

CENTRO UNIVERSITÁRIO SERRA DOS ÓRGÃOS – UNIFESO
CENTRO DE CIÊNCIAS E TECNOLOGIA – CCT
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TÉCNICAS DE OTIMIZAÇÃO DE PROGRAMAÇÃO DINÂMICA

GABRIEL LAGOA DUARTE

Teresópolis
2017

CENTRO UNIVERSITÁRIO SERRA DOS ÓRGÃOS – UNIFESO
CENTRO DE CIÊNCIAS E TECNOLOGIA – CCT
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TÉCNICAS DE OTIMIZAÇÃO DE PROGRAMAÇÃO DINÂMICA

GABRIEL LAGOA DUARTE

Trabalho de Conclusão de Curso apresentado
ao Centro Universitário Serra dos Órgãos –
UNIFESO como requisito obrigatório para
obtenção do título de Bacharel em Ciência da
Computação.

Orientador: Rafael Gomes Monteiro

Teresópolis
2017

CENTRO UNIVERSITÁRIO SERRA DOS ÓRGÃOS – UNIFESO
CENTRO DE CIÊNCIAS E TECNOLOGIA – CCT
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TÉCNICAS DE OTIMIZAÇÃO DE PROGRAMAÇÃO DINÂMICA

GABRIEL LAGOA DUARTE

Trabalho de Conclusão de Curso aprovado como requisito parcial para obtenção do título de Bacharel no Centro Universitário Serra dos Órgãos – UNIFESO pela banca examinadora:

Rafael Gomes Monteiro (Orientador), M.Sc.

Eugênio da Silva, D.Sc.

Hermano Lourenço Souza Lustosa, M.Sc.

Teresópolis
22 de novembro de 2017

*Dedico esta monografia a minha família,
pelo apoio fornecido e aos meus amigos das
maratonas de programação.*

AGRADECIMENTOS

A Deus por minha vida, família e amigos.

Ao meu orientador Rafael Monteiro, pelo suporte e suas correções neste trabalho. Além disso, pelo incentivo e apoio nas maratonas de programação.

Aos meus pais, pelo amor, incentivo e todo o apoio concedido durante toda a minha vida.

E a todos que direta ou indiretamente fizeram parte da minha formação, o meu muito obrigado.

"The mind is the limit. As long as the mind can envision the fact that you can do something, you can do it, as long as you really believe 100 percent."

(Arnold Schwarzenegger)

LISTA DE FIGURAS

Figura 1	Gráfico das principais classes de complexidade	14
Figura 2	Árvore de recursão do Fibonacci de 5	16
Figura 3	Equações lineares graficamente	35

LISTA DE TABELAS

Tabela 1	Principais classes de funções para analisar algoritmos	14
Tabela 2	Otimizações de programação dinâmica	17
Tabela 3	Problema Knuth realizando cortes da esquerda para a direita	30
Tabela 4	Problema Knuth realizando cortes da direita para a esquerda	30

LISTA DE SIGLAS

SBC	Sociedade Brasileira de Computação
ICPC	International Collegiate Programming Contest
ICT	Information and Communication Technology
IDE	Integrated Development Environment
LIS	Longest Increasing Subsequence
RMQ	Range Minimum Query

LISTA DE ALGORITMOS

Algoritmo 1	Implementação Fibonacci sem programação dinâmica	16
Algoritmo 2	Implementação Fibonacci com programação dinâmica	16
Algoritmo 3	Implementação Mochila	24
Algoritmo 4	Implementação LIS	26
Algoritmo 5	Implementação Divide and Conquer	29
Algoritmo 6	Implementação Knuth Optimization	33
Algoritmo 7	Implementação Convex Hull Trick 1	38

RESUMO

Programação dinâmica é um dos principais temas de problemas das competições de programação, e seu uso é bem discutido na literatura. Porém, as técnicas de otimização são pouco discutidas na literatura e estas estão em alta nas últimas competições de programação que acontecem em todo o mundo. O objetivo deste trabalho é a criação de um material didático que explique algumas das principais otimizações de programação dinâmica, abordando suas características e particularidades, com o intuito de tornar o leitor capaz de aplicar esses conceitos em diversos problemas que compartilham das propriedades similares dos problemas que aqui serão demonstrados. Inicialmente uma fundamentação teórica é feita com a finalidade de esclarecer todos os termos que são importantes no decorrer do trabalho. Além desta parte técnica dos assuntos, foi feito um levantamento teórico sobre o ensino de algoritmos e programação, a fim de deixar as explicações das técnicas mais formais, minimizando a dificuldade de entendimento. Para apresentar cada técnica, foi desenvolvida uma metodologia que permite ao leitor acompanhar toda a linha de raciocínio, desde a descrição de um problema proposto, até a sua solução final, após a inclusão da otimização. Com isso, a leitura fica mais simplificada, pois como otimização de programação dinâmica é considerado um tema complexo e não recomendado para iniciantes em programação, houve a preocupação de criar um texto capaz de guiar os pensamentos do leitor e que tenha a capacidade de generalizar cada técnica para outros problemas. Dentre as otimizações possíveis, foram escolhidas as mais comuns nas maratonas de programação. São elas: Redução de Espaço, Estrutura de Dados RMQ, *Divide and Conquer Optimization*, *Knuth Optimization* e *Convex Hull Trick*.

Palavras-chave: Programação dinâmica. Otimização. Maratona de programação. *Convex Hull Trick*.

ABSTRACT

Dynamic programming is one of the main themes of programming contests, and its use is well discussed in the literature. However, optimization techniques are scarcely discussed in the literature and these are on the rise in the latest programming contests that take place around the world. The objective of this work is the creation of a didactic material that explains some of the main optimizations of dynamic programming, approaching its characteristics and particularities, with the intention of making the reader able to apply these concepts in several problems that share the similar properties of the problems that will be demonstrated here. Initially a theoretical referencial is made with the purpose of clarifying all the terms that are important in the course of the work. Besides this technical part of the subjects, it was made a search in the literature about on the teaching of algorithms and programming, in order to leave the explanations of the techniques more formal, minimizing the difficulty of understanding. To present each technique, a methodology was developed that allows the reader to follow the whole line of reasoning, from the description of a proposed problem, to its final solution, after including the optimization. With this, the reading becomes more simplified, because as dynamic programming optimization is considered a complex theme and not recommended for beginners in programming, there was the concern to create a text capable of guiding the thoughts of the reader and make he to has the ability to generalize each technique for other problems. Among all optimizations possibilities, the most common in programming contests were chosen. Are they: Space reduction, RMQ data structures, Divide and Conquer Optimization, Knuth Optimization, Convex Hull Trick.

Keywords: Dynamic Programming. Optimization. Programming contest. Convex Hull Trick.

SUMÁRIO

1	INTRODUÇÃO	11
2	FUNDAMENTAÇÃO TEÓRICA.....	13
2.1	COMPLEXIDADE DE ALGORITMOS	13
2.2	PROGRAMAÇÃO DINÂMICA	14
2.2.1	Otimizações	17
2.3	ENSINO DE ALGORITMOS	18
3	TRABALHOS RELACIONADOS.....	21
4	METODOLOGIA	22
5	DESENVOLVIMENTO.....	23
5.1	REDUÇÃO DE ESPAÇO	23
5.2	ESTRUTURA DE DADOS RMQ	25
5.3	<i>DIVIDE AND CONQUER OPTIMIZATION</i>	27
5.4	<i>KNUTH OPTIMIZATION</i>	30
5.5	<i>CONVEX HULL TRICK</i>	34
6	CONSIDERAÇÕES FINAIS.....	39
	REFERÊNCIAS.....	40

1 INTRODUÇÃO

As maratonas de programação são competições que exigem criatividade, trabalho em equipe e a capacidade de resolver problemas sob pressão (PIEKARSKI *et al.*, 2015). Essas competições possuem diversos problemas, que normalmente simulam uma situação real, onde um profissional da área deveria estar apto a resolver, se este aparecesse em um ambiente de trabalho, por exemplo. Cada exercício que deve ser resolvido faz parte de alguma subárea de conhecimento que está inclusa no grande universo que é a Ciência da Computação. Dentre todos temas possíveis para os problemas, destacam-se os seguintes: grafos, estrutura de dados, geometria computacional e programação dinâmica. Estes ganham um foco maior pelos estudantes e autores das provas, devido ao fato de que todos eles são conteúdos amplos e comumente utilizados fora das competições em problemas reais. Sendo assim, os problemas conseguem ficar cada vez mais concretos.

Dentre as técnicas mais comuns nas provas de maratona de programação, deve-se dar um destaque maior a programação dinâmica, pois ela está presente em todas as competições mais importantes que acontecem ao redor do mundo. Além disso, esta técnica é muito utilizada na indústria, quando há a necessidade de otimizar algum código, pois muitas vezes é possível realizar isso com a utilização desta. Por esses motivos, programação dinâmica é um tema bem conhecido e discutido, estando presente na grade curricular de muitos cursos e em diversos livros, principalmente nos de projeto e análise de algoritmos.

Apesar da programação dinâmica ser uma técnica que consegue otimizar várias soluções, somente o seu uso nem sempre é suficiente. As vezes ainda é preciso algo mais eficiente, como pode ser visto no problema *Internet Trouble*¹, que fez parte da XXI Maratona de Programação que é promovida pela SBC (Sociedade Brasileira de Computação) e no problema *Branch Assignment*², que fez parte da final mundial, realizada pelo ICPC (do inglês, *International Collegiate Programming Contest*). Nos dois problemas citados, a solução esperada, segundo os próprios autores, seria através de programação dinâmica. Porém, utilizar a solução mais intuitiva consumiria muito tempo de processamento para encontrar a resposta da questão, fazendo com que esta não seja uma solução viável. Se for feita uma análise mais profunda dos problemas, é possível perceber que estes possuem algumas particularidades que possibilitam realizar uma otimização da solução que utiliza programação dinâmica, e esta seria suficiente para resolver o problema.

As técnicas de otimização de programação dinâmica são normalmente assuntos mais avançados, que requerem um pouco mais de maturidade em programação e de algumas

¹ <http://maratona.ime.usp.br/hist/2016/index.html>

² <https://icpc.baylor.edu/worldfinals/problems/icpc2016.pdf>

estruturas de dados e algoritmos. Sendo assim, esses temas não são amplamente explorados na literatura e, como algumas técnicas são derivadas por pessoas que precisavam otimizar algum problema específico que estava sendo resolvido, muitas vezes essas não são formalizadas para que futuramente outros possam utilizar da mesma solução encontrada, o que torna o aprendizado dessas técnicas mais difícil. Até a data da confecção deste trabalho, não foi encontrado na literatura nenhum livro que aborde esse assunto. Algumas publicações na internet podem ser encontradas, porém, não há um material completo que explique todas as características e formas de utilização.

O objetivo deste trabalho é a criação de um material didático que formalize e explique algumas das principais otimizações de programação dinâmica, abordando suas características e particularidades, com o intuito de tornar o leitor capaz de aplicar esses conceitos em diversos cenários que compartilham das propriedades similares dos problemas que aqui serão demonstrados. Apesar deste trabalho ter um viés teórico com uma proposta mais didática, este não é recomendado para iniciantes em programação. As técnicas aqui elucidadas requerem que o leitor possua um conhecimento amplo em programação dinâmica, análise de algoritmos, estruturas de dados e matemática, para poder compreender todo o texto. Mesmo sendo explicado todos os passos das otimizações, os assuntos que são considerados iniciantes ou intermediários não serão abordados.

Este trabalho está organizado da seguinte forma: no capítulo 2 são discutidos os principais assuntos necessários para o melhor entendimento das técnicas, que são: programação dinâmica, complexidade de algoritmos e otimização. Além disso, neste capítulo são apresentados alguns trabalhos que visam formalizar algumas maneiras de como ensinar algoritmos e programação. No capítulo 3 são apresentados alguns trabalhos de conclusão de curso que possuem um objetivo similar ao deste, mesmo que não em programação dinâmica, mas sim no ensino de algoritmos. No capítulo 4 é definida como será a metodologia de ensino aplicada no trabalho. O capítulo 5 é responsável pela apresentação e explicação de todas as otimizações selecionadas. Por fim, são feitas as considerações finais no capítulo 6.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, são apresentados os conceitos que servem de insumo para a elaboração das etapas seguintes desse trabalho. Na seção 2.1 são demonstrados alguns conceitos necessários para identificar e classificar a complexidade de um algoritmo. A seção 2.2 explica o básico sobre programação dinâmica, seus principais termos e o conceito de otimização, que servirá de base para a elaboração dos capítulos seguintes. Por fim, na seção 2.3 é feita uma análise de alguns trabalhos relacionados com o ensino de programação.

2.1 COMPLEXIDADE DE ALGORITMOS

Na ciência da computação, analisar um algoritmo está relacionado com a identificação da quantidade de recursos necessários para sua execução, podendo ser a quantidade de memória utilizada, largura de banda de comunicação ou o uso do hardware do computador. Porém mais frequentemente a preocupação maior é em se medir o tempo computacional gasto para realizar determinado código (CORMEN *et al.*, 2009).

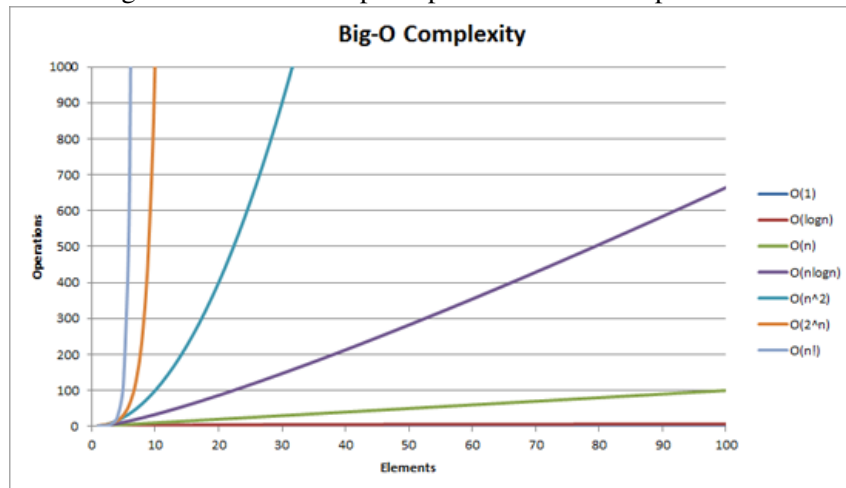
Quando é feita a análise de complexidade, é possível identificar a qual classe um determinado algoritmo pertence. A tabela 1 lista algumas classes de forma ordenada, sendo a primeira função a melhor possível, e a última, a pior. Após classificar um algoritmo, é possível escolher dentre diversas soluções para um mesmo problema, qual é a mais adequada no momento e poder saber, antes da execução, quanto tempo e memória o algoritmo irá gastar quando o tamanho da entrada for de tamanho n , onde n corresponde, geralmente, à quantidade de elementos que devem processados. Na figura 1 é possível observar o comportamento de algumas funções na medida que a quantidade de elementos aumenta.

Tabela 1 - Principais classes de funções para analisar algoritmos

Notação	Exemplo de algoritmos
$O(1)$	Determinar se um número é par ou ímpar
$O(\log n)$	Busca binária
$O(\sqrt{n})$	Determinar se um número é primo
$O(n)$	Procurar um elemento em um <i>array</i> não ordenado
$O(n * \log n)$	<i>Merge sort</i> ¹
$O(n^2)$	<i>Bubble sort</i> ²
$O(n^3)$	<i>Floyd-Warshall</i> ³
$O(n^c)$	Encontrar o maior emparelhamento em um grafo
$O(c^n)_{c>1}$	Resolver o problema do caixeiro viajante ⁴ com programação dinâmica
$O(n!)$	Resolver o problema do caixeiro viajante com força bruta

Fonte: Próprio Autor.

Figura 1 - Gráfico das principais classes de complexidade



Fonte: PERRETT, 2010

2.2 PROGRAMAÇÃO DINÂMICA

Programação dinâmica é uma técnica que combina soluções de subproblemas, de forma similar à técnica de divisão e conquista, que divide o problema em subproblemas, resolve

¹ Ver mais em: <http://quiz.geeksforgeeks.org/merge-sort/>

² Ver mais em: <http://quiz.geeksforgeeks.org/bubble-sort/>

³ Ver mais em: <http://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/>

⁴ Ver mais em: <http://www.geeksforgeeks.org/travelling-salesman-problem-set-1/>

cada um recursivamente e depois faz a junção das soluções para resolver o problema original. Porém este método é normalmente utilizado quando os subproblemas se sobrepõem, ou seja, um mesmo estado é encontrado diversas vezes na etapa de divisão. Portanto, se fosse aplicado um algoritmo ingênuo de divisão e conquista, um mesmo estado seria resolvido várias vezes, aumentando o custo computacional do algoritmo (CORMEN *et al.*, 2009).

Para resolver o problema de sobreposição, a técnica de programação dinâmica salva a resposta de todos os estados que vão sendo encontrados. Assim, no momento em que se depara com algo que já foi resolvido ela simplesmente retorna o valor que já estava armazenado.

A sequência de Fibonacci é um exemplo de fácil entendimento de quando é necessária a utilização de programação dinâmica. Esta é uma sequência de números inteiros que tem seu início com 0 e 1, e os termos subsequentes são uma soma dos dois últimos números. A sequência recebeu o nome do matemático Leonardo de Pisa, mais conhecido como Fibonacci, que no ano de 1202 descreveu o crescimento da população de coelhos utilizando esta sequência (HOM, 2013). Os primeiros termos são:

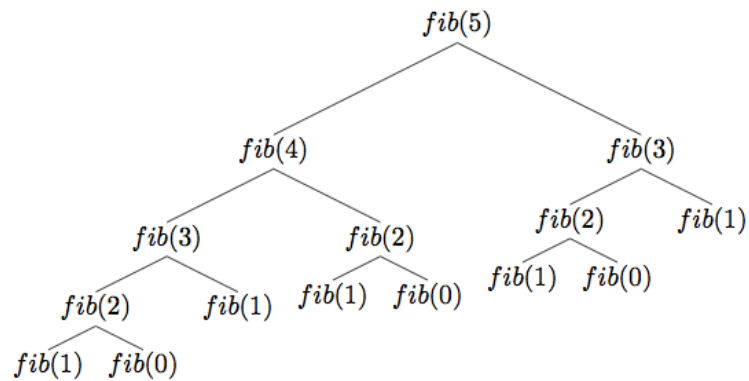
$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, \dots \quad (1)$$

A sequência pode ser representada através da seguinte recorrência, onde $fib(i)$ representa i -ésimo termo da sequência:

$$fib(i) = \begin{cases} i & \text{se } i \leq 1, \\ fib(i-1) + fib(i-2) & \text{se } i > 1. \end{cases} \quad (2)$$

Na figura 2 é apresentada a árvore de recursão gerada ao utilizar a equação 2 para o cálculo do $fib(5)$. Através dela, é fácil ver que diversos estados estão se repetindo, por exemplo: $fib(2)$ aparece três vezes e sempre que é encontrado ele é dividido no $fib(1)$ e $fib(0)$, assim deixando a complexidade deste algoritmo em $O(2^N)$. Ao aplicar programação dinâmica neste algoritmo é possível reduzir a complexidade para $O(N)$, pois cada estado será expandido uma única vez.

Figura 2 - Árvore de recursão do Fibonacci de 5



Fonte: SCHWARTZ, 2011

O código a seguir mostra como seria a implementação da função sem a utilização de programação dinâmica.

Algoritmo 1 - Implementação Fibonacci sem programação dinâmica

```

1 int fib(int i){
2     if(i <= 1)
3         return i;
4     return fib(i - 1) + fib(i - 2);
5 }

```

Para otimizar o código e utilizar programação dinâmica basta incluir uma tabela que salva todos os estados. Sua inclusão faz uma alteração mínima no código, como é mostrado no algoritmo 2.

Algoritmo 2 - Implementação Fibonacci com programação dinâmica

```

1 #define MAX 20
2 int tabela[MAX + 1];
3
4 int fib(int i){
5     if(tabela[i] != -1)
6         return tabela[i];
7     if(i <= 1)
8         return tabela[i] = i;
9     return tabela[i] = fib(i - 1) + fib(i - 2);
10 }

```

Para mais informações sobre programação dinâmica e suas técnicas, o site *TopCoder*⁵ possui um artigo amplo com vários problemas e dicas para soluções. Ele divide sua explicação em teoria e prática, começando nos tópicos mais simples e indo até alguns mais avançados.

2.2.1 Otimizações

Ao utilizar programação dinâmica para otimizar um problema, normalmente ocorre uma queda drástica na classe de complexidade associada à solução, como é o caso da sequência de Fibonacci, discutida na seção 2.2, onde foi possível sair de uma complexidade exponencial para uma linear. Apesar de parecer uma ótima forma de solucionar um problema, às vezes apenas aplicar programação dinâmica não é suficiente, e existem casos onde é possível e necessário otimizar ainda mais.

Para utilizar uma técnica de otimização de programação dinâmica, alguns critérios em relação a função de recorrência devem ser correspondidos. Cada técnica tem uma abordagem que possibilita a resolução de um conjunto de problemas que compartilham certas propriedades.

A tabela 2 ilustra algumas otimizações existentes, que possibilitam tanto a redução de espaço, quanto de tempo. Estas técnicas serão discutidas no capítulo 5.

Tabela 2 - Otimizações de programação dinâmica

Nome/Tipo	Característica
Redução de espaço	Reduz a quantidade de memória necessária quando um estado depende de uma quantidade fixa de outros estados
Estrutura de dados	Reduz a complexidade de tempo com o auxílio de uma estrutura de dados que consegue responder a consultas do tipo mínimo ou máximo em um intervalo de uma <i>array</i>
<i>Divide and Conquer</i>	Realiza divisão e conquista para encontrar o ponto ótimo necessário para se resolver o estado atual, reduzindo a complexidade temporal
<i>Knuth Optimization</i>	Utiliza informações de onde estava a solução ótima de um estado anterior para diminuir o espaço de busca dos outros, assim reduzindo a complexidade temporal
<i>Convex Hull Trick</i>	Através de conceitos geométricos, essa técnica possibilita a redução da complexidade temporal

Fonte: Próprio Autor.

⁵<https://www.topcoder.com/community/data-science/data-science-tutorials/>

2.3 ENSINO DE ALGORITMOS

O ensino de algoritmos, por tratar de assuntos que normalmente não são tão simples, com uma fundamentação matemática extensa, não é uma tarefa fácil. Portanto, foi feita uma busca na literatura de trabalhos que têm por objetivo propor uma metodologia de ensino, servindo de modelo para este trabalho.

Szlávi e Zsakó (2003) apresentaram diversas metodologias relacionadas ao ensino de programação. Os autores discutem sobre alguns modelos e exemplificam quando e para qual nível de estudante cada método será mais proveitoso ao ser aplicado. Estes métodos determinam a forma de estruturar o curso que deseja ser ensinado e a maneira de explicar os conteúdos. Os métodos discutidos no trabalho são:

- **Metodológico, orientado a algoritmos:** Neste há uma divisão bem clara em todas as etapas do processo de programação, desde a estruturação do que será desenvolvido, até a documentação final. Este é recomendado para ser utilizado com estudantes no final do ensino médio ou que estejam se preparando para um trabalho na área de informática.
- **Orientado a dados:** Este método é similar ao anterior, porém com foco nas estruturas de dados, assim, muitas vezes conseguindo evitar os grandes teoremas que normalmente são complexos e necessários para fazer a explicação dessas estruturas, propondo uma forma mais suave de transmitir o conhecimento.
- **Orientado à especificação:** A especificação da forma que o programa deve funcionar é a parte mais significativa, e todos os algoritmos são gerados sistematicamente através de rígidas instruções. Método indicado para estudantes universitários de informática, pois este será bem sucedido se os alunos possuírem um profundo conhecimento matemático.
- **Orientado a problemas:** Este método possui grandes diferenças em relação aos outros. Nele toda a etapa de programação é vista como uma única tarefa que não pode ser dividida. Dentre todos os métodos discutidos, este é o único recomendável para todos os tipos de estudantes, pois o foco é desenvolver nos alunos uma forma mais algorítmica de pensar sem estar preocupado com o treinamento profissional.
- **Orientado à linguagem:** Método semelhante ao orientado a problemas, porém o ensino depende totalmente de uma linguagem de programação e todos os ensinamentos são voltados especificamente para a linguagem adotada. Devido a estes fatores esse método está desatualizado e não é tão útil nos dias atuais.
- **Orientado a instruções:** Semelhante ao orientado a linguagem, porém aqui é definida uma linguagem geral, uma forma de pseudocódigo.
- **Orientado à matemática:** Os problemas a serem resolvidos com esta técnica, são retirados da matemática, onde cada problema individualmente é relacionado com algum outro utilizando

os princípios básicos da matemática. Esta não é uma metodologia eficaz para o ensino de programação. Porém, o ensino de matemática com o auxílio de programação pode ser útil devido à forma de raciocínio que as duas áreas possuem.

No trabalho de Robins e Rountree (2003), uma ampla pesquisa na literatura com o foco na parte educacional do estudo de programação foi feita. Diversos métodos e tópicos foram identificados e analisados para poder ser realizada uma classificação, e assim, auxiliar os professores a identificar em seus alunos características comuns e padrões, que poderão ser contornados com base no que já foi realizado e está documentado na literatura.

Pears *et al.* (2007) desenvolveu um *survey* que reúne algumas formas da literatura de ensinar introdução à programação. Além disso, os trabalhos reunidos, foram classificados e agrupados pela forma de ensino e pelos métodos aplicados.

Zsakó e Nóra (2008) realizaram uma análise dos principais métodos e aplicações que auxiliam no aprendizado e no ensino dos tópicos de ICT (do inglês, *Information and Communication Technology*). Para cada método é exemplificado o seu funcionamento, como realizar a sua aplicação e para qual nível de estudante ele é mais apropriado.

No trabalho desenvolvido por Papp-varga, Szlávi e Zsakó (2008), foi feita uma análise semelhante à que foi realizada no trabalho apresentado na subseção anterior. Porém, o foco deste é o ensino de uma linguagem de programação, portanto os métodos apresentados demonstram os passos ideais para transmitir os conceitos da linguagem proposta.

Radošević, Orehovački e Lovrenčić (2009), propõem a criação de uma ferramenta que facilite o aprendizado de linguagens de programação básicas, como C++, ao invés da utilização de IDE (do inglês, *Integrated Development Environment*). A finalidade desta ferramenta é ajudar os estudantes em não cometer erros que são comuns a quem está iniciando e, além disso, permitir ao professor auxiliar seus alunos de forma mais simples.

Vihavainen, Paksula e Luukkainen (2011), discutem como ensinar o básico de programação para quem está começando. O autor propõe um modelo de ensino e faz a aplicação deste em uma turma de um curso de ciência da computação. O modelo discutido tem três estágios. No primeiro o professor oferece aos estudantes um modelo conceitual de um problema e como o mesmo foi resolvido. A todo momento de explicação o professor verbaliza os seus pensamentos e explica o motivo de ter feito determinada decisão, deixando os alunos acompanharem toda sua linha de raciocínio. Na segunda etapa, o estudante é exposto a um problema que será resolvido com a orientação de um instrutor que não entrega diretamente a resposta, mas sim sugestões que permitam os alunos serem capazes de descobrir as respostas para suas próprias perguntas. A última etapa é quando os alunos conseguem resolver as tarefas sozinhos e não precisam mais do auxílio de um instrutor. Ao final do trabalho é feito um comparativo entre uma turma que utilizou a metodologia proposta e uma que não usou,

e os resultados foram positivos, mostrando que a quantidade de evasão e reprovação foram reduzidos.

Apesar de existirem diversos trabalhos que tratam sobre o ensino de algoritmos e programação, todos os encontrados têm um foco para um nível básico de conhecimento, levando em conta alunos que estão iniciando nesta área. Portanto, não será aplicado nenhum deste diretamente. No capítulo 4 será discutida a metodologia elaborada para a construção do trabalho.

3 TRABALHOS RELACIONADOS

Nesta seção são apresentados alguns trabalhos que tem um objetivo similar ao deste projeto. Em todos os encontrados a intenção do autor foi criar um conteúdo teórico sobre algoritmos e programação. Apesar de apenas um trabalho possuir conteúdo relacionado a programação dinâmica, todos eles foram considerados para a elaboração da metodologia a ser desenvolvida.

Tommasini (201-) em seu trabalho visa ensinar programação dinâmica para quem está iniciando nesta área. Sua maior motivação foi a falta de um bom material didático sobre esse tema. Seu trabalho tem um foco muito didático, apresentando diversas técnicas, problemas com soluções e propondo vários exercícios para o leitor praticar o que foi ensinado. Ao final do trabalho é apresentada uma lista de problemas de maratonas de programação sobre os assuntos desenvolvidos no texto.

No trabalho proposto por Dalalio (2013), foi realizado um estudo de algoritmos e estruturas de dados para a resolução de problemas relacionados a *String Matching*¹. Seu texto é bem didático, mostrando diversos problemas e as diversas formas de resolução, apresentando a complexidade e seu código. Ao final diversos problemas são propostos para que o leitor os resolva utilizando os conceitos elaborados no texto.

Couto (2016) desenvolveu um trabalho voltado ao ensino de algoritmos, com foco principal nos algoritmos aplicados em sequências de caracteres. Seu texto ficou dividido entre diversos capítulos, onde cada um deles apresentava uma estrutura de dados diferente, assim podendo mostrar ao leitor quando é melhor utilizar uma em relação às outras.

¹https://en.wikipedia.org/wiki/String_searching_algorithm

4 METODOLOGIA

Após a análise dos trabalhos levantados na seção 2.3, pode-se destacar os trabalhos elaborados por Szlávi e Zsakó (2003) e Vihavainen, Paksula e Luukkainen (2011). No primeiro, um dos métodos discutidos aborda uma forma mais metodológica para transmitir uma informação, onde em cada etapa do ensino há uma clara divisão. No segundo trabalho é apresentada uma forma de ensino baseada no desenvolvimento de uma linha de raciocínio, em que o aluno ou o leitor vai acompanhando todos os passos que o instrutor fez até chegar em uma conclusão.

A metodologia aplicada no próximo capítulo tomará como base esses dois trabalhos, com uma organização baseada em tópicos, porém sempre fazendo o leitor entender toda a linha de raciocínio. Serão inseridas algumas pausas a fim de deixar quem está lendo pensar em uma solução para um problema, antes de ser apresentada a resposta. Além disso, alguns exercícios serão propostos para quem desejar praticar o que está sendo demonstrado.

A estrutura de cada seção será dividida em seis partes, sendo elas:

- **Problema:** Será apresentado um problema com todas as informações que são importantes para que o mesmo seja resolvido através de programação dinâmica;
- **Solução ingênua:** A partir da descrição do problema será elaborada uma solução que utilize programação dinâmica, juntamente com uma análise de complexidade, podendo ser temporal e/ou de espaço;
- **Análise de particularidades:** Após uma solução ter sido desenvolvida serão levantadas algumas características do problema, principalmente com relação à forma que os cálculos estão sendo desenvolvidos, e como estão sendo armazenadas as informações, para que, com estes dados, possa ser feita alguma otimização. Neste ponto o leitor já teria condições de começar a pensar no que pode ser feito para melhorar a solução;
- **“Nome da técnica”:** Com todas as análises feitas, será apresentada a técnica em questão, mostrando o seu funcionamento e como esta pode ser aplicada no problema. O nome da seção irá variar de acordo com cada técnica;
- **Benefícios:** Tendo aplicada a otimização, uma análise dos benefícios será feita juntamente com a apresentação e discussão das novas complexidades;
- **Código final:** Ao final, uma forma de implementação do problema com a linguagem C++ será apresentada, com o objetivo de sanar possíveis dúvidas do leitor.

Entre cada tópico podem existir alguns momentos de reflexão com sugestões de tarefas que o leitor pode fazer para tentar exercitar os conteúdos. Estas serão apresentadas conforme o seguinte modelo:

<i>Sugestão de tarefa ao leitor...</i>
--

5 DESENVOLVIMENTO

Esta seção irá apresentar as técnicas selecionadas de otimização para programação dinâmica, de acordo com a metodologia proposta. Estas serão divididas em subseções, sendo as mesmas independentes entre si.

5.1 REDUÇÃO DE ESPAÇO

- **Problema:** São dados diversos itens, onde cada um possui um peso e valor associado. Deseja-se colocar alguns em uma mochila¹ com a finalidade de maximizar o valor dos que foram selecionados. Porém, a mochila possui uma capacidade máxima de peso. Além disso, nenhum item pode ser fracionado.
- **Solução ingênua:**

$$dp[i][j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0, \\ \max(valor[i-1] + dp[i-1][j - peso[i-1]], dp[i-1][j]) & \text{se } peso[i-1] \leq j, \\ dp[i-1][j] & \text{se } peso[i-1] > j \end{cases} \quad (3)$$

O problema da mochila pode ser resolvido através da relação de recorrência apresentada acima, onde $dp[i][j]$ representa o valor máximo que pode se conseguir ao colocar os i -ésimos primeiros itens em uma mochila de capacidade j . Os vetores *valor* e *peso*, representam o valor e peso associado a cada um dos n itens, respectivamente. A resposta para o problema estará em $dp[n][capacidade]$.

- **Análise de particularidades:** Analisando a complexidade da equação 3 é fácil ver que será necessário $O(n * capacidade)$, tanto de memória, quanto de tempo. Porém, pode-se perceber que para solucionar a linha i da matriz de programação dinâmica, só são necessárias as respostas que já foram calculadas na linha $i - 1$.
- **Redução de espaço:** Esta técnica visa solucionar problemas onde a quantidade de memória alocada não é sempre necessária, fazendo com que seja mantido em memória apenas o essencial. Para todos os problemas onde uma linha de uma tabela da programação dinâmica dependa de uma quantidade fixa de outras linhas, digamos K , é necessário apenas manter em memória K linhas.

No problema proposto, verificou-se que uma linha depende de apenas uma outra. Portanto, podemos trabalhar apenas com duas linhas consecutivas da matriz, sempre alternando

¹<http://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/>

entre linha par e ímpar. A equação a seguir demonstra como fazer essa alternância entre linhas:

$$dp[i\&1][j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0, \\ \max(valor[i-1] + dp[\sim i\&1][j - peso[i-1]], \\ dp[\sim i\&1][j]) & \text{se } peso[i-1] \leq j, \\ dp[\sim i\&1][j] & \text{se } peso[i-1] > j \end{cases} \quad (4)$$

Sugere-se ao leitor tentar utilizar essa técnica na resolução do Fibonacci, que foi explicado na seção 2.2.

- **Benefícios:** A equação 4 ilustra como reduzir a memória. Os valores que serão utilizados nas linhas da tabela de programação dinâmica serão sempre 0 ou 1, assim o total de memória necessária é de $2 * capacidade$, deixando com uma complexidade espacial de $O(capacidade)$. Uma maneira simples de alternar entre par e ímpar é utilizar operações binárias. Para verificar se um número é ímpar, basta fazer o *AND* entre o número desejado e o número 1, assim o valor retornado será 1 se for ímpar e 0 se for par. Agora para saber a paridade da linha anterior, basta inverter a paridade da linha atual, para isto pode-se usar o operador \sim , que é responsável por inverter todos os bits de um número. A resposta para o problema da mochila utilizando esta relação estará em $dp[n\&1][capacidade]$.
- **Código final:** O seguinte código mostra a implementação da solução para o problema da mochila com complexidade de memória linear.

Algoritmo 3 - Implementação Mochila

```

1
2  int mochila(){
3      int valor[] = {60, 100, 120};
4      int peso[] = {10, 20, 30};
5      int capacidade = 50, n = 3;
6
7      int dp[2][capacidade + 1];
8      for(int i = 0; i <= n; i++){
9          for(int j = 0; j <= capacidade; j++){
10             if(!i || !j)
11                 dp[i&1][j] = 0;
12             else if(peso[i - 1] <= j)
13                 dp[i&1][j] = max(valor[i-1] + dp[~i&1][j-peso[i-1]],
14                                     dp[~i&1][j]);
15             else
16                 dp[i&1][j] = dp[~i&1][j];

```

```

17         }
18     }
19     return dp[n+1][capacidade];
20 }

```

5.2 ESTRUTURA DE DADOS RMQ

- **Problema:** Um dos problemas clássicos de programação dinâmica é a LIS (do inglês, *Longest Increasing Subsequence*)². Neste, o objetivo é encontrar a maior subsequência de um *array* onde todos os elementos estão ordenados de forma crescente. Uma subsequência pode ser encontrada com a eliminação de alguns elementos do *array*.

Imaginemos o seguinte *array*:

2, 5, 3, 7, 11, 8

Uma das soluções para este conjunto é a subsequência 2, 3, 7, 8, que possui tamanho quatro, porém esta não é única, existem outras com o mesmo tamanho, como é o caso da subsequência 2, 5, 7, 11. Geralmente é esperado apenas o tamanho da maior LIS possível. Portanto, qual LIS será escolhida não terá muita relevância, pois só será necessário se preocupar com isso quando houver a necessidade da reconstrução da solução.

- **Solução ingênua:** Este problema pode ser facilmente resolvido com o auxílio de programação dinâmica utilizando a seguinte recorrência:

$$dp[i] = \begin{cases} 1 & \text{se } i = 0 \\ \max(dp[j] + 1)_{0 \leq j < i} & \text{se } i \neq 0 \text{ e } v[j] \leq v[i] \end{cases} \quad (5)$$

A equação 5, resolve o problema da LIS, com a complexidade $O(n^2)$ de tempo e $O(n)$ de memória. Nesta solução, $dp[i]$ representa qual a maior LIS que pode ser formada onde o último elemento é exatamente o índice i . Para calcular cada estado i é selecionado um estado j que já tenha sido calculado e possui maior valor. Além disso, o valor do elemento j tem que ser menor ou igual que o elemento i . Isto significa que a LIS que termina em i é uma junção da LIS terminada em j , adicionado o elemento i ao final dela. Após calcular todos os estados utilizando a recorrência, basta percorrer o *array* dp e obter seu maior valor.

- **Análise de particularidades:** É notório que quando está sendo calculada a $dp[i]$, ou seja, a maior LIS que termina no i -ésimo índice, estão sendo percorridos todos elementos do *array* que estão à esquerda e que possuem um valor menor ou igual que o elemento atual. Em outras

²<http://www.geeksforgeeks.org/longest-increasing-subsequence/>

palavras, queremos o maior elemento da dp cujo valor esteja no intervalo $[0..v[i]]$.

Sugere-se ao leitor pensar uma maneira mais eficiente de encontrar o maior elemento do array.

- **Estruturas de dados RMQ:** Existem algumas estruturas de dados que resolvem problemas parecidos com este, onde é dado um *array* e é necessário consultar o máximo ou mínimo elemento em um determinado intervalo. Essas operações normalmente são chamadas de RMQ (do inglês, *Range Minimum/Maximum Query*). Espera-se destas estruturas uma complexidade melhor que $O(n)$, que seria a forma mais simples de conseguir o máximo, bastando apenas iterar por todo o intervalo. É o caso da *SegmentTree*³ ou *BIT*⁴, que realizam estas operações em $O(\log n)$.
- **Benefícios:** Ao remover a busca linear para encontrar o maior elemento já calculado, adicionando uma das estruturas citadas, a complexidade da busca será reduzida de $O(n)$ para $O(\log n)$, deixando assim a LIS em uma complexidade final de $O(n * \log n)$.
- **Código final:** O código a seguir mostra uma maneira de implementar a solução para a LIS utilizando *BIT*.

Algoritmo 4 - Implementação LIS

```

1
2 #define MAXN 20
3 int v[] = {2, 5, 3, 7, 11, 8};
4 int bit[MAXN];
5 int dp[MAXN];
6
7 void update(int x, int v){
8     for(; x < MAXN; x+=x&-x)
9         bit[x] = max(bit[x], v);
10 }
11
12 int get(int x){
13     int ans = 0;
14     for(; x; x-=x&-x)
15         ans = max(ans, bit[x]);
16     return ans;
17 }
18
19 int lis(int n){

```

³<http://www.geeksforgeeks.org/segment-tree-set-1-range-minimum-query/>

⁴<http://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/>

```

20     int ans = 0;
21
22     for(int i = 0; i < n; i++){
23         dp[i] = 1+get(v[i]);
24         update(v[i], dp[i]);
25         ans = max(ans, dp[i]);
26     }
27
28     return ans;
29 }

```

5.3 DIVIDE AND CONQUER OPTIMIZATION

- **Problema:** São dados n objetos, cada um com um valor associado $x_1, x_2, x_3, \dots, x_n$, onde $x_i > 0 \forall i$ e é necessário dividi-los em k grupos consecutivos, de forma que esta divisão seja feita com o menor custo possível. O custo para criação de um grupo é igual a soma dos valores de todos elementos do grupo multiplicado pela quantidade de objetos.

Para exemplificar, supondo que são quatro objetos e seus valores são 1, 2, 3 e 4, é necessário dividi-los em dois grupos. Uma divisão possível seria colocar os objetos com valores 1, 2, 3 no primeiro grupo o que resultaria em um custo de 18, ou seja, a soma deles que é igual a 6, multiplicado pela quantidade de elementos, 3. Por fim, o elemento 4 ficaria no segundo grupo, tendo como custo o valor 4. Portanto para esta configuração o custo total é a soma dos dois grupos, o que gera um valor de $18 + 4 = 22$. Analisando um pouco melhor este exemplo é fácil notar que esta configuração não é a melhor possível. A solução ótima seria deixar os objetos com valores 1 e 2 no primeiro grupo e os demais elementos no segundo grupo, gerando um custo total de 20.

- **Solução ingênua:** Para resolver este problema, o primeiro passo é calcular uma matriz $C[i][j]$, que corresponde ao custo total para agrupar todos os elementos do índice i até o índice j . O tempo total para calcular a matriz poderá ser de ordem quadrática.

Fica como sugestão ao leitor implementar o cálculo da matriz C com tempo $O(n^2)$.

A equação 6 exemplifica uma maneira de implementar o problema, onde $dp[i][j]$ representa o menor custo para criar i grupos estando no j -ésimo objeto. A resposta estará em $dp[k][n]$, levando em consideração que o *array* está indexado a partir da posição 1. Para solucionar cada estado é necessário percorrer todos os elementos que são menores que j e verificar se criar um novo grupo naquela posição irá melhorar solução. Para tal, serão feitas em

média $n/2$ iterações, o que leva a uma complexidade final de $O(k * n^2)$.

$$dp[i][j] = \begin{cases} 0 & \text{se } i = 0 \\ C[i][j] & \text{se } i = 1 \\ \min_{l < j} (dp[i-1][l] + C[l+1][j]) & \text{se } i > 1 \end{cases} \quad (6)$$

- **Análise de particularidades:** Se for definida uma matriz $opt[i][j]$ que representa qual é o l ótimo para realizar a divisão da $dp[i][j]$, ou seja, o ponto ótimo que minimiza o valor daquele estado, é possível notar que para qualquer i e j , $opt[i][j] \leq opt[i][j+1]$. Portanto, j é monotônico⁵ para um i fixo. Isto é verdade pois ao inserir um novo objeto em um grupo o valor deste só poderá aumentar, devido a formulação da função de custo. Assim, um conjunto com $x+1$ elementos não poderá ter um ponto ótimo de divisão menor do que este mesmo conjunto com x elementos.
- **Divide and Conquer Optimization:** Devido a monotonicidade de opt , pode-se melhorar a complexidade da solução proposta, pois, como observado, para calcular o estado $(i, j+1)$, não é necessário testar valores de l que são menores do que $opt[i][j]$, apenas valores que são maiores ou iguais a $opt[i][j]$. Para a implementação desta técnica podemos recorrer à divisão e conquista⁶, onde pode-se criar uma função recursiva. Para um i fixo, é aplicada a divisão e conquista no j , sendo mantido o intervalo do j e o intervalo válido de l , que representa o ponto ótimo.
- **Benefícios:** Ao calcular o valor e o ponto ótimo x para um estado j , a divisão em conquista se responsabilizará por chamar a recursão para resolver o estado $j-1$ com o intervalo de l variando de $1..x$ e a recursão para o estado $j+1$ com o l entre $x..n$. Assim, é perceptível que o intervalo de busca do ponto ótimo que no início era $0..n$, foi dividido pela metade, e este é exatamente o pior caso, quando o ponto ótimo está exatamente no meio. Portanto, para calcular um estado j para um i fixo, a complexidade é $O(n * \log n)$, bem menor do que a solução ingênua que gastava $O(n^2)$ para cada i . Sendo assim, a complexidade final ao aplicar esta técnica neste problema fica $O(k * n * \log n)$.
- **Código final:** A seguir é apresentado um código que resolve este problema. Para obter a solução basta chamar a função `divideAndConquer()`, passando como parâmetro o total de elementos e a quantidade de grupos desejados.

Fica como sugestão ao leitor resolver o problema 2475 - Confecção de Presentes do site URI - Online Judge

⁵https://en.wikipedia.org/wiki/Monotonic_function

⁶https://pt.wikipedia.org/wiki/Divis%C3%A3o_e_conquista

 Algoritmo 5 - Implementação Divide and Conquer

```

1  #define MAXN 10
2  #define inf 99999999
3  int arr[MAXN] = {1, 2, 3, 4};
4  int C[MAXN][MAXN], dp[MAXN][MAXN];
5
6  void solve(int i, int jInicio, int jFim, int optInicio, int optFim){
7      if(jInicio > jFim) return;
8      int mid = (jInicio + jFim) / 2;
9      dp[i][mid] = inf;
10     int opt = -1;
11
12     for(int l = optInicio; l <= min(optFim, mid); l++){
13         if(dp[i-1][l] + C[l+1][mid] < dp[i][mid]){
14             dp[i][mid] = dp[i-1][l] + C[l+1][mid];
15             opt = l;
16         }
17     }
18     solve(i, jInicio, mid-1, optInicio, opt);
19     solve(i, mid+1, jFim, opt, optFim);
20 }
21
22 int divideAndConquer(int n, int k){
23     // Calculo da funcao custo  $O(n^2)$ 
24     for(int i = 0; i < n; i++){
25         int soma = 0;
26         for(int j = i; j < n; j++){
27             soma += arr[j];
28             C[i][j] = soma*(j-i+1);
29         }
30     }
31     // Casos base
32     for(int i = 0; i < n; i++)
33         dp[0][i] = 0;
34     for(int i = 0; i < n; i++)
35         dp[1][i] = C[0][i];
36
37     for(int i = 2; i <= k; i++)
38         solve(i, 0, n-1, 0, n-1);
39
40     return dp[k][n-1];
41 }

```

5.4 KNUTH OPTIMIZATION

- **Problema:** Uma *string* é dada e deseja-se realizar alguns cortes nela. A cada corte feito, ela será dividida em duas partes e haverá um custo associado para fazer tal operação, que é igual exatamente ao tamanho original antes de efetuar o corte. Imaginemos a *string* “knuthoptimization” que possui 17 caracteres e é preciso realizar os cortes nos índices 2, 6 e 8, levando em consideração que a *string* está indexada começando em 1. O valor total para realizar todos os cortes, está diretamente relacionado à ordem escolhida para efetuar as divisões. As tabelas 3 e 4 mostram os custos associados ao realizar os cortes de duas maneiras distintas. Na tabela 3 os cortes foram feitos da esquerda para a direita, enquanto a tabela 4 mostra como ficaria a resposta se os cortes fossem feitos da direita para a esquerda.

Tabela 3 - Problema Knuth realizando cortes da esquerda para a direita

Corte	String	Custo
-	knuthoptimization	0
2	kn uthoptimization	17
6	kn utho ptimization	15
8	kn utho pt imization	11
Total	-	43

Fonte: Próprio Autor.

Tabela 4 - Problema Knuth realizando cortes da direita para a esquerda

Corte	String	Custo
-	knuthoptimization	0
8	knuthopt imization	17
6	knutho pt imization	8
2	kn utho pt imization	6
Total	-	31

Fonte: Próprio Autor.

Ao analisar as tabelas, fica evidente que a ordem do corte fará com que o custo total aumente ou diminua. O problema consiste em determinar qual é o menor valor possível para realizar todas as divisões propostas.

Para melhor entendimento, é sugerido ao leitor encontrar o menor custo para o exemplo proposto.

- **Solução ingênua:** Para resolução deste problema, a primeira coisa a se notar é que o tamanho da *string* não é tão importante. O mais relevante é onde os cortes devem ser realizados. Assim, as complexidades das soluções estarão ligadas à quantidade k de cortes.

Sendo o *array* v_1, v_2, \dots, v_k representando as posições de cada divisão, onde $v_1 < v_2 < \dots < v_k$, pode ser pensado em uma solução com programação dinâmica que utilize dois ponteiros, i e j , que representam quais posições estão sendo analisadas no momento. Para solucionar cada estado basta escolher uma posição x onde $i < x < j$, que minimize o custo.

$$dp[i][j] = \begin{cases} 0 & \text{se } j - i \leq 1 \\ \min_{i < x < j} (dp[i][x] + dp[x][j] + v[j] - v[i]) & \text{se } j - i > 1 \end{cases} \quad (7)$$

A equação 7 apresenta uma possível solução para o problema, onde custo para realizar um corte está representado como sendo $v[j] - v[i]$, ou seja, o tamanho da *string* que está sendo considerada. Porém, através da definição do problema, é sabido que para realizar o primeiro corte o custo será igual ao tamanho original da *string*, mas esta solução não está tratando esse caso. Uma solução simples é inserir dois elementos extras no *array* de cortes, que irão representar o tamanho completo. Após essa modificação, este terá o formato $v_0, v_1, v_2, \dots, v_k, v_{k+1}$, onde v_0 tem o valor 0 e v_{k+1} possui o tamanho inteiro da *string*. Com estas modificações o algoritmo proposto irá funcionar e a resposta estará em $dp[0][k]$. Para solucionar cada estado (i, j) da matriz, é necessário iterar por todos elementos que estão neste intervalo para encontrar o melhor candidato à resposta. Por conta disso a complexidade final será $O(k^3)$.

- **Análise de particularidades:** Se for definida uma função de custo $C[i][j]$ que representa o valor para cortar um *string* estando nos índices i e j do *array* v , esta pode ser interpretada como sendo $C[i][j] = v[j] - v[i]$. É possível notar que o custo é monotônico, respeitando a equação 8.

$$C[b][c] \leq C[a][d], a \leq b \leq c \leq d \quad (8)$$

Essa equação implica que todos os subintervalos possuem um custo menor ou igual ao intervalo onde o mesmo está contido. Para melhor entendimento, pode-se imaginar o seguinte *array* de cortes, com os índices variando de 1 até 5: $v = \{1, 2, 5, 10, 15\}$. Se como intervalo maior forem selecionados os elementos nas posições 1 e 5, o custo será 14. Porém, ao selecionar duas posições x e y , onde $1 \leq x \leq y \leq 5$ é evidente que não será possível ter um custo maior que 14.

Sugere-se ao leitor avançar para a próxima parte da otimização apenas quando o mesmo tiver compreendido a condição de monotonicidade do problema.

- **Knuth Optimization:** Devido à monotonicidade da função de custo, é possível definir uma matriz $opt[i][j]$ que representa qual o menor x que gera a melhor solução para a $dp[i][j]$, ou seja, o ponto ótimo para realizar o corte. É possível notar que para qualquer par de i e j , onde $i \leq j$, a propriedade mostrada na equação 9 é mantida.

$$opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j] \quad (9)$$

Para melhor entendimento, a equação pode ser dividida em duas partes: na primeira, $opt[i][j-1] \leq opt[i][j]$, fica fácil ver que esta é verdade pois se fixarmos o i e aumentarmos o tamanho do intervalo em 1 para a direita, o ponto ótimo não poderá estar antes do intervalo menor. Isso se deve ao fato do custo ser monotônico. Assim, se fosse melhor realizar o corte na posição x para i e j , então para i e $j-1$ também seria ótimo escolher a posição x , ou então uma posição menor. O mesmo raciocínio vale para a segunda parte da equação, $opt[i][j] \leq opt[i+1][j]$, porém aqui o intervalo está sendo diminuído em 1. Baseado nessas características, agora é possível mudar a forma de calcular o valor de x da equação 7, fazendo com que ele itere apenas entre $opt[i][j-1]$ e $opt[i+1][j]$.

- **Benefícios:** O grande gargalo da solução proposta era o cálculo do ponto ótimo x de cada estado, que em média gastará $O(k)$. Ao utilizar as informações referentes ao opt , pode-se calcular a matriz dp em ordem crescente de $j-i$, ou seja, dos menores intervalos até os maiores. Além disso, o x não irá mais variar entre $i+1$ e $j-1$, mas sim entre $opt[i][j-1]$ e $opt[i+1][j]$. Portanto, agora para o cálculo de um estado de tamanho $y = j-i$, o tempo necessário é igual a $opt[y+1][2] - opt[y][1] + opt[y+2][3] - opt[y+1][2] + \dots + opt[k][k-y+1] - opt[k-1][k-y] = opt[k][k-d+1] - opt[k][1] = O(k)$. Sendo assim, o tempo total para resolver toda a matriz dp sai de $O(k^3)$ para $O(k^2)$.
- **Código final:** O algoritmo a seguir apresenta uma implementação da solução do problema. Para obter a resposta, basta preencher o array v com os cortes necessários. Estes deverão estar com os índices entre $1 \dots k$. Ao chamar a função `knuthOptimization()`, passando como parâmetro o tamanho da *string* e o total de cortes, será devolvido o custo mínimo.

Fica como sugestão ao leitor resolver o problema BRKSTRNG - Breaking String do site SPOJ.com

 Algoritmo 6 - Implementação Knuth Optimization

```

1  #define MAXN 1010
2  #define inf 99999999
3
4  int v[MAXN];
5  int dp[MAXN][MAXN];
6  int opt[MAXN][MAXN];
7
8  int knuthOptimization(int n, int k){
9      // Adicionado os elementos extras
10     v[0] = 0;
11     v[++k] = n;
12     k++;
13
14     for(int tam = 1; tam <= k; tam++){
15         for(int i = 0; i+tam <= k; i++){
16             int j = i+tam-1;
17             if(tam <= 2){
18                 dp[i][j] = 0;
19                 opt[i][j] = i;
20                 continue;
21             }
22             dp[i][j] = inf;
23             for(int x = opt[i][j-1]; x <= opt[i+1][j]; x++){
24                 if(dp[i][x] + dp[x][j] + v[j]-v[i] < dp[i][j]){
25                     dp[i][j] = dp[i][x] + dp[x][j] + v[j]-v[i];
26                     opt[i][j] = x;
27                 }
28             }
29         }
30     }
31     return dp[0][k-1];
32 }

```

5.5 CONVEX HULL TRICK

- **Problema:** Uma estrada está sendo construída e é necessário cobrir n pontos da mesma. O custo para se cobrir um pedaço da estrada que vai do ponto x até o ponto y , é definido como $(x - y)^2 + c$, sendo c uma constante qualquer. Para exemplificar, vamos supor o array $v = \{1, 3, 8, 10\}$, que representa os pontos que precisam de cobertura, além disso, o valor da constante c é 5. Portanto, uma solução possível seria realizar a cobertura do ponto 1 até o 8, com o custo de $(1 - 8)^2 + 5 = 54$ e uma cobertura única no ponto 10, totalizando $(10 - 10)^2 + 5 = 5$. Para tal configuração, o valor total para cobrir os 4 pontos seria de 59. O objetivo deste problema é encontrar o custo mínimo para realizar a cobertura.

Para melhor entendimento, é sugerido ao leitor encontrar a melhor solução para o exemplo proposto.

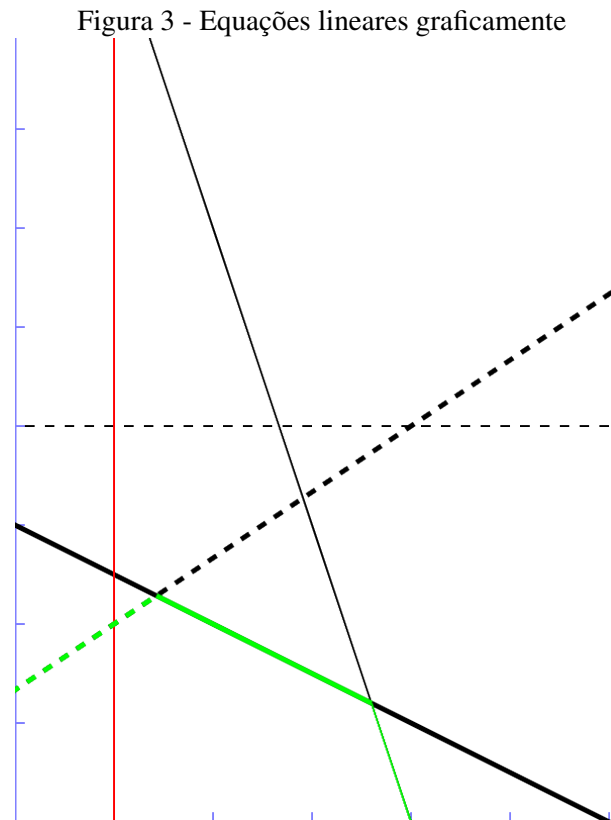
- **Solução ingênua:** Assumindo que os elementos no array v , estão indexado em 1, a equação 10 mostra uma possível solução.

$$dp[i] = \begin{cases} 0 & \text{se } i = 0 \\ \min_{1 \leq j \leq i} (dp[j - 1] + (v[i] - v[j])^2 + c) & \text{se } i \neq 0 \end{cases} \quad (10)$$

Para o cálculo de cada estado i da tabela, é necessário percorrer todos os índices que estão a sua esquerda, com a finalidade de escolher algum ponto onde seria colocado o início da cobertura. Devido a isto, a complexidade de cada estado seria $O(n)$ e a complexidade para o cálculo de todos os n estados seria $O(n^2)$. A solução para o problema estará em $dp[n]$.

- **Análise de particularidades:** Através do código ingênuo elaborado, pode-se notar que um dos motivos da solução não ficar tão rápida, é o fato de que, para resolver cada estado é necessário percorrer todos os outros que possuem um índice menor. Porém, ao expandir a equação de custo de cada estado, algumas propriedades importantes podem ser notadas. Sua expansão tem a seguinte forma: $dp[j - 1] + (v[i] - v[j])^2 + c = -2 * v[j] * v[i] + v[j]^2 + dp[j - 1] + v[i]^2 + c$. Para melhor clareza será substituído $v[i]$ por x e $v[j]$ por y , assim resultando em: $-2 * y * x + y^2 + dp[j - 1] + x^2 + c$. Além disso, se for definido que $a = -2 * y$ e $b = y^2 + dp[j - 1]$, é possível verificar que para o cálculo de um estado i , a iteração que é necessária para encontrar o mínimo, está tentando exatamente minimizar a função $f(x) = ax + b$, que pode ser vista graficamente como uma reta, pois os valores de x são apenas inteiros positivos. A $f(x)$ não contempla os valores de x^2 e c , pois estes serão somados para qualquer reta selecionada. Logo, não há necessidade de tentar minimizá-los.
- **Convex Hull Trick:** Imaginemos que diversas equações lineares sejam dadas, sendo no formato

$f(x) = ax + b$, e diversas consultas são efetuadas, onde para cada uma delas é necessário informar qual o menor valor que uma $f(x)$ possui para um determinado x se este for avaliado para todas as equações⁷. Como exemplo imaginemos quatro funções, $f(x) = 4$, $f(x) = \frac{4}{3} + \frac{2}{3}x$, $f(x) = 12 - 3x$ e $f(x) = 3 - \frac{1}{2}x$ e é desejável descobrir qual destas minimiza o valor de $x = 1$. A figura 3 mostra graficamente a distribuição destas equações que podem ser vistas como restas no plano.



Fonte: PEGWIKI, 2016

Para consultar o valor de $x = 1$, que está representado no gráfico acima através da linha vermelha, é possível ver que a reta que gera o menor valor possível é a que está pontilhada com traços grossos, sendo esta a $f(x) = \frac{4}{3} + \frac{2}{3}x$. Analisando esta figura, algumas características importantes podem ser observadas, como é o caso da reta $f(x) = 4$, que claramente nunca será escolhida, independente do valor de x , pois sempre haverá alguma outra que possui um valor menor. Dentre as outras três retas, é possível ver que elas são importantes, pois para algum valor de x elas possuem o menor valor de $f(x)$. Isso pode ser visto como a parte em verde pintada nas retas, a qual mostra o intervalo de valores de x onde cada reta é a melhor escolha. Com a informação de até que ponto uma determinada reta é ótima, elas podem ser ordenadas e cada consulta pode ser resolvida através de uma busca binária⁸, onde para cada momento um valor é

⁷https://wcipeg.com/wiki/Convex_hull_trick

⁸<http://www.geeksforgeeks.org/binary-search/>

testado e decidido se a reta ótima para o x em questão está a direita ou a esquerda desta. Assim, cada consulta teria a complexidade $O(\log n)$, sendo n o total de retas.

Uma forma de resolver seria adicionar todas as retas em alguma estrutura de dados, ordená-las pela angulação e depois realizar as consultas. Porém, nem sempre as retas estão disponíveis de antemão. Logo, se utilizarmos a ideia de sempre que adicionar uma nova reta for necessário ordenar todas elas, o algoritmo ficaria com a complexidade $O(n^2 * \log n)$.

Sugere-se ao leitor pensar em uma maneira mais eficaz de adicionar uma nova reta no conjunto.

Partindo do princípio que o conjunto de retas está devidamente ordenado e é necessário inserir mais uma reta com angulação maior que as já presentes, ao realizar essa operação alguns casos podem acontecer:

- A nova reta não é melhor que nenhuma outra que já estava presente, sendo assim, não há necessidade de incluir a mesma no conjunto;
- A nova reta é melhor do que algumas outras a partir de algum valor de x ;
- A nova reta é melhor que alguma reta em todos os pontos onde esta era ótima, portanto, não é necessário mais manter aquela reta.

Com esses casos, um algoritmo mais eficiente pode ser pensado, onde será mantida uma pilha⁹, com todas as retas que ainda são ótimas em algum ponto, contendo em seu topo a última que foi adicionada. Quando for inserida uma nova reta, basta verificar se a que estava no topo ainda será útil. Caso seja, a nova reta é inserida, mas se não for, pode-se realizar um *pop()* da pilha e ir repetindo este passo até que a pilha esteja vazia, ou que uma reta que ainda é útil for encontrada.

Para determinar o momento em que uma reta não é mais relevante e precisa ser removida da pilha, basta verificar se o ponto de intersecção da reta que está sendo inserida no momento com a penúltima reta está a esquerda do ponto de intersecção entre a penúltima e a última reta incluída. Caso isto ocorra, a última reta pode ser removida, pois a partir de agora ela se torna irrelevante. Com essas propriedades é notório que cada reta pode ser adicionada ou removida apenas uma vez, deixando assim a complexidade total $O(n)$ para incluir todas elas.

Esta técnica possui algumas variações e pode-se destacar três delas:

Convex Hull Trick 1: Nesta versão, as consultas realizadas são feitas de maneira crescente, ou seja, $x_1 \leq x_2 \leq \dots \leq x_n$, portanto, pode ser mantido um ponteiro p_i , que informa em qual reta estava a resposta para a consulta x_i . Assim, quando for necessário descobrir a resposta para a consulta x_{i+1} , só é necessário verificar as retas a partir do $p_i + 1$. Pois, no momento que

⁹<http://www.geeksforgeeks.org/stack-data-structure/>

para alguma consulta uma reta não for a melhor, é impossível ela se tornar ótima em algum outro momento. Além disso, todas as retas serão inseridas de maneira ordenada. Isso faz com que todas as consultas sejam respondidas em $O(q + n)$, onde q é o número de consultas e n é o total de retas. Isso se deve ao fato de que cada reta será avaliada apenas uma vez.

Convex Hull Trick 2: Esta variação é similar à primeira, porém, as consultas não estão ordenadas. Assim, se faz necessário o uso da busca binária, deixando a complexidade em $O(q * \log n)$.

Convex Hull Trick 3: Nesta versão não há garantias que as retas serão inseridas de forma ordenada. Assim, a solução proposta utilizando pilha não funciona mais, sendo necessário utilizar outra estrutura de dados. Com o intuito de deixar a complexidade o mais baixa possível, pode ser utilizada uma árvore balanceada para manter as retas ordenadas.

Fica como sugestão ao leitor pensar em uma maneira de resolver esta variação.

- **Benefícios:** Como visto na etapa de análise de particularidades, as funções de custo possuem a forma $f(x) = ax + b$, logo podem ser descritas como retas, permitindo a aplicação desta técnica. Como pode ser notado, para resolver o estado i , só as retas menores que i são importantes. Assim, pode ser inserida a reta relacionada a este estado no momento em que ele for resolvido. Devido a solução ingênua elaborada e a ordenação do *array* v , é possível ver que as retas inseridas e as consultas realizadas serão feitas de maneira ordenada. Portanto, pode ser utilizada a variação 1 do *Convex Hull Trick*. Ao aplicar a otimização, a complexidade será reduzida para $O(n + n) = O(n)$.
- **Código final:** O algoritmo a seguir apresenta a implementação da solução para o problema. Para obter a resposta basta preencher o *array* v com as posições que necessitam de cobertura. Este deverá estar com índices entre $1 \dots n$. Ao chamar a função $cht()$, passando como parâmetro a quantidade de elementos e a constante, será devolvido o custo mínimo para realizar a cobertura de todos os pontos.

Fica como sugestão ao leitor resolver o problema Covered Walkway do site Kattis.com

 Algoritmo 7 - Implementação Convex Hull Trick 1

```

1  #define MAXN 10
2  #define inf 99999999
3  int v[MAXN] = {0, 1, 3, 8, 10};
4  int dp[MAXN];
5  int A[MAXN], B[MAXN];
6  int tam, ponteiro;
7
8  int quadrado(int x){ return x*x; }
9
10 void adicionarReta(int a, int b){
11     while(tam >= 2 &&
12           (B[tam-2] - B[tam-1]) * (a-A[tam-1]) >=
13           (B[tam-1]-b) * (A[tam-1]-A[tam-2]))
14         tam--;
15     A[tam] = a;
16     B[tam] = b;
17     tam++;
18 }
19
20 int consultar(int x){
21     ponteiro = min(ponteiro, tam-1);
22     while(ponteiro+1 < tam &&
23           A[ponteiro+1]*x+B[ponteiro+1] <= A[ponteiro]*x + B[ponteiro])
24         ponteiro++;
25     return A[ponteiro]*x + B[ponteiro];
26 }
27
28 int cht(int n, int c){
29     // Inicializando a pilha do Convex Hull Trick
30     ponteiro = tam = 0;
31
32     // Caso base
33     dp[0] = 0;
34     adicionarReta(-2*v[1], quadrado(v[1]));
35     for(int i = 1; i <= n; i++){
36         dp[i] = consultar(v[i]) + quadrado(v[i]) + c;
37         if(i < n)
38             adicionarReta(-2*v[i+1], quadrado(v[i+1])+dp[i]);
39     }
40     return dp[n];
41 }

```

6 CONSIDERAÇÕES FINAIS

Através da metodologia proposta, este trabalho documentou e apresentou, de forma didática, técnicas avançadas de programação dinâmica. É esperado que este tenha servido como inspiração e como uma base para quem deseja se aprofundar nos estudos dessa grande área. O objetivo principal era a construção de um material didático completo onde um leitor que não possua um conhecimento sobre otimizações consiga apenas com este material entender os temas propostos e aplicar os conceitos em diversos problemas.

Apesar do trabalho ter um conteúdo mais complexo, com assuntos que normalmente apenas pessoas mais experientes conseguem absorver, acredita-se que o objetivo principal do trabalho foi atingido, tornando esses conteúdos mais acessíveis. Por mais que as técnicas propostas sejam densas, no decorrer de todo o texto as explicações foram feitas da forma mais simplificada e detalhada possível, com intuito de sanar grande parte das possíveis dúvidas que o leitor possa ter.

Espera-se que este trabalho seja relevante e útil para todos que venham a ler, principalmente aos competidores de maratonas de programação, onde esses temas são comumente abordados.

REFERÊNCIAS

- CORMEN, T. H. *et al.* **Introduction to Algorithms**. Favoritenstrasse 9/4th Floor/1863: The MIT Press, 2009.
- HOM, E. J. **What is the Fibonacci Sequence?** 2013. Disponível em: <<http://www.livescience.com/37470-fibonacci-sequence.html>>. Acesso em: 02 abr. 2017.
- PAPP-VARGA, Z.; SZLÁVI, P.; ZSAKÓ, L. Ict teaching methods – programming languages. In: **Annales Mathematicae et Informaticae** 35. [s.n.], 2008. p. 163–172. Disponível em: <<https://www.researchgate.net/publication/228955901>>.
- PEARS, A. *et al.* A survey of literature on the teaching of introductory programming. In: **Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education**. New York, NY, USA: ACM, 2007. (ITiCSE-WGR '07), p. 204–223. Disponível em: <<http://doi.acm.org/10.1145/1345443.1345441>>.
- PEGWIKI. **Convex hull trick - PEGWiki**. 2016. Disponível em: <https://wcipeg.com/wiki/Convex_hull_trick>. Acesso em: 09 out. 2017.
- PERRETT, D. **CompSci 101 - Big-O Notation**. 2010. Disponível em: <<http://www.daveperrett.com/articles/2010/12/07/comp-sci-101-big-o-notation/>>. Acesso em: 23 abr. 2017.
- PIEKARSKI, A. E. T. *et al.* A metodologia das maratonas de programação em um projeto de extensão: um relato de experiência. In: **CBIE & LACLO 2015 - IV Congresso Brasileiro de Informática na Educação e X Conferência Latino-Americana de Objetos e Tecnologias de Aprendizagem**. [s.n.], 2015. p. 1246–1254. Disponível em: <<http://www.br-ie.org/pub/index.php/wcbie/article/view/6276>>.
- RADOŠEVIĆ, D.; OREHOVAČKI, T.; LOVRENČIĆ, A. New approaches and tools in teaching programming. 2009. Disponível em: <<https://www.researchgate.net/publication/224930648>>.
- ROBINS, A.; ROUNTREE, J.; ROUNTREE, N. Learning and teaching programming: A review and discussion. **Computer Science Education**, v. 13, n. 2, p. 137–172, 2003. Disponível em: <<http://www.tandfonline.com/doi/abs/10.1076/csed.13.2.137.14200>>.
- SCHWARTZ, H. R. **Memoization using Closures**. 2011. Disponível em: <<https://harryrschwartz.com/2011/01/06/memoization-using-closures.html>>. Acesso em: 02 abr. 2017.
- SZLÁVI, P.; ZSAKÓ, L. Methods of teaching programming. 2003. Disponível em: <<https://www.researchgate.net/publication/235925815>>.
- VIHAVAINEN, A.; PAKSULA, M.; LUUKKAINEN, M. Extreme apprenticeship method in teaching programming for beginners. In: **Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2011. (SIGCSE '11), p. 93–98. ISBN 978-1-4503-0500-6. Disponível em: <<http://doi.acm.org/10.1145/1953163.1953196>>.

ZSAKÓ, L.; NÓRA, C. Ict teaching methods applications. In: **Mittermeir R., Syslo M. (eds.) Informatics Education contributing across the curriculum, proc. of selected papers - 3rd ISSEP conference.** [s.n.], 2008. p. 47–53. Disponível em: <<https://www.researchgate.net/publication/235925718>>.