Lista 2 - Problema da Mochila

18 de Dezembro de 2021

Entrada: Duas sequências de números naturais - uma representando peso ($w_1, \ldots, w_n$ e outra valores ($v_1, \ldots, v_n$) de n objetos - e um valor natural W que representa a capacidade da mochila.

Saída: Um conjunto de índices I tais que a soma dos pesos não supere a capacidade ($\sum_{i \in w_i} \leq W$) e a soma dos valores $\sum_i \in I v_i$ seja a maior possível.

---

1 Exercício 1:

Escreva um algoritmo guloso que selecione objetos em ordem do maior valor para o de menor valor que não excedam a capacidade W . Mostre com um exemplo que este algoritmo não resolve o problema da mochila.

1.1 Reposta:

Dado um algoritmo guloso baseado em buscar sempre o objeto de maior valor, podemos observar de forma fácil casos em que ele não funciona:

Imagine o caso a seguir da tabela 1:

	0	1	2	3	4	5	6	7	8	9	10	11
w	10	1	1	1	1	1	1	1	1	1	1	1
v	10	2	2	2	2	2	2	2	2	2	2	2

Tabela 1: Caption

Caso você rode o algorítimo com $W = 10$ você obteria $I = \{1\}$ e $v = 10$. Porém neste caso, a melhor solução seria $I = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ que terá $v = 20$. Provando que o algorítimo está quebrado!!

2 Exercício 2:

Escreva um algoritmo de programação dinâmica que resolva o problema da mochila. Em função de W e n, assintoticamente qual o tempo de processamento de pior caso deste algoritmo? Também em função das mesmas variáveis, assintoticamente qual é o espaço de memória ocupado no pior caso?

2.1 Reposta:

O Algorítimo para este problema está localizado no anexo 1 e também no código fonte pode ser observado na classe *Dinamic.java*.

Baseado nos calculos feito no arquivo do anexo, pode-se observar que o algorítimo tem tempo de $O(n*w)$ e consome $O(n * w)$ de memória

3 Exercício 3:

Considere um arranjo de k bits A representando um número natural em notação binária. Esse arranjo começa com todas posições com 0 e é incrementado de um em um utilizando o seguinte algoritmo:

Algoritmo 1 Incrementa

```
1:  procedure INCREMENTA(A)
2:      i ← 1
3:      while i < k e A[i] = 1 do
4:          do A[i] ← 0
5:          i ← i + 1
6:      end while
7:
8:      if i < k then
9:          A[i]1
10:     end if
11: end procedure
```

Mostre, utilizando a técnica do contador para análise amortizada, que o tempo total e n operações de incremento tomam tempo total O(k).

3.1 Reposta:

Algoritmo 2 Incrementa

```
1:  procedure INCREMENTA(A)
2:      i ← 1                                          ▷ t = 1
3:      while i < k e A[i] = 1 do                      ▷ t = k
4:          do A[i] ← 0                                ▷ t = k
5:          i ← i + 1                                  ▷ t = k
6:      end while                                      ▷ t = k
7:
8:      if i < k then                                  ▷ t = 1
9:          A[i] ← 1                                   ▷ t = 1
10:     end if
11: end procedure                                      ▷ T = 4 · k + 3 => T = O(k)
```

4 Anexo

4.1 Anexo1:

```
public class Dinamic extends Algorithm{

    @Override
    public void run(int limitValue) { //O(n²)

        this.backpack = new Backpack(limitValue);                          //1

        int[][] array = new int[objects.size() + 1][limitValue + 1];       //1
        for(int i = 1; i <= objects.size(); i++){                          //W
          for(int j = 1; j <= limitValue; j++){                            //n*W
              if(objects.get(i-1).getWeight() < backpack.getFreeSpace())   //...
                  array[i][j] = array[i-1][j];                             //...
              else {                                                       //...
                  int notUse = array[i-1][j];                              //...
                  int use = array[i-1][j] + objects.get(i-1).getValue();   //...
                  array[i][j] = Math.max(use, notUse);                     //...
              }
          }
        }

        int j = limitValue;                                                //1
        for(int i = objects.size(); i >= 1; i--){                          //n
            if(array[i][j] == array[i-1][j]) {                             //...
                getResult().add(objects.get(i-1).getPositionArray());      //...
                backpack.addObject(objects.get(i-1));                      //...
            }else                                                          //...
                j = j - objects.get(i-1).getWeight();                      //...
        }

    }

}

//T = n*w*7 + W + n + 2 => T=7n*w + W + n + 2 => T = O(n*W)

/**
JÁ EM MEMORIA:
array =[n][W] m = n*W
objects =[n] m =n
Result = [n] m =n

M = n*W + 2n => M O(n*W)
**/
```