Divisão e Conquista

SCC 211 - Lab. Algoritmos Avanc. II

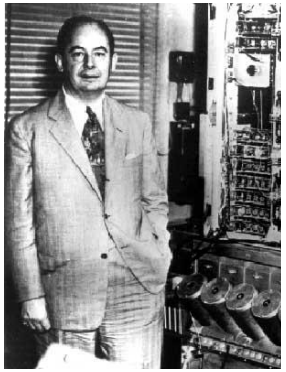
Esta aula não é sobre:

- Quicksort
- Mergesort
- Busca binária (o algoritmo de busca em um vetor ordenado)

- Aqui veremos estratégias interessantes para resolver problemas baseado no paradigma de divisão e conquista:
  - Sim… vamos repartir o problema em “pedaços” menores de forma a reduzir a complexidade e, no final, obter o resultado desejado
  - Nossa esperança: o que era $O(n^2)$, agora passa a ser $O(n \log n)$!!!

Mas caso vc se esqueceu…!

- Mergesort:
  - divida o vetor em duas metades
  - recursivamente ordene cada metade
  - mescle ambas metades de forma a obter o resultado!



Jon von Neumann (1945)

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R

I	T	H	M	S

divide O(n)

A	G	L	O	R

H	I	M	S	T

sort 2T(n/2)

A	G	H	I	L	M	O	R	S	T

merge O(n)

Complexidade - Divisão e Conquista

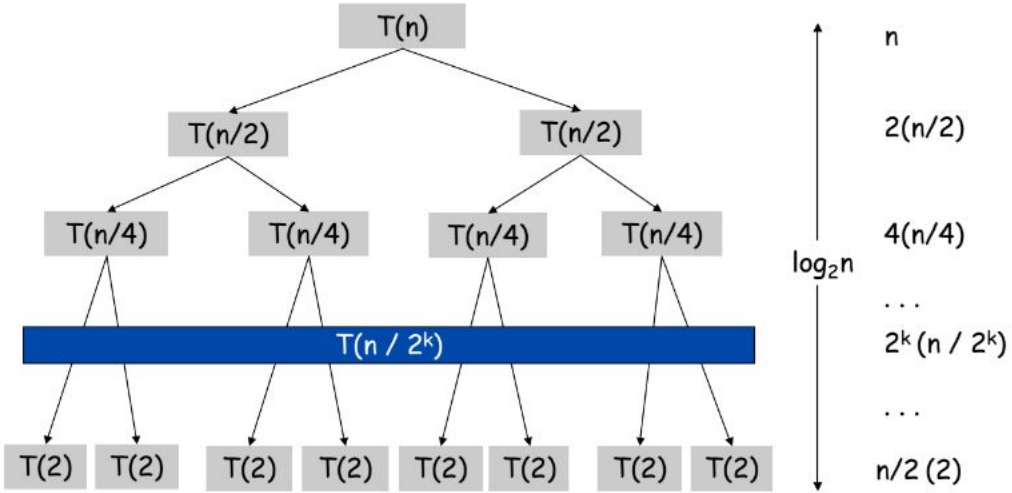
- Seja *T(n)* o tempo de execução do pior caso para uma entrada *n*.
- Se n é par, a estratégia é de ordem *O(n)* para separar os valores em 2 metades
- Leva T(n/2) para resolver cada metade (pior caso, claro…)
- Finalmente, *O(n)* para combinar as soluções provenientes das 2 **chamadas recursivas**.
- O tempo de execução T(n) satisfaz a seguinte relação de recorrência:

>>>> $T(n) \leq 2T(n/2) + cn.$ (para n > 2)

>>>> T(2) ≤ c informalmente, fazemos: $T(n) \leq 2T(n/2) + O(n)$

Prova por Árvore de Recursão

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Complexidade: Divisão e Conquista

- nível 0: problema de tamanho n: c(n) + recursões...
- nível 1: 2 problemas tam = cn/2: 2 cn/2 = cn + recursões subsequentes
- nível 2: 4 problemas tam = n/4: 4 cn/4 = cn + recursões….
- qual o padrão?
  - No nível j da recursão,o nro de subproblemas dobra j vezes >>> 2^j
  - O tamanho de cada subproblema encolhe por um fator de 2^j . Tam subprob = n / 2^j
  - O nivel j contribui com um tempo de $2^j(cn/ 2^j) = cn$
- Somando todos os níveis da recursão
  - vimos que cn se repete em todos os níveis.
  - O nro de vezes que a entrada deve dividida ao meio de n **até** 2 é $\log_2 n$
- Somando-se cn para todos os log n níveis de recursão dá: *O(n log n)*

Contando inversões

- Análise de *ranking* é algo muito usado hoje em dia (web). Vários sites usam o que se chama de filtragem colaborativa para tentar casar (dar *matching*) suas preferências.
  - livros, filmes, músicas, etc..
  - feito isso, vc recebe uma série de dicas sobre o que consumir.

Contando inversões

- Basicamente, o que procuramos aqui é comparar 2 *rankings*.
  - Eu elenco/"ranqueio" ***n*** filmes de que gosto rotulando-os entre 1, … , n
    - **1** é o de que mais gosto e **n** o de que menos gosto.
  - Agora ordene estes rótulos de acordo com a preferência do outro
  - Basta contar agora quantos pares estão "fora de ordem".

Contando inversões

A métrica de similaridade: o número de inversões entre 2 rankings.

- Meu rank: 1, 2, …, n.
- Outro rank: $a_1, a_2, \ldots, a_n$.
- Filmes i e j invertidos se i < j, mas $a_i > a_j$.

Filmes

	A	B	C	D	E
Eu	1	2	3	4	5
Outro	1	3	4	2	5

Inversões

3-2, 4-2

Contando inversões

- Um algoritmo força bruta para isso seria como
- E qual a complexidade?

Contando inversões

- Um algoritmo força bruta para isso teria que complexidade?
- Força Bruta: verifique todos $\Theta(n^2)$ pares i, j.

Contando inversões: Dividir-e-Conquistar

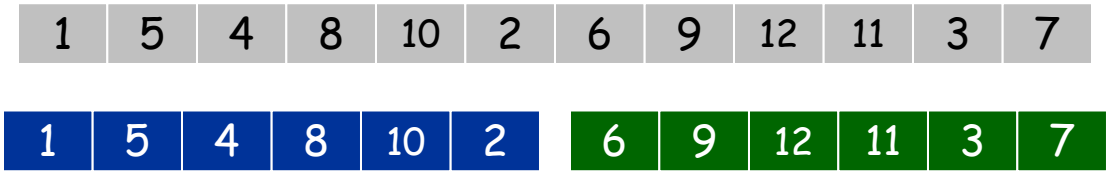
Suponha o vetor:

1	5	4	8	10	2	6	9	12	11	3	7

Contando inversões: Dividir-e-Conquistar

Suponha o vetor:

- Dividir: separar a lista em duas metades.



Dividir: $O(n)$.

Contando inversões: Dividir-e-Conquistar

Suponha o vetor:

- Dividir: separar a lista em duas metades.
- Conquistar: recursivamente, contar as inversões em cada metade.

Dividir: $O(1)$.

Conquistar: $2T(n / 2)$

5 inversões no azul

5-4, 5-2, 4-2, 8-2, 10-2

8 inversões no verde

6-3, 9-3, 9-7, 12-11, 12-3, 12-7, 11-3, 11-7

Contando inversões: Dividir-e-Conquistar

Suponha o vetor:

- Dividir: separar a lista em duas metades.
- Conquistar: recursivamente, contar as inversões em cada metade.
- Combinar: contar inversões para a_i and b_j em metades distintas, e retorne a soma das 3 quantidades.

1	5	4	8	10	2	6	9	12	11	3	7

1	5	4	8	10	2

5 inversões no azul

6	9	12	11	3	7

8 inversões no verde

Divide: $O(1)$.

Conquer: $2T(n / 2)$

9 inversões azul-verde

5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Total = 5 + 8 + 9 = 22.

Combinar: ???

Contando inversões: Combinar

Combinar: contar inversões em ambas metades

- Assuma que cada metade está ordenada.
- a_i e b_j estão em metades distintas.
- Combine as duas metades ordenadas em um único vetor ordenado.

isso preserva invariância quanto à ordenação!

3	7	10	14	18	19

2	11	16	17	23	25
6	3	2	2	0	0

13 inversões azul-verde: 6 + 3 + 2 + 2 + 0 + 0

Contar: O(n)

2	3	7	10	11	14	16	17	18	19	23	25

Mesclar: O(n)

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) \Rightarrow T(n) = O(n \log n)$$

Contando Inversões: Implementação

Pré-condição. [Merge-and-Count] A e B ordenados.

Pós-condição. [Sort-and-Count] L ordenado.

```
Sort-and-Count(L) {
   if list L has one element
      return 0 and the list L

   Divide the list into two halves A and B
   (r_A, A) ← Sort-and-Count(A)
   (r_B, B) ← Sort-and-Count(B)
   (r , L) ← Merge-and-Count(A, B)

   return r = r_A + r_B + r and the sorted list L
}
```

Pares de Pontos mais Próximos

- O Problema. Dados n pontos em um plano, encontre o par com a menor distância Euclidiana entre eles, ou seja, encontre o par mais próximo !

Pares de Pontos mais Próximos

- O Problema. Dados n pontos em um plano, encontre o par com a menor distância Euclidiana entre eles, ou seja, encontre o par mais próximo !

- Aplicações em Geometria Computacional
  - CG
  - Visão Computacional
  - GIS (Geographic Information Systems)
  - Modelagem Molecular

- Qual o algoritmo força bruta e sua complexidade ?

- E qual seria o algoritmo e complexidade se o problema fosse 1D, isto é, todos os pontos estivessem em uma única linha?

Pares de Pontos mais Próximos

- O Problema. Dados n pontos em um plano, encontre o par com a menor distância Euclidiana entre eles, ou seja, encontre o par mais próximo !

- Aplicações em Geometria Computacional
  - CG
  - Visão Computacional
  - GIS (Geographic Information Systems)
  - Modelagem Molecular

- Qual o algoritmo força bruta e sua complexidade?
  - Verifique todos os pares de pontos p e q em $\Theta(n^2)$ comparações.

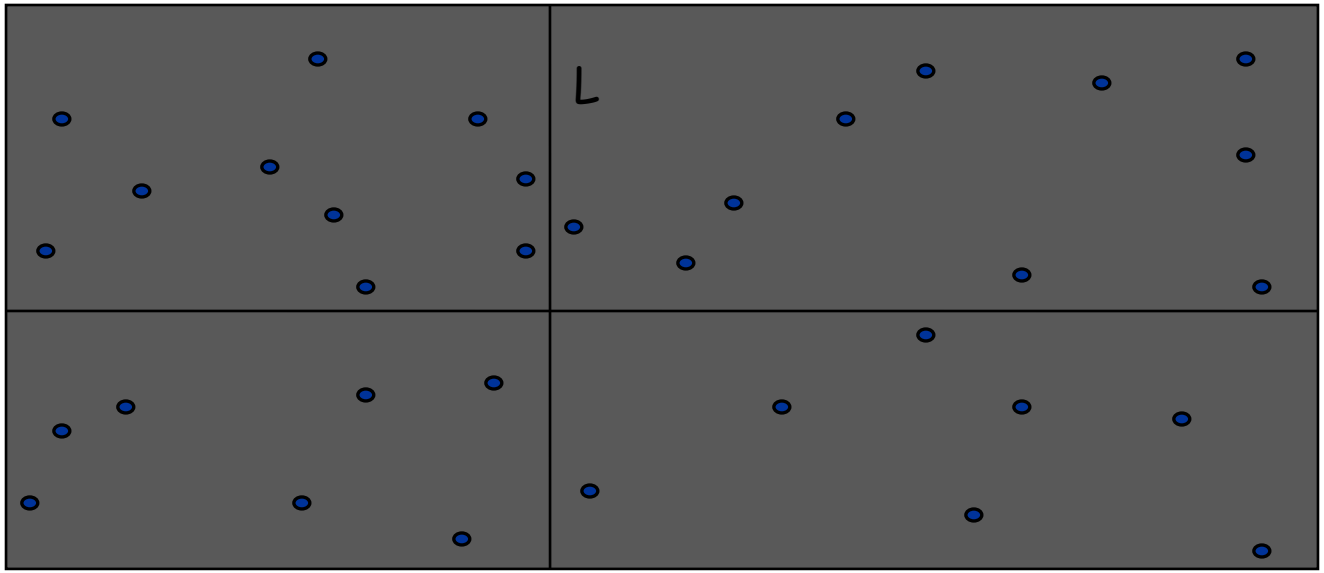
- E qual seria o algoritmo e complexidade de o problema fosse 1D, isto é, todos os pontos estivessem em uma única linha?
  - Ordene o pontos - O(n log n). A seguir, percorra a lista calculando a distância de um ponto com o próximo. Devolva o valor mínimo.

Pares de Pontos mais Próximos

- Notação
  - P é um conjunto de pontos $\{p_1, \ldots, p_n\}$, onde p_i tem coordenada (x_i,y_i)
  - Sejam p_i e p_j pertencentes a P.
    - $d(p_i,p_j)$ é a distância Euclidiana entre eles
  - Assuma que não há 2 pontos com as mesmas coordenadas y ou coordenadas x;
    - Esta restrição pode ser eliminada se aplicarmos uma rotação aos pontos de forma a tornar esta afirmação verdadeira e usar o algoritmo apresentado !

Pares de Pontos mais Próximos: Primeira Tentativa

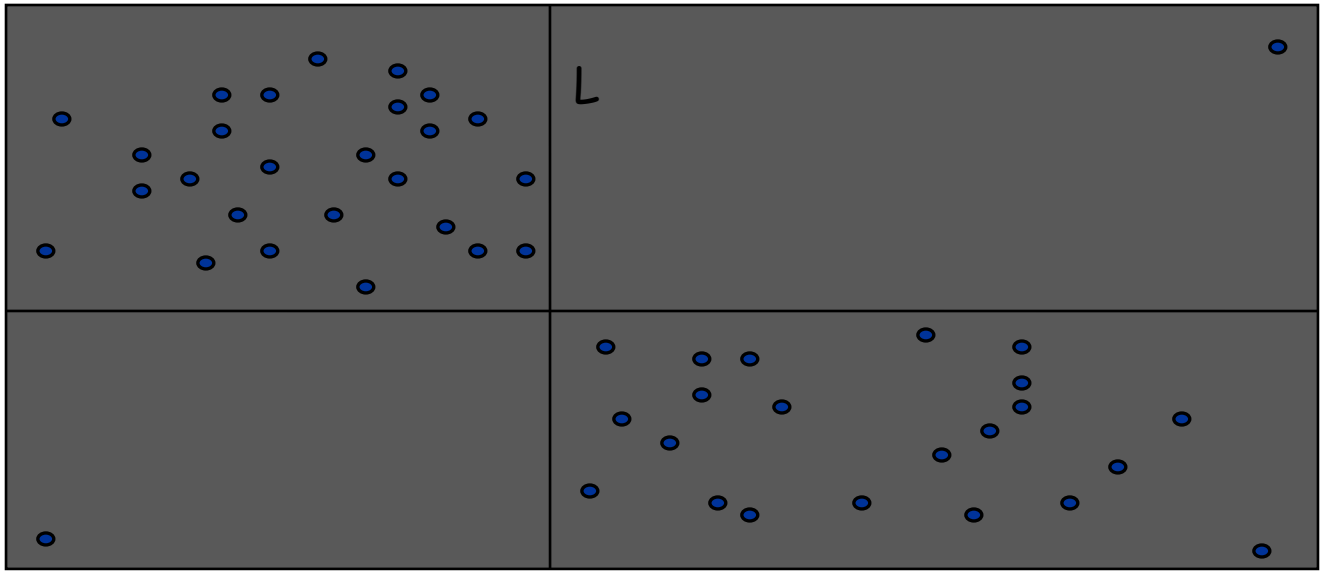
Divisão. Sub-divida a região em 4 quadrantes



Pares de Pontos mais Próximos: Primeira Tentativa

Divisão. Sub-divida a região em 4 quadrantes

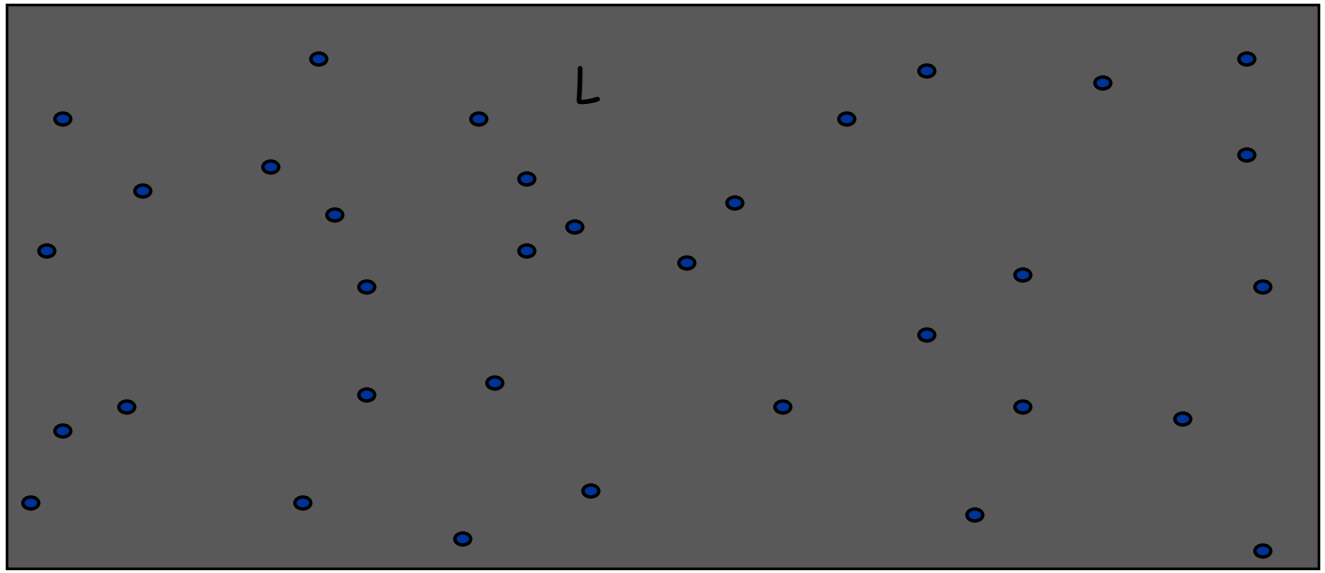
Obstáculo. Impossível garantir n/4 pontos em cada parte.



Pares de Pontos mais Próximos

Algoritmo.

- Antes da recursão: Ordenar os pontos em função do eixo **x**. (Lista P_x). Veremos que será preciso uma lista P_y , ordenada pelo eixo y.
  ….. As listas deverão guardar as posições de cada ponto….

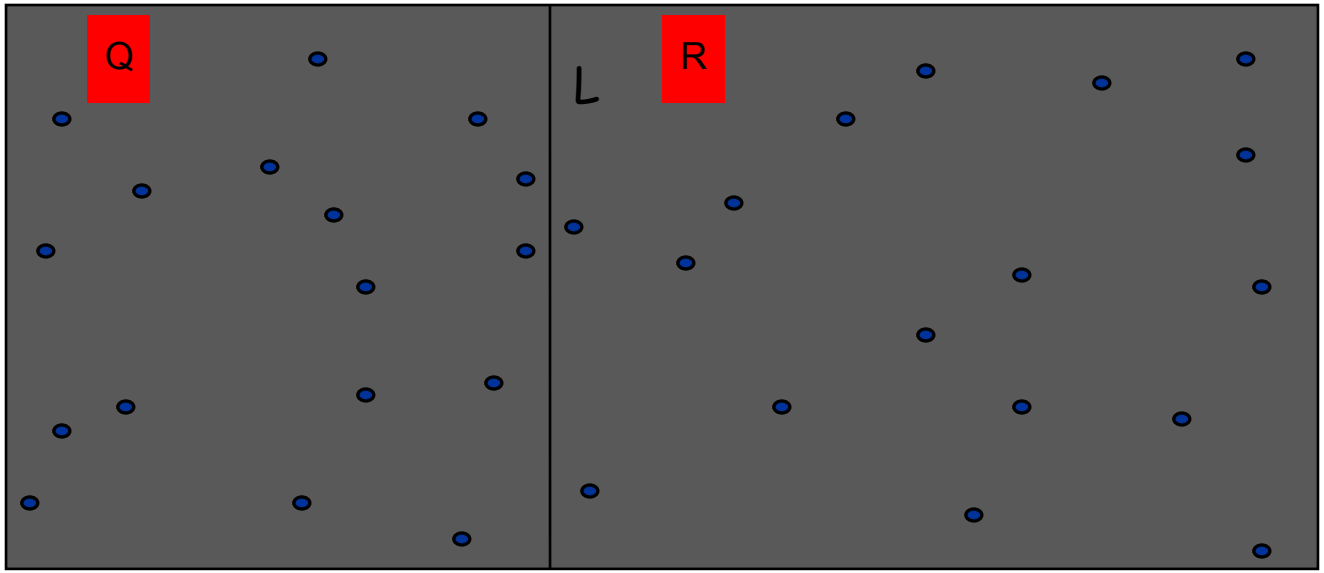


Pares de Pontos mais Próximos

Algoritmo.

- Divisão: desenhe uma linha vertical L de forma que aproximadamente metade dos pontos estejam em cada lado. Utilize P_x para isso.

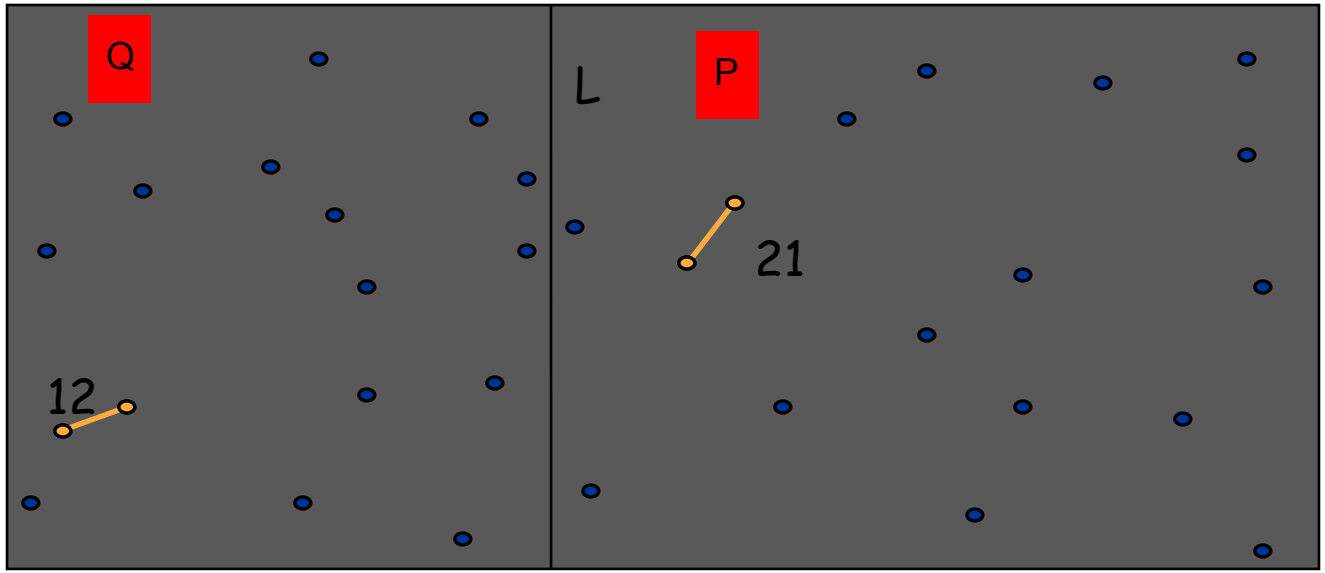
  Crie as listas Q_x, Q_y, R_x e R_z >> O(n)



Pares de Pontos mais Próximos

Algoritmo.

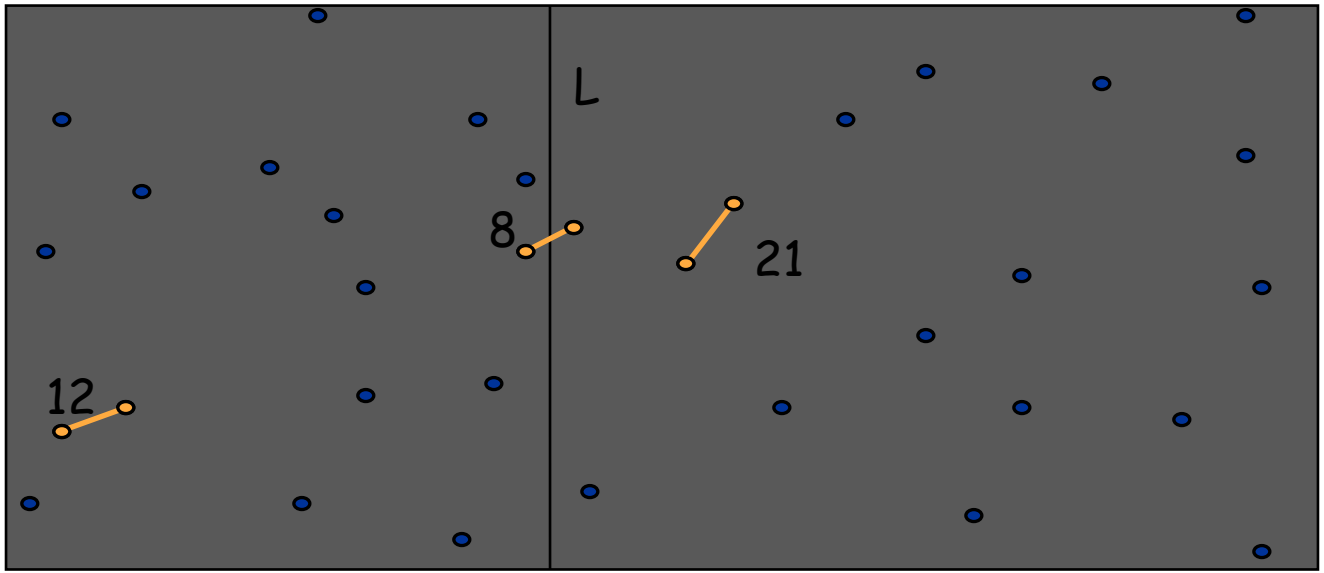
- Divisão: desenhe uma linha vertical L de forma que aproximadamente metade dos pontos estejam em cada lado.
- Conquista: Recursivamente, encontre o par mais próximo para cada lado.
  - (q_x, q_y) par mais próximo em Q. (p_x, p_y) par mais próximo em P.



Pares de Pontos mais Próximos

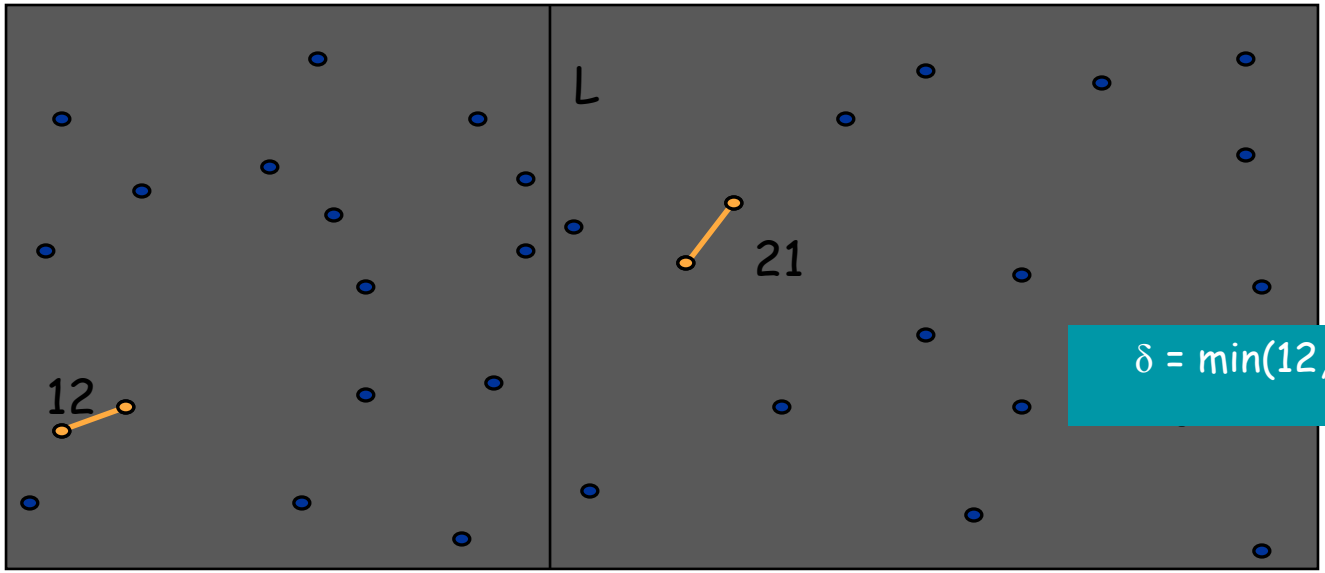
Algoritmo.

- Divisão: desenhe uma linha vertical L de forma que aproximadamente metade dos pontos estejam em cada lado.
- Conquista: encontre o par mais próximo para cada lado recursivamente.
- União: encontre o par mais próximo com um ponto em cada lado ← parece $\Theta(n^2)$
- Retorne a melhor das 3 soluções.



Pares de Pontos mais Próximos

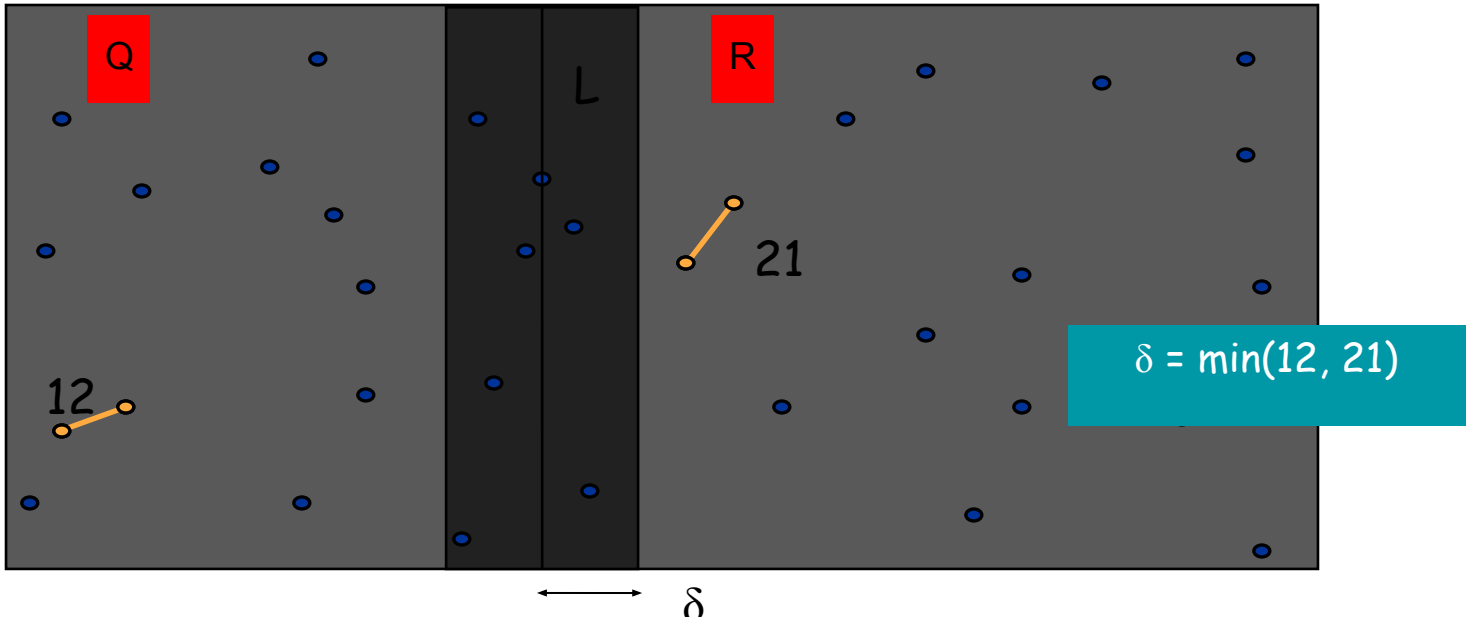
Seja δ = min ($d((q_x, q_y))$, $d((p_x, p_y))$)



Pares de Pontos mais Próximos

Seja δ = min (d((q_x, q_y)), d((p_x, p_y)))

- Observação: somente é preciso considerar os pontos dentro de δ da linha L. Obs: L é a reta posicionada em x*, ponto mais à direita de Q.



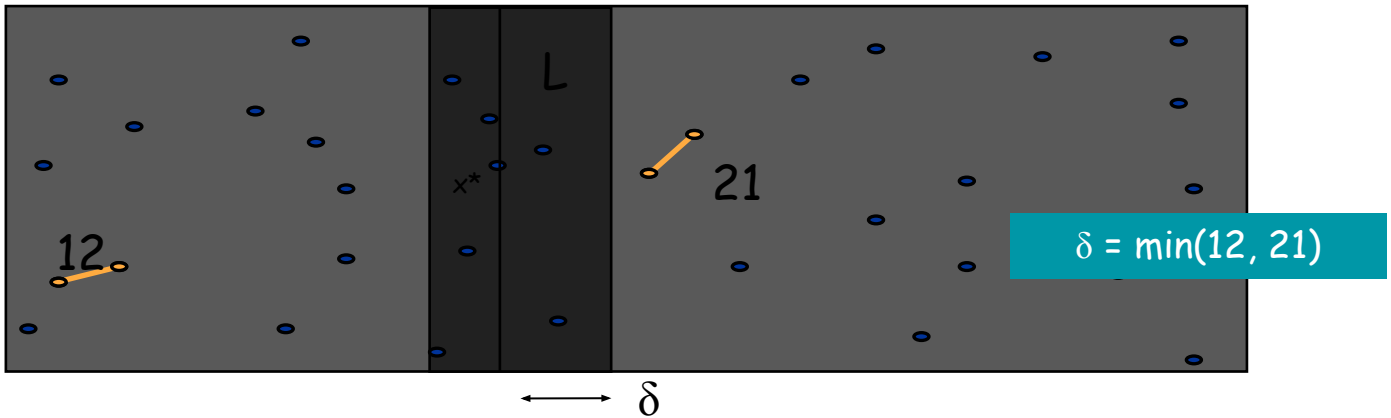
Pares de Pontos mais Próximos

- Se existe $q \in Q$ e $r \in R$ tal que $d(r,q) < \delta$, então cada q e r está a uma distância não superior a δ de L
- Prova:
  - $q = (q_x, q_y)$ e $r = (r_x, r_y)$. Por definição, sabemos que $q_x \leq x^* \leq r_x$. Então temos:

$x^* - q_x \leq r_x - q_x \leq d(r,q) < \delta$ e também

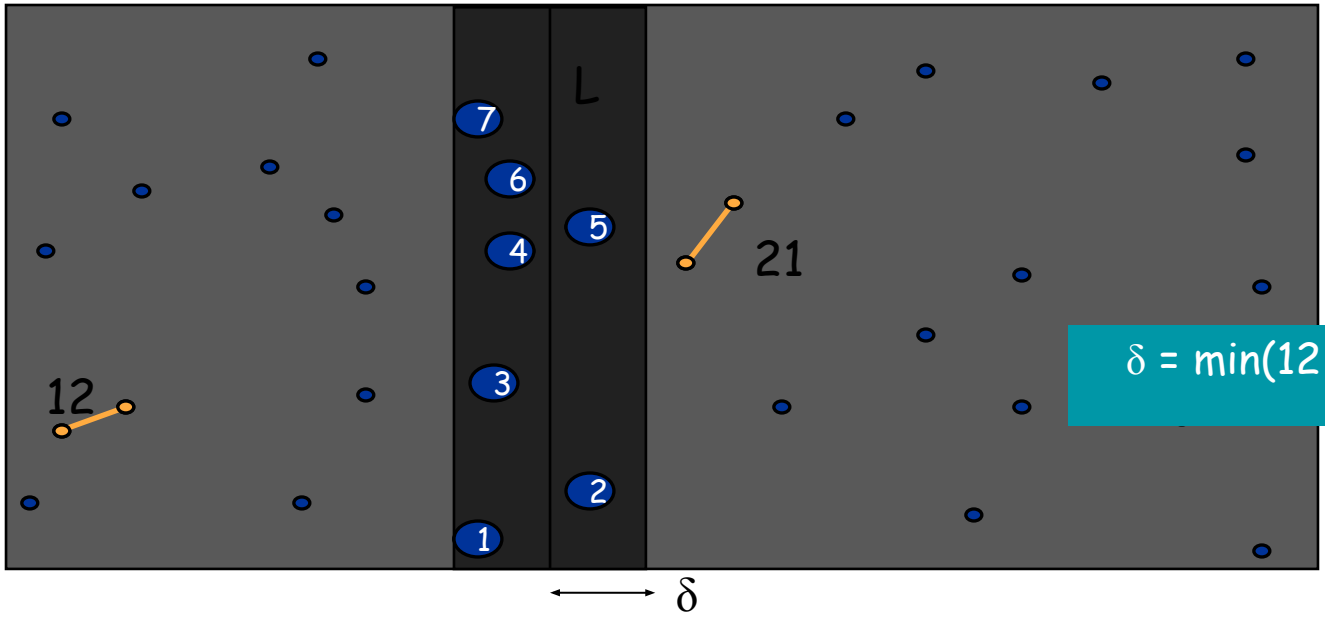
$r_x - x^* \leq r_x - q_x \leq d(r,q) < \delta$

Portanto, cada q e r tem uma coordenada x dentro do limite δ em x^* que é o mesmo que dizer que está a uma distância δ de L.



Pares de Pontos mais Próximos

- Portanto, podemos restringir a busca de r e q entre os pontos que estão em uma **porção da faixa** P, que é a faixa escura na figura abaixo… (pontos 1, 2, 3,...,7).
- Quantos pontos pode haver nesta faixa?
- Mas por quanto pontos precisamos procurar???
    - Seja S $\subseteq$ P, esta região em que os pontos estão ordenados crescentemente pela coord. y. Usando a lista P_y, isso pode ser feito em O(n)...
    - Se existe um par de pontos (q,r) cuja distância < δ , somente se existe s, s' $\in$ S

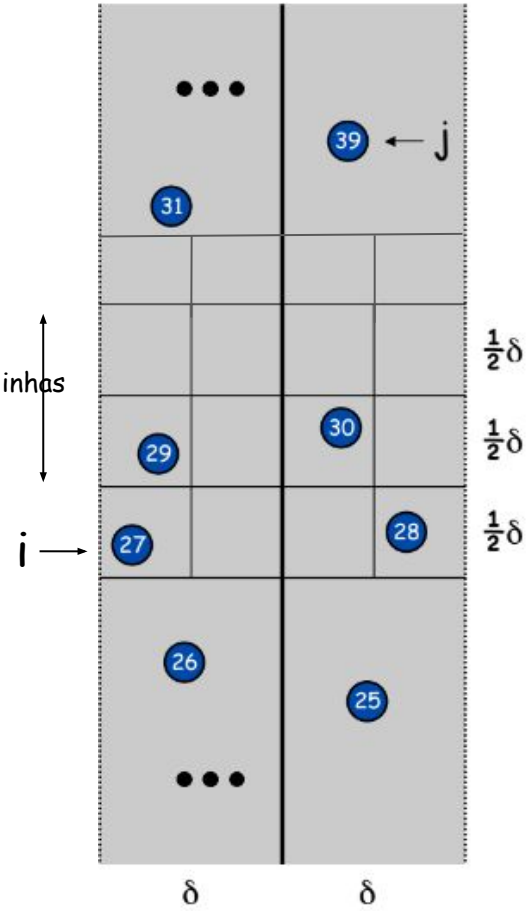


Pares de Pontos mais Próximos

- Seja s_i um ponto na faixa 2δ, sendo i[a] a menor coordenada y. (ponto 27)
- Se s_i, s ∈ S e d(s_i, s) < □, então s_i, s estão a no máximo 15 posições um do outro, na lista S_y.

- Observe a figura ao lado…
- Sobre a linha L, desenhamos quadrados de lado = δ/2. Uma linha consiste de 4 caixas cujos lados horizontais têm a mesma coordenada y.
- Não há dois pontos numa mesma caixa, pois isso contradiz nossa definição de □ mínimo em Q ou em R.

- Suponha que s_i e s_j tem d(s_i e s_j) < □ e que estejam a 16 posições de distância em S_y.
- Lembre que s_i tem a menor coord. y. Como há apenas um ponto por caixa, deve haver ao menos 3 faixas de caixas entre s_i e s_j.
- Mas qq par de pontos separados por 3 linhas de caixas deveriam estar a uma distância de no mínimo 3□/2 - **uma contradição**



Algoritmo

```
Closest-Pair(p_1, …, p_n) {
   Compute separation line L such that half the points
   are on one side and half on the other side.                  O(n log n)

   δ_1 = Closest-Pair(left half)
   δ_2 = Closest-Pair(right half)                               2T(n / 2)
   δ   = min(δ_1, δ_2)

   Delete all points further than δ from separation line L      O(n)

   Sort remaining points by y-coordinate.                       O(n log n)

   Scan points in y-order and compare distance between          O(n)
   each point and next 11 neighbors. If any of these
   distances is less than δ, update δ.
   return δ.
}
```

```
Closest-Pair(P)
  Construct P_x and P_y  (O(n log n) time)
  (p_0*, p_1*) = Closest-Pair-Rec(P_x,P_y)

Closest-Pair-Rec(P_x, P_y)
  If |P| ≤ 3 then
    find closest pair by measuring all pairwise distances
  Endif

  Construct Q_x, Q_y, R_x, R_y (O(n) time)
  (q_0*,q_1*) = Closest-Pair-Rec(Q_x, Q_y)
  (r_0*,r_1*) = Closest-Pair-Rec(R_x, R_y)

  δ = min(d(q_0*,q_1*), d(r_0*,r_1*))
  x* = maximum x-coordinate of a point in set Q
  L = {(x,y) : x = x*}
  S = points in P within distance δ of L.

  Construct S_y (O(n) time)
  For each point s ∈ S_y, compute distance from s
     to each of next 15 points in S_y
     Let s, s' be pair achieving minimum of these distances
     (O(n) time)

  If d(s,s') < δ then
     Return (s,s')
  Else if d(q_0*,q_1*) < d(r_0*,r_1*) then
     Return (q_0*,q_1*)
       Else
          Return (r_0*,r_1*)
       Endif
```

Pares de Pontos mais Próximos: Análise

Tempo de execução.

$$\mathrm{T}(n) \leq 2T(n/2) + O(n \log n) \Rightarrow \mathrm{T}(n) = O(n \log^2 n)$$

Q. Podemos calcular em O(n log n)?

Resp. Sim. Não ordenar pontos na faixa a partir do zero a cada vez.

- Cada recursão retorna duas listas: todos os pontos ordenados pela coordenada y, e todos os pontos ordenados pela coordenada x.
- Ordene por merging as duas listas pré-ordenadas.

$$T(n) \leq 2T(n/2) + O(n) \Rightarrow \mathrm{T}(n) = O(n \log n)$$