Pontifícia Universidade Católica de Minas Gerais
Instituto de Ciências Exatas e Informática
Departamento de Engenharia de Computação

Relatório: Trabalho Prático 2

NP Completo - Partição

Professor: Walisson Ferreira de Carvalho

Arthur Gonçalves Ayres Lanna
Marcus Leandro Gomes Campos Oliveira
Rafael Ramos de Andrade

Belo Horizonte
Campus Coração Eucarístico

5 de dezembro de 2024

Conteúdo

1 Resumo

Este trabalho apresenta um estudo comparativo de abordagens para a resolução do problema da Partição, classificado como NP-completo. Foram implementados quatro algoritmos: um de força bruta (*standard*) e três baseados em programação dinâmica (*memorized top-down*, *bottom-up tabulation* e *bottom-up optimized*). O estudo inclui a análise teórica de cada método, sua complexidade computacional e testes empíricos realizados com diferentes tamanhos de vetores e variações de instâncias. Os resultados obtidos demonstram a superioridade dos algoritmos heurísticos em termos de eficiência temporal, especialmente para instâncias maiores, enquanto o método de força bruta foi utilizado como base comparativa. Este trabalho contribui para o entendimento de técnicas eficientes para problemas combinatórios, oferecendo uma análise prática e teórica do problema da Partição.
Palavras-chave: NP-completo, Problema da Partição, força bruta, programação dinâmica, análise de desempenho.

2 Introdução

Os problemas NP-completos ocupam um lugar de destaque na ciência da computação devido à sua complexidade intrínseca e relevância prática. Esses problemas pertencem à classe *NP* (tempo polinomial não determinístico), que engloba aqueles cuja solução pode ser verificada em tempo polinomial. Para que um problema seja classificado como NP-completo, ele deve satisfazer duas condições fundamentais: ser pertencente à classe *NP* e ser pelo menos tão difícil quanto qualquer outro problema em *NP*, no sentido de que qualquer problema em *NP* pode ser reduzido a ele por meio de uma transformação em tempo polinomial. Essa propriedade foi formalizada por Stephen Cook em 1971, no famoso *Teorema de Cook* [1], que estabeleceu o primeiro problema NP-completo, o problema da satisfazibilidade booliana (SAT).

O problema da Partição, foco deste trabalho, é um exemplo clássico de problema NP-completo [2]. Ele consiste em determinar se um conjunto finito de números inteiros pode ser dividido em dois subconjuntos cuja soma dos elementos seja igual. Formalmente, dado um vetor S de números inteiros, o objetivo é encontrar dois subconjuntos disjuntos S_1 e S_2, tais que $S_1 \cup S_2 = S$ e $\sum_{x \in S_1} x = \sum_{x \in S_2} x$. O problema é diretamente relacionado ao *Subset Sum Problem* e, consequentemente, ao problema da mochila (*Knapsack Problem*) [3], ambos também pertencentes à categoria NP-completo.

Historicamente, o problema da Partição foi amplamente estudado devido à sua simplicidade na formulação e à complexidade subjacente na solução. Embora o conceito seja mencionado em literatura matemática desde o século XIX, sua formalização no contexto de algoritmos e complexidade computacional [5] ganhou relevância durante a segunda metade do século XX, impulsionada pelo avanço da teoria da computação e pelo interesse em resolver problemas combinatórios desafiadores.

As aplicações do problema da Partição são variadas e abrangem diversas áreas práticas. Na computação, ele é frequentemente utilizado em algoritmos de balanceamento de carga, onde tarefas devem ser distribuídas de maneira equitativa entre dois ou mais processadores. Na otimização de sistemas, o problema aparece em cenários como alocação de recursos e particionamento de redes. Além disso, possui implicações em teoria de jogos e análise de riscos financeiros, onde é necessário balancear custos ou benefícios entre diferentes participantes.

Apesar de sua dificuldade computacional, o problema da Partição é de grande relevância prática, motivando o desenvolvimento de algoritmos eficientes, tanto exatos quanto heurísticos, para lidar com instâncias específicas. Este trabalho busca explorar as características do problema, analisar algoritmos clássicos e propor implementações que contribuam para o entendimento de sua solução e impacto prático.

3 Análise dos algoritmos

3.1 Algoritmo *Standard* (Força Bruta)

O algoritmo *Standard* utiliza uma abordagem de força bruta para resolver o problema da Partição. Ele examina todas as combinações possíveis dos elementos do conjunto fornecido, verificando se é possível dividi-lo em dois subconjuntos com somas iguais. Esse método baseia-se no conceito clássico de *Subset Sum*.

3.1.1 Código

```
def is_subset_sum(array_size, array, new_sum):
    # Base cases
    if new_sum == 0:
        return True
    if array_size == 0:
        return False

    # If element is greater than sum, then ignore it
    if array[array_size - 1] > new_sum:
        return is_subset_sum(array_size - 1, array, new_sum)

    # Check if sum can be obtained by any of the following:
    # (a) including the current element
    # (b) excluding the current element
    return is_subset_sum(
        array_size - 1, array, new_sum) or
    is_subset_sum(
        array_size - 1, array, new_sum - array[array_size - 1])

def equal_partition(array):
    # Calculate sum of the elements in array
    array_sum = sum(array)

    # If sum is odd, there cannot be two
    # subsets with equal sum
    if array_sum % 2 != 0:
        return False

    # Find if there is subset with sum equal
    # to half of total sum
    return is_subset_sum(len(array), array, array_sum // 2)
```

3.1.2 Funcionamento

O algoritmo opera por meio de chamadas recursivas para identificar subconjuntos cuja soma seja igual à metade da soma total do vetor. Para isso, ele segue os seguintes passos:

1. **Casos base**:
   - Se a soma alvo (*target sum*) for igual a zero, significa que um subconjunto com a soma desejada foi encontrado.
   - Se não houver mais elementos no conjunto, a soma alvo não pode ser alcançada.

2. **Ignorar elementos maiores que a soma alvo**: Se o elemento atual do conjunto for maior que a soma alvo, ele é ignorado na busca.

3. **Explorar possibilidades recursivamente**: O algoritmo tenta resolver o problema incluindo ou excluindo o elemento atual, verificando se qualquer uma das opções leva a uma solução válida.

3.1.3 Complexidade

O algoritmo apresenta complexidade de tempo exponencial, representada por $O(2^n)$, onde n é o número de elementos no vetor. Isso ocorre porque, em cada nível de recursão, o algoritmo considera duas possibilidades para cada elemento: incluí-lo ou excluí-lo do subconjunto.

- **Complexidade de tempo**: $O(2^n)$.
- **Complexidade de espaço**: $O(n)$, correspondente à profundidade máxima da pilha de chamadas recursivas.

3.1.4 Análise

Apesar de sua simplicidade conceitual, o algoritmo *standard* é computacionalmente inviável para instâncias grandes, devido ao número exponencial de chamadas recursivas. Entretanto, ele serve como referência básica para avaliar o desempenho de algoritmos mais avançados, sendo fundamental em estudos comparativos.

3.2 Algoritmo Memorized Top-Down

O algoritmo *memorized top-down* é uma otimização da abordagem de força bruta que utiliza a técnica de programação dinâmica para reduzir cálculos redundantes. Ele armazena os resultados de subproblemas resolvidos em uma estrutura de memorização, reutilizando-os sempre que necessário.

3.2.1 Código

```
# Python program to partition a Set
# into Two Subsets of Equal Sum

def equal_partition_bottom_up(array):
    # Calculate sum of the elements in array
    array_sum = sum(array)
    array_size = len(array)

    # If sum is odd, there cannot be two
    # subsets with equal sum
    if array_sum % 2 != 0:
        return False

    array_sum = array_sum // 2

    # Create a 2D array for storing results
    # of subproblems
    table = [[False] * (array_sum + 1) for _ in range(array_size + 1)]

    # If sum is 0, then answer is true (empty subset)
```

```
    for i in range(array_size + 1):
        table[i][0] = True

    # Fill the table in bottom-up manner
    for i in range(1, array_size + 1):
        for j in range(1, array_sum + 1):
            if j < array[i - 1]:
                # Exclude the current element
                table[i][j] = table[i - 1][j]
            else:
                # Include or exclude
                table[i][j] = table[i - 1][j] or table[i - 1][j - array[i -

    return table[array_size][array_sum]
```

3.2.2 Funcionamento

A abordagem consiste em resolver o problema de forma recursiva, mas evitando a repetição de cálculos por meio da reutilização de resultados previamente armazenados. O processo ocorre da seguinte forma:

1. **Estrutura de memorização**: Uma tabela ou dicionário é criada para armazenar o resultado de cada subproblema, identificando-os por uma combinação única de parâmetros (como o índice atual e a soma alvo).
2. **Casos base**:
   - Se a soma alvo (*target sum*) for igual a zero, considera-se que um subconjunto válido foi encontrado.
   - Se não houver mais elementos no vetor (*array size* igual a zero), a soma alvo não pode ser atingida.
3. **Reutilização de resultados**: Antes de resolver um subproblema, o algoritmo verifica se o resultado já está armazenado na estrutura de memorização. Se estiver, o cálculo é evitado.
4. **Explorar possibilidades recursivas**: Caso o subproblema ainda não tenha sido resolvido, o algoritmo calcula o resultado considerando a inclusão ou exclusão do elemento atual do subconjunto.

3.2.3 Complexidade

Graças à memorização, o número de subproblemas distintos é reduzido em comparação à força bruta, resultando em uma complexidade mais eficiente.

- **Complexidade de tempo**: $O(n \cdot k)$, onde n é o número de elementos no vetor e k é a soma alvo (metade da soma total).
- **Complexidade de espaço**: $O(n \cdot k)$, devido ao armazenamento dos resultados na tabela de memorização.

3.2.4 Análise

O algoritmo *memorized top-down* representa um grande avanço em eficiência, reduzindo o número de chamadas recursivas e permitindo que instâncias de tamanho moderado sejam resolvidas com desempenho satisfatório.

3.3 Algoritmo Bottom-Up Tabulation

3.3.1 Código

```python
# Python program to partition a Set
# into Two Subsets of Equal Sum

def equal_partition_bottom_up(array):
    # Calculate sum of the elements in array
    array_sum = sum(array)
    array_size = len(array)

    # If sum is odd, there cannot be two
    # subsets with equal sum
    if array_sum % 2 != 0:
        return False

    array_sum = array_sum // 2

    # Create a 2D array for storing results
    # of subproblems
    table = [[False] * (array_sum + 1) for _ in range(array_size + 1)]

    # If sum is 0, then answer is true (empty subset)
    for i in range(array_size + 1):
        table[i][0] = True

    # Fill the table in bottom-up manner
    for i in range(1, array_size + 1):
        for j in range(1, array_sum + 1):
            if j < array[i - 1]:
                # Exclude the current element
                table[i][j] = table[i - 1][j]
            else:
                # Include or exclude
                table[i][j] = table[i-1][j] or table[i-1][j - array[i-1]]

    return table[array_size][array_sum]
```

O algoritmo *bottom-up tabulation* implementa uma abordagem iterativa para resolver o problema da Partição. Ele utiliza programação dinâmica para construir uma tabela que representa todas as combinações possíveis de somas alcançáveis pelos subconjuntos do vetor.

3.3.2 Funcionamento

A abordagem *bottom-up* elimina a recursão, substituindo-a por uma construção iterativa baseada em uma tabela de dimensões $n \times k$, onde n é o número de elementos no vetor e k é a soma alvo.

1. **Inicialização da tabela**: Uma tabela booliana é criada, onde cada entrada *dp[i][j]* indica se é possível alcançar a soma j utilizando os primeiros i elementos do vetor.
2. **Preenchimento iterativo**:
   - Para cada elemento do vetor, o algoritmo atualiza a tabela com base nas somas alcançáveis com e sem o elemento atual.

- Se a soma j puder ser alcançada com o elemento atual, ou se já for alcançável sem ele, a entrada correspondente na tabela é marcada como verdadeira.

3. **Verificação do resultado**: Após preencher a tabela, a entrada *dp[n][k]* indica se é possível particionar o vetor em dois subconjuntos com somas iguais.

3.3.3 Complexidade

A abordagem iterativa do algoritmo melhora a eficiência computacional, eliminando a necessidade de uma pilha de chamadas recursivas.

- **Complexidade de tempo**: $O(n \cdot k)$, devido ao preenchimento iterativo da tabela.
- **Complexidade de espaço**: $O(n \cdot k)$, correspondente ao tamanho da tabela de programação dinâmica.

3.3.4 Análise

O algoritmo *bottom-up tabulation* é mais eficiente em termos de tempo e espaço quando comparado à abordagem recursiva com memorização. Ele é amplamente utilizado em problemas de otimização combinatória devido à sua robustez e simplicidade.

3.4 Algoritmo Bottom-Up Optimized

O algoritmo *bottom-up optimized* é uma variação do método *bottom-up tabulation*, com melhorias específicas na utilização de memória. Em vez de utilizar uma tabela bidimensional, ele reduz o espaço necessário para apenas uma única dimensão, representando as combinações possíveis de somas de maneira compacta.

3.4.1 Código

```
# Python program to partition a Set
# into Two Subsets of Equal Sum

def equal_partition_bottom_up_optimized(array):
    # Calculate sum of the elements in array
    array_sum = sum(array)

    # If sum is odd, there cannot be two
    # subsets with equal sum
    if array_sum % 2 != 0:
        return False

    array_sum = array_sum // 2

    array_size = len(array)
    previous_row = [False] * (array_sum + 1)
    current_row = [False] * (array_sum + 1)

    # Mark previous_row[0] = true as it is true
    # to make sum = 0 using 0 elements
    previous_row[0] = True

    # Fill the subset table in
```

```
    # bottom-up manner
    for i in range(1, array_size + 1):
        for j in range(array_sum + 1):
            if j < array[i - 1]:
                current_row[j] = previous_row[j]
            else:
                current_row[j] = (previous_row[j] or
                previous_row[j-array[i-1]])

        previous_row = current_row.copy()

    return previous_row[array_sum]
```

3.4.2 Funcionamento

A otimização baseia-se no fato de que, ao preencher a tabela de programação dinâmica, cada linha depende apenas da linha anterior. Assim, é possível substituir a tabela bidimensional por um vetor unidimensional, atualizado iterativamente.

1. **Inicialização do vetor**: Um vetor booliano é criado, onde cada posição *dp[j]* indica se é possível alcançar a soma j utilizando os elementos do vetor processados até o momento.
2. **Atualização em ordem inversa**: Para cada elemento do vetor original, o vetor *dp* é atualizado de forma decrescente (do maior valor de soma para o menor). Isso garante que cada soma seja avaliada apenas uma vez por iteração, evitando sobreposição de cálculos.
3. **Verificação do resultado**: Após processar todos os elementos, a entrada *dp[k]* indica se é possível particionar o vetor em dois subconjuntos com somas iguais, sendo k igual à metade da soma total do vetor.

3.4.3 Complexidade

A otimização na utilização de memória reduz significativamente a complexidade espacial do algoritmo, tornando-o ideal para instâncias de grande escala.

- **Complexidade de tempo**: $O(n \cdot k)$, onde n é o número de elementos no vetor e k é a soma alvo.
- **Complexidade de espaço**: $O(k)$, pois o vetor unidimensional utiliza apenas o espaço necessário para armazenar as somas possíveis.

3.4.4 Análise

O algoritmo *bottom-up optimized* combina a eficiência temporal do método *bottom-up tabulation* com uma redução significativa no uso de memória. Ele é especialmente adequado para problemas onde o espaço disponível é um recurso crítico, mantendo alta eficiência mesmo em casos de maior escala.

4 Metodologia

Este trabalho utilizou métodos empíricos para avaliar o desempenho de diferentes algoritmos aplicados ao problema da Partição. Os experimentos foram conduzidos utilizando conjuntos de dados gerados aleatoriamente e divididos em três categorias principais de teste: *Single Length Tests*, *Multiple Length Tests* e *Comparative Tests*. Os resultados de cada experimento foram armazenados em arquivos no formato CSV, sendo posteriormente analisados e representados graficamente.

4.1 Execução dos Testes

4.1.1 Testes com Comprimento Único

Para os testes de comprimento único (*Single Length Tests*), arrays de tamanho fixo foram gerados aleatoriamente com valores inteiros no intervalo de 1 a 100. Cada experimento consistiu em executar um algoritmo selecionado sobre múltiplos arrays do mesmo comprimento, totalizando uma quantidade definida de repetições (`tests_quantity`). Durante cada execução, o tempo de processamento foi registrado em segundos, bem como o resultado obtido pelo algoritmo. Os dados gerados foram salvos no diretório `Tests/Single_Length_Tests`.

4.1.2 Testes com Múltiplos Comprimentos

Nos testes com múltiplos comprimentos (*Multiple Length Tests*), os experimentos foram configurados para gerar arrays de tamanhos variados, permitindo a análise do comportamento dos algoritmos em função do aumento do tamanho do vetor. Assim como nos testes anteriores, arrays com valores aleatórios entre 1 e 100 foram utilizados, e os tempos de execução e resultados foram registrados. Os arquivos de saída foram salvos no diretório `Tests/Multiple_Length_Tests`.

4.1.3 Testes Comparativos

Os testes comparativos (*Comparative Tests*) foram realizados para comparar diretamente o desempenho de todos os algoritmos implementados no trabalho. Para isso, arrays com comprimentos variados foram gerados, e cada algoritmo foi executado sobre os mesmos conjuntos de dados. Os resultados de tempo de execução e respostas foram registrados para posterior análise. Os dados gerados foram armazenados no arquivo `Tests/Combined_Tests/comparative_tests.csv`, e gráficos comparativos foram produzidos no diretório correspondente.

4.2 Formato e Organização dos Dados

Os resultados de cada execução foram organizados em arquivos CSV com os seguintes campos:

- **Array size**: Tamanho do vetor utilizado no teste.
- **Result**: Resultado retornado pelo algoritmo (verdadeiro ou falso).
- **Execution time (s)**: Tempo de execução do algoritmo, em segundos, registrado com precisão de até 10 casas decimais.

Os arquivos CSV foram posteriormente utilizados para gerar gráficos representativos em formato PNG, organizados nos subdiretórios `Graphs` dentro de cada categoria de teste.

4.3 Decisões de Implementação

Em relação à estrutura e distribuição dos dados de entrada, optou-se por uma abordagem prática que incluiu:

- Utilização de arrays aleatórios para simular diferentes cenários.
- Definição de intervalos e tamanhos de vetores baseados em boas práticas observadas na literatura de análise de algoritmos.
- Registro detalhado de tempos de execução e resultados para garantir a replicabilidade e confiabilidade dos experimentos.

Com essa abordagem, foi possível gerar dados consistentes para análise comparativa, garantindo a robustez dos resultados apresentados.

5 Resultados e Análise

5.1 Testes de múltiplas instâncias

Os testes realizados com conjuntos de instâncias múltiplas evidenciam diferenças significativas no desempenho dos algoritmos de programação dinâmica implementados. A partir de $N = 40$, os tempos de execução começam a crescer exponencialmente, com o algoritmo *standard* sendo incapaz de lidar com tamanhos maiores de entrada. Entre os algoritmos dinâmicos, verificamos a seguinte ordem de eficiência (do pior para o melhor): **memorized top-down**, **tabulation bottom-up** e **bottom-up optimized**.

Essas diferenças podem ser explicadas pela natureza das técnicas empregadas em cada abordagem:

- **Memorized Top-Down**: Apesar de reduzir a redundância por meio da memorização, essa técnica sofre com sobrecarga adicional de chamadas recursivas, o que prejudica a performance em casos de instâncias maiores. Cada chamada implica custos adicionais de manipulação de pilha e verificações no armazenamento intermediário (memória cache).
- **Tabulation Bottom-Up**: Ao substituir a recursão por uma abordagem iterativa, este método elimina a sobrecarga associada às chamadas recursivas. No entanto, ele ainda depende de tabelas de tamanho fixo que podem consumir memória adicional em casos de problemas muito grandes.
- **Bottom-Up Optimized**: Este método apresenta a melhor performance por otimizar o uso de memória. Em vez de armazenar toda a tabela, ele utiliza apenas as entradas necessárias no momento, reduzindo significativamente o espaço utilizado e, em consequência, o tempo de execução.

O gráfico do algoritmo Stantdard demonstra claramente a limitação do algoritmo padrão em lidar com tamanhos maiores, evidenciando sua inviabilidade prática para casos mais complexos. Além disso, os outros três gráficos ilustram a ordem crescente de eficiência, com a diferença de tempos de execução entre os algoritmos se tornando mais evidente à medida que o tamanho das instâncias aumenta.

Figura 1: Testes de múltiplas instâncias com o algoritmo *standard*

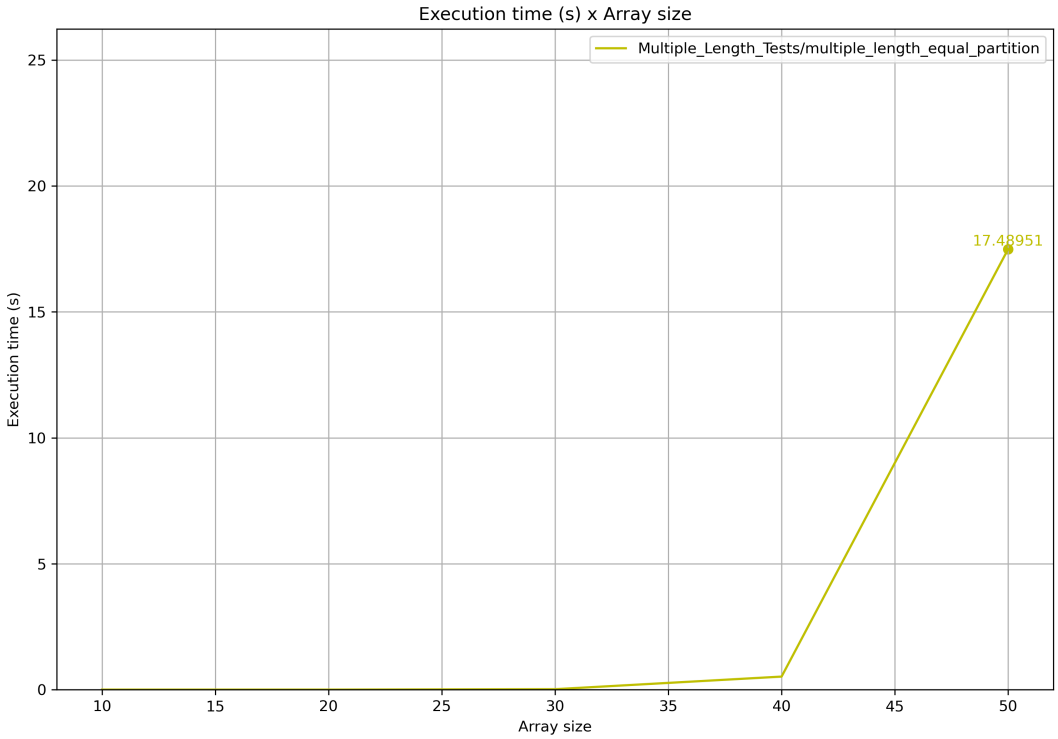


Figura 2: Testes de múltiplas instâncias com o algoritmo *memorized top-down*

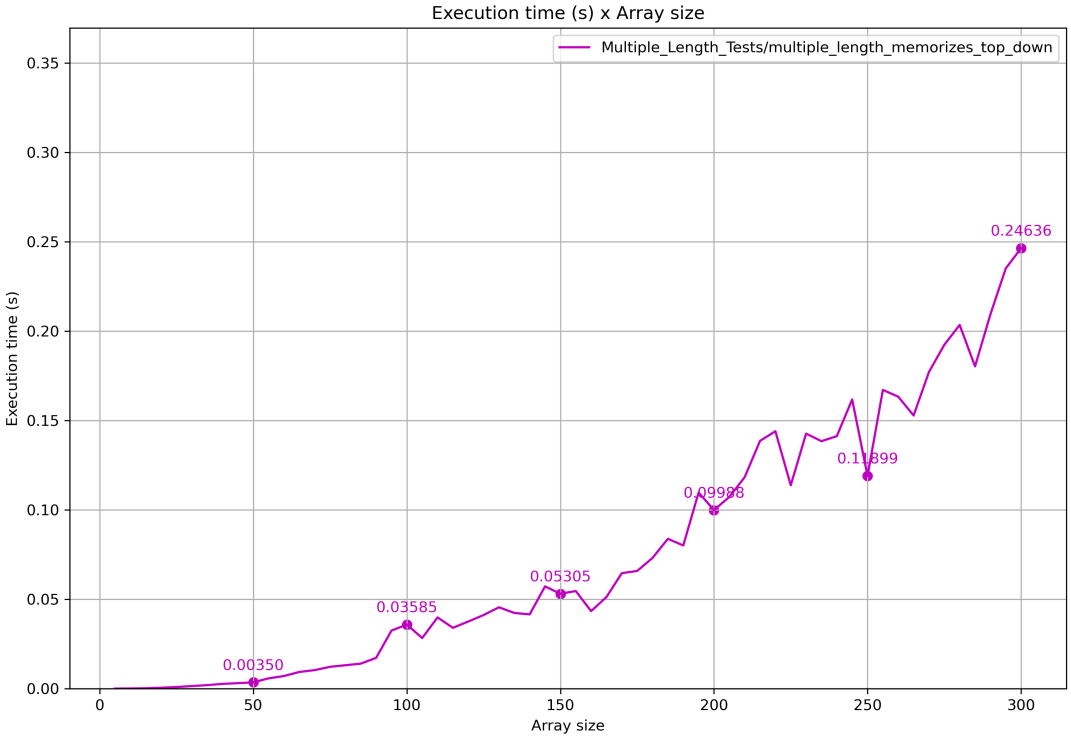


Figura 3: Testes de múltiplas instâncias com o algoritmo *tabulation bottom-up*

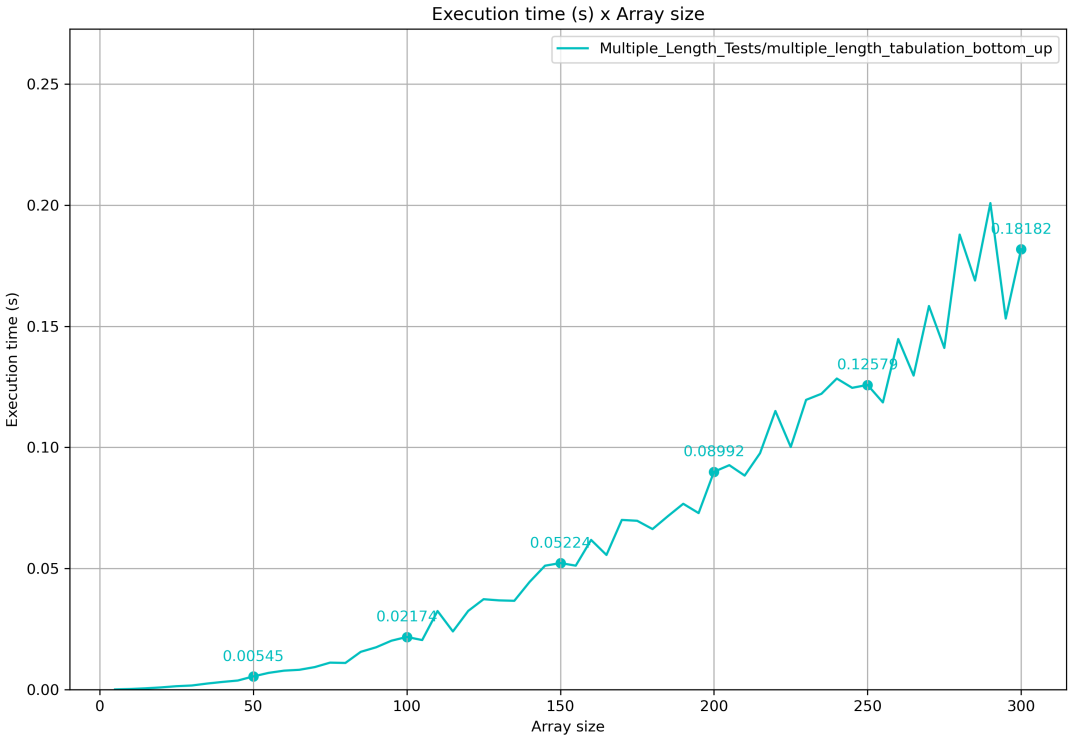
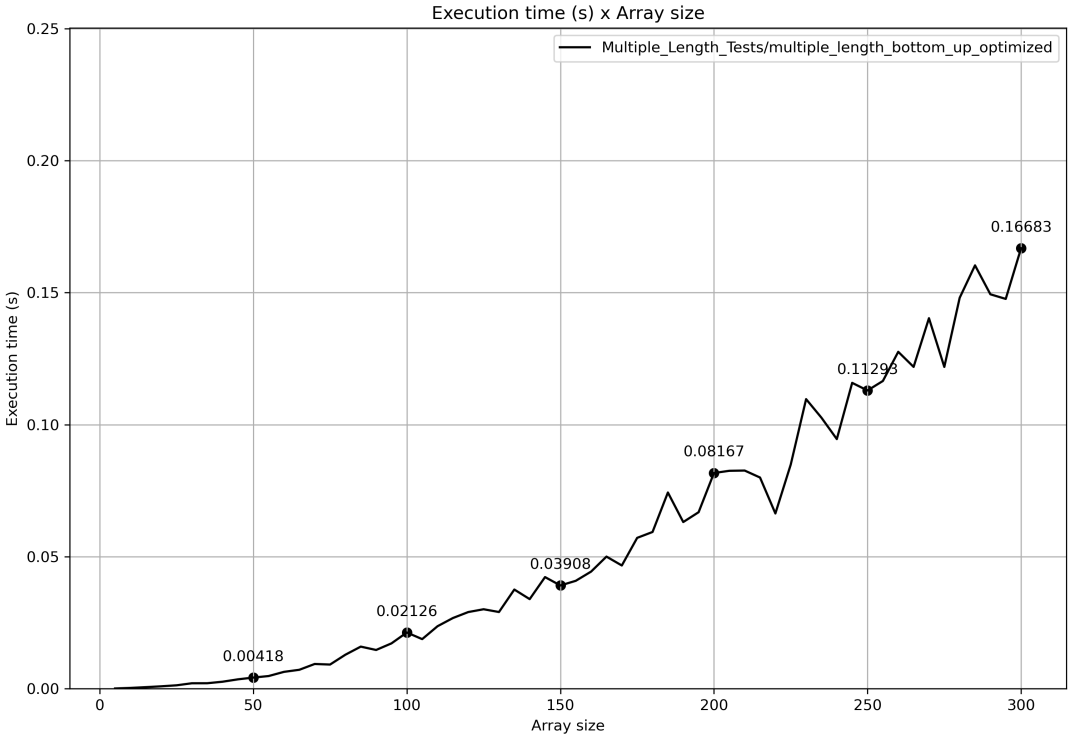


Figura 4: Testes de múltiplas instâncias com o algoritmo *bottom-up optimized*



5.2 Comparação geral de eficiência

Os testes comparativos consolidam os resultados obtidos nos testes individuais, destacando os pontos fortes e fracos de cada abordagem. A análise destes resultados mostra que, enquanto os algoritmos dinâmicos são significativamente mais rápidos em instâncias maiores, o *bottom-up optimized* se destaca como o mais eficiente devido ao equilíbrio entre consumo de memória e tempo de execução.

Os gráficos abaixo resumem esses achados, apresentando comparações diretas de desempenho entre todos os algoritmos avaliados.

Figura 5: Resultados comparativos de eficiência entre os algoritmos

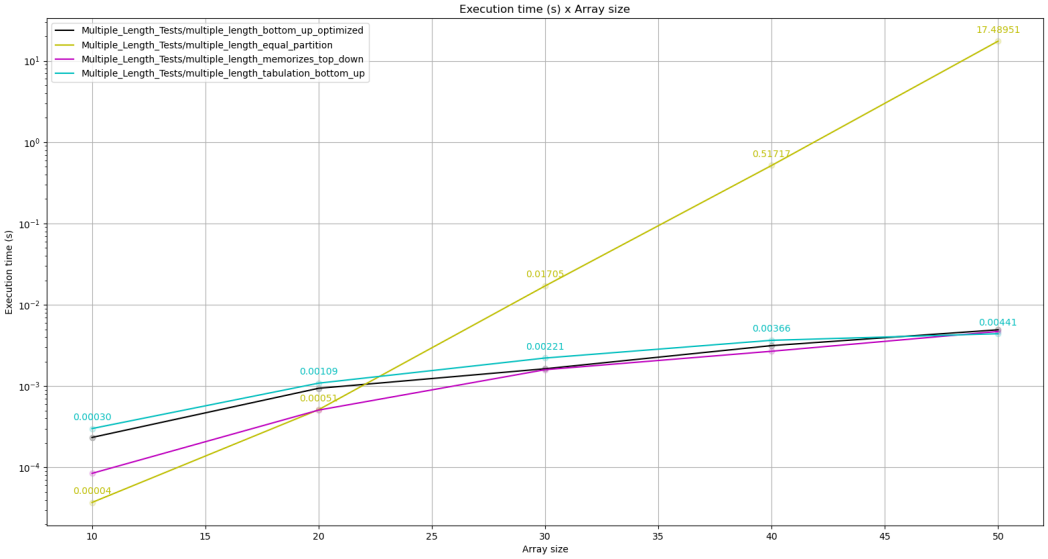
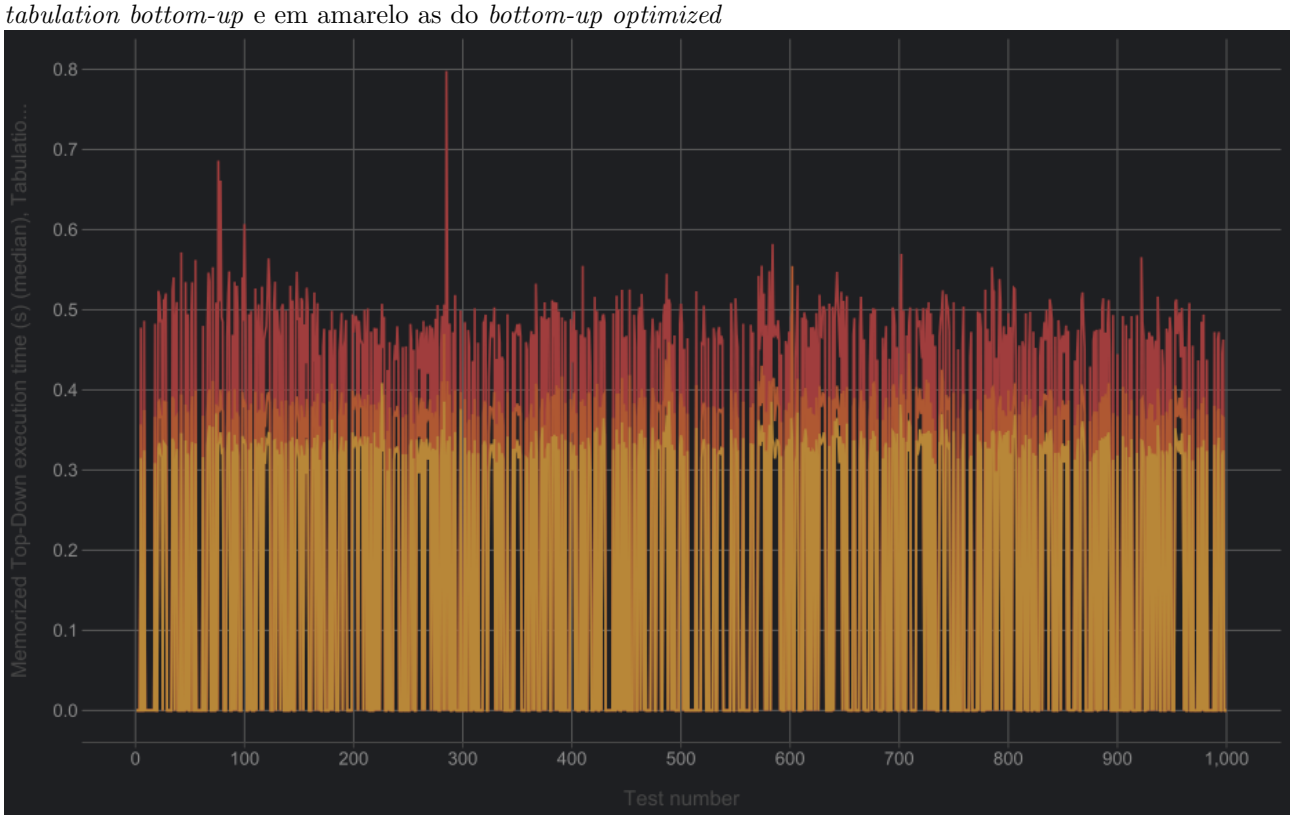


Figura 6: Resultados comparativos de eficiência entre os algoritmos, para tamanhos maiores de vetor, com exceção do algoritmo *Standard*. Em vermelho, execuções do algoritmo *memorized top-down*, em laranja as do *tabulation bottom-up* e em amarelo as do *bottom-up optimized*



6 Conclusão

Neste trabalho, foi investigada a eficiência de diferentes abordagens para a resolução do problema da Partição Igual, um problema clássico classificado como NP-completo. Foram implementados e analisados quatro algoritmos principais: *standard*, *memorized top-down*, *tabulation bottom-up* e *bottom-up optimized*. A partir de uma série de testes, foram comparados os desempenhos dessas abordagens em diferentes cenários.

Os resultados experimentais demonstraram variações significativas na eficiência entre os algoritmos. O algoritmo *standard* mostrou-se inadequado para entradas de tamanhos maiores, evidenciando as limitações da abordagem recursiva sem otimização, devido à explosão combinatória de chamadas recursivas. Por outro lado, os algoritmos baseados em programação dinâmica apresentaram melhor desempenho. Entre eles, o *bottom-up optimized* destacou-se como a abordagem mais eficiente, devido ao seu uso otimizado de memória e à eliminação de sobrecarga associada a estruturas adicionais.

Além disso, foi possível observar que o *memorized top-down*, embora funcional, apresentou desempenho inferior quando comparado às abordagens iterativas, especialmente para entradas de maior dimensão. O *tabulation bottom-up* obteve resultados intermediários, demonstrando ser uma solução eficiente, porém superada pela versão otimizada.

Este estudo contribui para o entendimento das diferenças práticas entre algoritmos recursivos e iterativos na solução de problemas computacionalmente intensivos. A análise realizada reforça a importância de estratégias de otimização em algoritmos de programação dinâmica, destacando como essas melhorias podem impactar diretamente a eficiência e escalabilidade. Assim, o trabalho oferece uma base sólida para futuras implementações e estudos relacionados à resolução de problemas NP-completos.

Referências

[1] COOK, S. The Complexity of Theorem-Proving Procedures. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (STOC '71), pp. 151-158. ACM, New York, NY, USA, 1971.

[2] KARP, R. M. Reducibility Among Combinatorial Problems. In: *Complexity of Computer Computations* (pp. 85-103). Springer, 1972.

[3] SCHROEppel, R.; SHAMIR, A. A Simple Method for Solving the Subset-Sum Problem. *SIAM Journal on Computing*, v. 10, n. 3, p. 598-608, 1981.

[4] CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. *Introduction to Algorithms*. 3rd ed. MIT Press, 2009.

[5] GAREY, M. R.; JOHNSON, D. S. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman and Company, 1979.

7 Anexo

Github: https://github.com/ArthurLanna36/np-complete-partition