Closest Pairs of Points

Menor Distância Entre Dois Pontos
Força Bruta x Dividir e Conquistar

CAL
Adriano Zanella Junior
Gustavo Diel

Closest Pairs of Points

O problema Closest Pairs of Points é um problema geométrico básico da computação, em que, dado um número n de pontos em um espaço Euclidiano, precisamos encontrar a menor distância entre dois deles.

Closest Pairs of Points

Um algoritimo que use força bruta teria complexidade de tempo de $O(n^2)$, com *n* como o número de ponto, mesmo aprimorando para não testar dois pontos mais de uma vez.

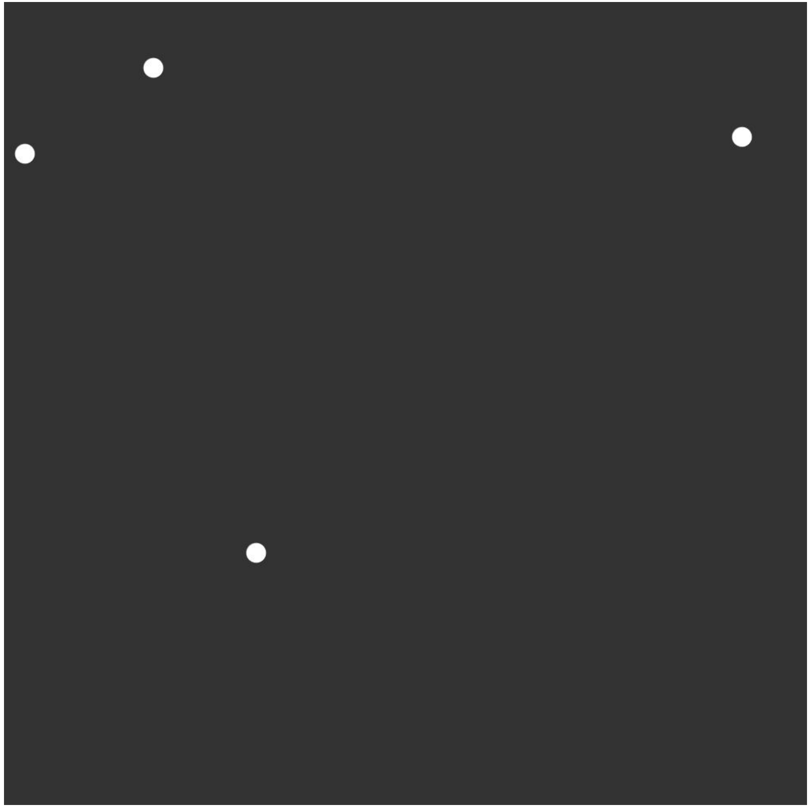
Força Bruta

Etapas:

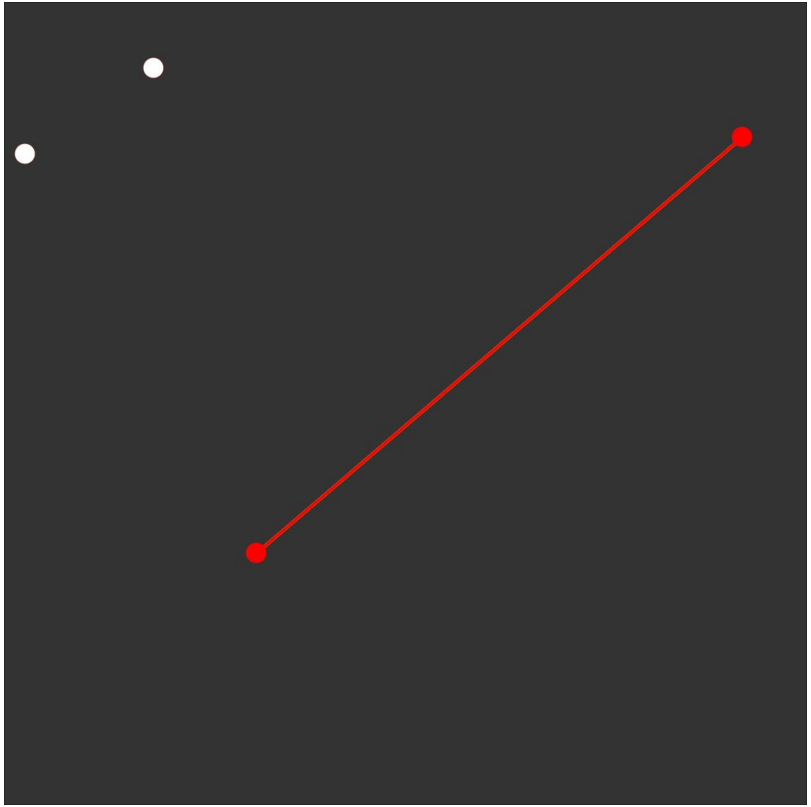
Para cada ponto, calcular a distância entre todos os outros pontos

Retorna a menor distância encontrada.

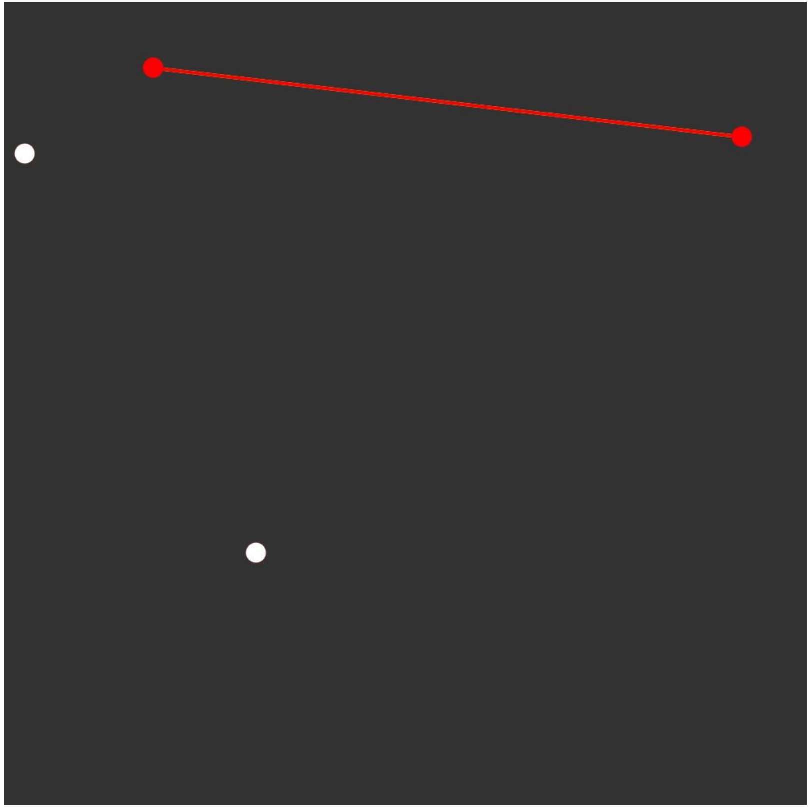
Força Bruta



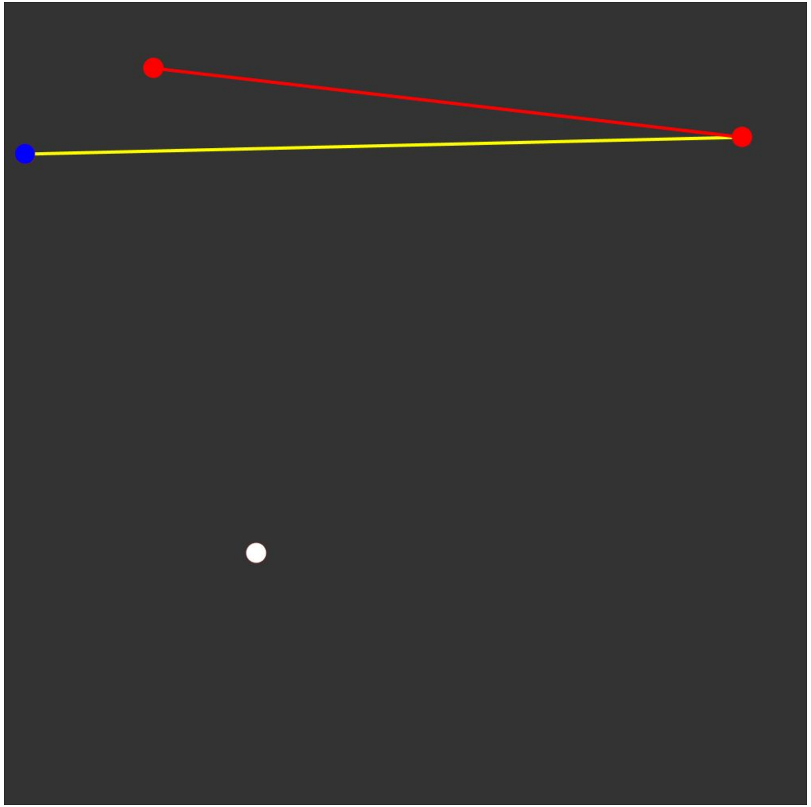
Força Bruta



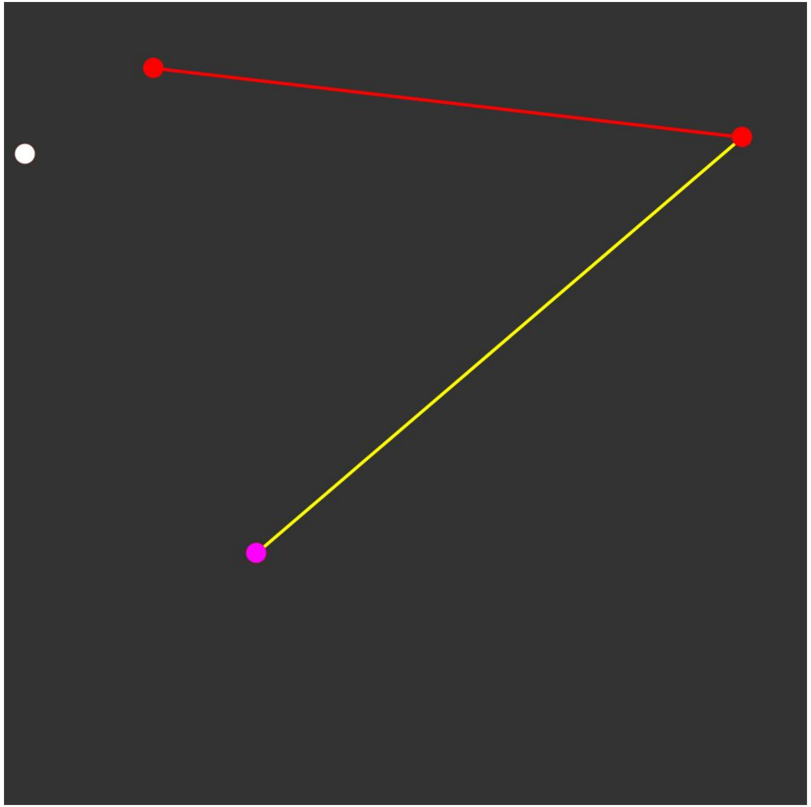
Força Bruta



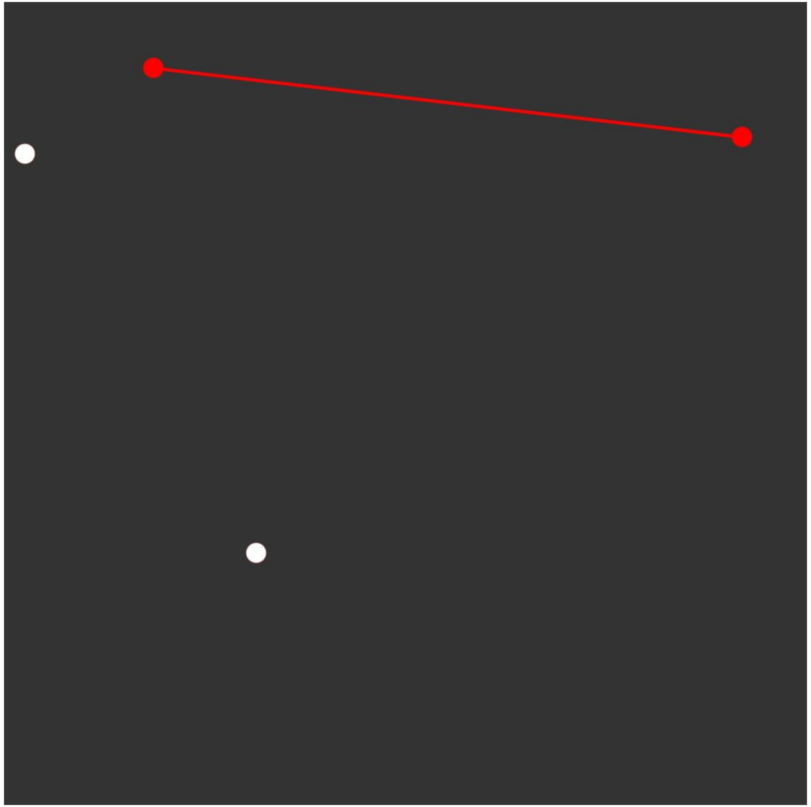
Força Bruta



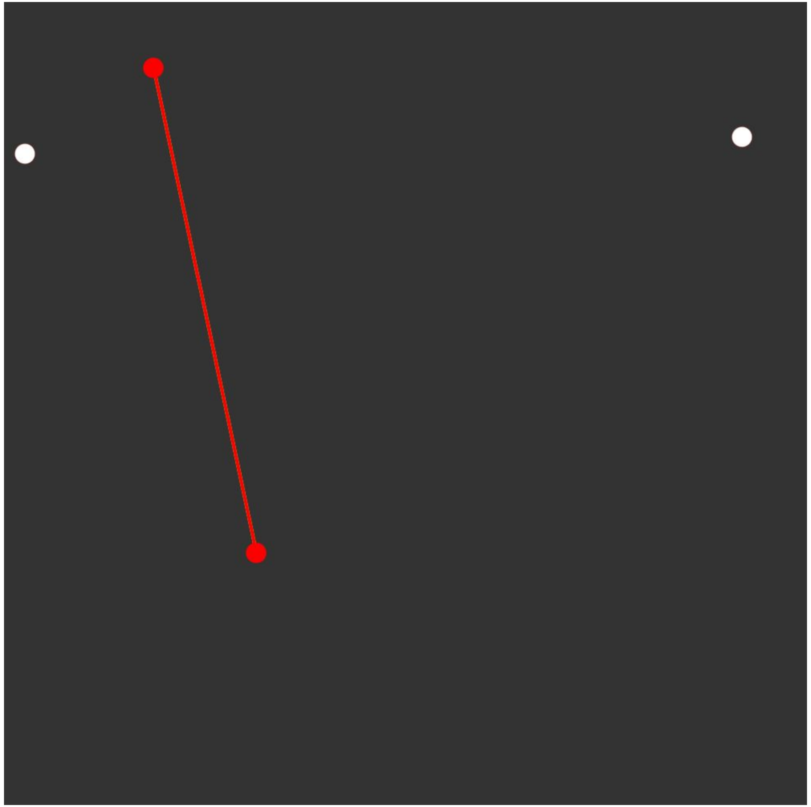
Força Bruta



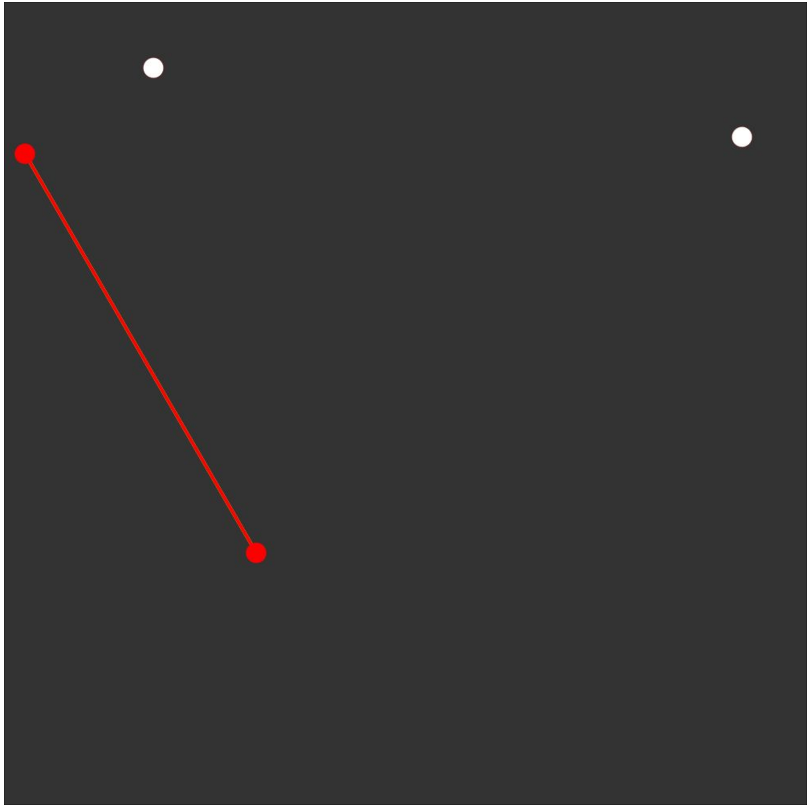
Força Bruta



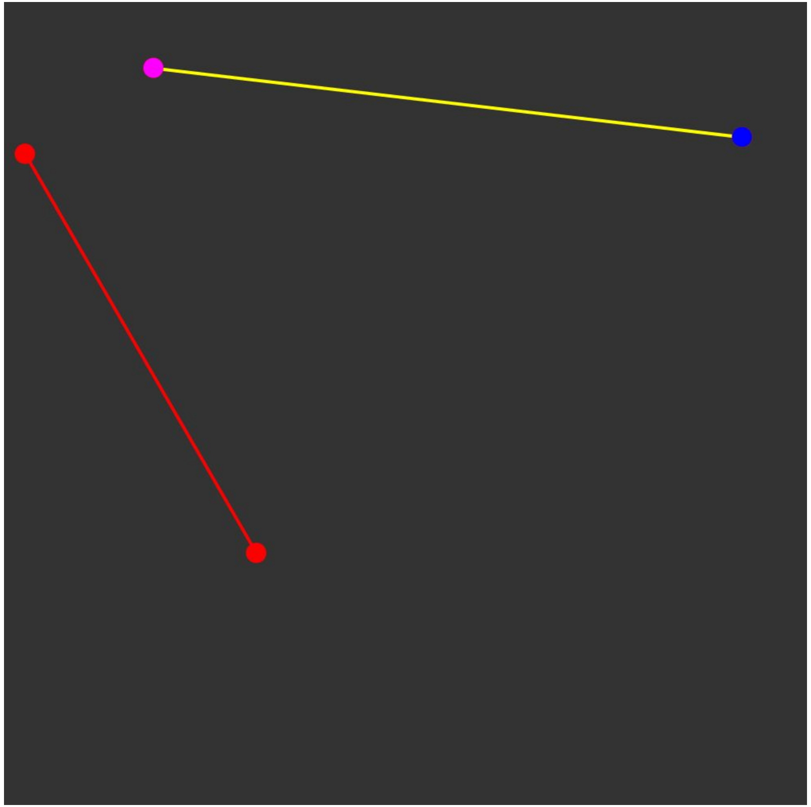
Força Bruta



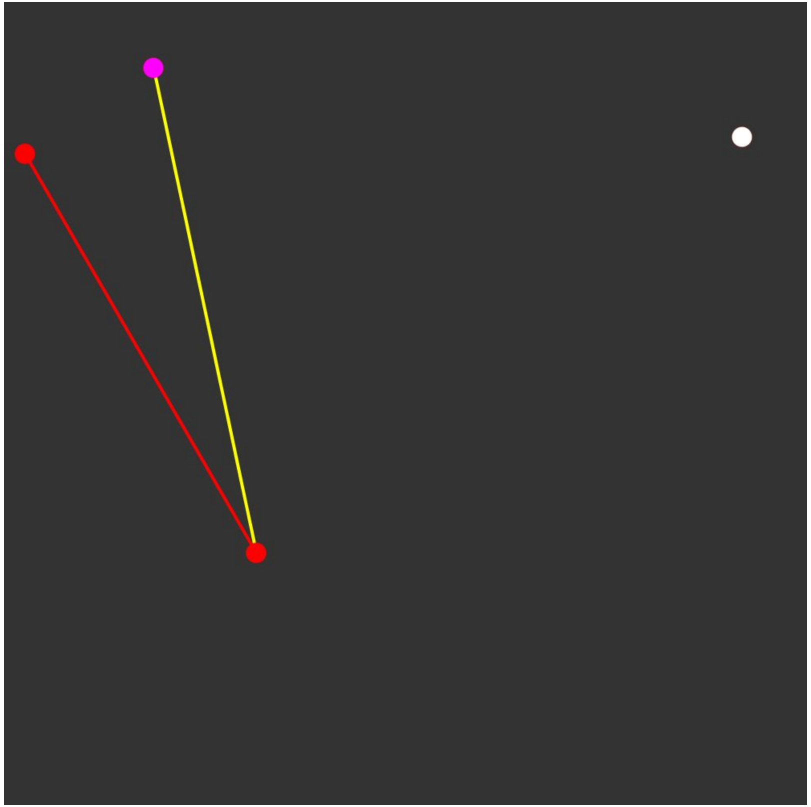
Força Bruta



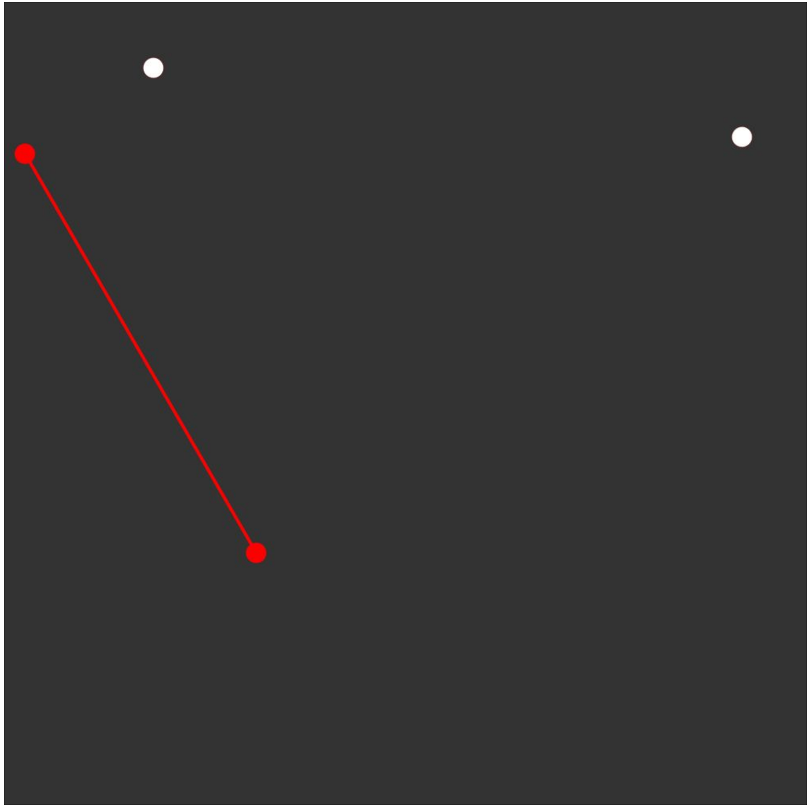
Força Bruta



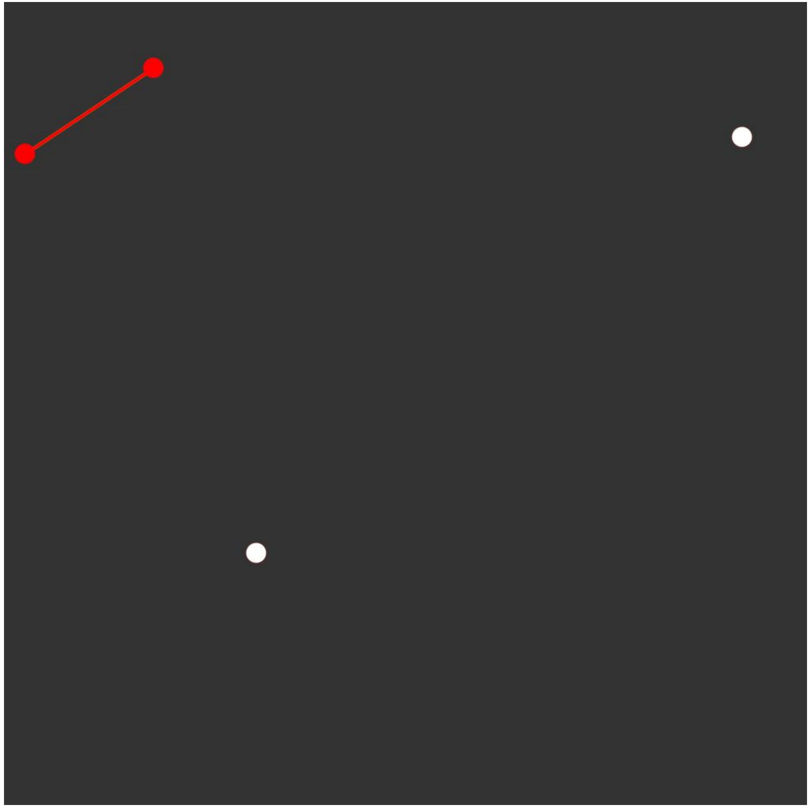
Força Bruta



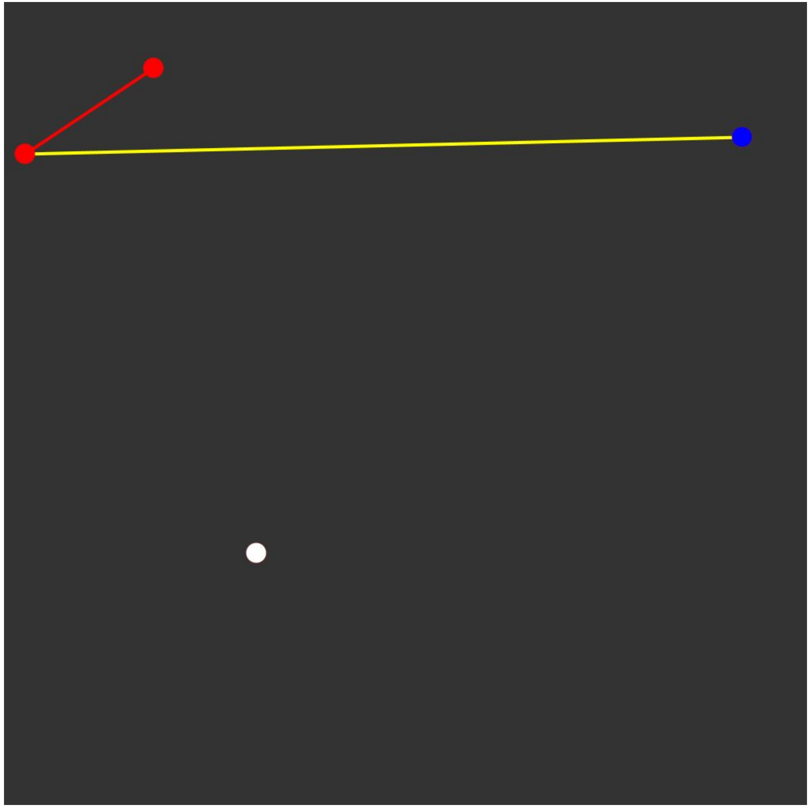
Força Bruta



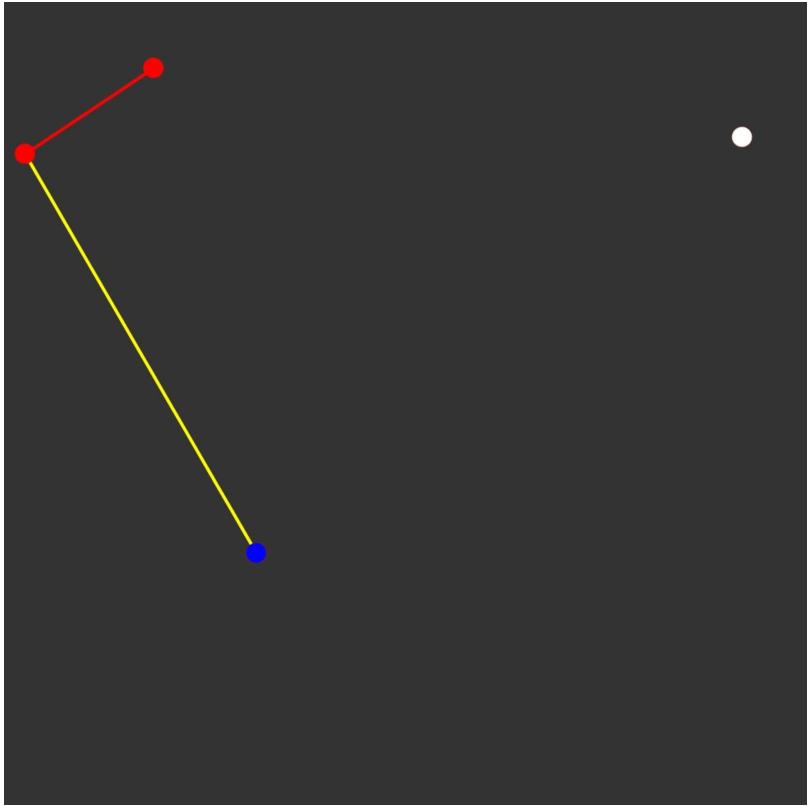
Força Bruta



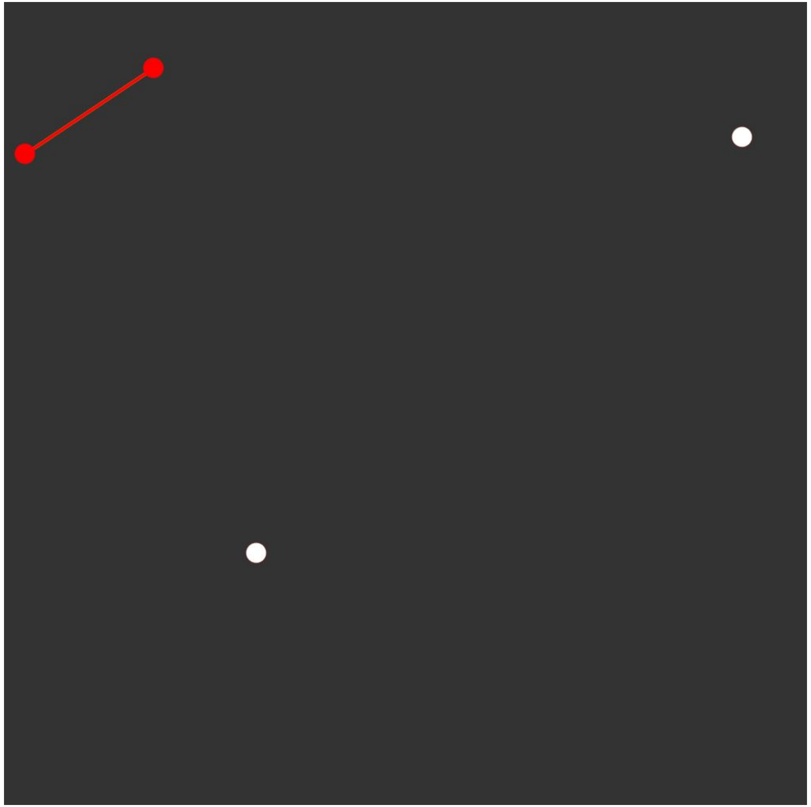
Força Bruta



Força Bruta



Força Bruta



Closest Pairs of Points

Um algoritimo eficiente, utilizando o método de divisão e conquista, consegue uma complexidade de tempo de $O(n * \log n)$. A primeira vista este algoritimo aparenta ser $O(n^2)$, mas os for aninhados acabam sendo tidos como $O(n) * O(1)$ por ocorrerem poucas vezes.

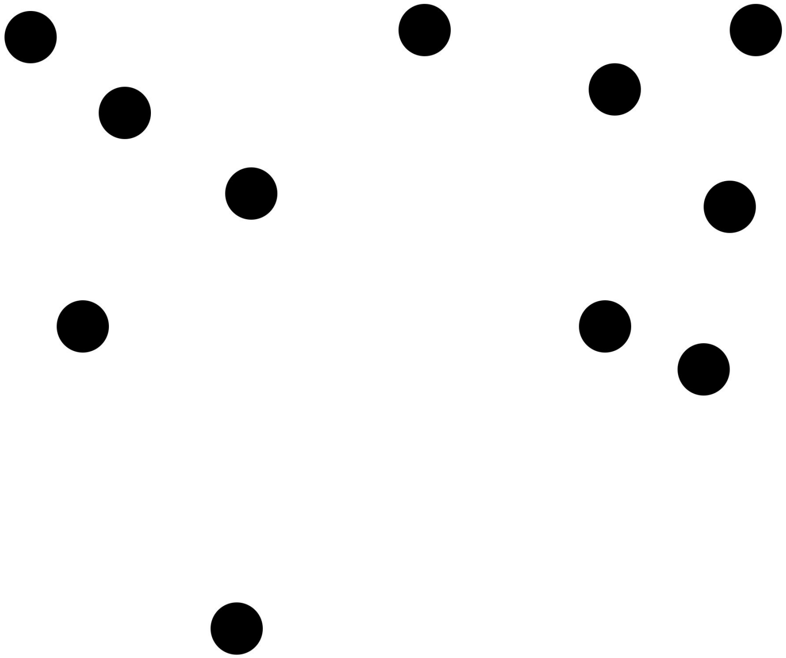
Dividir e Conquistar

Etapas:

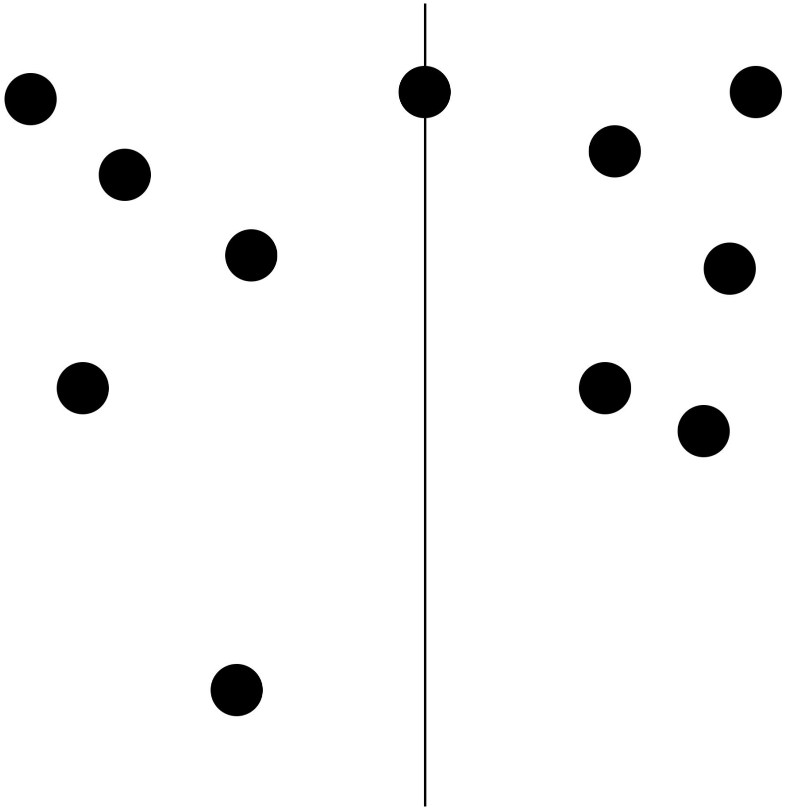
1. Ordenas os pontos em dois conjuntos em relação ao eixo x e ao eixo y;
2. Separar um dos conjuntos na metade do eixo x;
3. Resolver recursivamente pelos novos subgrupos a direita e à esquerda do meio e encontrar as menores distâncias entre esses conjunto;
4. Encontrar os pontos que estão a uma distância menor que a menor distância encontrada até então do meio, e separa los em um conjunto, então verificar se algum desses pontos possuem a menor distância entre si.

A menor distancia entre dois pontos será o minimo entre as distâncias encontradas em cada conjunto

Closest Pairs of Points

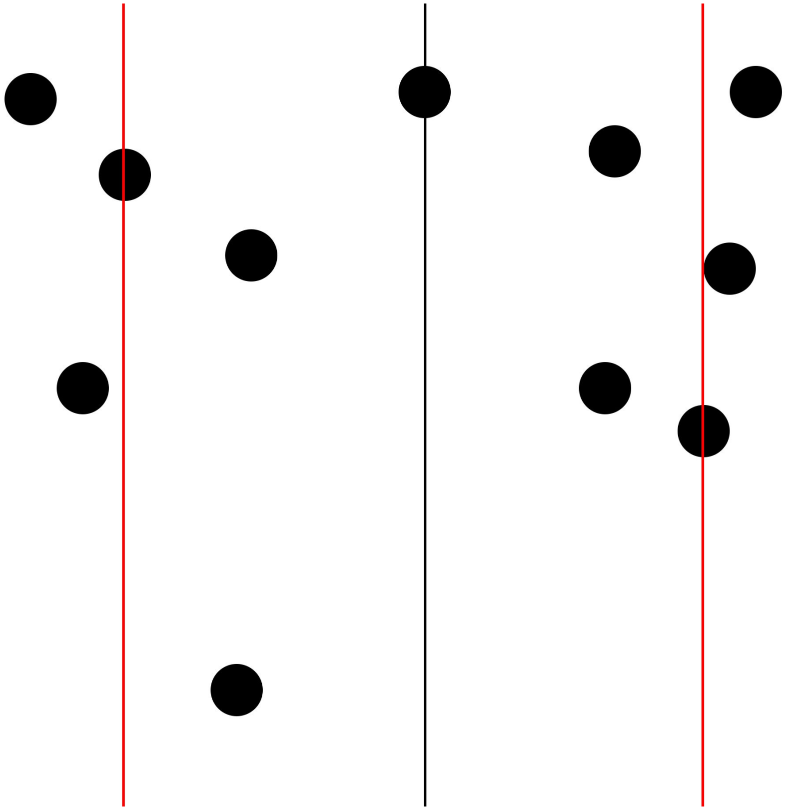


Closest Pairs of Points



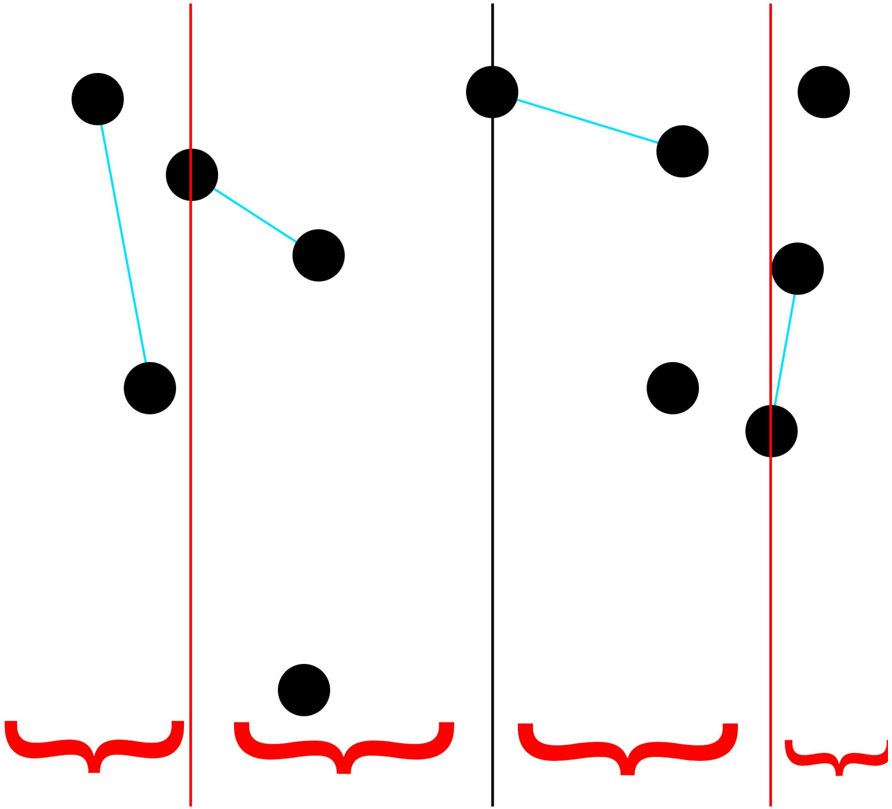
2. Separar um dos conjuntos na metade do eixo x;

Closest Pairs of Points



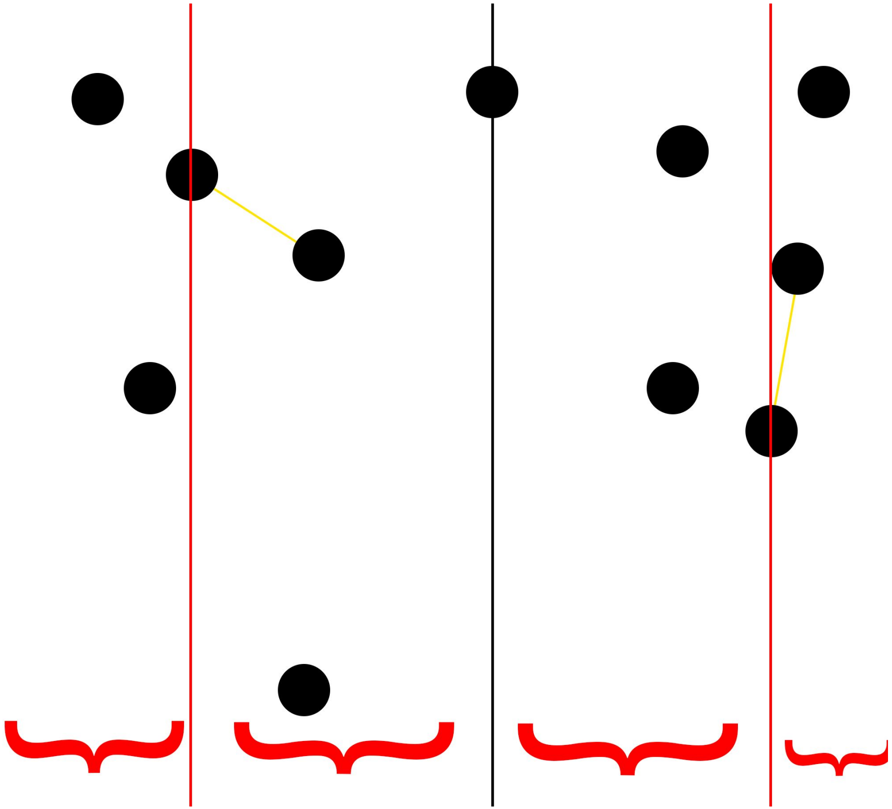
2. Separar um dos conjuntos na metade do eixo x (recursivamente);

Closest Pairs of Points



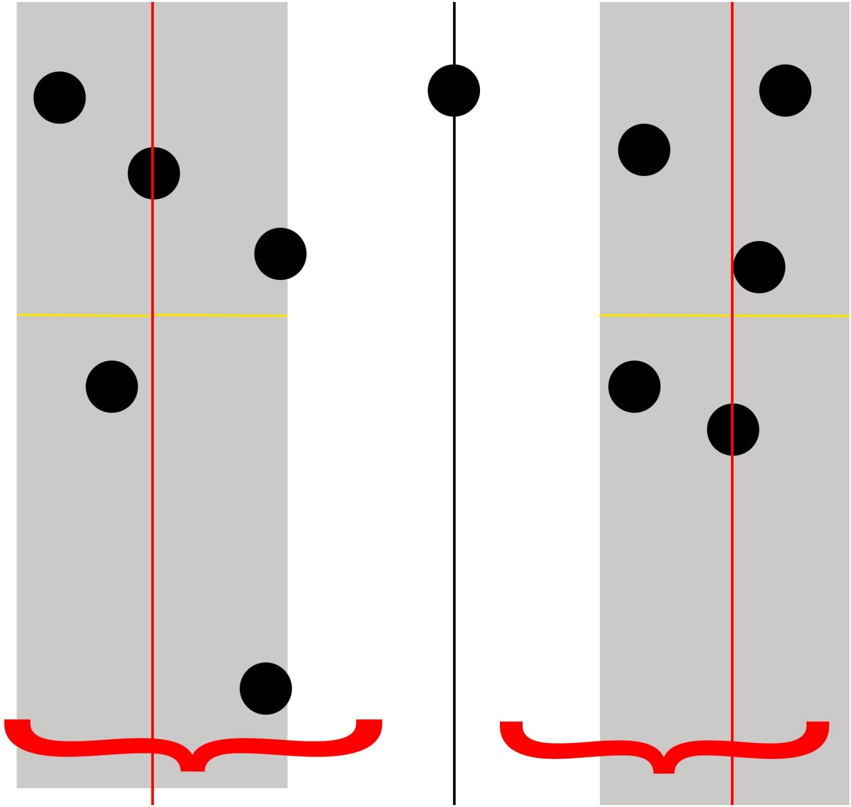
3. Resolver recursivamente pelos novos subgrupos a direita e à esquerda do meio e encontrar as menores distâncias entre esses conjunto;

Closest Pairs of Points



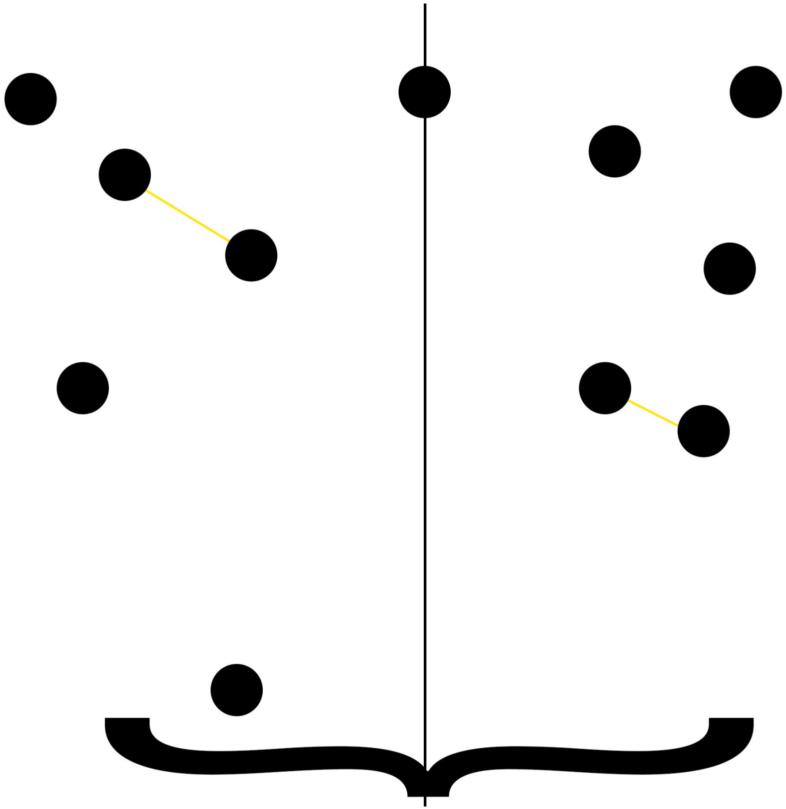
Buscamos sempre a menor distancia no conjunto.

Closest Pairs of Points

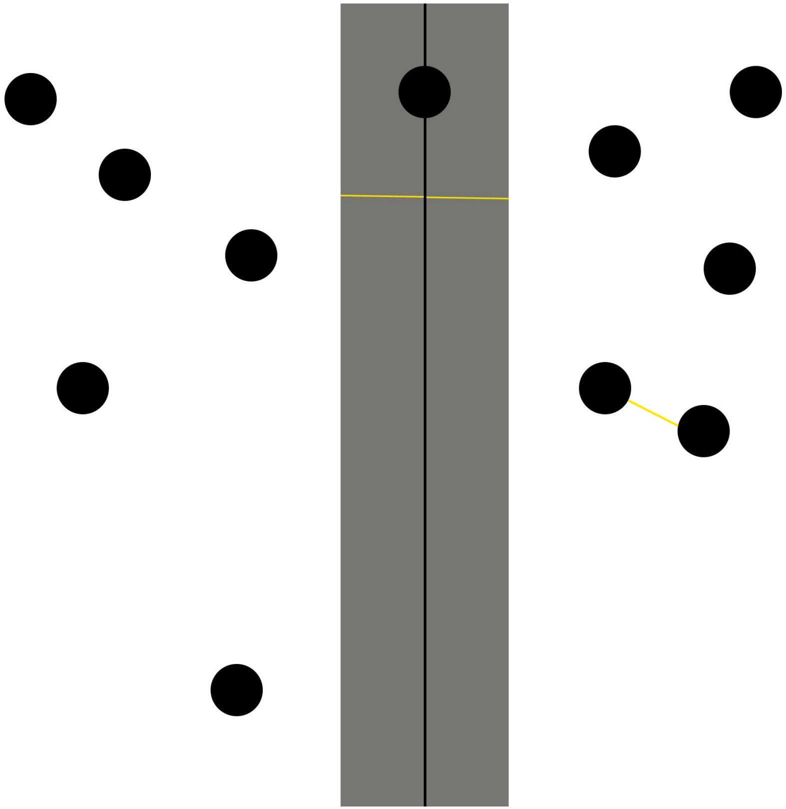


3. Encontrar os pontos que estão a uma distância menor que a menor distância encontrada até então do meio, e separa los em um conjunto, então verificar se algum desses pontos possuem a menor distância entre si.

Closest Pairs of Points

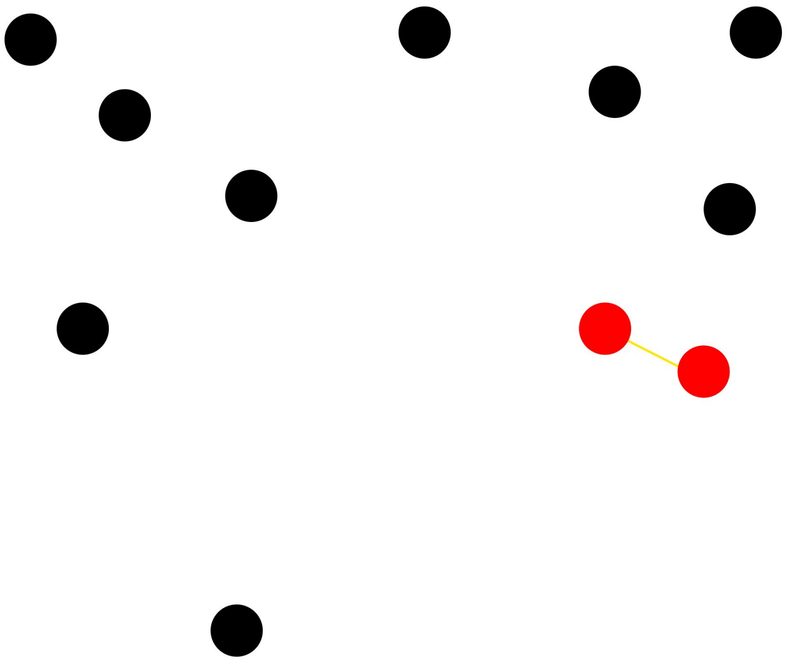


Closest Pairs of Points



2. Separar um dos conjuntos na metade do eixo x;

Closest Pairs of Points



Teremos a menor distância entre os dois pontos.

Dividir e Conquistar

2.

O algoritimo cria duas cópias das listas de pontos e ordena uma em relação a x e outra em relação a y.

```
for (auto i = 0; i < size; i++) {
	Px[i] = pontos[i];
	Py[i] = pontos[i];
}

std::qsort(Px, size, sizeof(Ponto), compareX);
std::qsort(Py, size, sizeof(Ponto), compareY);
```

Dividir e Conquistar

2.

Após isso, resolvemos recursivamente, separando em dois lados em relação ao eixo do x, mantendo a ordenação pelo y.

Uma vez que as metades estão pequenas o suficiente, utilizamos força bruta para calcular a menor distância, mas uma vez que temos poucas iterações, consideramos como O(n)

```
int meio = size / 2;
if (meio <= 3) {
        return bruteForce(Px, size);
}

...


double d1 = divideAndConquer(Px, meio1, meio);
double d2 = divideAndConquer(Px + meio, meio2, size
- meio);
```

Dividir e Conquistar

3.

Uma vez que as metades estão pequenas o suficiente, utilizamos força bruta para calcular a menor distância, mas uma vez que temos poucas iterações, consideramos como O(n).

```
int meio = size / 2;
if (meio <= 3) {
        return bruteForce(Px, size);
}

...


double d1 = divideAndConquer(Px, meio1, meio);
double d2 = divideAndConquer(Px + meio, meio2, size
- meio);
```

Dividir e Conquistar

4.

Uma vez terminado a recursividade, separamos os pontos que estão a até a menor distância encontrada em relação ao meio.

E buscamos uma possível menor distância entre esse conjunto de pontos.

```
for (int i = 0; i < size; i++) {    // O(n)
        Ponto p = Py[i];
        if (abs(p.x - mid.x) < menor) {
                strip[stripCounter++] = p;
        }
}

double dStrip = FaixaForce(strip, stripCounter);
```

Dividir e Conquistar

O cálculo dos pontos próximos ao meio pode aparentar como sendo de complexidade $O(n^2)$, o número máximo de pontos com distância inferior a menor distância encontrada até aquele ponto é de 6 pontos, fazendo com que tenha complexidade de $6n$, logo $O(n)$.

Dividir e Conquistar

Teremos como complexidade de tempo resultante:

T1(n) = t(n) + 2O(n*log n)   {2O(n*log n) sendo as funções de ordenação}

t(n) = 2t(n/2) + O(n*log n) {sendo O(n*log n) a função em relação ao conjunto próximo ao meio }

Pelo teorema mestre teremos:

t(n) = O(n*log n), e assim:

T(n) = O(n*log n)

Dividir e Conquistar

Em relação ao espaço, alocamos memória por recursividade, tendo sempre dois vetores de tamanho igual a metade do anterior, assim:

$E(n) = 2\Theta(n) + e(n)$

$e(n) = e(n/2) + 2\Theta(n)$

Pelo teorema mestre, teremos:

$e(n) = \Theta(n)$, assim:

$E(n) = \Theta(n)$

Closest Pairs of Points

Existem algoritmos mais eficientes que o Dividir e Conquistar.

Utilizando métodos mais avançados é possível reduzir para $O(n * \log \log n)$. Com a combinação desses e de outros métodos é possível reduzir para um tempo linear de $O(n)$.

Comparativo

Nº pontos	Força Bruta (us)	Dividir e Conquistar menos eficiente na faixa do meio (us)	Dividir e Conquistar eficiente (us)
10	10	9	7
100	370	128	81
1000	32723	1770	1156
10000	3098538	26889	10167
50000	77075612	188163	61297