# Projeto e Análise de Algoritmos
# Problema da Mochila

**Thiago O. da Silva[1], Vicente C. G. de S. Carvalho[2], Rhuan M. O. M. de Carvalho[3]**

[1]*Departamento de Ciência da Computação – Universidade Federal do Piauí (UFPI)*
*Teresina – PI – Brazil*

[2]*Departamento de Ciência da Computação – Universidade Federal do Piauí (UFPI)*
*Teresina – PI – Brazil*

[3]*Departamento de Ciência da Computação – Universidade Federal do Piauí (UFPI)*
*Teresina – PI – Brazil*

{thiago_silva,rhuanmourao}@ufpi.edu.br, vicentecleyton@gmail.com

***Abstract.*** *This paper aims to present a practical application for the Knapsack problem, which is an NP-Complete problem, implement and evaluate solutions for it.*

***Resumo.*** *Este artigo tem como objetivo apresentar uma aplicação prática para o problema da Mochila, que é um problema NP Completo, implementar e avaliar soluções para a mesma.*

## 1. Definição do Problema

Dada uma mochila com capacidade limitada e um conjunto de objetos, cada um com seu valor e peso, o objetivo é determinar a combinação de objetos que maximize o valor total colocado na mochila, respeitando a restrição de capacidade.

Em termos matemáticos, temos um conjunto de $n$ objetos, onde cada objeto $i$ possui um valor $v_i$ e um peso $w_i$. A capacidade máxima da mochila é representada por $W$. A tarefa é selecionar uma combinação de objetos que maximize a soma dos valores, sem exceder a capacidade da mochila.

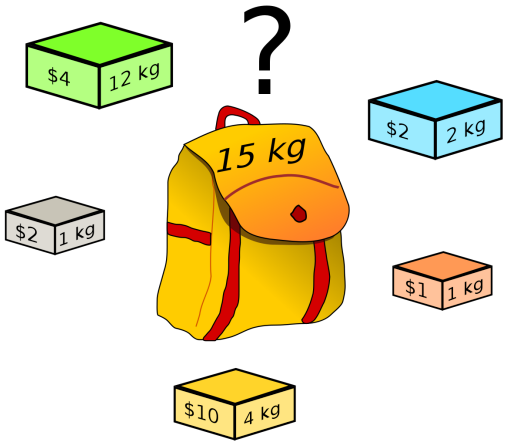


**Figure 1. Problema da Mochila**

## 2. Aplicação do Problema

Em um futuro apocalíptico, Marcos trabalha na Space XXX, uma fabricante de sistemas aeroespaciais, transporte espacial e comunicações. A humanidade está em guerra com Aliens em Marte, e o trabalho de Marcos na empresa é enviar armas através de naves espaciais para ajudar nossos soldados. Para isso, ele decidiu usar um algoritmo que otimizasse o transporte bélico, maximizando o poder bélico das armas enviadas. Assim, a nave possui sua área dimensional $W$ (medida em m²), cujo objetivo é armazenar um conjunto de armas, sem haver empilhamento, onde cada uma destas ocupa um espaço $X$ (em m²), e possui o dano que pode causar $Y$ (medido em Joules).

## 3. Implementação de uma instância

```
class Arma {
    constructor(area, dano) {
        this.area = area;
        this.dano = dano;
    }
}

// Função para gerar um número aleatório dentro de um intervalo
function gerarNumeroAleatorio(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

// Gerar as entradas de armas
const armas = [];
const quantidadeArmas = 20;
const capacidadeFoguete = 30;

var areaTotal = 0

for (let i = 0; i < quantidadeArmas; i++) {
    const area = gerarNumeroAleatorio(0.5, 10); // area aleatória entre
    ↪  0.5 e 10 metros
    const dano = gerarNumeroAleatorio(100, 1000); // Dano aleatório
    ↪  entre 100 e 1000 Joules

    const arma = new Arma(area, dano);
    armas.push(arma);

}
```

## 4. Implementação do algoritmo de força bruta

```
const { combinacao, danoTotal } = mochilaForcaBruta(armas,
↪  capacidadeFoguete);

function mochilaForcaBruta(armas, capacidade) {
    const quantidadeArmas = armas.length;
    const quantidadeCombinacoes = 2 ** quantidadeArmas; // número total
    ↪  de combinações possíveis
    let melhorDano = 0;
    let melhorComb = [];

    // Percorre todas as combinações possíveis de armas
    for (let i = 0; i < quantidadeCombinacoes; i++) {
        const combAtual = [];
        let areaAtual = 0;
        let danoAtual = 0;

        // Percorre todas as armas disponíveis
        for (let j = 0; j < quantidadeArmas; j++) {
            if ((i & (1 << j)) !== 0) {
                combAtual.push(armas[j]);
                areaAtual += armas[j].area;
                danoAtual += armas[j].dano;
            }
        }

        if (areaAtual <= capacidade && danoAtual > melhorDano) {
            melhorDano = danoAtual;
            melhorComb = combAtual;
        }
    }

    return { combinacao: melhorComb, danoTotal: melhorDano };
}
```

### 4.1. Análise da complexidade

Sabendo que *N* se refere à quantidade de armas disponíveis, infere-se que:

Geração das armas: A geração das armas possui uma complexidade de O($N$), pois envolve um loop simples que percorre a quantidade especificada.

Cálculo do número total de combinações: O cálculo da quantidadeCombinacoes é feito elevando 2 à potência da quantidadeArmas. Portanto, a complexidade é O($2^N$).

Loop externo: O loop externo percorre todas as combinações possíveis, ou seja, executa quantidadeCombinacoes iterações. Portanto, a complexidade é O($2^N$).

Loop interno: O loop interno percorre todas as armas disponíveis, que tem tamanho igual à quantidadeArmas. Portanto, a complexidade é O($N$).

Dessa forma, a complexidade total do algoritmo é O($2^N * N$).

## 5. Provar que o Problema da Mochila é NP-Completo

Para provar que um problema A pertence a NP-Completo é necessário provar que:

- O problema A pertence à classe NP(Nondeterministic Polynomial), apresentando em algoritmo não-determinista em tempo polinomial;
- Todo problema em NP é redutível para A em tempo polinomial.

### 5.1. Provar que o Problema está em NP

Para provar que um problema está na classe de complexidade NP, é preciso demonstrar que uma solução proposta para o problema pode ser verificada em tempo polinomial. Isso significa que, dada uma solução candidata, deve ser possível verificar em tempo polinomial se essa solução é válida ou não.

O procedimento KnapsackND é um algoritmo Não-Determinista para o problema de decisão da Mochila.
KnapsackND (cM,uM, n, P[1..n], U[1..n], X[1..n])

1 **for** i=1 to n
2 **do**
3 X[i] = *escolhe*(0,1);
4 **if**($\sum_1^n P[i] * X[i] > cM$) OR ($\sum_1^n U[i] * X[i] < uM$)
5 **then return** *Insucesso*
6 **else return** *Sucesso*

As linhas de 1 a 3 atribuem o valor 0/1 para o vetor solução X[i], $0 \leq$ i $\leq n$. Na linha 4 é feito um teste para verificar se a atribuição dos pesos é viável e não ultrapassa a capacidade da Mochila cM e se o resultado da Utilidade é pelo menos uM. Uma solução com sucesso é encontrada se as restrições são satisfeitas. A complexidade de tempo deste procedimento é O($n$).

### 5.2. Redução Polinomial

Todo problema que pertencente à Classe NP pode se reduzir em tempo polinomial, a um dos problemas NP-completo. Então, para provar que o problema da Mochila é NP-completo vamos fazer uma redução polinomial a partir do problema Subset Sum(Problema da Soma de Subconjunto).

**Problema da mochila**
**Entrada:** Um conjunto de $n$ itens, tal que todo $i$ tem utilidade $ci$ e peso $wi$, uma mochila tem capacidade $W$ e um valor positivo $C$.

**Questão:** Existe um subconjunto S $\subseteq \{1, ..., n\}$ de itens, tais que o peso total é no máximo $W$:

$$\sum_{i \in S} w_i \leq W$$

e o valor da utilidade total é pelo menos C:

$$\text{c(S)} = \sum_{i \in S} c_i \geq C \text{ ?}$$

**Subset Sum**
**Entrada:** Um conjunto finito $A$, um tamanho inteiro positivo $si$ para cada elemento $i$ $\in \$A$ , e um inteiro positivo $B$.

**Questão:** existe um subconjunto $A' \subseteq A$ tal que $\sum_{i \in A'} s_i = B$ ?

**Exemplo de uma instância válida:**

Conjunto $A$ =$\{2, 5, 8, 11\}$
$n$ = 4
Valor alvo $B$ = 13

Neste exemplo, o subconjunto $A'$ = $\{2, 11\}$ é uma solução válida, pois a soma é igual a 2+11=13, que é igual ao valor alvo $B$.

Agora, cada item do conjunto $A$ é um elemento da mochila, que chamamos de conjunto $U$.

Dado uma instância do problema Subset Sum, podemos fazer a redução para uma instância do Problema da Mochila da seguinte forma:

$$U = A, wi = ci = si, W = C = B$$

Esta redução é feita em tempo polinomial, desde que todas as atribuições sejam executadas em tempo polinomial.

Com uma resposta ”sim” para uma instância do Problema da Mochila significa que existe um subconjunto $S' \in U$ tal que:

$$\sum_{i \in S'} w_i \leq W \text{ e } \sum_{i \in S'} c_i \geq C$$

Isto significa que que existe um subconjunto $A' \in A$ tal que

$$B \leq \sum_{i \in A'} s_i \leq B$$

Isto é,

$$\sum_{i \in A'} s_i = B$$

Assim, por definição, é uma resposta ”sim “para problema Subset Sum, ou seja, reduz polinomialmente para o Problema da Mochila.
Logo, uma resposta ”não” para o Problema da Mochila, significa que tal conjunto não existe, que também é uma resposta negativa para o problema Subset Sum.

Com isso, provamos que:

- O problema da Mochila está em NP;
- Existe uma redução polinomial a partir do problema Subset Sum.

Logo, o Problema da Mochila é NP-Completo.

## 6. Solução Heurística de Maior Poder Bélico (Algoritmo Guloso)

Essa abordagem considera o valor das armas (medido pelo dano que podem causar) e sua eficiência em ocupar espaço. Com isso, essa heurística torna-se eficaz porque prioriza armas que possuem uma alta relação entre dano e espaço ocupado.

### 6.1. Planejamento

- Ordenação por Eficiência: Ordena as armas em ordem decrescente de sua eficiência em relação ao espaço ocupado. Isso pode ser feito dividindo o dano (Joules) pelo espaço ocupado (m²) para obter a razão.
- Seleção Iterativa: Insere as armas na nave espacial, começando com a arma mais eficiente, até que não haja mais espaço disponível na nave.
- Avaliação do Poder Bélico Total: Calcula o poder bélico total alcançado com as armas que foram selecionadas.
- Saída: A saída será a configuração de armas que maximiza o poder bélico dentro das restrições de espaço da nave.

### 6.2. Implementação

```
function mochilaGulosa(armas, capacidade) {
    const combinacao = [];
    let danoTotal = 0;
    let areaTotal = 0;

    // Calculando a razão de cada arma
    for (let i = 0; i < armas.length; i++) {
        const arma = armas[i];
        arma.razao = arma.dano / arma.area;
    }

    // Ordenando as armas por razão em ordem decrescente
    armas.sort((a, b) => b.razao - a.razao);

    // Adicionando os itens um a um na mochila, seguindo a ordem, até
    ↪  que não caiba mais nenhum ou todos os itens foram adicionados
    for (let i = 0; i < armas.length; i++) {
        const arma = armas[i];

        // Se a área da arma atual não ultrapassar a capacidade do
        ↪  foguete
        if (areaTotal + arma.area <= capacidade) {
            // Adiciona a arma na combinação (mochila)
            combinacao.push(arma);
            danoTotal += arma.dano;
            areaTotal += arma.area;
        }
    }

    return { combinacao, danoTotal, areaTotal };
}
```

### 6.3. Análise da complexidade

Sabendo que *N* se refere a quantidade de armas disponíveis, infere-se que:

Calculando a razão de cada arma: Isso é feito em um loop simples que percorre todas as armas. Portanto, a complexidade é O($N$).

Ordenando as armas: A ordenação é realizada usando um algoritmo de ordenação padrão do javascript, cuja complexidade é O($N * logN$).

Adicionando os itens na nave: Para cada arma, verifica se ela pode caber na nave. Como isso é feito em ordem decrescente de razão, cada arma é adicionada no máximo uma vez. Portanto, a complexidade para essa etapa é O($N$).

Dessa forma, a complexidade total do algoritmo é O($N * logN$).

## 7. Avaliação e Conclusão

Neste artigo foram implementados e avaliados dois algoritmos para o Problema da Mochila: Força Bruta e Guloso, a partir de três conjuntos de entradas diferentes referentes a quantidade de armas.

### 7.1. Condições de processamento:

- Javascript utilizando NodeJS;
- Windows 10;
- AMD Ryzen 7 1700;
- 2x8 GB de RAM 2800 MHz DDR4.

### 7.2. Tempo de execução

| Tempo de execução | | | |
|---|---|---|---|
| | 25 | 30 | 35 |
| **Força Bruta** | 4.243s | 152.003s | 5882.095s |
| **Guloso** | 0.115s | 0.115s | 0.116s |

**Figure 2. Tabela (Força Bruta x Guloso)**

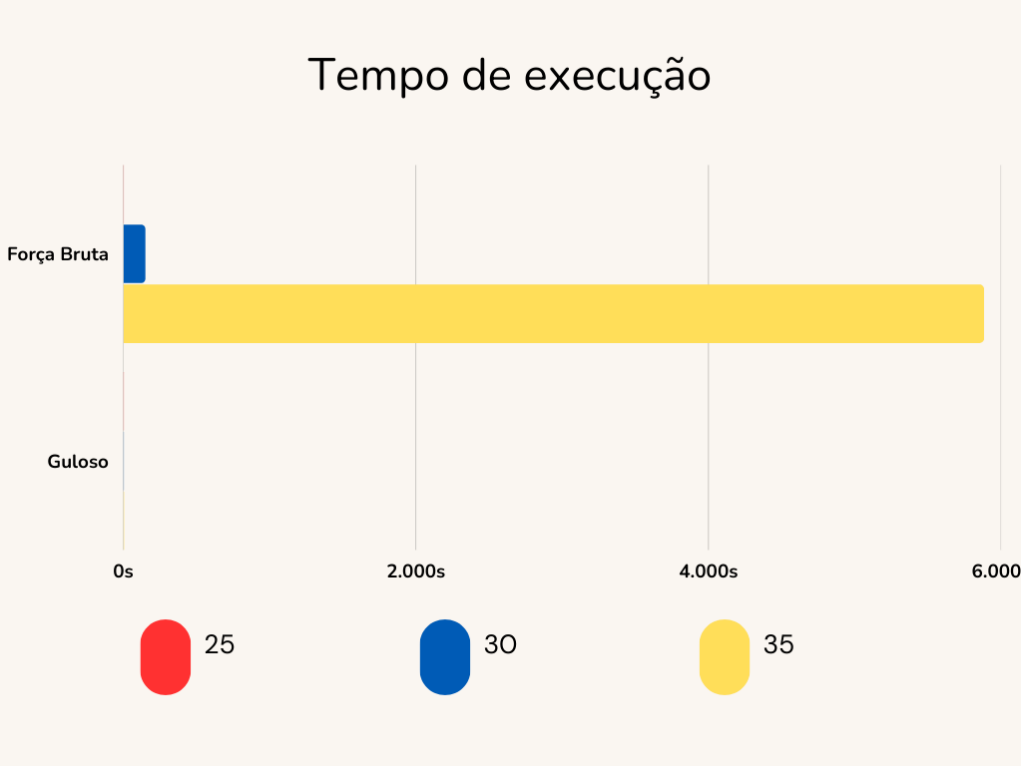


**Figure 3. Gráfico (Força Bruta x Guloso)**

### 7.3. Comparações realizadas

| Comparações realizadas | | | |
|---|---|---|---|
| | 25 | 30 | 35 |
| **Força Bruta** | 1291845683 | 49392123936 | 24178516322560 |
| **Guloso** | 146 | 181 | 218 |

**Figure 4. Tabela (Força Bruta x Guloso)**

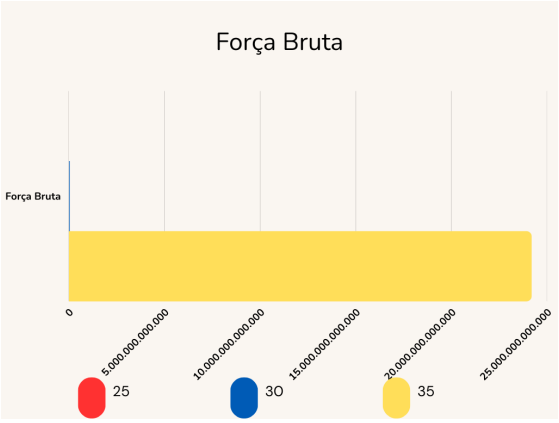


**Figure 5. Gráfico (Força Bruta)**

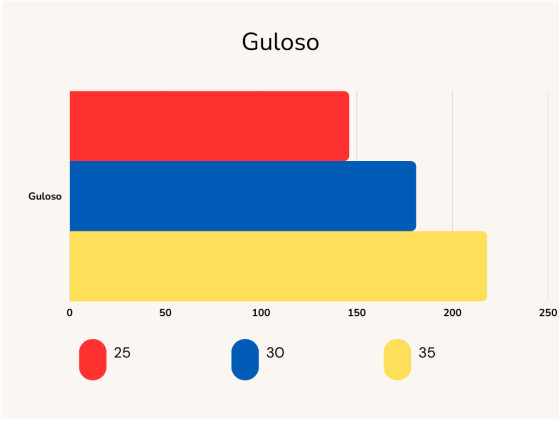


**Figure 6. Gráfico (Guloso)**

### 7.4. Conclusão

Considerando que a capacidade da nave foi fixada em 30, o algoritmo de Força Bruta, apesar de garantir a solução ótima, apresenta uma complexidade exponencial, inviabilizando sua aplicação prática mesmo para instâncias de tamanho moderado. Por exemplo, para um conjunto de entrada de apenas 35 armas, o tempo de execução foi de quase 2 horas.

Já o algoritmo Guloso, apesar de não garantir a solução ótima, tem complexidade polinomial e se mostrou extremamente eficiente na prática. Para o mesmo conjunto de 35 armas, seu tempo de execução foi de apenas 0,116 segundos, uma melhora de mais de 50.000 vezes em relação ao Força Bruta.

Essa grande diferença de desempenho se explica porque o Guloso faz uma busca direcionada, priorizando a seleção de itens por dano/área em vez de enumerar exaustivamente todas as possibilidades. Dessa forma, ele encontra rapidamente uma boa solução aproximada para a mochila.

Portanto, para o problema abordado neste artigo, fica evidente que o uso de uma abordagem heurística como o Algoritmo Guloso é crucial para viabilizar a obtenção de boas soluções em tempo hábil. Técnicas exatas como o Força Bruta, apesar da garantia de otimalidade, tem limitações claras de escalabilidade e uso prático.

## 8. Bibliografia

Referência:
Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2002). Algoritmos: Teoria e Prática. Editora Campus. Rio de Janeiro, Brasil.