UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

THIAGO HENRIQUE NEVES COELHO

PROGRAMAÇÃO DINÂMICA NA PRÁTICA

Do básico ao intermediário

RIO DE JANEIRO
2021

THIAGO HENRIQUE NEVES COELHO

PROGRAMAÇÃO DINÂMICA NA PRÁTICA

Do básico ao intermediário

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Claudson Ferreira Bornstein

RIO DE JANEIRO

2021

CIP - Catalogação na Publicação

C672p

Coelho, Thiago Henrique Neves
Programação dinâmica na prática: do básico ao intermediário / Thiago Henrique Neves Coelho. -- Rio de Janeiro, 2021.
58 f.

Orientador: Claudson Ferreira Bornstein.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Matemática, Bacharel em Ciência da Computação, 2021.

1. Programação dinâmica. I. Bornstein, Claudson Ferreira, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

THIAGO HENRIQUE NEVES COELHO

PROGRAMAÇÃO DINÂMICA NA PRÁTICA
Do básico ao intermediário

Trabalho de conclusão de curso de graduação apresentado ao Departamento de Ciência da Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 4 de março de 2021

BANCA EXAMINADORA:

Claudson Ferreira Bornstein
Prof. Claudson Ferreira Bornstein
PhD (UFRJ)

Participação por videoconferência
Prof. Vinícius Gusmão Pereira de Sá
D.Sc. (UFRJ)

Participação por videoconferência
Prof. João Antonio Recio da Paixão
D.Sc. (UFRJ)

AGRADECIMENTOS

Gostaria de agradecer, primeiramente, ao meu orientador Prof. Claudson Ferreira Bornstein, que me forneceu todos os direcionamentos necessários e tornou possível a elaboração deste trabalho.

Também dedico um agradecimento especial à Profª. Márcia Rosana Cerioli, que possibilitou minhas participações na Maratona de Programação, onde desenvolvi grande interesse por algoritmos, principalmente por programação dinâmica, que é o assunto que abordo aqui.

Por fim, agradeço à UFRJ pela estrutura fornecida durante minha graduação e a todos os professores dos quais fui aluno, que muito contribuíram para minha formação.

RESUMO

Programação dinâmica é uma técnica que consiste em dividir um problema em subproblemas menores, resolvê-los, armazenar as respostas e utilizá-las na solução do problema original. Este trabalho tem como objetivo introduzir essa técnica e deixá-la mais familiar ao leitor, utilizando de uma abordagem mais prática, onde serão apresentados problemas de programação dinâmica e explicadas, detalhadamente, as execuções de cada algoritmo. Foi feita uma categorização dos problemas apresentados em três níveis: básico, com o objetivo de deixar as ideias para o desenvolvimento de uma solução envolvendo programação dinâmica mais intuitivas; básico com *strings*, para trazer uma nova ideia que é bastante utilizada na solução dessa classe de problemas; e intermediário, que é composto de problemas cujas soluções envolvem alguma outra técnica combinada à programação dinâmica, com o objetivo de fazer o leitor entender o quão amplo pode ser o uso das técnicas de programação dinâmica para resolver problemas bastantes diversificados. Durante o estudo dos algoritmos apresentados para cada problema, será possível identificar diversas semelhanças entre alguns deles. Por fim, espera-se que o leitor esteja mais apto a resolver novos problemas de programação dinâmica após a leitura deste material.

Palavras-chave: programação dinâmica. algoritmos. programação competitiva.

ABSTRACT

Dynamic programming is a technique that consists in dividing a problem into smaller subproblems, solving them, storing the answers and using them to solve the original problem. This paper aims to introduce this technique and make it more familiar to the reader, using a more practical approach, where dynamic programming problems will be presented and the executions of each algorithm will be explained in detail. A categorization of the problems presented was made at three groups: basic, in order to make the ideas for the development of a solution involving dynamic programming more intuitive; basic with strings, to bring a new idea that is widely used to solve this class of problems; and intermediary, which is composed of problems whose solutions involve some other technique combined with dynamic programming, in order to make the reader understand how wide the use of dynamic programming techniques can be to solve quite diverse problems. During the study of the algorithms presented for each problem, it will be possible to identify several similarities between some of them. Finally, it is expected that the reader will be better able to solve new dynamic programming problems after reading this material.

Keywords: dynamic programming. algorithms. competitive programming.

LISTA DE CÓDIGOS

SUMÁRIO

1 INTRODUÇÃO

A QUEM SE DIRIGE ESTE TRABALHO

De forma geral, este trabalho dirige-se a programadores interessados em aprender mais sobre programação dinâmica. Os tópicos abordados aqui vão desde a apresentação do conceito de programação dinâmica, quanto a algoritmos de nível intermediário. Embora o conteúdo seja focado em programação competitiva, ele também pode ser bem aproveitado por outros estudantes que desejam aprofundar seus conhecimentos sobre programação dinâmica.

PRÉ-REQUISITOS PARA ESTE CONTEÚDO

Para que o conteúdo deste trabalho seja proveitoso, é interessante que o leitor tenha um entendimento prévio de lógica de programação, conhecimento básico sobre análise de complexidade e familiaridade com alguma linguagem de programação. Os códigos dos algoritmos apresentados serão escritos em C++, mas não é necessário ter conhecimento específico dessa linguagem para compreender os tópicos, pois as ideias serão explicadas de forma didática antes do código em si.

O QUE É PROGRAMAÇÃO DINÂMICA?

Programação dinâmica é uma técnica que consiste em dividir um problema em subproblemas menores, resolvê-los, armazenar as respostas e utilizá-las na solução do problema original. A ideia é que os subproblemas sejam versões menores do mesmo problema e que a solução do problema inicial também seja facilmente obtida uma vez que tenhamos as respostas dos menores. Outro ponto é que essa divisão não ocorre apenas uma vez, dado que os subproblemas normalmente também serão divididos em outros subproblemas menores ainda, até chegar em um ponto em que os problemas podem ser resolvidos diretamente e não há mais necessidade de dividir.

Antes de aprofundar mais no conceito, vamos ver um exemplo do que seria um caso de programação dinâmica e identificar todos os elementos citados na definição acima:

Problema: Sequência de Fibonacci

A sequência de Fibonacci é uma famosa sequência matemática definida da seguinte forma: $F(0) = 0$; $F(1) = 1$; $F(n) = F(n-1) + F(n-2)$, para $n > 1$. Dado um valor de n, deseja-se saber $F(n)$.

A solução mais direta para este problema seria criar uma função F que recebe um parâmetro n e retorna:

- 0, se $n = 0$
- 1, se $n = 1$
- $F(n-1) + F(n-2)$, se $n > 1$

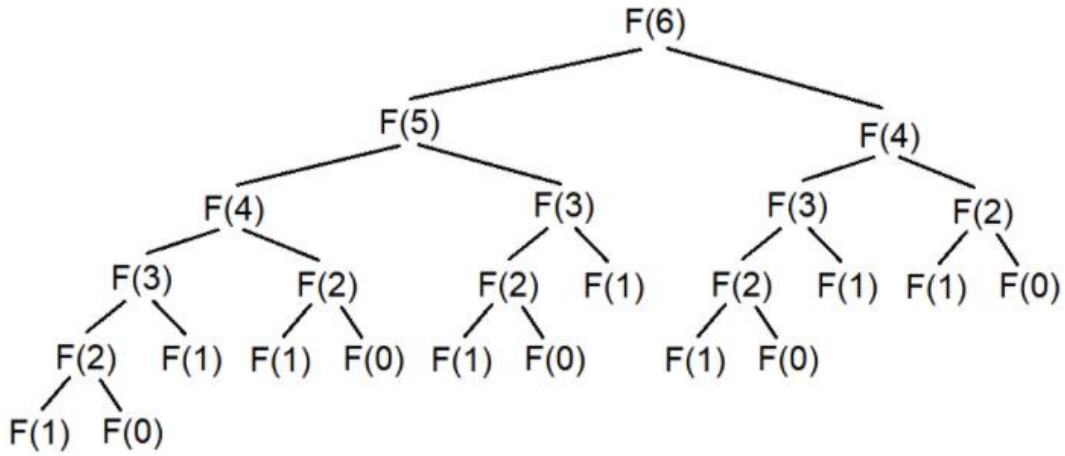
O Código 1 demonstra a implementação dessa função.

Código 1 – Sequência de Fibonacci

```
int fib(int n) {
        if(n == 0) return 0;
        if(n == 1) return 1;
        return fib(n-1) + fib(n-2);
}
```

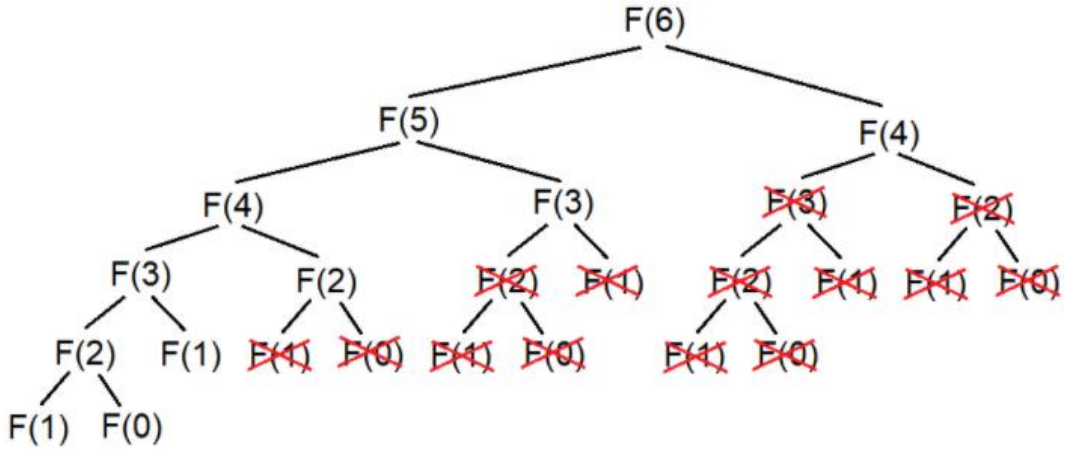
Dessa forma, para cada $F(n)$, seria realizado um cálculo recursivo completo dos valores de $F(n-1)$ e $F(n-2)$. O problema disso é que muitos valores serão computados mais de uma vez, como pode ser observado na Figura 1, que ilustra a árvore de execução para $F(6)$.

Figura 1 – Árvore de execução completa da sequência de Fibonacci



Observe que o valor de $F(4)$ aparece sendo calculado 2 vezes, o $F(3)$ aparece 3 vezes, o $F(2)$ aparece 5 vezes e o $F(1)$, 8 vezes. É possível notar que a quantidade de chamadas também segue a sequência de Fibonacci, logo ela cresce exponencialmente conforme o valor de n aumenta. Se armazenarmos os valores de cada número da sequência logo após calculá-los pela primeira vez e utilizarmos esta resposta para as próximas chamadas a este valor, conseguiremos garantir que nada precise ser calculado mais de uma vez. Dessa forma, grande parte das chamadas de função para valores já calculados anteriormente não precisarão mais ser realizadas. No caso analisado neste exemplo, as chamadas que puderam ser descartadas são as riscadas na Figura 2.

Figura 2 – Árvore de execução podada da sequência de Fibonacci



Se o ganho de performance não ficou claro com a imagem, há outro fato relevante: o número de chamadas totais cresce exponencialmente em função do valor de n, enquanto que a quantidade de chamadas úteis (aquelas utilizadas para calcular um valor pela primeira vez ou para retornar seu valor já armazenado) cresce linearmente. Para se ter uma ideia, no caso de $n = 40$, serão feitas 331160281 chamadas no total, das quais 331160202 não precisariam ser feitas e apenas 79 são úteis.

Agora podemos dizer que estamos usando programação dinâmica para resolver o problema. Em termos de código, uma das formas de utilizar a estratégia descrita é criando um vetor global *Fib* que armazenará os valores da sequência já calculados. Esse vetor deve ser inicializado com algum valor inválido[1], como -1, e atualizado a cada cálculo feito. Se a chamada for feita a um elemento cujo valor esteja registrado em *Fib*, ele será diretamente retornado.

Essa técnica de armazenar resultados em uma espécie de *cache* para reduzir a quantidade de chamadas recursivas (ou, em outras palavras, "podar" a árvore de recursão) é chamada de memoização (*memoization* em inglês).

Uma possível implementação está no Código 2.

Código 2 – Sequência de Fibonacci com memoização

```
int Fib[MAXN]; // Inicializado com -1 em todas as posicoes

int fib(int n) {
        if(Fib[n] != -1) return Fib[n];
        if(n == 0) return 0;
        if(n == 1) return 1;
        return Fib[n] = fib(n-1) + fib(n-2);
}
```

[1] Na prática, esse valor não será realmente impossível de aparecer no resultado de algum cálculo devido ao *overflow* dos números inteiros no C++, mas seria impossível se estivéssemos utilizando aritmética modular, por exemplo

A complexidade do algoritmo sem memoização é exponencial em n. Uma forma simples de perceber isso é comparando a quantidade de chamadas recursivas com os valores da própria sequência de Fibonacci. Como a nossa primeira função só retorna 0, 1 ou a soma de novas chamadas recursivas, concluímos que foi necessário retornar $F(n)$ vezes o valor 1 para que o valor de $F(n)$ fosse calculado. Além disso, há os casos em que foi retornado 0 ou a soma das chamadas recursivas.

Por outro lado, a complexidade do algoritmo com memoização é $O(n)$, isto é, linear[2] em n. Isso significa que o tempo de execução desse algoritmo tende a crescer menos do que o anterior conforme cresce o valor de n. É possível perceber esse ganho na prática. Ao rodar o primeiro código com valor de n igual a 20, aumentando-o até chegar a 100, o tempo de execução será tão grande que não será mais viável esperar o retorno da função. Entretanto, ao rodar o segundo código, o tempo de execução será irrisório até para valores de n maiores que 10 milhões. Como os valores retornados pela função fib serão tão grandes que ocasionarão um *overflow*, pode-se realizar uma alteração no código antes de fazer esse teste, retornando a soma em módulo 10000000 (nesse contexto, $fib(n-1) + fib(n-2)$ seria trocado por $(fib(n-1) + fib(n-2))\%10000000$). Assim, serão obtidos os 7 últimos dígitos do termo da sequência de Fibonacci. Essa modificação não é necessária para que a diferença de tempo seja medida, mas ela faz com que os resultados voltem a fazer algum sentido, pois não significavam nada devido ao *overflow*.

TOP-DOWN X *BOTTOM-UP*

O algoritmo proposto no exemplo anterior é uma solução *top-down* do problema. Este tipo de solução baseia-se em resolver o problema partindo de um caso maior e mais complexo, quebrando-o em casos menores, até chegar nos casos triviais. No exemplo dado, escrevemos uma função que começa em $F(n)$ e "desce" recursivamente, resolvendo problemas para valores menores e juntando-os para construir a solução. Uma vantagem das soluções *top-down* é que a "fórmula" utilizada na recursão normalmente fica bem explícita no código.

Entretanto, há outra estratégia: a solução *bottom-up*, que se propõe a resolver o problema a partir dos casos triviais, utilizando-os para construir as soluções dos casos maiores. No exemplo dado, começaríamos com um vetor inicializado nos casos bases, que são $Fib(0) = 0$ e $Fib(1) = 1$. A partir disso, é feito um *loop* que preencherá os valores de 2 a n utilizando, a cada preenchimento, as posições anteriores já preenchidas do vetor. Uma vantagem deste tipo de solução é que não há recursão. O *loop* diz exatamente o passo a passo a ser executado pelo computador, o que pode fazer com que o código seja mais fácil de debugar. Além disso, por não utilizar chamadas recursivas, é esperado que a solução

[2] Note que o algoritmo não é linear no tamanho de sua entrada, pois ela não é composta por n valores, mas sim por $log(n)$ *bits* que representam o número n. Isso é chamado de complexidade pseudo-polinomial

bottom-up tenha uma performance melhor que a *top-down*. Uma solução *bottom-up* para o problema da Sequência de Fibonacci está ilustrada no Código 3.

Código 3 – Sequência de Fibonacci *bottom-up*

```
Fib[0] = 0;
Fib[1] = 1;

for(int i = 2; i <= n; i++) {
        Fib[i] = Fib[i-1] + Fib[i-2];
}
```

2 PROBLEMAS BÁSICOS

O problema anterior serviu para apresentar alguns conceitos iniciais importantes, como memoização, complexidade e as técnicas de solução *top-down* e *bottom-up*. Entretanto, ele possui uma característica que os outros problemas não possuem: a relação de recorrência foi dada no próprio enunciado. Dessa forma, foi necessário apenas copiar a fórmula dada na questão para o código, aplicando uma técnica *top-down* ou *bottom-up*. Os novos problemas não darão a relação de recorrência de forma explícita, sendo necessário que ela seja descoberta para que a programação dinâmica possa ser aplicada.

Para os problemas apresentados neste capítulo e para grande parte dos problemas de programação dinâmica, o maior desafio será encontrar a relação de recorrência. Após isso, fazer o código será a parte mais fácil.

2.1 SOMA CONTÍGUA MÁXIMA 1D

Problema: É dada uma sequência S de n números inteiros não-nulos. Deseja-se saber qual o maior valor possível de se obter com uma soma contígua de elementos de S.

Uma *soma contígua* é, para dados índices i e j, o valor de $\sum_{k=i}^{j} S(k)$. Por exemplo, para a sequência (5, -10, 2, 3, 6, -5, 7, -20, 10), a maior soma contígua tem valor 13, que corresponde à soma da subsequência (2, 3, 6, -5, 7).

A solução ingênua para este problema consiste em calcular, para todos os pares i e j de índices, sendo $i \leq j$, a soma $\sum_{k=i}^{j} S(k)$ e escolher a de maior valor. Como há cerca de n^2 pares de índices, e é necessário realizar no máximo n operações para calcular a soma de uma subsequência, a complexidade desta solução seria de $O(n^3)$.

É possível melhorar a solução ingênua realizando um pré-processamento da sequência S, armazenando sua soma acumulada em um vetor S_{sum}, ou seja, $S_{sum}(i) = \sum_{k=1}^{i} S(k)$. Para o primeiro índice, teremos $S_{sum}(1) = S(1)$, para os demais, devemos percorrer toda a sequência, a partir do segundo índice até o enésimo, realizando a seguinte operação: $S_{sum}(i) = S_{sum}(i-1) + S(i)$. Tendo feito isso, é possível calcular a soma contígua de quaisquer pares i e j de índices em uma operação, que será $S_{sum}(j) - S_{sum}(i-1)$. Dessa forma, a complexidade da solução ingênua passa a ser $O(n^2)$, devido aos $O(n^2)$ pares de índices. Entretanto, ainda é possível obter uma solução melhor.

Em 1977, Joe Kadane propôs uma solução de complexidade $O(n)$ para este problema. Apresentaremos um algoritmo de programação dinâmica, também de complexidade $O(n)$, inspirado na solução de Kadane (BENTLEY, 1999). O algoritmo consiste em percorrer toda a sequência S, de 1 a n, e, para cada índice i que estivermos considerando no

momento, atualizar uma variável *sum*, que armazenará o valor da soma contígua máxima de uma subsequência que termina no elemento $S(i)$, necessariamente incluindo-o. A cada iteração, deve-se considerar 2 cenários para determinar o valor de *sum*:

1. Consideramos o elemento $S(i)$ como parte de uma subsequência contígua já existente para realizar a soma. Dessa forma, como a variável *sum* já possui a soma contígua máxima até $S(i-1)$, seu novo valor será de $sum + S(i)$

2. Começamos uma nova subsequência. Dessa forma, descartamos a soma anterior, fazendo o novo valor de *sum* ser igual a $S(i)$

O valor de *sum* será igual ao máximo entre os dois cenários. A cada iteração, também atualizaremos uma variável max_sum para armazenar a soma contígua máxima já encontrada até o momento.

Um ponto que pode ser observado é que o segundo cenário só será o melhor quando o valor de *sum* for menor que 0. Dessa forma, uma outra abordagem seria considerar que devemos somar os elementos da sequência e zerar a soma obtida sempre que ela for negativa, pois vale mais iniciar a soma de uma subsequência com um valor nulo do que um negativo.

Por exemplo, para $S = (5, -10, 2, 3, 6, -5, 7, -20, 10)$, os valores de *sum* para cada iteração estão representados no Quadro 1 (o asterisco indica as posições em que a soma foi recomeçada, conforme descrito no segundo cenário).

Quadro 1 – Resolução para soma contígua máxima 1D

S	5	-10	2	3	6	-5	7	-20	10
sum	5	-5	2*	5	11	6	13	-7	10*

De forma resumida, devemos iniciar o valor de *sum* com 0 e aplicar a seguinte regra a cada elemento de S:

- $sum = \max(S(i), S(i) + sum)$, para todo i

A resposta do problema será o maior valor de *sum* obtido em qualquer passo.

Uma possível implementação desse algoritmo é a apresentada no Código 4. A resposta final estará armazenada em max_sum.

Código 4 – Soma Contígua Máxima 1D *bottom-up*

```
int sum = 0, max_sum = 0;
for(int i = 0; i < N; i++) {
        sum = max(S[i], S[i] + sum);
        max_sum = max(max_sum, sum);
}
```

Complexidade

O algoritmo consiste em percorrer todos os n elementos de S, realizando uma quantidade constante de operações em cada um. Dessa forma, sua complexidade é $\boldsymbol{O(n)}$.

2.2 SOMA CONTÍGUA MÁXIMA 2D

Problema: É dada uma matriz $\boldsymbol{A}$ de dimensões $\boldsymbol{n \times m}$ com valores inteiros não-nulos. Deseja-se saber qual o valor máximo da soma dos elementos de alguma submatriz de $\boldsymbol{A}$.

Na Figura 3, a matriz à esquerda é um exemplo possível de uma matriz A de dimensões 4×5. Na matriz à direita, a submatriz de soma máxima está marcada. Então, para esse caso, a resposta seria igual a 7 + 1 - 2 - 1 + 3 + 1 + 9 -15 + 13 = 16.

Figura 3 – Exemplo de soma contígua máxima 2D

5	-10	2	3	6
-8	7	1	-2	-7
2	-1	3	1	-5
-1	9	-15	13	-20

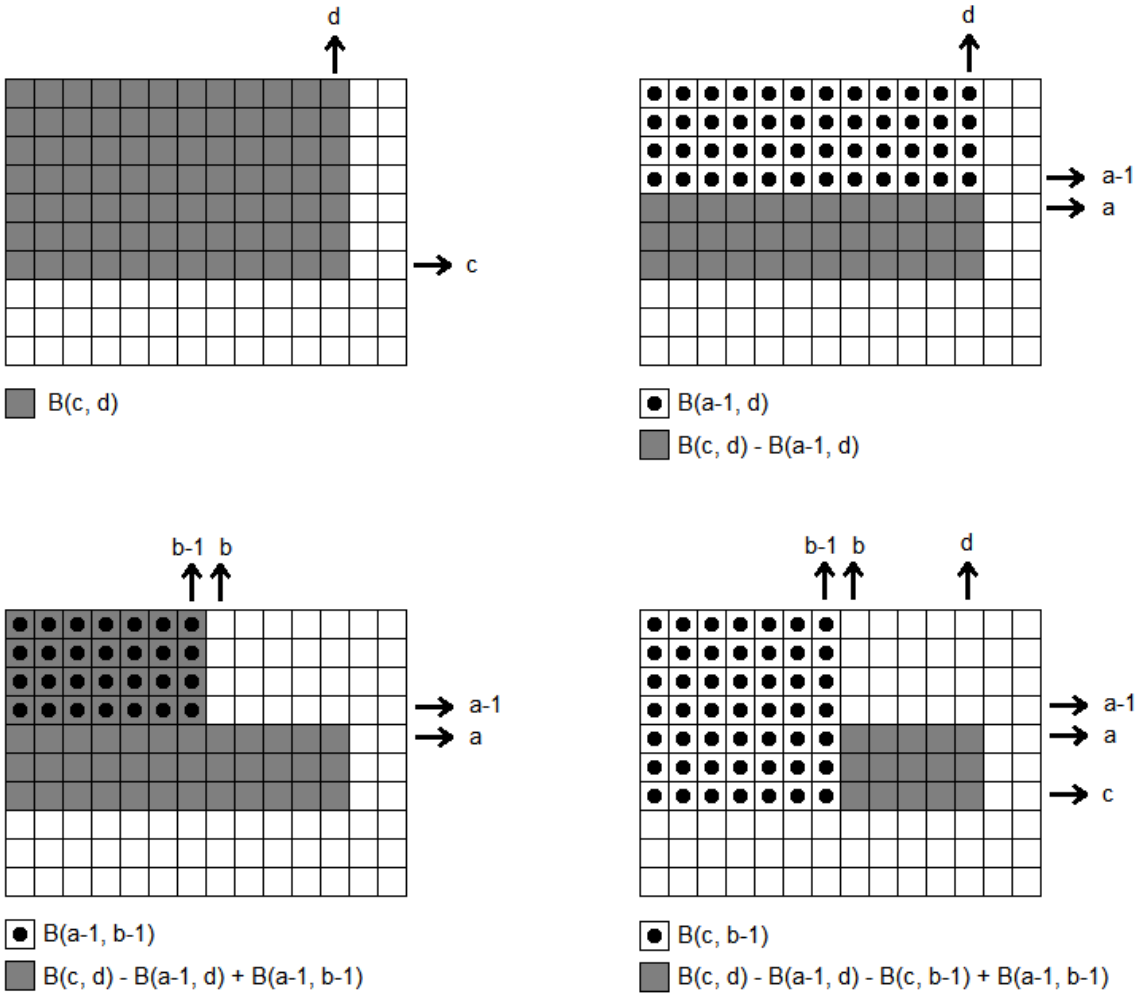
5	-10	2	3	6
-8	**7**	**1**	**-2**	-7
2	**-1**	**3**	**1**	-5
-1	**9**	**-15**	**13**	-20

A solução ingênua para este problema seria computar a soma de todas as submatrizes possíveis e escolher a maior dentre elas. Cada submatriz pode ser representada por dois pares de índices (a, b) e (c, d), que representam, respectivamente, as extremidades superior esquerda e inferior direita da submatriz. Como há $O(n \cdot m)$ possibilidades de escolha para cada par de índices, haverá $O(n^2 \cdot m^2)$ submatrizes possíveis. Dessa forma, visto que seriam necessários $O(n \cdot m)$ passos para determinar a soma dos elementos de cada submatriz, a complexidade final dessa solução seria $O(n^3 \cdot m^3)$.

Assim como na versão 1D deste problema, é possível melhorar a complexidade realizando um pré-processamento na matriz A, armazenando sua soma acumulada em outra matriz B (HALIM; HALIM, 2013). Dessa forma, $B(i, j)$ armazenará a soma dos valores da submatriz de A que está entre os pares de índices $(1, 1)$ e (i, j), ou seja, $B(i, j) = \sum_{w=1}^{i} \sum_{z=1}^{j} A(w, z)$. A soma dos valores de uma submatriz entre os pares de índices (a, b) e (c, d) é igual a $B(c, d) - B(a - 1, d) - B(c, b - 1) + B(a - 1, b - 1)$. O valor $B(a - 1, b - 1)$ é somado no final porque este intervalo é subtraído duas vezes ao subtrairmos os valores $B(a-1, d)$ e $B(c, b-1)$. A Figura 4 ilustra a lógica dessa operação entre os intervalos.

O pré-processamento pode ser realizado em tempo $O(n \cdot m)$ realizando o cálculo $B(i, j) = B(i - 1, j) + B(i, j - 1) - B(i - 1, j - 1) + A(i, j)$ para todo par de índices

Figura 4 – Ilustração da soma acumulada em matrizes. Seguindo a ordem da esquerda para a direita e de cima para baixo, a primeira matriz mostra o intervalo B(c,d) completo. A segunda ilustra, na área cinza, a subtração do intervalo B(a-1, d). A terceira ilustra a adição do intervalo B(a-1, b-1). Por fim, a quarta ilustra, na área cinza, a subtração de B(c, b-1) do intervalo cinza da terceira matriz. Dessa forma, a operação para obter a soma acumulada do intervalo cinza da quarta matriz está mostrada em sua legenda



(i, j). Ainda será necessário comparar todas as $O(n^2 \cdot m^2)$ submatrizes, mas o tempo para calcular a soma de seus elementos será da ordem de $O(1)$, então a complexidade final desta solução será $O(n^2 \cdot m^2)$.

Entretanto, ainda há como melhorar a complexidade da solução deste problema. Na Seção 2.1, foi mostrada uma solução de complexidade $O(n)$ para encontrar a soma contígua máxima em um vetor (uma dimensão) de tamanho n. O algoritmo que será apresentado a seguir é uma extensão dessa solução à versão 2D do problema.

Primeiramente, considere o seguinte subproblema, que é uma simplificação do original: **Deseja-se saber qual o valor máximo da soma dos elementos de algum subvetor de uma linha de A**. Com isso, seria possível resolver o subproblema em tempo $O(n \cdot m)$

aplicando o algoritmo da versão 1D em cada uma das linhas da matriz, visto que há n linhas de tamanho m.

Agora considere o subproblema para submatrizes de exatamente duas linhas. Para isso, podemos utilizar uma matriz B para realizar um pré-processamento, fazendo a atribuição $B(i, j) = A(i, j) + A(i + 1, j)$ para todos os pares de índices da matriz e considerando B como uma matriz de dimensões $(n - 1) \times m$. Dessa forma, cada linha i da matriz B corresponderá à soma das linhas i e $i + 1$ da matriz A, conforme pode ser visto na Figura 5. Assim, o subproblema pode ser resolvido aplicando o algoritmo da versão 1D a cada uma das $n - 1$ primeiras linhas de A.

Figura 5 – Pré-processamento para armazenar linhas somadas de 2 em 2 (matriz antes do processamento à esquerda, e após à direita)

5	-10	2	3	6
-8	7	1	-2	-7
2	-1	3	1	-5
-1	9	-15	13	-20

-3	-3	3	1	-1
-6	6	4	-1	-12
1	8	-12	14	-25

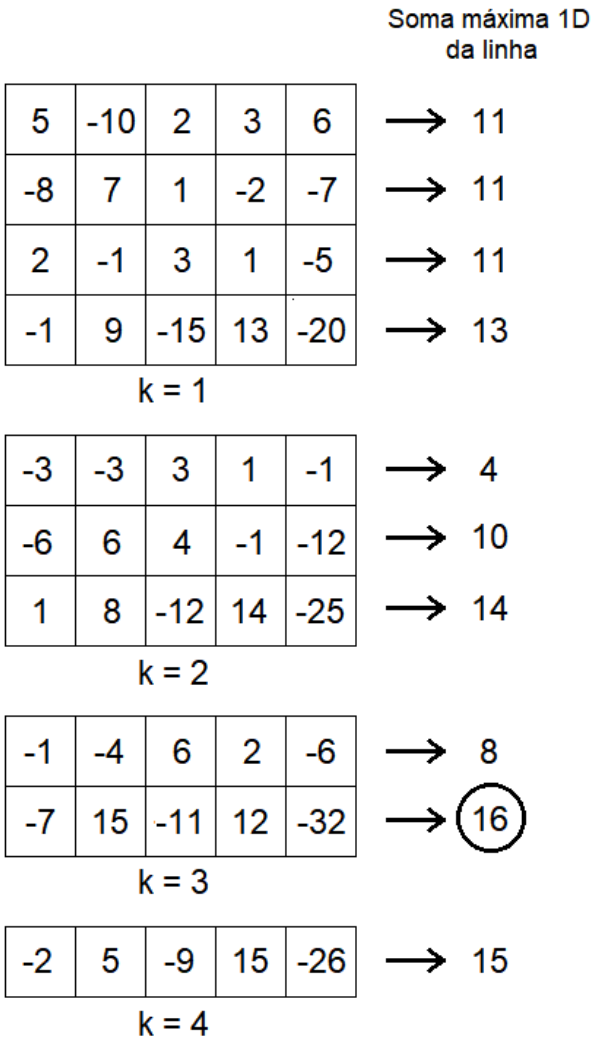
Seguindo o mesmo raciocínio para um subproblema de submatrizes de três linhas, podemos aplicar um pré-processamento semelhante, fazendo com que $B(i, j)$ seja igual a $A(i, j) + A(i + 1, j) + A(i + 2, j)$. Isso pode ser feito com apenas uma operação de soma, caso o pré-processamento do subproblema de duas linhas já tenha sido realizado e esteja armazenado em B. Dessa forma, a operação seria apenas a atribuição $B(i, j) = B(i, j) + A(i + 2, j)$, visto que o valor de $B(i, j)$ já será igual a $A(i, j) + A(i + 1, j)$. Assim, o subproblema pode ser resolvido aplicando a solução do problema 1D a cada uma das linhas de B, considerando-a como uma matriz $(n - 2) \times m$.

Por fim, o algoritmo consiste em resolver todos esses subproblemas de submatrizes de k linhas, para $1 \leq k \leq n$, sempre aproveitando o resultado do pré-processamento do subproblema anterior e, para realizar o novo, aplicando a operação de atribuição $B(i, j) = B(i, j) + A(i + k - 1, j)$ para todos os pares de índices de B. As dimensões que serão consideradas para a matriz B serão $(n - k + 1) \times m$. O maior valor de soma contígua máxima encontrado entre todos os valores de k será a resposta do problema.

Na Figura 6, pode-se ver como seria a execução do algoritmo para uma matriz. Repare que a soma máxima encontrada foi 16 e ela pode ser obtida somando uma submatriz de 3 linhas, pois o valor de k foi igual a 3.

Para a implementação, podemos considerar B como uma matriz inicialmente zerada. A primeira etapa do algoritmo, com $k = 1$, será responsável por igualar B a A, visto que $B(i, j) = B(i, j) + A(i, j)$ será equivalente a $B(i, j) = A(i, j)$, para $B(i, j) = 0$.

Figura 6 – Pré-processamento para armazenar linhas somadas de k em k



O algoritmo para resolver esse problema pode ser resumido da seguinte forma (a resposta para o problema estará armazenada na variável max_sum):

1. $k \leftarrow 1$, max_sum $\leftarrow 0$
2. Crie uma matriz B de dimensões $n \times m$ com todos os valores zerados
3. Enquanto $k < n$, faça:
   a) Para cada linha i de B, faça:
      i. Some a k-ésima linha de A na i-ésima linha de B
      ii. Aplique o algoritmo de soma contígua máxima 1D na i-ésima linha de B e armazene o resultado em uma variável chamada line_sum
   b) max_sum $\leftarrow$ max(line_sum, max_sum)
   c) $k \leftarrow k + 1$
   d) Diminua em 1 a altura de B

Uma possível implementação desse algoritmo está ilustrada no Código 5.

Código 5 – Soma Contígua Máxima 2D *bottom-up*

```
// B tem dimensoes NxM e todos os seus valores sao 0
int max_sum = 0;
for(int k = 1; k <= N; k++) {
        for(int i = 0; i < N - k + 1; i++) {
                int sum = 0;
                for(int j = 0; j < M; j++) {
                        B[i][j] += A[i + k - 1][j];
                        sum = max(B[i][j], B[i][j] + sum);
                        max_sum = max(max_sum, sum);
                }
        }
}
```

Nas linhas 9 e 10 do Código 5, pode-se perceber que está sendo aplicado o mesmo algoritmo de soma 1D apresentado anteriormente na linha i da matriz B. A cada nova linha, o valor de sum é zerado para que ele não acumule a soma entre linhas linhas diferentes.

Complexidade

Cada operação do algoritmo de soma contígua máxima 1D nas linhas de B terá complexidade de $O(m)$, pois cada linha tem tamanho m. Para cada k, a operação será realizada em $n - k + 1$ linhas. Dessa forma, a complexidade é de $O(n \cdot m)$ para cada k.

Por fim, como o valor de k vai de 1 a n, a complexidade final será $\boldsymbol{O(n^2 \cdot m)}$.

2.3 *ROD CUTTING*

Problema: Deseja-se cortar um bastão de N metros em pedaços de comprimentos inteiros. É dada uma tabela de preços que indica, para cada comprimento de i metros do bastão, seu valor $p(i)$. Qual a soma máxima em dinheiro possível de se obter cortando o bastão de N metros?

Por exemplo, suponha que N seja igual a 10 e a tabela de preços seja a seguinte:

Quadro 2 – Relação de cada tamanho de corte do bastão com seu preço

i	1	2	3	4	5	6	7	8	9	10
p(i)	3	4	8	10	10	11	23	23	24	25

Há muitas formas de dividir o bastão de 10 metros. A primeira é simplesmente não cortar, sobrando um pedaço de tamanho 10 e preço 25. Se cortarmos, por exemplo, em dez pedaços de tamanho 1, teremos um valor total de 30. Entretanto, o melhor cenário para este exemplo é cortar três pedaço de tamanho 1 e um de 7, totalizando um preço de 32.

A abordagem trivial para este problema seria gerar todas as combinações de cortes possíveis para um bastão de tamanho N e verificar a combinação que corresponde ao maior preço. Embora produza uma resposta correta, a complexidade dessa solução é exponencial, tornando-se muito lenta para valores um pouco maiores de N.

Outra abordagem que resultará em um tempo de execução muito menor para valores maiores de N é utilizar programação dinâmica. Discutiremos abaixo um algoritmo, com base no que foi apresentado em (CORMEN et al., 2009), de complexidade polinomial para resolver o problema.

A primeira coisa a se pensar é em como os subproblemas devem ser divididos. A estratégia que será utilizada resume-se em considerar como subproblemas casos em que o bastão tem um tamanho menor do que o que foi apresentado e utilizar as soluções desses subproblemas para chegar na resposta do problema original. De fato, para resolver o problema de um bastão de tamanho i, será necessário ter resolvido, anteriormente, todos os subproblemas que consideram bastões de tamanho 1 a $i-1$. Por exemplo, para saber a soma máxima possível de se obter com um bastão de 5 metros, primeiro deve-se determinar a soma máxima para bastões de 1, 2, 3 e 4 metros.

Utilizaremos um vetor de nome dp para armazenar as respostas dos subproblemas. Assim, a soma máxima possível de se obter com um bastão de i metros estará armazenada em $dp(i)$. Para resolver o subproblema do bastão de tamanho i, a estratégia utilizada será cortar em um pedaço de tamanho j e somar o custo desse corte à solução do subproblema relativo ao pedaço remanescente de tamanho $i-j$, que estará armazenado em $dp(i-j)$. Como não é conhecido a priori o valor de j que maximize essa soma de $c(j) + dp(i-j)$, deve-se realizar essa operação para todos os valores de j menores ou iguais a i e escolher aquele cuja soma seja máxima.

Para que fique claro como a estratégia descrita acima será feita de um modo geral, primeiro começaremos a construir para o caso específico dado no problema. O raciocínio é o seguinte:

- Para $i = 1$, temos o caso trivial. Só existe uma forma de cortá-lo, que é a de não fazer corte algum, logo $dp(1) = p(1) = 3$. O corte será [1].
- Para $i = 2$, há 2 opções: não cortar ou cortar em dois pedaços de tamanho 1. Em ambos os casos, a resposta será $dp(2) = 6$. O corte será [1,1].

Vamos resolver agora o caso de $i = 3$. Neste momento, entra a ideia da programação dinâmica. Aparentemente, teríamos 4 cenários possíveis de cortes: [1,1,1], [1,2], [2,1] e [3], mas podemos ignorar alguns desses. Como sabemos que a soma máxima para um bastão de tamanho 2 é obtida com dois cortes de tamanho 1, não faz sentido pensarmos no cenário [1,2] para um bastão de tamanho 3, por exemplo, pois essa remanescente de tamanho 2 seria melhor aproveitada se fosse cortada em duas de tamanho 1. Dessa forma, já sabemos previamente que o caso [1,1,1] é melhor que o [1,2].

Na prática, quem nos dá essa informação de quais casos precisamos calcular e quais podemos ignorar é o vetor *dp*. O raciocínio para calcular o caso $i = 3$ é o seguinte:

- Nenhum corte é aplicado (ou um corte de tamanho 3 é aplicado): $dp(3) = p(3) = 8$
- Cortamos um pedaço de tamanho 1 e aplicamos a solução ótima já conhecida para o pedaço de tamanho 2 que sobrou: $dp(3) = p(1) + dp(2) = 9$
- Cortamos um pedaço de tamanho 2 e aplicamos a solução ótima já conhecida para o pedaço de tamanho 1 que sobrou: $dp(3) = p(2) + dp(1) = 7$

De forma genérica, será considerada a seguinte relação de recorrência para determinar as respostas de cada subproblema:

- $dp(n) = 0$, se $n = 0$
- $dp(n) = \max_{1 \leq i \leq n} (p(i) + dp(n - i))$, se $n > 0$

Para o exemplo dado, o quadro final, com i, $p(i)$ e $dp(i)$ será o seguinte:

Quadro 3 – Menor custo possível de cortar todo o bastão para cada tamanho

i	1	2	3	4	5	6	7	8	9	10
p(i)	3	4	8	10	10	11	23	23	24	25
dp(i)	3	6	9	12	15	18	23	26	29	32

Um código em C++ para resolver este problema em poucas linhas utilizando a estratégia *bottom-up* é apresentado no Código 6.

Código 6 – *Rod-Cutting bottom-up*

```
dp[0] = 0;
for(int i = 1; i <= N; i++) {
        dp[i] = -INF;
        for(int j = 1; j <= i; j++) {
                dp[i] = max(dp[i], p[j] + dp[i-j]);
        }
}
```

Outra forma de escrevê-lo poderia ser conforme ilustrado no Código 7.

Código 7 – *Rod-Cutting bottom-up* - Versão mais compacta

```
for(int i = 1; i <= N; i++) {
        dp[i] = p[i];
        for(int j = 1; j < i; j++) {
                dp[i] = max(dp[i], p[j] + dp[i-j]);
        }
}
```

Para a solução *top-down*, pode-se aplicar a técnica de memoização já citada anteriormente. Criamos uma função *rod_cutting*, que recebe um inteiro n e retorna o preço máximo que se pode obter ao cortar um bastão de tamanho n. Essa implementação está ilustrada no Código 8.

Código 8 – *Rod-Cutting top-down*

```
int p[MAXN];
int dp[MAXN]; // Inicializado com -1 em todas as posicoes

int rod_cutting(int n) {
        if(dp[n] != -1) return dp[n];
        if(n == 0) return dp[n] = 0;

        for(int i = 1; i <= n; i++) {
                dp[n] = max(dp[n], p[i] + rod_cutting(n-i));
        }
        return dp[n];
}
```

Complexidade

Para encontrar a solução para um bastão de tamanho i, já tendo calculado todas as outras para bastões menores, executamos i passos para percorrer cada uma dessas respostas. Como essa operação foi realizada para todos os números inteiros positivos menores que n, a complexidade final do algoritmo será $\boldsymbol{O(n^2)}$.

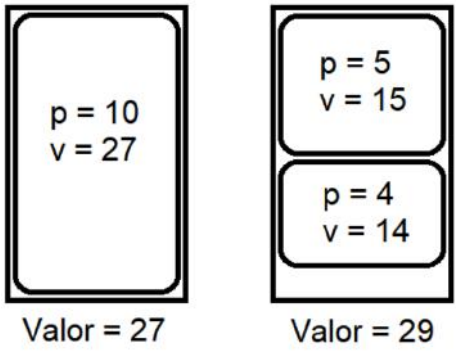
2.4 MOCHILA BOOLEANA

Problema: Há uma mochila com capacidade c, que indica o peso máximo que ela pode carregar. Além disso, há n objetos, cada qual com peso $p(i)$ inteiro e valor $v(i)$. Qual a soma máxima de valores que pode-se obter colocando alguma quantidade de objetos na mochila sem que o peso seja extrapolado?

Por exemplo, considere o cenário em que a mochila possui capacidade 10 e há 4 objetos com os seguintes [peso,valor]: [4,14], [10,27], [5,15], [2,10]. Poderíamos escolher o segundo item, lotando a mochila e tendo valor igual a 27, entretanto, nossa melhor alternativa seria escolher o primeiro e terceiro itens, somando um peso de 9 e valor de 29, conforme ilustrado na Figura 7. Não seria possível adicionar mais nenhum item na mochila, pois todos os pesos são maiores que a capacidade restante. É interessante notar também que o quarto item não foi incluído na solução ótima, mesmo tendo o melhor custo-benefício de valor/peso.

A solução trivial seria tentar todas as possibilidades de combinações de itens e escolher a que possui maior soma de valores. Entretanto, essa solução possui uma complexidade

Figura 7 – Possíveis escolhas de itens para a mochila



exponencial, pois há 2^n combinações possíveis de itens.

Com programação dinâmica, é possível desenvolver uma solução mais eficiente. A ideia do algoritmo é resolver, primeiramente, o mesmo problema considerando cargas máximas menores que o tamanho da mochila e com menos possibilidades de itens e utilizar as respostas desses problemas mais simples para construir a solução. Após isto, resolveremos o problema para cargas máximas cada vez maiores e com mais possibilidades de itens, até chegar no cenário criado no problema: uma carga máxima c considerando todos os n itens.

Dessa forma, deve-se considerar que os n itens estão numerados de 1 a n e tentaremos inseri-los, um de cada vez, respeitando esta ordem (a numeração em si não afeta o resultado do algoritmo, visto que o importante é só que os itens sejam inseridos seguindo alguma ordem). Para cada tentativa de inserção de um item i, de peso $p(i)$ e valor $v(i)$ em uma mochila de capacidade k, haverá apenas dois cenários a se considerar:

1. Não inserimos o novo item e mantemos a soma dos valores na mochila igual à que era antes, considerando apenas os $i - 1$ itens anteriores ao atual

2. Inserimos o novo item. Nossa melhor resposta para este caso seria combinar o valor do item que estamos inserindo com a carga que, combinada com este item, produzirá um peso de no máximo k. Dessa forma, a soma ótima dos valores na mochila atual será igual à soma ótima obtida com uma carga $k - p(i)$ e no máximo $i - 1$ itens anteriores mais o valor $v(i)$ do novo item

Para ficar clara a ideia, vamos entender o algoritmo resolvendo o caso dado no exemplo do problema: uma mochila de capacidade 10 e os itens [4,14], [10,27], [5,15] e [2,10].

Como já foi feito antes, considere primeiro um subproblema mais simples: vamos resolver considerando apenas o primeiro item, de [peso,valor] igual a [4,14]. Além de resolver subproblemas com menos itens, vamos simplificar ainda mais, considerando, inicialmente, que a carga máxima corresponderá a valores inferiores à capacidade da mochila, até chegar em 10. Para o item [4,14], qualquer carga máxima entre 0 e 3 não suportaria seu peso,

logo teria soma máxima de valores igual a 0, enquanto que as cargas máximas maiores ou iguais a 4 teriam a soma igual ao valor do item, que é 14. O quadro de soma/carga máxima seria o seguinte (*Carga* é a carga máxima possível que está considerada e S1 é a soma máxima colocando apenas o primeiro item):

Quadro 4 – Resolução da mochila booleana considerando 1 item

Carga	0	1	2	3	4	5	6	7	8	9	10
S1	0	0	0	0	14	14	14	14	14	14	14

Vamos agora colocar o segundo item (sem descartar o primeiro), de [peso,valor] igual a [10,27]. Para as cargas máximas entre 0 e 3, é evidente que a soma continuará 0, pois nenhum dos dois itens pode ser inserido. Para as cargas máximas entre 4 e 9, não podemos inserir o segundo item, mas sabemos que podemos obter uma soma de valores igual a 14 colocando o primeiro item, então essa ainda será a resposta para essas cargas. Por fim, podemos inserir o segundo item ao considerar a carga máxima igual a 10, deixando-a com valor igual a 27. Sendo S2 igual à soma máxima possível de obter considerando uma carga máxima igual ao valor de *Carga* podendo inserir até o segundo item apenas, teremos o seguinte quadro:

Quadro 5 – Resolução da mochila booleana considerando 2 itens

Carga	0	1	2	3	4	5	6	7	8	9	10
S1	0	0	0	0	14	14	14	14	14	14	14
S2	0	0	0	0	14	14	14	14	14	14	27

Inserindo o terceiro item, de [peso,valor] igual a [5,15], a ideia do algoritmo ficará explícita. Para as cargas máximas entre 0 e 4, o peso se mantém, pois não podemos inserir este item. Entre as cargas 5 e 8, escolhemos inserir o item, pois isso aumentará o valor da soma, deixando-a igual a 15, que é o valor do item. Na carga máxima igual a 9 acontece algo interessante: também vamos escolher inserir o item, mas a nova soma não será 15, mas sim 29. Isso ocorre porque a carga 9 pode ser obtida com os mesmos itens que utilizados para se obter a carga 4 (na linha de S2) mais o item de peso 5. Por fim, na carga 10, teremos novamente soma igual a 29 se inserirmos o item e soma igual a 27 se não inserirmos, logo a escolha será de inseri-lo. A tabela ficará desta forma:

Quadro 6 – Resolução da mochila booleana considerando 3 itens

Carga	0	1	2	3	4	5	6	7	8	9	10
S1	0	0	0	0	14	14	14	14	14	14	14
S2	0	0	0	0	14	14	14	14	14	14	27
S3	0	0	0	0	14	15	15	15	15	29	29

Para o último item, [2,10], será vantajoso inseri-lo nas cargas 2 e 3, aumentando a soma para 10. Nas cargas 4 e 5, teremos prejuízo inserindo, pois a soma nova seria igual

a 10. A soma para a carga 6 também melhora se inserirmos, ficando igual a 24. Entre as cargas 7 e 10, a nova soma seria de 24, o que só é uma melhora para as cargas entre 7 e 8. A tabela final ficaria da seguinte forma:

Quadro 7 – Resolução da mochila booleana considerando 4 itens

Carga	0	1	2	3	4	5	6	7	8	9	10
S1	0	0	0	0	14	14	14	14	14	14	14
S2	0	0	0	0	14	14	14	14	14	14	27
S3	0	0	0	0	14	15	15	15	15	29	29
S4	0	0	10	10	14	15	24	25	25	29	29

Considerando $dp(i, j)$ como o valor máximo obtido ao formar um peso menor ou igual a j considerando apenas os i primeiros itens, podemos, de forma genérica, considerar o seguinte algoritmo para resolver o problema:

- $dp(0, j) = 0$, para todo j
- $dp(i, j) = dp(i - 1, j)$, se $i > 0$ e $j < p(i)$
- $dp(i, j) = \max(dp(i - 1, j), dp(i - 1, j - p(i)) + v(i))$, se $i > 0$ e $j \geq p(i)$

Um código para resolver esse problema aplicando uma solução *bottom-up* poderia ser o seguinte (OBS: O código considera que os vetores v e p estão indexados em 1):

Código 9 – Mochila Booleana *bottom-up*

```
for(int j = 0; j <= C; j++) {
        // Casos em que nenhum item eh considerado
        dp[0][j] = 0;
}
for(int i = 1; i <= N; i++) {
        for(int j = 0; j <= C; j++) {
                if(j < p[i]) {
                        // O item nao pode ser inserido, pois seu
                        // peso eh maior que a carga maxima
                        dp[i][j] = dp[i-1][j];
                }
                else {
                        // Escolhendo entre nao inserir ou inserir
                        // o item
                        dp[i][j] = max(dp[i-1][j],
                                       dp[i-1][j-p[i]] + v[i]);
                }
        }
}
```

Complexidade

Observando o código da solução *bottom-up*, é bem simples deduzir a complexidade do algoritmo. Executamos n passos para percorrer todos os itens e, para cada item, executamos $c + 1$ passos para percorrer todas as cargas máximas. Dessa forma, a complexidade do algoritmo é $\boldsymbol{O(n \cdot c)}$.

2.5 PROBLEMA DO TROCO (*COIN CHANGE*)

Problema: Em um país fictício, há n tipos distintos de moedas, sendo os valores de cada um deles representados por $c(1)$, $c(2)$, $c(3)$, ..., $c(n)$. No Brasil, por exemplo, n seria igual a 6 e os valores de c seriam 1, 5, 10, 25, 50 e 100 (1 real). Dado um valor k, deseja-se saber qual a quantidade mínima de moedas que devem ser utilizadas para obter uma soma de valores igual a k.

Por exemplo, utilizando o sistema de moedas brasileiro e considerando $k = 127$, a solução seria utilizar 4 moedas: $1 + 1 + 25 + 100$.

A primeira solução que pode passar pela cabeça é utilizar um algoritmo guloso, escolhendo sempre as moedas de maior valor possível. Entretanto, embora essa abordagem funcione para o sistema brasileiro, ela não funciona para qualquer sistema de moedas. Imagine, por exemplo, que tenhamos 3 tipos de moedas, com valores 1, 7, 10, e desejamos escolhê-las de forma a somar o valor $k = 14$. Utilizando a estratégia gulosa, obteríamos uma resposta com 5 moedas: $1+1+1+1+10$, enquanto que a solução ótima é composta de 2 moedas: $7 + 7$.

Outra abordagem natural de se pensar é a solução trivial, gerando todos os conjuntos possíveis com m moedas, começando com $m = 1$ e incrementando m até encontrar o número mínimo de moedas para o qual exista um conjunto cuja soma seja igual a k. Se esse conjunto não for encontrado até chegarmos em $m = k$, então não há solução. Esse algoritmo, embora produza o resultado correto, possui uma complexidade exponencial em k.

Por fim, também podemos utilizar programação dinâmica para construir a solução deste problema. A ideia é manter um vetor de $k + 1$ posições que, para cada posição j, armazenará a menor quantidade de moedas que podemos utilizar para obter uma soma igual a j, podendo utilizar apenas tipos de moedas pertencentes a algum subconjunto dos n tipos dados no enunciado. Isso quer dizer que, antes do final da execução do algoritmo, o vetor não possuirá a resposta para o problema, mas sim para algum subproblema que consiste em poder utilizar menos tipos de moedas.

O algoritmo consiste em considerar, inicialmente, um conjunto vazio de tipos de moedas possíveis de serem utilizados e, a cada iteração, adicionar um tipo novo a esse conjunto e atualizar os valores do vetor, até que todos os tipos de moedas possam ser utilizados e, dessa forma, tenhamos a solução ótima armazenada no vetor.

Considere que os tipos de moedas estejam numerados de 1 até n e que dp seja o vetor com $k + 1$ posições citado nos parágrafos anteriores. Inicialmente, $dp(0)$ deve ser igual a 0, enquanto todas as outras posições de $dp(j)$ devem ser iguais a infinito para $j > 0$, pois não é possível formar nenhuma soma de valores não-nula sem poder utilizar algum tipo de moeda. Depois, devemos, para cada tipo i de moeda, percorrer todo o vetor dp de 1 a k atualizando o valor em $dp(j)$ considerando 2 cenários:

1. Não alteramos o valor de $dp(j)$, pois a solução anterior era válida para um conjunto menor de tipos de moedas possíveis, logo ela continua válida

2. Utilizamos o tipo i de moeda para diminuir o valor de $dp(j)$. Como é possível usar uma quantidade qualquer de moedas de cada tipo para construir a solução, podemos adicionar uma moeda do tipo i a uma resposta já conhecida armazenada em $dp(j - c(i))$, que pode já considerar moedas desse tipo, pois essa posição foi atualizada antes da posição j. Dessa forma, o valor de $dp(j)$ seria atualizado para $dp(j - c(i)) + 1$, que equivale a considerar a solução do subproblema cuja soma de moedas é igual a $j - c(i)$ acrescida de uma moeda, que está sendo adicionada nesse momento

Para a ideia do algoritmo ficar mais clara, vamos simular sua execução em um exemplo:

Considere que haja 5 tipos de moedas de valores iguais a 1, 2, 5, 6, 11 e que desejamos encontrar a menor quantidade de moedas necessárias para formar o valor 13.

Primeiramente, consideremos o caso em que só uma moeda é permitida para formar os valores. Como a primeira moeda possui valor 1, para cada soma j que tentamos formar, serão necessárias exatamente j moedas. Dessa forma, considerando que o rótulo M1 indica que a moeda 1 está sendo considerada, M2 indica o mesmo para as moedas 1 e 2, M3 para as moedas 1, 2 e 3, e assim por diante, a tabela de respostas será, inicialmente, a seguinte:

Quadro 8 – Resolução do problema do troco considerando 1 moeda

Valor	0	1	2	3	4	5	6	7	8	9	10	11	12	13
M1	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Considerando agora a moeda de valor 2. Para as somas 0 e 1, não podemos utilizar esta moeda ainda, então o valor anterior é mantido. Na soma 2, podemos reduzir a quantidade de moedas utilizadas considerando a nova resposta igual a 1, que equivale a adicionar a moeda de valor 2 na solução que somava 0. Na soma 3, a nova resposta fica igual a 2, que equivale a adicionar a nova moeda na solução da soma igual a 1. Na soma 4, a nova resposta também fica igual a 2, pois a resposta para a soma 2 considerando a nova moeda foi reduzida para 1. Nesse caso, podemos adicionar nossa moeda na solução de soma 2, totalizando uma soma 4 e deixando a resposta igual a 2, que equivale à resposta anterior,

1, acrescida de 1, que seria a nova moeda adicionada. O raciocínio é o mesmo para todas as outras somas da tabela. O resultado fica assim:

Quadro 9 – Resolução do problema do troco considerando 2 moedas

Valor	0	1	2	3	4	5	6	7	8	9	10	11	12	13
M1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
M2	0	1	1	2	2	3	3	4	4	5	5	6	6	7

Vamos agora atualizar os valores considerando também a moeda de valor 5. Para as somas de valor menor ou igual a 4, não há como utilizar essa moeda. Entretanto, podemos utilizá-la para formar a soma 5, que antes necessitava de 3 moedas, com apenas 1 moeda. Todas as somas maiores também terão o valor de quantidade de moedas reduzido em comparação ao anterior. A soma 12, por exemplo, poderá ser obtida com 3 moedas, pois é a combinação da nova moeda com a solução da soma 7, que, no momento de atualização da soma 12, já estará com seu valor de quantidade de moedas atualizado para 2, que equivale à combinação da nova moeda com a solução de soma 2.

Quadro 10 – Resolução do problema do troco considerando 3 moedas

Valor	0	1	2	3	4	5	6	7	8	9	10	11	12	13
M1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
M2	0	1	1	2	2	3	3	4	4	5	5	6	6	7
M3	0	1	1	2	2	1	2	2	3	3	2	3	3	4

Aplicando a mesma ideia dos casos acima, podemos construir os vetores com as soluções que consideram, respectivamente, as moedas até o índice 4 e até o índice 5. Lembrando que a moeda de índice 4 possui valor igual a 6 e a de índice 5 possui valor igual a 11.

Quadro 11 – Resolução do problema do troco considerando 5 moedas

Valor	0	1	2	3	4	5	6	7	8	9	10	11	12	13
M1	0	1	2	3	4	5	6	7	8	9	10	11	12	13
M2	0	1	1	2	2	3	3	4	4	5	5	6	6	7
M3	0	1	1	2	2	1	2	2	3	3	2	3	3	4
M4	0	1	1	2	2	1	1	2	2	3	2	2	2	3
M5	0	1	1	2	2	1	1	2	2	3	2	1	2	2

Dessa forma, é possível construir um algoritmo genérico para resolver esse problema dado um conjunto qualquer de moedas e um valor qualquer de k. Para cada tipo de moeda i (sendo a primeira moeda a de índice igual a 1) e para cada soma de valores j, o valor no vetor dp será o seguinte:

- $dp(j) = 0$, se $i = 0$ e $j = 0$

- $dp(j) = \infty$, se $i = 0$ e $j > 0$
- $dp(j) = \min(dp(j), dp(j - c(i)) + 1)$, se $i > 0$ e $j \geq c(i)$

Uma possível implementação *bottom-up* para este algoritmo é a seguinte:

Código 10 – Problema do Troco *bottom-up*

```
dp[0] = 0; // Formar soma 0 com 0 moeda
for(int j = 1; j <= K; j++)
        dp[j] = INF; // Impossivel formar soma >0 com 0 moeda

// Primeira moeda tem indice 0 porque eh um vetor
for(int i = 0; i < N; i++) {
        for(int j = c[i]; j <= K; j++) {
                // Tenta melhorar o valor de dp[j] adicionando a
                // moeda de valor c[i] na solucao de soma j-c[i]
                dp[j] = min(dp[j], dp[j-c[i]] + 1);
        }
}
```

Uma outra abordagem possível para este problema utilizando programação dinâmica é inverter a lógica de preenchimento do vetor *dp* em comparação à solução anterior. Na primeira solução, foram atualizadas todas as posições (possíveis somas de valores de moedas) do vetor para cada tipo de moeda fixado, até que isso tenha sido aplicado a todos. Baseado no algoritmo mostrado em https://www.codesdope.com/course/algorithms-coin-change (Acessado em 13/01/2021), a segunda abordagem propõe encontrar a solução ótima para cada possível soma de moedas, considerando todos os tipos de moedas possíveis para cada posição fixada no vetor, até que todas as posições tenham sido atualizadas.

Vale ressaltar que essa segunda solução está sendo apresentada para demonstrar um outro raciocínio possível para resolver o problema. Entretanto, sua complexidade é igual à da primeira.

Na execução desse algoritmo, começaremos encontrando as soluções ótimas primeiramente para somas de moedas menores e, então, encontraremos as soluções para somas cada vez maiores, aproveitando as respostas já encontradas das menores. Para cada soma de moedas possível j, tentaremos minimizar a quantidade de moedas necessárias para formá-la utilizando todos os tipos i de moedas dados no problema. Os 2 cenários existentes para cada atualização da resposta para uma posição no vetor são os seguintes:

1. Não utilizaremos a moeda i, logo não atualizamos a resposta em $dp(j)$
2. Utilizaremos a moeda i, logo atualizamos o valor de $dp(j)$ com o tipo de moeda i. Como todos os valores de $dp(0)$, $dp(1)$, ..., $dp(j-1)$ já correspondem às soluções ótimas desses subproblemas, a melhor forma de minimizar a resposta em $dp(j)$ utilizando a moeda de tipo i é atualizando esse valor para $dp(j - c(i)) + 1$, que é

a quantidade mínima de moedas para atingir soma igual a $j - c(i)$ adicionada de 1, que corresponde à moeda de tipo i que está sendo adicionada. A solução em $dp(j - c(i))$ pode já considerar a utilização de outra moeda do tipo i, o que não é um problema, pois mais de uma moeda do mesmo tipo pode ser usada

Assim como foi feito na primeira solução, para cada iteração de soma e tipo de moedas, deve-se avaliar, dentre os cenários existentes, quais são possíveis e qual deles minimiza a resposta do subproblema que estamos resolvendo.

Para que a ideia do algoritmo fique mais clara, vamos simular sua execução para o mesmo exemplo feito anteriormente: um conjunto de 5 tipos de moedas igual a 1, 2, 5, 6, 11 e objetivo de descobrir qual a menor quantidade de moedas necessária para atingir uma soma de valores $k = 13$.

Primeiramente, o valor de $dp(0)$ deve ser 0, pois nenhuma moeda precisa ser utilizada para somar 0. Para $dp(1)$, o único tipo de moeda que podemos utilizar é o de valor 1, logo $dp(1) = dp(0) + 1 = 1$. Para $dp(2)$, podemos utilizar o tipo de moeda de valor 1, aproveitando o valor já calculado de $dp(1)$, sendo $dp(2) = dp(1) + 1 = 2$, entretanto, também podemos utilizar o tipo de moeda de valor 2, fazendo com que $dp(2) = dp(0)+1 = 1$, sendo esta a resposta mínima. Nesse momento, o vetor dp estaria preenchido conforme ilustrado no Quadro 12.

Quadro 12 – Resolução alternativa do problema do troco para somas até 2

Valor	0	1	2	3	4	5	6	7	8	9	10	11	12	13
dp	0	1	1	*	*	*	*	*	*	*	*	*	*	*

Tanto a moeda de valor 1 quanto a de valor 2 produzirão a mesma resposta para $dp(3)$: 2 moedas. Já para $dp(4)$, utilizamos a moeda de valor 1 para atualizar sua resposta para $dp(4) = dp(3) + 1 = 3$, mas atualizamos logo em seguida utilizando a moeda de valor 2, terminando com $dp(4) = dp(2) + 1 = 2$. Para $dp(5)$, há mais 1 opção de tipo de moeda, então atualizamos primeiramente com o tipo de moeda de valor 1, fazendo $dp(5) = dp(4) + 1 = 3$, não atualizaremos com a moeda de valor 2, pois a resposta seria a mesma de anteriormente, visto que $dp(3) + 1 = 3$ e, por fim, atualizaremos com a moeda de valor 5, chegando na solução ótima: $dp(5) = dp(0) + 1 = 1$. Dadas essas soluções, o vetor dp, até esse momento, estaria da seguinte forma:

Quadro 13 – Resolução alternativa do problema do troco para somas até 5

Valor	0	1	2	3	4	5	6	7	8	9	10	11	12	13
dp	0	1	1	2	2	1	*	*	*	*	*	*	*	*

O raciocínio para o preenchimento do restante do vetor é o mesmo aplicado acima. Fixamos cada posição ainda não preenchida e tentamos achar a menor resposta possível,

combinando cada um dos tipos de moedas com as soluções armazenadas nas posições de índices menores de *dp*. Por fim, o estado final do vetor *dp* está descrito no Quadro 14.

Quadro 14 – Resolução alternativa do problema do troco para somas até 13

Valor	0	1	2	3	4	5	6	7	8	9	10	11	12	13
dp	0	1	1	2	2	1	1	2	2	3	2	1	2	2

Uma possível implementação *bottom-up* desse algoritmo está no Código 11.

Código 11 – Problema do Troco *bottom-up* - Solução Alternativa

```
dp[0] = 0; // Formar soma 0 com 0 moeda

for(int j = 1; j <= K; j++) {
        dp[j] = INF; // Ainda nao utilizou nenhuma moeda
        for(int i = 0; i < N; i++) {
                // Verifica se eh possivel utilizar a moeda
                if(c[i] <= j) {
                        // Tenta melhorar o valor de dp[j]
                        // adicionando a moeda de valor c[i]
                        // na solucao de soma j-c[i]
                        dp[j] = min(dp[j], dp[j-c[i]] + 1);
                }
        }
}
```

É possível notar a semelhança entre esse código e o da solução anterior. A maior diferença entre as duas implementações é que a ordem dos *loops* nas posições do vetor (índice j) e nos tipos de moedas (índice i) foi trocada. Na implementação anterior, o *loop* mais externo percorria os tipos de moedas, mas nessa o mais externo percorre os índices do vetor *dp*.

Complexidade

Os dois algoritmos apresentados possuem a mesma complexidade. No primeiro, passamos por todos os n possíveis tipos de moedas e, para cada um, passamos por todas as k possíveis somas de valores. No segundo algoritmo, a análise é análoga. Dessa forma, a complexidade dos algoritmos apresentados é $\boldsymbol{O(n \cdot k)}$.

3 PROBLEMAS BÁSICOS COM *STRINGS*

Neste capítulo, serão abordados três problemas envolvendo *strings*. Todos os três problemas pedem algum tipo de comparação entre duas *strings* para obter alguma informação, como a maior *substring* em comum ou o custo total de transformar uma *string* em outra utilizando certas operações com custo definido.

Cada um dos seguintes problemas terá uma solução diferente, mas todas utilizarão uma base semelhante: dividir o problema em subproblemas que consideram apenas prefixos das *strings*.

3.1 MAIOR SUBSEQUÊNCIA COMUM (MSC)

Problema: Dadas duas *strings* A e B, respectivamente de tamanhos n e m, deseja-se saber qual é a maior subsequência comum entre elas.

Dizemos que uma *string*[1] C é uma subsequência de outra *string* A se, e somente se, todos os caracteres de C ocorrerem em A na mesma ordem. Por exemplo, "ABCA" é subsequência de "AXBYCZA", mas não é subsequência de "ABXYZA". Além disso, uma *string* C é uma subsequência comum entre duas *strings* A e B se, e somente se, ela for subsequência de A e também de B.

Por exemplo, se $A =$ "XYZK" e $B =$ "YLOZPXK", temos que "XK" é uma subsequência comum entre A e B, entretanto, a maior subsequência comum é "YZK".

Primeiramente, vamos resolver o problema de apenas encontrar o tamanho da maior subsequência comum e depois aplicar uma pequena ideia adicional para a reconstrução da solução para descobrir qual é essa subsequência de fato.

Uma solução trivial seria gerar todas as subsequências de A e de B e compará-las até encontrar, dentre as que fossem comuns entre as duas *strings*, a de maior tamanho. No entanto, essa solução é exponencial no tamanho das *strings* da entrada, pois uma *string* de tamanho n possui 2^n subsequências possíveis[2].

Podemos, no entanto, obter um algoritmo com complexidade bem melhor utilizando a técnica da programação dinâmica. A ideia dessa solução se baseará em calcular a maior subsequência comum para todos os pares de prefixos das strings A e B dadas na entrada. Para o algoritmo que será detalhado abaixo, um *prefixo* é um pedaço da *string* original formado pelos k primeiros caracteres dessa *string*, sendo k um valor qualquer entre 0 e o tamanho total da *string*. Dessa forma, é possível que o prefixo seja vazio, no caso em que

[1] Embora essas definições estejam sendo dadas no contexto de *strings*, elas também podem ser aplicadas a subsequências em geral, inclusive o conceito de Maior Subsequência Comum

[2] A quantidade de subsequências possíveis não é 2^n caso haja caracteres repetidos, mas seu cálculo exato foge do escopo deste trabalho

k é 0, ou então que seja igual à *string* original, no caso em que k é igual ao tamanho da *string*.

Como foi feito nos problemas apresentados anteriormente, utilizaremos as soluções dos subproblemas menores, que serão as respostas para **prefixos** de tamanho menor, para construir a solução de subproblemas maiores, até que esteja resolvido o problema original. Consideremos que, para cada subproblema, tenhamos dois valores, i e j, que correspondem aos tamanhos dos prefixos de A e B que estão sendo comparados, respectivamente, e que ao resolver um subproblema para dados i e j, já sabemos a solução para todos os pares de tamanhos de prefixos menores que i e j.

Os casos base do problema serão quando um prefixo for vazio, ou seja, quando $i = 0$ ou $j = 0$. Nesses casos, a maior subsequência comum deverá ser uma *string* vazia e seu tamanho será igual a 0.

Nos demais casos, para valores de i e j maiores que 0, teremos 2 cenários a considerar:

1. Não há uma correspondência nas posições i e j das strings. Em outras palavras, $A(i)$ é diferente de $B(j)$. Nesse caso, devemos considerar que a maior subsequência comum até as posições i e j corresponde a alguma outra encontrada anteriormente em posições menores. Como desejamos manter a resposta igual ao máximo obtido até então, devemos escolher o valor de uma posição imediatamente anterior à atual, que pode ser $dp(i-1, j)$ ou $dp(i, j-1)$. Dessa forma, nossa resposta será o máximo entre essas duas.

2. Há uma correspondência nas posições i e j, ou seja, $A(i) = B(j)$. Nesse caso, nossa resposta será igual à melhor resposta obtida desconsiderando essas posições, ou seja, $dp(i-1, j-1)$, acrescida de 1, que corresponde à contagem da correspondência $A(i)$ com $B(j)$.

Vale notar que não é necessário escolher o máximo entre os dois cenários, pois se houver correspondência das posições i e j, o melhor sempre será escolher o cenário 2, visto que $dp(i-1, j-1) + 1$ sempre será maior ou igual a $dp(i-1, j)$ e $dp(i, j-1)$.

Para que a ideia do algoritmo fique mais clara, vamos utilizar um exemplo. As entradas serão as seguintes: $A =$ "XYZK" e $B =$ "YLOZPXK". Podemos entender este algoritmo como o preenchimento de uma tabela $(n+1) \times (m+1)$. Esse preenchimento será feito da esquerda para a direita e de cima para baixo, sendo cada linha representada por um caractere de A e cada coluna por um caractere de B.

Primeiramente, temos que a primeira linha ($i = 0$) corresponde aos casos base, então será toda preenchida com 0, assim como ocorrerá para os valores da primeira coluna ($j = 0$). Para $i = 1$, não haverá correspondência até chegarmos em $j = 6$, que será o momento em que os seguintes prefixos "X" e "YLOZPX" serão comparados. Nesse caso, a resposta deve ser igual a 1, pois é a primeira correspondência. A posição $(1, 7)$ deverá ser preenchida com 1, pois é a maior resposta dentre as das posições anteriores $(0, 7)$ e $(1, 6)$.

Após isso, teremos outra correspondência na posição $(2, 1)$, que corresponde à comparação de "XY" com "Y". Sua resposta será 1, que é a correspondência da letra "Y" nas *strings*. Todo o resto da linha será preenchido com 1.

Na linha 3, a resposta será 0 para o caso base ($j = 0$) e 1 para os valores de j até 3, pois as posições $(i - 1, j)$ serão iguais à 1 devido à correspondência encontrada com a letra "Y" anteriormente. Por exemplo, na posição $(3, 1)$, que corresponde à comparação de "XYZ" com "Y", será igual a 1 porque, mesmo que não haja correspondência nesses índices, a resposta em $(2, 1)$, que representa a comparação de "XY" com "Y", é igual a 1, o que denota que já havia ocorrido 1 correspondência. Já na posição $(3, 4)$ há uma nova correspondência. Dessa forma, a resposta para estes índices será igual a $dp(2, 3) + 1 = 2$. Como a posição $(2, 3)$ representa as strings "XY" e "YLO", a correspondência dos caracteres "Y" já foi contada para o cálculo do seu valor, então a posição $(3, 4)$, que representa as strings "XYZ" e "YLOZ" considera essa correspondência do "Y" contada anteriormente mais a correspondência do "Z" contada neste momento. O resto da linha será preenchida com o valor 2.

Utilizando o mesmo raciocínio descrito acima, preenchemos a última linha, terminando com a situação retratada no Quadro 15.

Quadro 15 – Maior Subsequência Comum para cada par de índices

	0	1 (Y)	2 (L)	3 (O)	4 (Z)	5 (P)	6 (X)	7 (K)
0	0	0	0	0	0	0	0	0
1 (X)	0	0	0	0	0	0	1	1
2 (Y)	0	1	1	1	1	1	1	1
3 (Z)	0	1	1	1	2	2	2	2
4 (K)	0	1	1	1	2	2	2	3

Para poder reconstruir a solução, podemos armazenar também, para cada par de índices da tabela, um registro de qual foi a posição anterior cujo valor foi utilizado para determinar seu valor. Por exemplo, para encontrar o valor de $dp(3, 4)$, utilizamos o valor de $dp(2, 3)$ somado a 1, então na posição $(3, 4)$ deve-se indicar, de alguma forma, que seu valor foi determinado a partir do valor na posição $(2, 3)$. Já no caso de $dp(2, 4)$, o valor foi determinado a partir de $dp(2, 3)$, então deve-se indicar que a posição utilizada para encontrar o valor em $(2, 4)$ foi $(2, 3)$.

Esse procedimento é importante para sabermos, para cada subproblema, qual era o subproblema anterior e qual ação foi tomada para chegar na solução, como ter considerado a última letra como parte da MSC ou não. Para que fique mais claro, o Quadro 16 foi preenchido com os mesmos valores do anterior, mas agora também com um rastro em cada célula de qual foi a posição anterior cujo valor foi utilizado para determiná-la. No caso de empate na escolha entre a célula acima ou à esquerda, está sendo utilizada a da esquerda. A inspiração para a representação do rastro utilizada nesse quadro foi retirada de (CORMEN et al., 2009).

Quadro 16 – Reconstrução da solução para encontrar a maior subsequência comum

	0	1 (Y)	2 (L)	3 (O)	4 (Z)	5 (P)	6 (X)	7 (K)
0	0	$\leftarrow 0$	$\leftarrow 0$	$\leftarrow 0$	$\leftarrow 0$	$\leftarrow 0$	$\leftarrow 0$	$\leftarrow 0$
1 (X)	$\uparrow 0$	$\leftarrow 0$	$\leftarrow 0$	$\leftarrow 0$	$\leftarrow 0$	$\leftarrow 0$	$\nwarrow 1$	$\leftarrow 1$
2 (Y)	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\leftarrow 1$
3 (Z)	$\uparrow 0$	$\uparrow 1$	$\uparrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\leftarrow 2$	$\leftarrow 2$
4 (K)	$\uparrow 0$	$\uparrow 1$	$\uparrow 1$	$\uparrow 1$	$\uparrow 2$	$\leftarrow 2$	$\leftarrow 2$	$\nwarrow 3$

Dado esse quadro, podemos reconstruir a solução partindo da última posição, que corresponde à solução do problema original, e seguindo o caminho que passa pelos subproblemas até chegar no índice $(0, 0)$.

Para reconstruir a solução do exemplo dado, seria feito o seguinte: na última posição, que corresponde aos índices $(4, 7)$, temos uma indicação que o subproblema anterior estava na diagonal, ou seja, para a resposta deste problema, a letra "K" foi considerada como parte da MSC, logo, a MSC deve terminar com a letra "K". Então aplicamos a mesma ideia para a posição $(3, 6)$, que corresponde ao subproblema anterior. Ela nos leva ao subproblema à esquerda, na posição $(3, 5)$, que aponta para a posição $(3, 4)$, indicando que nenhum caractere novo foi adicionado à MSC entre esses subproblemas. Na posição $(3, 4)$, a indicação é de que a resposta veio a partir de uma posição na diagonal, então o subproblema anterior está representado na posição $(2, 3)$ e a letra "Z" foi adicionada à MSC na transição de $(2, 3)$ para $(3, 4)$. Como a reconstrução é feita de trás para frente, a letra "Z" deve vir antes da letra "K", então nosso sufixo será "ZK".

Continuando com a ideia, a posição $(2, 3)$ nos leva à $(2, 2)$, que nos leva à $(2, 1)$. Todas sem adição de caracteres. A posição $(2, 1)$ aponta para $(1, 0)$, que está na diagonal, indicando que o sufixo da MSC agora é "YZK". Por fim, a posição $(1, 0)$ aponta para cima, levando ao par de índices $(0, 0)$ e finalizando o algoritmo. Dessa forma, a MSC encontrada corresponde à *string* "YZK".

O Código 12 é uma possível implementação para o algoritmo descrito acima.

Código 12 – Maior Subsequência Comum *bottom-up*

```
int dp[N+1][M+1];
pair<int,int> trace[N+1][M+1];
for(int i = 0; i <= N; i++) {
        for(int j = 0; j <= M; j++) {
                if(i == 0 || j == 0) {
                        dp[i][j] = 0;
                }
                // Os indices de A e B foram subtraidos de 1
                // porque as strings sao indexadas em 0
                else if(A[i-1] == B[j-1]) {
                        dp[i][j] = dp[i-1][j-1] + 1;
                        trace[i][j] = make_pair(i-1, j-1);
                }
                else if(dp[i-1][j] > dp[i][j-1]) {
                        dp[i][j] = dp[i-1][j];
                        trace[i][j] = make_pair(i-1, j);
                }
                else {
                        dp[i][j] = dp[i][j-1];
                        trace[i][j] = make_pair(i, j-1);
                }
        }
}
```

Para recuperar a *string* correspondente à MSC, conforme descrito acima, podemos percorrer o caminho armazenado em *trace* a partir da posição final da tabela até chegar em algum caso base. Uma implementação possível é a do Código 13.

Código 13 – Reconstrução da solução para encontrar a maior subsequência comum

```
#define st first
#define nd second

string MSC = "";
pair<int,int> caminho = make_pair(N, M);
while(caminho.st != 0 && caminho.nd != 0) {
        pair<int,int> anterior = trace[caminho.st][caminho.nd];
        if(anterior == make_pair(caminho.st-1, caminho.nd-1)) {
                MSC = A[caminho.st-1] + MSC;
        }
        caminho = anterior;
}
```

Complexidade

As etapas de obtenção da solução do tamanho da maior subsequência comum e da construção do caminho para a reconstrução da solução envolvem preencher uma matriz $(n+1) \times (m+1)$, realizando um número constante de operações em cada célula. Dessa forma, essa parte do algoritmo possui complexidade $O(n \cdot m)$.

A etapa de reconstrução da solução, por sua vez, pode ser realizada em complexidade $O(n+m)$, pois o número de iterações será, no máximo, igual à distância de Manhattan entre a posição inicial e final da tabela, visto que cada iteração se aproxima da posição inicial em ao menos 1 unidade. Embora a implementação apresentada aqui não tenha essa complexidade devido à forma que foram feitas concatenações para construção da *string* MSC, é possível obtê-la fazendo essa construção de forma mais otimizada.

Dessa forma, a complexidade final do algoritmo é $\boldsymbol{O(n \cdot m)}$.

3.2 MAIOR *SUBSTRING* COMUM

Problema: Dadas duas *strings* A e B, respectivamente de tamanhos iguais a n e m, deseja-se saber qual é a maior *substring* comum entre elas.

Dizemos que uma *string* C é *substring* de outra *string* A se todos os caracteres de C estiverem em A na mesma ordem e de forma contínua, em outras palavras, se é possível retirar um número qualquer de caracteres (inclusive 0) do prefixo e do sufixo de A de modo que a *string* restante seja igual a C.

Uma *string* C é uma *substring* comum entre duas *strings* A e B se C for *substring* de A e de B.

Por exemplo, se A = "ACBCAB" e B = "ABACABA", temos que "AC" é uma *substring* comum entre A e B, entretanto, a maior *substring* comum é "CAB".

Assim como feito no problema da maior subsequência comum, será apresentado primeiro um algoritmo para encontrar apenas o tamanho da maior *substring* comum e, após isso, mostraremos como reconstruir a solução.

Como sempre, há a solução trivial, que consiste em gerar todas as *substrings* de A e B e escolher, dentre as que ocorrerem tanto em A quanto em B, a de maior tamanho. Entretanto, a implementação das otimizações para essa ideia não será simples caso deseje-se obter uma complexidade baixa.

A solução utilizando programação dinâmica consiste em construir uma matriz dp de dimensões $(n+1) \times (m+1)$ onde cada valor de $dp(i, j)$ indica o tamanho do maior sufixo comum entre os prefixos $A(1, \ldots, i)$ e $B(1, \ldots, j)$, ou seja, caso o sufixo tenha tamanho k, ele deve ser igual a $A(i-k+1, \ldots, i)$ e $B(j-k+1, \ldots, j)$. A importância de cada subproblema armazenar o tamanho do sufixo, e não de uma *substring* qualquer, é que assim é possível encontrar a solução para um subproblema utilizando a solução de um

subproblema anterior, pois um sufixo de $A(1, \ldots, i-1)$ concatenado com $A(i)$ forma um sufixo de $A(1, \ldots, i)$. A lógica é análoga para a *string* B.

Como a maior *substring* comum é um sufixo comum de algum prefixo de A com algum prefixo de B, visto que a *substring* precisa terminar em algum par de índices de A e B, então é possível determinar o tamanho da maior *substring* comum encontrando os maiores sufixos comuns para cada par de prefixos e escolhendo, dentre todos, o maior. Em outras palavras, será o maior valor da matriz dp.

Dessa forma, há 2 cenários a se considerar na hora de preencher algum valor em dp:

1. $A(i) = B(j)$, ou seja, há uma correspondência no par de índices (i, j). Como $dp(i-1, j-1)$ é o maior sufixo comum entre $A(1, ..., i-1)$ e $B(1, ..., j-1)$, então $dp(i, j) = dp(i-1, j-1) + 1$, pois o caractere $A(i)$, que é igual a $B(j)$, será contabilizado no fim do sufixo encontrado em $dp(i-1, j-1)$, formando o maior sufixo comum entre $A(1, \ldots, i)$ e $B(1, ..., j)$

2. $A(i) \neq B(j)$. Não há sufixo comum de tamanho não-nulo entre $A(1, \ldots, i)$ e $B(1, \ldots, j)$, logo $dp(i, j) = 0$

Para exemplificar o algoritmo, vamos explicar a lógica de sua execução para o caso em que temos as seguintes *strings*: $A =$ "ACBCAB" e $B =$ "ABACABA".

Consideraremos que as *strings* estão indexadas a partir de 1, ou seja, o primeiro caractere da *string* A é $A(1)$. Dessa forma, os casos bases ocorrerão em $dp(i, 0)$ e $dp(0, j)$ para quaisquer i e j, que é quando um dos dois índices é igual a 0. Nesses casos, nenhum caractere ainda está sendo considerado, logo seus valores devem ser iguais a 0.

Para $i = 1$, teremos correspondência para valores de j iguais a 1, 3, 5 e 7. Essas posições deverão ter valor igual a 1 e as demais terão valor igual a 0. Isso ocorre porque o prefixo de A que está sendo comparado é apenas a primeira letra "A"[3], já os prefixos de B serão, respectivamente para cada valor de j citado anteriormente, "A", "ABA", "ABACA" e "ABACABA". A correspondência nesses índices ocorre porque esses prefixos terminam em "A".

Para $i = 2$, a única correspondência que ocorrerá será para $j = 4$, pois os prefixos que serão comparados nesse par de índices serão "AC" e "ABAC". É possível reparar que na comparação do par de índices $(i-1, j-1)$, que é $(1, 3)$ neste caso, os prefixos comparados eram "A" e "ABA", cujo maior sufixo comum tinha tamanho 1. Quando adicionamos $A(2)$ e $B(4)$, o caractere "C" aparece no fim dos dois prefixos, fazendo com que o maior sufixo comum para esse par de índices seja igual ao anterior somado de 1, que corresponde ao caractere "C", logo $dp(2, 4) = dp(1, 3) + 1 = 2$. Para os demais valores de j, não há correspondência, logo o valor em dp é igual a 0.

[3] O sublinhado nas *strings* indica o sufixo em comum encontrado

Para $i = 3$, há 2 correspondências: em $j = 2$, pois os prefixos comparados são "AC<u>B</u>" e "A<u>B</u>", e em $j = 6$, com os prefixos "AC<u>B</u>" e "ABACA<u>B</u>". Nesses dois casos, o valor armazenado em *dp* será igual a 1. Para os demais valores de j, a resposta será 0.

Seguindo a lógica aplicada nos casos acima, é possível preencher o resto dos valores, terminando com a tabela de *dp* representada pelo Quadro 17:

Quadro 17 – Maiores sufixos comuns entre prefixos de ABACABA e ACBCAB

	0	1 (A)	2 (B)	3 (A)	4 (C)	5 (A)	6 (B)	7 (A)
0	0	0	0	0	0	0	0	0
1 (A)	0	1	0	1	0	1	0	1
2 (C)	0	0	0	0	2	0	0	0
3 (B)	0	0	1	0	0	0	1	0
4 (C)	0	0	0	0	1	0	0	0
5 (A)	0	1	0	1	0	2	0	1
6 (B)	0	0	2	0	0	0	3	0

O maior valor da tabela é $dp(6, 6)$, que é igual a 3, logo este é o tamanho da maior *substring* comum entre as *strings* "ACB<u>CAB</u>" e "ABA<u>CAB</u>A".

Reconstruir a maior *substring* comum é bastante simples a partir da tabela acima. Como cada valor na tabela indica o maior sufixo comum que utiliza os caracteres naquele par de índices, então a maior *substring* comum será composta pelos 3 últimos caracteres das *substrings* que estavam sendo comparadas naquele par de índices, ou seja, "ACB<u>CAB</u>" ou "ABA<u>CAB</u>". De forma geral, após descobrir que o tamanho da maior *substring* comum é igual a k e que este valor ocorre no par de índices (x, y), também descobrimos que a maior *substring* comum é igual a $A(x - k + 1, x)$, assim como $B(y - k + 1, y)$, que terá o mesmo valor. Nos casos em que todos os valores da tabela são 0, pois nenhuma correspondência foi encontrada, a maior *substring* comum será uma *string* vazia.

De forma resumida, o algoritmo consiste em aplicar a seguinte lógica para cada par de i e j, sendo $0 \leq i \leq n$ e $0 \leq j \leq m$:

- $dp(i, j) = 0$, se $i = 0$ ou $j = 0$
- $dp(i, j) = 0$, se $i > 0$, $j > 0$ e $A(i) \neq B(j)$
- $dp(i, j) = dp(i - 1, j - 1) + 1$, se $i > 0$, $j > 0$ e $A(i) = B(j)$

Em cada passo, armazenamos o par de índices onde está guardado o maior valor encontrado até o momento. No final, o par de índices com o maior valor de toda a tabela possuirá o tamanho da maior *substring* comum e também pode ser utilizado para recuperá-la.

Uma possível implementação para o algoritmo é a do Código 14:

Código 14 – Maior *Substring* Comum - *bottom up*

```
int N = A.size(), M = B.size();
int dp[N+1][M+1];
int max_i = 0, max_j = 0;

for(int i = 0; i <= N; i++) {
        for(int j = 0; j <= M; j++) {
                if(i > 0 && j > 0 && A[i-1] == B[j-1]) {
                        dp[i][j] = dp[i-1][j-1] + 1;
                        if(dp[i][j] > dp[max_i][max_j]) {
                                /* Atualiza o par de indices que
                                armazena a maior resposta */
                                max_i = i;
                                max_j = j;
                        }
                }
                else {
                        dp[i][j] = 0;
                }
        }
}
```

Para reconstruir a solução, basta pegar a *substring* de A que comece no índice $max\_i - dp(max\_i, max\_j) + 1$ e possua tamanho igual a $dp(max\_i, max\_j)$. A implementação pode ser a demonstrada no Código 15:

Código 15 – Reconstrução da solução para encontrar a maior *substring* comum

```
string long_comm_substr;
/* Foi retirado o +1 do indice porque a
string eh indexada a partir de 0 */
long_comm_substr = A.substr(max_i - dp[max_i][max_j], dp[max_i][max_j]);
```

Complexidade

O algoritmo baseia-se em percorrer uma tabela $(n+1) \times (m+1)$ realizando um número constante de operações em cada célula, tendo complexidade de $O(n \cdot m)$. Já a reconstrução tem complexidade de $O(n+m)$, devido ao tamanho máximo da maior *substring* comum.

Dessa forma, a complexidade final do algoritmo é $\boldsymbol{O(n \cdot m)}$.

3.3 DISTÂNCIA DE EDIÇÃO

Problema: Dadas duas *strings* A e B, respectivamente de tamanhos n e m, deseja-se realizar uma quantidade de operações em A para transformá-la

em B, tal que a soma do custo dessas operações seja mínimo. As operações possíveis são:

- **Substituir um caractere da *string* por qualquer outro a custo C_{sub}**
- **Remover um caractere em qualquer posição da *string* a custo C_{rem}**
- **Inserir um caractere em qualquer posição da *string* a custo C_{adc}**

Qual o custo mínimo para transformar a *string* A em B?

Esse problema parece complicado à primeira vista devido à quantidade de operações permitidas. Uma ideia baseada em força bruta não seria viável, pois aplicar todas as operações em todos os caracteres da *string* resultaria em uma complexidade de tempo exponencial. A solução gulosa, no entanto, pode parecer uma alternativa, mas logo chega-se à conclusão de que não é simples definir uma regra que funcione corretamente para um algoritmo guloso.

Aplicando a técnica da programação dinâmica, é possível obter uma solução simples com complexidade polinomial para este problema. Baseado no algoritmo apresentado em (CHARRAS; LECROQ, 1998), a ideia consiste em descobrir o custo de transformar um prefixo de A em um prefixo de B utilizando as operações descritas no enunciado, até que toda a *string* esteja transformada. Para cada subproblema, representado por um par de prefixos, será avaliada qual ação deve ser tomada para transformar um prefixo no outro, considerando os custos da ação e dos subproblemas já resolvidos.

O algoritmo consiste em preencher uma tabela dp de dimensões $(n+1) \times (m+1)$, em que $dp(i, j)$ indica o custo mínimo de transformar o prefixo $A(1, ..., i)$ em $B(1, ..., j)$. Dessa forma, para cada par de índices (i, j), os seguintes cenários serão considerados:

1. $A(i) = B(j)$. Nesse caso, não é necessário realizar nenhuma operação nos caracteres $A(i)$ e $B(j)$, logo a resposta será igual à encontrada para os prefixos $A(1, ..., i-1)$ e $B(1, ..., j-1)$, ou seja, $dp(i, j) = dp(i-1, j-1)$

2. $A(i) \neq B(j)$, isto é, não há uma correspondência. Então será necessário realizar uma das três operações para que os prefixos de A e B sejam iguais até esse par de índices. Para tomar a decisão de qual operação realizar, serão avaliadas as seguintes opções e, dentre elas, escolhida a de menor custo:

   a) Substituir o caractere $A(i)$ por $B(j)$. Utilizaremos a resposta já descoberta do custo de transformar o prefixo $A(1, ..., i-1)$ em $B(1, ..., j-1)$ e somaremos isso ao custo de transformar o caractere $A(i)$ em $B(j)$. Dessa forma, $dp(i, j) = dp(i-1, j-1) + C_{sub}$

   b) Remover o caractere $A(i)$. Como estamos removendo o caractere de A na posição i, o novo custo deve ser igual ao de transformar o prefixo $A(1, ..., i-1)$

em $B(1, ..., j)$ somado ao custo de remoção do caractere $A(i)$, fazendo com que $dp(i, j) = dp(i - 1, j) + C_{rem}$

c) Adicionar o caractere $B(j)$ logo após $A(i)$. Nesse caso, a resposta deve ser igual ao custo de transformar o prefixo $A(1, ..., i)$ em $B(1, ..., j - 1)$ somado ao custo de adicionar o caractere $B(j)$, tendo $dp(i, j) = dp(i, j - 1) + C_{adc}$

A inicialização da tabela *dp* ocorre da seguinte forma:

- $dp(0, 0) = 0$. Esse caso considera a comparação de duas *strings* vazias
- $dp(i, 0) = dp(i - 1, 0) + C_{rem}$, para $i > 0$. Como o prefixo de B comparado é uma *string* vazia, deve-se remover os caracteres do prefixo de A
- $dp(0, j) = dp(0, j - 1) + C_{adc}$, para $j > 0$. Como o prefixo de A comparado é uma *string* vazia, deve-se transformá-lo no prefixo de B adicionando os caracteres

Para que a ideia fique mais clara, vamos aplicar o algoritmo descrito acima para a seguinte entrada: A = "BRC", B = "ABRA", $C_{sub} = 3$, $C_{rem} = 2$, $C_{adc} = 2$. Nesse caso, será apresentada primeiramente a tabela *dp* totalmente preenchida que sejam feitos comentários sobre alguns pontos importantes.

Vale ressaltar que, como a entrada é pequena, podemos saber de antemão que o menor custo será obtido adicionando o caractere "A" na primeira posição da *string* A, resultando em "ABRC", a custo 2, e substituindo o caractere "C" por "A", a custo 3, resultando na *string* "ABRA", que é igual à *string* B. O custo total será igual a 5.

Ao aplicar de fato o algoritmo, a tabela *dp* será preenchida conforme mostra o Quadro 18.

Quadro 18 – Distância de edição entre prefixos de ABRA e BRC

	0	1 (A)	2 (B)	3 (R)	4 (A)
0	0	2	4	6	8
1 (B)	2	3	2	4	6
2 (R)	4	5	4	2	4
3 (C)	6	7	6	4	5

Nos casos de $dp(i, j)$ onde $i = 0$ ou $j = 0$, podemos notar que, como dito anteriormente, as únicas operações possíveis de se realizar são as de remoção e adição de caracteres. No caso de $i = 0$, temos que transformar o prefixo "" de A em "A", "AB", "ABR" e "ABRA", respectivamente para cada valor de j, adicionando todos esses caracteres, e em $j = 0$, temos que transformar todos os prefixos "B", "BR" e "BRC" de A em "", respectivamente para cada valor de i removendo todos esses caracteres.

No par de índices $(1, 2)$, que armazena o custo mínimo de transformar o prefixo "B" em "AB", há uma correspondência com o caractere "B". Dessa forma, ocorreu a situação

explicada no caso 1 descrito acima, logo a resposta utilizada foi a mesma do par de índices $(0, 1)$, que guardava o custo de transformar "" em "A". Se podemos transformar "" em "A" a custo 2, então podemos transformar "B" em "AB" a custo 2.

A mesma coisa acima aconteceu na posição $(2, 3)$. Como houve uma correspondência com o caractere "R", foi utilizada a mesma resposta da posição $(1, 2)$, o que equivale a dizer que transformamos "BR" em "ABR" com o mesmo custo de transformar "B" em "AB".

Na posição $(2, 4)$ não houve correspondência, fazendo com que seja considerado o caso 2 descrito acima, então seria necessário aplicar uma das 3 operações. Seguindo as fórmulas dadas no caso 2 para o cálculo do custo final de utilizar cada uma das operações para transformar um prefixo no outro, a substituição geraria um custo de $dp(1, 3) + C_{sub} = 4 + 3 = 7$, a remoção resultaria num custo de $dp(1, 4) + C_{rem} = 6 + 2 = 8$, e a adição, de $dp(2, 3) + C_{adc} = 2 + 2 = 4$. Como o menor custo foi obtido com a adição, a ação escolhida foi a de adicionar o caractere "A". Dessa forma, a resposta foi igual à soma de $dp(2, 3)$, que corresponde ao custo de transformar "BR" em "ABR" com 2, que é o custo de adicionar o caractere "A" no fim da *string*, formando "ABRA".

Na posição $(3, 4)$ também não houve correspondência. Dentre as opções possíveis, a de menor custo foi substituir o caractere "C" pelo caractere "A". A resposta em $dp(2, 3)$ indica que é possível transformar "BR" em "ABR" a custo 2. Como o custo de substituição é 3, então foi possível transformar "BRC" em "ABRA" a custo 5, que é a resposta em $dp(2, 3)$ somada de 3.

De forma resumida, para executar o algoritmo podemos aplicar, para cada par de índices i e j, tal que $0 \leq i \leq n$ e $0 \leq j \leq m$, a seguinte regra:

- Para os casos em que $i = 0$ ou $j = 0$:
  - $dp(0, 0) = 0$
  - $dp(i, 0) = dp(i - 1, 0) + C_{rem}$, para todo $i > 0$
  - $dp(0, j) = dp(0, j - 1) + C_{adc}$, para todo $j > 0$
- Para os casos em que $i > 0$ e $j > 0$:
  - $dp(i, j) = dp(i - 1, j - 1)$, $seA(i) = B(j)$
  - $dp(i, j) = \min(dp(i - 1, j - 1) + C_{sub}, dp(i - 1, j) + C_{rem}, dp(i, j - 1) + C_{adc})$, se $A(i) \neq B(j)$

Uma possível implementação para este algoritmo é a apresentada no Código 16.

Código 16 – Distância de Edição *bottom-up*

```
int N = A.size(), M = B.size();
int dp[N+1][M+1];

// Inicializacao da tabela
dp[0][0] = 0;
for(int i = 1; i <= N; i++)
        dp[i][0] = dp[i-1][0] + C_rem;
for(int j = 1; j <= M; j++)
        dp[0][j] = dp[0][j-1] + C_adc;

// Aplicacao do algoritmo
for(int i = 1; i <= N; i++) {
        for(int j = 1; j <= M; j++) {
                int sub_cost = (A[i-1] == B[j-1] ? 0 : C_sub);
                dp[i][j] = MIN(dp[i-1][j-1] + sub_cost,
                               dp[i-1][j] + C_rem,
                               dp[i][j-1] + C_adc);
        }
}
```

Complexidade

O algoritmo consiste em preencher uma tabela de dimensões $(n+1) \times (m+1)$, realizando um número constante de operações em cada célula preenchida. Dessa forma, a complexidade é $\boldsymbol{O(n \cdot m)}$.

4 PROBLEMAS INTERMEDIÁRIOS

Neste capítulo serão abordados dois problemas cujas soluções envolvem a utilização de alguma outra técnica combinada à programação dinâmica. No caso do problema da Maior Subsequência Crescente, será apresentada uma solução que utiliza programação dinâmica e, depois, uma forma de melhorar sua complexidade combinando a ideia desse algoritmo com busca binária. Já no problema do Caixeiro Viajante, será utilizada a técnica de representar subconjuntos através de uma máscara de *bits*. Essa técnica também é utilizada em muitos outros problemas que não possuem solução polinomial.

4.1 MAIOR SUBSEQUÊNCIA CRESCENTE

Problema: Dada uma sequência S com n valores inteiros, encontre a subsequência de S que seja estritamente crescente de tamanho máximo.

Uma subsequência pode ser obtida removendo quaisquer caracteres da sequência original, mantendo a ordem dos remanescentes.

Por exemplo, para a sequência $S = (10, 4, 5, 6, 5, 15, 2, 3, 11, 12)$, a maior subsequência estritamente crescente é (4, 5, 6, 11, 12), que é obtida selecionando os elementos da seguinte forma: (10, 4, 5, 6, 5, 15, 2, 3, 11, 12).

A solução trivial utilizando a estratégia da força bruta seria gerar todas as subsequências e, dentre as crescentes, selecionar a maior. Entretanto, assim como já observado em outros problemas, essa solução possui uma complexidade de tempo exponencial.

É possível obter uma complexidade polinomial utilizando programação dinâmica. Baseado no algoritmo apresentado em (HALIM; HALIM, 2013), a ideia consiste em, para cada elemento em S, guardar o tamanho da maior subsequência crescente que termina neste elemento. Para determinar esse tamanho para cada elemento, basta verificar o tamanho da maior subsequência crescente que termina em cada um dos elementos anteriores que são menores que ele. Dando um índice i qualquer, o resultado para o elemento $S(i)$ considerado no momento será igual ao maior tamanho de subsequência crescente encontrada em um elemento $S(j)$ somado com 1, para $j < i$ e $S(j) < S(i)$.

Utilizando um vetor dp para armazenar em $dp(i)$ o tamanho da maior subsequência crescente que termina no elemento $S(i)$, teremos como caso base $dp(0) = 0$, quando nenhum elemento é considerado, ou então $dp(1) = 1$, considerando que a única subsequência crescente que termina no primeiro elemento é a que só consiste dele. Dessa forma, cada valor de dp poderia ser determinado da seguinte forma:

- $dp(1) = 1$
- $dp(i) = \max_{S(j)<S(i)\,/\,j<i} dp(j) + 1$, para $i > 1$

O resultado final será o maior valor encontrado dentro do vetor *dp*. Os valores desse vetor para o exemplo do enunciado seriam iguais aos apresentados no Quadro 19.

Quadro 19 – Maior subsequência crescente terminando em cada índice

i	1	2	3	4	5	6	7	8	9	10
S	10	4	5	6	5	15	2	3	11	12
dp	1	1	2	3	2	4	1	2	4	5

Para cada um dos n elementos, foram realizadas $O(n)$ operações para determinar o tamanho da maior subsequência crescente que termina nele, logo a complexidade do algoritmo apresentado é de $O(n^2)$.

Analisando a tabela acima, é possível perceber que serão feitas algumas comparações desnecessárias ao longo da execução do algoritmo. Por exemplo, em $i = 1$, a sequência que termina no elemento de valor 10 tem tamanho 1, assim como em $i = 2$, com o elemento de valor 4. Como a subsequência deve ser crescente, todo elemento que pode ser colocado após o 10 também pode ser colocado após o 4. Mesmo assim, o índice 1 será considerado desnecessariamente nas comparações desse algoritmo. De forma geral, se uma subsequência de tamanho k termina em um elemento de valor $S(i)$, então todos os outros elementos de valor maior ou igual a $S(i)$ que finalizam uma subsequência de tamanho k não são mais necessários. A mesma coisa também acontece, por exemplo, nos índices $i = 3$ e $i = 5$, com elementos de valor 5 que finalizam subsequências de tamanho 2.

Outro ponto que pode ser percebido é que são realizadas buscas completas. Por exemplo, em $i = 9$, com $S(i) = 11$, sabemos que o menor valor que finaliza uma subsequência de tamanho 1 é 4, o que finaliza uma subsequência de tamanho 2 é 5, o que finaliza uma de tamanho 3 é 6 e o que finaliza uma de tamanho 4 é 15. Estruturando os vetores de forma diferente, poderia haver uma forma de realizar uma busca binária para descobrir que o 11 deve ser colocado após o 6, em vez de fazer uma comparação com todos esses valores para isso.

Considerando os pontos levantados acima, é possível construir um novo algoritmo com uma complexidade menor. A ideia para esse algoritmo é realizar uma iteração sobre os elementos de S e, durante isso, preencher um vetor que denotaremos como LIS. Para cada elemento $S(i)$, o vetor LIS deverá indicar no valor $LIS(j)$ qual é o menor elemento da sequência S até o índice i que pode ser o último elemento de uma subsequência crescente de tamanho j. A construção desse vetor leva em conta a premissa de que, como queremos encontrar o tamanho da maior subsequência crescente, só precisamos de duas informações: o tamanho das subsequências até o momento e o último elemento de cada uma delas, para que saibamos se é possível incluir o elemento atual no fim de alguma subsequência. No caso de haver duas subsequências de tamanhos iguais, será mantida aquela cujo último elemento seja menor, pois tem maior possibilidade de outro elemento poder ser incluído.

Podemos notar que há algumas características no vetor LIS importantes para aplicarmos o algoritmo:

- O vetor é estritamente crescente, pois se é possível fazer uma subsequência crescente de tamanho j que termina com o elemento x, então também é possível fazer uma de tamanho $j-1$ que termina com um elemento menor que x
- Seu último índice indica a maior subsequência crescente até o momento, pois cada índice é o tamanho de uma subsequência crescente

Realizando uma iteração por todos os elementos de S, deve-se, para cada elemento $S(i)$, atualizar o vetor LIS para considerar este elemento. O primeiro passo é descobrir o tamanho da maior subsequência crescente até então que termina com um elemento menor que $S(i)$. Como o vetor LIS é estritamente crescente, é possível realizar uma busca binária para encontrar o índice de LIS que corresponde ao maior elemento que seja menor que $S(i)$. Supondo que o índice desse elemento seja j, haverá 2 cenários a se considerar:

1. j é o último índice do vetor LIS. Neste caso, adicionaremos $S(i)$ na posição $j+1$ ainda vazia do vetor, sendo $j+1$ o valor da maior subsequência crescente até o momento
2. j não é o último índice do vetor LIS. Neste cenário, já haverá um elemento em $LIS(j+1)$, entretanto, ele deve ser menor ou igual a $S(i)$, pois $LIS(j)$ é o maior valor do vetor LIS que é menor do que $S(i)$. Dessa forma, podemos substituir o valor em $LIS(j+1)$ por $S(i)$.

Pode-se perceber que, embora a análise tenha sido diferente para cada cenário, a ação tomada foi a mesma: atribuir o valor de $S(i)$ em $LIS(j+1)$.

Para facilitar a generalização do algoritmo, podemos considerar $LIS(0) = -\infty$ como caso base. Isso permitirá que qualquer valor possa formar uma subsequência crescente de tamanho 1.

Utilizando como exemplo o valor de $S = (10, 4, 5, 6, 5, 15, 2, 3, 11, 12)$, foi representado no Quadro 20 como ficaria o vetor LIS a cada momento da iteração da sequência S. O elemento sublinhado na terceira coluna indica, para cada i, a posição no vetor LIS onde se encontra o elemento $S(i)$.

Em $i = 2$, é possível perceber que o último valor da subsequência crescente de tamanho 1, que era 10, foi atualizado para 4. Já em $i = 3$, como havia uma subsequência crescente de tamanho 1 terminando com o valor 4, então é possível formar uma subsequência crescente de tamanho 2 terminando com o valor 5. A lógica é análoga para o caso em $i = 4$.

Em $i = 5$, o valor 5 teve de ser colocado logo após o 4 no vetor LIS, mas esse espaço já era ocupado por um valor 5, então o vetor se manteve com os mesmos valores.

Quadro 20 – Execução do algoritmo de maior subsequência crescente

i	S(i)	LIS
1	10	(10)
2	4	(4)
3	5	(4, 5)
4	6	(4, 5, 6)
5	5	(4, 5, 6)
6	15	(4, 5, 6, 15)
7	2	(2, 5, 6, 15)
8	3	(2, 3, 6, 15)
9	11	(2, 3, 6, 11)
10	12	(2, 3, 6, 11, 12)

Em $i = 7$, pode-se perceber que o valor 4 foi alterado pelo valor 2, pois este agora é o menor elemento que pode finalizar uma subsequência crescente de tamanho 1. Esse é o motivo para o vetor LIS não possuir necessariamente a resposta da maior subsequência crescente: é possível alterar valores menores que são parte da subsequência crescente de outros valores maiores. Em $i = 8$, um evento parecido acontece.

É possível perceber que o vetor LIS não dá a resposta de qual é a maior subsequência crescente, visto que ao final da iteração seu valor era (2, 3, 6, 11, 12), enquanto que a maior subsequência crescente é (4, 5, 6, 11, 12). Dessa forma, será necessário fazer 2 pequenas modificações no funcionamento do algoritmo para poder reconstruir a solução:

1. Em vez de salvar o valor de $S(i)$ em LIS, será salvo o valor do índice i. Dessa forma, será possível saber mais facilmente onde está, em S, cada elemento armazenado em LIS. Também é possível recuperar o valor de $S(i)$ acessando a sequência S com o índice i armazenado

2. Será mantido um vetor $trace$, de tamanho n, que armazenará, para cada índice i, qual elemento antecede o elemento $S(i)$ na subsequência crescente terminada por ele. Esse vetor será atualizado com o valor $LIS(j)$, que corresponde ao índice do maior elemento que seja menor que $S(i)$. Em outras palavras, ao inserir i na posição $LIS(j + 1)$, também deverá ser inserido $LIS(j)$ em $trace(i)$

Dessa forma, é possível reconstruir a solução. Com a alteração feita, os valores armazenados em cada vetor durante a iteração de S serão iguais aos mostrados no Quadro 21.

Para o exemplo dado, o último elemento da maior subsequência crescente será o valor $S(10)$, que é igual a 12. O índice em S do valor anterior ao 12 na maior subsequência

[1] O vetor LIS nesse quadro contém as modificações explicadas após o primeiro exemplo, então armazena os índices de S, não seus valores propriamente ditos

Quadro 21 – Execução do algoritmo de maior subsequência crescente com rastro

i	S(i)	LIS[1]	trace(i)
1	10	(1)	-
2	4	(2)	-
3	5	(2, 3)	2
4	6	(2, 3, 4)	3
5	5	(2, 5, 4)	2
6	15	(2, 5, 4, 6)	4
7	2	(7, 5, 4, 6)	-
8	3	(7, 8, 4, 6)	7
9	11	(7, 8, 4, 9)	4
10	12	(7, 8, 4, 9, 10)	9

crescente será $trace(10)$, que é 9, logo o valor anterior ao 12 é $S(9)$, que é 11. Continuando, temos que $trace(9)$ é igual a 4, e $S(4)$ é igual a 6, depois $trace(4)$ é 3 e $S(3)$ é 5, $trace(3)$ é 2 e $S(2)$ é 4. Como $trace(2)$ não possui valor armazenado, terminamos a execução do algoritmo, chegando à conclusão que a maior subsequência crescente de S é (4, 5, 6, 11, 12).

De forma resumida, o algoritmo consiste em aplicar, para cada i entre 1 e n, as seguintes operações:

1. Utilizar busca binária para encontrar o maior índice j tal que $S(LIS(j)) < S(i)$
2. Atribuir o valor i em $LIS(j+1)$
3. Atribuir o valor $LIS(j)$ em $trace(i)$. Caso j seja 0, deve-se deixar $trace(i)$ marcado como vazio

O Código 17 é uma possível forma de implementar esse algoritmo.

Código 17 – Maior Subsequência Crescente *bottom-up*

```
int N = S.size();

// Construindo o vetor LIS e trace
vector<int> LIS, trace(N);
for(int i = 0; i < N; i++) {
        // Encontra o primeiro idx tal que S[LIS[idx]] >= S[i]
        // Se nao existir, retorna o tamanho do vetor LIS
        int idx = bin_search(LIS, S, S[i]);

        if(idx == LIS.size())
                LIS.push_back(i);
        else
                LIS[idx] = i;

        if(idx == 0)
                trace[i] = -1; // Sem antecedente
        else
                trace[i] = LIS[idx-1];
}

// Reconstruindo a solucao
vector<int> ans; // Maior subsequencia crescente
if(LIS.size() > 0) {
        int idx = LIS.back();
        // Loop ate encontrar indice sem antecedente
        while(idx != -1) {
                ans.push_back(S[idx]);
                idx = trace[idx];
        }
}
// Inverte, pois a subsequencia foi montada de tras para frente
reverse(ans.begin(), ans.end());
```

Complexidade

É realizada uma iteração por todos os n elementos da sequência S. Para cada um desses elementos, é feita uma busca binária no vetor LIS, que pode ter até n elementos. Dessa forma, cada busca binária tem complexidade $O(log(n))$. Como essa operação é realizada n vezes, a complexidade da etapa de construção dos vetores LIS e trace é $O(n \cdot log(n))$.

Para reconstruir a solução, apenas seguimos o caminho inverso indicado pelo vetor *trace*, realizando um número constante de operações a cada elemento iterado. Como haverá, no máximo, n elementos para iterar, a complexidade desta etapa é $O(n)$.

Portanto, a complexidade final do algoritmo é $\boldsymbol{O(n \cdot log(n))}$.

4.2 CAIXEIRO VIAJANTE

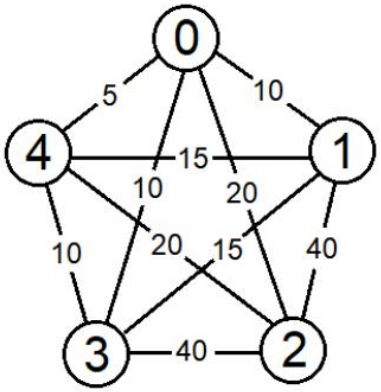
Problema: São dadas n cidades, representadas por um grafo completo de n vértices e uma matriz com as distâncias entre cada par de cidades, sendo $dist(a, b)$ a distância entre as cidades a e b.

Um caixeiro deseja passar por todas as cidades sem que nenhuma seja visitada mais de uma vez e retornar à de origem. Qual a distância mínima que ele deve percorrer?

De acordo com as definições em (FEOFILOFF, 2017), um *caminho* é uma sequência de vértices do grafo sem arcos repetidos, onde um arco $u - w$ está presente no caminho se os vértices u e w são adjacentes na sequência. Um *ciclo* é um caminho que possui mais de um arco e cujo vértice de origem (o primeiro da sequência) é igual ao de término (o último da sequência). Por fim, um *ciclo hamiltoniano* é um ciclo em que todos os vértices do grafo aparecem exatamente uma vez. Dessa forma, o problema nos pede o ciclo hamiltoniano de menor custo.

Por exemplo, considerando o grafo representado na Figura 8, o custo do menor ciclo hamiltoniano é 75. Nesse caso, há mais de um ciclo com custo ótimo. Um exemplo seria $0 \to 1 \to 3 \to 4 \to 2 \to 0$.

Figura 8 – Grafo completo com 5 vértices



A solução ingênua para o problema consiste em gerar todas as permutações possíveis de cidades e, dentre elas, escolher uma de menor custo. Note que, como o grafo é completo, isto é, todos os pares de vértices estão conectados por um arco, então cada permutação de cidades representará um ciclo hamiltoniano, visto que haverá um arco para cada par adjacente da sequência. A complexidade para gerar todas as soluções dessa forma é de $O(n!)$.

Utilizando programação dinâmica, é possível obter uma solução de complexidade $O(n^2 \cdot 2^n)$, que, embora não seja polinomial, é melhor do que o fatorial da solução ingênua. Para valores muito pequenos de n, como 4 ou 5, o tempo de execução dos dois algoritmos não é tão diferente, mas para valores um pouco maiores, como 20, é possível

perceber como o fatorial cresce muito mais rápido, visto que $2^{20} \approx 10^6$, enquanto que $20! \approx 2.4 \cdot 10^{18}$. O algoritmo que será explicado a seguir é uma abordagem *top-down* conforme a apresentada em (HALIM; HALIM, 2013).

Uma observação que pode ser feita para melhorar a solução é que não há necessidade de considerar a ordem exata em que os vértices foram visitados, como é feito ao gerar as permutações. Em vez disso, utilizando a técnica da programação dinâmica, serão considerados apenas os subconjuntos com os vértices que já foram visitados para cada subproblema e o último vértice que foi visitado, ou seja, o vértice onde o caixeiro encontra-se no momento. A última condição é necessária para que possamos expandir a solução de um subproblema para o outro, pois sem ela não seria possível descobrir o custo de chegar a um subconjunto de vértices a partir de outro.

Outro ponto importante de se observar é que podemos fixar um vértice de origem qualquer, pois a resposta será o custo mínimo de percorrer um ciclo no grafo, logo todos os vértices podem ser a origem deste ciclo. Dessa forma, considerando que todos os vértices do grafo estejam numerados de 0 a $n - 1$, o vértice de número 0 será definido como a origem.

Para representar um subconjunto de vértices, utilizaremos máscaras de *bits*. Dessa forma, cada subconjunto será representado por um número inteiro cuja representação binária determina, nas posições em que o *bit* é 1, a presença de um vértice no subconjunto e, nas posições em que é 0, sua ausência. Para o conjunto $\{4, 3, 2, 1, 0\}$[2], por exemplo, se quisermos representar o subconjunto $\{3, 0\}$ como uma sequência de zeros e uns que representam ausência ou presença de um elemento, ela seria igual a (0, 1, 0, 0, 1). Pensando nesta mesma sequência na forma de *bits*, teríamos 01001, que equivale, na representação decimal, ao número 9. Dessa forma, esse subconjunto seria representado pelo número 9.

O uso da máscara de *bits* nos possibilita, além de representar subconjuntos, realizar operações entre eles com a mesma velocidade de uma operação entre números inteiros utilizando os operadores de *bit* e algumas funções internas de linguagens como o C++.

Tendo em mãos todas as observações feitas acima, encontraremos a solução aplicando um algoritmo *top-down* de programação dinâmica. Utilizando a técnica de memoização, teremos uma função recursiva tsp, tal que $tsp(mask, j)$ represente a situação em que todas as cidades presentes no subconjunto representado por $mask$ já tenham sido visitadas e j seja a cidade onde o caixeiro encontra-se atualmente. Para este subproblema, $tsp(mask, j)$ deve retornar o custo mínimo de visitar, a partir de j, todos os vértices restantes, isto é, que não estão no subconjunto de $mask$ dos visitados, e retornar ao vértice de origem, que é o vértice 0. Em outras palavras, $tsp(mask, j)$ retornará o custo mínimo de finalizar o problema tendo como ponto de partida a situação descrita por seus parâmetros $mask$ e

[2] Os elementos do conjunto foram colocados em ordem decrescente para simplificar a visualização e implementação da máscara de *bits*, pois os *bits* mais à esquerda são mais significativos que os da direita, então representam potências de 2 de valores mais altos

j. Uma matriz *memo* será utilizada para evitar cálculos repetidos. Dessa forma, o valor de $tsp(mask, j)$ estará armazenado em $memo(mask, j)$.

Como desejamos começar no vértice 0 e visitar todos os vértices do grafo, o valor de $mask$ deve ser, inicialmente, 2^0, que é igual a 1, indicando que o vértice 0 já está no subconjunto, e o de j deve ser igual a 0. Deve-se considerar os seguintes cenários para montar a relação de recorrência, considerando que o vértice j sempre estará incluso no subconjunto denotado por $mask$:

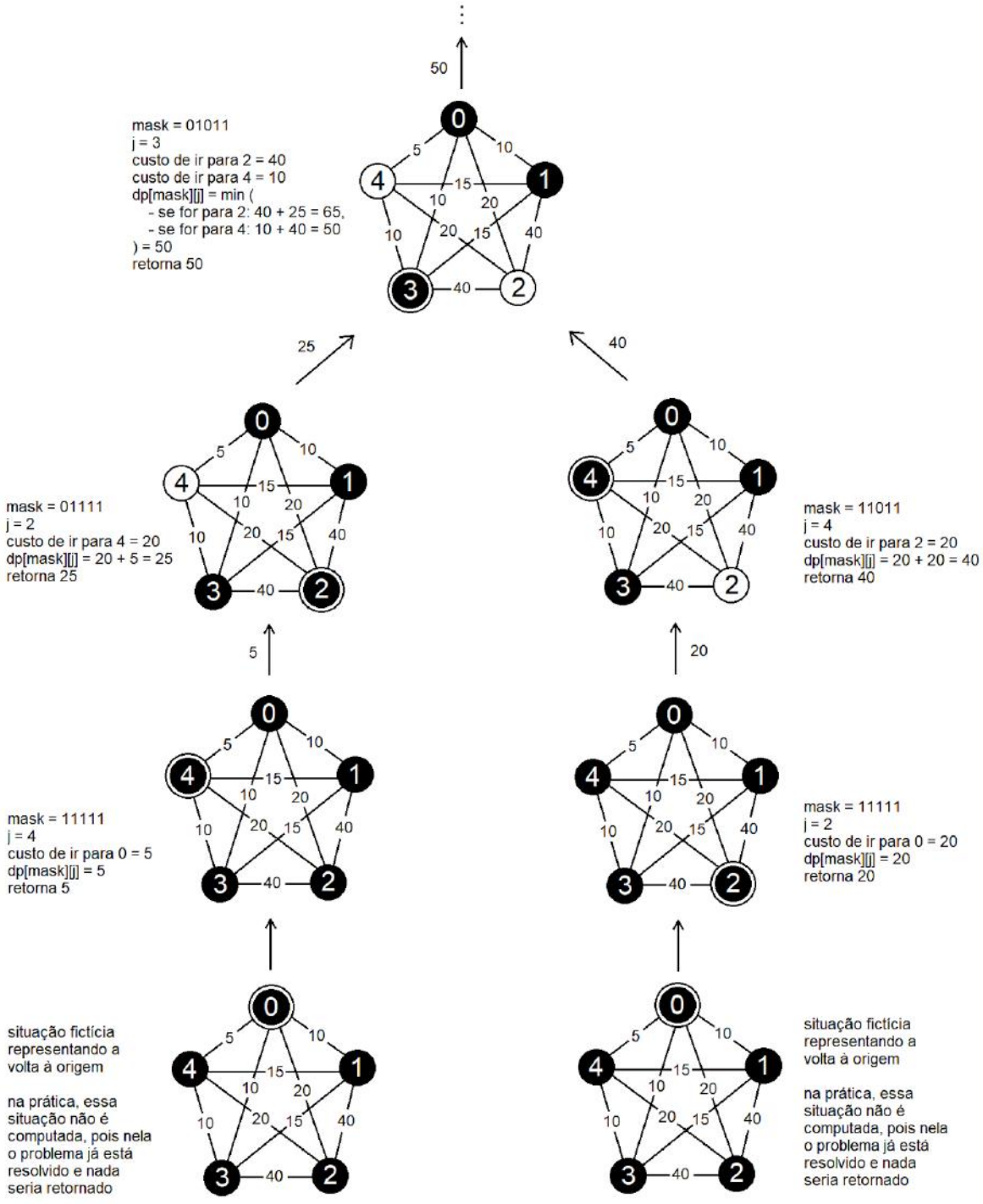
1. Caso geral: $mask \neq 2^n - 1$. O valor $2^n - 1$, na forma binária, é um número com os n últimos dígitos iguais a 1. Quando $mask$ é diferente deste valor, significa que há vértices que ainda não foram visitados. Nesse caso, devemos escolher um vértice k, dentre todos os que não estão no subconjunto dos visitados, para ser o próximo a ser visitado. Para escolher esse valor, deve-se calcular recursivamente o custo para todos os possíveis valores de k e escolher aquele que possuir o menor, considerando que esse custo deve ser equivalente à soma do custo de visitar o vértice k a partir de j com o custo de resolver o novo subproblema, que considera o vértice k no subconjunto. Em outras palavras, o custo será de $dist(j, k) + tsp(mask \,|\, 2^k, k)$, onde "|" indica o operador de bits *or*

2. Caso base: $mask = 2^n - 1$. Nesse caso, todos os vértices já foram visitados, então deve ser retornado apenas o custo de voltar para a cidade de origem. Logo, o valor retornado será $dist(j, 0)$

Para que a ideia do algoritmo fique mais simples, a Figura 9 exemplifica a recursão que seria realizada para calcular o valor de $tsp(11, 3)$ para o grafo dado na Figura 8 (note que 11 em sua forma binária é 01011, correspondendo ao conjunto de vértices {0, 1, 3}). O vértice marcado indica a última cidade visitada até o momento. Repare também que, por razões didáticas, todas as etapas da recursão foram demonstradas, embora no caso prático isso não aconteça porque a memoização torna desnecessário o cálculo de subproblemas já calculados anteriormente.

Quando o caixeiro encontra-se na cidade 3, ele tem a opção de viajar para a cidade 2 ou para a 4. Se ele for para a cidade 2 (situação representada no ramo à esquerda), o subproblema poderá ser resolvido com custo igual à distância entre as cidades 3 e 2, que é 40, somada ao custo mínimo de resolver o restante do problema, que consiste em visitar a cidade 4, com custo 20, e retornar à origem, com custo 5, totalizando 25. Logo, o custo mínimo para resolver este subproblema tomando como próximo passo a cidade 2 é igual a $40 + 25 = 65$. Entretanto, se ele for para a cidade 4 (situação representada no ramo à direita), o custo total será apenas 50, que corresponde a viagem à cidade 4, com custo 10, seguindo para a cidade 2, com custo 20, e retornando à origem, com custo 20. Dessa forma, como queremos o custo mínimo, o valor retornado em $tsp(11, 3)$ será 50, e a próxima cidade a ser visitada deve ser a 4.

Figura 9 – Resolução parcial do problema do Caixeiro Viajante



Repare que cada subproblema supõe que todas as cidades pintadas de preto já foram visitadas. Além disso, o valor retornado por tsp corresponde à soma do custo de visitar uma nova cidade com o custo mínimo de finalizar o ciclo hamiltoniano considerando essa nova cidade no conjunto das cidades visitadas. Este último valor é calculado através de uma chamada recursiva. Por exemplo, o cálculo do custo mínimo de finalizar o ciclo hamiltoniano tomando a cidade 4 como próxima a ser visitada é igual a $10 + tsp(27, 4)$,

tendo em vista que 27 em sua forma binária é igual a 11011, que corresponde ao conjunto de vértices {0, 1, 3, 4}.

Por fim, ao final de cada um dos ramos, foi representada a situação do problema já resolvido, visto que o ciclo hamiltoniano foi finalizado. Ela foi colocada apenas para ilustrar o objetivo do algoritmo, mas não é alcançada na prática pelo código, pois $tsp(2^n - 1, j)$, para qualquer j, representa o caso base e já retornará o valor da distância de j à origem.

Pode-se resumir a regra para o retorno de $tsp(mask, j)$ como:

- $\min_{k \notin mask} (dist(j,k) + tsp(mask \mid 2^k, k))$, se $mask \neq 2^n - 1$
- dist(j, 0), se $mask = 2^n - 1$

A chamada que retornará o resultado da solução do problema é $tsp(0, 0)$.

Este algoritmo pode ser implementado conforme mostra o Código 18.

Código 18 – Caixeiro Viajante *top-down*

```
// Iniciado com -1 em todas as posicoes
int memo[1 << N][N];

int tsp(int mask = 1, int j = 0) {
        if(memo[mask][j] != -1)
                return memo[mask][j];

        // Caso base
        if(mask == (1 << N) - 1)
                return dist[j][0];

        int ans = INF;
        for(int k = 0; k < N; k++)
                // Verifica se a cidade ainda nao foi visitada
                if(!(mask & (1 << k)))
                        /* Escolhe a proxima cidade que
                           minimiza o custo total */
                        ans = min(ans,
                                  dist[j][k] + tsp(mask | (1 << k), k));

        return memo[mask][j] = ans;
}
```

Complexidade

Para um subproblema qualquer, representado por um par de valores de $mask$ e j, serão realizadas $O(n)$ operações em $tsp(mask, j)$ caso seja a primeira vez que a função é

chamada e $O(1)$ caso contrário, devido à memoização. Como há $O(n \cdot 2^n)$ subproblemas possíveis, a complexidade do algoritmo é $\boldsymbol{O(n^2 \cdot 2^n)}$.

Vale ressaltar que a complexidade de uma chamada de função de *tsp* não é $O(n)$. O parágrafo acima considerou $O(n)$ operações na função porque, para facilitar a análise, as operações dentro das outras chamadas de *tsp* feitas recursivamente foram consideradas como operações dessas outras chamadas, não da original.

5 CONCLUSÃO

De acordo com tudo que foi visto nesse trabalho, pode-se perceber que há bastantes semelhanças entre as soluções com programação dinâmica de diversos problemas distintos. Algumas delas utilizam a mesma base para a ideia do algoritmo, como foi o caso dos problemas com *strings*, que tiveram como ideia principal a divisão em subproblemas considerando prefixos das *strings*. Outros tiveram semelhanças mais sutis, como a mochila booleana e o problema do troco, em que no primeiro, a ideia era dividir em subproblemas com menos itens e menor capacidade na mochila, e no segundo, em subproblemas com menos moedas e menor valor total a ser somado.

Há muitos outros problemas além dos que foram abordados neste trabalho, mas é esperado que o leitor tenha mais facilidade para desenvolver soluções para esses novos problemas após o entendimento deste conteúdo.

O conhecimento sobre programação dinâmica no geral não está consolidado em apenas uma referência, mas difundido através de diversas contribuições feitas pela comunidade. Dessa forma, espera-se que este material seja mais uma dessas contribuições e possa trazer bastante valor a iniciantes no tema.

Para aprender mais sobre programação dinâmica, é recomendada a leitura dos tópicos que envolvem essa técnica em (HALIM; HALIM, 2013) e (CORMEN et al., 2009). Além disso, há muito conteúdo *online*, sobretudo em portais e comunidades sobre programação competitiva e ciência da computação no geral, como https://codeforces.com e https://www.geeksforgeeks.org.

REFERÊNCIAS

BENTLEY, J. **Programming Pearls (2nd Ed.)**. USA: ACM Press/Addison-Wesley Publishing Co., 1999. ISBN 0201657880.

CHARRAS, C.; LECROQ, T. **Sequence comparison**. 1998. Disponível em: http://www-igm.univ-mlv.fr/~lecroq/seqcomp/node1.html#SECTION001. Acesso em: 2020-12-31.

CORMEN, T. H. et al. **Introduction to Algorithms, Third Edition**. 3rd. ed. [S.l.]: The MIT Press, 2009. ISBN 0262033844.

FEOFILOFF, P. **Caminhos e ciclos em grafos**. 2017. Disponível em: https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/paths-and-cycles.html. Acesso em: 2020-12-30.

HALIM, S.; HALIM, F. **Competitive Programming 3: The New Lower Bound of Programming Contests**. [S.l.]: Lulu.com, 2013. ISBN 9788392212355.