Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação

Problema da Mochila 0/1

Introdução a Ciência da Computação II

Discentes: Cleyton José Rodrigues Macedo
Kauã Benjamin Trombim Silva

Docente: Marcelo Garcia Manzato

São Carlos
Outubro de 2025

1 Introdução

O Problema da Mochila 0/1 é uma versão clássica de otimização combinatória onde, dado um conjunto de n itens com pesos e valores distintos, deve-se escolher um subconjunto de itens cuja soma dos valores seja máxima, sem que a soma dos pesos exceda a capacidade W da mochila. A restrição "0/1"implica que cada item deve ser inteiramente incluído ou totalmente descartado.

Este relatório apresenta uma análise teórica de três abordagens para a solução deste problema:

- **Força Bruta:** Uma busca exaustiva que garante a otimalidade.
- **Algoritmo Guloso:** Uma versão rápida que busca uma solução aproximada.
- **Programação Dinâmica:** Uma abordagem ótima e mais eficiente que a força bruta.

2 Estruturas de Dados

A principal estrutura de dados utilizada para representar cada objeto é a `Item`, que armazena seu identificador, peso, valor e a razão valor/peso, crucial para a abordagem gulosa.

Para a mochila, é usado a estrutura `Mochila`, que armazena o peso máximo suportado, o valor total na mochila e o peso total na mochila.

```c
typedef struct {
    int id;
    int peso;
    int valor;
    double razao; // valor / peso
} Item;

typedef struct {
    int max_w;
    long long valor_total;
    int peso_total;
} Mochila;
```

Listing 1: Estrutura para representar um item e a mochila.

3 Abordagem por Força Bruta

3.1 Descrição Teórica

A abordagem de força bruta resolve o problema da forma mais direta possível: gerando e avaliando todos os subconjuntos de itens possíveis. Para um conjunto de n itens, existem 2^n combinações. O algoritmo verifica cada combinação, calcula seu peso e valor totais e mantém o registro daquela que oferece o maior valor sem exceder capacidade W.

3.2 Análise da Implementação

A implementação é dividida em duas funções. A primeira, `forca_bruta`, prepara as variáveis e inicia a busca. A segunda, `forca_bruta_recursiva`, contém o núcleo da lógica de exploração da árvore de combinações.

```
void forca_bruta(Item *itens, int n, int w){
    if (n >= 64) {
        printf("Erro: O n mero de itens deve ser menor
           que 64...\n");
        return;
    }
    printf("\n--- Executando Solu  o For a Bruta ---\
       n");

    long long maxVal = 0;
    long long bestComb = 0LL;

    // Inicia a chamada recursiva
    forca_bruta_recursiva(itens, n, w, 0, 0, 0, &maxVal,
        0LL, &bestComb);

    // ... (c digo para imprimir o resultado) ...
}
```

Listing 2: Função principal da Força Bruta.

```
void forca_bruta_recursiva(Item* itens, int n, int w,
                           long long valAtual, int
                              pesoAtual, int i,
                           long long* maxVal, long long
                              comb, long long* bestComb)
{
```

```
5       // --- Caso base: todos os itens foram processados
            ---
6       if (i == n) {
7           if (pesoAtual <= w && valAtual > *maxVal) {
8               *maxVal = valAtual;
9               *bestComb = comb;
10          }
11          return;
12      }
13
14      // --- Ignora o item atual ---
15      forca_bruta_recursiva(itens, n, w, valAtual,
            pesoAtual, i + 1, maxVal, comb, bestComb);
16
17      // --- Inclui o item atual ---
18      double novoPeso = pesoAtual + itens[i].peso;
19      double novoValor = valAtual + itens[i].valor;
20
21      // S  explora se ainda h  chance de ser v lido (
            rvore  com "poda", j  que n o explora toda a
            rvore )
22      if (novoPeso <= w) {
23          forca_bruta_recursiva(itens, n, w, novoValor,
                novoPeso, i + 1, maxVal, comb | (1LL << i),
                bestComb);
24      }
25  }
```

Listing 3: Núcleo recursivo da Força Bruta (com poda).

A função recursiva busca a melhor solução da seguinte forma:

- **Lógica:** Para cada item, a função explora duas ramificações: uma onde o item é incluído na mochila e outra onde ele não é. Isso cria uma árvore de recursão de profundidade n, cujas folhas representam todas as 2^n combinações.
- **Rastreamento de Subconjuntos:** A implementação utiliza um long long comb como uma máscara de bits (bitmask). Cada bit na variável representa um item. Se o i-ésimo bit está ativo (1), significa que o item i está na combinação atual. Esta é uma forma eficiente de representar e passar os subconjuntos através das chamadas recursivas.
- **Limitação:** A utilização de long long para a máscara de bits impõe

uma limitação de $n \leq 64$ itens, o que é explicitamente verificado no código, porém, na prática, isso não é um problema já que 64 itens é um número inviavelmente grande para a força bruta.

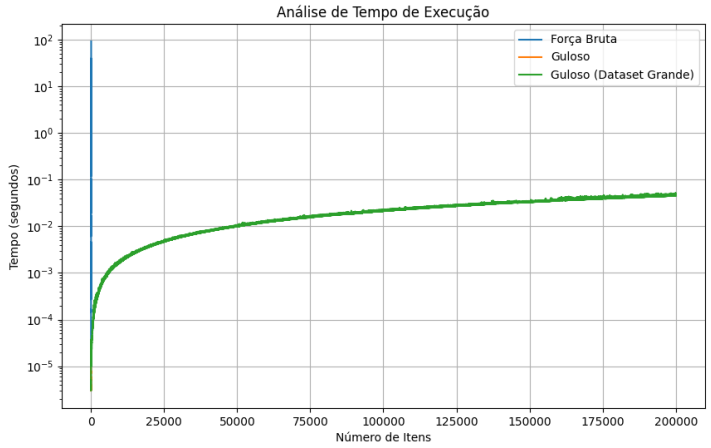

Figura 1: Força bruta com apenas 60 itens vs Guloso com 200 mil itens.

3.3 Análise de Complexidade

O algoritmo de força bruta pode ser feito de duas formas, com ou sem poda, sendo a poda uma otimização da árvore de recursão que evita que o algorítmo explore toda a árvore em alguns casos.

3.3.1 Algorítmo sem Poda (Brute-Force Completo)

Nesta versão, o algoritmo explora TODOS os caminhos possíveis da árvore de recursão, sem exceção. Para cada um dos n itens, ele faz duas chamadas recursivas, independentemente do peso acumulado na mochila.

$$T(i) = \begin{cases} c & \text{se } i = n \\ 2T(i+1) + c & \text{se } i \neq n \end{cases}$$

Isso significa que ele constrói a árvore binária completa de profundidade n Figura 2.

- **Análise da Complexidade de Tempo:** $O(2^n)$
  A complexidade de tempo é proporcional ao número total de nós visitados na árvore de recursão. A soma dos nós em uma árvore binária

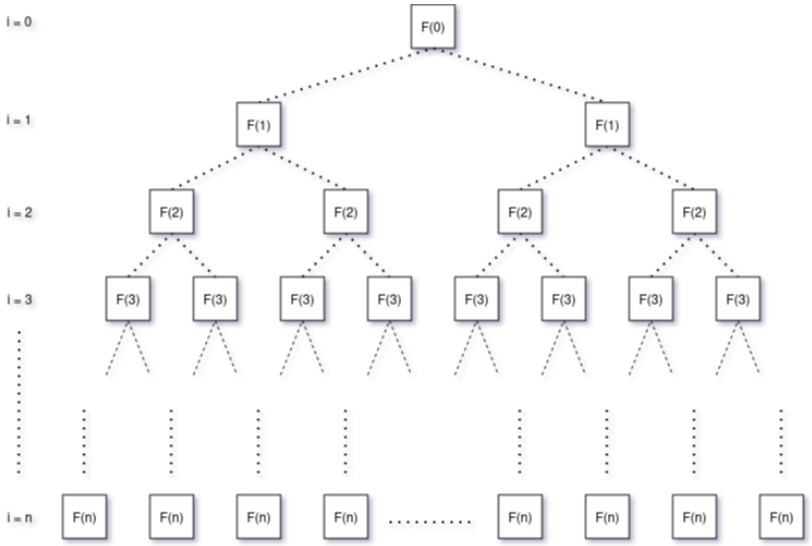

Figura 2: Árvore de recursão.

completa de profundidade n é:

$$1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1$$

Dessa forma, a versão sem poda sempre executará um número de operações proporcional a 2^n, não importa quais sejam os pesos dos itens ou a capacidade da mochila, sendo assim $O(2^n)$.

O crescimento exponencial do algoritmo pode ser observado no gráfico da Figura 3.

3.3.2 Algorítmo com Poda (Brute-Force Otimizado)

A instrução if (novoPeso ¡= w) atua como um mecanismo de poda (*prunnig*).

```
1     // --- Inclui o item atual ---
2     double novoPeso = pesoAtual + itens[i].peso;
3     double novoValor = valAtual + itens[i].valor;
4
5     // S  explora se ainda h  chance de ser v lido (
          rvore  com "poda", j  que n  o explora toda a
          rvore )
6     if (novoPeso <= w) {
```

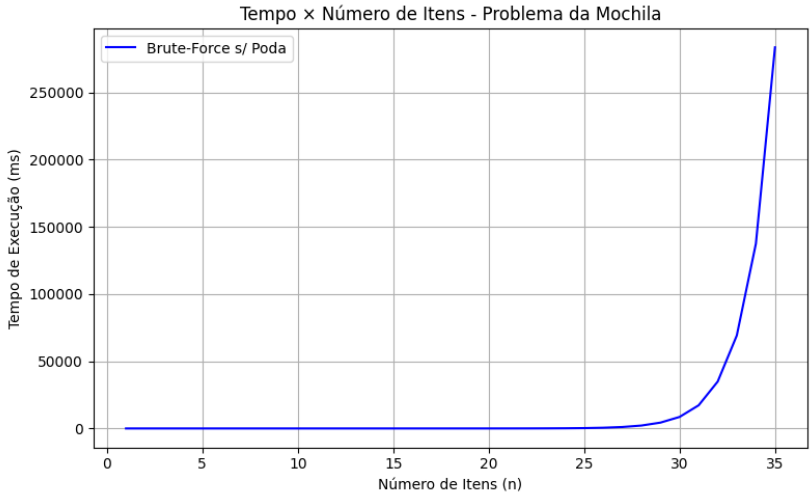

Figura 3: Tempo de execução do algoritmo Força Bruta sem poda em função de n.

```
7           forca_bruta_recursiva(itens, n, w, novoValor,
                novoPeso, i + 1, maxVal, comb | (1LL << i),
                bestComb);
8       }
9   }
```

Listing 4: Implementação do mecanismo de poda.

Se, ao tentar adicionar um item, o peso da mochila ultrapassa a capacidade W, o algoritmo não realiza a chamada recursiva correspondente. Assim, ele "corta"(poda) um galho inteiro da árvore de recursão porque sabe que todos os nós descendentes daquele ponto também representarão soluções inválidas (com excesso de peso).

- **Pior caso:** $O(2^n)$
  A complexidade de pior caso não muda. O pior cenário para o algoritmo com poda ocorre quando a poda raramente ou nunca acontece, ou seja, quando a capacidade W da mochila é muito grande, maior que a soma dos pesos de todos os itens, ou quando todos os itens têm pesos muito pequenos em relação a W. Nesses casos, a condição novoPeso ¡= w será quase sempre verdadeira, e o algoritmo será forçado a explorar a árvore de recursão quase inteira

  Dessa forma, a versão sem poda sempre executará um número de operações proporcional a 2^n, não importa quais sejam os pesos dos

itens ou a capacidade da mochila, sendo assim $O(2^n)$.

- **Melhor Caso e Caso Médio**
  Na prática, a poda oferece um ganho de desempenho significativo. Se os pesos dos itens são significativos em relação à capacidade W, muitos galhos da árvore serão cortados cedo, e o número de nós visitados será muito menor que 2^n. O crescimento exponencial do algoritmo pode ser observado no gráfico da Figura 4, e a comparação na Figura 5, o algoritmo com poda sempre se levará menos tempo ou será igual ao sem poda.

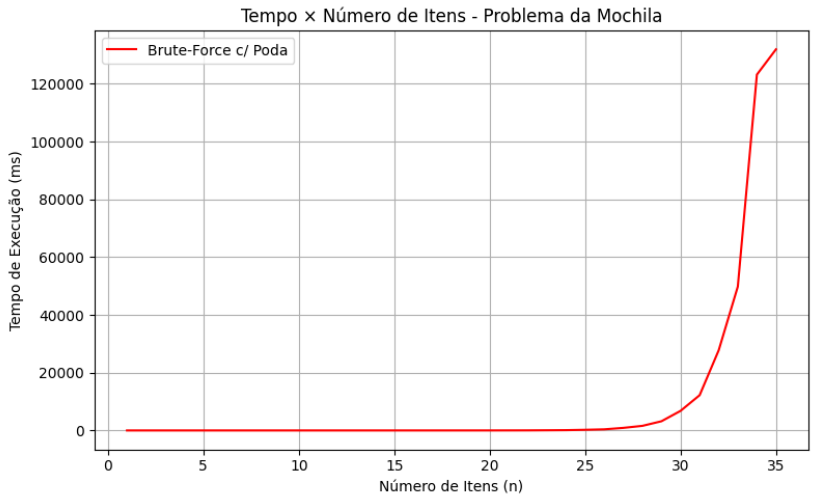

Figura 4: Tempo de execução do algoritmo Força Bruta com poda em função de n.

3.4 Garantia de Otimização

Esta abordagem é garantidamente ótima, pois ao avaliar todas as possibilidades, é impossível não encontrar a melhor solução.

4 Abordagem Gulosa

4.1 Descrição Teórica

O algoritmo guloso não busca a solução ótima, mas sim uma solução "boa o suficiente"de forma muito rápida. Ele constrói a solução passo a passo,

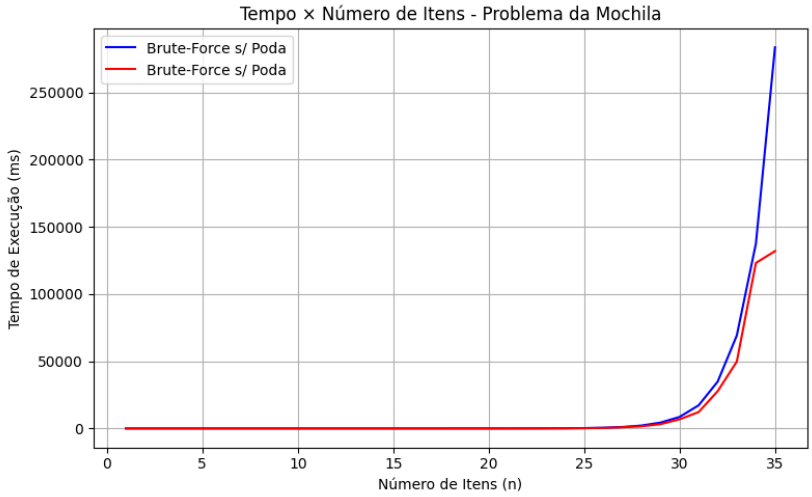

Figura 5: Comparação dos tempos de execução ($n = 35$ mostra como a poda pode reduzir drasticamente o tempo de procura)

fazendo a escolha localmente ótima em cada etapa. A heurística padrão para o Problema da Mochila é baseada na razão valor/peso.

O algoritmo segue os seguintes passos:

- 1. Calcular a razão valor/peso para cada item.
- 2. Oredenar os itens em ordem decrescente com base nessa razão.
- 3. Percorrer a lista ordenada, adicionando cada item à mochila se e ele couber na capacidade restante.

4.2 Análise da Implementação

A implementação em `guloso_alg` segue essa estrutura geral de ordenar e depois preencher a mochila.

- **Ordenação:** A função utiliza um $heap_sort$ para ordenar os itens. Este passo é o mais custoso do algoritmo.
- **Preenchimento** Após a ordenação, um laço for percorre a lista de itens e os adiciona à mochila se houver capacidade.

4.3 Análise de Complexidade

- **Complexidade de Tempo:** $O(n \log n)$
  A complexidade é dominada pela etapa de ordenação dos itens. O *quick_sort*, assim como outros algoritmos eficientes (Heap Sort, Merge Sort), tem essa complexidade de tempo no caso médio e pior. O gráfico de tempo do algoritmo pode ser visto em Figura 6 e Figura 7

- **Complexidade de Espaço:** $O(n)$ É necessário espaço para armazenar os ponteiros dos itens para a ordenação, resultando em complexidade de espaço linear.

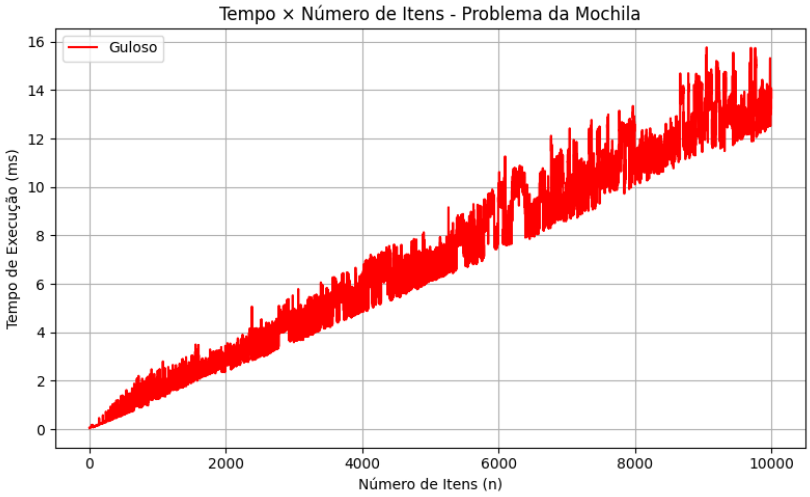

Figura 6: Gráfico de tempo para 10 mil itens

4.4 Garantia de Otimização

Esta abordagem não garante a solução ótima para o Problema da Mochila 0/1. Uma escolha localmente ótima (pegar o item com a melhor razão) pode impedir a escolha de uma combinação de outros itens que resultaria em um valor final maior. Exemplo clássico:

Itens com (peso, valor): A = (30, 90), B = (20, 80) e C = (35, 150), e mochila com capacidade 50.

A solução gulosa escolheria o item c por ter a maior razâo, e nâo teria espaço para mais nenhum item, terminando com valor 150, porém, a solução otima com A e B possui valor 170.

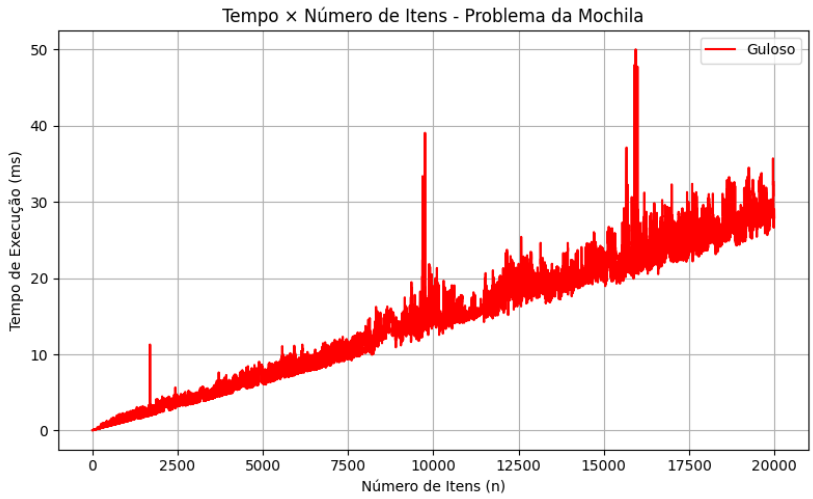

Figura 7: Gráfico de tempo para 20 mil itens (alguns outliers devido ao dispositivo utilizado)

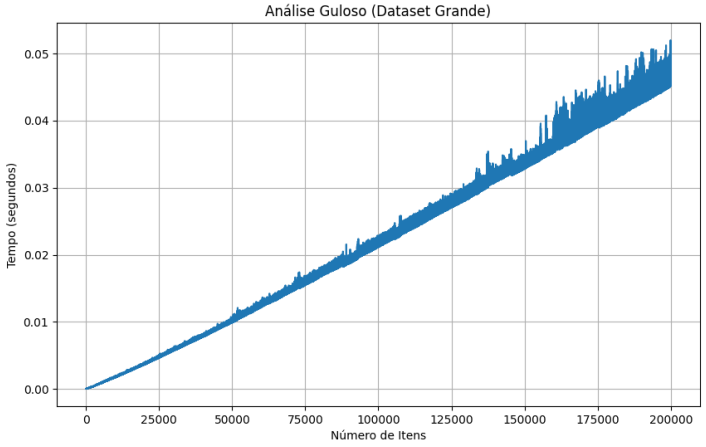

Figura 8: Gráfico de tempo para 200 mil itens (executado em maquina diferente)

5 Abordagem Dinâmica

5.1 Descrição Teórica

A Programação Dinâmica é uma técnica de programação que consiste em resolver versões menores do problema e escalar até resolver o problema de fato. Nessa abordagem, calculamos a resolução para mochilas com capaci-

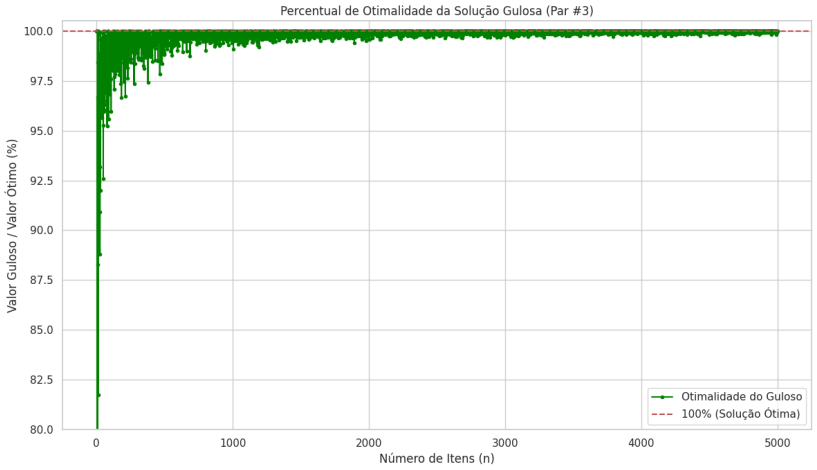

Figura 9: Gráfico de relação entre o resultado do aloritmo dinâmico e o guloso

dade máxima 1, até W. A lógica usada para cada resolução é construir uma tabela $(n+1) \times (W+1)$, onde n é o numero de itens e W o peso máximo da mochila, e para cada iteração, é feita a decisão de inserir ou não o item, escolhendo a opção que mais gera valor na célula (Somamos 1 as dimensões para facilitar a escrita do laço, além de criar a base para as comparações que começam em i = 1).
Tome como exemplo o seguinte exemplo e seu algoritmo

Considere os itens abaixo e uma mochila de tamanho 3

- **Item 0**: Peso 2, valor 5;
- **Item 1**: Peso 1, valor 3;
- **Item 2**: Peso 3, valor 7.

Note que na última célula da tabela sempre é exibida a solução otima para o problema. O algoritmo consiste em tentar inserir um item na mochila, e comparar com as decisões anteriores. Existem duas opções: inserir o novo elemento combinado com a melhor escolha até então, ou repetir a solução anterior, ambas tendo como limite o peso atual da mochila.

5.2 Análise da Implementação

A implementação dessa abordagem segue uma linha simples: criar a tabela auxiliar, preenchê-la conforme as as escolhas descritas anteriormente e preencher possíveis espaços vagos. Para exibir os itens mostrados, o sistema apenas retorna contando as inserções.

Itens x Capacidade			
	Capacidade 1	Capacidade 2	Capacidade 3
Item 0	0	5	5
Item 1	3	5	8
Item 2	3	5	8

Figura 10: Forma visual da Programação Dinâmica.

```c
//Funcao Auxiliar
int max(int a, int b) {
    return (a > b) ? a : b;
}

void prog_dinamica(Item *itens, int n, int w) {
    if (itens == NULL || n <= 0 || w <= 0) {
        printf("Entrada invalida.\n");
        return;
    }

    //Aloca espaco para a tabela
    int **tabela = (int **) calloc(n + 1, sizeof(int *))
        ;
    if (tabela == NULL) {
        printf("Erro de alocacao\n");
```

```
16          return;
17      }
18      for (int i = 0; i <= n; i++) {
19          tabela[i] = (int *) calloc(w + 1, sizeof(int));
20          if (tabela[i] == NULL) {
21              printf("Erro de alocacao\n");
22              for(int k=0; k < i; k++) free(tabela[k]);
23              free(tabela);
24              return;
25          }
26      }
27
28      //Preenche as celulas da tabela
29      for (int i = 1; i <= n; i++) {
30          for (int j = 1; j <= w; j++) {
31              int pesoAtual = itens[i-1].peso;
32              int valorAtual = itens[i-1].valor;
33              int valorAnterior = tabela[i-1][j];
34
35              //Se o item nao cabe
36              if (pesoAtual > j) {
37                  tabela[i][j] = valorAnterior;
38              } else {
39                  //O novo valor    a soma do item atual
                       com a melhor possibilidade de resto
40                  int novoValor = valorAtual + tabela[i
                       -1][j - pesoAtual];
41                  tabela[i][j] = max(valorAnterior,
                       novoValor); //Escolhe o maior
42              }
43          }
44      }
45
46      //Exibe os itens guardados e os dados finais
47      int melhor = tabela[n][w];
48      int peso_final = 0;
49      int espacoVago = w;
50      int qtdItens = 0;
51      for (int i = n; i > 0 && espacoVago > 0; i--) {
52          if (tabela[i][espacoVago] != tabela[i-1][
                espacoVago]) {
53              Item itemAtual = itens[i-1];
54              printf("- Item %d (Peso: %d, Valor: %d)\n",
```

```
                itemAtual.id, itemAtual.peso, itemAtual.
                valor);
55              espacoVago -= itemAtual.peso;
56              peso_final += itemAtual.peso;
57              qtdItens++;
58          }
59      }
60
61      printf("Valor final: %d\n", melhor);
62      printf("Peso final: %d\n", peso_final);
63      printf("Itens inseridos: %d\n", qtdItens);
64
65      for (int i = 0; i <= n; i++) {
66          free(tabela[i]);
67      }
68      free(tabela);
69  }
```

Listing 5: Implementação da Programação Dinâmica

Foi criada uma função auxiliar *max* para facilitar a atribuição no laço principal, mas que poderia ter sua lógica desenvolvida sem o uso dessa função externa.

5.3 Análise de Complexidade

- **Complexidade de Tempo:** $O(n \cdot W)$
  A complexidade de tempo desse algoritmo é dominada principalmente pelo laço for aninhado, envolvendo os n itens nas linhas e os W pesos nas colunas. O gráfico de crescimento, comparado com o algoritmo guloso pode ser visto a seguir:

- **Complexidade de Espaço:** $O(n \cdot W)$ Devido o uso da tabela $n \cdot W$ como auxiliar, o gasto em termos de memória é principalmente por causa dessa estrutura.

5.4 Garantia de Otimização

Essa solução sempre garante a melhor solução para o problema, mesmo que isso custe mais tempo de processamento comparado ao Algoritmo Guloso. Vale notar que para n muito grande, essa técnica é inviável pelo gasto alto de memória e processamento.

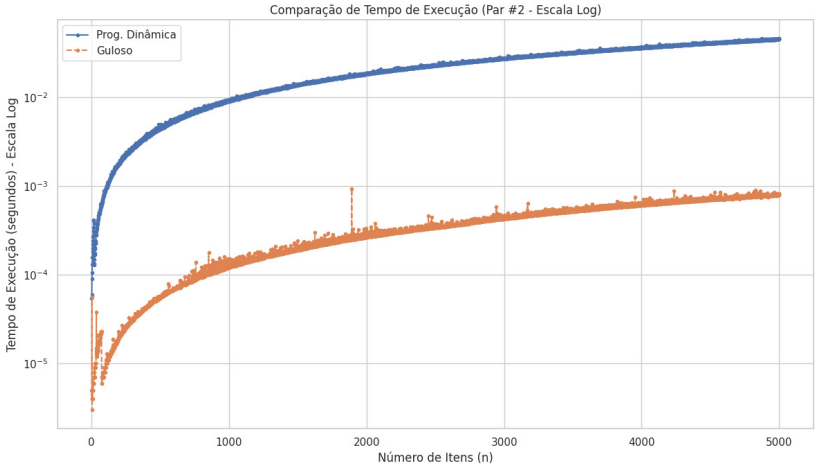

Figura 11: Gráfico comparativo Programação Dinâmica e Algoritmo Guloso.