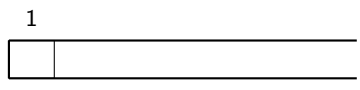
Programação dinâmica

Bom, agora a gente vai tentar pensar de uma maneira diferente.

Quer dizer, ainda é possível fazer uma outra observação

- Focando no primeiro elemento da lista



só existem duas opções: ou `L[1]` está na solução, ou ele não está

Quer dizer, agora nós podemos considerar em separado

- as soluções que contém o elemento `L[1]`
- as soluções que não contém o elemento `L[1]`

Em ambos os casos, a decisão sobre `L[1]` produz um único subproblema um pouco menor



A ideia é que essa estratégia pode ser aplicada recursivamente.

E isso nos dá mais um algoritmo de divisão e conquista para o problema

```
                Algoritmo SubconjMax-DC3 ( L, inicio, n )
                 {
O(1)  |             Se ( inicio = n )   Retorna ( L[n] )

                    // caso 1: L[inicio] é selecionado

T(n − 2)  ←         S1  <--   SubconjMax-DC ( L, inicio+2, n )
                             +
                              L[inicio]

                    // caso 2: L[inicio] não é selecionado

T(n − 1)  ←         S2  <--   SubconjMax-DC ( L, inicio+1, n )

                    Retorna ( Max{ S1, S2 } )
O(1)  |          }
```

A partir das anotações, nós obtemos a seguinte recorrência para o tempo de execução do algoritmo

$$T(n) \quad = \quad T(n-1) \quad + \quad T(n-2) \quad + \quad O(1)$$

Mas, isso é quase a recorrência de Fibonacci — (*é um pouco mais que isso ...*)

Daí que,

$$T(n) \quad = \quad O(\mathsf{F_n})$$

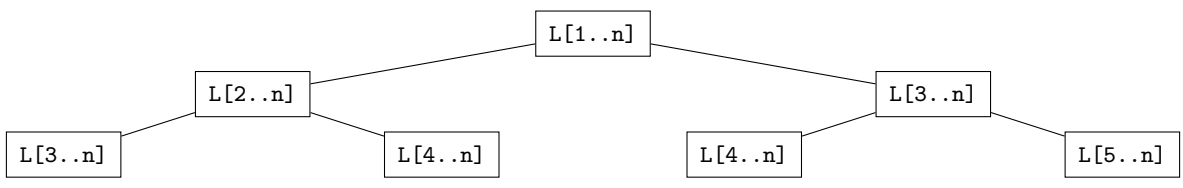
Só que $\mathsf{F_n}$ é uma quantidade exponencial em n.

O que significa que esse algoritmo é horrível.

Mas, ainda há esperança ...

Quer dizer, existe uma razão para o tempo de execução ser tão ruim assim.

E essa razão aparece quando a gente examina a árvore de recursão do algoritmo



Sim! o algoritmo está fazendo a mesma chamada recursiva múltiplas vezes.

Só que isso é um problema bem fácil de resolver.

Quer dizer, basta armazenar o resultado das chamadas recursivas em uma tabela auxiliar.

Daí, na hora em que a chamada está para ser realizada, a gente verifica se o seu resultado já está na tabela.

E se o resultado estiver lá, a chamada não é realizada.

Essa técnica é conhecida como a *programação dinâmica*.

E abaixo, nós temos a nova versão do algoritmo

```
Algoritmo SubconjMax-PD ( L, inicio, n )
 {
    Se ( inicio = n )   Retorna ( L[n] )

    // caso 1: L[inicio] é selecionado
    Se ( T[inicio+2] != Nulo )
        S1  <--  T[inicio+2]  +  L[inicio]
    Senão
        S1  <--  SubconjMax-DC ( L, inicio+2, n )  +  L[inicio]

    // caso 2: L[inicio] não é selecionado
    Se ( T[inicio+2] != Nulo )
        S2  <--  T[inicio+1]  +  L[inicio]
    Senão
        S2  <--  SubconjMax-DC ( L, inicio+1, n )  +  L[inicio]

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

Análise

Legal.

Mas, quanto tempo leva isso ?

Bom, a primeira observação é que

1. Ignorando o tempo gasto com as chamadas recursivas, a função executa em tempo $O(1)$
   — (*i.e., esse é o tempo dentro da caixinha ...*)

E a segunda observação é a seguinte

2. Não ocorrem duas chamadas recursivas com o mesmo valor do parâmetro `início`
   — (*i.e., cada caixinha só executa uma vez*)

Com base nessas observações, é fácil ver que

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

Mas, nós temos uma chamada para cada valor do parâmetro `início`.

E o parâmetro `início` varia de 1 até n.

Daí que,

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

Não é legal?