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 $n > 1$

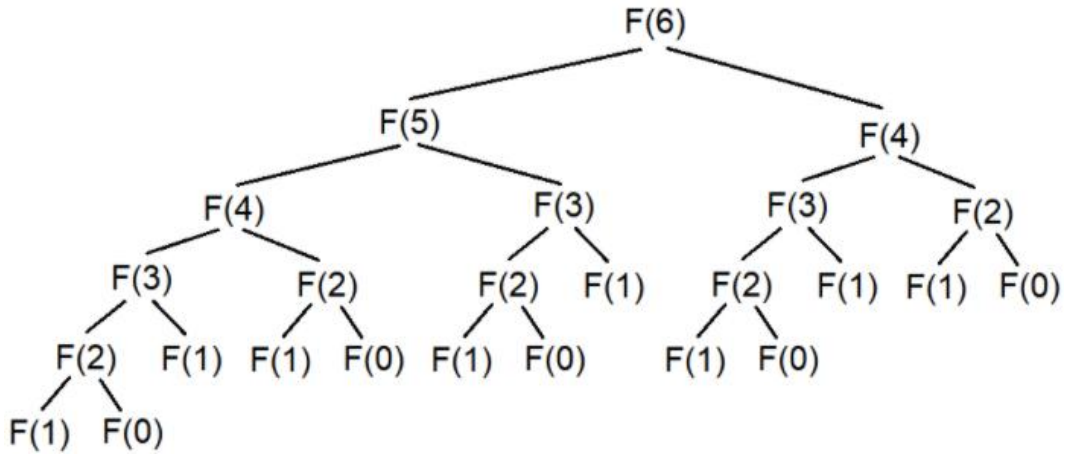
O Código 1 demonstra a implementação dessa função.

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);
}
```

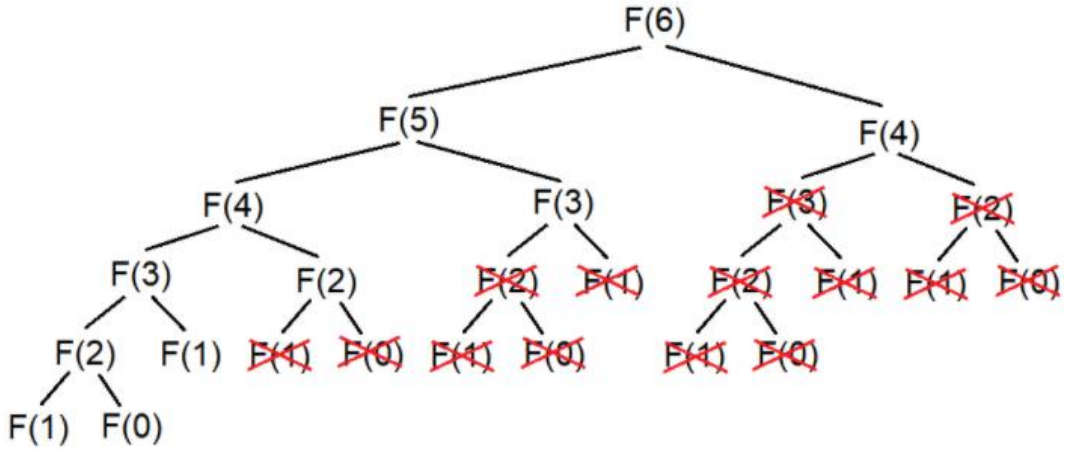
Dessa forma, para cada $F(n)$, seria realizado um cálculo recursivo completo dos valores de $F(n-1)$ e $F(n-2)$. O problema disso é que muitos valores serão computados mais de uma vez, como pode ser observado na Figura 1, que ilustra a árvore de execução para $F(6)$.

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

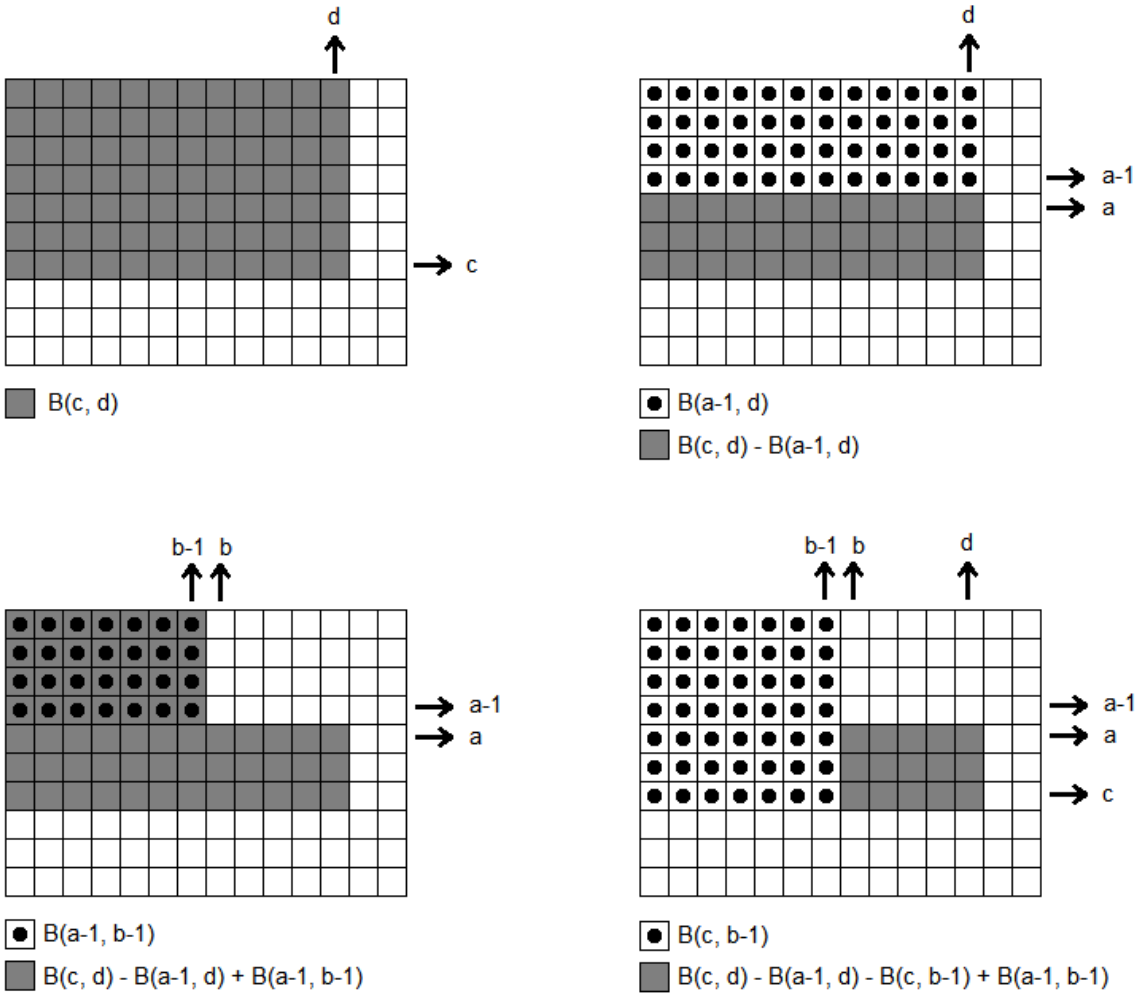
5	-10	2	3	6
-8	**7**	**1**	**-2**	-7
2	**-1**	**3**	**1**	-5
-1	**9**	**-15**	**13**	-20

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

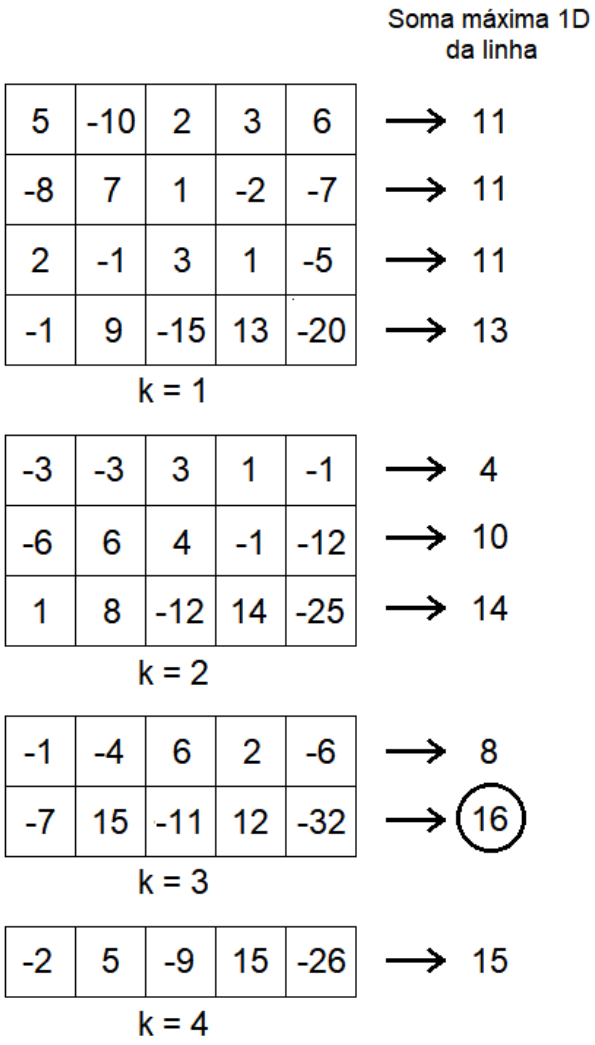
Seguindo o mesmo raciocínio para um subproblema de submatrizes de três linhas, podemos aplicar um pré-processamento semelhante, fazendo com que $B(i, j)$ seja igual a $A(i, j) + A(i + 1, j) + A(i + 2, j)$. Isso pode ser feito com apenas uma operação de soma, caso o pré-processamento do subproblema de 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.