Imagine que L é uma lista de tamanho n que contém números inteiros

O problema é

→ *Encontrar o par de elementos que nos dá o maior salto*

Na aula passada, nós vimos um algoritmo de tempo $O(n \log n)$ para esse problema.

E agora, a gente vai obter um algoritmo melhor.

Como?

Ora, aplicando a ideia da aula de hoje — (*raciocinando a partir de baixo*)

Quer dizer, é bem fácil resolver o problema quando $n = 1$

— (*porque o único salto tem tamanho 0*)

E também é bem fácil resolver o problema quando $n = 2$

— (*porque basta verificar se o salto de um elemento para o outro é maior que 0*)

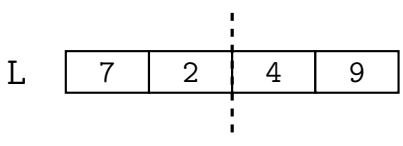
Então, nós vamos começar escrevendo o algoritmo que que resolve o problema nesses dois casos

```
func PMS-base ( L, inicio, fim )

    Se (inicio = fim)     Retorna (0)

    Senão                                              // fim = inicio + 1

        salto ← L[fim] - L[inicio]

        Se (salto > 0)     Retorna (salto)

        Senão                 Retorna (0)
```

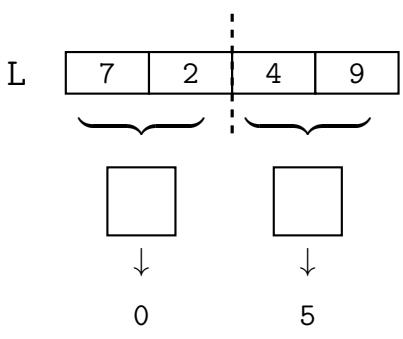
Legal.

Agora a gente imagina que o problema tem tamanho 4

E daí, a gente quebra o problema na metade

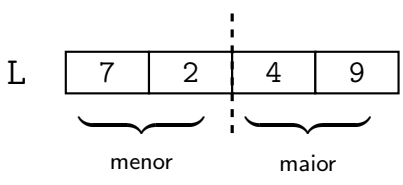


A ideia é chamar o algoritmo base em cada metade



Só que isso ainda não resolve o problema.

Quer dizer, ainda é preciso encontrar o menor do lado esquerdo, e o menor do lado direito



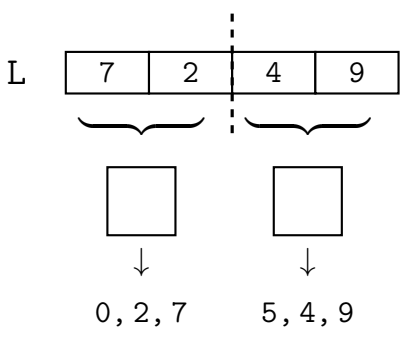
— (*para encontrar o maior salto que cruza o meio*)

E aqui é a hora para uma esperteza.

Quer dizer, a ideia é modificar o algoritmo base para que ele também retorne o menor e o maior elementos da lista

```
func PMS-base ( L, inicio, fim )

    Se (inicio = fim)      Retorna (0, L[inicio], L[inicio])

    Senão

        salto ← L[fim] - L[inicio]

        Se (salto > 0)     Retorna (salto, L[inicio], L[fim])

        Senão                 Retorna (0, L[fim], L[inicio])
```

Agora, a situação fica assim



E nós já temos o menor do lado esquerdo e maior do lado direito — (*sem precisar percorrer cada metade ...*)

Mas para a coisa continuar funcionando, o algoritmo também precisa retornar o menor e o maior elemento da lista.

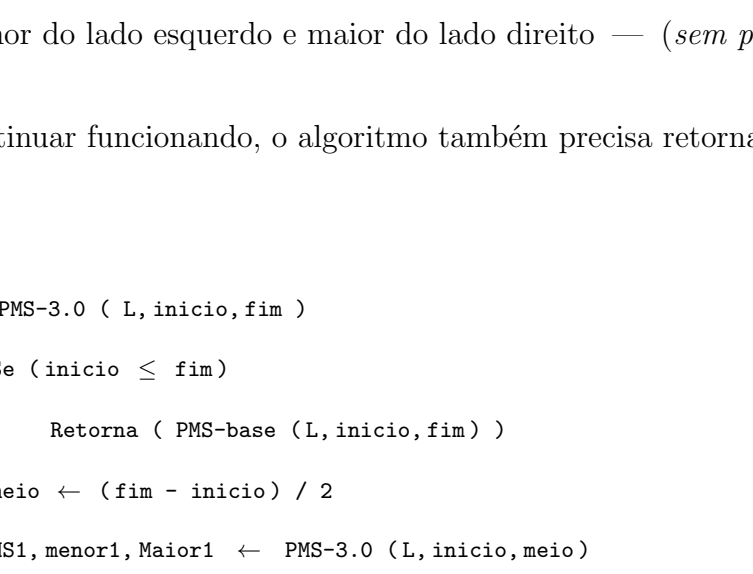
A coisa fica assim

```
func PMS-3.0 ( L, inicio, fim )

    Se (inicio ≤ fim)

        Retorna ( PMS-base (L, inicio, fim) )

    meio ← (fim - inicio) / 2

    MS1, menor1, Maior1 ← PMS-3.0 (L, inicio, meio)

    MS2, menor2, Maior2 ← PMS-3.0 (L, meio+1, fim)

    MS3 ← Maior2 - menor1

    MS ← Max {MS1, MS2, MS3}

    menor ← Min {menor1, menor2}

    Maior ← Max {Maior1, Maior2}

    Retorna (MS, menor, Maior)
```

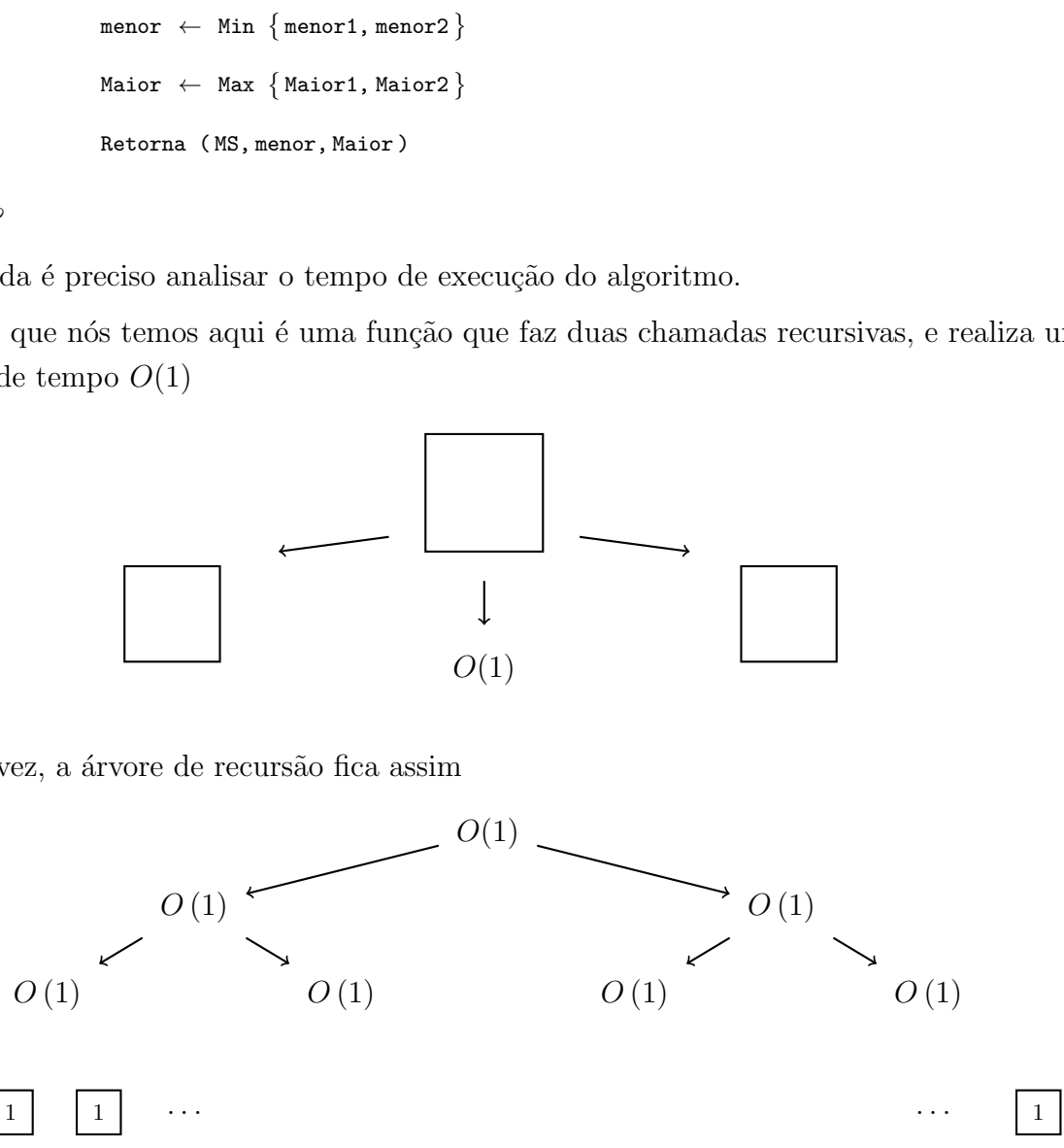
Legal, não é?

Sim, mas ainda é preciso analisar o tempo de execução do algoritmo.

Quer dizer, o que nós temos aqui é uma função que faz duas chamadas recursivas, e realiza uma computação de tempo $O(1)$



Então dessa vez, a árvore de recursão fica assim



Daí fazendo a soma de baixo para cima, nós obtemos

$$n + \frac{n}{2} + \frac{n}{4} + \ldots + 4 + 2 + 1$$

E todo mundo sabe que isso é $O(n)$ — (*ou será que não?*)

Então agora, nós temos um algoritmo de tempo $O(n)$ para o problema do maior salto.

Não é legal?