Lista 2

Kennedy Rohab Menezes da Silva, nºUSP: 12683395

19 de dezembro de 2021

1 Exercício 1

PROBLEMADAMOCHILA$(A_v, A_w, capacidade)$

1 $Ordenar(A_v)$
2 $A_I = \{\}$
3 $soma = 0$
4 **for** $i \leftarrow 1$ até $A_v.length$
5 **do if** $A_w[i] \leq capacidade$
6 **do if** $soma \leq capacidade$
7 **do if** $(soma + A_w[i]) \geq capacidade$
8 **then return** A_I
9 **else** $A_I[i] \leftarrow \{A_v[i], A_w[i]\}$
10 $soma = soma + A_w[i]$
11 **return** A_I

Na linha 5 Se peso for menor que a capacidade, entra.

Na linha 6 Se a soma dos pesos for menor que a capacidade, entra.

Na linha 7 Se a soma, mais o respectivo peso ultrapassar a capacidade, irá retorna o elemento.

O problema da mochila consiste em pegarmos o maior valor possível de uma sequência com o peso que a mochila possa carregar(capacidade). Vamos mostrar como o algoritmo funciona:

capacidade: 15 (por exemplo)
valores: 8 9 7 5
pesos: 2 15 5 10

O nosso algoritmo ordenaria os valores como segue: 9, 8, 7, 5, e na primeira rodada já entregaria o elemento de valor 9 e peso 15 (capacidade da nossa mochila). Porém há duas outras possibiliades de conseguir; uma com a corretude do algoritmo, ou seja, com o maior valor e menor peso, que é:

valores: 8+7 = 15

pesos: 2+5 = 7, muito melhor que o algoritmo anterior.

Então, fica claro que o nosso algoritmo não resolve o problema da mochila.

2 Exercício 2

2.1 O algoritmo[1]

O seguinte algoritmo, para que funcione, é necessário uma fórmula que auxiliará a decisão do item entrar ou não na mochila. Essa fórmula é seguida de uma tabela (V), vejamos:

V[i, j] = *máx*(V[i-1, j], V[i-1, w-w[i] + P[i]), em que:

i: é a linha da tabela;

j: é a coluna;

P: é o valor do respectivo item, assim, o algoritmo que resolve o problema da mochila é:

PROBLEMADAMOCHILA2($A_v, A_w, capacidade$)

$n = Av.length$
$m = capacidade$
$A_I = \{\}$
$tab[n+1][capaciadade+1]$
for $i \leftarrow 0$ até n
  do for $j \leftarrow 0$ até m
    if $i == 0$ ou $j == 0$
      then $tab[i][j]$
    if $A_w[i] \leq j$
      then $tab[i][j] = máx A_v[i] + tab[i-1][j - A_w[i]], tab[i-1][j])$
      else $tab[i][j] = tab[i-1][w]$
return ...

Tomando A_v = [1, 2, 5, 6] e A_w = [2, 3, 4, 5]

i/j	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0	0	1	2	5	5	6	7	7
4	0	0	1	2	5	6	6	7	8

Tabela 1: Saída após a execução do algoritmo *problemaDaMochila2*

Teremos uma tabela assim:

Lembrando que essa tabela surge com a aplicação da fórmula que vimos anteriormente e que também está no código. N entanto, note, ainda não sabemos quais elementos que iremos colocar na mochila, então precisamos completar o código:

```
PROBLEMADAMOCHILA2(A_v, A_w, capacidade)
n = Av.length
m = capacidade
A_I = {}
tab[n + 1][capaciadade + 1]
for i ← 0 até n
        do for j ← 0 até m
              if i == 0 ou j == 0
                 then tab[i][j]
              if A_w[i] ≤ j
                 then tab[i][j] = máx A_v[i] + tab[i − 1][j − A_w[i]], tab[i − 1][j])
                 else  tab[i][j] = tab[i − 1][w]
i = n; j = m
for i > 0 e j > 0
        do if tab[i][j] == tab[i − 1][j]
              then    A_I[i] = 0
     else
              A_I[i] = 1
              j = j − A_v[i]
return A_I
```

Agora sim! Quando a função retorna A_I, sabemos, pelos 1's e 0's quais elementos colocar na mochila e com o maior valor e o menor peso. Esta última parte ira olhar para a linha anterior e se houver um elemento igual, então esse elemento não entrará. Podemos ver que o último elemento entra na mochila, pois $8 \neq 7$. Depois j diminuirá o peso do item que entrar na mochila, nesse caso ele irá comparar com a coordenadas (i=3, j=2) e nesse caso haverá um 2 no item anterior ao terceiro, o terceiro não entra, compara-se então o segundo com o primeiro como $2 \neq 1$ o segundo item entra. Diminui um as coordenadas não permitem entrada, pois entram nos zeros da tabela. Entram na mochila o quarto e o segundo item, valor total de 8 e peso também 8.

2.2 O tempo

O tempo, como o nosso algoritmo possui dois *for*'s um dentro do outro (nested for loops), será $n * w$ uma vez que o primeiro *for* considera o número de elementos e o segundo a capacidade. Então, no pior caso, o nosso algoritmo toma tempo $\theta(n * w)$.

2.3 A memória

A memória é algo interessante de se notar. O Exercício pede que retornemos apenas a maior soma, o que evitaria a memória do array A_I, mas acredito que seria mais proveitoso, nesse caso, saber quais elementos entram e não apenas a soma.

Assim sendo, o algoritmo, usará memória para: 2 arrays necessários para armazenamento do valor e peso, 1 array bidimensional para a tabela, e 1 array para armazenar os 1's e 0's indicando a entrada ou não dos elementos.

Respectivamente, então, temos: 2n + n^w + n, esse seria o gasto de memória do algoritmo *problemaDaMochila2*.

3 Exercícios 3[2]

3.1 Contador Binário

O nosso algoritmo de incrementar números binário, cujo código podemos ver abaixo, vai incrementar o valor de acordo com duas condições simples:

enquanto $A[i] = 1$ $A[i]$ receberá 0 e o contador incrementará, saindo do laço se $i < A.comprimento$, então $A[i] = 1$.

```
INCREMENTA(A)
i = 0
while i < A.comprimento e A[i] = 1
    then A[i] = 0
         i = i + 1
if i < A.comprimento
    then A[i] = 1
```

Notemos que o $A[0]$ mudará todas as vezes que for incrementar o valor binário, $A[1]$ a metade das vezes, $A[2]$ um quarto das vezes, notamos então que se trata de uma inversão do tipo: $\frac{n}{2^i}$, em que n é o número de operações.

Para uma sequência de binário somente com 1's o algoritmo tomara tempo $O(n)$, pois todas as posições serão alteradas para 0, exemplo:

1 1 1 1 1 1 1 1 1

Como todos são 1's, eles irão mudar para 0, só para bem do exemplo vamos adicionar um 0 no final, ficaremos com:

0 1 1 1 1 1 1 1 1 , antes

1 0 0 0 0 0 0 0 0, depois. Veja que todas posições foram alteradas.

Note também que a sequência 1 1 0 0 1 1 0 1 1 1 1, com o incremento passará a ser: 1 1 0 0 1 1 1 0 0 0 0, e os últimos números de sequência 1 mudaram.

Então, a notação do tempo que toma a execução do algoritmo é

$$\sum_{i=0}^{k-1} \frac{n}{2^i} < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Assim, o tempo de pior caso num contador com apenas zero elementos é igual a $O(n)$

3.2 Contador binário com créditos

Como já vimos que o tempo de execução é $O(n)$, analisemos também pelo método da contabilidade. Nele é cobrado um valor de 2 crédito para cada inversão de bits 1, algo semelhante com o que vimos em sala de aula no exemplo da Pilha em que havia inserções e retiradas, e seus respectivos créditos.

Essa lógica se aplica para descobrirmos, ao final da execução com quanto ficamos, e nesse caso, nunca poderá ser um número negativo, pois de fato, não há tempo de execução negativa e porque temos os devidos créditos para comprovar.

Quando, no caso do contador de bits, é atribuído um número 1, são cobrados 2 créditos e nele mesmo já usado, o que nos deixa com um crédito reserva, e quando usado para atribuir o 0 temos esse crédito reserva para pagar, assim, quando no fim da execução do ***while***, devido a característica de alternância do contador de bits teremos que ao fim das operações o custo amortizado total é $O(n)$, limitando o custo real total.

Referências

[1] Abdul Bari. *4.5.1 0/1 Knapsack Problem (Program) - Dynamic Programming*. 2018. URL: `youtube.com/watch?v=zRza99HPvkQ&ab_channel=AbdulBari`.

[2] T. H. Cormen. *Algoritmos: teoria e prática*. Campus, 2012. ISBN: 9788535236996.