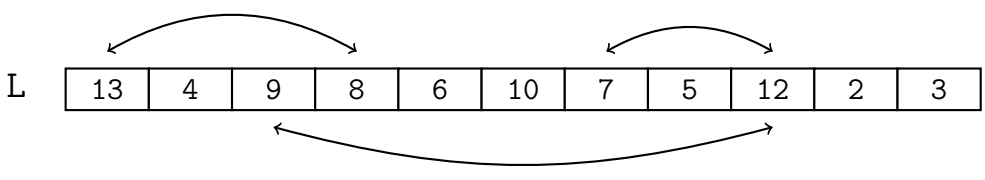
Imagine que L é uma lista de tamanho n que contém números inteiros

Daí, nós podemos localizar pares de elementos que estão fora de ordem



— (*é isso o que nós chamamos de inversão*)

E daí a gente define o problema assim

→ *Contar o número total de inversões na lista*

Certo.

É bem fácil escrever um algoritmo porcaria para esse problema.

Mas hoje nós vamos fazer uma coisa diferente.

Quer dizer, é bem fácil resolver esse problema quando $n = 2$

— (*porque basta comparar os dois elementos para verificar se eles formam uma inversão*)

E é ainda mais fácil resolver o problema quando $n = 1$

— (*porque nesse caso não há inversão nenhuma*)

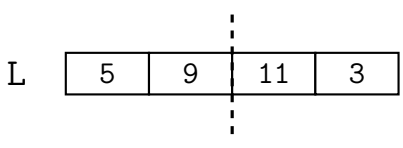
Então hoje nós vamos começar escrevendo o algoritmo que resolve o problema nesses dois casos

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

     Se ( inicio = fim )     Retorna ( 0 )

     Senão                                                    // fim = inicio + 1

          Se ( L[inicio]  >  L[fim] )     Retorna ( 1 )

          Senão                               Retorna ( 0 )
```

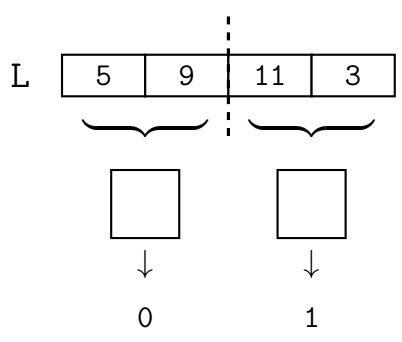
Legal.

Agora a gente imagina que a lista tem tamanho 4

Daí, é natural pensar em dividir o problema em dois



E chamar o nosso algoritmo base em cada metade

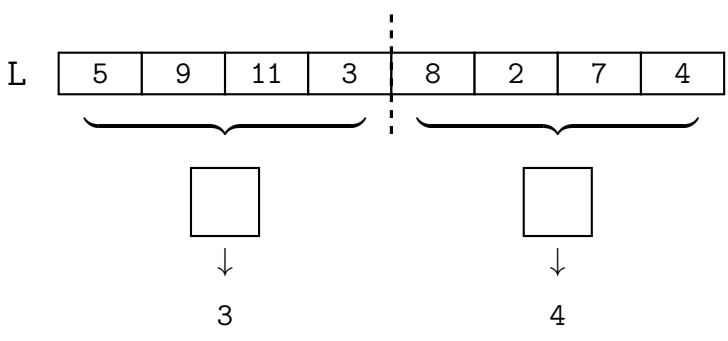


Só que, isso ainda não resolve o problema.

Quer dizer, ainda é preciso contar as inversões envolvendo elementos em metades diferentes.

Nesse caso em que a gente tem apenas 4 elementos a coisa não é difícil.

Mas imagine agora que a lista tem tamanho 8

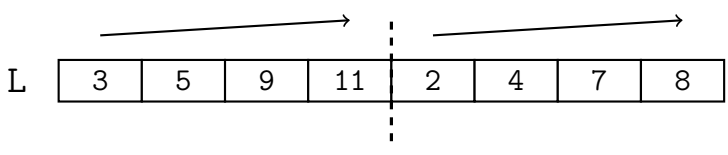


Quer dizer, comparar cada elemento da esquerda com cada elemento da direita leva tempo $O(n^2)$.

E isso vai nos dar um algoritmo porcaria.

Então, a gente precisa ser um pouco mais esperto.

E a esperteza nesse caso é ordenar as duas metades



E agora, a gente conta as inversões assim

```
func  CI-ord ( L, inicio, meio, fim )

     cont ← 0

     i ← inicio

     j ← meio + 1

     Enquanto ( i ≤ meio  E  j ≤ fim )

          Se (L[i]  >  L[j])     Retorna (1)

               cont ← cont  +  (meio - i + 1)

               j ++

          Senão

               i ++

     Retorna ( cont )
```

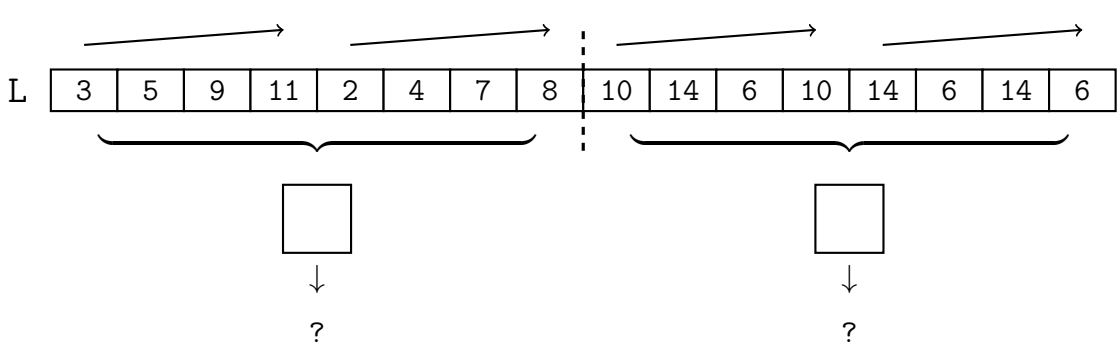
A boa notícia é que essa função executa em tempo $O(n)$.

Mas ainda é preciso considerar o tempo da ordenação.

Em todo caso, isso já nos dá um algoritmo mais rápido que $O(n^2)$.

Só que, a gente ainda pode ser um pouco mais esperto.

Quer dizer, imagine agora que a lista tem tamanho 16



Sim! depois que a gente resolve os problemas de tamanho 8 de cada lado, as suas metades já estão ordenadas.

Então agora, basta a gente intercalar — (*em tempo* $O(n)$)

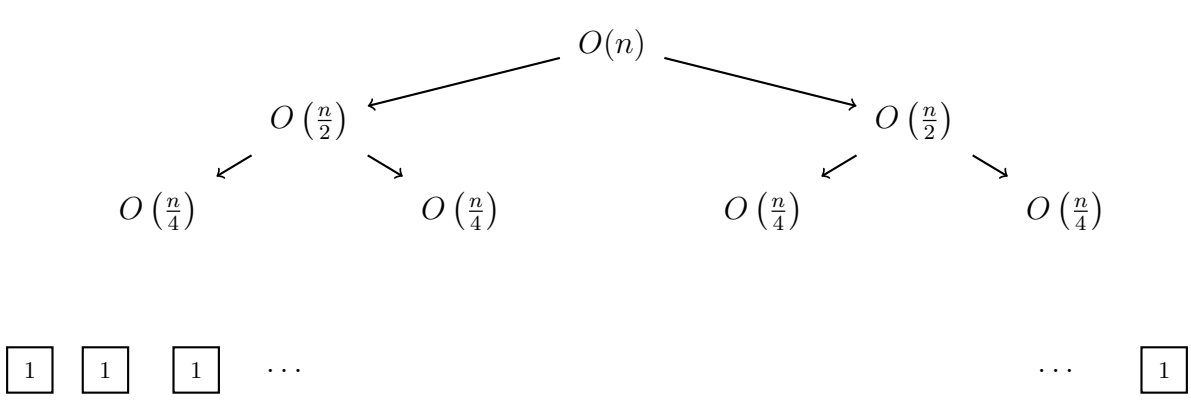
De fato, a primeira ordenação pode acontecer no algoritmo de base

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

     Se ( inicio = fim )     Retorna ( 0 )

     Senão                                                    // fim = inicio + 1

     Se (L[inicio]  >  L[fim] )

          Troca (L, inicio, fim);     Retorna ( 1 )

     Senão                                Retorna ( 0 )
```

E nos outros casos, só o que a gente faz é intercalação

```
func  PCI-main ( L, inicio, fim )

     Se ( fim  ≤  inicio + 1 )

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

     meio ← ( inicio  +  fim )

     C1 ← PCI-main (L, inicio, meio)

     C2 ← PCI-main (L, meio+1, fim)

     C3 ← PCI-ord (L,inicio, meio, fim)

     Intercala (L, inicio, meio, fim)

     Retorna ( C1 + C2 + C3 )
```

Agora que não há mais ordenação no meio do caminho, a árvore de recursão fica assim



E a gente já sabe que isso dá $O(n \log n)$.