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

4 de dezembro de 2024

Conteúdo

1 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* (nondeterministic polynomial time), 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*, que estabeleceu o primeiro problema NP-completo, o problema da satisfatibilidade booleana (SAT).

O problema da Partição, foco deste trabalho, é um exemplo clássico de problema NP-completo. 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*), 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 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.

2 Algoritmos

- **Recursão simples**: A abordagem recursiva explora todas as possíveis partições do conjunto, caracterizando-se por uma complexidade exponencial $O(2^n)$. Apesar de ser conceitualmente simples, essa abordagem não é eficiente, dado o número significativo de subproblemas repetidos, o que a torna impraticável para conjuntos maiores [4].

- **Top-Down com memoization**: Essa técnica melhora a eficiência da recursão utilizando uma tabela (ou cache) para armazenar os resultados dos subproblemas já resolvidos. Dessa forma, evita-se a recalculação redundante, reduzindo a complexidade para $O(n \cdot T)$, onde T é a soma total dos elementos do conjunto. O uso de memoization é especialmente vantajoso em problemas onde apenas uma parte das combinações possíveis é explorada, como no cálculo de subsequências ou caminhos otimizados [4].

- **Bottom-Up com tabulação**: Neste método, resolve-se os subproblemas menores iterativamente, partindo dos casos base até o problema completo. A tabulação utiliza uma tabela para armazenar os resultados intermediários de forma sistemática, garantindo que

cada subproblema seja resolvido uma única vez. Essa abordagem elimina a sobrecarga de chamadas recursivas e o risco de estouro de pilha, com a mesma eficiência de $O(n \cdot T)$. Exemplos incluem algoritmos para subsequências e cálculo de somas parciais [4, 1].

- **Bottom-Up com otimização de espaço**: Esta variação da tabulação reduz o consumo de memória ao utilizar apenas o armazenamento necessário para resolver o subproblema atual. Em vez de manter uma tabela completa, utiliza-se um número fixo de variáveis ou uma estrutura compacta que armazena apenas as últimas iterações relevantes. Essa técnica é eficiente em problemas lineares, como a resolução do problema da mochila ou subsequências [4, 1].

3 Metodologia

4 Resultados e análise

Os testes foram realizados em uma máquina com Ryzen 7 5700x e 33 GB de RAM 3200 MHz e os gráficos gerados utilizando Pandas e MatPlotLib.

4.1 Tempo de execução com vetores aleatórios

Os próximo testes foram realizados com vetores aleatórios buscando uma ánalise individual de cada algoritmo, demonstrando a tendência de crescimento quando aumentamos o tamanho do vetor, é importante salientar que em cada teste geramos um vetor aleatório de tamanho maior que o anterior, como a soma dos valores influenciam tanto quanto o tamanho do vetor, quanto maior o vetor, mais a soma dos valores aleatórios influencia no tempo de execução causando instabilidade nos resultados.

Figura 1: Equal Partition $O(2^n)$

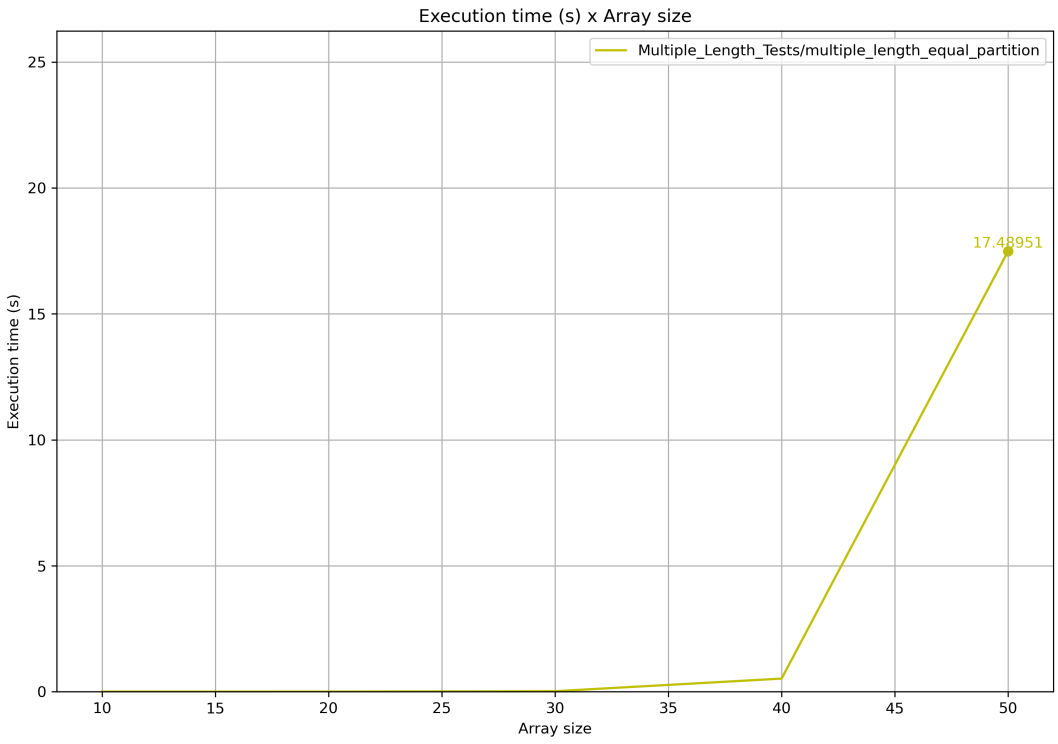


Figura 2: Bottom up optimized $O(n \cdot T)$

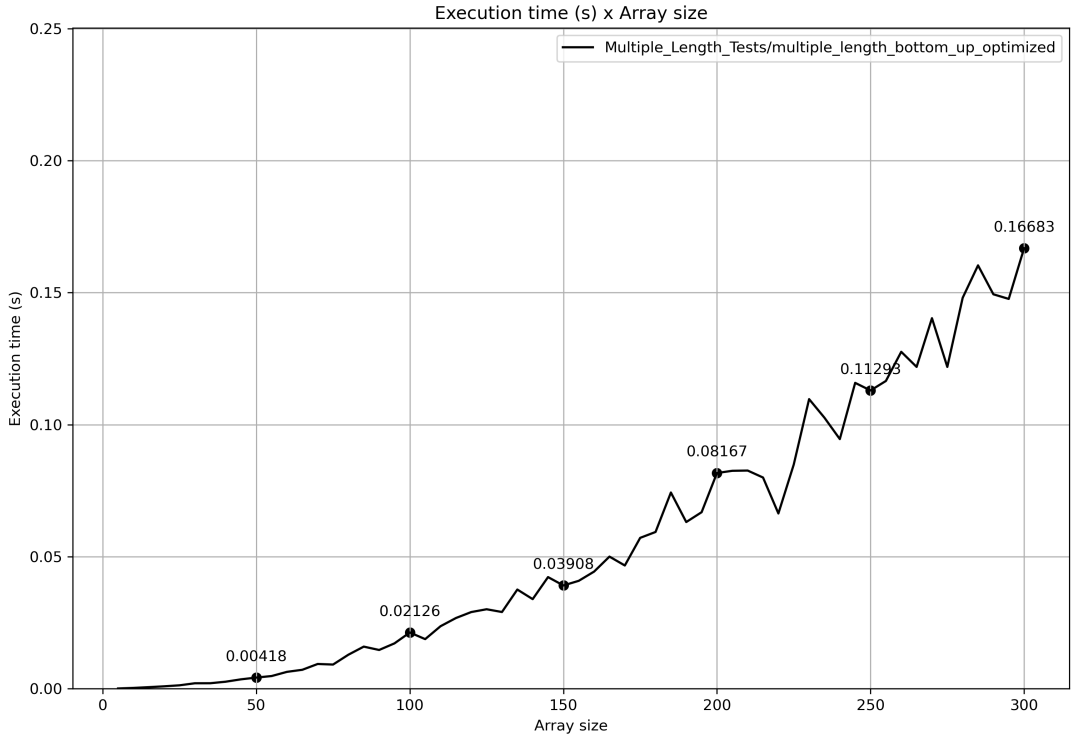


Figura 3: Memorizes Top Down $O(n \cdot T)$

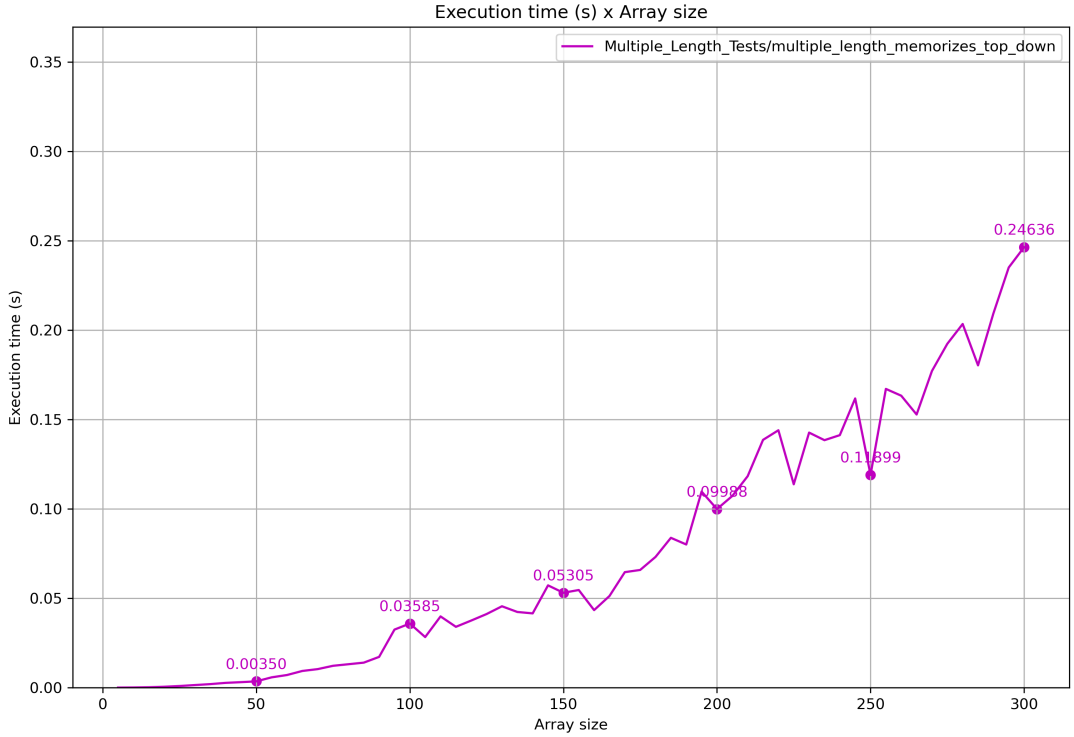
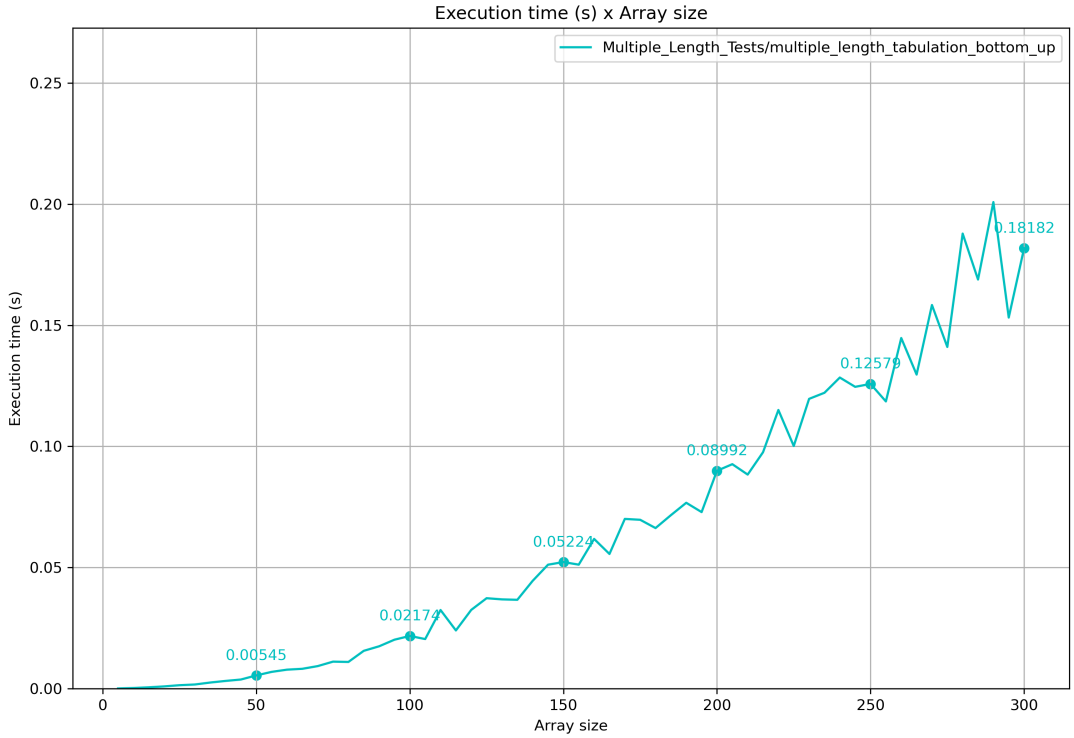


Figura 4: Tabulation Bottom Up $O(n \cdot T)$

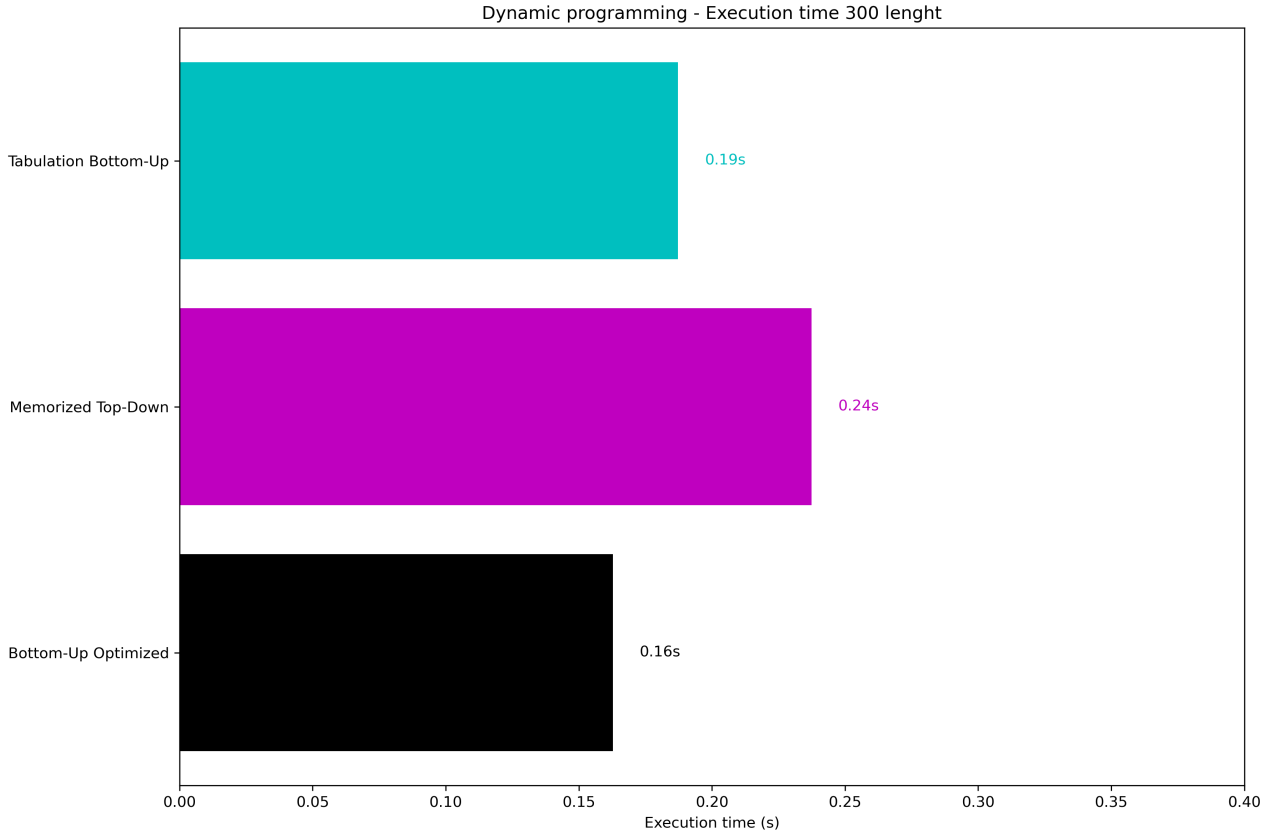


Ao analisarmos os tempos de execução, é possível observar a diferença significativa entre as implementações usando programação dinâmica e em relação ao uso da força bruta (Equal Partition). Para vetores maiores que 22 a implementação por força bruta domina assintoticamente todas as outras implementações. No entanto, para tamanhos menores que 22, ela se mostra extremamente rápida, executando 7 vezes mais rápida que a implementação utilizando programação dinâmica bottom-up, além disso, as implementações utilizando programação dinâmica apresentam um desempenho parecido como era esperado.

4.2 Tempo de execução com vetores iguais

Nesse testes é gerado um vetor aleatório de tamanho 300, cada algoritmo com excessão do algoritmo de força bruta é executado tendo como entrada esse vetor. Este teste tem como objetivo comparar o desempenho dos algoritmos que utiilizam de programação dinâmica para resolverem o problema.

Figura 5: Teste algoritmos de Programação Dinâmica $O(n \cdot T)$



5 Conclusão

Referências

[1] Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

[2] Karp, R. M. (1972). Reducibility Among Combinatorial Problems. *Complexity of Computer Computations*, 85-103.

[3] Martello, S., & Toth, P. (1981). *Knapsack Problems: Algorithms and Computer Implementations*. Wiley.

[4] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

[5] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

6 Anexo