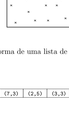
O problema dos pontos mais próximos

Imagine que nós temos uma coleção de pontos em uma região do plano



Ou melhor, os pontos são dados na forma de uma lista de coordenadas (x, y).

Por exemplo,

L	(0,2)	(1,2)	(5,3)	(7,3)	(2,5)	(3,3)	(4,8)	(3,7)	(1,8)	(5,6)

A tarefa consiste em

→ *Identificar o par de pontos que tem a menor distância entre si*

Como é que a gente faz isso?

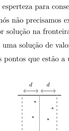
Bom, é fácil escrever um programa que calcula as distâncias entre todos os pares de pontos.

Mas, esse algoritmo é uma porcaria ...

Então, nós vamos aplicar o método da divisão e conquista.

Quer dizer, a ideia é a seguinte

1. primeiro a gente divide a região no meio
2. daí, a gente encontra o par de pontos mais próximos do lado esquerdo
3. depois, encontra o par de pontos mais próximos do lado direito
4. e depois, a gente encontra a melhor solução envolvendo um ponto de cada lado
5. finalmente, a gente escolhe a melhor solução das 3



Abaixo nós temos o código do algoritmo

```
func  Par+Proximo (L, início, fim)
{
    Se ( fim = início + 1 )    Retorna ( dist(início,fim) )

  meio  <--  (início + fim) / 2

    D1  <--  Par+Proximo (L, início, meio)

    D2  <--  Par+Proximo (L, meio+1, fim)

    D3  <--  Sol_Fronteira (L, início, meio, fim)

    Retorna ( Min ( D1, D2, D3 ) )
}
```

Análise e reconstrução

Mas, do jeito que está, o algoritmo ainda é uma porcaria.

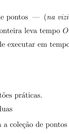
Porque só a função `Sol_Fronteira()` já leva tempo $O(n^2)$ — (*porque?*)

Então, nós vamos precisar de alguma esperteza para consertar isso.

E a esperteza consiste em notar que nós não precisamos examinar todos os pares de pontos (um de cada lado) para encontrar a melhor solução na fronteira.

Quer dizer, imagine que nós já temos uma solução de valor d em um dos dois lados.

Então, a gente só precisa examinar os pontos que estão a uma distância d da fronteira



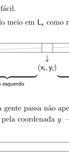
— (*porque?*)

Mas, isso ainda não ajuda muito.

Porque, em princípio, todos os pontos podem estar dentro dessa faixa.

Então agora, nós precisamos de uma segunda esperteza.

Para chegar lá, a gente foca a atenção em um desses pontos[a]



E daí pergunta

- *Quantos pontos podem estar perto de* `p` *?*

Quer dizer, se a gente já sabe que um ponto está a distância ao menos d de `p`, então a gente não precisa calcular a sua distância para `p`.

Mas, quando é que a gente sabe isso?

Bom, se a diferença entre as coordenadas y dos dois pontos é ao menos d, então a distância entre eles é ao menos d — (*porque?*)

Daí que, agora a gente pode concentrar a atenção no quadradinho

À primeira vista, isso também não ajuda muito.

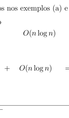
Porque, em princípio, todos os pontos podem estar dentro desse quadradinho.

Mas, isso não é verdade.

Quer dizer, a gente já sabe que a menor distância entre dois pontos do mesmo lado é d.

Logo, não pode haver dois pontos do mesmo lado do quadradinho com distância menor do que d.

No pior caso, a gente teria algo assim



Quer dizer, é uma quantidade $O(1)$ de pontos — (*na vizinhança de* `p`)

Logo, a inspeção de cada ponto na fronteira leva tempo $O(1)$.

Logo, a função `Sol_Fronteira()` pode executar em tempo $O(n)$.

Implementação e análise

Legal.

Mas agora, é preciso pensar nas questões práticas.

Quer dizer, as questões práticas são duas

1. Como é que a gente particiona a coleção de pontos no meio? — (*para fazer as chamadas recursivas*)
2. Como é que a gente localiza os pontos próximos de `p` na fronteira? — (*para calcular a melhor solução de fronteira*)

Bom, a resposta para as duas questões é a mesma

→ *mantendo a lista de pontos ordenada*

Só que, para a primeira questão, nós precisamos ordenar os pontos pela coordenada x.

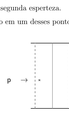
E para a segunda questão, nós precisamos ordenar os pontos pela coordenada y.

Mas, a boa notícia é que nós só precisamos fazer isso uma vez.

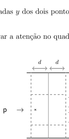
Quer dizer, a gente vai fazer essas ordenações antes de iniciar o algoritmo, e armazenar os resultados em duas listas auxiliares

Agora, a divisão da região no meio é fácil.

Quer dizer, basta tomar o elemento do meio em L_x como referência



No momento da chamada recursiva, a gente passa não apenas a faixa de pontos corresponde em L_x, mas os mesmos pontos ordenados pela coordenada y — (*filtrados da lista* L_y)



Dessa maneira, cada chamada recursiva começa exatamente da mesma maneira que a chamada anterior.

E a boa notícia é que tudo isso leva tempo $O(n)$.

Certo.

Agora é preciso examinar o cálculo da solução de fronteira.

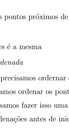
Para isso, é preciso calcular a fronteira.

E para isso, a gente calcula

```
d  <--  Min ( D1, D2 )
```

Daí, a gente seleciona em L_{x1} e L_{x2} os pontos que estão à distância no máximo d da fronteira

E filtra esses pontos em L_{y1} e L_{y2} — (*ordenados pela coordenada* y)



Agora é fácil.

Quer dizer, basta fazer a intercalação de F_1 e F_2 — (*pela coordenada* y)

E percorrer a lista resultante, calculando a distância entre cada ponto e os 7 pontos seguintes na lista.

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

E agora, a gente pode perguntar

- *Qual é o tempo de execução do algoritmo como um todo ?*

Bom, as ordenações antes do algoritmo começar levam tempo

$$O(n \log n)$$

E o algoritmo aplica o método da divisão e conquista

1. dividindo o problema na metade, resolvendo cada metade recursivamente
2. e levando tempo $O(n)$ nas etapas de divisão e conquista

Esse é o mesmo padrão dos algoritmos nos exemplos (a) e (b).

E nós já sabemos que isso leva tempo

$$O(n \log n)$$

Logo, o tempo total do algoritmo é

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

---

[a] Dar um nome é uma forma de focar a atenção.