3.7 Subsequência de Maior Soma

1. Explicação do problema

Este problema busca, a partir da entrada de um vetor de inteiros (positivos ou negativos), retornar a maior soma de uma subsequência deste vetor.

2. Exemplos

Por exemplo, para a sequência $\{5,2,3,-2,80000,-4,-5,1\}$, a saída deverá ser 8008. Pois, a maior soma possível de ser formada se encontra entre $\{5,2,3,-2,8000\}$.

Também, se recebermos como entrada a sequência $\{-2,-4,-1,0,-3\}$, nosso retorno deverá ser 0 como a maior soma. Pois, ela é composta apenas por $\{0\}$.

3. Explicação da solução

Para solucionar este problema, foi utilizada a técnica de programação dinâmica. Foi elaborada a seguinte abordagem, definimos que nosso estado atual é a nossa atual posição no vetor. Ou seja, se o nosso vetor possui 6 posições, podemos afirmar que

$$SMS(6) = \begin{cases} SMS(5) + V[6], & if\ SMS(5) + V[6] > V[6] \\ V[6], & otherwise \end{cases}$$

Sendo assim, podemos concluir que cada posição $\boldsymbol{i}$ do nosso vetor, é composta por suas maiores somas das $\boldsymbol{i - 1}$ subsequências mais o valor da própria posição, mas isso é válido somente se esta soma for maior que o valor na posição atual. Caso contrário, devemos apenas considerar o valor em $\boldsymbol{i}$. Esta verificação ocorre, pois, se a soma for menor que o valor atual, isto implica que não estamos considerando a maior subsequência para nossa posição $\boldsymbol{i}$.

Podemos concluir então, que para uma dada posição $\boldsymbol{i}$ do nosso vetor, devemos calcular as $\boldsymbol{i - 1}$ maiores somas, para assim acharmos a maior soma da nossa atual localização.

Dessa forma, obtemos a seguinte recursão baseada em divisão e conquista

$$SMS(i) = \begin{cases} SMS(i-1) + V[i], & if\ SMS(i) + V[i] > V[i] \\ V[i], & otherwise \end{cases}$$

No entanto esta solução além de, calcular múltiplas vezes o mesmo subproblema, também computa apenas a maior soma para nosso estado atual. Desta forma, implica o uso de um

loop para iterarmos sobre cada estado do vetor fornecido e somente assim saberemos o estado que possui a subsequência com a maior soma.

Aplicando as técnicas de programação dinâmica, podemos perceber que nosso caso base sempre será composto pela primeira posição do nosso vetor e, como existem sobreposições de subproblemas, utilizaremos memoização para evitarmos processamento de casos já efetuados.

Sendo assim nossa solução por *DP* é composta por um vetor auxiliar V_2 do mesmo tamanho do vetor de entrada V_1. Iniciaremos nosso $V_2[0] = V_1[0]$, pois, este representa nosso caso base e que também nos auxiliará na solução *bottom-up*. Após estas inicializações, iremos percorrer o V_1 da sua posição 1 até seu final e salvaremos em $V_2[i]$ o valor máximo entre $V_2[i-1] + V_1[i]$ e $V_1[i]$.

Ao final deste processo, teremos a nossa maior soma de uma subsequência salva em nosso vetor V_2, sendo necessário apenas retornar o maior valor de V_2.

4. Implementação

https://github.com/duccl/CC5661-DynamicProgrammingList

5. Análise Assintótica

```
1. def sms(vetor):
2.     vetor_memo = [0 for i in range(len(vetor))]                  n
3.     vetor_memo[0] = vetor[0]                                     1
4.     for i in range(1,len(vetor)):                                n
5.         vetor_memo[i] = max(vetor_memo[i-1]+vetor[i],vetor[i]) (n-1)
6.     return max(vetor_memo)                                       n
7.
8. print(sms([-2,-3,4,-1,-2,0,1,5,-3]))
9. print(sms([5,2,3,-2,80000,-4,-5,1]))
```

$$\mathrm{T}(n) = n + 1 + n + (n-1) + n$$
$$\mathrm{T}(n) = 4n$$

Desta forma, pode-se afirmar que nossa solução por programação dinâmica é linear. Pois sua complexidade é $O(n)$.