Mais programação dinâmica

Sim.

Mas agora, a gente pode tentar otimizar os dois primeiros algoritmos também.

Quer dizer, não é (muito) difícil ver que eles também realizam chamadas recursivas repetidas

— (*você consegue ver?*)

Daí que, a gente pode tentar reduzir o seu tempo de execução utilizando a técnica da programação dinâmica.

Abaixo, nós temos a versão adaptada do primeiro algoritmo

```
Algoritmo SubconjMax-DC ( L, inicio, fim )
 {
    Se ( fim < inicio + 2 )            // i.e., menos do que 3 elementos
     {
        Retorna ( Max{ L[inicio..fim] } )
     }

    meio  <--  (inicio + fim) / 2


    // caso 1: L[meio-1] é selecionado

    S1  <--   L[meio-1]

    Se ( T[inicio,meio-3] != Nulo )  S1  <--  S1 + T[inicio,meio-3]
    Senão                            S1  <--  S1 + SubconjMax-DC ( L, inicio, meio-3 )

    Se ( T[meio+1,fim] != Nulo   )   S1  <--  S1 + T[meio+1,fim]
    Senão                            S1  <--  S1 + SubconjMax-DC ( L, meio+1, fim )


    // caso 2: L[meio] é selecionado

    S2  <--   L[meio-1]

    Se ( T[inicio,meio-2] != Nulo )  S2  <--  S2 + T[inicio,meio-2]
    Senão                            S2  <--  S2 + SubconjMax-DC ( L, inicio, meio-3 )

    Se ( T[meio+2,fim] != Nulo   )   S2  <--  S2 + T[meio+2,fim]
    Senão                            S2  <--  S2 + SubconjMax-DC ( L, meio+2, fim )



    // caso 3: L[meio+1] é selecionado

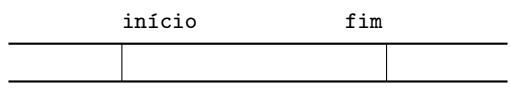
    S3  <--   L[meio-1]

    Se ( T[inicio,meio-1] != Nulo )  S3  <--  S3 + T[inicio,meio-1]
    Senão                            S3  <--  S3 + SubconjMax-DC ( L, inicio, meio-1 )

    Se ( T[meio+3,fim] != Nulo   )   S3  <--  S3 + T[meio+3,fim]
    Senão                            S3  <--  S3 + SubconjMax-DC ( L, meio+3, fim )


    Retorna ( Max{ S1, S2, S3 } )
 }
```

Bom, a novidade aqui é que os subproblemas são definidos por uma faixa da lista.



Daí que, ocorre uma chamada recursiva para cada par de valores dos parâmetros `início` e `fim`.

E dessa vez, a tabela `T` é indexada por duas variáveis.

Por outro lado, não é difícil ver que

- o tempo dentro da caixinha é $O(1)$ — (*porque não há laços ...*)

Daí que, o tempo de execução do algoritmo pode ser estimado assim

$$T(n) \quad = \quad \#\ \text{caixinhas} \quad \times \quad O(1)$$

Só que, nós temos uma caixinha para cada par de valores dos parâmetros `início` e `fim`.

E cada um desses parâmetros varia de 1 a n.

Logo, o tempo de execução do algoritmo fica assim

$$T(n) \quad = \quad O(n^2) \quad \times \quad O(1) \quad = \quad O(n^2)$$

Quer dizer, o tempo de execução do algoritmo foi reduzido de $O(n^{2,58})$ para $O(n^2)$.

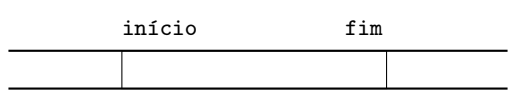
E isso nos deixa otimistas para otimizar o segundo algoritmo também.

Mas, pensando um pouquinho sobre o assunto, a gente vê que dessa vez isso não vai ser possível.

Quer dizer, o algoritmo já executa em tempo $O(n^2)$.

E ele também quebra a lista no meio.

Daí que, os subproblemas também tem a forma



E será preciso fazer uma chamada recursiva para cada par de valores dos parâmetros `início` e `fim`.

Só isso já leva tempo $O(n^2)$ — (*mesmo eliminando as chamadas recursivas repetidas*)

Logo, a técnica da programação dinâmica não nos dá um ganho significativo nesse caso.

Nota: *Devido ao padrão de quebra do problema, essa análise talvez possa ser melhorada ...*