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:

_____

Nome do(a) Professor(a) Orientador(a)
(Presidente de Banca) por extenso - titulação abreviada

_____

Nome do(a) Convidado(a) por extenso - titulação abreviada

_____

Nome do(a) Convidado(a) por extenso - titulação abreviada

Teresópolis

dia de mês da defesa de 2017

Dedico esta monografia a minha família,
pelo apoio fornecido e aos meus amigos.

AGRADECIMENTOS

Texto dos agradecimentos.

Epígrafe

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

LISTA DE SIGLAS

ICT	Information and Communication Technology
IDE	Integrated Development Environment
LIS	Longest Increasing Subsequence
RMQ	Range Minimum Query

LISTA DE ALGORITMOS

RESUMO

Escrever um texto que contemple todo o conteúdo do trabalho, com espaçamento 1,5, justificado. Conforme as normas NBR 14724:2011 e NBR 6028:2003,da ABNT, o resumo é elemento obrigatório, constituído de parágrafo único; uma seqüência de frases concisas e objetivas e não de uma simples enumeração de tópicos, não ultrapassando 500 palavras, O resumo deve ressaltar o objetivo, o método, os resultados e as conclusões do documento. Deve-se usar o verbo na voz ativa e na terceira pessoa do singular. Devem ser seguido, logo abaixo, das palavras representativas do conteúdo do trabalho, isto é, palavras-chave e/ou descritores, que são palavras principais do texto, sendo de 3 a 5, separadas por ponto)

Palavras-chave: Palavra-chave 1. Palavra-chave 2. ...

ABSTRACT

Abstract text (maximum of 500 words).

Keywords: Keyword 1. Keyword 2. ...

SUMÁRIO

1 INTRODUÇÃO

Motivação: - Uma das áreas mais importantes da maratona de programação - Está em alta nas últimas competições

Justificativa: - Poucos trabalhos na área - Livros não abordam esse assunto

Objetivos - Gerar um trabalho que explique as principais técnicas de otimização utilizadas

Alterações necessárias no modelo LaTeX

- Adicionar uma lista de equações
- Tentar adicionar links no sumário
- Quadro != Tabela

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 demostrados 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, no capítulo 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, 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).

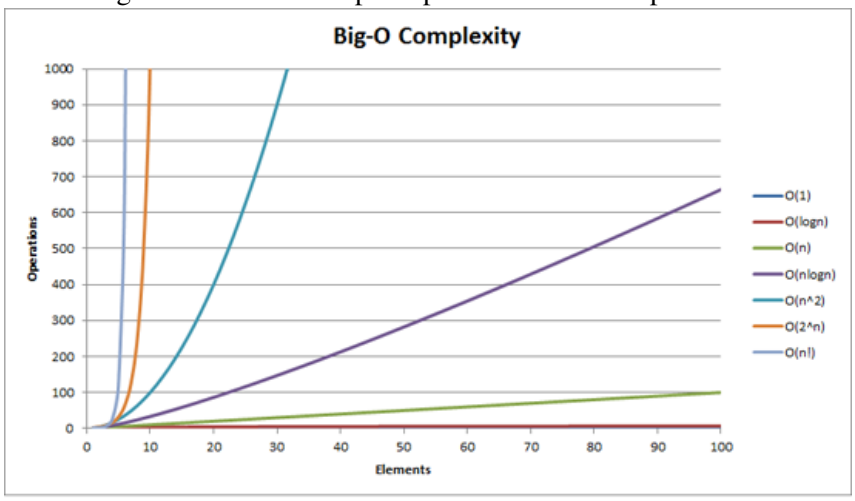
Quando é feita a análise de complexidade, é possível identificar qual classe um determinado algoritmo pertence. A tabela 1 lista algumas classes de forma ordenada, sendo a primeira função a melhor possível e a última a pior. Após classificar um algoritmo, é possível escolher entre diversas soluções para um mesmo problema, qual é a mais adequada no momento e poder saber antes de executar quanto tempo e memória o algoritmo irá gastar quando o tamanho da entrada for n, onde n corresponde, geralmente a 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
$O(logn)$	Busca binária
$O(\sqrt{n})$	Determinar se um número é primo
$O(n)$	Procurar um elemento em um *array* não ordenado
$O(n*logn)$	*Merge sort*[1]
$O(n^2)$	*Bubble sort*[2]
$O(n^3)$	*Floyd-Warshall*[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, da mesma maneira que a divisão e conquista, que divide o problema em subproblemas, resolve cada

[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/

um recursivamente e depois é feita 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 que se deparar com algo que já foi resolvido ela simplesmente retorna o valor que já estava armazenado.

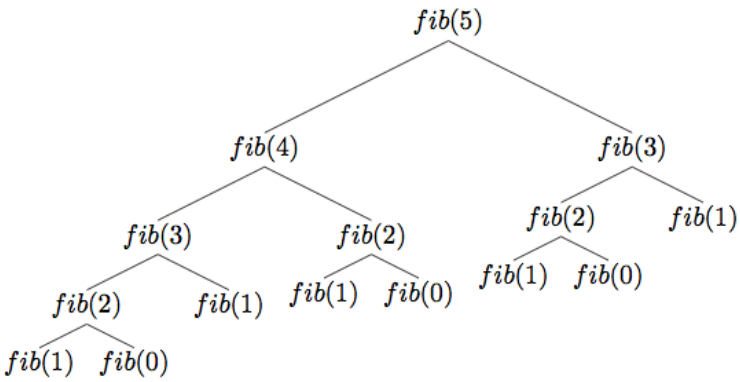
A sequência de Fibonacci é um exemplo de fácil entendimento de quando é necessário a utilização de programação dinâmica. Esta é uma sequência de números inteiros que tem seu início com 0 e 1, os termos subsequentes são uma soma dos dois últimos números. A sequência recebeu o nome do matemático Leonardo 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:

$$0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,... \tag{1}$$

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

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

Figura 2 - Árvore de recursão do Fibonacci de 5



Fonte: SCHWARTZ, 2011

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 divido 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.

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 a solução, como é o caso da sequência de Fibonacci, discutida na seção 2.2, onde foi possível sair de uma complexidade exponencial para

[5]https://www.topcoder.com/community/data-science/data-science-tutorials/

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 formas otimizações existentes, que possibilitam tanto na 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 *array*
Divide and Conquer	Realiza divisão e conquista para encontrar o ponto ótimo necessário para se resolver o estado atual, reduzindo a complexidade temporal
Knuth Optimization	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
Convex Hull Trick	Através de conceitos geométricos, essa técnica possibilita a redução da complexidade temporal

Fonte: Próprio Autor.

2.3 ENSINO DE ALGORITMOS

O ensino de algoritmos, por se tratar de assuntos que normalmente não são tão simples, com um fundamentação matemática extensa, não é uma tarefa fácil, portanto foi feita uma busca na literatura de trabalhos que têm por objetivo criar uma metodologia de ensino. Assim servindo de modelo para o que está aqui sendo desenvolvido.

Szlávi e Zsakó (2003) apresentaram diversas metodologias relacionadas ao ensino de programação. O autor discuti sobre alguns modelos e exemplifica 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 aqui tem o foco nas estruturas de dados, assim, muitas vezes conseguindo evitar os grandes teoremas que normalmente são complexos.
- **Orientado a especificação:** A especificação da forma que o programa deve funcionar é a parte mais significativa, todos os algoritmos são gerados sistematicamente através das 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 do outros, neste 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 a linguagem:** Método semelhante ao orientado a problemas, porém o ensino dependente totalmente de uma linguagem de programação, 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 definido uma linguagem geral, uma forma de pseudocódigo.
- **Orientado a matemática:** Os problemas a serem resolvido com esta técnica, são retirados da matemática, onde cada problema individual 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 a forma de raciocínio que as duas áreas possuem.

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 realizado 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, é mostrado por Robins, Rountree (2003).

Pears *et al.* (2007) desenvolveu um *survey* que reúne algumas formas da literatura de ensinar a introdução de 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 nos 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 a 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 demostram os passos ideais para transmitir os conceitos da linguagem proposta.

Radošević, Orehovački e Lovrenčić (2009), se propõem em criar 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. Além disso, oferecer uma forma mais simples do professor auxiliar seus alunos.

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 fala todos os seus pensamentos e explica o motivo de ter feito determinada decisão, deixando os alunos a acompanharem toda a 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 façam 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 auxilio de um instrutor. Ao final do trabalho é feito um comparativo entre uma turma que utilizou da 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 discutido 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 são úteis para a elaboração da metologia 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 apresentado 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, seu foco principal são os 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 as outras.

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

4 METODOLOGIA

Após a análise dos trabalhos levantados no seção 2.3, pode-se destacar os trabalhos elaborados por Szlávi e Zsakó (2003) e Vihavainen, Paksula e Luukkainen (2011). O primeiro em um dos métodos discutidos é abordado uma forma mais metodológica na hora de transmitir uma informação, onde em cada etapa do ensino há uma clara divisão. No segundo trabalho é mostrado uma maneira 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 numa conclusão.

A metodologia aplicada no próximo capítulo tomará como base esses dois trabalhos, levando uma linha bem divida entre os tópicos, porém sempre fazendo o leitor entender toda a linha de raciocínio, com algumas pausas a fim de deixar quem está lendo pensar em uma solução para um problema, antes de ser mostrado 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 ser resolvido através de programação dinâmica;
- **Solução ingênua:** A partir da descrição do problema será elaborada uma solução que utilize programação dinâmica , juntamente com a análise de complexidade, podendo ser temporal e/ou de espaço;
- **Análise de particularidades:** Após uma solução ter sido desenvolvida será levantado as características do problema, sendo elas a forma que os cálculos estão sendo desenvolvidos, como está sendo armazenado as informações, para que com estes dados possa ser feita alguma otimização. Neste ponto o leitor já começa a imaginar o que pode ser realizado para melhorar a solução;
- **"Nome da técnica":** Com todas as análises feitas, será formulado a técnica em questão, mostrando o seu funcionamento e como esta pode ser aplicada no problema;
- **Benefícios:** Tendo aplicado a otimização, uma análise dos benefícios será feita juntamente com as 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 pode existir algumas paradas com sugestões de tarefas que o leitor pode fazer, para tentar exercitar os conteúdos, estas serão apresentadas conforme o seguinte modelo:

Sugestão de tarefa ao leitor...

5 DESENVOLVIMENTO (ALTERAR NOME)

Nesta seção serão apresentadas algumas técnicas de otimização para programação dinâmica, estas serão dividas em subseções, sendo cada uma delas independentes entre si.

5.1 REDUÇÃO DE ESPAÇO

- **Problema:** São dados diversos itens e cada um possui um peso e valor associado, deseja-se colocar alguns em uma mochila[1] 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 dividido.

- **Solução ingênua:**

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

  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 1 é fácil ver que será necessário $O(n * capacidade)$, tanto de memória, quanto de tempo. Porém, é notório 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.

  Voltando ao problema proposto foi analisado que uma linha depende de apenas uma outra, portanto podemos trabalhar apenas com duas linhas consecutivas da matriz, sempre alternando entre linha par e ímpar. A equação abaixo demonstra como fazer essa

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

alternância entre linhas:

$$dp[i\&1][j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0, \\ max(valor[i-1] + dp[\sim i\&1][j - peso[i-1]], & \\ dp[\sim i\&1][j]) & \text{se } peso[i-1] \leq j, \\ dp[\sim i\&1][j] & \text{se } peso[i-1] > j \end{cases} \quad (2)$$

Sugere-se ao leitor tentar utilizar essa técnica na resolução do Fibonacci, que foi explicado no capítulo 5.

- **Benefícios:** A equação 2 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 de 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 ~, 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 do problema da mochila com memória linear.

Algoritmo 1 - 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 o 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 elemento do *array*.
  Imaginemos o seguinte *array*:
  $$2,5,3,7,11,8$$
  Uma das solução 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, 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 auxilio de programação dinâmica utilizando a seguinte recorrência:
  $$dp[i] = \begin{cases} 1 & \text{se } i = 0 \\ max(dp[j]+1)_{0 \leq j < i} & \text{se } i \neq 0 \text{ e } v[j] \leq v[i] \end{cases} \quad (3)$$
  A equação 3, resolve o problema do 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 pegar o maior valor dele.
- **Análise de particularidades:** É notório que quando está sendo calculado a $dp[i]$, ou seja, a maior LIS que termina no i-ésimo índice, está sendo percorrido todos elementos do *array* que estão a esquerda e que possuem um valor menor ou igual que o elemento atual. Em outras palavras, queremos o maior elemento da dp que seu valor está no intervalo $[0..v[i]]$.

[2] http://www.geeksforgeeks.org/longest-increasing-subsequence/

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, estas operações normalmente são chamadas de RMQ (do inglês, *Range Minimum 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], ambas realizam estas operações em $O(logn)$.
- **Benefícios:** Ao remover a busca linear para encontrar o maior elemento já calculado e adicionando uma das estruturas citadas, a complexidade da busca será reduzida de $O(n)$ para $O(logn)$, deixando assim a LIS em uma complexidade final de $O(n*logn)$.
- **Código final:** O código a seguir mostra uma maneira de implementar o LIS utilizando *BIT*.

Algoritmo 2 - 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){
20     int ans = 0;
21
```

[3]http://www.geeksforgeeks.org/segment-tree-set-1-range-minimum-query/
[4]http://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/

```
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 $x_1, x_2, x_3, ..., x_n$, onde $x_i > 0_{\forall i}$ e é necessário dividi-los em k grupos consecutivos, porém é necessário 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. Supondo que são quatro objetos e seus valores são 1, 2, 3, 4 e é necessário dividir estes 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 de todos os 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 pode ser 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 4 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 será feito em média $n/2$ iterações o que leva em uma complexidade final de $O(k * n^2)$.

$$dp[i][j] = \begin{cases} 0 & \text{se } i = 0 \\ C[i][j] & \text{se } i = 1 \\ min_{l<j}(dp[i-1][l] + C[l+1][j]) & \text{se } i > 1 \end{cases} \quad (4)$$

- **Análise de particularidades:** Se for definido 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 monotônico[5] 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 $opt[i][j]$, mas sim só é necessário os valores que são maiores ou iguais a $opt[i][j]$. Para a implementação desta técnica podemos recorrer a divisão e conquista[6], onde pode-se criar uma função recursiva que para um i fixo, é aplicado 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 ótimo está exatamente no meio. Portanto para calcular um estado j para um i fixo, a complexidade é $O(n * logn)$, 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 $O(k * n * logn)$.
- **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

Algoritmo 3 - Implementação Divide and Conquer

[5]https://en.wikipedia.org/wiki/Monotonic_function
[6]https://pt.wikipedia.org/wiki/Divisão_e_conquista

```
#define MAXN 10
#define inf 99999999

int arr[MAXN] = {1, 2, 3, 4};
int C[MAXN][MAXN], dp[MAXN][MAXN];

void solve(int i, int jInicio, int jFim, int optInicio, int optFim){
    if(jInicio > jFim)
    return;

    int mid = (jInicio + jFim) / 2;
    dp[i][mid] = inf;
    int opt = -1;

    for(int l = optInicio; l <= min(optFim, mid); l++){
        if(dp[i-1][l] + C[l+1][mid] < dp[i][mid]){
            dp[i][mid] = dp[i-1][l] + C[l+1][mid];
            opt = l;
        }
    }

    solve(i, jInicio, mid-1, optInicio, opt);
    solve(i, mid+1, jFim, opt, optFim);
}

int divideAndConquer(int n, int k){
    // Calculo da funcao custo O(n^2)
    for(int i = 0; i < n; i++){
        int soma = 0;
        for(int j = i; j < n; j++){
            soma += arr[j];
            C[i][j] = soma*(j-i+1);
        }
    }

    // Casos base
    for(int i = 0; i < n; i++)
        dp[0][i] = 0;
    for(int i = 0; i < n; i++)
        dp[1][i] = C[0][i];

    for(int i = 2; i <= k; i++)
        solve(i, 0, n-1, 0, n-1);
```

```
44
45      return dp[k][n-1];
46  }
```

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 terá 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, estão diretamente relacionados a 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 foi escolhido fazer os cortes indo da esquerda para direita, enquanto a tabela 4 mostra como ficaria a resposta se fosse feito da direita para esquerda.

Tabela 3 - Problema Knuth realizando cortes da esquerda para 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 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, portanto esse problema deseja saber qual é o menor valor possível para realizar todas as divisões propostas.

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

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

  Sendo o *array* $v_1, v_2, ..., v_k$ representando as posições de cada divisão, onde $v_1 < v_2 < ... < v_k$, 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, e para solucionar cada estado basta escolher uma posição x onde $i < x < j$ que minimize o custo.

$$dp[i][j] = \begin{cases} 0 & \text{se } j - i \leq 1 \\ min_{i<x<j}(dp[i][x] + dp[x][j] + v[j] - v[i]) \end{cases} \tag{5}$$

  A equação 5 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 visto que para realizar o primeiro corte o custo será igual ao tamanho original da *string*, mas esta solução não está tratando isso. Para resolver este detalhe, uma maneira simples é inserir dois elementos extras no *array* de cortes, estes irão representar o tamanho completo. Após a modificação este ficará como sendo $v_0, v_1, v_2, ..., v_k, v_{k+1}$, 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 a resposta, por conta disso a complexidade final fica $O(k^3)$.

- **Análise de particularidades:** Se for definido 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 6.

$$C[b][c] \leq C[a][d], a \leq b \leq c \leq d \tag{6}$$

  Essa equação implica que todos os subintervalos possuem um custo menor ou igual ao

intervalo que ele está contido. Para melhor entendimento vamos imaginar o seguinte *array* de cortes, com os índices variando de 1 até 5: $v = \{1,2,5,10,15\}$, se como intervalo maior for selecionado o elementos nas posição 1 e 5, o custo será 14, agora ao pegar 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 tiver entendido a condição de monotonicidade do problema.

- **Knuth Optimization:** Devido a monotonicidade da função 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 qualquer par de i e j, onde $i \leq j$ a propriedade mostrada na equação 7 é mantida.

$$opt[i][j-1] \leq opt[i][j] \leq opt[i+1][j] \tag{7}$$

  Para melhor entendimento pode ser divida a equação 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, isto 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 5, 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 $opt[y+1][2] - opt[y][1] + opt[y+2][3] - opt[y+1][2] + ... + opt[k][k-y+1] - opt[k-1][k-y] = opt[k][k-d+1] - opt[k][1] = O(k)$. Sendo assim, o tempo total para resolver toda a matriz dp sai de $O(k^3)$ para $O(k^2)$.
- **Código final:** O algoritmo abaixo apresenta a implementação para o problema. Para obter a resposta basta preencher o *array* v com os cortes necessários, estes deverão estar com os índices entre $1...k$. 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 4 - Implementação Knuth Optimization

```
#define MAXN 1010
#define inf 99999999

int v[MAXN];
int dp[MAXN][MAXN];
int opt[MAXN][MAXN];

int knuthOptimization(int n, int k){
    // Adicionado os elementos extras
    v[0] = 0;
    v[++k] = n;
    k++;

    for(int tam = 1; tam <= k; tam++){
        for(int i = 0; i+tam <= k; i++){
            int j = i+tam-1;
            if(tam <= 2){
                dp[i][j] = 0;
                opt[i][j] = i;
                continue;
            }
            dp[i][j] = inf;
            for(int x = opt[i][j-1]; x <= opt[i+1][j]; x++){
                if(dp[i][x] + dp[x][j] + v[j]-v[i] < dp[i][j]){
                    dp[i][j] = dp[i][x] + dp[x][j] + v[j]-v[i];
                    opt[i][j] = x;
                }
            }
        }
    }
    return dp[0][k-1];
}
```

5.5 CONVEX HULL TRICK

- **Problema:** Uma estrada está sendo construída e é necessário cobrir n pontos dela. O custo para se cobrir um pedaço da estrada que vai do ponto x até o ponto y, é definido como sendo $(x-y)^2+c$, onde 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 8 mostra uma possível solução.

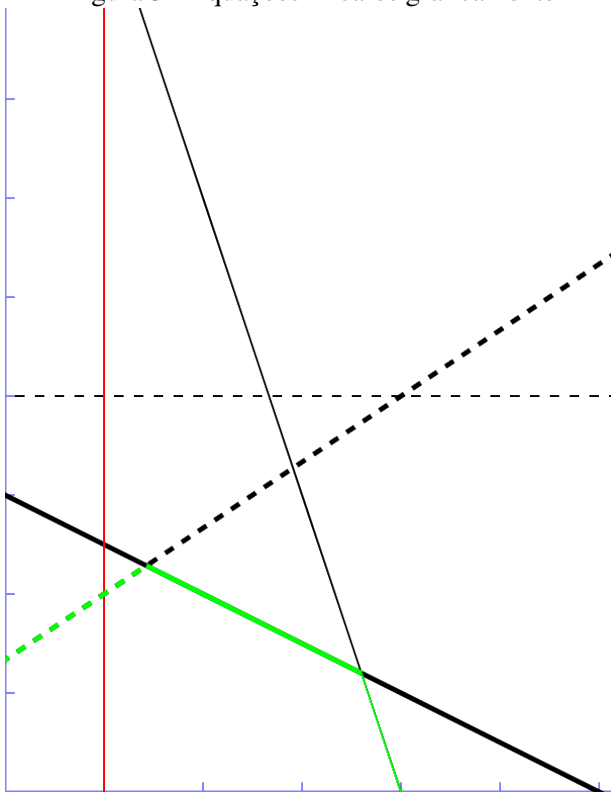
$$dp[i] = \begin{cases} 0 & \text{se } i = 0 \\ min_{1 \leq j \leq i}(dp[j-1] + (v[i]-v[j])^2 + c) \end{cases} \tag{8}$$

  Para o cálculo de cada estado i da tabela, é necessário percorrer todos os índices que estão a sua direita, com a finalidade de escolher algum ponto onde seria colocado o início da cobertura. Devido a isto, a complexidade de cada estado fica $O(n)$ e a complexidade para o cálculo de todos os n estados fica $O(n^2)$. 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. $dp[j-1] + (v[i]-v[j])^2 + c = -2 * v[j] * v[i] + v[j]^2 + dp[j-1] + v[i]^2 + c$. 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 $a = -2 * y$ 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 meia parábola, 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 parábola selecionada, assim não há necessidade de tentar minimizá-los.

- **Convex Hull Trick:** Imaginemos que diversas equações lineares são dadas, todas da forma $f(x) = ax + b$ 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 equações[7]. 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 minimizam o valor de $x = 1$. A figura 3 mostra graficamente a distribuição destas equações que podem ser vistas como restas no plano.

Figura 3 - Equações lineares graficamente



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 $f(x) = 4$, esta 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 $f(x)$, isto 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

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

uma busca binária[8], onde para cada momento um valor é 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 $O(logn)$, 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, assim, se utilizado a ideia de sempre que adicionar uma nova reta for necessário ordenar todas elas, o algoritmo ficaria com a complexidade $O(n^2 * logn)$.

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 um reta com angulação maior das já presentes, ao realizar essa operação, alguns casos podem acontecer:

1º) A nova reta não é melhor que nenhuma outra que já estava presente, sendo assim, não há necessidade de incluir esta no conjunto;

2º) A nova reta é melhor do que algumas outras a partir de algum de valor de x;e

3º) 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 mantido uma pilha[9], com todas as retas que ainda são ótimas em algum ponto e no seu topo possui 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 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 da penúltima e 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.

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, $x_1, x_2, ..., x_n$, portanto, pode ser mantido um ponteiro p_i, que informa

[8] http://www.geeksforgeeks.org/binary-search/

[9] http://www.geeksforgeeks.org/stack-data-structure/

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 $p_i + 1$. 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, isto se deve ao fato de que cada reta será avaliada apenas uma vez, no momento que para alguma consulta esta não for a melhor, é impossível em algum momento ela se tornar ótima.

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

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, portanto, é preciso uma outra estrutura de dados. Com o intuito de deixar a complexidade o mais baixo possível pode ser utilizado uma árvore balanceada para manter as retas ordenadas.

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

- **Benefícios:** Apesar da técnica demostrada realizar operações em retas, é possível utilizá-la no problema proposto, onde estão são parábolas, mas, como visto na etapa de análise de particularidades, as funções de custo tem a forma $f(x) = ax + b$, ou seja, pode ser descrito da mesma maneira de uma reta, podendo assim ser aplicado neste problema. Como pode ser notado, para resolver o estado *i*, só as retas menores que *i* são importantes, assim, pode ser inserido a reta relacionado a este estado no momento 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 usado 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 abaixo apresenta a implementação 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 $1...n$. 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 5 - Implementação Convex Hull Trick 1

```
#define MAXN 10
#define inf 99999999
int v[MAXN] = {0, 1, 3, 8, 10};
int dp[MAXN];
int A[MAXN], B[MAXN];
int tam, ponteiro;

int quadrado(int x){ return x*x; }

void adicionarLinha(int a, int b){
    while(tam >= 2 &&
            (B[tam-2] - B[tam-1]) * (a-A[tam-1]) >=
             (B[tam-1]-b) * (A[tam-1]-A[tam-2]))
        tam--;
    A[tam] = a;
    B[tam] = b;
    tam++;
}

int consultar(int x){
    ponteiro = min(ponteiro, tam-1);
    while(ponteiro+1 < tam &&
        A[ponteiro+1]*x+B[ponteiro+1] <= A[ponteiro]*x + B[ponteiro])
        ponteiro++;
    return A[ponteiro]*x + B[ponteiro];
}

int cht(int n, int c){
    // Inicializando a pilha do Convex Hull Trick
    ponteiro = tam = 0;

    // Caso base
    dp[0] = 0;
    adicionarLinha(-2*v[1], quadrado(v[1]));
    for(int i = 1; i <= n; i++){
        dp[i] = consultar(v[i]) + quadrado(v[i]) + c;
        if(i < n)
            adicionarLinha(-2*v[i+1], quadrado(v[i+1])+dp[i]);
    }
    return dp[n];
}
```

6 CONSIDERAÇÕES FINAIS

Considerações finais do trabalho, com conclusão e trabalhos futuros

6.1 CONCLUSÕES

6.2 TRABALHOS FUTUROS

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.

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>.

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

DECLARAÇÃO DE PRÓPRIA AUTORIA

Teresópolis, XX/XX/2017

Eu, Gabriel Lagoa Duarte, declaro para fins de conclusão do Curso de Bacharelado em Ciência da Computação do UNIFESO, que este Trabalho de Conclusão de Curso é de minha própria autoria, estando ciente das consequências disciplinares a que estarei sujeito caso seja comprovada fraude ou má-fé. Sem mais, subscrevo-me,

Atenciosamente,

_____

Gabriel Lagoa Duarte