Construção e Análise de Algoritmos

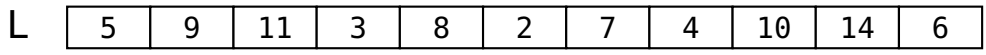
aula 07: Raciocinando a partir de baixo

1 Introdução

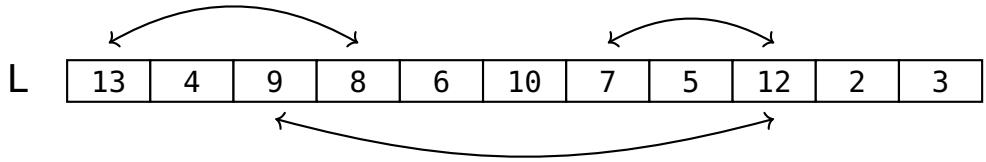
(. . .)

2 Contando inversões

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

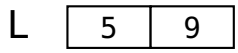
→ *Contar o número total de inversões na lista*

Certo.

É bem fácil escrever um algoritmo porcaria para esse problema.

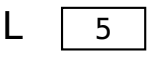
Mas hoje nós vamos fazer uma coisa diferente.

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 $n = 1$



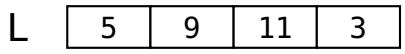
— (*porque nesse caso não há inversão nenhuma*)

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

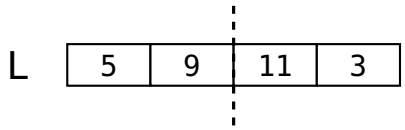
```
func  PCI-base ( L,inicio,fim )
    Se (inicio = fim)    Retorna (0)
    Senão                                              // fim = inicio + 1
        Se (L[inicio] > L[fim])    Retorna (1)
        Senão                      Retorna (0)
```

Legal.

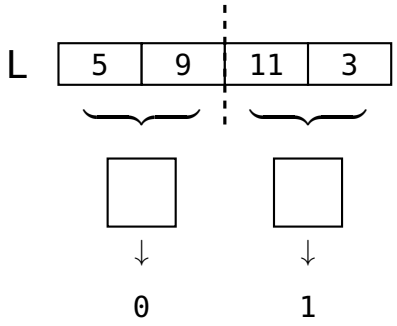
Agora a gente imagina que a lista tem tamanho 4



Daí, é natural pensar em dividir o problema em dois



E chamar o nosso algoritmo base em cada metade

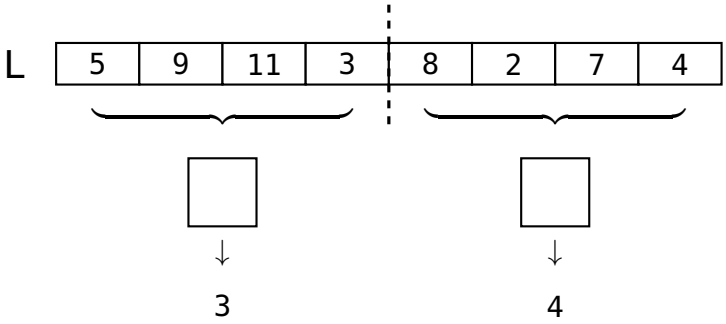


Só que, isso ainda não resolve o problema.

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

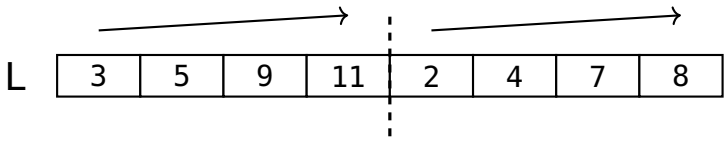


Quer dizer, comparar cada elemento da esquerda com cada elemento da direita leva tempo $O(n^2)$.

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)
```

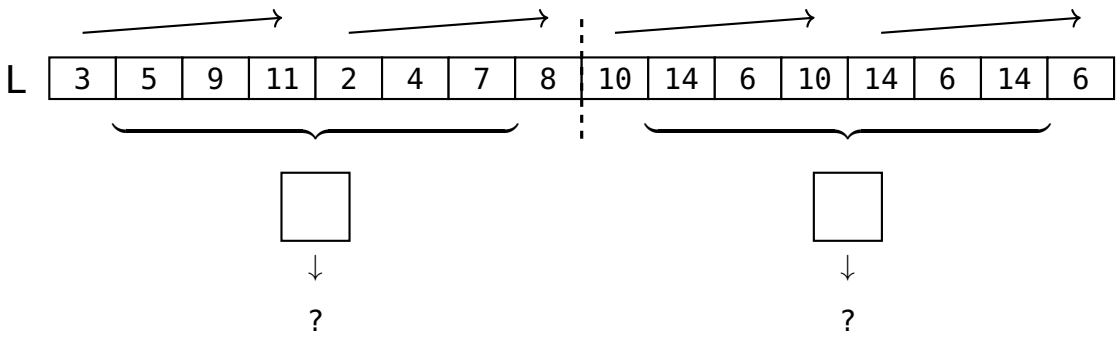
A boa notícia é que essa função executa em tempo $O(n)$.

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* $O(n)$)

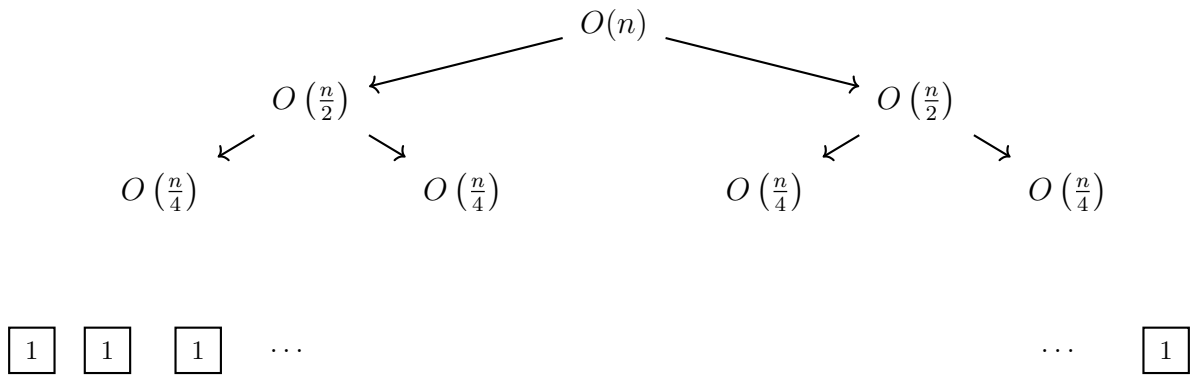
De fato, a primeira ordenação pode acontecer no algoritmo de base

```
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)
```

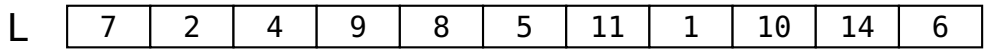
Agora que não há mais ordenação no meio do caminho, a árvore de recursão fica assim



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

3 Revisitando o problema do maior salto

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*

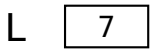
Na aula passada, nós vimos um algoritmo de tempo $O(n \log n)$ para esse problema.

E agora, a gente vai obter um algoritmo melhor.

Como?

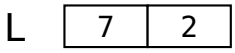
Ora, aplicando a ideia da aula de hoje — (*raciocinando a partir de baixo*)

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 $n = 2$



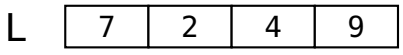
— (*porque basta verificar se o salto de um elemento para o outro é maior que 0*)

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

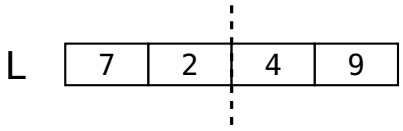
```
func  PMS-base ( L,inicio,fim )
    Se (inicio = fim)    Retorna (0)
    Senão                                          // fim = inicio + 1
        salto ← L[fim]  -  L[inicio]
        Se (salto > 0)    Retorna (salto)
        Senão             Retorna (0)
```

Legal.

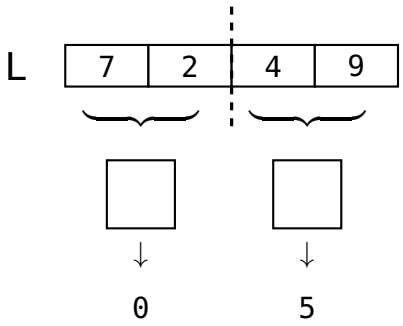
Agora a gente imagina que o problema tem tamanho 4



E daí, a gente quebra o problema na metade

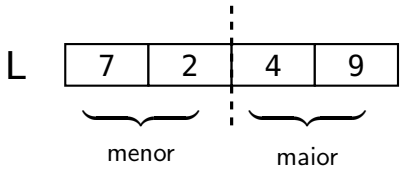


A ideia é chamar o algoritmo base em cada metade



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



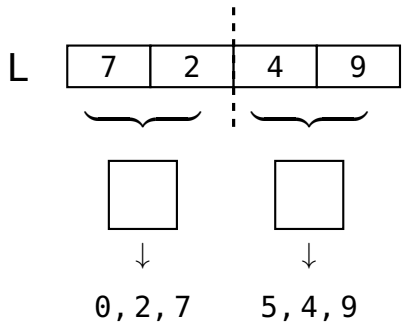
— (*para encontrar o maior salto que cruza o meio*)

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.

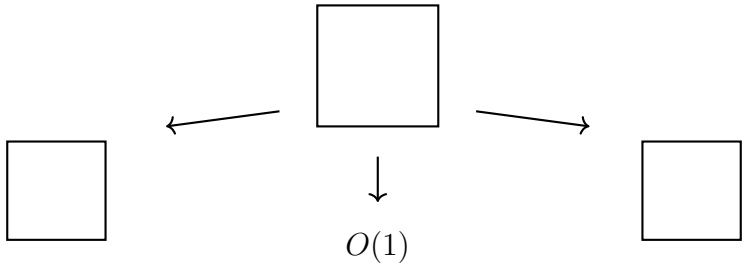
A coisa fica assim

```
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 )
```

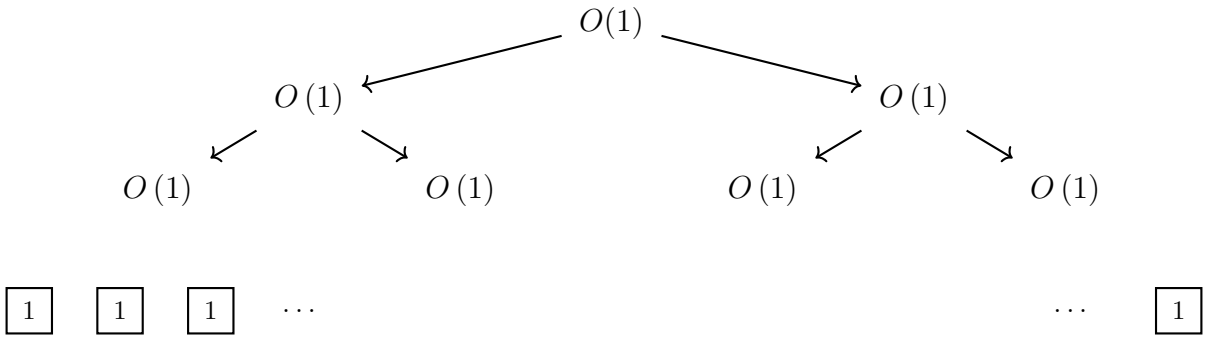
Legal, não é?

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

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

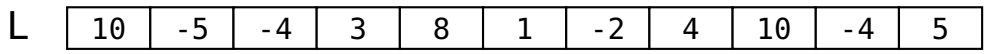
E todo mundo sabe que isso é $O(n)$ — (*ou será que não?*)

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

Não é legal?

4 Faixa máxima de comprimento par

Imagine que L é uma lista de tamanho n que contém números positivos e negativos



Aqui, nós vamos revisitar o problema da faixa com soma máxima.

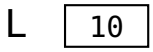
Só que dessa vez, nós adicionamos uma restrição

→ *Encontrar uma faixa de elementos consecutivos com soma máxima*
cujo comprimento é par

E agora?

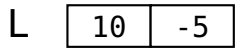
Bom, agora nós vamos raciocinar a partir de baixo.

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



— (*porque a única faixa de comprimento par é a faixa vazia*)

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



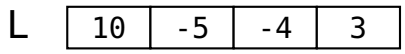
— (*porque basta verificar se a faixa de tamanho 2 tem soma maior que 0*)

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

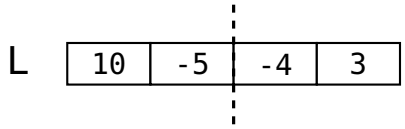
```
func PFSMCP-base ( L, inicio, fim )
    Se (inicio = fim)      Retorna (0)
    Senão                                          // fim = inicio + 1
        soma ← L[inicio] + L[fim]
        Se (soma > 0)      Retorna (soma)
        Senão              Retorna (0)
```

Legal.

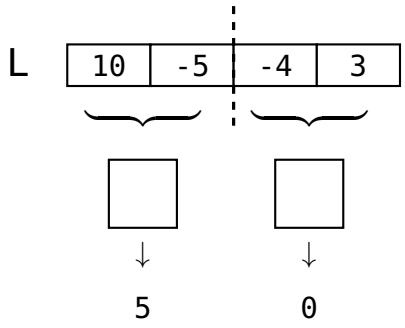
Agora a gente imagina que o problema tem tamanho 4



E daí, a gente quebra o problema na metade

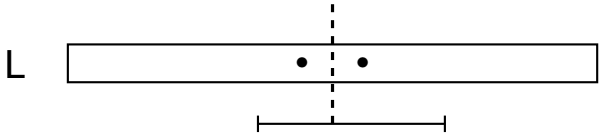


A ideia é chamar o algoritmo base em cada metade



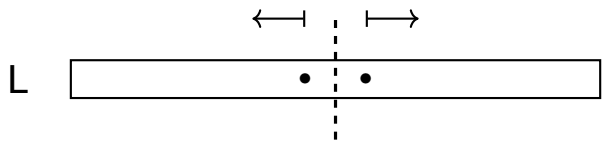
Só que isso ainda não resolve o problema.

Porque pode haver uma solução melhor que contém elementos das duas metades



Nesse ponto a gente lembra que a solução precisa ter comprimento par.

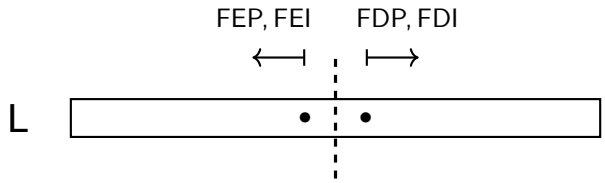
Então de cada lado a gente vai encontrar a faixa de comprimento par que tem a maior soma



Mas, essa não é a única maneira de construir uma faixa de comprimento par.

Quer dizer, a gente também pode ter uma faixa de comprimento ímpar de cada lado.

Então a ideia é percorrer nas duas direções, para encontrar faixas de comprimento par e ímpar com a soma máxima



A coisa fica assim

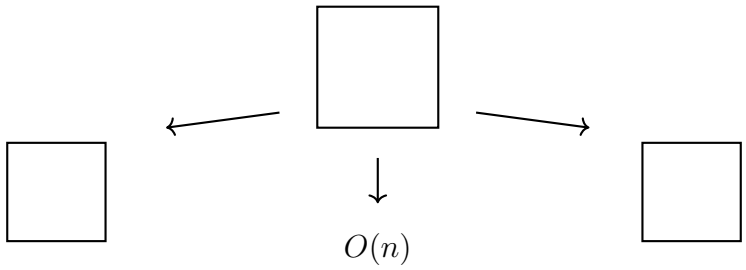
```
func faixa-Meio (L, inicio, meio, fim)
    FEP ← 0;     FEI ← 0
    soma ← 0;    comp ← 0
    Para i ← meio  Até  inicio
        soma ← soma + L[i];     comp++
        Se (comp é par  E  soma > FEP)
            FEP ← soma
        Se (comp é ímpar  E  soma > FEI)
            FEI ← soma
    //
    // a mesma coisa para o lado direito ...
    //
    MF ← Max { FEP+FDP, FEI+FDI }
    Retorna (MF)
```

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

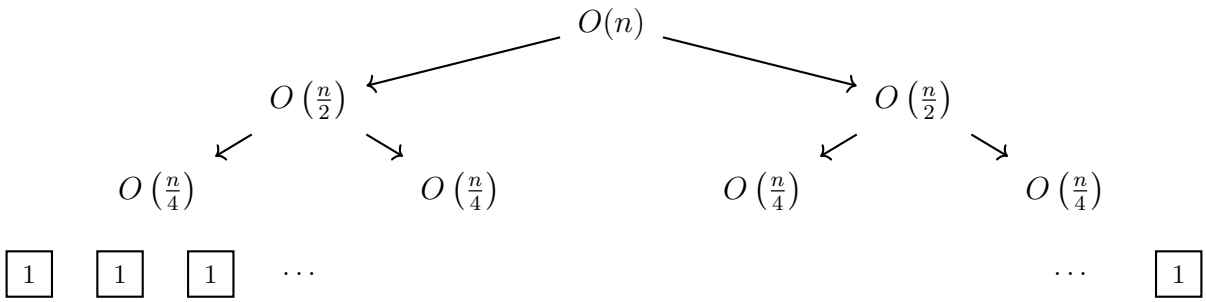
E agora, nós podemos escrever a função que resolve o problema

```
func PFSMCP (L,inicio,fim)
    Se ( fim = inicio )
        Retorna ( PFSMCP-base (L,inicio,fim) )
    meio ← ( inicio + fim ) / 2
    MF1 ← PFSMCP (L,inicio,meio)
    MF2 ← PFSMCP (L,meio+1,fim)
    MF3 ← faixa-Meio (L,inicio,meio,fim)
    MF ← Max {MF1,MF2,MF3)
```

Essa função faz duas chamadas recursivas, e realiza uma computação de tempo $O(n)$



Então dessa vez, a árvore de recursão fica assim



E nós já sabemos que isso dá tempo de execução $O(n)$.