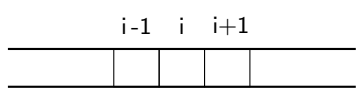
Divisão e conquista

Bom, a única coisa que a gente sabe fazer é divisão e conquista.

E a seguinte observação nos dá uma primeira estratégia de decomposição

- Em qualquer grupo de 3 elementos consecutivos



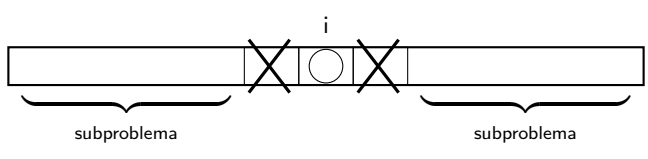
  ao menos um deles vai estar na solução

  — (*porque uma solução que não contém nenhum dos três pode ser melhorada adicionando o elemento do meio*)

Quer dizer, agora nós podemos considerar em separado

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

Em cada um desses casos, a inclusão do elemento na solução quebra o problema em dois



A ideia é que essa estratégia pode ser aplicada recursivamente.

E isso já nos dá um algoritmo de divisão e conquista

```
T(n)  =
                Algoritmo  SSMax-DC ( L, inicio, fim )
                 {
                    Se ( fim < inicio + 2 )                 //  menos do que 3 elementos
      O(1)
                        Retorna ( Max{ L[inicio..fim] } )

                    meio  <--  (inicio + fim) / 2

                    // caso 1: L[meio-1] é selecionado
2 · T(n/2) ←
                    S1  <--   SSMax-DC ( L, inicio, meio-3 )
                            +
                              SSMax-DC ( L, meio+1, fim )  +  L[meio-1]

                    // caso 2: L[meio] é selecionado
2 · T(n/2) ←
                    S2  <--   SSMax-DC ( L, inicio, meio-2 )
                            +
                              SSMax-DC ( L, meio+2, fim )  +  L[meio]

                    // caso 3: L[meio+1] é selecionado
2 · T(n/2) ←
                    S3  <--   SSMax-DC ( L, inicio, meio-1 )
                            +
                              SSMax-DC ( L, meio+3, fim )  +  L[meio+1]

                    Retorna ( Max{ S1, S2, S3 } )
      O(1)      }
```

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

$$T(n) \quad = \quad 6 \cdot T\left(\tfrac{n}{2}\right) \quad + \quad O(1)$$

E para obter a solução, nós desenrolamos a recorrência

$$\begin{aligned}
T(n) &= 6 \cdot T\left(\tfrac{n}{2}\right) + 1 \\
&= 6 \cdot \left[6 \cdot T\left(\tfrac{n}{2^2}\right) + 1\right] + 1 \\
&= 6^2 \cdot T\left(\tfrac{n}{2^2}\right) + 6 + 1 \\
&= 6^3 \cdot T\left(\tfrac{n}{2^3}\right) + 6^2 + 6 + 1 \\
&= \ldots \\
&= 6^{\log_2 n} \cdot T(1) + 6^{\log_2 n - 1} + \ldots + 6 + 1
\end{aligned}$$

Quer dizer, a gente obteve uma soma de PG.

E isso é uma coisa que a gente resolve assim

$$\mathsf{S} \quad = \quad 1 + 6 + 6^2 + \ldots + 6^k$$

$$= \quad 1 + 6 \cdot \underbrace{\left[1 + 6 + \ldots + 6^{k-1}\right]}_{\mathsf{S} \; - \; 6^k}$$

donde

$$\mathsf{S} \quad = \quad 1 + 6 \cdot \mathsf{S} - 6^{k+1}$$

donde

$$\mathsf{S} \quad = \quad \frac{6^{k+1} - 1}{5}$$

Daí que,

$$T(n) \quad = \quad \frac{6^{\log_2 n + 1} - 1}{5} \quad = \quad O\left(6^{\log_2 n}\right) \quad = \quad O\left(n^{\log_2 6}\right) \quad = \quad O\left(n^{2{,}58}\right)$$

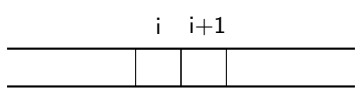
Legal.

Agora, a gente já tem um algoritmo porcaria para o problema.

Mas, como é que a gente faz melhor do que isso ?

Outra estratégia de decomposição

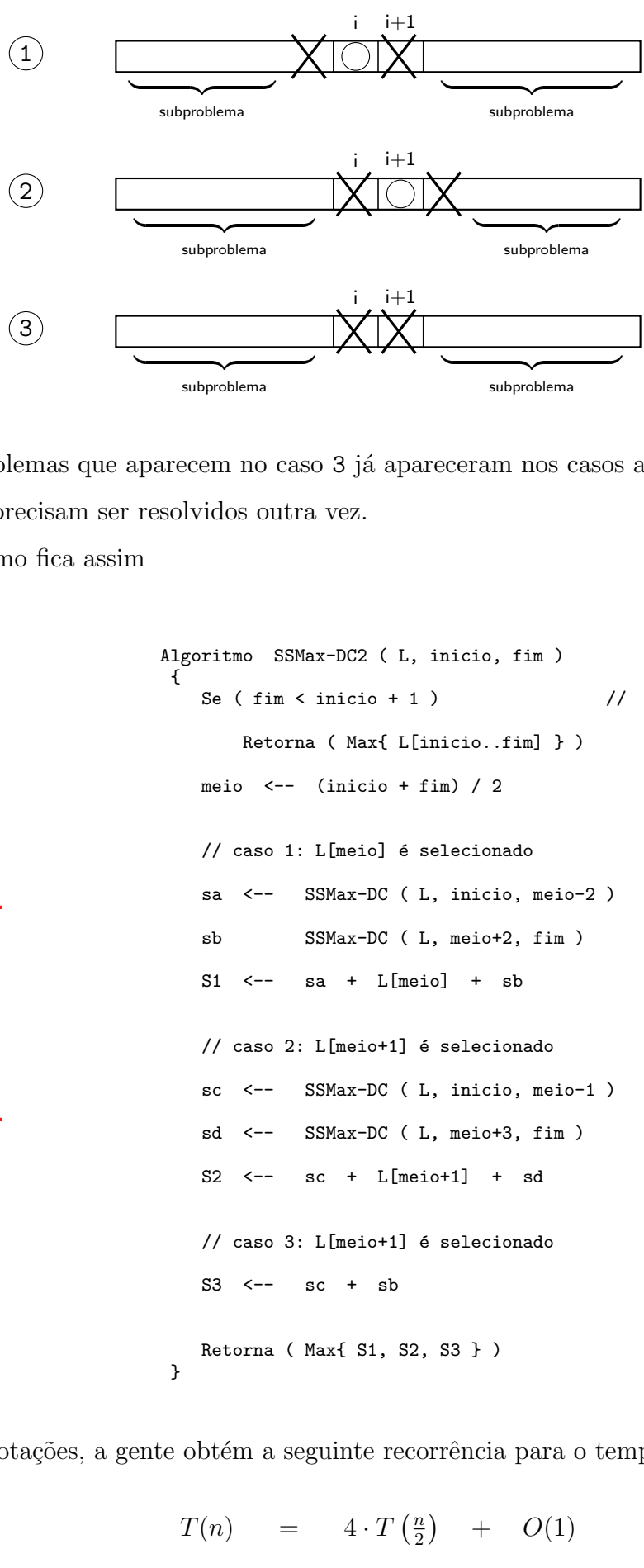
Bom, agora a gente pode olhar para dois elementos consecutivos



E daí, a gente nota que das 3 uma

1. ou o elemento `L[i]` está na solução
2. ou o elemento `L[i + 1]` está na solução
3. ou nenhum dos dois elemento está na solução

A seguir, a gente vê o que acontece em cada caso



Sim! os subproblemas que aparecem no caso 3 já apareceram nos casos anteriores.

Logo, eles não precisam ser resolvidos outra vez.

E daí, o algoritmo fica assim

```
T(n)  =
                Algoritmo  SSMax-DC2 ( L, inicio, fim )
                 {
                    Se ( fim < inicio + 1 )                 //  menos do que 2 elementos
      O(1)
                        Retorna ( Max{ L[inicio..fim] } )

                    meio  <--  (inicio + fim) / 2

                    // caso 1: L[meio] é selecionado
2 · T(n/2) ←
                    sa  <--   SSMax-DC ( L, inicio, meio-2 )
      O(1)          sb        SSMax-DC ( L, meio+2, fim )
                    S1  <--   sa  +  L[meio]  +  sb

                    // caso 2: L[meio+1] é selecionado
2 · T(n/2) ←
                    sc  <--   SSMax-DC ( L, inicio, meio-1 )
      O(1)          sd  <--   SSMax-DC ( L, meio+3, fim )
                    S2  <--   sc  +  L[meio+1]  +  sd

                    // caso 3: L[meio+1] é selecionado
      O(1)          S3  <--   sc  +  sb

                    Retorna ( Max{ S1, S2, S3 } )
      O(1)      }
```

A partir das anotações, a gente obtém a seguinte recorrência para o tempo de execução do algoritmo

$$T(n) \quad = \quad 4 \cdot T\left(\tfrac{n}{2}\right) \quad + \quad O(1)$$

Daí, olhando para o padrão da recorrência, a gente advinha que ela desenrola assim

$$T(n) \quad = \quad 4^{\log_2 n} \cdot T(1) \quad + \quad 4^{\log_2 n - 1} \quad + \quad \ldots \quad + \quad 4 \quad + \quad 1$$

O que tem solução

$$T(n) \quad = \quad O\left(n^{\log_2 4}\right) \quad = \quad O\left(n^2\right)$$

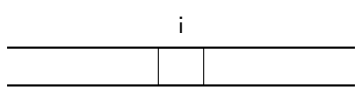
Quer dizer, o algoritmo melhorou um pouquinho.

Mas, ele ainda é uma porcaria.

E agora?

Mais uma estratégia de decomposição

Bom, agora a gente olha para um único elemento da lista



Em ambos os casos, o problema é quebrado em dois



E aqui a gente reconhece os mesmos subproblemas que apareceram na estratégia anterior.

Quer dizer, o algoritmo baseado nessa decomposição também vai fazer 4 chamadas recursivas.

E o seu tempo de execução também vai ser descrito pela recorrência

$$T(n) \quad = \quad 4 \cdot T\left(\tfrac{n}{2}\right) \quad + \quad O(1)$$

Logo, o seu tempo de execução também vai ser

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

E agora?