

# **ANÁLISE DE ALGORITMOS**

Variações para solucionar o problema da mochila

Outros códigos no [GitHub](#)

## Índices

1. Objetivos
2. Testes de Corretude
3. Gerando Entradas
4. Tabela de Tempos
  - 4.1. Comentários sobre complexidade de tempo e memória
5. Conclusão
6. Equipamento Utilizado

## 1 Objetivos

Estudar os seguintes algoritmos para solução do problema da mochila (Knapsack problem): **recursão, recursão com memorização, programação dinâmica e aproximação**, fazendo as comparações de tamanho de entrada vs tempo e analisando complexidade de tempo e memória para todos os algoritmos.

Obs. Quando eu copio o código do VS Code e colo aqui neste doc ele perde a indentação. O código também está no [github](#).

## 2 Testes de corretude

Foi usada a seguinte função para fazer teste de corretude quando possível:

```
def _verify(weights, values, capacity, n, memo, my_ans):
# run the 3 algorithms with the same input
# if they return the same ans as mine then it may be right...

ans1 = knapsack_recursive(weights, values, capacity, n)
ans2 = knapsack_memoization(weights, values, capacity, n, memo)
ans3 = knapsack_dynamic(weights, values, capacity, n)

return (ans1 == my_ans) and (ans2 == my_ans) and (ans3 == my_ans)
```

*weights, values, capacity, n* é uma entrada para o problema.  
*my\_ans* é o retorno que obtive para a tal entrada.

Usa a mesma entrada para 3 algoritmos diferentes, se todos retornarem a mesma resposta que *my\_ans* é provavel que esteja correto.

### 3 Gerando as Entradas

As entradas para os testes foram geradas com o seguinte algoritmo:

```
def gen_input_size(n):  
    # given size 'n', generate 'weights' and 'values' arrays with 'n' items  
  
    # where  
    # 2 <= weight[i] <= 50  
    # 2 <= value[i] <= 50  
  
    weights = []  
    values = []  
  
    for i in range(n):  
        randomWeight = random.randint(2, 50)  
        weights.append(randomWeight)  
  
        randomValue = random.randint(2, 50)  
        values.append(randomValue)  
  
    with open('input.txt', 'w') as file:  
        print("weights = ", weights, file=file)  
        print("values = ", values, file=file)
```

- Foram geradas entradas de tamanho  $n = 25, 50$  e  $75$ .
- A capacidade da mochila sempre sera de  $100$ .

## 4 Tabela de tempos

N (# itens)	Tempo p/ Recursivo Direto (ms)	Tempo p/ Recursivo com Memo (ms)	Tempo p/ Programaçã o Dinâmica (ms)	Tempo p/ Aproximação (ms)
25	29.46	1.27	1.30	0.03
50	38155	4.06	2.71	0.04
75	275078	10.69	5.36	0.09

### 4.1 Comentários sobre complexidade de tempo e memória

- O **recursivo direto** pode gerar uma **call stack muito grande** aumentando o tempo de execução e o **gasto de memória**. Tem complexidade de **tempo  $O(2^n)$**  no pior caso, quando cada item pode ou não ser incluído (estrutura das chamadas recursivas forma uma árvore binária).
- O **recursivo com memorização** vai alocar uma tabela de dimensões  $N+1 \times C+1$ , onde  $N$  é a quantidade de itens e  $C$  é capacidade da mochila, isso **evita a recomputação** de alguns subproblemas, o que torna o algoritmo mais rápido que o Recursivo Direto, mas também pode **gastar muita memória**. Complexidade de **tempo e memória são  $O(N \times C)$** .
- A solução com **PD (Programação Dinâmica)** precisa **alocar uma matriz** de dimensões  $N+1 \times C+1$  e não usa recurção. Sua complexidade de **tempo e memória são  $O(N \times C)$** .
- A complexidade de **tempo** para o **algoritmo de aproximação** é  **$O(n \cdot \lg n)$**  pois ele ordena o vetor (**value-to-weight ratio**) e a de

**memória é  $O(n)$**  pois ele **aloca um vetor** de tamanho  $n$ . Ele não garante uma resposta “100% certa”.

## 5 Conclusão

A solução **recursiva** sem nenhuma otimização **não é eficiente** pois pode **gastar muito tempo e memória**.

Utilizando **memorização** na solução recursiva **melhora o tempo** porém o **consumo de memória** ainda pode ser **muito alto**.

A solução com **programação dinâmica** tem **eficiência** similar a da recursão com memorização e **gasta menos memória** por não usar recursão, é **ideal para casos onde a entrada é grande e é preciso retornar o valor 100% correto para tal entrada**.

O algoritmo de **aproximação (greedy)** é o **mais eficiente** porém pode retornar uma **resposta aproximada**, pode ser **usado em casos onde uma aproximação para a solução é o suficiente**.

## 6 Equipamento utilizado

Hardware Model	Dell Inc. Inspiron N5110
Memory	8.0 GiB
Processor	Intel® Core™ i5-2410M CPU @ 2.30GHz × 4
Graphics	AMD® Caicos / Mesa Intel® HD Graphics 3000 (SNB GT2)
Disk Capacity	256.1 GB
OS Name	Ubuntu 22.04.2 LTS