Relatório: Análise de Eficiência dos Algoritmos para Resolução do Problema da Mochila 0/1

Gustavo de Faria Fernandes (16871221) e Rafael Pavon Diesner (16898096)

9 de outubro de 2025

Sumário

1 Resolução por Força Bruta

1.1 Implementação

A resolução por força bruta consiste em tentar todas as combinações de itens possíveis e escolher a melhor delas.

```
1  MOCHILA *brute_force(ITEM **itens, int pesoMax, int n, int
       indexItem) //no inicio, IndexItem vai ser o item inicial (vai
       valer 0)
2  {
3      clock_t inicio, fim; //marca de tempo
4      double tempoExec; //marca o tempo de execu  o
5      inicio = clock();
6      //cria mochila que armazenara a melhor mochila ate entao
7      MOCHILA *mochilaMelhor = (MOCHILA *)malloc(sizeof(MOCHILA));
8      mochilaMelhor->valor = 0;
9      mochilaMelhor->peso = 0;
10     mochilaMelhor->nItensArmazenados = 0;
11     mochilaMelhor->pesoMax = pesoMax;
12     mochilaMelhor->itensArmazenados = (ITEM**) malloc (sizeof(ITEM
           *) * n);
13     //cria mochila que sera operada a cada passo
14     MOCHILA *mochilaAtual = (MOCHILA *)malloc(sizeof(MOCHILA));
15     mochilaAtual->pesoMax = pesoMax;
16     mochilaAtual->valor = 0;
17     mochilaAtual->peso = 0;
18     mochilaAtual->nItensArmazenados = 0;
19     mochilaAtual->itensArmazenados = (ITEM**) malloc (sizeof(ITEM
           *) * n);
20     bruteforce_recursao(mochilaAtual, mochilaMelhor, indexItem,
           itens, n);
21     //libera a mochila atual, deixando apenas a melhor mochila
22     free(mochilaAtual->itensArmazenados);
23     free(mochilaAtual);
24     fim = clock();
25     tempoExec = ((double)(fim - inicio)/CLOCKS_PER_SEC);
26     printf("Tempo de execucao: %lf",tempoExec);
27     return mochilaMelhor;
28 }
29
30 void bruteforce_recursao(MOCHILA *mochilaAtual, MOCHILA *
       mochilaMelhor, int indexItem, ITEM **itens, int n) //
       mochilaMelhor vai sendo enchida com o tempo
31 {
32     if (indexItem == n) //caso base (acabou os itens a serem
           verificados)
33     {
34         if(mochilaAtual->valor > mochilaMelhor->valor) //se a
               mochila atual for a nova melhor, bota ela como melhor
35         {
36             mochilaMelhor->valor = mochilaAtual->valor;
```

```
37              mochilaMelhor->peso = mochilaAtual->peso;
38              mochilaMelhor->nItensArmazenados = mochilaAtual->
                    nItensArmazenados;
39              for (int i = 0; i < mochilaAtual->nItensArmazenados; i
                    ++)
40              {
41                  mochilaMelhor->itensArmazenados[i] = mochilaAtual
                        ->itensArmazenados[i];
42              }
43          }
44          return;
45      }
46      bruteforce_recursao(mochilaAtual, mochilaMelhor, indexItem +
            1, itens, n); //recursao para o caso em que nao se escolhe
            o item
47      if (mochilaAtual->peso + get_peso(itens[indexItem]) <=
            mochilaAtual->pesoMax) //se for possivel adicionar o item,
            chama a recursao com o item incluido
48      {
49          mochilaAtual->itensArmazenados[mochilaAtual->
                nItensArmazenados] = itens[indexItem];
50          mochilaAtual->peso += get_peso(itens[indexItem]);
51          mochilaAtual->valor += get_valor(itens[indexItem]);
52          mochilaAtual->nItensArmazenados++;
53          bruteforce_recursao(mochilaAtual, mochilaMelhor, indexItem
                + 1, itens, n);
54          mochilaAtual->nItensArmazenados--; //desfazekmos a adicao
                para nao afetar os outros ramos da recursao
55          mochilaAtual->peso -= get_peso(itens[indexItem]);
56          mochilaAtual->valor -= get_valor(itens[indexItem]);
57      }
58  }
```

Listing 1: Implementação em C da Força Bruta

1.2 Equações de Recorrência

Para a montagem da equação de recorrência, analisamos a recursão do código, que ocorre na função `bruteforce_recursao`, onde também se concentra a complexidade do algoritmo. Definimos $T(n)$ como o tempo de execução da função para uma entrada com n itens a serem considerados.

- **Caso Base:** O caso base da recursão ocorre quando não há mais itens para serem avaliados, ou seja, `indexItem == n`. Nesse caso, o custo é considerado constante (c_1), pois a função realiza um número constante de operações. Assim, $T(0) = c_1$.
- **Passo Recursivo:** O passo recursivo ocorre sempre que `indexItem < n`.
    1. Em cada chamada recursiva, temos um custo constante (c_2) para as operações de comparação e atribuição.

2. Cada chamada recursiva reduz o problema em um item, portanto, o tamanho do subproblema é $n - 1$.
3. No pior dos casos (quando o item sempre cabe na mochila), ocorrem duas chamadas recursivas, resultando em duas chamadas de $T(n - 1)$.
4. Portanto, o tempo total $T(n)$ é a soma do custo dessas chamadas com o custo constante: $T(n) = 2T(n - 1) + c_2$.

Juntando-se o caso base e o passo recursivo, obtemos a equação de recorrência:

$$T(n) = \begin{cases} c_1 & \text{se } n = 0 \\ 2T(n-1) + c_2 & \text{para } n > 0 \end{cases}$$

1.3 Análise de Complexidade

Para analisar a complexidade do algoritmo, utilizamos o método da árvore de recursão.

1. **Montagem da Árvore:** A raiz da árvore é o problema original de tamanho n. No nível 1, temos 2 nós, cada um sendo um subproblema de tamanho $n - 1$. A cada novo nível, cada nó origina 2 novos nós.

2. **Cálculo do Custo por Nível:** No nível i, teremos 2^i nós, cada um com um custo constante c. O custo total do nível i é $c \cdot 2^i$.

3. **Altura da Árvore:** A recursão para quando o tamanho do problema atinge o valor zero. Isso ocorre quando $n - m = 0$, onde m é o nível. Portanto, a altura da árvore é n.

4. **Soma dos Custos:** O custo total é a soma dos custos de todos os níveis. O último nível (folhas) possui 2^n nós, com custo total de $c_1 \cdot 2^n$. A soma dos níveis anteriores é dada pela série geométrica:

$$\sum_{i=0}^{n-1} c \cdot 2^i$$

que pode ser resolvida aplicando-se a fórmula $\sum_{k=0}^{m} ar^k = a\frac{r^{m+1}-1}{r-1}$, obtendo-se $c \cdot (2^n - 1)$.

5. **Determinação da Complexidade:** Substituindo o resultado da soma de volta na equação de custo total, obtemos:

$$T(n) = c \cdot (2^n - 1) + c_1 \cdot 2^n = (c + c_1) \cdot 2^n - c$$

Ignorando-se constantes e termos de menor grau, obtemos o termo dominante 2^n. Portanto, a complexidade é $O(2^n)$.

1.4 Análise Empírica

*C*om a análise feita anteriormente, percebemos que a complexidade da resolução por força bruta é O(2^n), o que nos mostra que o tempo de execução do programa cresce exponencialmente de acordo com o crescimento de n. A tabela e o gráfico abaixo nos mostram isso aplicado na realidade e como esse algoritmo se torna inviável com valores

grandes de n. Exemplo disso é que quando n = 100, o tempo de execução corresponde a mais de 30 trilhões de anos, mostrando que o algoritmo so funciona para valores pequenos de n. Os tempos de execução com n valendo 100, 500 e 1000, nesse caso, foram estimados considerando que 1 segundo equivale a aproximadamente 10^9 operações, já que seria impossível de completar a execução, considerando tais aproximações.

Tabela 1: Tempo de execução da força bruta para diferentes tamanhos de entrada.

Numero de Itens (n)	**Tempo de Execucao (s)**
5	5×10^{-6}
10	$1,1 \times 10^{-5}$
20	$1,16 \times 10^{-4}$
100*	10^{21}
500*	10^{141}
1000*	10^{291}

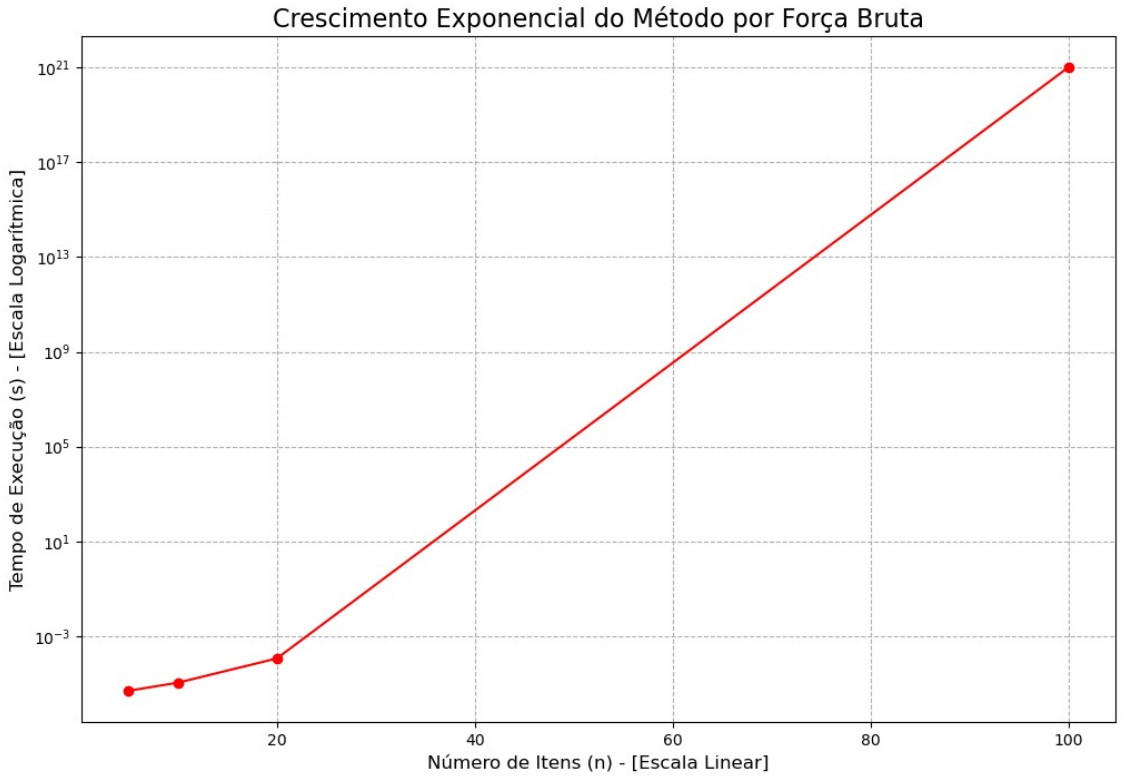

Figura 1: Gráfico com tempos de execução da força bruta

2 Resolução por Algoritmo Guloso

2.1 Implementação

*P*ara o algoritmo Guloso, usamos o método de ordenação quickSort para ordenar os itens pela sua razão valor/peso, de modo que fique mais fácil de pegar os itens com uma maior razão para colocá-los na mochila final. Foi criada também uma struct chamada "NOGULOSO"para relacionar um item à sua razão.

```
typedef struct noguloso_ {
    ITEM* item;
    float razao;
}NOGULOSO;

MOCHILA *guloso(ITEM **itens, int pesoMax, int n)
{
    clock_t inicio, fim; //marca de tempo
    double tempoExec; //marca o tempo de execu  o
    inicio = clock();
    MOCHILA* mochila = (MOCHILA*)malloc (sizeof(MOCHILA));
    mochila->pesoMax = pesoMax;
    mochila->valor = 0;
    mochila->peso = 0;
    mochila->nItensArmazenados = 0;
    mochila->itensArmazenados = (ITEM**)malloc(sizeof(ITEM*) * n);
        //aloca meoria temporariamente para operar nos itens
       armazenados
    if (mochila->itensArmazenados == NULL)
    {
        printf("Erro: Sem espa o para alocacao de memoria.\n");
        return mochila; // Retorna a mochila vazia em caso de
           falha
    }
    NOGULOSO noguloso[n];
    for (int i = 0; i < n; i++) //loop que cria o vetor de structs
    {
        //bota os itens dentro dos nos e no vetor
        noguloso[i].item = itens[i];
        noguloso[i].razao = get_valor(itens[i]) / get_peso(itens[i
           ]);
    }
    /*ALGORITMO DE ORDENA  O MOMENT*/
    int infInicial = 0;
    int supInicial = n - 1;
    quicksort(noguloso, infInicial, supInicial);
    for (int i = n - 1; i >= 0; i--)//percorre de tras pra frente
       pq o vetor foi ordenado em ordem crescente
    {
    /*nItensArmazenados    usado como ind ce pois a quantidade de
        itens na mochila
    sempre sera igual ao index do item que estar  sendo operado
       no momento*/
```

```
37          if (mochila->pesoMax >= mochila->peso + get_peso(noguloso[
               i].item)) //verifica se o peso estoura o pesomax da
               mochila antes de adicionar o item
38          {
39              mochila->itensArmazenados[mochila->nItensArmazenados]
                   = noguloso[i].item; //armazena o item na mochila
40              mochila->valor+= get_valor(noguloso[i].item); //vai
                   somando o valor total da mochila
41              mochila->peso+= get_peso(noguloso[i].item); //vai
                   somando o peso total da mochila
42              mochila->nItensArmazenados += 1; //contador de itens
                   dentro da mochila
43          }
44      }
45      fim = clock();
46      tempoExec = ((double)(fim - inicio)/CLOCKS_PER_SEC);
47      printf("Tempo de execucao: %lf",tempoExec);
48      return mochila;
49  }
50
51  void quicksort(NOGULOSO *noguloso, int inf, int sup)
52  {
53      NOGULOSO aux;
54      int meio = (inf + sup) / 2;
55      float pivo = noguloso[meio].razao;
56      int i = inf;
57      int j = sup;
58      do{
59          while(noguloso[i].razao < pivo)
60          {
61              i++;
62          }
63          while(noguloso[j].razao > pivo)
64          {
65              j--;
66          }
67          if (i <= j)
68          {
69              aux = noguloso[i];
70              noguloso[i] = noguloso[j];
71              noguloso[j] = aux;
72              i++;
73              j--;
74          }
75      }while (i < j);
76      //chamadas recursivas
77      if (inf < j) {
78          quicksort(noguloso, inf, j);
79      }
80      if (i < sup) {
81          quicksort(noguloso, i, sup);
```

```
82      }
83  }
```

Listing 2: Implementação em C do Algoritmo Guloso

2.2 Equações de Recorrência

Não se aplica uma equação de recorrência para a abordagem gulosa implementada, pois o método foi iterativo.

2.3 Análise de Complexidade

Para analisar a complexidade, podemos dividir o algoritmo guloso em 3 partes:

- **Primeira parte: Inicialização e Cálculo das razões.** Nessa parte percorremos todos os n itens para a criação da struct `noguloso` e cálculo da razão valor/peso de cada um. Portanto, essa parte possui complexidade de $O(n)$.
- **Segunda parte: Ordenação.** Nessa parte chamamos a função `quicksort` para ordenar os n itens com base na razão valor/peso. Como a complexidade padrão do quicksort é $O(n \log n)$, a complexidade dessa etapa é $O(n \log n)$.
- **Terceira parte: Preenchimento da Mochila.** Nessa parte percorremos a lista de itens uma única vez para decidir quais colocar na mochila (um único laço `for`). Portanto, a complexidade dessa parte é $O(n)$.

Conclusão: A complexidade total do algoritmo é dada pela soma de cada etapa. Como o termo dominante é $O(n \log n)$, a complexidade do algoritmo guloso pode ser corretamente descrita por $O(n \log n)$.

2.4 Porque não atinge a solução ótima

A base do algoritmo guloso é realizar a melhor escolha em cada etapa a fim de encontrar a melhor solução final. Para o problema da mochila 0/1, esse ato se baseia em pegar sempre o item de maior razão valor/peso. O problema disso é que, no problema da mochila 0/1, pegar um item pode significar impedir a colocação de uma combinação de outros itens que, juntos, possuem um valor total maior. Portanto, o algoritmo guloso não fornece uma garantia de que a melhor solução será escolhida.

- **Exemplo:**
    1. Capacidade máxima 20kg.
    2. Itens disponíveis: Item 1 (20kg, R$100), Item 2 (10kg, R$60), Item 3 (12kg, R$72).
    3. O guloso ordenaria pela maior razão (Item 2 = Item 3 > Item 1) e escolheria o primeiro item de maior razão.
    4. Nesse caso, a escolha seria a pior possível, resultando em um valor de R$60, enquanto a solução ótima seria R$100 (apenas o Item 1).

2.5 Análise Empírica

*N*a análise de complexidade feita anteriormente, chegamos a conclusão que o algoritmo guloso tem complexidade temporal de $O(n \log n)$, o que já mostra que é uma escolha bem eficiente. Ao analisar a tabela e o gráfico abaixo, percebemos que a eficiência se comprova já que mesmo com um input grande, como n valendo 1 milhão, o programa demora um tempo muito baixo (menos de 1 segundo) para completar a execução. O gráfico, baseado nos valores da tabela também demonstra bem o comportamento da função, com uma forma definida de modo correspondente à complexidade teórica do algoritmo guloso.

Tabela 2: Tempo de execução do algoritmo guloso para diferentes tamanhos de entrada.

Numero de Itens (n)	**Tempo de Execucao (s)**
5	4×10^{-6}
10	4×10^{-6}
20	4×10^{-6}
100	$1,4 \times 10^{-5}$
500	$6,3 \times 10^{-5}$
1000	$1,73 \times 10^{-4}$
100000	$1,32 \times 10^{-2}$
1000000	$9,1 \times 10^{-1}$

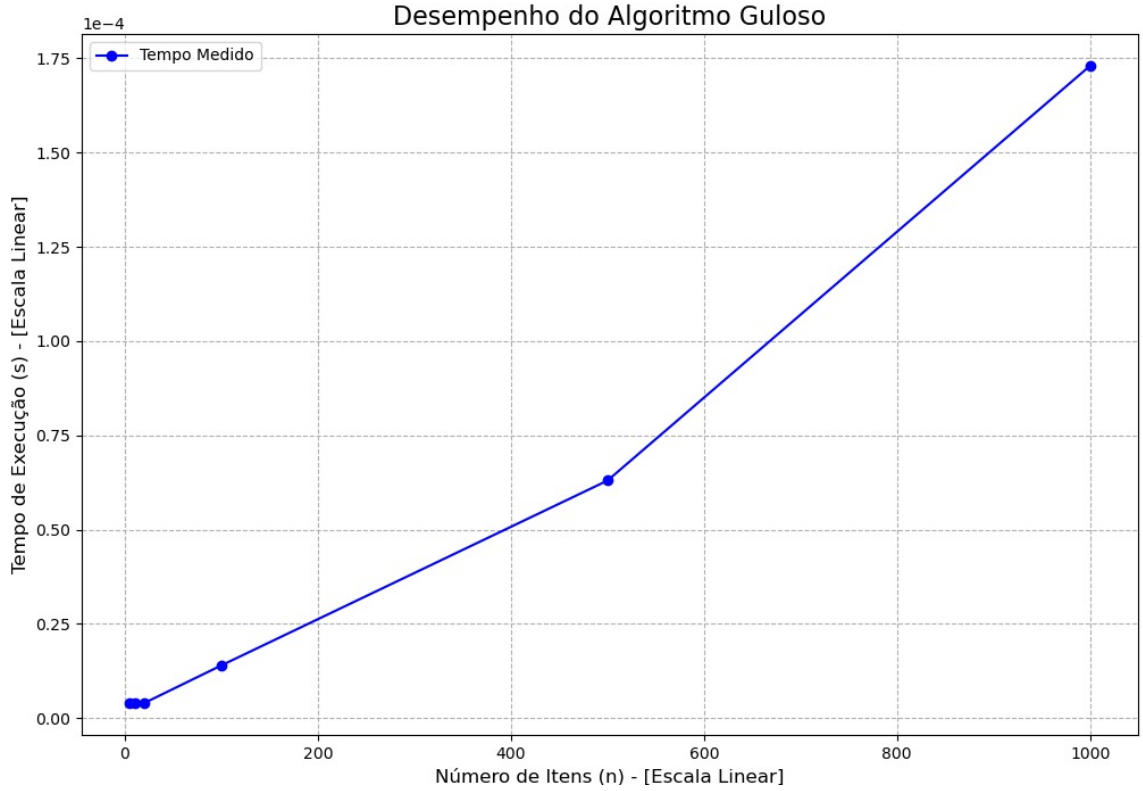

Figura 2: Gráfico com tempos de execução do algoritmo guloso

3 Resolução por Programação Dinâmica

3.1 Implementação

A lógica da dp é quebrar o problema em subproblemas e armazenar o resultado desses subproblemas para evitar recálcu-los, construindo uma tabela. Os subproblemas podem ser divididos em um caso base (mochila com capacidade nula ou quantidade nula de itens) e um geral, em que se o item atual tem o peso menor ou igual ao da mochila do sub-problema, temos a opção de incluir o item ou não, tudo depende se o valor total ao incluir ele é maior ou menor do que a solução anterior (sem incluir). Se o item atual não pode ser inserido, o melhor resultado é o anterior.

```
1  MOCHILA *programacao_dinamica(ITEM **itens, int pesoMax, int
       nItens){
2
3      clock_t inicio, fim; //marca de tempo
4      double tempoExec; //marca o tempo de execu  o
5      inicio = clock();
6
7      // Cria  o da tabela da programa  o din mica
8      float **dp = (float **)malloc((nItens + 1) * sizeof(float *));
9      if (dp == NULL){
10         return NULL;
11     }
12     for (int i = 0; i <= nItens; i++){
13         dp[i] = (float *)malloc((pesoMax + 1) * sizeof(float));
14         if (dp[i] == NULL) {
15             while(--i >= 0) free(dp[i]);
16             free(dp);
17             return NULL;
18         }
19     }
20
21     // Preenchimento da tabela da dp
22     for (int i = 0; i <= nItens; i++){
23         for (int j = 0; j <= pesoMax; j++){
24             // Sub-problema base (mochila com capacidade nula ou
                   quantidade nula de itens)
25             if (i == 0 || j == 0){
26                 dp[i][j] = 0;
27             }
28             else if(get_peso(itens[i-1]) <= j){
29                 /*Se o item atual tem o peso menor ou igual ao da
                       mochila do sub-problema, temos a op   o de
                      incluir o item ou n o, tudo depende se o
                      valor total ao incluir ele    maior ou menor
                      do que a solu   o anterior (sem incluir) */
30                 float valorPegar = get_valor(itens[i-1]) + dp[i
                       -1][j - get_peso(itens[i-1])];
31                 float valorNPegar = dp[i-1][j];
32                 dp[i][j] = maior(valorPegar, valorNPegar);
33             }
```

```
34              else{
35                  // Se o item atual n o pode ser inserido, o
                       melhor resultado    o anterior
36                  dp[i][j] = dp[i-1][j];
37              }
38          }
39      }
40
41      // Obtendo os itens da melhor solu   o e o peso dela
42      float valorDaSolucao = dp[nItens][pesoMax];
43      int pesoTotal = 0;
44      int cap = pesoMax;
45
46      // Contar quantos itens fazem parte da solu   o   tima .
47      int nItensSolucao = 0;
48      for (int i = nItens; i > 0 && cap > 0; i--){
49          if (dp[i][cap] != dp[i-1][cap]){
50              nItensSolucao++;
51              cap -= get_peso(itens[i-1]);
52          }
53      }
54
55      // Criando a mochila da melhor solu   o
56      MOCHILA *melhorMochila = (MOCHILA*) malloc(sizeof(MOCHILA));
57      if (melhorMochila == NULL){
58          printf("Erro ao alocar memoria para a melhor mochila (DP)\
                   n");
59          for (int i = 0; i <= nItens; i++){
60              free(dp[i]);
61          }
62          free(dp);
63          return NULL;
64      }
65
66      // Criando o array de itens da solu   o
67      melhorMochila->itensArmazenados = (ITEM**) malloc(
           nItensSolucao * sizeof(ITEM*));
68      if (melhorMochila->itensArmazenados == NULL){
69          printf("Erro ao alocar mem ria para o array de itens da
                   melhor mochila (DP)\n");
70          for (int i = 0; i <= nItens; i++){
71              free(dp[i]);
72          }
73          free(dp);
74          free(melhorMochila);
75          return NULL;
76      }
77
78      // Preenchendo a melhor mochila
79      melhorMochila->pesoMax = pesoMax;
80      melhorMochila->valor = valorDaSolucao;
```

```
81
82      // Encontrando os itens da melhor mochila e preenchendo o
            array interno
83      cap = pesoMax;
84      int k = 0; //  ndice  para o array 'itensArmazenados'
85      pesoTotal = 0; // Inicializa o acumulador de peso
86
87      for (int i = nItens; i > 0 && cap > 0; i--){
88          // Se o valor na c lula atual    diferente da c lula de
                cima, significa que a inclus o do item foi feita e ele
                 faz parte da melhor solu  o
89          if (dp[i][cap] != dp[i-1][cap]){
90              // Adi  o do item e seu peso
91              melhorMochila->itensArmazenados[k] = itens[i-1];
92              pesoTotal += get_peso(itens[i-1]);
93
94              cap -= get_peso(itens[i-1]);
95              k++;
96          }
97      }
98
99      // Atribui o peso total e o n mero de itens armazenados na
            melhor mochila
100     melhorMochila->peso = pesoTotal;
101     melhorMochila->nItensArmazenados = nItensSolucao;
102
103     // Liberar a mem ria da tabela DP
104     for (int i = 0; i <= nItens; i++) {
105         free(dp[i]);
106     }
107     free(dp);
108     fim = clock();
109     tempoExec = ((double)(fim - inicio)/CLOCKS_PER_SEC);
110     printf("Tempo de execucao: %lf",tempoExec);
111     return melhorMochila;
112 }
113 // Fun  o auxiliar para calcular o maior entre 2 n meros
114 float maior(float a, float b){
115     if (a > b){
116         return a;
117     }
118     else{
119         return b;
120     }
121 }
```

Listing 3: Implementação em C da Programação Dinâmica

(Favor inserir aqui a implementação em C do algoritmo de Programação Dinâmica.)

3.2 Equações de Recorrência

Apesar de ter sido feito por uma abordagem iterativa, é possível montar a equação de recorrência que define a estrutura do problema. Como a função depende do número de itens e da capacidade da mochila, não é possível descrever uma função $T(n)$ para o tempo de execução. A recorrência define o valor ótimo.

- **Definição do Subproblema:** Cada célula $dp[i][w]$ pode ser definida como o valor de uma função $F(i, w)$ que depende do número de itens (i) e capacidade da mochila (w).
- **Casos Base:** Ocorrem quando a mochila possui capacidade nula ($w = 0$) ou quando não temos itens para escolher ($i = 0$). Sendo $F(i, w) = 0$, se $i = 0$ ou $w = 0$.
- **Passo Recursivo:** Para cada item i e capacidade w, temos duas situações. Sendo v_i o valor do item i e p_i o peso do item i:
  1. Se o item não cabe ($p_i > w$), a solução é a mesma da célula $dp[i-1][w]$, portanto $F(i, w) = F(i-1, w)$.
  2. Se o item cabe ($p_i \leq w$), a solução é dada pelo maior valor entre incluir ou não o item: $F(i, w) = \max(F(i-1, w), v_i + F(i-1, w-p_i))$.

Por fim, ao juntar os casos, obtemos a equação de recorrência final:

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

3.3 Análise de Complexidade

Como a abordagem da função é iterativa, podemos analisar o código diretamente.

- **Primeira Etapa: Criação da tabela da DP.** Nela é alocada dinamicamente uma matriz de tamanho $(n+1) \times (W+1)$. Isso requer um laço que executa $O(n)$ vezes, e dentro de cada iteração, uma alocação de $O(W)$. Portanto, a complexidade de tempo desta etapa é $O(n \cdot W)$.
- **Segunda etapa: Preenchimento da tabela da DP.** Nessa etapa, temos dois laços `for` aninhados. O laço externo é executado $n+1$ vezes e o laço interno $W+1$ vezes. A complexidade total é dada pelo produto, portanto $O(n \cdot W)$.
- **Terceira etapa: Construção da melhor mochila.** Nesta etapa, a tabela é percorrida para encontrar os itens que compõem a melhor solução. Para isso, um laço `for` é executado n vezes, portanto a complexidade é $O(n)$.

Conclusão: A complexidade total é dada pela soma das complexidades: $T(n, W) = O(n \cdot W) + O(n \cdot W) + O(n)$. Portanto, a complexidade é $O(n \cdot W)$.

3.4 Análise Empírica

*C*onsiderando as entradas n e W, pode-se estabelecer algumas relações quanto o tempo de execução médio do algoritmo da dp. Como a complexidade teórica é $O(n \times W)$ não podemos analisar apenas o valor da entrada n, mas devemos analisar o peso máximo da mochila também. Considerando casos onde W é sempre igual a n, teríamos um algoritmo de complexidade $\mathrm{O}(n \times n) = \mathrm{O}(n^2)$. Nesse caso, o tempo de execução cresceria de modo exponencial em relação ao n. Porém, apenas empíricamente, podemos perceber que se considerarmos $\mathrm{x} = n \times W$, a função teria complexidade temporal de O(x), e portanto o tempo de execução cresce linearmente em relação ao produto $n \times W$, o que fica visível no gráfico e pela tabela. Apesar disso, deve-se reforçar que só é possível considerar esse crescimento linear em uma análise empírica, considerando o algoritmo como se possuísse apenas um input de valor $n \times W$.

Tabela 3: Tempo de execução da programação dinâmica para diferentes tamanhos de entrada.

Peso máximo da mochila	**Numero de Itens**	$n \times W$	**Tempo de Execução (s)**
10	5	50	4×10^{-6}
50	10	500	$1,8 \times 10^{-5}$
100	20	2×10^3	$4,6 \times 10^{-5}$
500	100	5×10^4	$9,22 \times 10^{-4}$
10000	1000	10^7	$1,31 \times 10^{-1}$
10000	10000	10^8	1,1
10000	100000	10^9	10,97
100000	100000	10^{10}	119,77

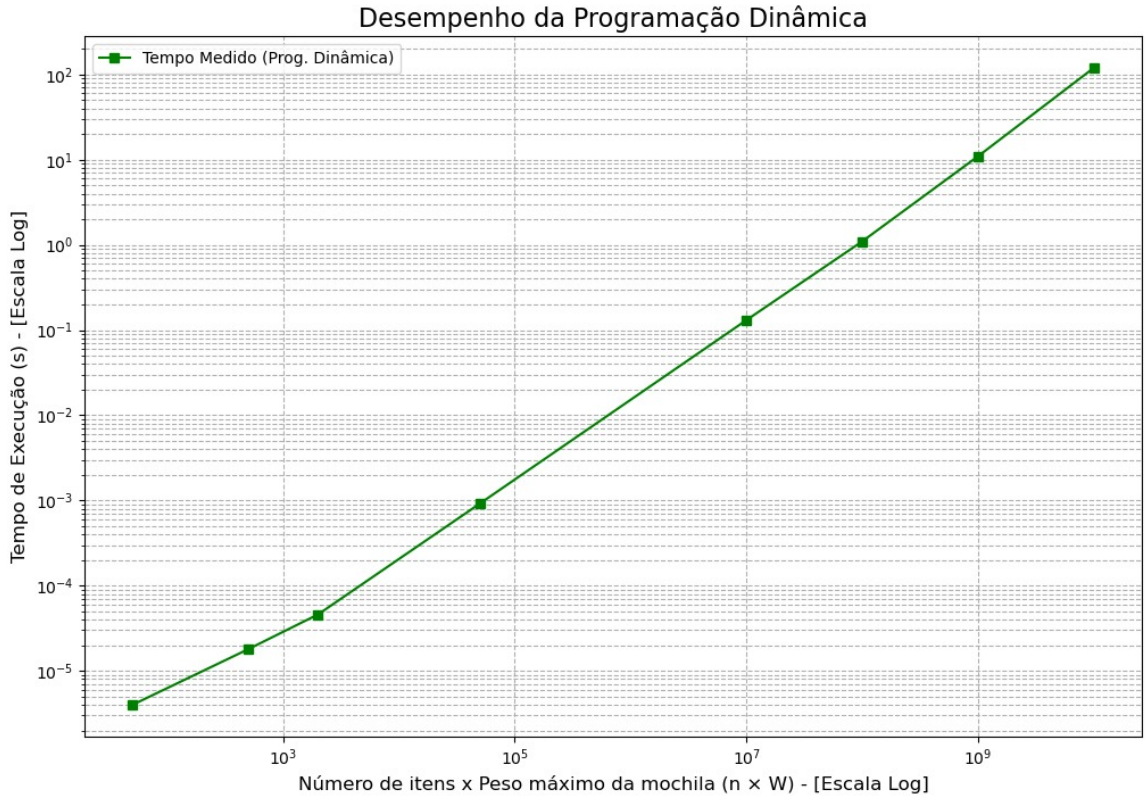

Figura 3: Tempos de execução da programação dinâmica

4 Comparações entre Abordagens Teóricas e Empíricas

4.1 Discussão da Diferença na Eficiência de cada Abordagem

Cada abordagem de solução possui características únicas e um trade-off diferente.

- **Abordagem de Força Bruta:** A lógica dela é gerar e avaliar todas as 2^n combinações possíveis de itens. Essa abordagem garante a solução ótima, pois todas as combinações são verificadas, sendo a solução mais direta e intuitiva. Contudo, é a mais ineficiente, com uma complexidade teórica de $O(2^n)$.

- **Abordagem do Algoritmo Guloso:** A lógica dela é sempre tomar a melhor decisão local, ou seja, escolher os itens com maior razão valor/peso. Para o correto funcionamento, é necessário ordenar esses itens pela razão, o que é o fator que define a complexidade de $O(n \log n)$. Teoricamente, essa abordagem é extremamente mais eficiente do que a força bruta. Contudo, essa abordagem não garante a solução ótima.

- **Abordagem da Programação Dinâmica:** A lógica dela é quebrar o problema em subproblemas e armazenar o resultado desses subproblemas para evitar recálculos, construindo uma tabela. A eficiência teórica desse algoritmo está relacionada ao preenchimento da tabela, sendo $O(n \cdot W)$. Teoricamente, essa abordagem oferece um meio termo, garantindo a solução ótima de forma mais eficiente que a força bruta, desde que a capacidade W não seja desproporcionalmente grande em comparação a n.

4.2 Comparação entre as Análises Empíricas de cada Abordagem

*C*omparando o algoritmos pelos resultados empíricos, nós percebemos que os resultados das análises teóricas de complexidade das abordagens condizem com a realidade. O algoritmo guloso é o mais eficiente, conseguindo, graças à sua complexidade de $n \log n$, processar 1 milhão de itens em menos de 1 segundo para resolver o problema, sem a necessidade de considerar o peso máximo da mochila. A abordagem de força bruta, com complexidade temporal de 2^n é inviável para a resolução do problema, a não ser que o valor de n seja pequeno, já que, com valores não tão grandes de n (100, por exemplo), temos tempos de execução de trilhões de anos, de modo que o tempo de execução pra essa abordagem, a depender do valor do n, só é possível de ser calculado com estimativas. A programação dinâmica, com sua complexidade de O($n \times W$), é um meio termo entre as outras 2 abordagens. Apesar de ter como desvantagem ter um tempo de execução que depende não só da quantidade de itens, mas também do peso máximo da mochila, essa abordagem possui um tempo de execução razoável para a maioria dos casos teste, mas se o número de itens for muito grande e o peso máximo da mochila não for pequeno para compensar, essa abordagem pode ter tempos de execução muito grandes também (mas nunca tanto quanto a força bruta). Portanto, para medir a eficiência desse algoritmo, é sempre necessário, como visto anteriormente com a análise teórica da complexidade, considerar o produto $n \times W$, de modo que o tempo de execução cresce aproximadamente linearmente em relação a esse produto.
*P*ortanto, baseado na análise empírica das abordagens, percebemos que se considerarmos apenas valores pequenos de n (até 20, por exemplo), não faz muita diferença qual

abordagem será escolhida, já que o processamento será quase instantâneo (se o peso máximo não for muito grande também, no caso da programação dinâmica), mas quando o n começa a crescer mais, atingindo valores de 100, 1000 ou até 100000, a força bruta já deve ser descartada como possibilidade. As outras continuam sendo viáveis em relação ao tempo de execução, mas, enquanto o tempo de execução da programação dinâmica não for muito grande e for utilizável, deve-se priorizar usá-la para resolver o problema da mochila, já que, apesar de ser a abordagem mais eficiente, o algoritmo guloso não garante que retornará a melhor mochila possível, como já discutido anteriormente. Por isso, a abordagem da programação dinâmica nos oferece o maior equilíbrio entre tempo de execução eficiente e resultado preciso.

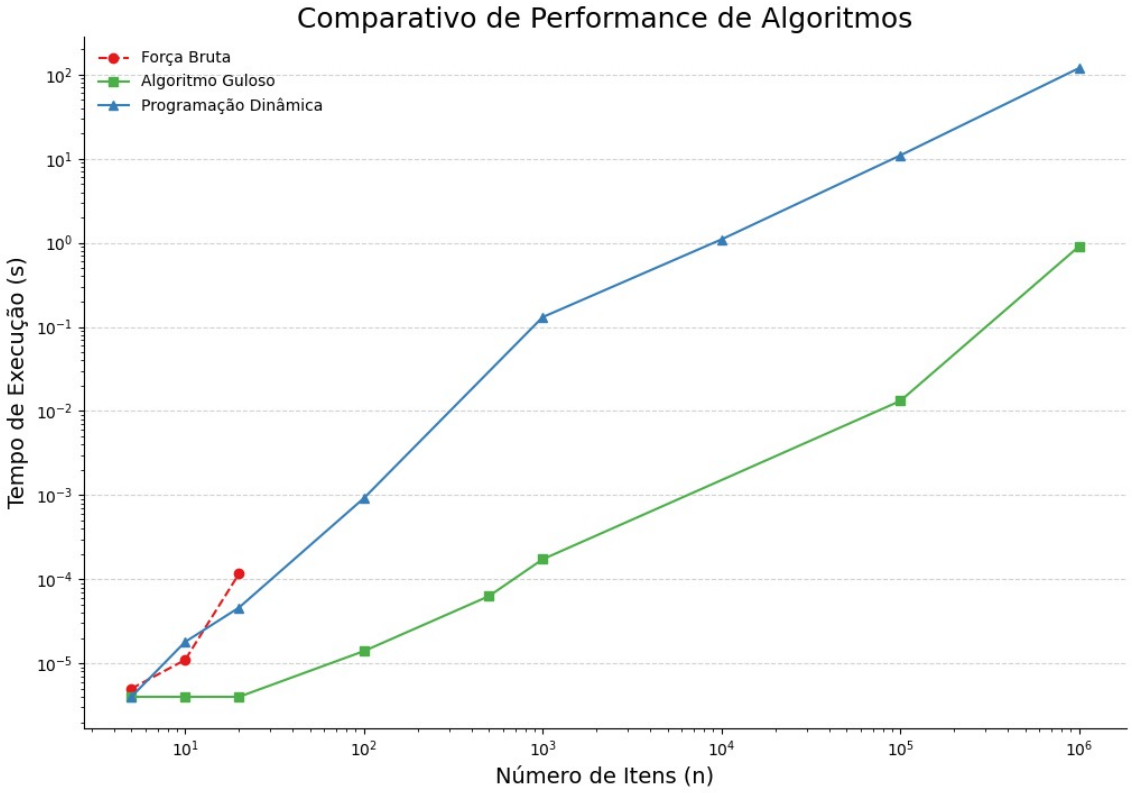

Figura 4: Comparação entre as Abordagens

O gráfico abaixo mostra uma visão comparativa da eficiência dos algoritmos utilizados. Para a programação, consideramos como n já o produto $n \times W$ para que se tenha uma visão aproximada mais realista do funcionamento do algoritmo. Podemos reforçar, pelo gráfico, como o algoritmo guloso é o mais eficiente em questão de tempo de execução e como o algoritmo de força bruta é tão pouco eficiente em relação a esse quesito, que o tempo de execução para um n valendo em torno de 100 já supera o dos outros algoritmos quando n vale 10^6 (por isso que o gráfico da força bruta aparece curto na imagem, já que o tempo de execução com os outros valores de n ultrapassam os limites do gráfico).