando essa variação intratável dentro do limite de tempo do problema.

5. PROBLEMA 4: MURALHAS REFORÇADAS

O problema *Muralhas Reforçadas* começa revelando que, antigamente, a proteção das cidades de Minas Gerais era feita apenas por muralhas frontais, devido à sua localização geográfica privilegiada e ao relevo favorável da região. Uma muralha era composta por um número de segmentos consecutivos, em que cada segmento era formado por uma determinada quantidade de blocos empilhados um em cima do outro.

Eventualmente, reforços eram aplicados a essas muralhas da seguinte maneira: escolhia-se um segmento específico e nele eram empilhados K blocos adicionais. Além disso, os $K - 1$ segmentos imediatamente à esquerda também eram reforçados em formato de escada, de modo que o primeiro recebia $K - 1$ blocos, o seguinte $K - 2$, e assim por diante, até que apenas 1 bloco fosse adicionado ou não houvesse mais segmentos à esquerda. O poder defensivo de uma muralha era estipulado pelo tamanho do seu menor segmento.

O objetivo do problema é, dada a descrição de uma muralha e um inteiro K, determinar qual a maior altura mínima possível ao aplicar uma única operação de reforço. A versão completa do enunciado encontra-se disponível em: https://codeforces.com/gym/106073/problem/M.

A solução para o problema baseia-se na identificação das características e da propriedade monótona de uma função que verifica se é possível garantir que a menor altura de toda a muralha seja ao menos um determinado valor X. Essa propriedade justifica a aplicação correta de uma busca binária para encontrar a resposta desejada. A adoção dessa abordagem foi motivada pelo editorial oficial disponibilizado pela Sociedade Brasileira de Computação (SBC).

Ao longo deste capítulo, serão apresentados os conceitos presentes na solução proposta, sendo eles o de função monótona e de busca binária. Com base nesses conceitos, será apresentada a solução completa para o problema. Mais adiante, será analisada a complexidade da solução adotada e serão mencionadas outras abordagens para o problema, baseadas em uma perspectiva distinta da utilizada na solução desenvolvida neste capítulo.

5.1 Conceitos

5.1.1 Função monótona

Uma função monótona não decrescente é uma função $f(X)$ tal que, sempre que $X < Y$, é tido que $f(X) \leq f(Y)$. De forma análoga, uma função monótona não crescente é aquela em que $X < Y$ implica $f(X) \geq f(Y)$ (CORMEN et al., 2022). A Figura 5.1 ilustra ambos esses casos, com $f(X)$ sendo uma função $f : \mathbb{R} \to \mathbb{R}$ monótona não decrescente e $g(X)$ sendo uma função $g : \mathbb{Z} \to \{0, 1\}$ monótona não crescente.

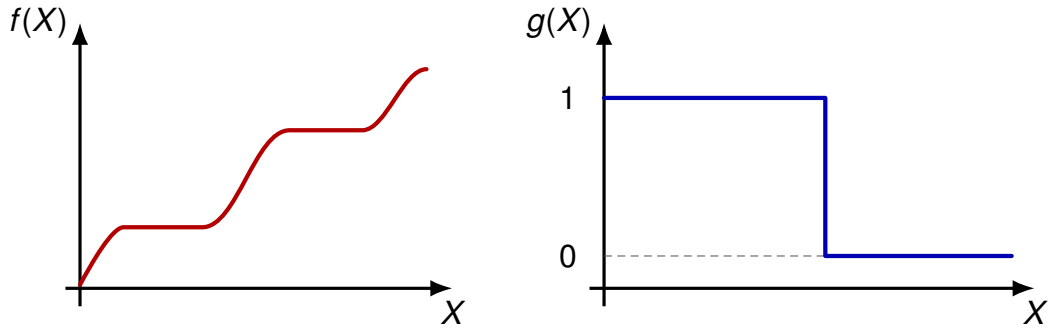


Figura 5.1: Exemplo de $f(X)$ monótona não decrescente e $g(X)$ monótona não crescente

5.1.2 Busca binária

A busca binária é uma das técnicas mais fundamentais na programação competitiva e em algoritmos de modo geral. Ela segue uma lógica de divisão e conquista na qual, em cada iteração, o espaço de busca é dividido ao meio; em seguida, realiza-se uma avaliação que permite descartar uma das metades e prosseguir apenas com a parte que ainda pode conter a solução. Esse processo se repete sucessivamente até que um determinado objetivo seja alcançado, reduzindo de forma eficiente o número de candidatos que precisam ser considerados (LAAKSONEN, 2020).

Um dos usos mais comuns da busca binária na programação competitiva ocorre em problemas de otimização, nos quais se busca maximizar ou minimizar uma resposta. Essa abordagem é conhecida como busca binária pela resposta (*binary search the answer*) (HALIM; HALIM; EFFENDY, 2018). Nesse cenário, dado um problema com seu objetivo e suas restrições, a busca procura determinar o maior valor X

admissível como resposta (em casos de maximização) ou o menor valor X (em casos de minimização). Para encontrar tal valor, é definida uma função *booleana* $f(X)$ que retorna *verdadeiro* quando X é avaliado como uma resposta admissível e *falso* caso contrário. Uma característica necessária para essa técnica funcionar, é que f deve ser uma função monótona. Sem essa propriedade, a busca binária não funcionará, pois o descarte de uma das metades do espaço de busca durante a sua execução pode resultar em decisões equivocadas que não levam à resposta ótima.

5.2 Algoritmo

5.2.1 Motivação para a busca binária

A busca binária é um método eficiente para encontrar uma solução sempre que o problema apresenta as características necessárias para garantir que a técnica conduza à resposta ótima. Sua lógica de particionar o espaço de busca ao meio e descartar uma das metades a cada iteração implica em uma complexidade de tempo de $O(\log M)$, em que M é o tamanho do espaço de busca considerado. Em cada um desses passos, uma função f é avaliada para decidir qual metade deve ser mantida. Assim, assumindo que a avaliação de f tenha um determinado custo $O(N)$, a complexidade total passa a ser $O(N \log M)$.

No problema apresentado neste capítulo, o uso dessa técnica torna-se adequado porque estão presentes as condições exigidas para uma aplicação correta da busca binária, conforme será discutido na Seção 5.3.3. Dessa forma, é possível determinar o valor ótimo desejado de maneira simples e eficiente, sem a necessidade de testar exaustivamente todos os valores candidatos à resposta.

5.2.2 Funcionamento da busca binária

A primeira etapa da busca binária consiste em definir o espaço inicial de busca, delimitado por dois valores: um limitante inferior e um limitante superior. O intervalo entre esses limitantes compõe o conjunto de candidatos que pode conter a resposta. Geralmente, esse espaço é formado pelos valores inteiros entre esses limitantes, mas há situações em que se trabalha com números reais, desde que exista um bom critério de parada.

Após essa definição inicial, a busca binária repete os seguintes passos:

1. Calcula-se o ponto médio entre os limitantes inferior e superior;
2. Avalia-se se esse valor é admissível como resposta;
3. Com base no retorno dessa avaliação, descarta-se uma das metades do intervalo, atualizando-se os limitantes para restringir o espaço de busca:
    - Se a resposta desejada estiver na metade inferior, o limitante superior é reduzido;
    - Caso contrário, o limitante inferior é aumentado.

Esse procedimento se repete até que os limitantes se igualem, que será a resposta retornada pela busca. Em alguns problemas, pode ocorrer de não existir nenhuma resposta válida dentro do intervalo considerado. Nessa situação, a busca pode retornar um valor especial indicando isso, ou pode-se realizar uma verificação final do valor retornado pela busca.

5.2.3 Implementação da busca binária

Existem diferentes maneiras de implementar a busca binária, onde cada variação pode atender melhor ao objetivo estabelecido pelo respectivo problema. O Código 5.1 implementa uma busca binária pelo maior valor possível pelo qual uma função `f` retorna `true`. Ela recebe dois parâmetros `inf` e `sup`, que representam os limitantes inferior e superior do espaço de busca (linha 1). A busca segue executando os passos citados na Seção 5.2.2, enquanto os limitantes `inf` e `sup` não se igualam (linha 3).

Em cada iteração, a variável `mid` recebe o cálculo do ponto médio entre os limitantes (linha 4). A forma com que esse cálculo é conduzido tem dois propósitos importantes. O primeiro é evitar *overflow*, pois não soma diretamente `inf + sup`. O segundo é uma ação necessária quando o objetivo é a maximização da resposta: ao usar o teto da divisão por 2 (somando 1 antes de dividir por 2), são evitadas situações em que `mid` seja igual a `inf` em intervalos pequenos, o que poderia impedir o avanço do limitante inferior quando `f(mid)` é verdadeira, fazendo com que a busca entre em *loop* infinito.

A função `f` é avaliada para `mid` dentro de cada iteração. Se o seu retorno for *verdadeiro*, o espaço de busca é restringido ao descartar todo o intervalo de busca

de `inf` até `mid - 1`, ajustando o limitante inferior para `mid` (linha 5). Caso contrário, o intervalo superior (que inclui o ponto médio) é descartado, ajustando o limitante superior para `mid - 1` (linha 6).

Quando `inf` e `sup` finalmente se igualam, o intervalo de busca foi reduzido a um único valor possível, que corresponde ao maior valor para o qual `f` retorna *verdadeiro*. Esse valor é então retornado pela função (linha 8).

```
1 int bin_search(int inf, int sup) {
2     int mid;
3     while(inf < sup) {
4         mid = inf + (sup - inf + 1) / 2;
5         if(f(mid) == true) inf = mid;
6         else sup = mid - 1;
7     }
8     return inf;
9 }
```

Código 5.1: Implementação da função que executa a busca binária

5.3 Resolução

Tendo em vista a descrição do problema exposta no início do capítulo, o problema *Muralhas Reforçadas* consiste em determinar, a partir das alturas iniciais dos segmentos de uma muralha e de um inteiro K, qual a maior altura mínima que pode ser obtida após a aplicação de uma única operação de reforço com K blocos. Nas seções subsequentes, será construída a intuição a respeito da solução, seguida de sua apresentação completa.

5.3.1 Descrição da entrada do problema

A entrada consiste, primeiramente, em uma linha contendo dois inteiros N e K ($1 \leq K \leq N \leq 10^5$), representando o número de segmentos da muralha e a quantidade de blocos adicionada ao segmento escolhido. A segunda linha é composta por N inteiros $v_1, v_2, \ldots, v_N$ ($1 \leq v_i \leq 10^9$), correspondendo à altura inicial de cada segmento.

5.3.2 Descrição da saída do problema

A saída deve consistir em um inteiro, representando qual a maior altura mínima de toda a muralha após a aplicação de um único reforço.

5.3.3 Solução

Suponha que, para uma instância qualquer do problema, a melhor resposta possível (isto é, a maior altura mínima que pode ser garantida após o reforço) seja um certo valor X. Para chegar a esse valor, é útil fazer algumas observações que guiarão a construção da solução para o problema.

O primeiro passo é determinar quais são os possíveis índices que devem receber o reforço inicial para garantir a resposta X desejada. Considere o último índice i tal que a altura inicial naquele segmento é estritamente menor que X, isto é, o maior i para o qual $v_i < X$. É obrigatório que esse segmento receba blocos adicionais, ou a altura mínima pretendida nunca chegará a X. Isso significa que qualquer escolha de índice para recebimento do reforço inicial de K blocos que não afete v_i a ponto de tornar esse valor ao menos X conduzirá a uma resposta errada.

Por outro lado, escolher um índice de reforço à direita de i, ainda que garanta que v_i receba os blocos necessários para fazer com que o seu valor final seja ao menos X, resultaria no desperdício de blocos de reforço em posições que não precisariam ser reforçadas. Isso poderia fazer com que faltassem blocos para segmentos mais à esquerda que também estão abaixo de X, impedindo que todos eles atinjam a altura desejada. Dadas as características descritas, a melhor escolha de índice para receber o reforço inicial de K blocos, de forma a garantir o valor ótimo X, é justamente o último índice i tal que $v_i < X$.

O segundo passo da solução consiste em traçar uma estratégia para encontrar o próprio valor ótimo X. É possível notar que, para qualquer valor subótimo Y tal que $1 \leq Y < X$, a escolha do índice inicial em que o reforço será aplicado, seguindo a lógica detalhada anteriormente, tende a deslocar-se mais à esquerda em relação à posição identificada para o valor X. Embora isso aconteça, garantir que a menor altura de toda a muralha seja pelo menos Y sempre será possível. Isso se justifica ao observar que, para qualquer posição i que necessitava de reforço para atingir o valor ótimo X, ao considerar o valor subótimo Y, ela vai receber ainda mais blocos ou não será sequer necessário que essa posição seja reforçada.

A Figura 5.1 demonstra a ideia descrita acima, para uma instância com $N = 6$ segmentos e incremento inicial $K = 5$, cuja resposta ótima é 5. A figura expõe quais índices foram escolhidos que garantiram a resposta ótima e todos os outros valores subótimos maiores que 0.

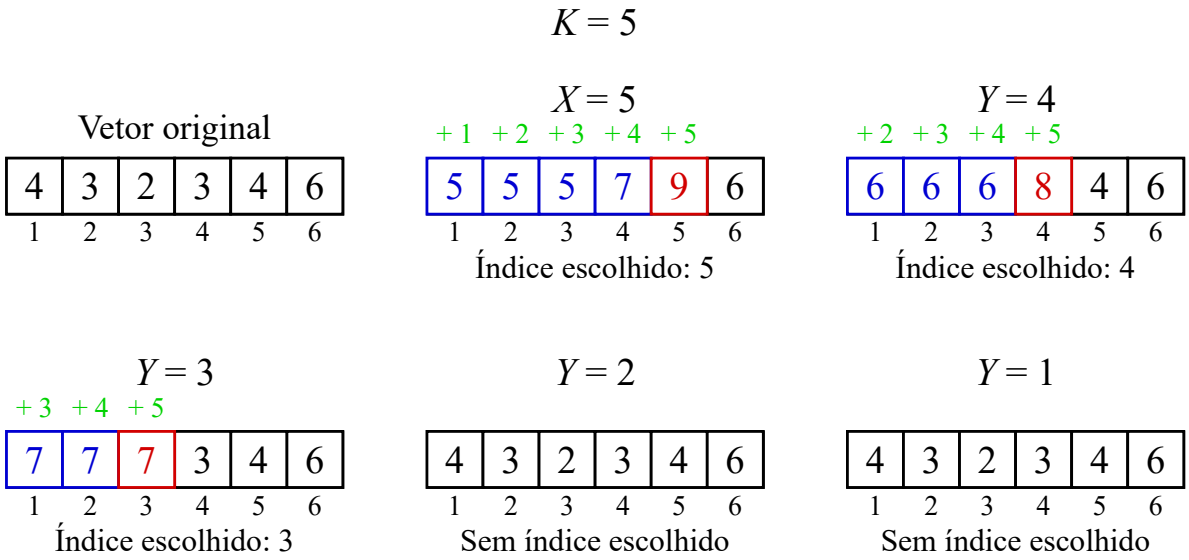


Figura 5.2: Índices escolhidos para reforço considerando a resposta ótima X e todos os valores inferiores Y maiores que 0

Pode-se então definir uma função *booleana* $f(X)$ da seguinte forma:

- $f(X)$ = *verdadeiro* se é possível escolher um índice para aplicar o reforço tal que, após a operação, a altura mínima da muralha seja pelo menos X;
- $f(X)$ = *falso* caso contrário.

As características identificadas anteriormente indicam uma natureza monótona da função f: se $f(X)$ é verdadeira para um certo valor X, então também será verdadeira para qualquer valor $Y < X$. A solução para o problema consistirá, então, na definição de um espaço de busca compatível com as restrições do problema, e na execução de uma busca binária, pelo maior valor X tal que $f(X)$ retorna *verdadeiro*. Esse valor será a resposta impressa pelo programa.

5.3.4 Implementação da solução

Primeiramente, são declaradas algumas variáveis globais que armazenam informações importantes utilizadas na solução do problema, conforme pode ser observado no Código 5.2. São elas:

- `#define MAX 112345`: quantidade máxima de segmentos que podem existir;
- `int n`: número de segmentos;
- `int k`: quantidade de blocos a serem adicionados na posição escolhida e nas posições à esquerda, em formato de escada (conforme a descrição do problema apresentada no início do capítulo);
- `walls[MAX]`: vetor que conterá as alturas dos segmentos.

```
1 const int MAX = 112345;
2 int n, k, walls[MAX];
```

Código 5.2: Variáveis globais utilizadas na solução

A leitura e o armazenamento das informações fornecidas pela entrada é feita no Código 5.3 (linhas 1-4). Em seguida, a função de busca binária é chamada (linha 5). A implementação dessa função é a mesma apresentada previamente na Seção 5.2.3. Os limitantes inferior e superior passados como parâmetros para a função cobrem todo o espaço de soluções possíveis, com base nas restrições estabelecidas pelo enunciado do problema.

```
1 cin >> n >> k;
2 for(int i = 0; i < n; i++) {
3     cin >> walls[i];
4 }
5 cout << bin_search(1, 1123456789) << '\n';
```

Código 5.3: Leitura da entrada e chamada da função de busca binária

A implementação da função `f`, utilizada pela busca binária, é exposta no Código 5.4. O parâmetro `x` representa qual o valor que a função deve avaliar se, aplicando a operação de reforço, é possível garantir que a menor altura contida no vetor seja ao menos `x`. A variável `flag` registrará quando a última posição do vetor que contém um valor menor que `x` já foi encontrada (veja a explicação da Seção 5.3.3). Já a variável `add` armazenará a quantidade de blocos `k` inicialmente adicionada ao segmento escolhido, valor que é reduzido em uma unidade a cada nova posição à esquerda.

A iteração sobre o vetor `walls` é feita de trás para frente (linha 3), para encontrar qual a última posição cujo valor contido é menor que `x` (linhas 4-6). Assim que identificado esse índice, a operação de reforço deve ser realizada (linhas 7-10).

```
1  bool f(int x) {
2      int flag = 0, add = k;
3      for(int i = n - 1; i >= 0; i--) {
4          if(walls[i] < x) {
5              flag = 1;
6          }
7          if(flag) {
8              if(walls[i] + add < x) return false;
9              add = max(add - 1, 0);
10         }
11     }
12     return true;
13 }
```

Código 5.4: Processo de avaliação da função `f` para um determinado `x`

A partir desse momento, se alguma posição cujo valor somado ao reforço recebido não é suficiente para atingir o valor `x`, a função retorna `false` imediatamente. O valor do reforço destinado a cada posição é então decrementado para seguir o padrão de escada, com o uso de um mecanismo para impedir que esse valor se torne negativo, caso existam posições ao início do vetor que não sofrerão impacto da operação de reforço. Finalmente, após todas as posições serem avaliadas, a função retorna *verdadeiro* se nenhum valor contido no vetor ficou abaixo de `x` (linha 12).

5.4 Análise e discussão

5.4.1 Análise da complexidade da solução

A execução da função `f` consiste em uma iteração sobre todas as `n` posições do vetor `walls`, utilizando operações constantes para controle das informações necessárias. Portanto, o custo de tempo dessa função é $O(N)$. Conforme mencionado na Seção 5.2.1, a busca binária executa $O(\log M)$ vezes a função `f`, em que M é o tamanho do espaço de busca estabelecido. A complexidade final de tempo da solução é então definida como $O(N \log M)$. Dado que M é um número da ordem de 10^9 e que N é no máximo 10^5, a solução proposta executa uma quantidade de aproximadamente $3 \cdot 10^6$ operações, o que é aceitável dentro do tempo limite estabelecido de 0,5 segundos. A complexidade de espaço fica em $O(N)$, visto que nada além do mínimo necessário para armazenar os dados de entrada é utilizado.

5.4.2 Discussão da solução

A propriedade monótona presente no problema, que permite o uso da busca binária, faz com que a solução apresentada seja direta, simples e eficiente. Em possíveis variações desse problema nas quais essa propriedade não se mantém, outras técnicas devem ser utilizadas, como algoritmos gulosos ou programação dinâmica.

Outra solução possível para esse problema envolve o uso da técnica de janela deslizante (*sliding window*). Nessa abordagem, é possível utilizar essa técnica em conjunto com uma estrutura que permita inserção e remoção de valores de forma eficiente, bem como a consulta ao menor valor nela contido. Um exemplo de estrutura desse tipo é o `multiset`, presente na biblioteca padrão da linguagem C++. Esse algoritmo consiste em testar a aplicação do reforço em todas as posições do vetor, ao invés de verificar se um determinado valor X pode ser alcançado ao escolher uma posição específica do vetor, como é feito na solução apresentada neste capítulo. O incremento em formato de escada aplicado pela operação de reforço pode ser tratado de forma eficiente com o emprego de mais uma técnica chamada de *Venice Set* (TORIBIO, 2018), para os elementos dentro da janela (os que sofrerão o impacto da aplicação da operação de reforço). A complexidade desse algoritmo é $O(N \log N)$.

Uma terceira alternativa de solução, também baseada na técnica de janela deslizante e no uso de *Venice Set*, consiste em precalcular os mínimos para todos os prefixos e sufixos do vetor e utilizar uma fila de duas pontas (`deque`) para realizar em tempo $O(1)$ amortizado, as operações de inserção, remoção e consulta ao valor mínimo dentro da janela (HALIM; HALIM; EFFENDY, 2018). Como a consulta aos mínimos dos prefixos e dos sufixos também possui custo constante por operação, a complexidade dessa solução é $O(N)$. Embora atingir essa complexidade não seja necessário para a resolução do problema apresentado nesse capítulo, essa última técnica pode ser extremamente útil ao lidar com variações desse problema com limites mais severos.

6. CONCLUSÃO

Este trabalho teve como objetivo apresentar e analisar a resolução de quatro problemas selecionados da Maratona de Programação da Sociedade Brasileira de Computação (SBC), explicando os conceitos necessários para compreendê-los e justificando as estratégias adotadas em cada solução. Isso serviu para criar um material que possa auxiliar competidores na preparação para as competições.

A metodologia consistiu na seleção desses problemas, no estudo detalhado dos enunciados e na identificação das propriedades relevantes para cada caso, seguidos pela elaboração das soluções, implementação em C++ e análise de complexidade. A revisão bibliográfica foi utilizada para fundamentar os conceitos e técnicas empregados, servindo como apoio teórico para as explicações apresentadas ao longo do trabalho.

O primeiro problema, *Kool Strings*, foi um problema de otimização cujo objetivo era realizar o menor número possível de operações de mudança de caractere para tornar uma *string* binária válida seguindo um determinado parâmetro. A solução adotada baseou-se em um algoritmo guloso que identificava, de forma sistemática, quais caracteres que deveriam ser modificados, de modo a assegurar o menor número possível de operações. Além disso, foi necessário tratar separadamente um caso especial que exigia uma estratégia específica para ser resolvido.

O segundo problema, *Harmônicos Interferentes*, foi resolvido por meio de uma busca completa, explorando todas as possíveis combinações de atribuições para os *bits* que foram perdidos durante a transmissão. Além disso, devido à quantidade elevada de bits de uma das sequências da entrada, foi necessário utilizar a aritmética modular para garantir que fosse possível testar a divisibilidade dos números sem depender de mecanismos para representação de números de precisão arbitrária.

O terceiro problema, *Desvio*, exigiu observações complexas sobre o conceito de caminhos mínimos em um grafo ponderado. Isso incluiu a identificação de características não triviais da árvore de caminhos mínimos construída pelo algoritmo de Dijkstra para calcular o valor ótimo dos desvios. Essas propriedades permitiram evitar a abordagem direta de executar o algoritmo separadamente para cada trecho de rua, resultando em uma solução mais eficiente. Além disso, foi necessário identificar casos de exceção que não seguiam o padrão principal da estratégia e que exigiam tratamento especial para garantir a corretude da solução.

O quarto problema, *Muralhas Reforçadas*, permitiu a aplicação de uma busca binária para determinar a maior altura mínima possível após uma única operação de

reforço. A solução explorou o fato de que a função que avaliava a viabilidade de uma solução candidata apresentava comportamento monótono, partindo de observações que identificaram qual índice deveria receber a operação de reforço para cada valor candidato. Dessa forma, tornou-se possível obter a resposta ótima de modo eficiente.

A partir deste trabalho, foi possível observar a interconexão entre o desenvolvimento de habilidades lógicas e a participação em competições de programação. Os problemas estudados e os conceitos discutidos favorecem a ideia de que aprofundar-se na programação competitiva fortalece a capacidade de abstração, o raciocínio analítico e a construção de soluções eficientes. Sendo assim, o desenvolvimento deste trabalho permitiu a criação de um material de apoio voltado a competidores e demais estudantes, reunindo conceitos, estratégias e técnicas relevantes para a resolução de problemas. Com isso, os benefícios desse tipo de prática tendem a se refletir tanto na formação acadêmica quanto na trajetória profissional, provendo conhecimento e maturidade para lidar com desafios diversos dentro da computação.

REFERÊNCIAS BIBLIOGRÁFICAS

CORMEN, T. et al. *Introduction to Algorithms*. 4th. ed. 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. (Competitive Programming 4: The New Lower Bound of Programming Contests in the 2020s, bk. 1).

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

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

SBC. *Maratona SBC de Programação: Sobre*. 2026. Acesso em: 25 de maio de 2026. Disponível em: <https://maratona.sbc.org.br/sobre/index.html>.

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

TORIBIO, C. *Venice Technique*. 2018. Acesso em: 25 de maio de 2026. Disponível em: <https://codeforces.com/blog/entry/58316>.

WING, J. M. Computational thinking. *Communications of the ACM*, Association for Computing Machinery, v. 49, n. 3, p. 33–35, 2006.

APÊNDICE A – CÓDIGO COMPLETO DO PROBLEMA 1

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

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    int n, k, counter = 0, answer = 0, op1 = 0, op2 = 0;
    char current_char;
    string str;
    cin >> k >> str;
    n = (int) str.size();
    if(k > 2) {
        current_char = '#';
        for(int i = 0; i < n; i++) {
            if(str[i] == current_char) {
                counter++;
            } else {
                current_char = str[i];
                counter = 1;
            }
            if(counter == k) {
                if(i < n - 1 && str[i] != str[i + 1]) {
                    str[i - 1] = str[i] == '1' ? '0' : '1';
                } else {
                    str[i] = str[i] == '1' ? '0' : '1';
                }
                current_char = str[i];
                counter = 1;
                answer++;
            }
        }
    } else { // k == 2
        string str_op1, str_op2;
        for(int i = 0, f = 0; i < n; i++, f ^= 1) {
            str_op1.push_back(f ? '1' : '0');
            str_op2.push_back(f ? '0' : '1');
        }
        for(int i = 0; i < n; i++) {
```

```
39              if(str_op1[i] != str[i]) op1++;
40              if(str_op2[i] != str[i]) op2++;
41          }
42          if(op1 <= op2) {
43              answer = op1;
44              str = str_op1;
45          } else {
46              answer = op2;
47              str = str_op2;
48          }
49      }
50      cout << answer << ' ' << str << '\n';
51      return 0;
52  }
```

Código A.1: Solução completa para o Problema 1 (*Kool Strings*), na linguagem C++

APÊNDICE B – CÓDIGO COMPLETO DO PROBLEMA 2

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

void search_x(string &m, int x, int y, int i) {
    if(i == (int) m.size()) {
        if(x == 0) {
            cout << m << '\n';
            exit(0);
        }
    } else {
        if(m[i] != '*') {
            x *= 2;
            x += m[i] - '0';
            x %= y;
            search_x(m, x, y, i + 1);
        } else {
            x *= 2;
            m[i] = '0';
            search_x(m, x % y, y, i + 1);
            m[i] = '1';
            search_x(m, (x + 1) % y, y, i + 1);
            m[i] = '*';
        }
    }
}

void search_y(string &m, string &n, int y, int i) {
    if(i == (int) n.size()) {
        search_x(m, 0, y, 0);
    } else {
        if(n[i] != '*') {
            y *= 2;
            y += n[i] - '0';
            search_y(m, n, y, i + 1);
        } else {
            y *= 2;
            search_y(m, n, y, i + 1);
            search_y(m, n, y + 1, i + 1);
```

```
39          }
40      }
41  }
42
43  int main() {
44      ios::sync_with_stdio(0);
45      cin.tie(0);
46      string m, n;
47      cin >> m >> n;
48      search_y(m, n, 0, 0);
49      return 0;
50  }
```

Código B.1: Solução completa para o Problema 2 (*Harmônicos Interferentes*), na linguagem C++

APÊNDICE C – CÓDIGO COMPLETO DO PROBLEMA 3

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

#define MAX 312
#define INF 1123456789

int n, m, dist[MAX], first[MAX], start_edge[MAX], is_neighbor[MAX],
    answer[MAX][MAX];
vector<vector<pair<int, int>>> graph(MAX);

void dijkstra(int source) {
    priority_queue<pair<int, int>, vector<pair<int, int>>,
        greater<pair<int, int>>> pq;
    fill(dist, dist + n, INF);
    fill(first, first + n, -1);
    fill(is_neighbor, is_neighbor + n, 0);
    for(auto &[v, w] : graph[source]) {
        dist[v] = w;
        first[v] = v;
        start_edge[v] = w;
        is_neighbor[v] = 1;
        pq.emplace(w, v);
    }
    while(!pq.empty()) {
        auto [du, u] = pq.top();
        pq.pop();
        if(du > dist[u]) continue;
        if(is_neighbor[u] == 1 && first[u] != u) {
            int detour = dist[u] + start_edge[u] - start_edge[first[u]];
            answer[source][u] = dist[u];
            answer[source][first[u]] = min(answer[source][first[u]], detour);
        }
        for(auto &[v, w] : graph[u]) {
            if(v == source) continue;
            if(first[v] != first[u] && first[v] != -1) {
                int detour_su = dist[v] + dist[u] + w - start_edge[first[u]];
                answer[source][first[u]] = min(answer[source][first[u]],
                    detour_su);
```

```
39                  int detour_sv = dist[v] + dist[u] + w - start_edge[first[v]];
40                  answer[source][first[v]] = min(answer[source][first[v]],
41                      detour_sv);
42              }
43              if(dist[v] > dist[u] + w) {
44                  pq.emplace(dist[u] + w, v);
45                  dist[v] = dist[u] + w;
46                  first[v] = first[u];
47              }
48          }
49      }
50  }
51
52  int main() {
53          ios::sync_with_stdio(0);
54          cin.tie(0);
55          int u, v, w;
56      cin >> n >> m;
57      vector<pair<int, int>> edges(m);
58      for(int i = 0; i < m; i++) {
59          cin >> u >> v >> w;
60          u--; v--;
61          graph[u].emplace_back(v, w);
62          graph[v].emplace_back(u, w);
63          edges[i] = {u, v};
64      }
65      for(int i = 0; i < n; i++) {
66          for(int j = 0; j < n; j++) {
67              answer[i][j] = INF;
68          }
69      }
70      for(int u = 0; u < n; u++) {
71          dijkstra(u);
72      }
73      for(auto &[u, v] : edges) {
74          cout << (answer[u][v] != INF ? answer[u][v] : -1) << '\n';
75      }
76      return 0;
77  }
```

Código C.1: Solução completa para o Problema 3 (*Desvio*), na linguagem C++

APÊNDICE D – CÓDIGO COMPLETO DO PROBLEMA 4

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

#define MAX 112345
int n, k, walls[MAX];

bool f(int x) {
    int flag = 0, add = k;
    for(int i = n - 1; i >= 0; i--) {
        if(walls[i] < x) {
            flag = 1;
        }
        if(flag) {
            if(walls[i] + add < x) return false;
            add = max(add - 1, 0);
        }
    }
    return true;
}

int bin_search(int inf, int sup) {
    int mid;
    while(inf < sup) {
        mid = inf + (sup - inf + 1) / 2;
        if(f(mid) == true) inf = mid;
        else sup = mid - 1;
    }
    return inf;
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    cin >> n >> k;
    for(int i = 0; i < n; i++) {
        cin >> walls[i];
    }
    cout << bin_search(1, 1123456789) << '\n';
```

```
39      return 0;
40  }
```

Código D.1: Solução completa para o Problema 4 (*Muralhas Reforçadas*), na linguagem C++

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

Certificate of Achievement

The 2023 ICPC South America/Brazil First Phase

02 September 2023

Honorable Mention

Universidade Federal da Fronteira Sul

João Henrique Alves dos Santos

Marco Balestrin

Bernardo Facchi

Andrei Braga, Coach

icpc International Collegiate Programming Contest

icpc.foundation

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

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

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

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