Universidade Federal de Viçosa - Campus Florestal

Guilherme Corrêa Melos 3882
Isabella Menezes Ramos 3474

Algoritmos e Estruturas de Dados I

Implementação do “Problema da Mochila”

Florestal - MG
7 de Novembro de 2019

Guilherme Corrêa Melos 3882
Isabella Menezes Ramos 3474

Implementação do “Problema da Mochila”

2° trabalho prático de Algoritmos e Estruturas de Dados 1. Trabalho que visa a avaliação do impacto causado pelo desempenho dos algoritmos intratáveis em sua execução real.

Professora: Prof ª. Thais R. M. Braga Silva
Disciplina: Algoritmos e Estruturas de Dados 1

Florestal - MG
7 de Novembro de 2019

Sumário

1 Introdução

O trabalho visa a implementação do algoritmo do “Problema da Mochila”, que é um exemplo de um algoritmo intratável, pois sua solução exata somente é possível através do cálculo e avaliação de todas as possíveis saídas, o que é chamado de força bruta.

O problema da mochila é um problema de otimização combinatória. O nome dá-se devido ao modelo de uma situação em que é necessário preencher uma mochila com objetos de diferentes pesos e valores. O objetivo é que se preencha a mochila com o maior valor possível, não ultrapassando o peso máximo.

2 Objetivos do Trabalho

O objetivo deste trabalho é reforçar os conhecimentos adquiridos em sala de aula e permitir a avaliação do impacto causado pelo desempenho de um algoritmo intratável e de complexidade exponencial em sua execução real. Perceber como varia o seu tempo de execução de acordo com os valores de entrada.

3 Desenvolvimento do Trabalho

O trabalho consiste em dois arquivos de cabeçalho, nomeados de *menu.h* e *knapsack.h.* O menu.h é composto de duas funções, o *mainMenu* é responsável pelo menu interativo e o *inputItem,* função usada para ler as entradas por arquivo.

```
12
13  void mainMenu(Item *items);
14  int inputItems(Item **items);
15
```

Figura 01. Funções implementadas no menu.h

Já o knapsack.h é composto das funções necessárias para se implementar a mochila. Apresenta uma estrutura chamada *Item* que é composta de três variáveis inteiras declaradas como *value*, *weight* e *qtd*, utilizadas para armazenar valor do objeto, peso e quantidade de itens, respectivamente. É definido também uma estrutura do tipo *Item* denominada *Backpack*, que será nossa mochila.

```
12  typedef struct item{
13      Value value;
14      Weight weight;
15      int qtd;
16  } Item;
17
18  typedef struct item Backpack;
```

Figura 02. Estruturas do knapsack.h

A função *knapsack* implementa o algoritmo do problema da mochila. O algoritmo original foi retirado site *Geek for Geeks*, é recursivo, e é responsável por fazer todas as combinações possíveis de acordo com um com conjunto de entradas. Adaptamos o algoritmo para que além de gerar todas as combinações, ele pegue essas combinações e compare se um conjunto de itens gerado na combinação na cabe na mochila e se ela estiver cheia, avaliar se o conjunto de itens que eu gerei é mais valioso que algum conjunto já presente na mochila, caso seja, ele irá substituir o conjunto por outro. E para gerar combinações de 1 até o valor de entrada do usuário, utilizamos um comando de repetição que, a cada chamada da função por ela mesma, o valor de início da estrutura de repetição é incrementado.

Para que a função knapsack não fique tão extensa, foi implementado funções auxiliares, e que são chamadas dentro dela. A função *setBackpack* inicializa uma variável do tipo de estrutura *Backpack*. Já a função *sumAttributes* adiciona valor e peso numa variável do tipo *Backpack*. O *storeItems* é responsável por fazer a alocação de itens que irão ser inseridos por arquivo e a função *freeItems* é responsável por liberar memória alocada de itens. Há também a função *printKnapsack* que é responsável por iniciar a função knapsack.

```c
void knapsack(Backpack *ans, Item *items , Item *res, Backpack backpack[], int start, int end, int index, int size) ;
void storeItems(int n, Item **itens);
void printKnapsack(Backpack *ans, Item *res, Item *items, int n, int size);
void setBackpack(Backpack *backpack, int size);
Backpack sumAttributes(Backpack backpack[], int size);
void freeItems(Item **items);
```

Figura 03. Funções implementadas no knapsack.h

```c
#include <stdio.h>
void combinationUtil(int arr[], int data[], int start, int end,
                     int index, int r);

void printCombination(int arr[], int n, int r)
{
    // A temporary array to store all combination one by one
    int data[r];

    // Print all combination using temprary array 'data[]'
    combinationUtil(arr, data, 0, n-1, 0, r);
}

void combinationUtil(int arr[], int data[], int start, int end,
                     int index, int r)
{
    if (index == r)
    {
        for (int j=0; j<r; j++)
            printf("%d ", data[j]);
        printf("\n");
        return;
    }

    for (int i=start; i<=end && end-i+1 >= r-index; i++)
    {
        data[index] = arr[i];
        combinationUtil(arr, data, i+1, end, index+1, r); } }

```

Figura 04. Algoritmo que imprime todas as combinações de um conjunto de números
Disponível em: https://www.geeksforgeeks.org/print-all-possible-combinations-of-r-elements-in-a-given-array-of-size-n/

Foi utilizado a biblioteca *time.h* para medir o tempo de execução. E durante os testes de tamanho de entrada, percebemos que quanto maior era a entrada, maior era o tempo que demorava para o algoritmo resolver o Problema da Mochila. Na descrição do trabalho prático, foi pedido para executar valores de entrada N de tamanhos, 50, 80 e 100. O nosso programa, não foi capaz de finalizar a execução com uma entrada N de tamanho 50, levou mais de 20 horas e o programa não havia terminado de executar. Com isso, não foi possível fazer testes com entradas de tamanho 80 e 100. Uma alternativa que achamos foi testar entradas com tamanhos menores, até um N em que seria possível ser executado num tempo que não levasse dias.

Calculamos então o tempo de execução para N de 1 até 34, pois a partir desse valor, a execução desse algoritmo se torna lenta. As especificações de hardware e software, obtidas através da ferramenta Inxi, da máquina que foi usada para testes, se encontra na figura abaixo.

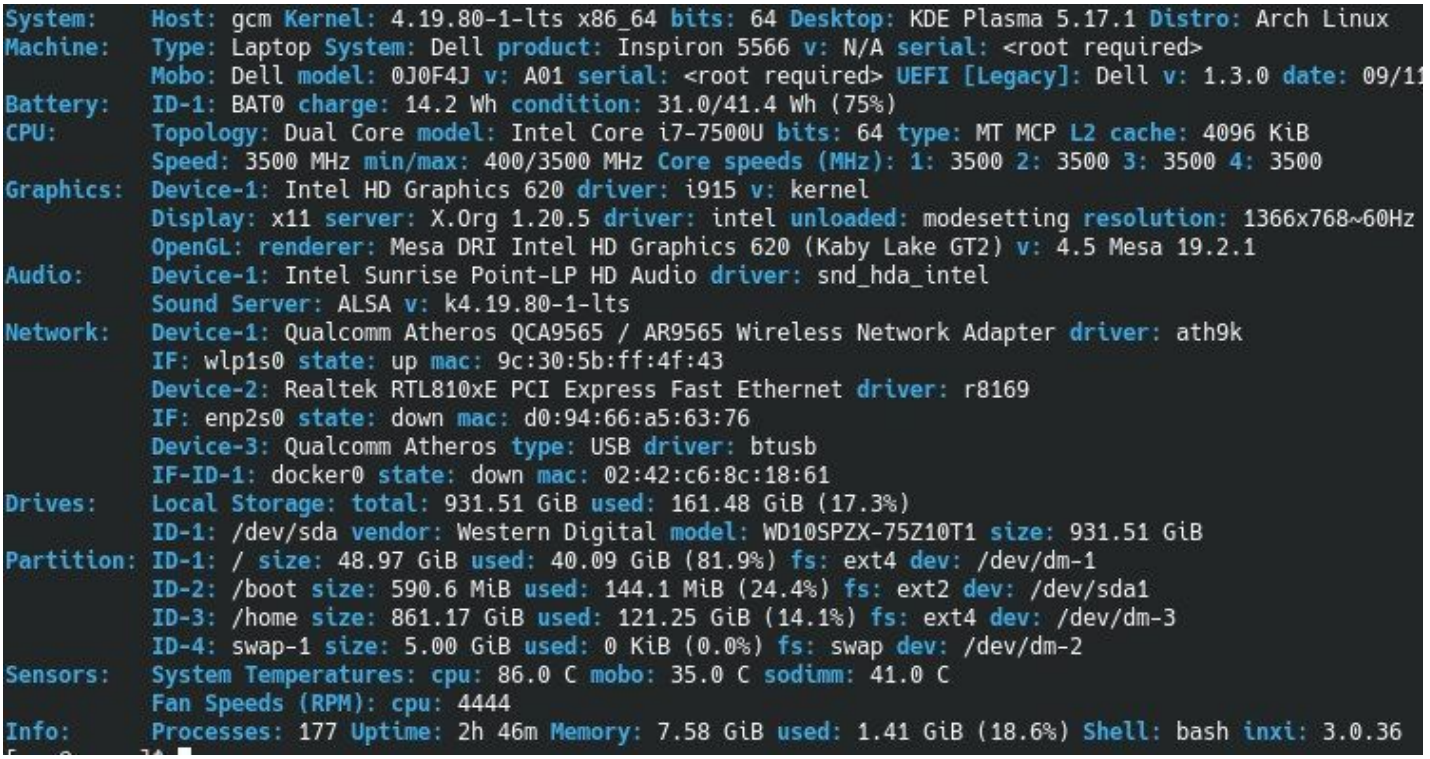

Figura 05. Configurações de hardware e software da máquina utilizada para dos testes

A seguir, a tabela e o gráfico com os resultados obtidos:

Tamanho da entrada	Tempo de execução	Tamanho da entrada	Tempo de execução	Tamanho da entrada	Tempo de execução
1	0.000036s	6	0.000021s	11	0.000115s
2	0.000013s	7	0.000027s	12	0.000219s
3	0.000016s	8	0.000029s	13	0.000417s
4	0.000016s	9	0.000047s	14	0.000833s
5	0.000027s	10	0.000067s	15	0.001673s

Tamanho da entrada	Tempo de execução	Tamanho da entrada	Tempo de execução	Tamanho da entrada	Tempo de execução
16	0.003554s	23	0.501270s	30	73.459009s
17	0.007286s	24	1.027818s	31	152.614568s
18	0.016152s	25	2.077023s	32	309.233767s
19	0.033016s	26	4.233190s	33	637.282370s

20	0.061439s	27	8.630929s	34	1261.441470s
21	0.127483s	28	17.664003s		
22	0.253447s	29	35.832439s		

Tabela 01. Tempo de execução de acordo com as entradas

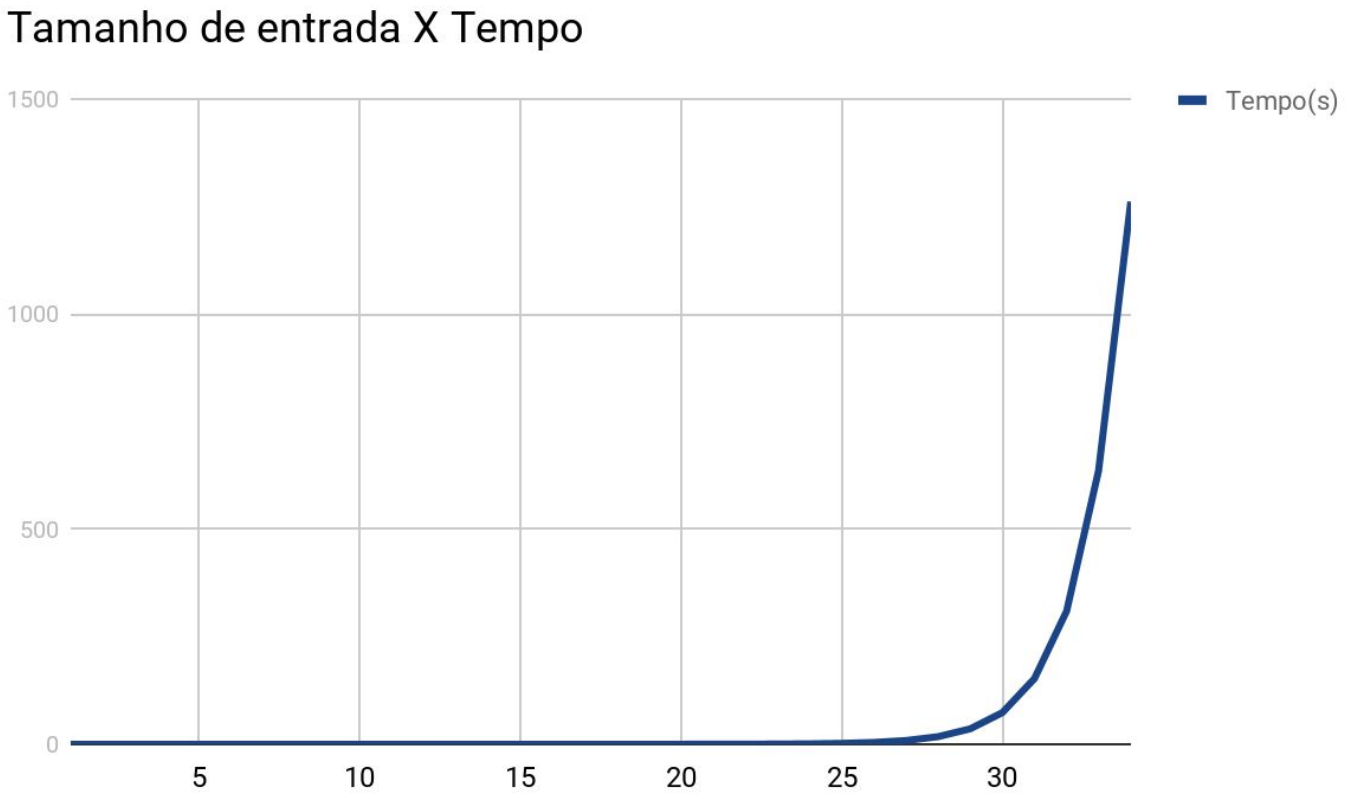

Gráfico 01. Relação do tamanho da entrada e tempo

Podemos perceber, pela tabela, que quando chega em um determinado número N na entrada, o algoritmo parece dobrar seu tempo de execução em relação com o número de entrada anterior a ele (N-1), que de fato é justificável pelo comportamento matemático do algoritmo representado pela série do número total de combinações $f(n)$, que também representa a função de complexidade do algoritmo, abaixo:

$$f(n) = \sum_{k=1}^{n} \binom{n}{k}$$

Portanto, temos que todo subconjunto S é um i-subconjunto. Sendo assim, pelo princípio da adição temos que a série pode ser representada por:

$$f(n) = \binom{n}{1} + \binom{n}{2} + \binom{n}{3} + \ldots + \binom{n}{n}$$

No entanto, tal identidade é decorrente também do Teorema Binomial abaixo:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k.$$

Tornando $x = 1$ e $y = 1$ obtemos:

$$f(n) = \sum_{k=1}^{n} \binom{n}{k} = 2^n - 1$$

Sendo assim, temos um algoritmo de complexidade O(2^n), portanto, utilizando uma mesma máquina podemos supor por indução o tempo tomado pelo algoritmo em uma máquina, dado os seus tempos de execução anteriores.

Se usarmos isso como referência, podemos supor que, o tamanho de entrada de N = 50, que foi pedido para ser testado no trabalho, demoraria aproximadamente 2.44388432268 nas condições da máquina em que os testes foram executados. Isso vale também para as outras entradas pedidas, de tamanho 80 e 100, que demorariam, respectivamente, na ordem de 10^7 e 10^{13} séculos, valores quais são maiores até mesmo do que a idade do universo, 13,6 bilhões de de anos. Números maiores que isso poderão levar para realizar o cálculo da mochila, como é o caso de se a entrada for de tamanho 200, que foi questionada na especificação, que não é viável executar o programa para esses tamanhos de entrada.

4 Conclusão

Com os resultados obtidos, percebemos que, o algoritmo do problema da mochila apresenta resultados variáveis de acordo com a entrada, quanto maior for o aumento dela, o seu tempo de execução irá aumentar drasticamente, podendo assim identificar que a curva de tempo cresce aproximadamente na proporção 2^n.

5 Referências Bibliográficas

ZIVIANI, Nivio. **Projeto de Algoritmos com Implementações em Pascal e C.** 3. ed. [s.i.]: Cengage Learning, 2010. 660 p.

PRINT all possible combinations of r elements in a given array of size n. Disponível em: <https://www.geeksforgeeks.org/print-all-possible-combinations-of-r-elements-in-a-given-array-of-size-n/>. Acesso em: 31 out. 2019.