Construção e Análise de Algoritmos

aula 06: A estratégia da divisão

1 Introdução

Essa é a primeira aula da parte II da disciplina.

E esse é o momento em que a gente começa a perder a ingenuidade — (*ou a inocência ...*)

Quer dizer, existem basicamente dois tipos de ingenuidade algorítmica.

O primeiro tipo é algo assim: "E se eu encontrar o maior elemento de todos, depois o menor elemento de todos, e daí ..." — (*ignorando o aspecto combinatorial do problema ...*)

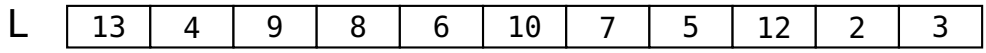
O segundo tipo é algo assim: "Tá bom, então eu vou examinar todas as possibilidades" — (*partindo para a força bruta, o que vai nos dar um algoritmo porcaria ...*)

Infelizmente, o primeiro tipo de ingenuidade não tem muito futuro — (*a gente pode ficar um tempão rodando, sem resolver o problema*)

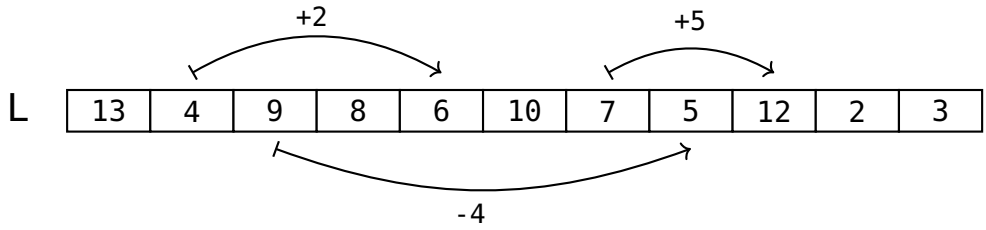
Mas a boa notícia é que o segundo tipo de ingenuidade pode ser um bom começo.

2 A estratégia da divisão

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



Daí, nós calculamos o tamanho do salto de um elemento para outro assim



E daí a gente define o problema assim

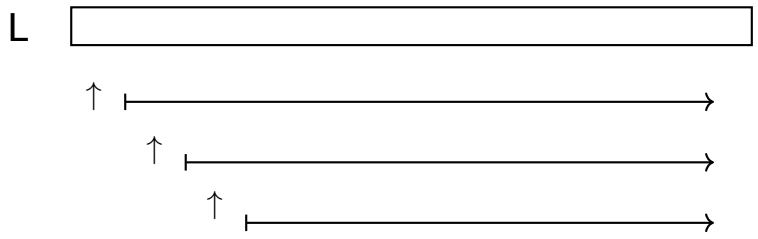
→ *Encontrar o par de elementos que nos dá o maior salto*

Certo.

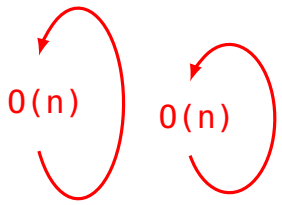
Agora a gente encontra um algoritmo porcaria para esse problema.

Só que, isso é a coisa mais fácil do mundo.

Quer dizer, para cada elemento, a gente vai examinar todos os saltos que ele pode dar pra frente — (*usando dois dedos*)



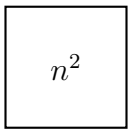
E a coisa fica assim




```
func PMS-1.0 ( L )
    MS ← 0
    Para i ← 1 Até N-1
        Para j ← i Até N
            salto ← L[j] - L[i]
            Se ( salto > MS )
                MS ← salto
    Retorna (MS)
```

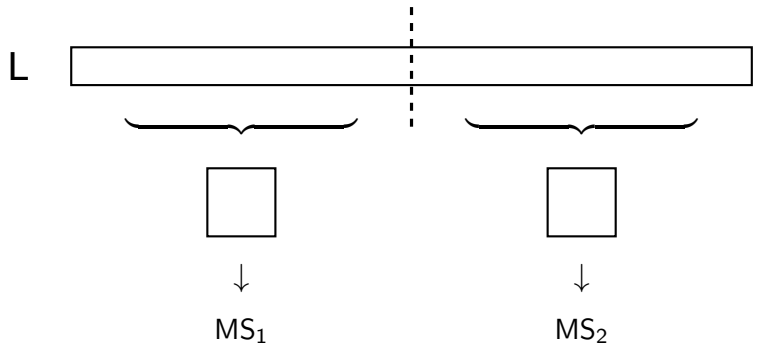
O tempo de execução desse algoritmo é $O(n^2)$ — (*o que é uma porcaria ...*)

Mas, isso é um bom começo



Porque agora, a gente pode aplicar a estratégia da divisão.

Quer dizer, a ideia é aplicar o algoritmo porcaria nas duas metades da lista



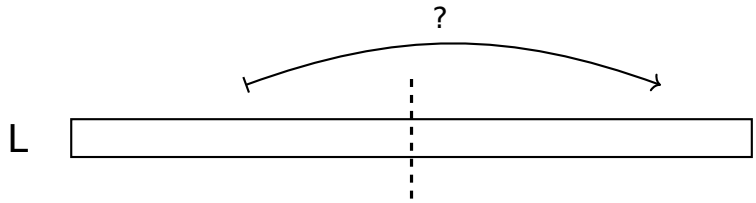
para obter o maior salto em cada metade.

E daí, a gente pega o maior dos dois

$$\mathsf{MS} \quad \leftarrow \quad \mathsf{Max}\left\{\, \mathsf{MS}_1, \mathsf{MS}_2 \,\right\}$$

Só que, isso ainda não é a solução do problema.

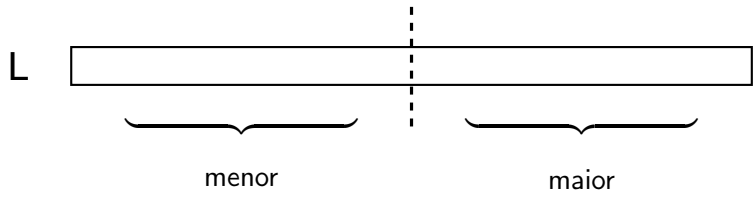
Porque pode haver um salto ainda maior que começa na primeira metade e termina na segunda metade



Então agora a gente precisa calcularo maior salto desse tipo.

Só que isso é bem fácil.

Quer dizer, basta encontrar o menor elemento do lado esquerdo, o maior elemento do lado direito, e depois calcular a sua diferença — (*em tempo* $O(n)$)



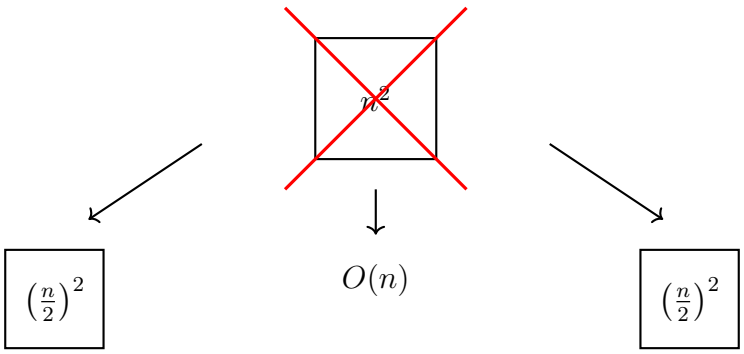
$$\mathsf{MS_3} \quad \leftarrow \quad \mathsf{maior} \; - \; \mathsf{menor}$$

E daí, a gente pega o maior dos 3

$$\mathsf{MS} \quad \leftarrow \quad \mathsf{Max}\left\{ \mathsf{MS_1}, \mathsf{MS_2}, \mathsf{MS_3} \right\}$$

Legal.

Agora, para ver se isso é uma boa ideia, a gente faz as contas



O que dá

$$\frac{n^2}{4} \quad + \quad \frac{n^2}{4} \quad + \quad O(n) \qquad = \qquad \frac{n^2}{2} \quad + \quad O(n)$$

E aqui já dá pra ver que a gente ganhou alguma coisa.

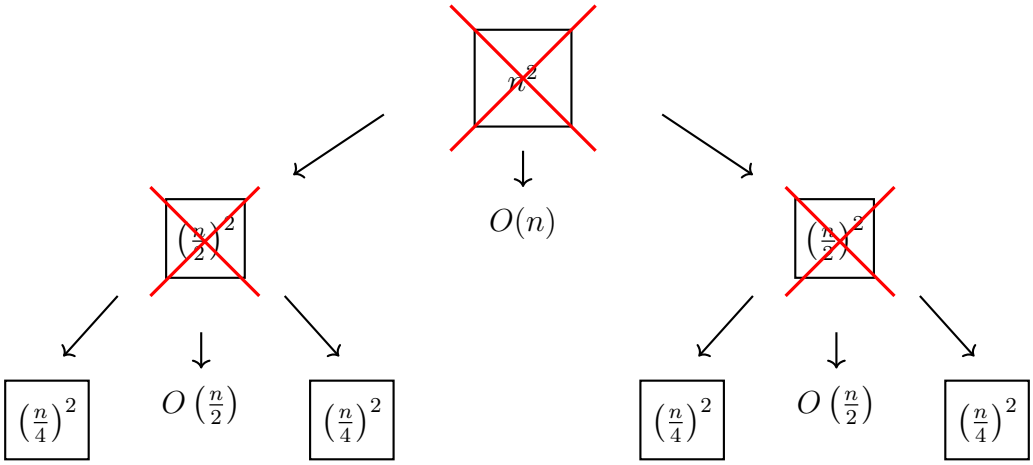
Girando a manivela

Agora, é só girar a manivela.

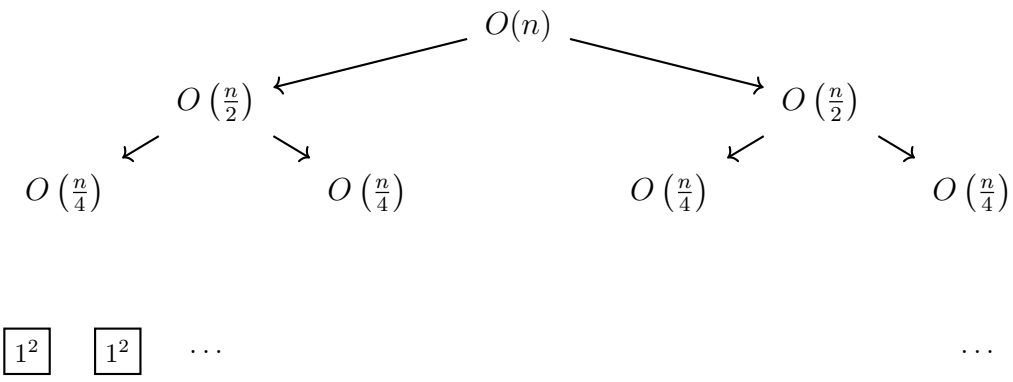
Quer dizer, a gente acabou de ver que vale a pena trocar a caixinha quadrática por duas caixinhas com a metade do tamanho — (*mais* $O(n)$)

Mas quando a gente faz isso, aparecem duas caixinhas quadráticas outra vez.

Então, a gente faz a mesma coisa outra vez



E fazendo isso pra sempre, a gente chega nisso aqui



A esperteza nesse ponto é notar que todos os níveis da árvore tem soma $O(n)$.

E que a árvore tem $O(\log n)$ níveis.

Então agora, nós temos um algoritmo que resolve o problema em tempo $O(n \log n)$.

Legal, não é?

Sim, mas ainda falta escrever o pseudo-código do algoritmo.

Só que isso é bem fácil.

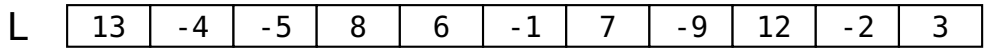
Porque o que nós temos aqui é uma estrutura recursiva — (*uma cebola ...*)

A coisa fica assim

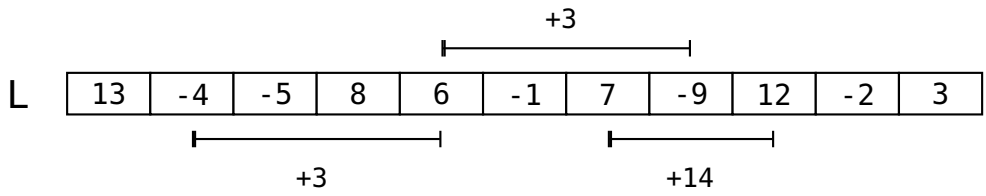
```
func PMS-2.0 ( L, inicio, fim )
    Se ( inicio = fim )
        Retorna ( PMS-1.0 ( L, inicio, fim ) )
    meio ← ( inicio + fim ) / 2
    MS1 ← PMS-2.0 ( L, inicio, meio )
    MS2 ← PMS-2.0 ( L, meio+1, fim )
    menor ← Menor ( L, inicio, meio )
    maior ← Maior ( L, meio+1, fim )
    MS3 ← maior - menor
    MS ← MAX { MS1, MS2, MS3 }
    Retorna ( MS )
```

3 O problema da faixa com a soma máxima

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



Daí, nós calculamos a soma de uma faixa de elementos consecutivos



E daí, a gente define o problema assim

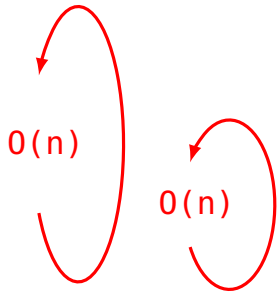
→ *Encontrar a faixa que tem a soma máxima*

Certo.

Agora, a gente considera o algoritmo porcaria para esse problema — (*que calcula a soma de todas as faixas*)



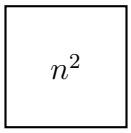
A coisa fica assim




```
func PFSM-1.0 ( L )
    MF ← 0
    Para i ← 1 Até N-1
        soma ← 0
        Para j ← i Até N
            soma ← soma + L[j]
            Se ( soma > MF )
                MF ← soma
    Retorna (MS)
```

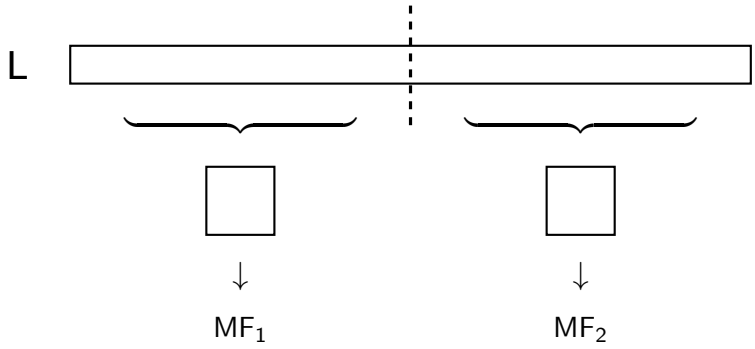
O tempo de execução desse algoritmo é $O(n^2)$ — (*o que é uma porcaria ...*)

Mas isso é um bom começo



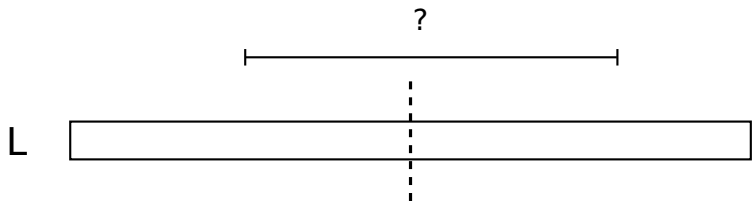
Porque agora, a gente pode aplicar a estratégia da divisão.

Quer dizer, a ideia é aplicar o algoritmo porcaria nas duas metades da lista



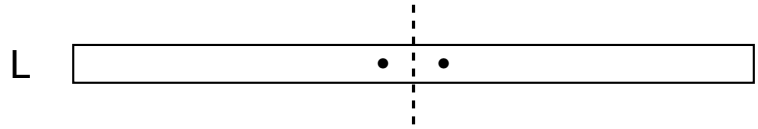
Só que isso ainda não dá a solução do problema.

Porque pode haver uma faixa com soma maior que contém elementos nas duas metades

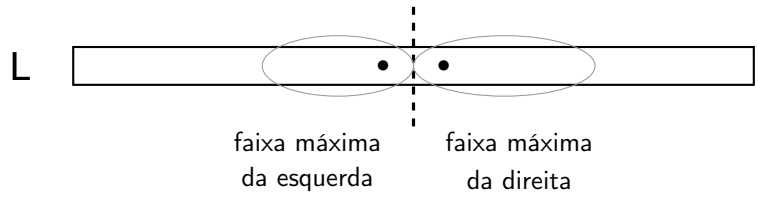


Então agora, a gente precisa encontrar a melhor faixa desse tipo.

Para isso, a gente nota que essa faixa vai conter os elementos que ficam dos dois lados da quebra

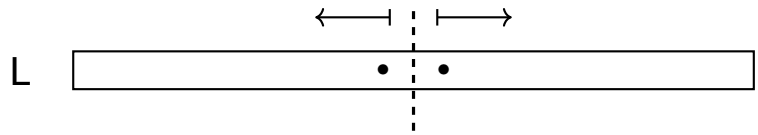


E a ideia é que dos dois lados nós vamos pegar a faixa de soma máxima que contém o elemento



Porque daí, juntando as duas, nós vamos ter a faixa máxima que contém elementos nas duas metades — (*porque?*)

E para obter a faixa máxima de cada lado, basta percorrer a lista — (*do meio para as pontas*)

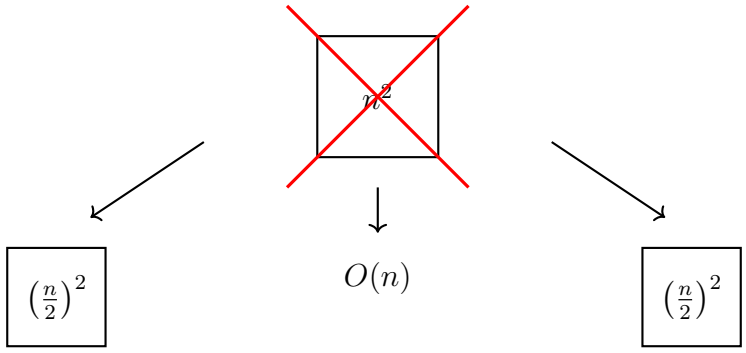


A coisa fica assim

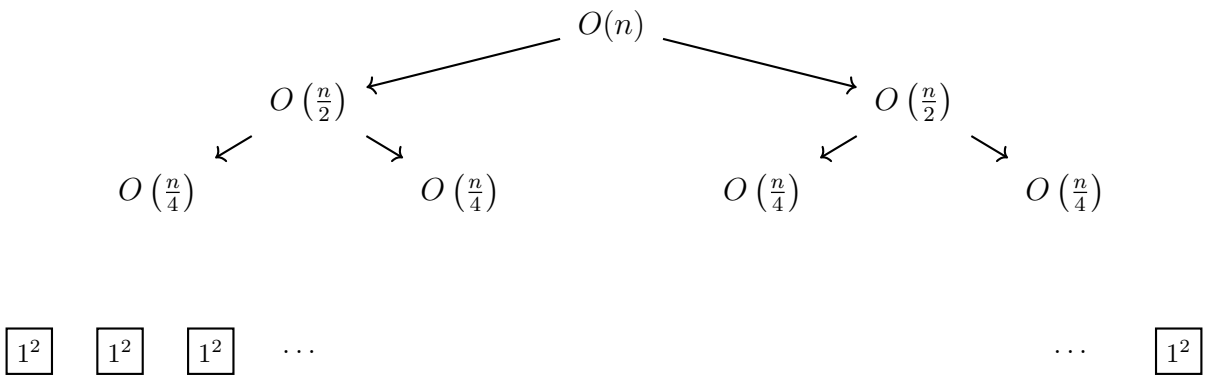
```
func faixaMeio ( L, inicio, meio, fim )
    FE ← 0;   soma ← 0
    Para i ← meio Até inicio
        soma ← soma + L[i]
        Se ( soma > FE )
            FE ← soma
    FD ← 0;   soma ← 0
    Para j ← meio+1 Até fim
        soma ← soma + L[j]
        Se ( soma > FD )
            FD ← soma
    Retorna (FE + FD)
```

A boa notícia é que isso leva tempo $O(n)$.

E mais uma vez, a gente chegou na situação



Daí, girando a manivela a gente chega outra vez nisso aqui



Quer dizer, o nosso algoritmo resolve o problema em tempo $O(n \log n)$.

Só que ainda falta escrever o pseudo-código

```
func PFSM-2.0 ( L, inicio, fim )
    Se ( inicio = fim )
        Retorna (PFSM-1.0 (L,inicio,fim))
    meio ← (inicio + fim) / 2
    FM1 ← PMS-2.0 (L,inicio,meio)
    FM2 ← PMS-2.0 (L,meio+1,fim)
    FM3 ← faixaMeio (L,inicio,meio,fim)
    FM ← MAX {FM1,FM2,FM3}
    Retorna (FM)
```

Legal, não é?

4 O problema ...