OTIMIZAÇÃO DE ALGORITMOS POR MEIO DA PROGRAMAÇÃO DINÂMICA: ESTUDO DE CASO COM A SEQUÊNCIA DE FIBONACCI

Rafael do Nascimento de Andrade[1]
Tânia Camila Kochmanscky Goulart [2]

RESUMO

O presente artigo aborda a aplicação da Programação Dinâmica (PD) na otimização de algoritmos, com foco em reduzir o tempo de execução e a complexidade computacional. A partir de um estudo comparativo, utilizamos a geração da sequência de Fibonacci como base para explorar a diferença entre a abordagem tradicional de força bruta e a técnica de PD. A notação Big O é utilizada para quantificar a melhoria de desempenho, demonstrando a redução de $O(2^n)$ para $O(n)$ no caso estudado, proporcionando uma solução mais rápida e eficaz. Além disso, são apresentados exemplos de outras aplicações práticas de PD em áreas em que a técnica é amplamente aplicada em algoritmos, como economia, grafos e inteligência artificial. Os resultados evidenciam que a PD oferece soluções ótimas ao evitar o cálculo repetido de subproblemas, com grande impacto na eficiência computacional, por economizar tempo e recursos.

Palavras-chave: programação dinâmica; otimização; complexidade temporal; Fibonacci; algoritmos.

ABSTRACT

This article addresses the application of Dynamic Programming (PD) in algorithm optimization, with a focus on reducing execution time and computational complexity. From a comparative study, we used the generation of the Fibonacci sequence as a basis to explore the difference between the traditional brute force approach and the PD technique. The Big O notation is used to quantify the performance improvement, demonstrating the reduction from $O(2^n)$ to $O(n)$ in the case studied, providing a faster and more effective solution. Furthermore, examples of other practical applications of PD are presented in areas where the technique is widely applied in algorithms, such as economics, graphs and artificial intelligence. The results show that PD offers optimal solutions by avoiding the repeated calculation of subproblems, with a great impact on computational efficiency, by saving time and resources.

Keywords: dynamic programming; optimization; temporal complexity; Fibonacci; algorithms.

[1] Discente do curso de Ciência da Computação do Centro Universitário Filadélfia (UniFil)
[2] Docente do curso de Ciência da Computação do Centro Universitário Filadélfia (UniFil)

1 INTRODUÇÃO

A otimização de algoritmos é uma área de estudo voltada para a eficiência computacional, especialmente em problemas complexos que demandam soluções rápidas e precisas. A Programação Dinâmica (PD) foi introduzida por Richard Bellman na década de 1950 como uma maneira sistemática de resolver problemas que apresentam sobreposição de subproblemas e subestruturas ótimas. Desde então, a PD tem sido aplicada em uma vasta gama de áreas, como teoria dos grafos, inteligência artificial, engenharia de software, economia, bioinformática e finanças, proporcionando soluções eficientes para problemas que, de outra forma, seriam computacionalmente intransponíveis. (Lew, 2006).

O conceito central da Programação Dinâmica é dividir um problema complexo em subproblemas menores e resolver cada um deles uma única vez, armazenando seus resultados para reutilizá-los sempre que necessário. Esse processo evita a recomputação de soluções de subproblemas já resolvidos, o que, em termos práticos, leva a uma redução significativa do tempo de execução. Um exemplo clássico de aplicação de PD é o cálculo da sequência de Fibonacci, que tradicionalmente possui uma complexidade exponencial, mas pode ser resolvido em tempo linear com o uso de programação dinâmica. Outros algoritmos notáveis que fazem uso de PD incluem o algoritmo de Bellman-Ford, usado para encontrar o caminho mais curto em grafos, e o problema da mochila, que é um problema clássico de otimização combinatória. (Bellamy, 1972).

Conforme estudos mostram, a eficiência da Programação Dinâmica em relação a algoritmos de força bruta é notável, principalmente em cenários onde há repetição massiva de cálculos (Cholissodin; Riyandani, 2016). A principal vantagem da PD está na sua capacidade de reduzir a complexidade temporal de problemas complexos, tornando-os computacionalmente tratáveis, sem comprometer a exatidão da solução.

No entanto, a técnica impõe um maior consumo de memória, já que as soluções intermediárias dos subproblemas precisam ser armazenadas. Essa troca entre o tempo de execução e o consumo de memória é um dos principais desafios ao se utilizar a PD, particularmente quando aplicada em grandes conjuntos de dados

(Bellman, 1972; Lew, 2006). Este trabalho aborda a otimização de tempo de execução de algoritmos, sem aplicações no aprimoramento do uso da memória.

Dado o aumento contínuo no volume de dados e na necessidade de algoritmos rápidos e eficazes, a otimização dos tempos de execução é uma prioridade em várias áreas da ciência e tecnologia. O desenvolvimento de soluções eficientes que equilibram os recursos computacionais disponíveis tem sido amplamente discutido na literatura científica. Segundo Brasil e Dias (2017), os algoritmos de otimização computacional surgiram da necessidade de encontrar soluções apropriadas para problemas considerados complexos, e a programação dinâmica se destaca como uma das técnicas mais eficazes nesse sentido.

Nesse contexto, o presente artigo busca explorar como a PD pode ser aplicada para resolver problemas clássicos de otimização, com ênfase na sua capacidade de reduzir a complexidade temporal de algoritmos, proporcionando soluções práticas e escaláveis.

A fim de ilustrar os benefícios e limitações dessa técnica, o estudo utiliza a sequência de Fibonacci como exemplo central. Embora seja um exemplo relativamente 3 simples, ele destaca o contraste entre a abordagem de força bruta, que resulta em uma complexidade de $O(2^n)$, e a Programação Dinâmica, que reduz essa complexidade para $O(n)$ (Monegat *et al.*, 2020). Além disso, serão discutidos os trade-offs entre tempo de execução e uso de memória, bem como a viabilidade da implementação de PD em problemas de grande escala.

O objetivo principal deste trabalho é fornecer uma visão detalhada sobre a aplicação da Programação Dinâmica na otimização temporal de algoritmos, comparando sua eficiência com outras abordagens, como a força bruta.

Espera-se, com isso, demonstrar que, embora a PD envolva um maior consumo de memória, ela é vantajosa em termos de tempo de execução, especialmente em problemas que apresentam subproblemas repetitivos. Ademais, o artigo visa contribuir para a compreensão de como a PD pode ser utilizada em cenários práticos, identificando suas limitações e apontando possíveis áreas de aplicação futura.

2 REVISÃO DA LITERATURA

A Programação Dinâmica tem sido amplamente estudada e aplicada em diferentes campos da ciência, sendo reconhecida como uma técnica benéfica na otimização de algoritmos. Diversos autores discutem suas propriedades fundamentais, como a subestrutura e a sobreposição de subproblemas, que fazem com que seja uma ferramenta eficaz para a solução de problemas recursivos. (Bellman, 1972).

Bellman (1972), o criador da técnica, aponta a Programação Dinâmica como uma atuação bottom-up[3] e iterativa para a resolução de problemas com grande complexidade. Ele ressalta a importância de resolver problemas menores, que precisam ser resolvidos várias vezes, uma única vez, e armazenar os resultados para uso futuro. Para Bellman, essa abordagem impacta na eficiência do algoritmo, especialmente em problemas onde há uma sobreposição significativa de subproblemas.

Um dos exemplos mais citados na literatura sobre PD é o algoritmo para calcular a sequência de Fibonacci. O método tradicional de força bruta possui uma complexidade de tempo maior, devido ao grande número de subproblemas repetidos que precisam ser recalculados a cada chamada recursiva. Ao aplicar a Programação Dinâmica, a complexidade do algoritmo é reduzida, pois cada subproblema é resolvido uma única vez e o resultado é guardado para uso futuro em uma tabela. Esse exemplo simples ilustra o poder da PD para reduzir a complexidade temporal de algoritmos. (Scott; Marketos, 2014).

Em casos de otimização combinatória, tais quais *knapsack problem*[4] e o *shortest path problem*[5], a Programação Dinâmica também pode ser uma boa opção. O *knapsack problem* é um problema clássico de otimização em que se busca maximizar o valor de itens que podem ser colocados em uma mochila com capacidade limitada. Ao dividi-lo em partes menores e armazenar os resultados intermediários, a Programação Dinâmica apresenta uma solução ótima com uma complexidade muito

[3] De baixo pra cima
[4] Problema da mochila
[5] Problema dos caminhos mínimos

menor do que a de algoritmos de força bruta. (Abdelaziz; Torre; Alaya, 2021).

No âmbito da inteligência artificial, o uso da Programação Dinâmica é recorrente na aplicação de problemas de aprendizado por reforço e planejamento. No algoritmo de aprendizado por reforço chamado "value iteration", a PD é utilizada para calcular os valores ótimos de estados em um ambiente baseado em um modelo de Markov (Lox-Ley; Cheung, 2021). Nessa situação, a subestrutura ótima da PD permite a construção de soluções a partir de subsoluções menores, o que acelera significativamente o processo de aprendizado.

Além disso, Monegat *et al.* (2020) ressalta o uso da Programação Dinâmica em otimização na área de produção industrial. Nesse estudo, a PD foi utilizada para minimizar os custos de alocação de recursos em linhas de produção, resultando em economias significativas de tempo e custo. O estudo concluiu que a Programação Dinâmica é uma boa opção de ferramenta para a otimização de sistemas de produção complexos, oferecendo soluções rápidas e eficazes.

Apesar de suas vantagens, a Programação Dinâmica também apresenta algumas limitações. A principal desvantagem dessa técnica é o maior consumo de memória, já que as soluções intermediárias precisam ser armazenadas para evitar recomputações. Em problemas muito grandes, o uso excessivo de memória pode se tornar um fator limitante, especialmente em sistemas com recursos computacionais restritos. No entanto, existem técnicas, como a "PD com compactação de espaço", que buscam reduzir esse consumo, mantendo a eficiência temporal da PD. (Lew, 2006).

Dessa forma, a Programação Dinâmica apresenta-se como uma das técnicas mais eficazes para otimização de algoritmos, especialmente em problemas com sobreposição de subproblemas. Estudos recentes sugerem que o uso de PD em grandes conjuntos de dados e em problemas complexos de otimização oferece ganhos significativos em termos de tempo de execução, tornando-a uma escolha preferencial para muitos cientistas da computação e engenheiros. (Cechin *et al.*, 2019).

3 METODOLOGIA

A metodologia deste trabalho foi estruturada em três fases principais, que podem ser separadas em definição do problema, implementação dos algoritmos e análise dos resultados. Cada etapa foi planejada e executada com o objetivo de avaliar a eficiência da Programação Dinâmica em comparação com algoritmos de força bruta. A seguir, detalha-se cada uma dessas fases, descrevendo o conjunto de métodos utilizados e o caminho percorrido desde o início até a conclusão do estudo.

Este trabalho parte da seguinte questão de pesquisa: como a Programação Dinâmica pode ser aplicada para otimizar algoritmos utilizados pela programação usual? Para abordar essa questão, o problema escolhido para análise foi a geração da sequência de Fibonacci. Essa escolha se deve à simplicidade e à clareza desse problema, que permite uma fácil implementação tanto pelo método de força bruta quanto pela técnica de PD.

O objetivo central foi comparar a eficiência temporal de ambas as abordagens, verificando como a Programação Dinâmica pode reduzir o tempo de execução ao evitar recomputações desnecessárias, armazenando as soluções de subproblemas já resolvidos. Dessa forma, dois algoritmos distintos foram elaborados em Python para a geração da sequência de Fibonacci.

O ambiente de programação escolhido para o desenvolvimento dos algoritmos foi o Python, devido à sua popularidade, simplicidade e à vasta disponibilidade de bibliotecas e ferramentas para análise de desempenho.

Especificamente, foi utilizado Python, na versão 3.12.4, para implementar os algoritmos de força bruta e PD. Também foi utilizada a biblioteca time, para medir o tempo de execução básico de cada algoritmo, fornecendo uma maneira simples de medir o tempo decorrido durante a execução do código.

Além disso, o ambiente de desenvolvimento foi configurado em um sistema com processador Intel i5, 16 GB de RAM, e sistema operacional Windows 11 como configurações de hardware e software. Essas especificações garantiram que os testes pudessem ser realizados de forma eficiente e sem interferências significativas no desempenho das medições.

3.1 ALGORITMO DE FORÇA BRUTA

O primeiro algoritmo foi implementado utilizando a definição recursiva tradicional da sequência de Fibonacci, onde a função é chamada repetidamente para calcular os valores de F(*n*-1) e F(*n*-2) até que o valor da sequência seja alcançado. Este método foi escolhido para demonstrar os impactos da abordagem recursiva não otimizada, cuja complexidade temporal é $O(2^n)$. A implementação seguiu a fórmula básica F(*n*) = F(*n*-1) + F(*n*-2), sem qualquer técnica de otimização, permitindo que o mesmo subproblema fosse resolvido repetidas vezes. A seguir, é apresentado o código fonte usado no algoritmo de abordagem de força bruta.

```
def fibonacci(n):
    if n == 1 or n == 0
        return n
```


3.2 ALGORITMO DE PROGRAMAÇÃO DINÂMICA

O segundo algoritmo foi implementado utilizando a técnica de memorização da PD, onde as soluções de subproblemas são armazenadas em uma tabela (ou array) e reutilizadas quando necessário. Com isso, cada subproblema é resolvido uma única vez, excluindo recursivas redundantes. A complexidade temporal deste algoritmo é reduzida para O(*n*), já que, uma vez resolvido, cada subproblema pode ser acessado diretamente em tempo constante. Essa abordagem permite uma otimização significativa do tempo de execução, conforme descrito por Lew (2006). A seguir, é apresentado o código fonte usado no algoritmo de PD.

```
def fibonacciPD(n, table = {}):
        if n == 1 or n == 0
                return  n
        try:
                return table[n]
```

```
        except:
                table[n] = fibonacciPD(n-1) + fibonacci(n-2)
                return table[n]
```

3.3 IMPLEMENTAÇÃO

Após a implementação, os algoritmos foram testados repetidamente com diferentes entradas de dados (*n*), variando de valores pequenos até valores grandes de Fibonacci, como F(10), F(20), F(30), e assim por diante. Cada teste foi repetido 10 vezes pra cada valor de n, a fim de assegurar a consistência dos resultados e minimizar os efeitos de variações ocasionais no tempo de processamento.

Para garantir a precisão dos tempos de execução medidos, a biblioteca time foi utilizada, uma vez que ela fornece uma medição mais detalhada e precisa do tempo que cada função leva para ser executada. Abaixo, é mostrado o código fonte de testes, utilizando a biblioteca time.

```
import time
for n in range(10,1000):
        start = time.time()
        result = fibonacciPD(n)
        finish = time.time()
        print(n, round(finish - start, 12))
```

A principal métrica utilizada para avaliar o desempenho dos algoritmos foi a complexidade de tempo, expressa em termos da notação Big O. Com essa notação,

foi possível quantificar a eficiência relativa de cada algoritmo, permitindo comparações objetivas entre a abordagem de força bruta e a de PD.

Conforme previsto na literatura Loxley e Cheung (2021), esperava-se que o algoritmo de força bruta exibisse uma escalabilidade exponencial em termos de tempo de execução à medida que o valor de *n* aumentava, enquanto o algoritmo otimizado com PD apresentaria uma escalabilidade linear. Essa hipótese foi confirmada nos resultados experimentais, conforme será detalhado na seção de Resultados e Discussão.

Os testes foram realizados de maneira repetida, tanto para o algoritmo de força bruta quanto para o algoritmo otimizado com PD, permitindo que médias fossem calculadas para cada conjunto de experimentos. Isso garantiu que os tempos de execução obtidos fossem consistentes e representassem com precisão a performance de cada abordagem.

A comparação final entre os tempos de execução foi realizada com base nos resultados médios de cada conjunto de testes, e a notação Big O foi usada para quantificar a redução na complexidade temporal observada. O principal objetivo desta fase foi validar a hipótese de que a programação dinâmica oferece ganhos significativos de eficiência quando comparada aos métodos tradicionais de força bruta. Na próxima seção, faremos uma abordagem dos resultados.

4 RESULTADOS

Nesta seção, serão apresentados os resultados obtidos a partir da implementação dos algoritmos de força bruta e programação dinâmica para a geração da sequência de Fibonacci. Os tempos de execução foram registrados para diferentes valores de entrada *n*, a fim de comparar o desempenho de ambas as abordagens. Para assegurar a precisão das medições, cada teste foi repetido várias vezes, e as médias dos tempos de execução foram calculadas. A seguir, são detalhados os tempos de execução obtidos para os valores de *n* escolhidos.

A tabela 1 resume os tempos de execução médios (em segundos) para os algoritmos de força bruta e programação dinâmica em diferentes valores de *n*:

Tabela 1 – Tempos de execução médios

Valor de *n*	Tempo (s) - Força Bruta	Tempo (s) - Programação Dinâmica
0	$1{,}1920\times10^{-6}$	$2{,}3841\times10^{-6}$
3	$3{,}0994\times10^{-6}$	$3{,}3378\times10^{-6}$
7	$1{,}1205\times10^{-5}$	$3{,}5762\times10^{-6}$
11	$7{,}9870\times10^{-5}$	$2{,}3841\times10^{-6}$
15	0,0005	$2{,}1457\times10^{-6}$
19	0,0018	$1{,}6689\times10^{-6}$
23	0,0137	$4{,}5299\times10^{-6}$
27	0,0907	$2{,}3841\times10^{-6}$
31	0,6204	$2{,}1457\times10^{-6}$
35	4,2132	$2{,}1457\times10^{-6}$
39	32,5018	$2{,}1457\times10^{-6}$

Fonte: Os Autores.

Os dados da tabela mostram que, conforme o valor de *n* aumenta, o tempo de execução do algoritmo de força bruta cresce de maneira exponencial. Para pequenos valores de *n*, como 0 ou 1, o tempo de execução é muito baixo em ambas as abordagens. No entanto, à medida que o valor de *n* cresce, a técnica de força bruta passa a ter um desempenho significativamente menor, apresentando um aumento no tempo de execução.

Quando *n* = 3, o algoritmo de força bruta levou 3,0994x10^{-6} segundos, enquanto o de programação dinâmica foi capaz de resolver o mesmo problema em 3,3378x10^{-6} segundos. Observando os valores para *n* = 15, é possível notar o aumento da diferença entre os tempos de execução dos dois algoritmos, pois o de abordagem de força bruta saltou para 0,0005 segundos, enquanto a programação dinâmica manteve um tempo de 2,1457x10^{-6} segundos. A diferença se tornou ainda mais significativa para *n* = 39, visto que o algoritmo de força bruta levou 32,5018 segundos, enquanto o algoritmo otimizado por PD continuou a apresentar um tempo aproximadamente constante.

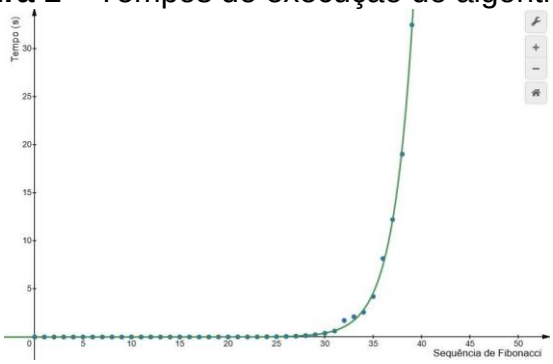
Esses resultados ilustram a diferença de desempenho entre as duas abordagens, especialmente em valores mais altos de *n*. O tempo de execução do algoritmo de programação dinâmica se manteve praticamente constante para todos

os valores de *n* testados, confirmando a sua eficiência ao evitar a recomputação de subproblemas já resolvidos.

Os tempos de execução apresentados indicam que, enquanto o algoritmo de força bruta sofre com um rápido crescimento de complexidade conforme o valor de *n* aumenta, a técnica de PD permanece estável e eficiente. Para valores de *n* maiores que 15, o tempo de execução do algoritmo de força bruta cresce rapidamente, tornando-se impraticável para aplicações que exijam soluções em tempo real ou em grandes escalas.

A Figura 1 mostra o gráfico do tempo de execução do algoritmo de força bruta, gerado na ferramenta Desmos. No eixo *y*, temos o tempo em segundos, enquanto o eixo *x* representa os números da sequência de Fibonacci.

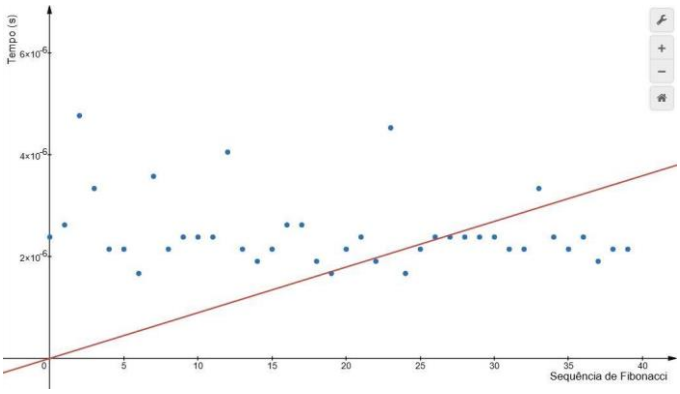
Figura 1 – Tempos de execução do algoritmo de força bruta



Fonte: Os Autores.

A Figura 2 apresenta o gráfico do tempo de execução do algoritmo de Programação Dinâmica. O gráfico mostra uma linha quase constante ao longo do eixo *x*, representando um tempo de execução baixo e praticamente inalterado para cada valor da sequência de Fibonacci. A equação de regressão ajustada é uma reta, $y = m \cdot x$, com $m = 8{,}9726 \times 10^{-8}$, e um valor de $R^2 = -3{,}977$, indicando que não houve um ajuste significativo entre o tempo e o índice de Fibonacci. Esse valor negativo de R^2 sugere que a relação linear não explica bem a variação dos dados, o que é comum em casos onde o tempo de execução é quase constante.

Figura 2 – Tempos de execução do algoritmo de PD



Fonte: Os Autores.

Esse crescimento mínimo no tempo de execução indica uma complexidade de tempo muito baixa, provavelmente O(1) (constante) ou O(*n*) (linear) para o algoritmo que gera a sequência de Fibonacci. Isso sugere o uso de uma abordagem otimizada, como um método iterativo ou com memorização, que evita recalcular valores repetidos, resultando em um desempenho muito mais eficiente do que o método de força bruta.

A redução de complexidade de tempo, de O(2^n) para O(*n*), obtida pela programação dinâmica é claramente refletida nos resultados observados. Na próxima seção, será realizada uma análise crítica desses resultados, comparando-os com as expectativas teóricas e discutindo as implicações da utilização da programação dinâmica em problemas de otimização, abordando as conclusões obtidas no projeto.

5 CONCLUSÃO E DISCUSSÃO DOS RESULTADOS

Nesta seção, os resultados obtidos durante a comparação dos tempos de execução dos algoritmos de força bruta e Programação Dinâmica para a geração da sequência de Fibonacci serão analisados a partir do que foi discutido na revisão da literatura.

Conforme abordado anteriormente, a principal vantagem da Programação Dinâmica é a sua capacidade de reduzir a complexidade temporal ao armazenar resultados de subproblemas, evitando recomputações desnecessárias. Diversos autores Scott e Marketos (2014) e Lew, (2006) destacam que a complexidade de

tempo de um algoritmo de força bruta para resolver o problema de Fibonacci é de $O(2^n)$, enquanto o uso de PD reduz essa complexidade para $O(n)$. A comparação entre os resultados obtidos neste estudo e as expectativas teóricas mostrou-se bastante consistente.

Nos testes realizados, observou-se que o algoritmo de força bruta começou a apresentar tempos de execução elevados a partir de valores de *n* maiores que 15. Para $n = 39$, por exemplo, o algoritmo de força bruta levou 32,5018 segundos, enquanto a PD manteve-se constante com $2{,}1457 \times 10^{-6}$ segundos. Esses valores confirmam as previsões da literatura, que indicam uma explosão combinatória do tempo de execução conforme o valor de *n* aumenta.

Os resultados também estão de acordo com as observações de Loxley e Cheung (2021), que apontaram que a PD é particularmente eficiente em problemas com sobreposição de subproblemas, como o de Fibonacci. O tempo de execução da PD manteve-se praticamente constante, essa estabilidade reforça a eficácia da PD em cenários onde o desempenho computacional é crítico.

A implementação do algoritmo de força bruta, embora seja simples e direta, revelou rapidamente suas limitações à medida que os valores de *n* aumentaram. A complexidade exponencial foi evidenciada pelo crescimento dramático dos tempos de execução, tornando a abordagem inviável para valores mais altos de *n*. Mesmo para valores intermediários, como $n = 15$, o tempo de execução de 0,005 segundos já indica que o algoritmo de força bruta não seria adequado para aplicações em larga escala ou em tempo real.

Em contrapartida, a implementação do algoritmo utilizando PD apresentou um desempenho bastante eficiente. Os resultados confirmam que a PD consegue resolver o problema de Fibonacci com complexidade $O(n)$, como esperado. Esse comportamento se deve à memorização das soluções de subproblemas, que elimina a necessidade de recomputação. A eficiência da PD foi observada mesmo para valores elevados de *n*, o que demonstra a eficácia desta técnica em cenários que envolvem cálculos repetitivos.

Os resultados obtidos neste estudo possuem implicações diretas para o desenvolvimento de algoritmos eficientes, especialmente em problemas que apresentam sobreposição de subproblemas. A Programação Dinâmica, como

evidenciado, oferece uma solução robusta para evitar o crescimento rápido da complexidade do problema, tornando-se uma ferramenta essencial para otimizar o desempenho em aplicações computacionais intensivas.

No contexto de problemas complexos que envolvem grandes conjuntos de dados, a escolha da abordagem correta — força bruta ou otimizada — pode determinar a viabilidade de uma solução em termos de tempo de execução. Neste estudo, o algoritmo de força bruta rapidamente se tornou inviável, enquanto a PD demonstrou que pode ser aplicada com sucesso para valores de *n* muito maiores, sem comprometer a eficiência computacional.

Esses resultados também confirmam os estudos de Brasil (2017), que apontam que a programação dinâmica é uma técnica extremamente eficaz quando aplicada a problemas de otimização, ao evitar cálculos redundantes, melhorando a eficiência do algoritmo. A comparação com a força bruta também serve para ilustrar a importância de se considerar a complexidade de tempo ao desenvolver soluções para problemas computacionais.

Embora a Programação Dinâmica tenha se mostrado altamente eficaz para o problema da sequência de Fibonacci, é importante destacar que essa técnica também tem suas limitações. O armazenamento das soluções de subproblemas exige memória adicional, e em casos onde o número de subproblemas é extremamente elevado, isso pode se tornar um gargalo. Assim, há um trade-off entre a economia de tempo e o uso de memória, conforme mencionado por Bellman (1972).

Para trabalhos futuros, seria interessante explorar como a Programação Dinâmica pode ser combinada com outras técnicas de otimização para balancear o uso de memória e tempo de execução, especialmente em problemas de maior complexidade. Além disso, a aplicação de PD em outros tipos de problemas, como na teoria de grafos e em algoritmos de otimização combinatória, poderia expandir o escopo dos resultados obtidos neste trabalho.

A comparação entre os algoritmos de força bruta e Programação Dinâmica para a geração da sequência de Fibonacci demonstrou claramente a superioridade da PD em termos de eficiência temporal. Enquanto a técnica de força bruta apresentou crescimento exponencial do tempo de execução, a Programação Dinâmica conseguiu manter um desempenho constante, mesmo para valores maiores de *n*.

Esses resultados confirmam as previsões teóricas encontradas na literatura e destacam a importância de se utilizar técnicas de otimização, como a PD, em problemas que envolvem cálculos repetitivos e grandes conjuntos de dados. Assim, a Programação Dinâmica se apresenta como uma abordagem essencial no desenvolvimento de algoritmos mais rápidos e eficientes, garantindo soluções viáveis para problemas complexos.

REFERÊNCIAS

ABDELAZIZ, F. B.; TORRE, D. L.; ALAYA, H. Dynamic programming and optimal control for vector-valued functions: A state-of-the-art review. **RAIRO - Operations Research**, v. 55, p. 351–364, 2021.

BELLMAN, R. **Dynamic Programming**. 6th ed. New Jersey: Princeton University Press, 1972.

BRASIL, C.; DIAS, J. Comparando algoritmos de otimização computacional aplicados ao problema de predição de estruturas proteicas com modelo HP-2D. **Revista Brasileira de Computação Aplicada**, v. 9, n. 3, p. 87–99, 2017.

CECHIN, R. B. et al. Programação dinâmica aplicada à redução de custos nas compras de vacinas de um hospital. **Revista de Medicina, Ribeirão Preto**, v. 52, n. 4, p. 287–294, 2019.

CHOLISSODIN, I.; RIYANDANI, E. Review: A State-of-the-Art of Time Complexity: Non-Recursive and Recursive Fibonacci Algorithm. **Journal of Information Technology and Computer Science**, v. 1, n. 1, p. 14–27, 2016.

LEW, A. **Dynamic Programming**: A Computational Tool. USA: Springer, 2006.

OXLEY, P.; CHEUNG, K. W. **A dynamic programming algorithm for finding an optimal sequence of informative measurements**. 2021. Disponível em: https://arxiv.org/abs/2109.11808. Acesso em: 01 out. 2024.

MONEGAT, A. D. R. et al. Aplicação de Programação Dinâmica para Otimização de um Layout de Produção: Um Estudo de Caso. **Tecno-Lógica**, Santa Cruz do Sul, v. 24, p. 317–323, 2020.

SCOTT, T. C.; MARKETOS, P. **On the origin of the Fibonacci Sequence**. 2014. Disponível em: https://mathshistory.st-andrews.ac.uk/Publications/fibonacci.pdf. Acesso em: 01 out. 2024.