Construção e Análise de Algoritmos

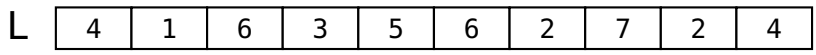
aula 12: Praticando a divisão e conquista

1 Introdução

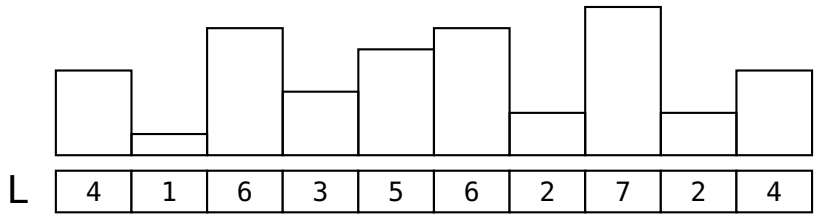
(. . .)

2 O problema do maior retângulo

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



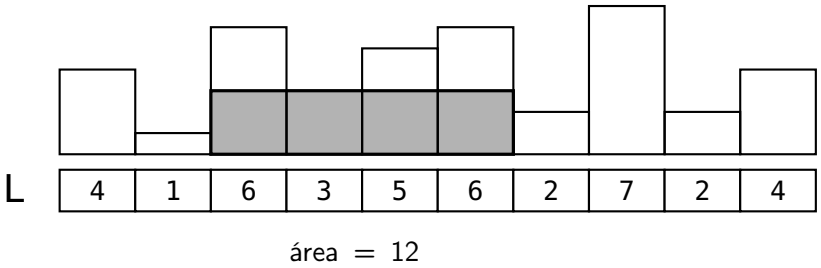
A ideia é que cada número corresponde à altura de uma coluna — (*de largura 1*)



E o problema é

→ *Encontrar um retângulo com a maior área possível*
que fica inteiramente no interior das colunas

No exemplo acima, isso poderia nos dar



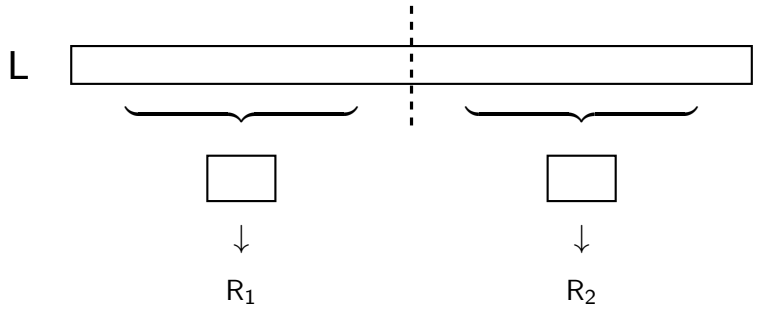
Certo.

É bem fácil resolver esse problema em tempo $O(n^2)$ — (*como?*)

Mas, $O(n^2)$ é uma porcaria ...

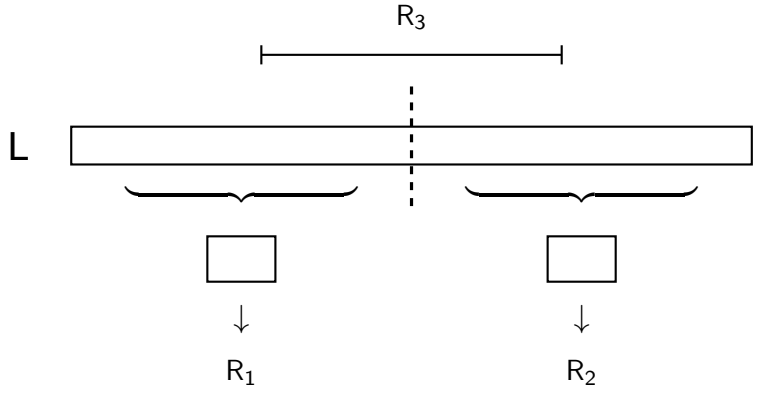
Então, a gente vai tentar algo melhor com a divisão e conquista.

Quer dizer, imagine que a gente quebra a lista no meio, e resolve os problemas de cada lado — (*recursivamente*)



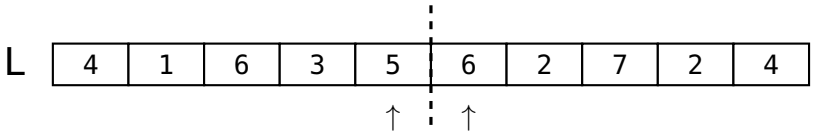
Isso, é claro, ainda não resolve o nosso problema.

Porque ainda é preciso verificar os retângulos que tem uma parte em cada metade

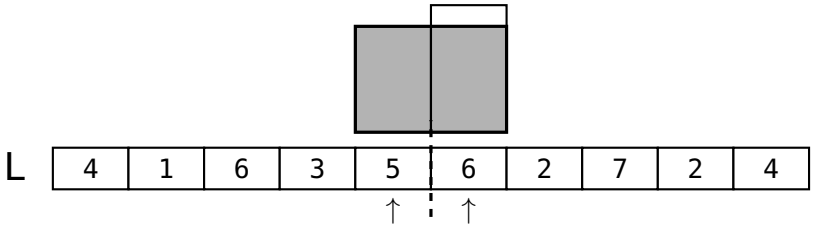


Então agora, a gente se concentra nesse problema.

E para isso, a gente foca a atenção nas duas colunas do meio



Quer dizer, o maior retângulo que cabe nessas colunas é dado pela altura da menor delas



Isso já nos dá uma primeira solução

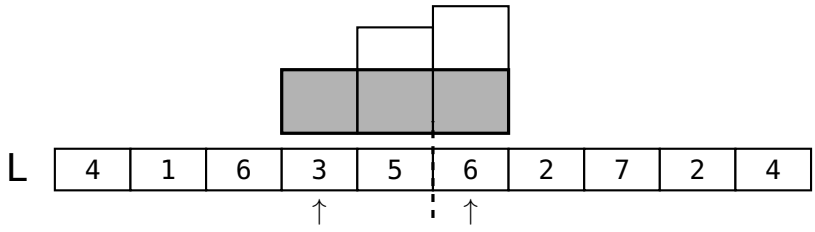
$$\texttt{R} \quad = \quad 2 * 5 \quad = \quad 10$$

E agora é preciso ver se é possível conseguir algo melhor do que isso

Para onde a gente vai?

Ora, a gente vai na direção da maior coluna.

E nesse caso, isso corresponde a ir para o lado esquerdo



Ao fazer isso, a altura diminui um pouquinho.

Mas nós ganhamos alguma coisa na largura.

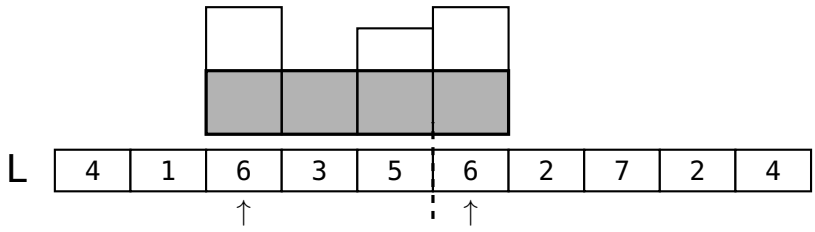
O resultado é

$$\texttt{R} \quad = \quad 3 * 5 \quad = \quad 9$$

o que ainda não melhora a nossa solução.

Então, a gente dá mais um passo.

E o passo é novamente para a esquerda



Dessa vez a altura não diminui — (*e ela não pode nunca aumentar — porque?*)

Mas nós ganhamos alguma coisa na altura.

O resultado é

$$\texttt{R} \quad = \quad 4 * 5 \quad = \quad 12$$

E dessa vez a gente melhorou a nossa solução.

Agora, é só continuar fazendo a mesma coisa.

E ver qual é a melhor solução que a gente consegue obter

A coisa fica assim

```
func retanguloMeio ( L, inicio, meio, fim )
    i ← meio;    j ← meio+1
    MR ← larg * alt
    larg ← 2;    alt ← Min{A[i],A[j]}
    Enquanto ( i ≥ inicio  OU  j ≤ fim )
        larg++
        Se ( j > fim  OU  ( i ≥ inicio  E  L[i] > L[j] ) )
            alt ← Min {alt,L[i]}    i--
        Senão
            alt ← Min {alt,L[j]}    j++
        R ← larg * alt
        Se ( R > MR )    MR ← R
    Retorna ( MR )
```

É bem fácil ver que esse algoritmo executa em tempo $O(n)$.

Agora, nós podemos escrever o nosso algoritmo de divisão e conquista

```
T(n) =  func maiorRetangulo-DC ( L, inicio, fim )
          Se ( inicio = fim )    Retorna ( L[inicio] )
  O(1) |
          meio ← ( inicio + fim ) / 2
T(n/2) ← R1 ← maiorRetangulo-DC ( L, inicio, meio )
T(n/2) ← R2 ← maiorRetangulo-DC ( L, meio+1, fim )
  O(n) ← R3 ← retanguloMeio ( L, inicio, meio, fim )
          Retorna ( Max { R1, R2, R3 } )
```

Análise

Seguindo as anotações, nós obtemos a seguinte equação de recorrência para o tempo de execução do algoritmo

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

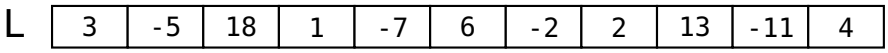
E todo mundo sabe que isso dá

$$T(n) \quad = \quad O(n \log n)$$

— (*ou será que não?*)

3 Contando faixas bacanas

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



Imagine que nós temos dois números MIN, MAX — (*onde o primeiro é menor do que o segundo*)

E imagine que uma faixa de elementos consecutivos é *bacana* se a soma dos seus elementos está entre MIN e MAX.

O problema é

→ *Contar quantas faixas bacanas existem na lista* L

Certo.

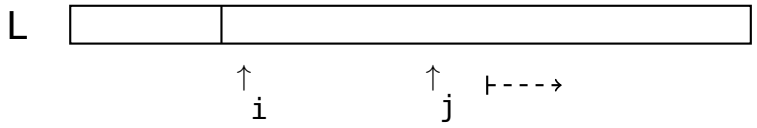
É bem fácil resolver esse problema em tempo $O(n^3)$.

Basta calcular a soma de cada faixa `L[i..j]` em separado — (*e ir contando quantas delas são bacanas*)

Mas, $O(n^3)$ é uma grande porcaria ...

Daí, pensando um pouquinho, a gente vê que dá pra calcular as somas de todas as faixas que começam em `L[i]` em tempo $O(n)$.

Basta manter um dedo parado em `L[i]`, e fazer um dedo `j` andar



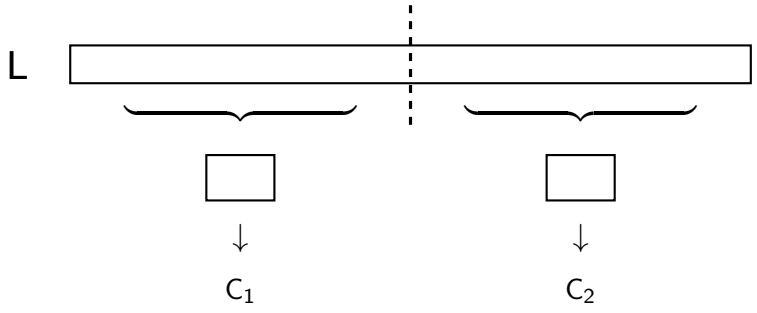
Isso vai dar um algoritmo de tempo $O(n^2)$.

Mas isso ainda é uma porcaria ...

Divisão e conquista

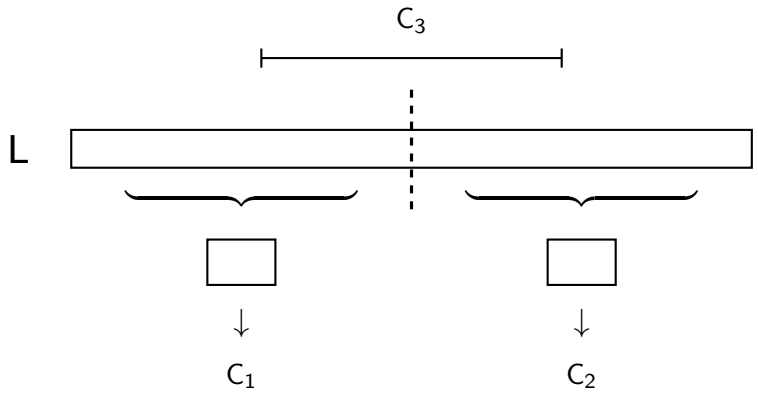
Então agora, está na hora de usar o que a gente aprendeu.

Quer dizer, imagine que a gente quebra a lista no meio, e resolve os problemas de cada lado — (*recursivamente*)



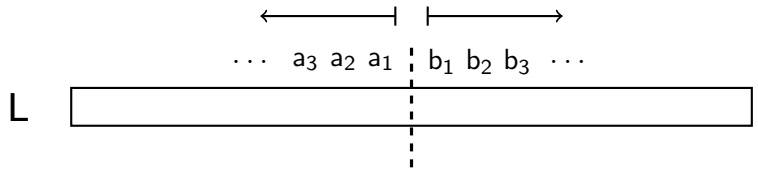
Isso, é claro, ainda não resolve o nosso problema.

Porque ainda é preciso contar as faixas bacanas que possuem elementos nas duas metades

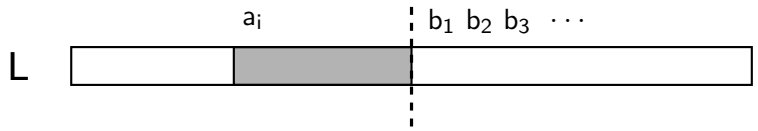


Então agora, a gente se concentra nesse problema.

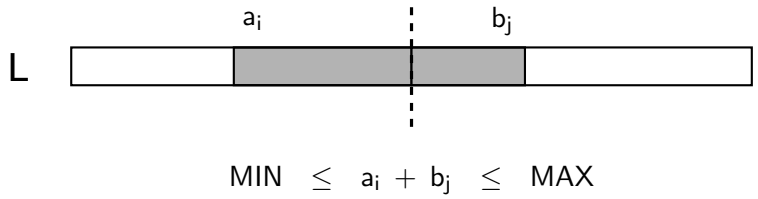
E a ideia natural é pecorrer a lista do meio para as duas pontas —: (*calculando a soma até cada ponto*)



Daí, a gente foca a atenção em um dos pedaços do lado esquerdo



E vai contar quantos pedaços do lado direito completam uma faixa bacana

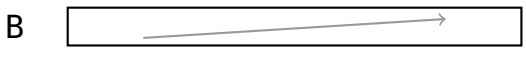


Só que, examinar cada pedaço do lado direito leva tempo $O(n)$.

E fazer isso para cada pedaço do lado esquerdo leva tempo $O(n^2)$ — (*uma porcaria ...*)

Então, a gente precisa ser mais esperto do que isso.

E a esperteza consiste em ordenar os números do lado direito



Agora, a gente consegue determinar quantos b_j's satisfazem

$$\mathsf{MIN} - \mathsf{a_i} \leq \mathsf{b_j} \leq \mathsf{MAX} - \mathsf{a_i}$$

em tempo $O(\log n)$ — (*usando a busca binária*)

E fazer isso para cada pedaço do lado esquerdo leva tempo $O(n \log n)$.

Agora a gente pode escrever a função que faz isso

```
func contaFaixaMeio ( L, inicio, meio, fim )
    k ← 0;     soma ← 0
    Para i ← meio Até inicio
        soma ← soma + L[i]
        k++;     A[K] ← soma
    ℓ ← 0;     soma ← 0
    Para j ← meio+1 Até fim
        soma ← soma + L[j]
        ℓ++;     B[ℓ] ← soma
    ordena ( B )
    C ← 0
    Para i ← 1 Até K                // p/ cada elemento de A
        b1 ← buscaBin ( Min - A[i], B )
        b2 ← buscaBin ( Max - A[i], B )
        C ← b2 − b1
    Retorna ( C )
```

É bem fácil ver que esse algoritmo executa em tempo $O(n \log n)$.

E agora, nós podemos escrever o nosso algoritmo de divisão e conquista

```
T(n) =  func CFB-DC ( L, inicio, fim )
        |  Se (inicio = fim)
O(1)    |      Se (MIN ≤ L[inicio] ≤ MAX)     Retorna ( 1 )
        |      Senão                          Retorna ( 0 )
        |  meio ← ( inicio + fim ) / 2
```

```
T(n/2) ⟵ C1 ← maiorRetangulo-DC ( L, inicio, meio )
T(n/2) ⟵ C2 ← maiorRetangulo-DC ( L, meio+1, fim )
O(n log n) ⟵ C3 ← retanguloMeio ( L, inicio, meio, fim )
           Retorna ( C1 + C2 + C3 )
```

Análise

Seguindo as anotações, nós obtemos a seguinte equação de recorrência para o tempo de execução do algoritmo

$$T(n) \quad = \quad 2 \cdot T\left(\frac{n}{2}\right) \quad + \quad O(n \log n)$$

E agora, é só desenrolar o rocambole

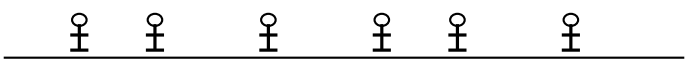
(. . .)

O que nos dá

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

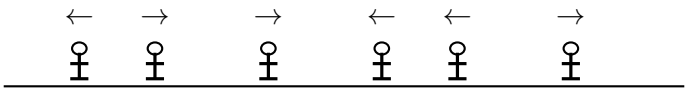
4 Contando encontros

Imagine que existe n pessoas sobre uma linha



E imagine que as posições das pessoas estão armazenadas no vetor `A[1..n]`.

Imagine que as pessoas vão andar para uma outra posição da linha — (*todas com a mesma velocidade*)



E imagine que as posições destino das pessoas estão armazenadas no vetor `B[1..n]`.

Daí, algumas dessas pessoas vão se encontrar.

E quando elas se encontram, elas se cumprimentam: "*Olá, bom dia!*".

O problema é

→ *Contar a quantidade de encontros que vão acontecer*

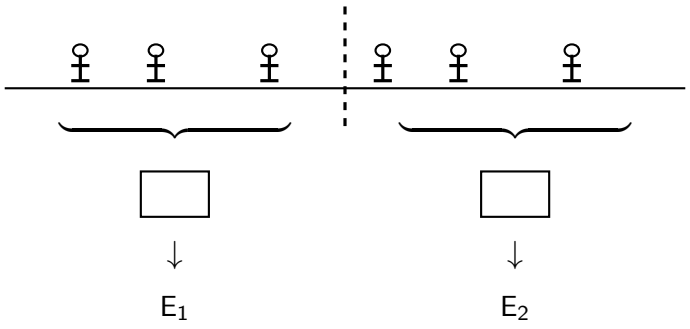
Certo.

É bem fácil resolver esse problema em tempo $O(n^2)$ — (*como?*)

Mas, $O(n^2)$ é uma porcaria ...

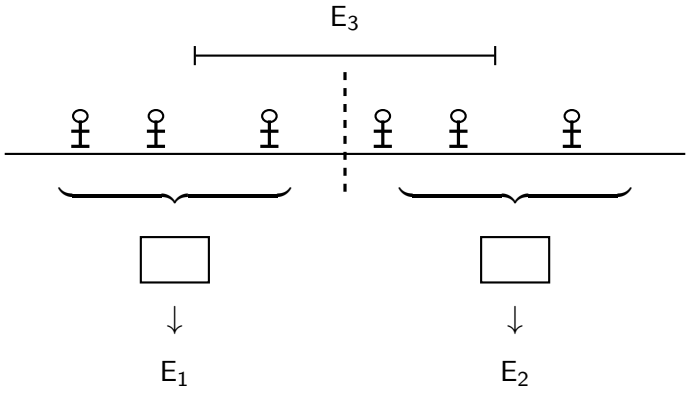
Então a gente vai tentar obter algo melhor com a divisão e conquista.

Quer dizer, imagine que a gente quebra a lista no meio, e resolve os problemas de cada lado — (*recursivamente*)



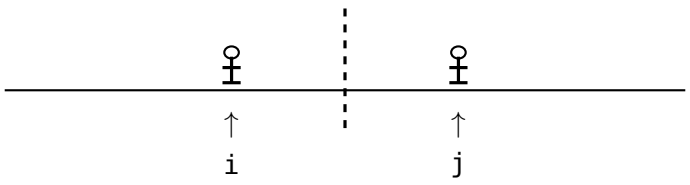
Isso, é claro, não resolve o nosso problema.

Porque ainda é preciso contar os encontros entre pessoas de lados diferentes



Então agora, a gente se concentra nesse problema.

Para isso a gente foca a atenção em uma pessoa de cada lado



E nota que, em qualquer caso, o encontro vai acontecer se

$$\texttt{B[i]} \quad > \quad \texttt{B[j]}$$

Então, para contar quantos encontros a pessoa `i` vai ter, basta contar quantas pessoas `j` do lado direito satisfazem a desigualdade acima.

Para fazer essa contagem de maneira eficiente, a gente ordena o lado direito da lista B.

E faz a contagem usando a busca binária.

A coisa fica assim

```
func encontrosMeio ( A, B, inicio, meio, fim )
    B-aux ← ordena ( B,meio+1,fim )
    C ← 0
    Para i ← meio Até inicio
        j ← buscaBin ( B[i], B-aux )
        C ← C + ( j − meio + 1 )
    Retorna ( C )
```

É bem fácil ver que esse algoritmo executa em tempo $O(n \log n)$.

Agora, nós podemos escrever o nosso algoritmo de divisão e conquista

```
T(n) =  func contaEncontros-DC ( A, B, inicio, fim )
          Se ( inicio = fim )    Retorna ( 0 )
O(1)  |
          meio ← ( inicio + fim ) / 2
T(n/2) ←  E1 ← contaEncontros-DC ( A,B,inicio,meio )
T(n/2) ←  E2 ← contaEncontros-DC ( A,B,meio+1,fim )
O(n log n) ←  E3 ← encontrosMeio ( A,B,inicio,meio,fim )
          Retorna ( E1 + E2 + E3 )
```

Análise

Seguindo as anotações, nós obtemos a seguinte equação de recorrência para o tempo de execução do algoritmo

$$T(n) \quad = \quad 2 \cdot T\left(\frac{n}{2}\right) \quad + \quad O(n \log n)$$

O que nos dá outra vez

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