UNIVERSIDADE FEDERAL DE UBERLÂNDIA

CIÊNCIA DA COMPUTAÇÃO

DISCENTES:

LEONARDO HENRIQUE CARVALHO OLIVEIRA – 12311BCC003

RAQUEL EMILLENE FREIRE THOMÉ – 12311BCC026

YAN LUCAS SANTOS MARRA – 12311BCC013

ANÁLISE DE ALGORITMOS

PROGRAMAÇÃO DINÂMICA – PROBLEMA DA MOCHILA 0/1

DOCENTE:

Dra. MÁRCIA APARECIDA FERNANDES

UBERLÂNDIA

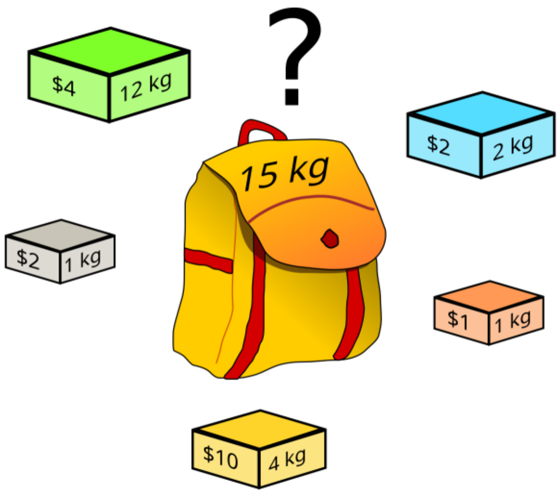
3 SETEMBRO DE 2025

SUMÁRIO

1. DEFINIÇÃO DO PROBLEMA DA MOCHILA 0/1 3
2. PROVA DA SUBESTRUTURA ÓTIMA 4
3. DEFINICÃO DA RECORRÊNCIA 6
4. ESPECIFICAÇÃO DOS ALGORITMOS 7
5. ESPECIFICANDO ALGORITMO PARA SOLUÇÃO ÓTIMA 8
6. EXEMPLO 9
7. ANÁLISE DE COMPLEXIDADE ABORDAGEM RECURSIVA 10
8. ANÁLISE DE COMPLEXIDADE ABORDAGEM ITERATIVA 13
9. ANÁLISE DE COMPLEXIDADE DO RETORNO DA SOLUÇÃO ÓTIMA 15
10. ESTATÍSTICAS 16
11. REFERÊNCIAS 18

1. DEFINIÇÃO DO PROBLEMA DA MOCHILA 0/1

Um ladrão que assalta uma loja encontra n itens. O i-ésimo item vale vi reais e pesa wi quilos, onde vi e wi são inteiros. Ele deseja levar consigo a carga mais valiosa possível, mas só pode carregar no máximo W quilos em sua mochila, sendo W um número inteiro. Que itens ele deve levar? (Esse problema da mochila chama-se 0-1 porque, para cada item, o ladrão tem de decidir se o carrega consigo ou se o deixa para trás; ele não pode levar uma quantidade fracionária de um item nem um item mais de uma vez.)



2. PROVA DA SUBESTRUTURA ÓTIMA

Seja $I(n, W)$ o problema da mochila com n itens e capacidade W.

Seja S uma solução ótima de $I(n, W)$, onde S ⊆ {1, …, n}.

Ao analisar o último elemento do conjunto {1, …, n}, conseguimos decompor o problema original em subproblemas menores. Isso porque qualquer solução ótima deve se enquadrar em um dos dois casos possíveis:

1. Item n pertence à solução ótima, ou

2. Item n não pertence à solução ótima.

Assim, conseguimos dividir todas as soluções possíveis nesses dois cenários, o que nos leva naturalmente à recorrência.

Caso 1: $n \notin S$

Se n não está em S, então S deve ser uma solução ótima para a entrada I(n−1, W).

Por contradição, suponha que exista S*, solução ótima para I(n−1, W), com v(S*) > v(S).

Como: $\sum i \in S^* \ pi \leq W$ e $S^* \subseteq \{1, ..., n-1\} \subseteq \{1, \ldots, n\}$, então S* também seria uma solução viável para I (n, W). Isso contradiz a hipótese de que S é solução ótima para I(n, W).

Caso 2: $n \in$ S

Se n pertence a S, defina $S' = S - \{n\}$, $S' \subseteq \{1, \ldots, n-1\}$. O peso total de S' é: $\sum i \in S' \ pi = \sum i \in S \backslash \{n\} \ pi = \sum i \in S \ pi - pn \leq W - pn$.

Logo, S' é solução viável de I(n−1, W−pn).

Por contradição, suponha que exista S* solução ótima de I(n−1, W−pn), com $v\ (S^*) > v(S')$.

Se adicionarmos o item n a S*, obtemos: $v\ (S^* \cup \{n\}) = v(S^*) + vn > v(S') + vn = v(S)$. Isso contradiz a hipótese de que S é ótima para I(n, W).

Conclusão: Portanto, em ambos os casos, mostramos que a solução ótima S para I(n, W) é formada por uma solução ótima de um subproblema menor. Isso prova que o problema da mochila 0/1 possui subestrutura ótima.

3. DEFINICÃO DA RECORRÊNCIA

Definindo M(i, W) como o **valor máximo** que podemos obter escolhendo itens apenas do conjunto {1, …, i} quando a capacidade disponível é W.

Casos-base

- Se i = 0 (nenhum item disponível) ou w=0 (capacidade zero), não há valor a obter: M(i, w) = 0.

Ideia central (subestrutura ótima)

Para cada item i e capacidade w, a solução ótima depende apenas de subproblemas menores:

- **Não incluir i:** o valor ótimo é M(i−1, w).
- **Incluir i (se couber):** o valor ótimo é vi + M(i−1, w−pi).

Assim, a recorrência surge naturalmente como o máximo entre essas duas opções.

$$M(i,w) = \begin{cases} 0, & \text{se } i = 0 \text{ ou } w = 0 \\ M(i-1,w), & \text{se } p_i > w \\ \max\big(M(i-1,w),\ v_i + M(i-1,w-p_i)\big), & \text{se } p_i \le w \end{cases}$$

4. ESPECIFICAÇÃO DOS ALGORITMOS

- Pseudocódigo da versão recursiva:

```
Função mochila(n, W):
    Se n == 0 ou W == 0:
        Retornar 0

    excluir = mochila(n-1, W)

    incluir = 0
    Se W >= peso[n]:
        incluir = mochila(n-1, W - peso[n]) + valor[n]

    Retornar max(incluir, excluir)
```

- Pseudocódigo do método iterativo:

```
Função Mochila0_1(n, W, peso[1..n], valor[1..n])
    dp[0..n][0..W]

    Para i de 0 até n
        Para w de 0 até W
            Se i == 0 ou w == 0
                dp[i][w] = 0
            Senao se peso[i] <= w
                dp[i][w] = max(dp[i-1][w], valor[i] + dp[i-1][w - peso[i]])
            Senao
                dp[i][w] = dp[i-1][w]

    retorne dp[n][W]
```

5. ESPECIFICANDO ALGORITMO PARA SOLUÇÃO ÓTIMA

A reconstrução percorre a matriz de trás para frente — do último item até o primeiro — porque a matriz foi construída progressivamente, e agora queremos **desfazer** essa construção para entender **quais decisões foram tomadas**.

A cada passo:

- Comparamos dp[i][pAtual] com dp[i-1][pAtual].
- Se forem diferentes, o item i foi escolhido.
- Atualizamos pAtual subtraindo o peso de i, pois agora temos menos capacidade restante.
- Continuamos até o primeiro item.

A reconstrução da solução se baseia em uma observação fundamental:

Se o valor dp[i][p] é diferente de dp[i-1][p], então o item i foi incluído na solução ótima.

Por quê?

Porque se o item i não tivesse sido escolhido, o valor ótimo até esse ponto seria o mesmo que o valor ótimo considerando apenas os i-1 primeiros itens. A única forma de dp[i][p] ser maior é se o item i tivesse contribuído com seu valor, ou seja, tivesse sido incluído.

- Pseudocódigo da construção da solução ótima

```
inicializar itens_escolhidos como lista vazia
i ← n
j ← C

enquanto i > 0 e j > 0 faça
    se matriz[i][j] ≠ matriz[i-1][j] então
        adicionar i à lista itens_escolhidos
        j ← j - pesos[i]
    i ← i - 1

retornar itens_escolhidos
```

6. EXEMPLO

Construção de da matriz para uma instância do problema, composta por 5 itens e a capacidade total da mochila igual a 10.

Dados do Problema:

Capacidade da Mochila: C = 10

Itens Disponíveis:

Item 1: (valor=3, peso=2) Item 2: (valor=4, peso=3) Item 3: (valor=8, peso=4) Item 4: (valor=8, peso=5)

Item 5: (valor=10, peso=9)

Matriz de Programação Dinâmica

dp[i][w] = valor máximo usando itens 0...i-1 com capacidade w

i\w	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7	7	7
3	0	0	3	4	8	8	11	12	12	15	15
4	0	0	3	4	8	8	11	12	12	16	16
5	0	0	3	4	8	8	11	12	12	16	16

Resultados do algoritmo para o exemplo **acima:**

```
==========================================================
 Teste com 5 itens | Capacidade: 10
===========================================================
                  mochila -> Resultado:  16 | Tempo: 0.000000s
               mochila_PD -> Resultado:  16 | Tempo: 0.000000s
   mochila_PD_com_solucao -> Resultado:  16 | Itens: [3, 4] | Tempo: 0.000000s
```

O custo da solução ótima = 16 e a solução ótima é composta pelos itens 3 e 4

7. ANÁLISE DE COMPLEXIDADE ABORDAGEM RECURSIVA

Análise de Complexidade de Tempo

Formulação da Equação de Recorrência

O tempo de execução, T(n, W), pode ser definido pela seguinte relação de recorrência, que reflete as ramificações de decisão do algoritmo:

$$\mathbf{T(n, W) = \Theta(1)\ ,\ se\ n = 0\ ou\ W = 0}$$

$$\mathbf{T(n, W) = T(n - 1, W) + T(n - 1, W - wn) + \Theta(1),\ se\ n > 0}$$

Onde:

- **T(n, W)** é o tempo necessário para resolver o problema com n itens e capacidade W.
- **T(n − 1, W)** representa o custo da chamada recursiva que **exclui** o item n.
- **T(n − 1, W − wn)** representa o custo da chamada recursiva que **inclui** o item n, reduzindo a capacidade da mochila pelo peso wn.
- **Θ(1)** representa o custo das operações de tempo constante em cada chamada, como comparações, atribuições e a função para determinar o máximo valor entre duas opções.

Simplificação da Recorrência para o Pior Caso

Para determinar o limite assintótico do tempo de execução, analisamos o **pior caso**, que ocorre quando o algoritmo é forçado a explorar todas as duas ramificações de sua árvore de recursão em cada passo. Isso acontece quando a capacidade da mochila é grande o suficiente para que o item atual possa ser incluído em todas as chamadas recursivas, levando a uma ramificação dupla consistente e formando uma árvore de recursão completa.

Nesse cenário de pior caso, a variação do peso pode ser ignorada, pois o número de subproblemas gerados é determinado pela duplicação de chamadas, e não pelo valor específico da capacidade restante. Portanto, podemos simplificar a equação de recorrência para uma única variável, **M(n)**, que representa o tempo de execução em função do número de itens, n. A relação se torna:

$$\mathbf{M(n) = \Theta(1)\ ,\ se\ n = 0}$$

$$\mathbf{M(n) = 2M(n - 1) + \Theta(1)\ ,\ se\ n > 0}$$

Aqui, o termo **2M(n − 1)** reflete o fato de que cada problema com n itens se desdobra em dois subproblemas com n − 1 itens. O termo **Θ(1)** representa o trabalho fixo realizado em cada nível da recursão, como a comparação e a soma.

Resolução da Recorrência

Utilizando o método da expansão, teremos:

1º Passo:

M(n) = 2M(n - 1) + 1

2º Passo:

M(n) = 4M(n - 2) + 2 + 1

3º Passo:

M(n) = 8M(n - 3) + 4 + 2 + 1

4º Passo:

M(n) = 16M(n - 4) + 8 + 4 + 2 + 1

Generalização:

Após i iterações, teremos:

$$M(n) = 2^i * M(n - i) + \sum_{j = 0}^{i - 1} 2^j$$

Mas, no intuito de parar a recorrência, n - i deve ser 0. Com isso, n = i e M(n - n = 0) = 1. Além disso, esse somatório possui fórmula fechada equivalente a 2^i - 1.
Assim:

M(n) = 2^n * 1 + 2^n - 1
M(n) = (2 * 2^n) - 1

Como para valores de n muito grandes, 2^n domina o comportamento dessa função. Sendo assim:

$$\mathbf{M(n) = \Theta(2^n)}$$

Dessa maneira, é provado que o tempo de execução da solução recursiva para o problema da Mochila 0/1 cresce exponencialmente em relação ao número de itens.

Análise de Complexidade de Espaço

A complexidade de espaço de um algoritmo recursivo é determinada pela profundidade máxima de sua **pilha de chamadas**. Cada chamada de função recursiva adiciona um novo registro à pilha, que armazena as variáveis e o endereço de retorno daquela chamada.

No pior caso, essa recursão da Mochila 0/1 pode se aprofundar em um caminho que corresponde ao processamento de cada um dos n itens. A cada passo, uma nova chamada para um problema com $n - 1$ itens é feita, sem que a anterior seja resolvida, até que o caso base seja atingido.

Portanto, a profundidade máxima da pilha de chamadas é proporcional ao número de itens, n.

A complexidade de espaço para essa abordagem recursiva é, assim, $\Theta(n)$.

8. ANÁLISE DE COMPLEXIDADE ABORDAGEM ITERATIVA

Relembrando o problema

O problema da Mochila 0/1 busca encontrar o subconjunto de itens que maximiza o valor total, respeitando a capacidade de peso da mochila. Para isso, definimos:

- **N**: o número de itens disponíveis.
- **W**: a capacidade máxima de peso da mochila.
- **vi** : o valor do item i
- **wi** : o peso do item i.

Subestrutura Ótima e Subproblemas

O problema possui a propriedade de **subestrutura ótima**, pois a solução de um problema maior depende das soluções ótimas de subproblemas menores. A solução para uma entrada com N itens e capacidade W é construída a partir de subproblemas que consideram os primeiros i itens com uma capacidade de peso w.

Sendo assim, cada subproblema pode ser definido por um par (i,w), onde $0 \leq i \leq N$ e $0 \leq w \leq W$.

O número total de subproblemas é a quantidade de combinações possíveis desses pares, o que resulta em um número de subproblemas na ordem de **Θ (N·W)**.

Análise de Complexidade de Tempo

A solução de programação dinâmica para o problema utiliza dois laços aninhados para preencher a tabela de subproblemas:

- O laço externo itera sobre os itens (de 1 a N), com um custo de **Θ** (N).
- O laço interno itera sobre as capacidades de peso (de 0 a W), com um custo de **Θ** (W).

Para cada par (i,w), o algoritmo realiza um número constante de operações. Portanto, o tempo total de execução é proporcional ao número de iterações totais, que é o produto do número de iterações de cada laço.

A complexidade de tempo é, portanto, **Θ (N·W)**.

Análise de Complexidade de Espaço

O algoritmo utiliza uma matriz dp para armazenar os resultados dos subproblemas.

- O tamanho dessa matriz é dado pelo número de itens e pela capacidade da mochila...
- As dimensões são de **N+1** por **W+1**.

O espaço de armazenamento necessário é, portanto, proporcional ao tamanho da matriz. A complexidade de espaço é **Θ (N·W)**.

9. ANÁLISE DE COMPLEXIDADE DO RETORNO DA SOLUÇÃO ÓTIMA

Complexidade Temporal

O algoritmo percorre os itens da mochila em um laço **while,** controlado pela variável i, que inicia em n (número de itens) e é decrementada a cada iteração. Dessa forma, o laço é executado no máximo **n** vezes.

Em cada iteração, todas as operações realizadas possuem custo constante, isto é, **Θ(1).**

Portanto, a complexidade temporal do algoritmo é:

$$T(n) = \Theta(n).$$

Complexidade Espacial

A complexidade de espaço é determinada pela lista **itens_escolhidos.** No pior caso, a lista **itens_escolhidos** pode armazenar até n elementos, o que acontece quando todos os itens são incluídos na solução ótima. O espaço ocupado por essa lista, portanto, é proporcional ao número de itens, **n**. As variáveis auxiliares (i e j) ocupam apenas espaço constante, **Θ(1)**, que é desprezível em relação à lista.

Assim, a complexidade espacial do algoritmo é:

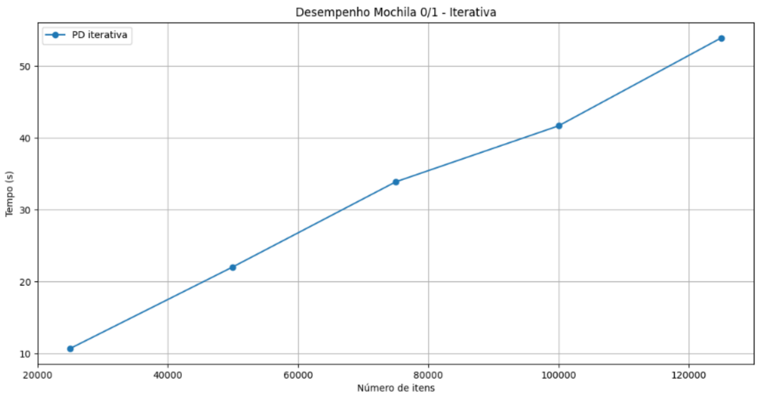
$$E(n) = \Theta(n).$$

10. ESTATÍSTICAS

OBS.: Para a análise do desempenho dos algoritmos consideramos uma capacidade da mochila fixa igual a 1000, com os preços dos itens podendo variar de 1 até 200 e o peso podendo variar de 1 até 100.

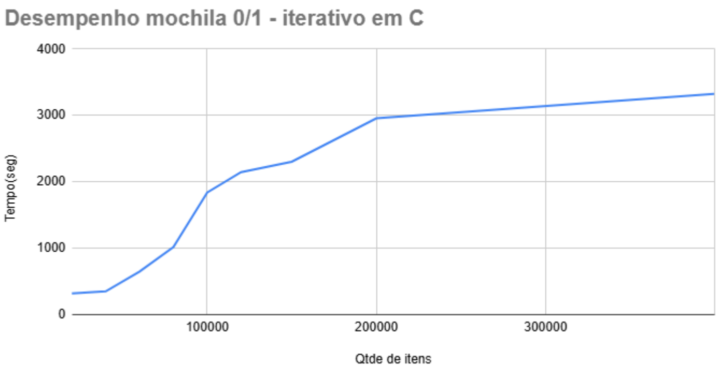
- Análise Visual do Desempenho do Algoritmo Iterativo em Python

Algoritmo Iterativo	
Qtde de itens	**Tempo(seg)**
20000	8.5
40000	16.9
60000	27.2
80000	37.4
100000	41.6
120000	53.8



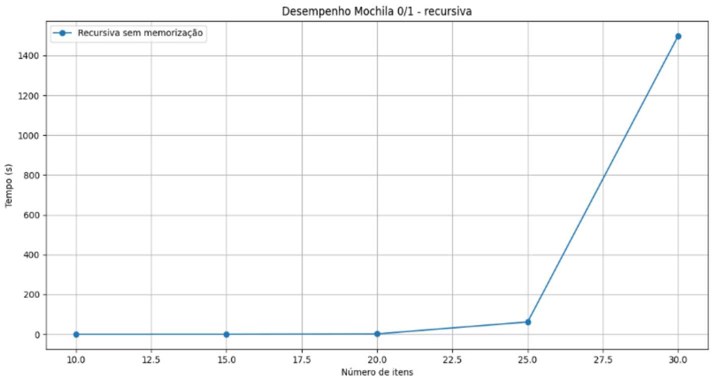
- Análise Visual do Desempenho do Algoritmo Iterativo em C

Algoritmo Iterativo em C	
Qtde de itens	**Tempo(seg)**
20000	0.319
40000	0.350
60000	0.648
80000	1.018
100000	1.839
120000	2.145
150000	2.303
200000	2.956
400000	3.325



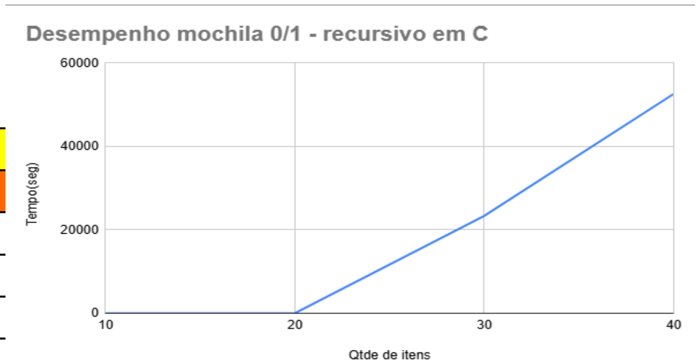
- Análise Visual do Desempenho do Algoritmo recursivo em Python

Algoritmo recursivo	
Qtde de itens	Tempo(seg)
10	3.005
15	7.256
20	16.101
25	65.007
30	1,505.793

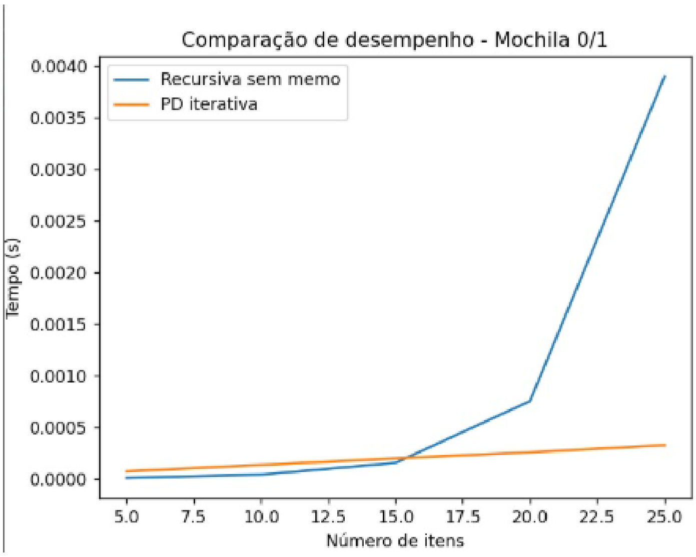


- Análise Visual do Desempenho do Algoritmo recursivo em C

Algoritmo recursivo em C	
Qtde de itens	Tempo(seg)
10	0.003
20	0.004
30	23.343
40	52.588



- Análise Visual da Comparação do método iterativo e do método recursivo (capacidade = 100):



11. REFERÊNCIAS

1. CORMEN, T. H. Algoritmos, Teoria e Prática. 3 ed.
2. Notas de aula do Prof. Mário César San Felice (http://www.aloc.ufscar.br/felice) do Departamento de Computação (DC) da Universidade Federal de São Carlos (UFSCar).
3. Notas de aula do Prof. Tim Roughgarden (http://timroughgarden.org/) do Departamento de Ciência da Computação na Columbia University.
4. Notas de aula da Prof. Carla Negri Lintzmayer (http://professor.ufabc.edu.br/~carla.negri/) do Centro de Matemática, Computação e Cognição da Universidade Federal do ABC