

# SCC0201 - Introdução à Ciência da Computação II

## Relatório do Projeto 1

Alunos	NUSP
Alec Campos Aoki	15436800
Fernando Valentim Torres	15452340

## Análise Teórica

### Força Bruta

#### □ Implementação

```
#include "../brute.h"

int brute(ITEM **itemList, int maxWeight, int itemsQuantity) {
    if (itemsQuantity == 0 || maxWeight == 0) { //O(1)
        return 0; //O(1)
    }

    int itemWeight = getWeight(itemList[itemsQuantity - 1]); //O(1)
    if (itemWeight > maxWeight) //O(1)
        return brute(itemList, maxWeight, itemsQuantity - 1); //T(n-1)

    int bestPathIncludingTheItem = //O(1)
        getValue(itemList[itemsQuantity - 1]) + //O(1)
        brute(itemList, maxWeight - itemWeight, itemsQuantity - 1); //T(n-1)

    int bestPathExcludingTheItem = brute(itemList, maxWeight, itemsQuantity - 1); //T(n-1)
    return maxBetween(bestPathExcludingTheItem, bestPathIncludingTheItem); //O(1)
}

// T(n) = c, n=0 (caso base)
// T(n) = 2*T(n-1) + 2*n*c, n>0
```

#### □ Equações de Recorrência/Análise de Complexidade

A implementação força bruta consiste de testar todos os casos possíveis; isto é feito criando, a cada item, dois caminhos: o de incluí-lo e o de não incluí-lo. Obviamente, nos casos em que adicionar o item

é impossível (peso máximo da mochila ultrapassado), a única possibilidade é o de não incluí-lo. O caso base é quando não há mais itens a serem checados, ou a mochila já se encontra cheia. Analisando o código, chegamos em sua equação de recorrência (ver comentários no código abaixo). Resolvendo a árvore de recorrência, temos que a complexidade do algoritmo de força bruta é  $O(2^n)$ , ou seja, exponencial. Essa complexidade é coerente com a lógica do algoritmo, visto que ele checa todos os casos possíveis, e a taxa de crescimento exponencial é uma das mais rápidas possíveis, tornando este o método mais lento dentre os 3.

FORÇA BRUTA

$$T(n) = \begin{cases} c, & n=0 \\ 2T(n-1) + nc, & n>0 \end{cases}$$

[incluir ou não incluir item]

$T(n)$	$i$	ÁRVORE	CUSTO
$T(n) = 2T(n-1) + nc$	$i=0$	$n$	$2nc$
$T(n-1) = 2T(n-2) + (n-1)c$	$i=1$	$n-1$ $n-1$	$2(n-1)c$
$T(n-2) = 2T(n-3) + (n-2)c$	$i=2$	$n-2$ $n-2$ $n-2$ $n-2$	$4(n-2)c$
$T(n-3) = 2T(n-4) + (n-3)c$	$i=3$	$n-3$ $n-3$ $n-3$ $n-3$ $n-3$ $n-3$ $n-3$ $n-3$	$8(n-3)c$
		...	$2^i(n-i)c$

Altura da árvore =  $T(n-n) = T(1) = 1 \rightarrow 1 = n-i \rightarrow i = n-1$

Custo total =  $\sum_{i=0}^{n-1} 2^i(n-i)c = c \sum_{i=0}^{n-1} 2^i(n-i)$

$$= c \cdot (-n + 2^{n+1} - 2) \Rightarrow T(n) = O(c \cdot (-n + 2^{n+1} - 2))$$

$$\Rightarrow T(n) = O(-n + 2^{n+1} - 2) = O(2^{n+1}) = \boxed{O(2^n)}$$

## Algoritmo Guloso

### □ Implementação

```
#include "../greedy.h"
```

```
int greedy(ITEM **itemList, int maxWeight, int itemsQuantity) {  
  
    quicksort(itemList, 0, itemsQuantity - 1); //O(n^2) pior caso, O(n*log(n)) caso médio  
    float biggestMoneyAmount = 0; //O(1)  
    for (int i = 0; i < itemsQuantity; i++) { //O(n)  
        int itemWeight = getWeight(itemList[i]); //O(1)  
        if (itemWeight <= maxWeight) { //O(1)  
            maxWeight -= itemWeight; //O(1)  
            biggestMoneyAmount += getValue(itemList[i]); //O(1)  
        }  
    }  
    return biggestMoneyAmount; //O(1)  
}  
//Análise de recorrência não aplicável. T(n) = O(n^2)
```

## □ Equações de Recorrência/Análise de Complexidade

O algoritmo guloso funciona selecionando os arquivos com a maior relação  $\frac{\text{valor}}{\text{peso}}$ . Para isso, o vetor precisa estar ordenado. Em nossa implementação, calculamos todas as relações e as ordenamos usando o algoritmo *quicksort*, que tem complexidade  $O(n^2)$  em seu pior caso. Após o *quicksort*, a única operação dependente do tamanho da entrada é um laço *for* que percorre o vetor ordenado, ou seja, tem complexidade  $O(n)$ . Levando como limite superior a função de maior taxa de crescimento, temos  $T(n) = \theta(n^2)$ . Não há recursão, logo, não há caso base. Importante notar que esse algoritmo não garante uma solução ótima (não alcança a melhor resposta em 100% dos casos) para o problema da mochila 0/1 pois nesse problema devemos incluir ou excluir o item em sua totalidade (não podemos dividi-lo, como no problema da mochila fracionada). Podemos mostrar que o algoritmo guloso não é válido no problema 0/1 usando um exemplo: suponhamos uma mochila de peso 6, 1 item de valor 10 e peso 5 (relação = 2), e 2 itens de valores 5.5 e peso 3 (relação = 1,83); o primeiro item tem maior relação e seria escolhido pelo algoritmo guloso, mas os dois outros itens juntos resultam em um valor maior, apesar de terem uma relação menor. Esse método é o segundo mais eficiente dentre os três (a taxa de crescimento de  $n^2$  é menor que  $2^n$ ), apesar de não conseguir atingir a resposta certa para todos os casos. Selecionando um algoritmo de ordenação com pior caso menor que  $n^2$ , como o *mergesort* ( $O(n * \log(n))$ ), o limite superior passa a ser a checagem do vetor de razões, ou seja,  $O(n)$ .

## Programação Dinâmica

### □ Implementação

```
#include "../dynamic.h"

int dynamic(ITEM **itemList, int maxWeight, int itemQuantity) {
    int dp[itemQuantity + 1][maxWeight + 1]; //O(1)

    for (int i = 0; i <= itemQuantity; i++) { //O(n)
        for (int weight = 0; weight <= maxWeight; weight++) { //O(w)
            if (i == 0 || weight == 0) { //O(1)
                dp[i][weight] = 0; //O(1)
            } else if (getWeight(itemList[i - 1]) <= weight) { //O(1)
                dp[i][weight] = //O(1)
                    maxBetween(getValue(itemList[i - 1]) +
                                dp[i - 1][weight - getWeight(itemList[i - 1])],
                                dp[i - 1][weight]);
            } else {
                dp[i][weight] = dp[i - 1][weight]; //O(1)
            }
        }
    }

    return dp[itemQuantity][maxWeight]; //O(1)
}
```

## □ Equações de Recorrência/Análise de Complexidade

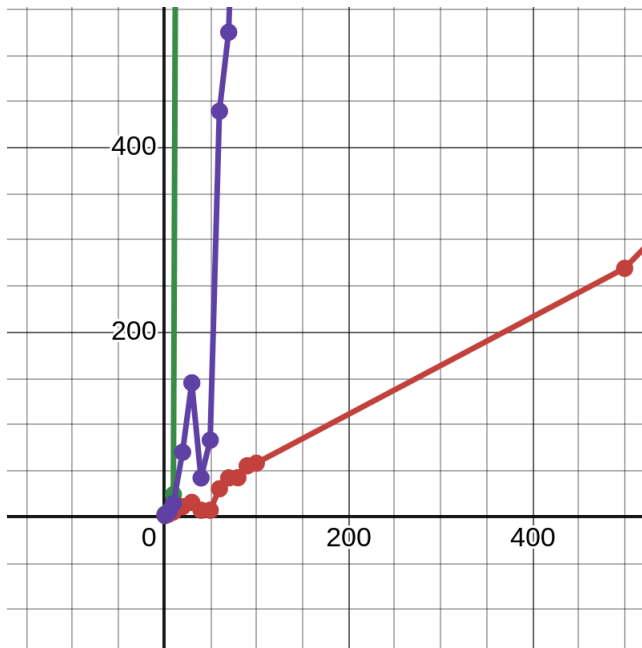
A solução do problema utilizando programação dinâmica consiste em **indiretamente** “testar” todas as possibilidades, sem necessariamente calculá-las repetidamente, por meio do uso do método da tabulação. Começamos construindo uma tabela com as colunas sendo os pesos máximos da mochila, e as linhas sendo cada item inserido. Analisamos coluna a coluna para cada linha; se conseguimos incluir o item da linha, verificamos se seu valor é superior ao definido para a mesma coluna, mas na linha anterior; se for, escrevemos esse valor na tabela, senão copiamos o valor da linha anterior.

Vamos dividir a análise de complexidade dessa solução em duas: uma chamada de *estados únicos* e outra chamada de *cache*. A complexidade de cache se refere à complexidade de cada execução da função (sem considerar recursões). Analisando o código acima, temos dois laços *for*; o primeiro tem limites  $[0, n]$ ,  $n$  a quantidade de itens recebidas, e o segundo tem limites  $[0, w]$ ,  $w$  o peso máximo máximo da mochila. Ambos são incrementados 1 a 1. Isso nos dá uma complexidade de cache de  $O(nw)$ . A complexidade de estados únicos se refere à chamadas recursivas (ou seja, o total de vezes que a função será chamada). Nesse caso, com a função sendo chamada somente uma vez (ela não é recursiva), temos que a complexidade de estados dela é  $O(1)$ . A complexidade de todo algoritmo pode ser dado unindo essas duas complexidades, o

que resulta em  $O(nw)$ . Podemos pensar, também, que por estarmos preenchendo uma matriz  $n$  por  $w$ , temos  $O(nw)$ .

## Análise Empírica

### Gráficos



**Força bruta** ( $O(2^n)$ )

**Guloso** ( $O(n^2)$ )

**Programação dinâmica** ( $O(nw)$ )

**Eixo X:** quantidade de itens ( $n$ )

**Eixo Y:** tempo de execução (em nanossegundos)

$x_1$	$y_1$
1	3
5	6
10	24
20	2561
30	8228.6
40	38672
50	197499

$x_2$	$y_2$
1	1.6
5	3
10	5.33
20	11
30	15.6
40	7
50	7
60	30.3
70	42
80	42.3
90	55
100	58
500	269
750	524

$x_3$	$y_3$
1	1.3
5	5
10	14.3
20	70
30	145
40	42
50	83
60	439
70	524.6
80	812.3
90	954
100	1067.3
500	14268
750	39628.6

## Discussões

---

### ▣ Resultados Obtidos (Comparação)

Conforme esperado, temos que a programação dinâmica é notavelmente mais eficiente que os outros dois métodos (como indicado por sua complexidade e implementação). Analogamente, temos que o algoritmo de força bruta é o menos eficiente (como também indicado por sua complexidade e implementação).

### ▣ Análise Teórica x Empírica

Conforme observável pelo gráfico, temos que, a grosso modo, cada algoritmo realmente opera conforme sua notação *big O*. No entanto, devido à própria natureza da notação, além de fatores inerentes ao computador que roda o código, o próprio código e os casos testes, que podem causar resultados que fogem do padrão esperado (como as quedas súbitas no tempo de execução observados no algoritmo guloso e na programação dinâmica), temos algumas divergências do comportamento puramente teórico esperado.