

025089 – Projeto e Análise de Algoritmos

Aula 05

Design de Algoritmos

- Padrões/Projeto/Design/Paradigma de algoritmos
 - Estratégias frequentemente utilizadas para resolver problemas
- Modelos:
 - Força-bruta (busca-completa)
 - Greedy (guloso ou ganancioso)
 - Divisão e conquista
 - Programação dinâmica
 - Transformação/Redução e conquista

Busca completa (ou força-bruta)

- Resolver o problema testando todas (ou parte) das soluções possíveis!
- Características:
 - Simplicidade da solução (não necessariamente para problemas simples)
 - Estratégia mais intuitiva
 - Complexidade muito alta (espaço de busca grande)
 - Boa solução para entradas pequenas
 - Podemos otimizar limitando o espaço de busca
 - Útil para auxiliar no entendimento do problema, para em seguida, desenvolver uma solução mais eficiente

Busca completa (ou força-bruta)

- Abordagens:
 - Iterativa
 - Recursiva
- Otimização:
 - Diminuir o espaço de busca
 - Eliminar respostas impossíveis
 - Já previamente definidas na solução do problema
 - Decididas ao longo da solução
 - Aproveitar-se da simetria de soluções
 - Alterar ou inverter a abordagem da solução
 - Pré computar cálculos recorrentes

Exemplo 1

- Busca em um conjunto:

ALGORITHM *SequentialSearch2*($A[0..n]$, K)

//Implements sequential search with a search key as a sentinel

//Input: An array A of n elements and a search key K

//Output: The index of the first element in $A[0..n - 1]$ whose value is

// equal to K or -1 if no such element is found

$A[n] \leftarrow K$

$i \leftarrow 0$

while $A[i] \neq K$ **do**

$i \leftarrow i + 1$

if $i < n$ **return** i

else return -1

Exemplo 2

- Busca por palavra:

Padrão:

A	C	T	G
---	---	---	---

Texto:

A	C	G	A	A	G	A	C	T	G	T	C	A	C	T	G	A	C	T	C	A	C	G	G
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Exemplo 2

- Busca por palavra:

ALGORITHM *BruteForceStringMatch*($T[0..n - 1]$, $P[0..m - 1]$)

//Implements brute-force string matching
//Input: An array $T[0..n - 1]$ of n characters representing a text and
// an array $P[0..m - 1]$ of m characters representing a pattern
//Output: The index of the first character in the text that starts a
// matching substring or -1 if the search is unsuccessful

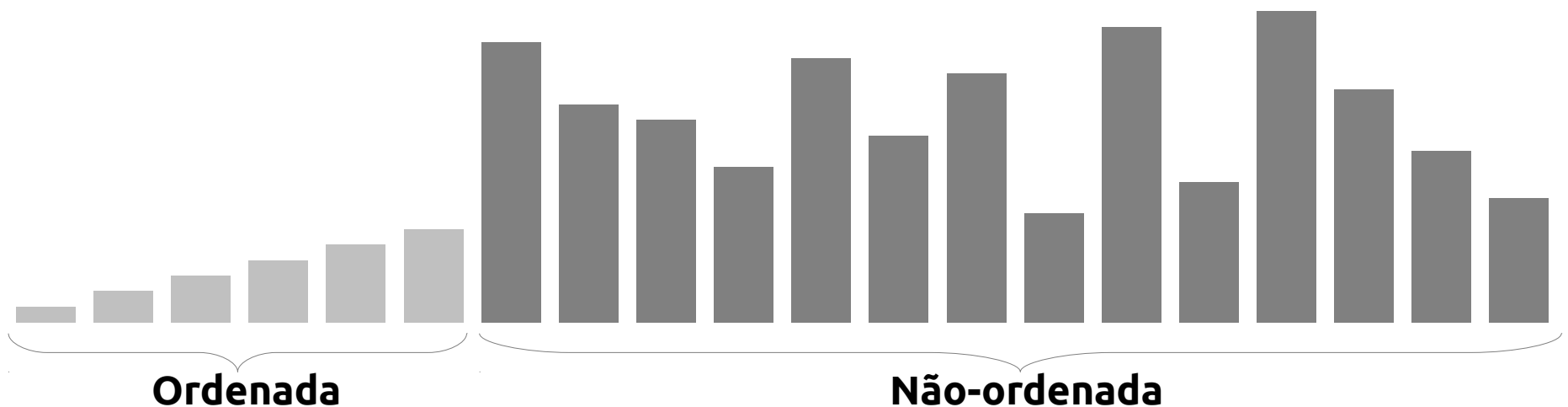
for $i \leftarrow 0$ **to** $n - m$ **do**
 $j \leftarrow 0$
 while $j < m$ **and** $P[j] = T[i + j]$ **do**
 $j \leftarrow j + 1$
 if $j = m$ **return** i
return -1

Exemplo 3

- Ordenação de um conjunto:
 - SelectionSort
 - InsertionSort (Levitin classifica como redução e conquista!)
 - BubbleSort

Selection Sort

- Divisão dos dados em duas sequências: ordenada e não-ordenada
- Iteração: procurar pelo **menor** elemento da sequência não-ordenada e concatená-lo na sequência ordenada



Selection Sort

```
template <class Item>
void selection(Item vetor[], int n)
{
    for (int i = 0; i < n; i++)
    {
        int min = i;

        for (int j = i+1; j < n; j++)
            if (vetor[j] < vetor[min])
                min = j;

        swap(vetor[i], vetor[min]);
    }
}
```

i

controla a iteração,
índice da sequência ordenada

j

controla a busca pelo valor
mínimo, índice da sequência
não-ordenada

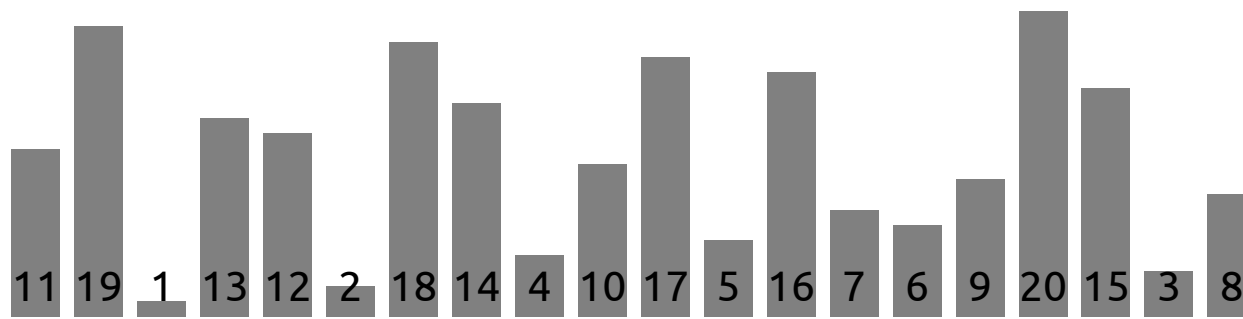
min

Índice do menor elemento da
sequência não-ordenada

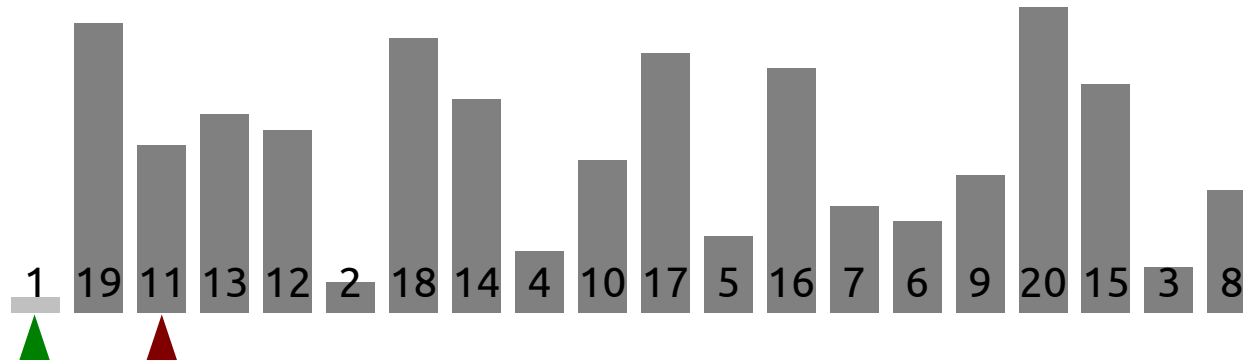
```
template <class Item>
void swap(Item &A, Item &B)
{ Item t = A ; A = B; B = t; }
```

SelectionSort

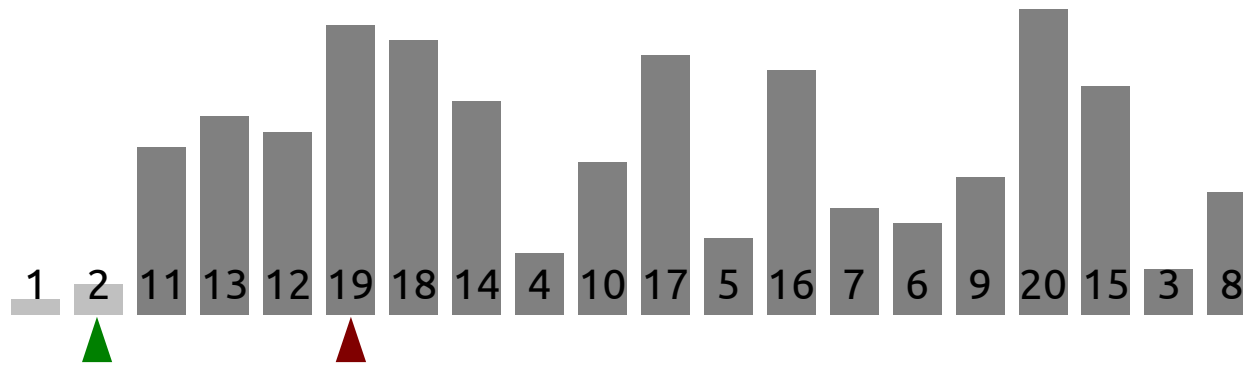
Entrada:



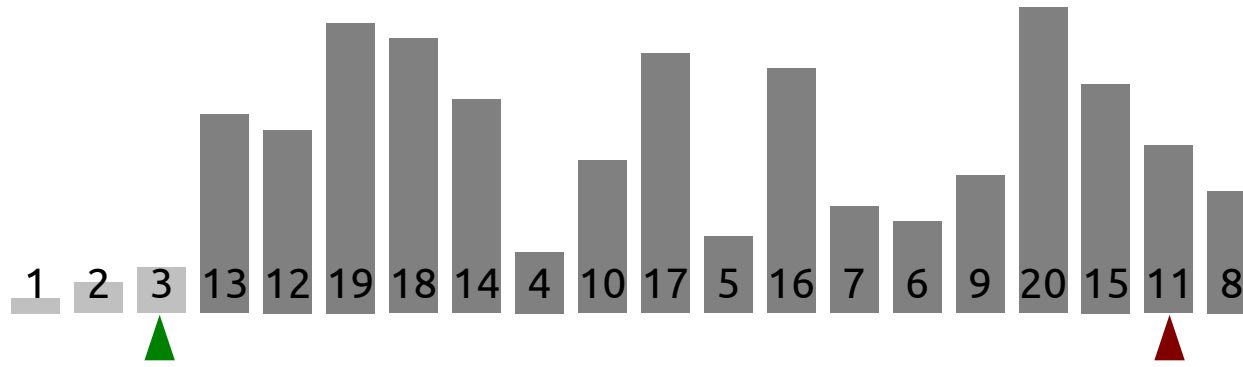
Passo 1:



Passo 2:



Passo 3:



0-3

SelectionSort

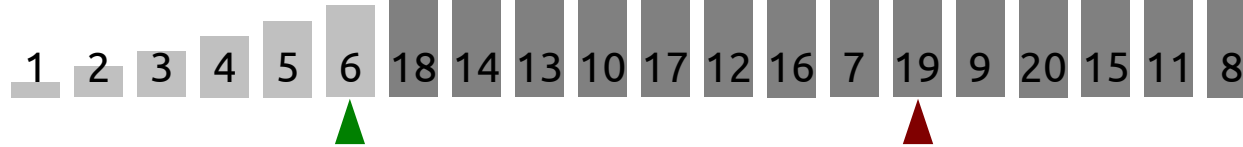
Passo 4:



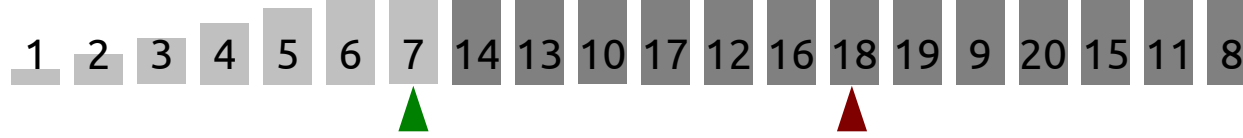
Passo 5:



Passo 6:



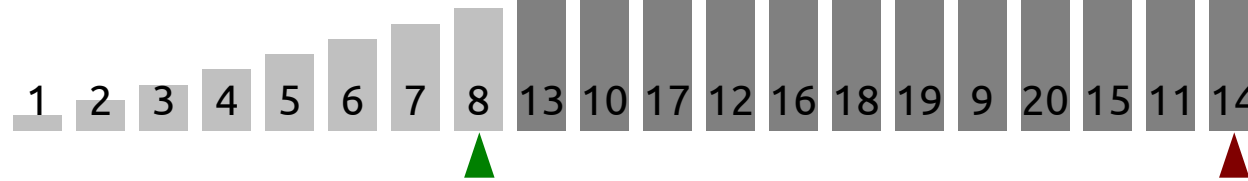
Passo 7:



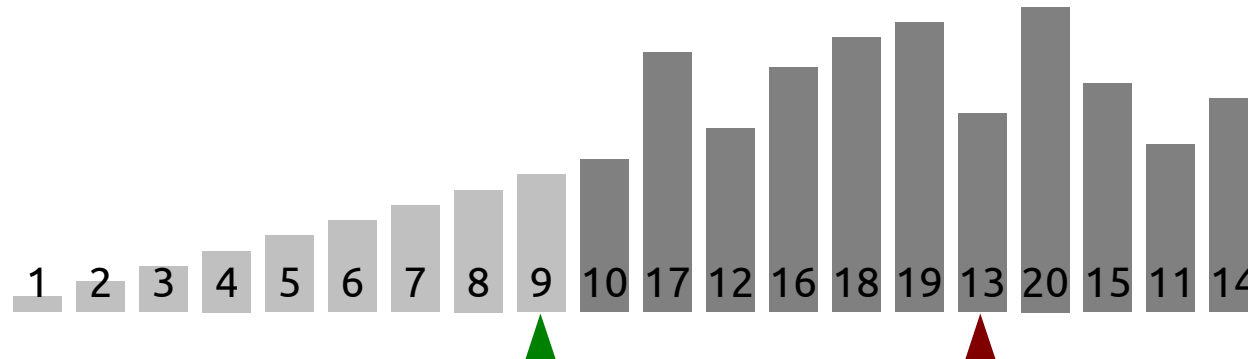
4-7

SelectionSort

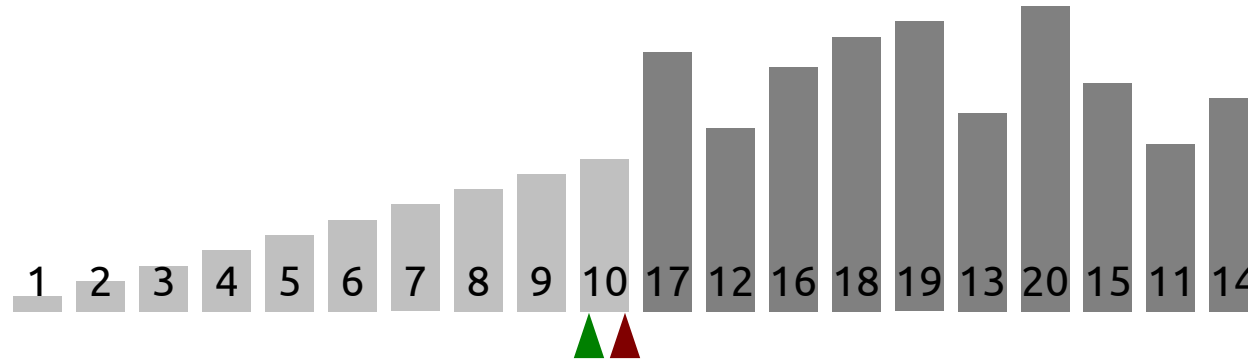
Passo 8:



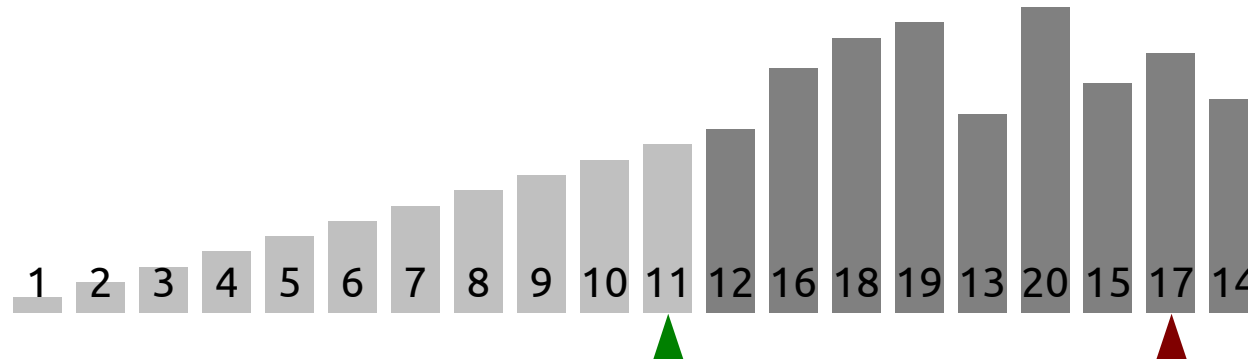
Passo 9:



Passo 10:



Passo 11:



8-11

SelectionSort

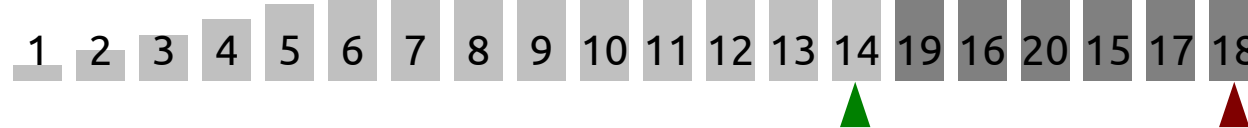
Passo 12:



Passo 13:



Passo 14:



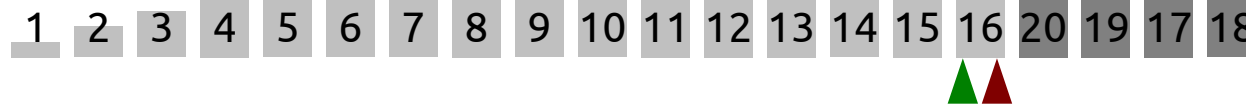
Passo 15:



12-15

SelectionSort

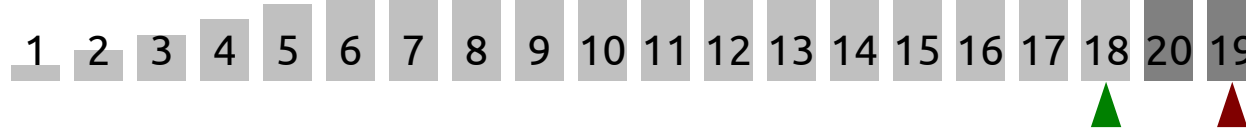
Passo 16:



Passo 17:



Passo 18:



Passo 19:



15-19

Selection Sort

- Quantas comparações são executadas?
- Quantas trocas são executadas?

```
template <class Item>
void selection(Item vetor[], int n)
{
    for (int i = 0; i < n; i++)
    {
        int min = i;

        for (int j = i+1; j < n; j++)
            if (vetor[j] < vetor[min])
                min = j;

        swap(vetor[i], vetor[min]);
    }
}
```


Selection Sort

- Quantas comparações são executadas?
- Quantas trocas são executadas?

Sempre!

O número de comparações ou trocas não dependem dos valores dos dados.

n-1, n-2, n-3, ..., 1

n x 1

```
template <class Item>
void selection(Item vetor[], int n)
{
    for (int i = 0; i < n; i++)
    {
        int min = i;

        for (int j = i+1; j < n; j++)
            if (vetor[j] < vetor[min])
                min = j;

        swap(vetor[i], vetor[min]);
    }
}
```

Selection Sort

- Os valores dos dados não interferem na execução do algoritmo
- Crescimento do número de trocas em relação ao tamanho de entrada: **linear**
- Crescimento do número de comparações em relação ao tamanho de entrada: **quadrático**
- Crescimento do uso de memória em relação ao tamanho da entrada: **constante**
- Algoritmo não é estável

Estabilidade

- Preservação da ordem relativa de elementos com chaves iguais

sorted by time	sorted by location (not stable)	sorted by location (stable)
Chicago 09:00:00	Chicago 09:25:52	Chicago 09:00:00
Phoenix 09:00:03	Chicago 09:03:13	Chicago 09:00:59
Houston 09:00:13	Chicago 09:21:05	Chicago 09:03:13
Chicago 09:00:59	Chicago 09:19:46	Chicago 09:19:32
Houston 09:01:10	Chicago 09:19:32	Chicago 09:19:46
Chicago 09:03:13	Chicago 09:00:00	Chicago 09:21:05
Seattle 09:10:11	Chicago 09:35:21	Chicago 09:25:52
Seattle 09:10:25	Chicago 09:00:59	Chicago 09:35:21
Phoenix 09:14:25	Houston 09:01:10	Houston 09:00:13
Chicago 09:19:32	Houston 09:00:13	Houston 09:01:10
Chicago 09:19:46	Phoenix 09:37:44	Phoenix 09:00:03
Chicago 09:21:05	Phoenix 09:00:03	Phoenix 09:14:25
Seattle 09:22:43	Phoenix 09:14:25	Phoenix 09:37:44
Seattle 09:22:54	Seattle 09:10:25	Seattle 09:10:11
Chicago 09:25:52	Seattle 09:36:14	Seattle 09:10:25
Chicago 09:35:21	Seattle 09:22:43	Seattle 09:22:43
Seattle 09:36:14	Seattle 09:10:11	Seattle 09:22:54
Phoenix 09:37:44	Seattle 09:22:54	Seattle 09:36:14

no longer sorted by time

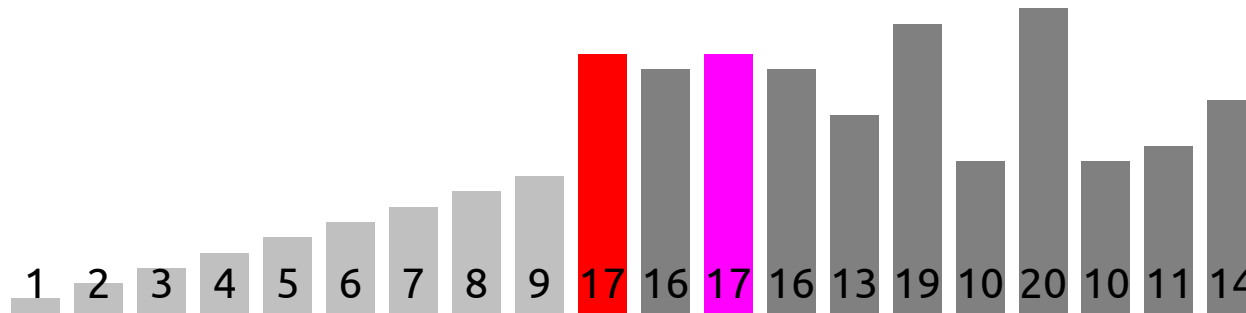
still sorted by time

Porque o SelectionSort não é estável?

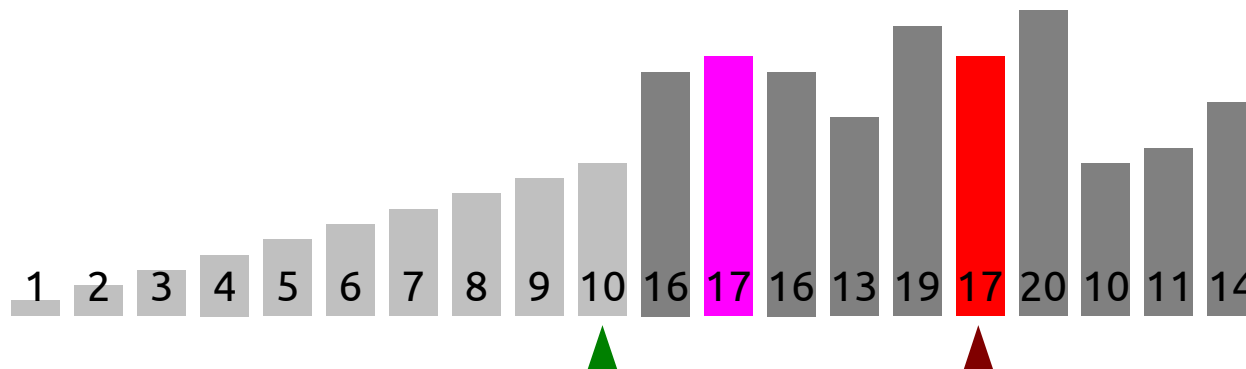
Selection Sort

- Não é estável:
 - No momento da troca pelo menor elemento da sequência não-ordenada, a ordem relativa entre chaves iguais pode ser quebrada

Passo i:

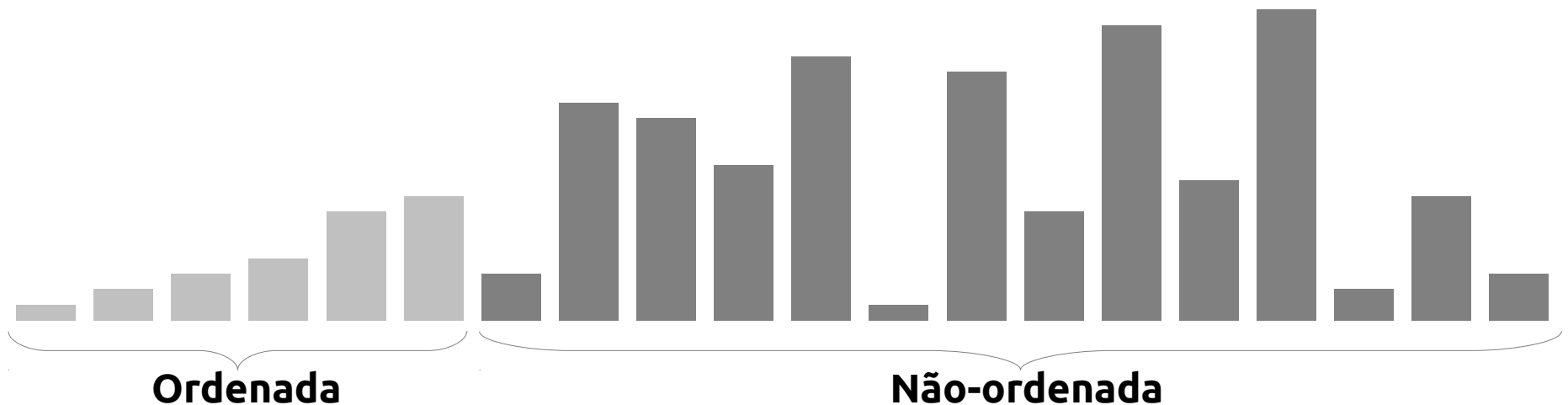


Passo i+1:



Insertion Sort

- Divisão dos dados em duas sequências: ordenada e não-ordenada
- Iteração: inserir o primeiro elemento da sequência não-ordenada na sequência ordenada



Insertion Sort

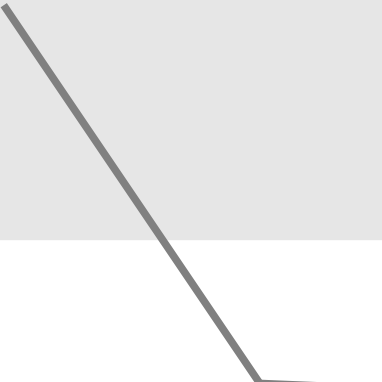
```
template <class Item>
void insertion(Item vetor[], int n)
{
    for (int i = 1; i < n; i++)
    {
        int j = i;
        while ( j > 0 && vetor[j-1] > vetor[j])
        {
            swap(vetor[j-1], vetor[j]);
            j--;
        }
    }
}
```

i

controla a iteração,
índice da sequência
ordenada

j

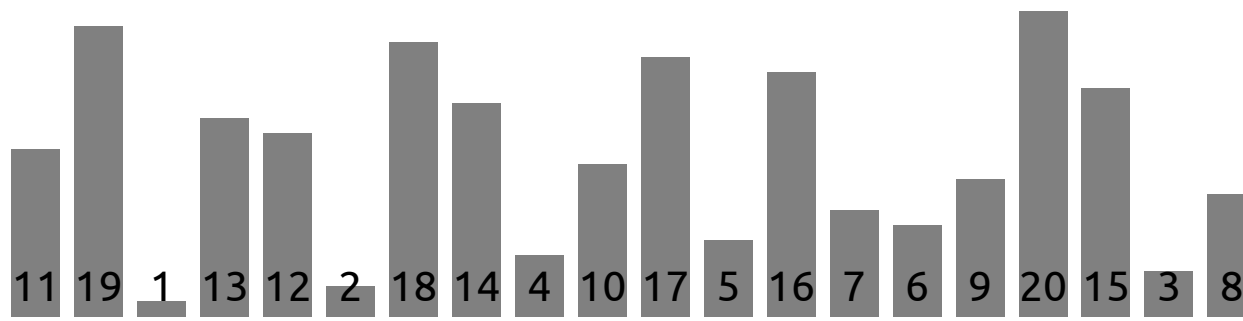
controla a inserção do
elemento da sequência
não-ordenada na
sequência ordenada



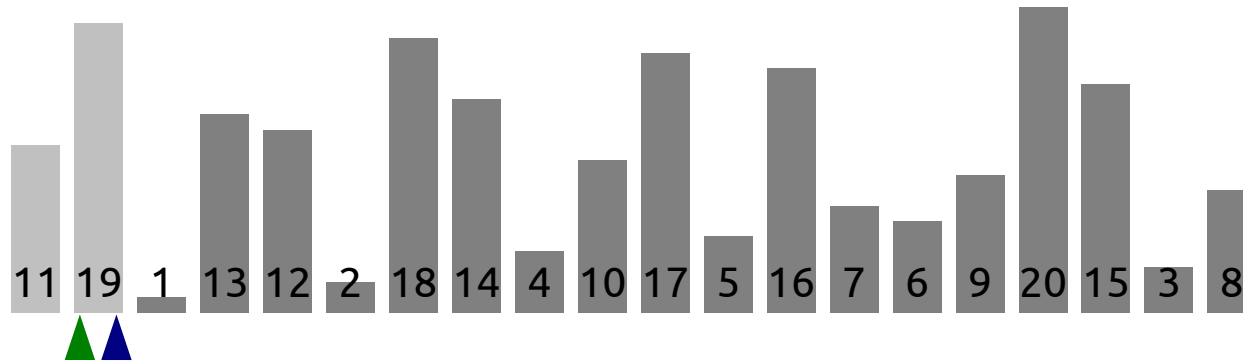
```
template <class Item>
void swap(Item &A, Item &B)
{ Item t = A ; A = B; B = t; }
```

InsertionSort

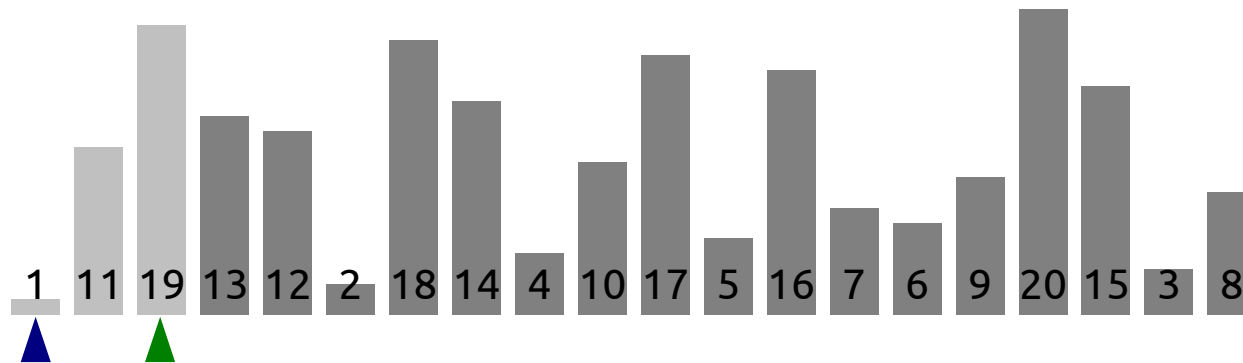
Entrada:



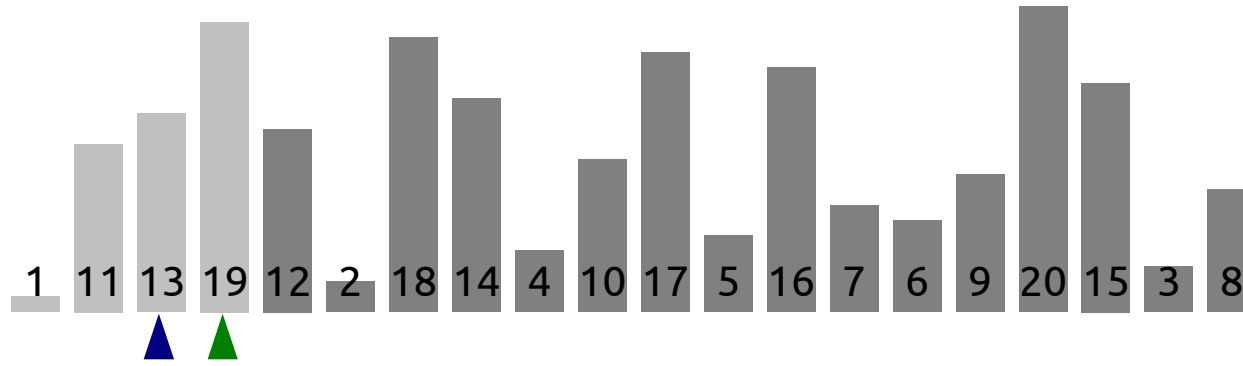
Passo 1:



Passo 2:



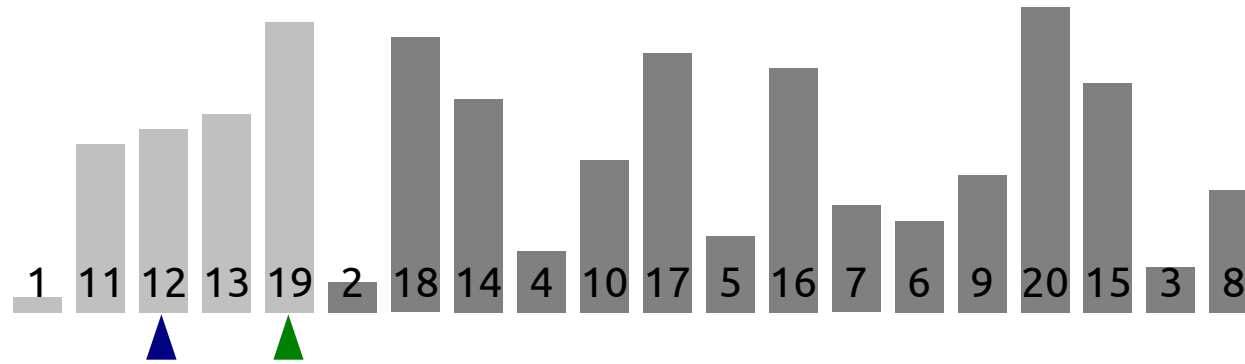
Passo 3:



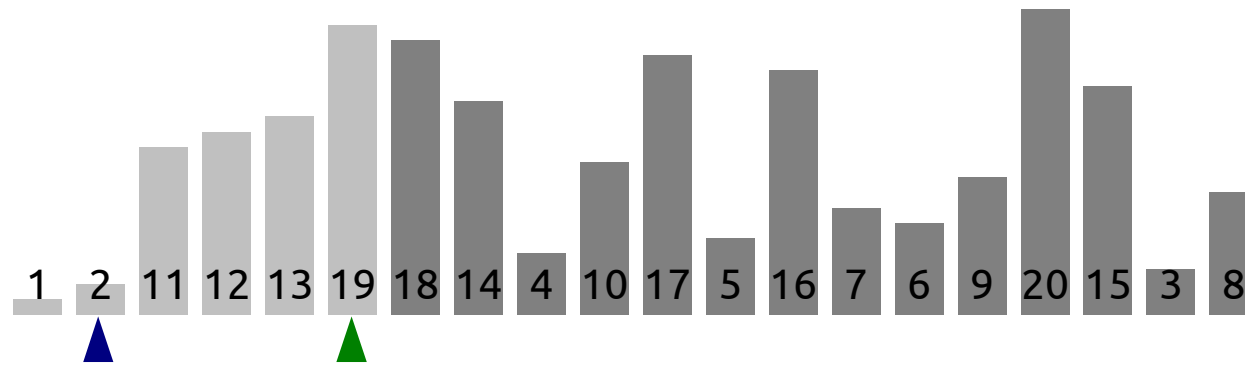
0-3

InsertionSort

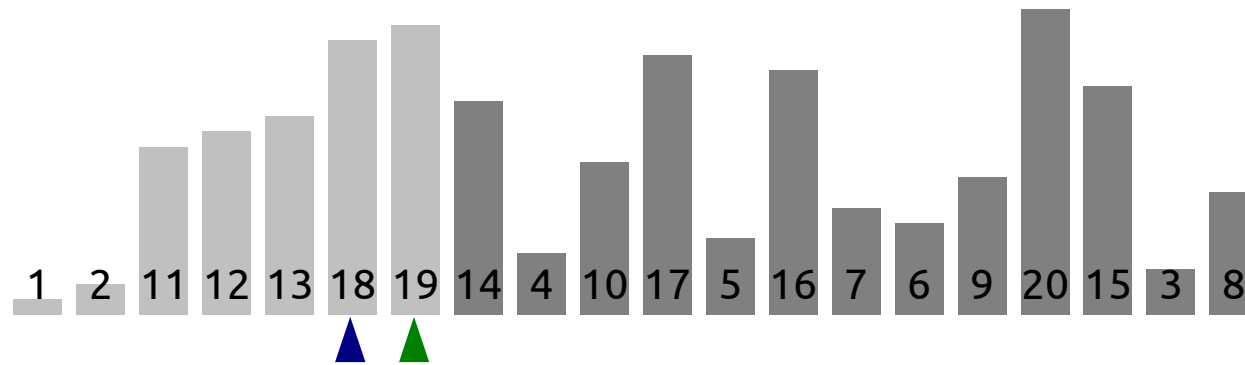
Passo 4:



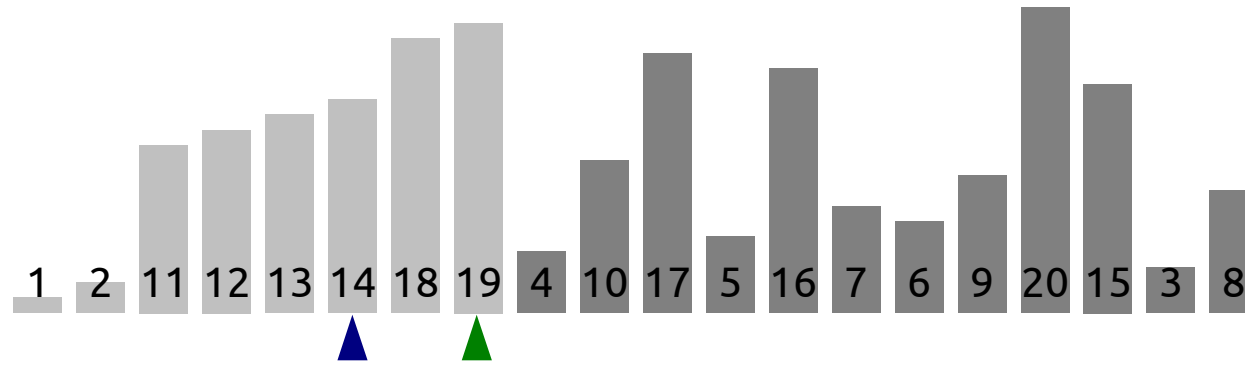
Passo 5:



Passo 6:



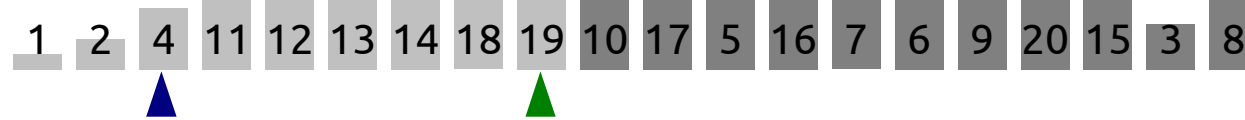
Passo 7:



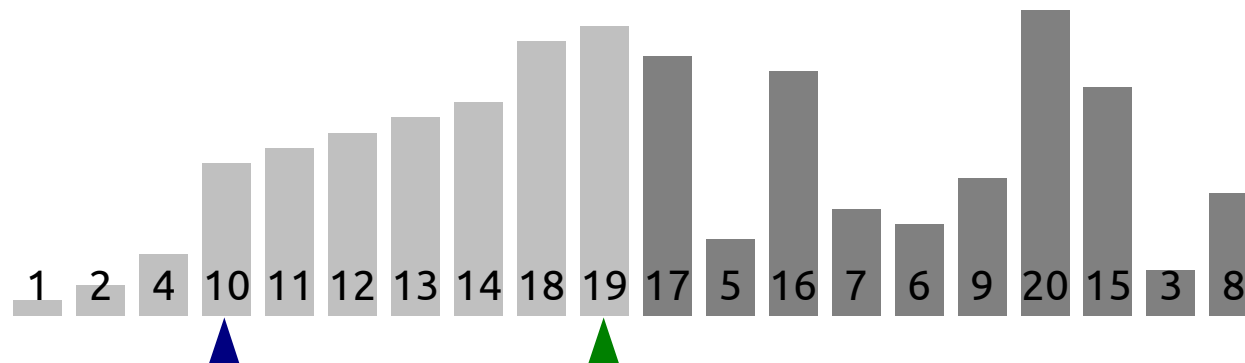
4-7

InsertionSort

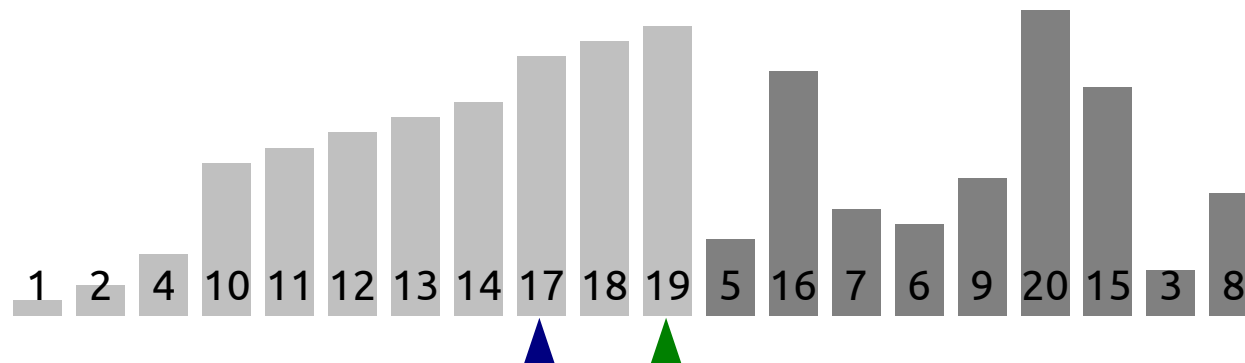
Passo 8:



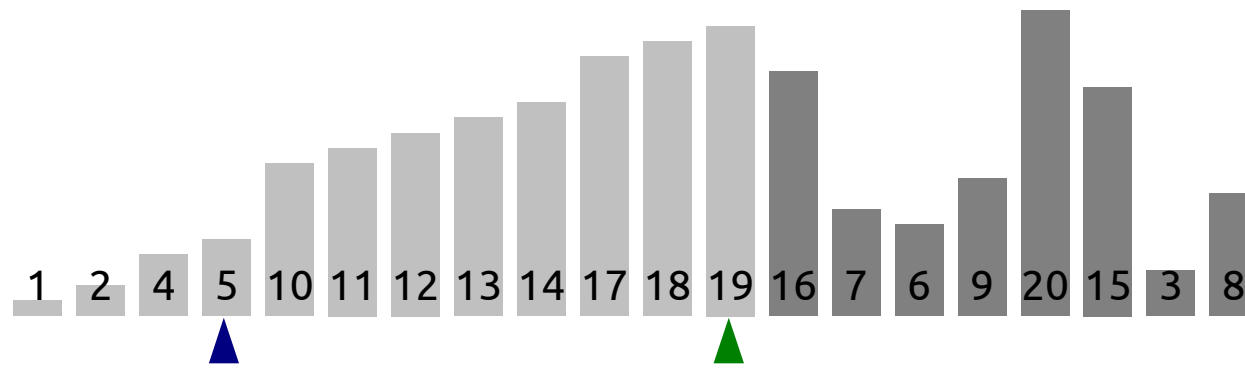
Passo 9:



Passo 10:



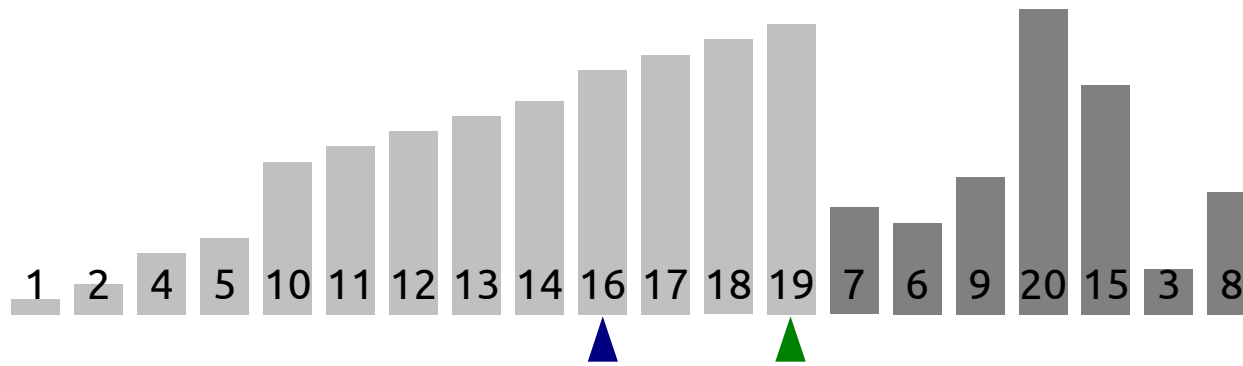
Passo 11:



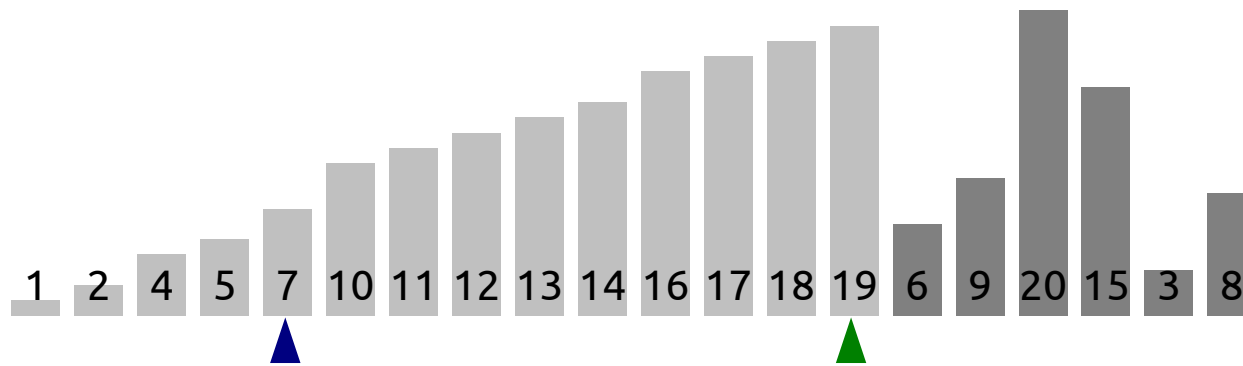
8-11

InsertionSort

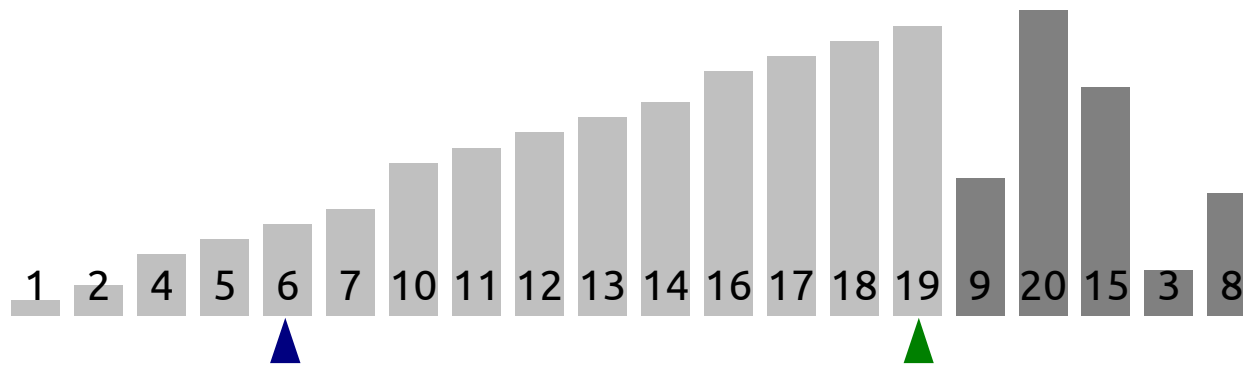
Passo 12:



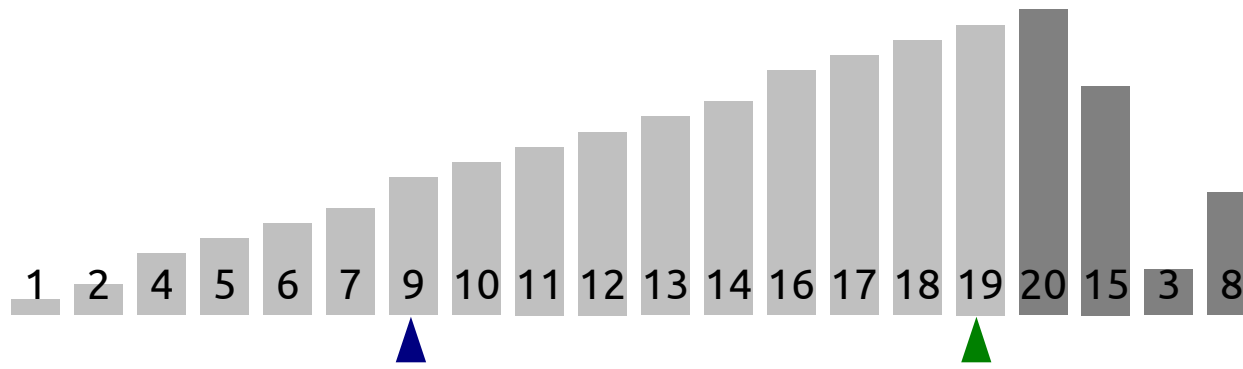
Passo 13:



Passo 14:



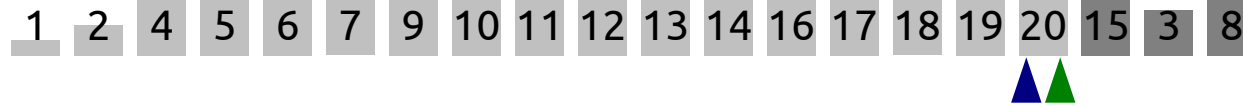
Passo 15:



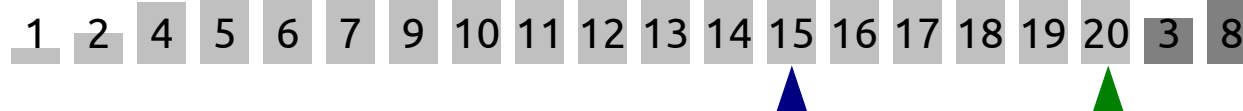
12-15

InsertionSort

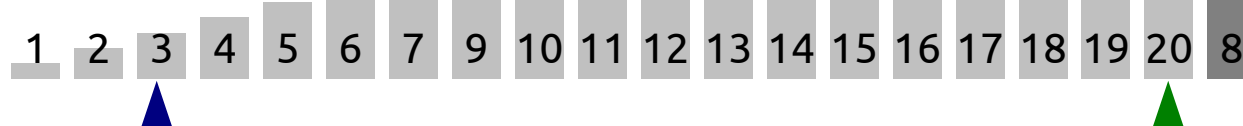
Passo 16:



Passo 17:



Passo 18:



Passo 19:



16-19

Insertion Sort

- Quantas comparações são executadas?
- Quantas trocas são executadas?
- É estável?
- Quantidade de memória?

```
template <class Item>
void insertion(Item vetor[], int n)
{
    for (int i = 1; i < n; i++)
    {
        int j = i;
        while ( j > 0 && vetor[j-1] > vetor[j])
        {
            swap(vetor[j-1], vetor[j]);
            j--;
        }
    }
}
```

Insertion Sort

- Melhor caso: vetor já ordenado
 - Comparações: **linear**
 - Trocas: **constante**

$i=1,2,3,4,\dots,n-1$

Nunca é executado!

```
template <class Item>
void insertion(Item vetor[], int n)
{
    for (int i = 1; i < n; i++)
    {
        int j = i;
        while ( j > 0 && vetor[j-1] > vetor[j])
        {
            swap(vetor[j-1], vetor[j]);
            j--;
        }
    }
}
```

Insertion Sort

- Pior caso: vetor inversamente ordenado
 - Comparações: **quadrático**
 - Trocas: **quadrático**

$i=1,2,3,4,\dots,n-1$

$2,3,4,5,6,7, \dots, n$

$1,2,3,4, \dots, n-1$

```
template <class Item>
void insertion(Item vetor[], int n)
{
    for (int i = 1; i < n; i++)
    {
        int j = i;
        while ( j > 0 && vetor[j-1] > vetor[j])
        {
            swap(vetor[j-1], vetor[j]);
            j--;
        }
    }
}
```

Insertion Sort

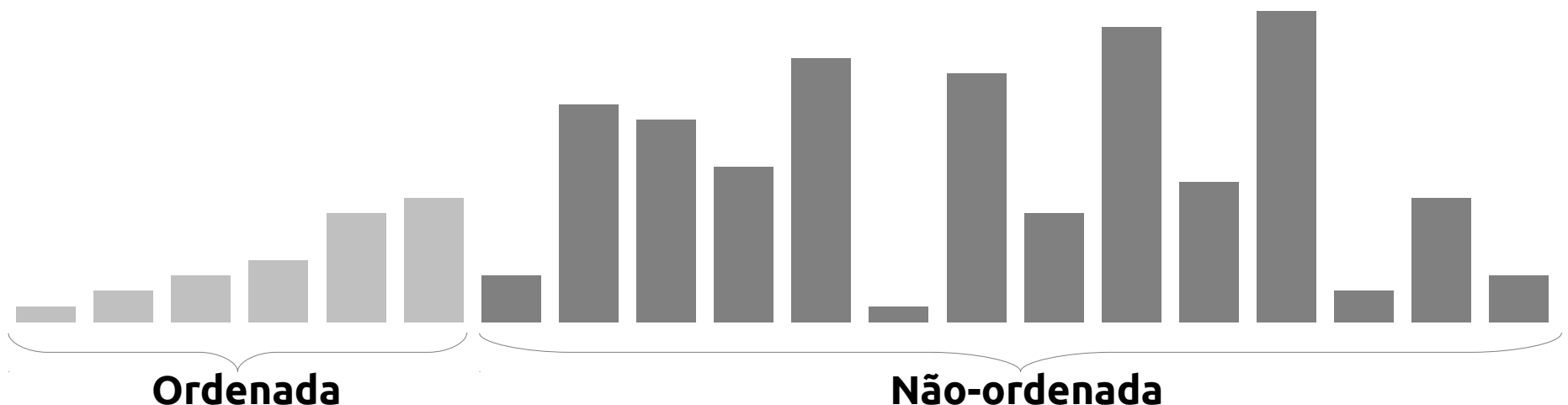
- Os valores dos dados interferem na execução do algoritmo
- Crescimento do número de comparações em relação ao tamanho de entrada:
 - **linear** (no melhor caso)
 - **quadrático** (no pior caso)
- Crescimento do número de trocas em relação ao tamanho de entrada:
 - **constante** (no melhor caso)
 - **quadrático** (no pior caso)
- Crescimento do uso de memória em relação ao tamanho da entrada: **constante**
- E o caso médio?
- O algoritmo é estável?

Insertion Sort

- Se não conhecemos nada das possíveis entradas (valores aleatórios)
 - Assumimos a média, logo:
 - Comparações: **quadrático**
 - Trocas: **quadrático**
- Se conhecemos: depende!
 - Imagine entradas quase sempre ordenadas ...
- O algoritmo é estável? Sim! (Porque?)

Bubble Sort

- Aplicação sucessiva de comparações entre vizinhos (na prática também separa a sequência em duas partes: ordenada e não-ordenada)
- Iteração: percorrer toda a sequência não-ordenada comparando todos os vizinhos e trocando de posição quando necessário, no final, o menor elemento poderá ser concatenado na sequência ordenada



Bubble Sort

```
template <class Item>
void bubble(Item vetor[], int n)
{
    bool swapped;
    int i = 0;
    do {
        swapped = false;
        for( int j = n-1; j > i; j--)
        {
            if ( vetor[j-1] > vetor[j])
            {
                swap(vetor[j-1], vetor[j]);
                swapped = true;
            }
        }
        i++;
    } while( swapped && i < n );
}
```

i

controla a iteração,
índice da sequência
ordenada

j

controla a varredura na
sequência não-ordenada

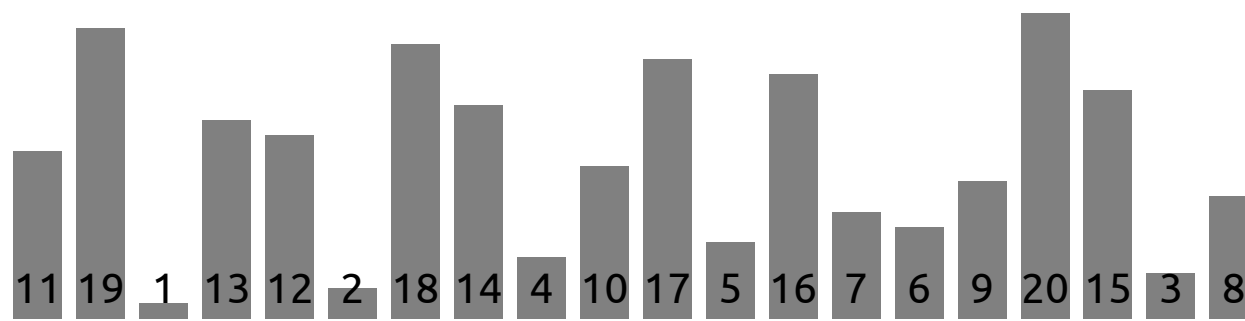
swapped

marca se houve ou não
trocas

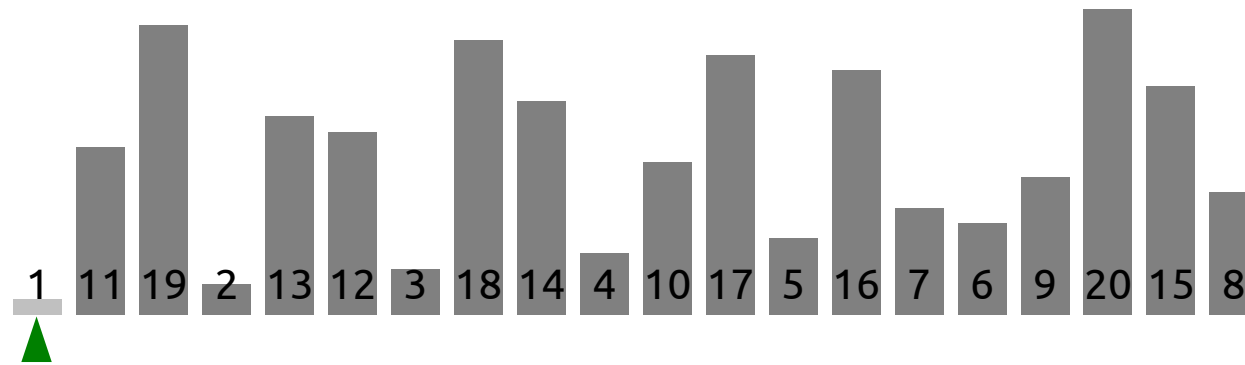
```
template <class Item>
void swap(Item &A, Item &B)
{ Item t = A ; A = B; B = t; }
```

BubbleSort

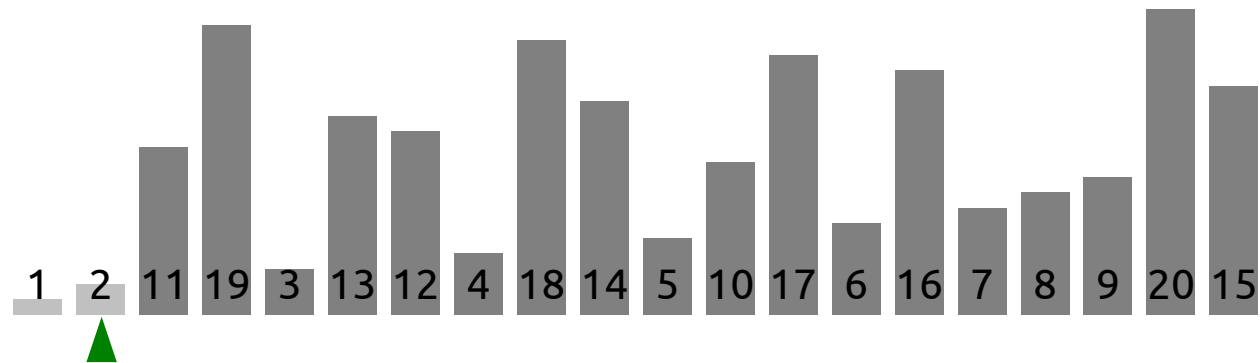
Entrada:



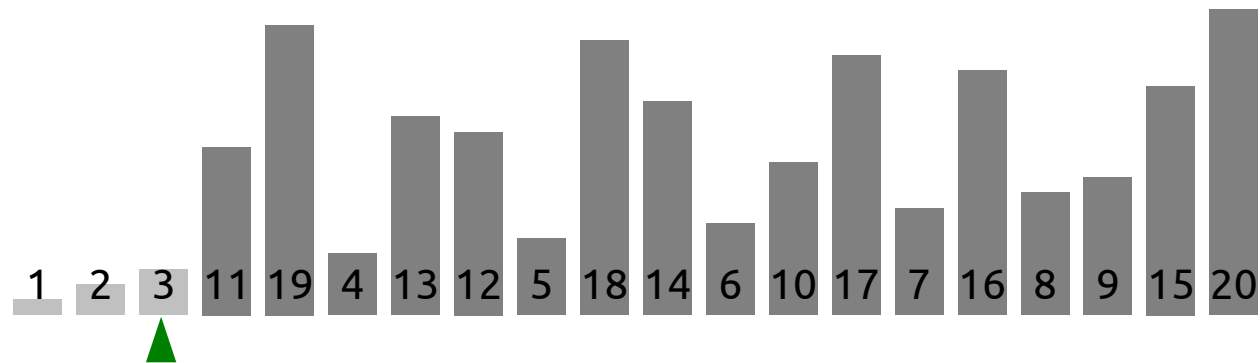
Passo 1:



Passo 2:



Passo 3:



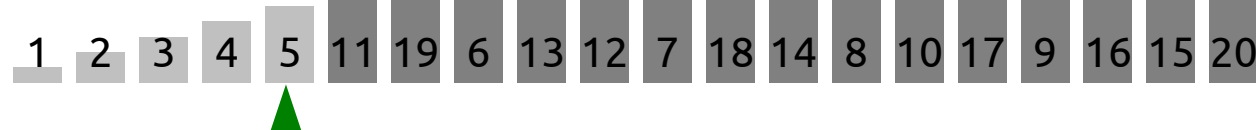
0-3

BubbleSort

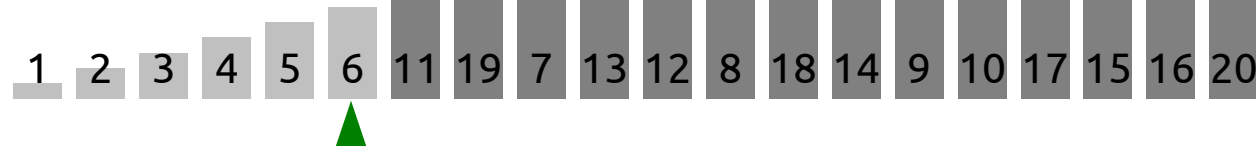
Passo 4:



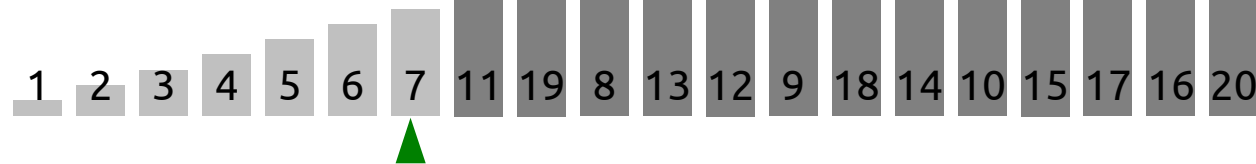
Passo 5:



Passo 6:



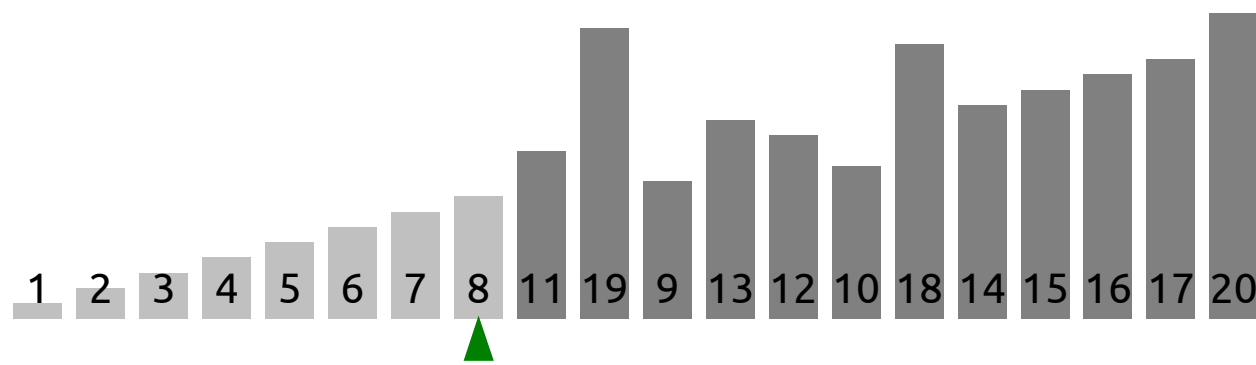
Passo 7:



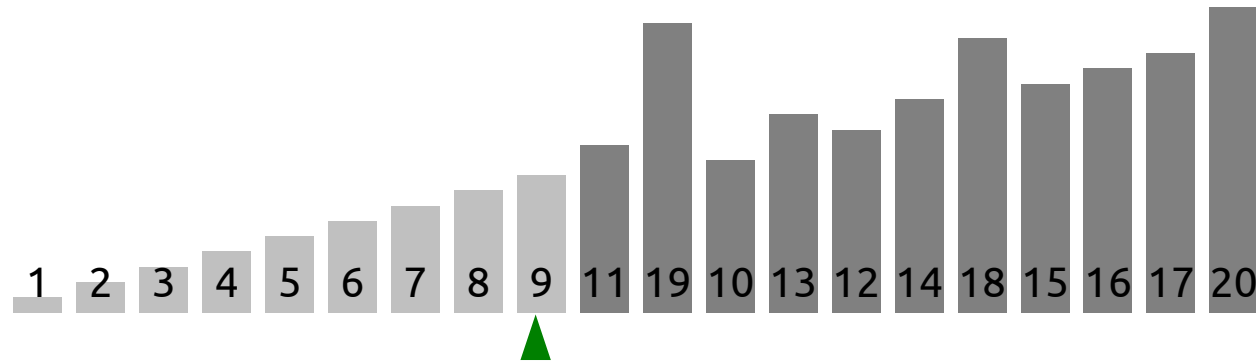
4-7

BubbleSort

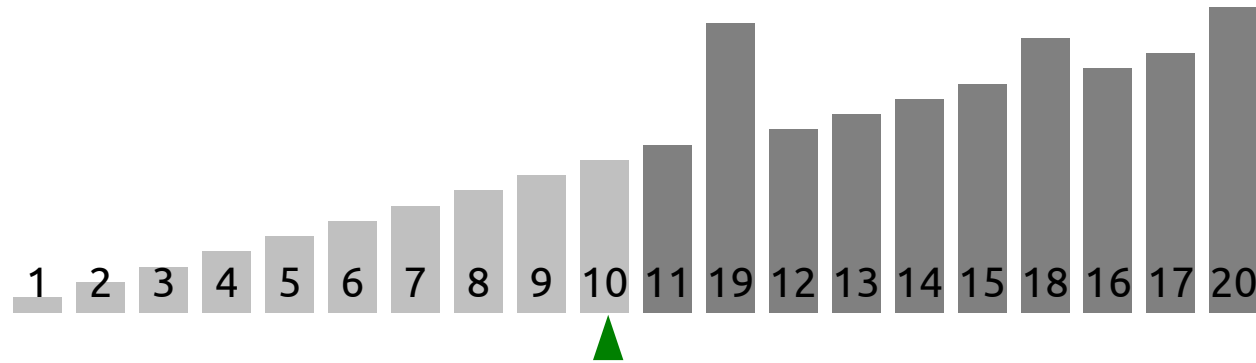
Passo 8:



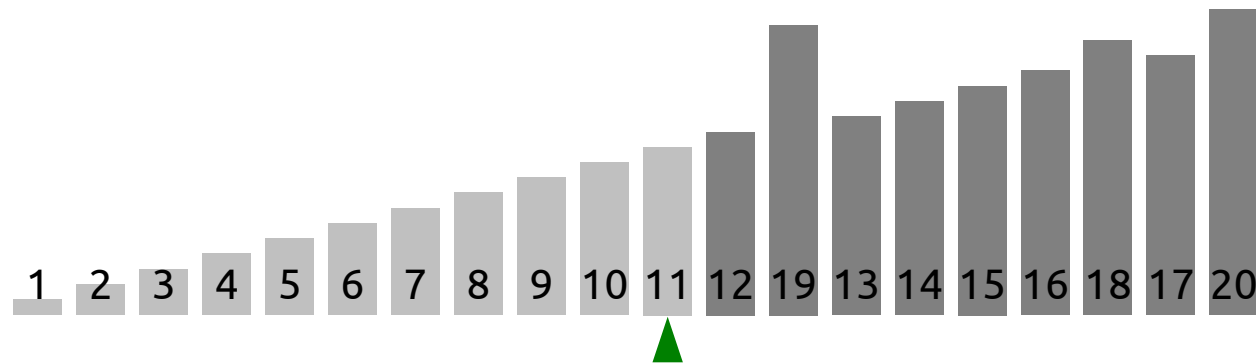
Passo 9:



Passo 10:



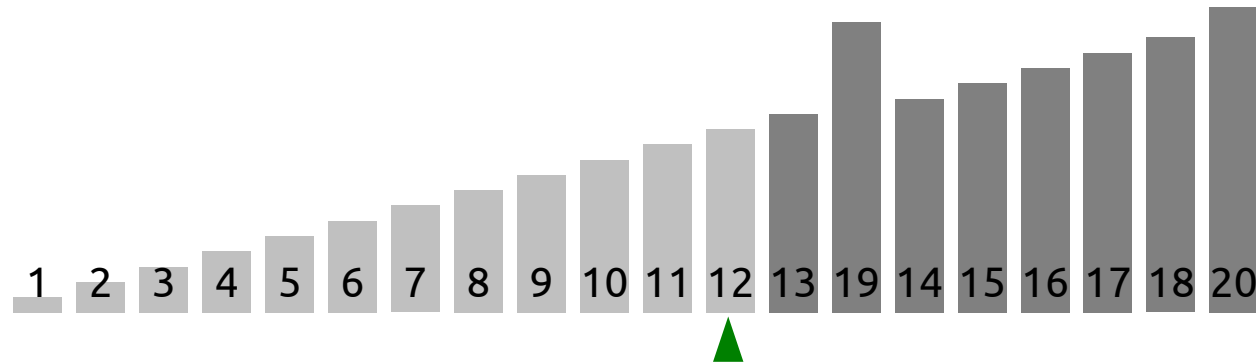
Passo 11:



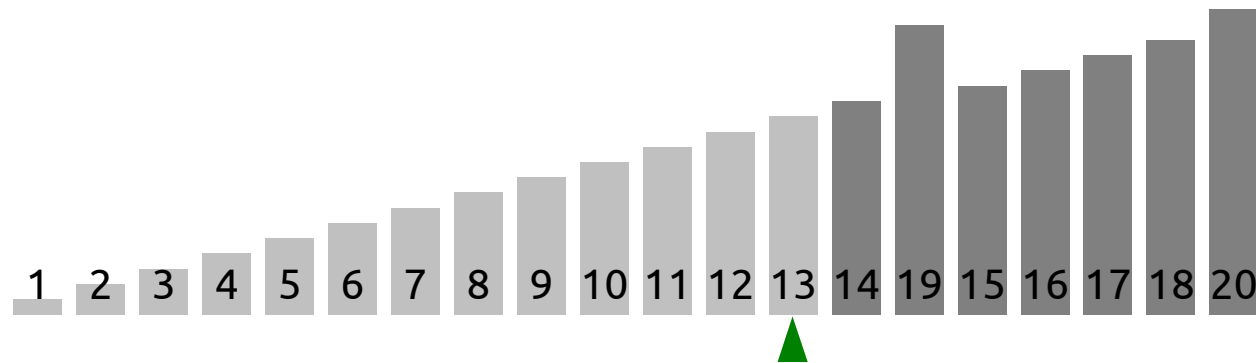
8-11

BubbleSort

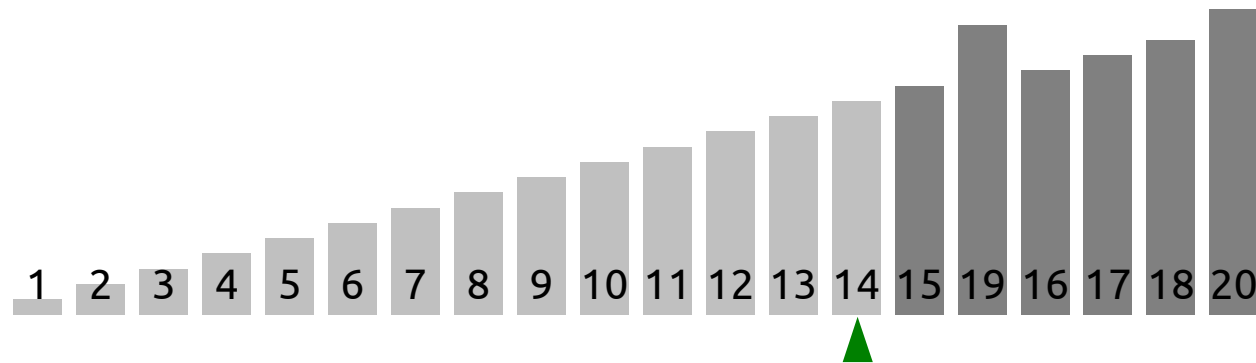
Passo 12:



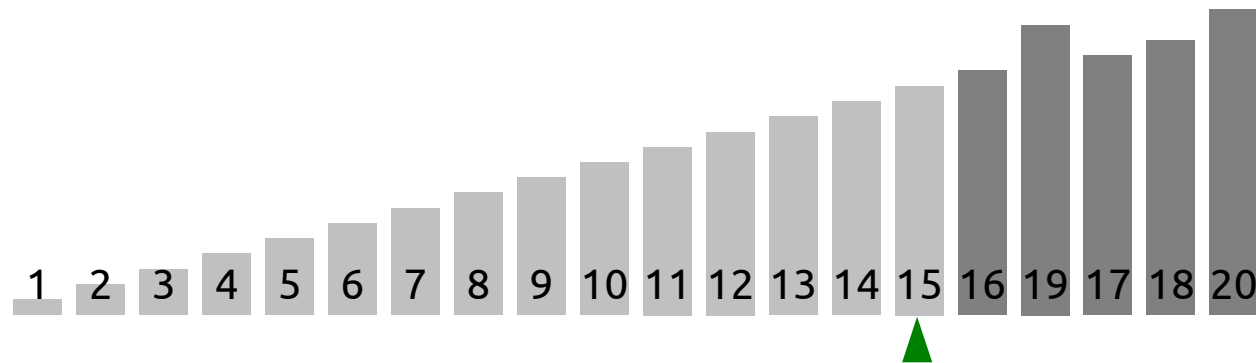
Passo 13:



Passo 14:



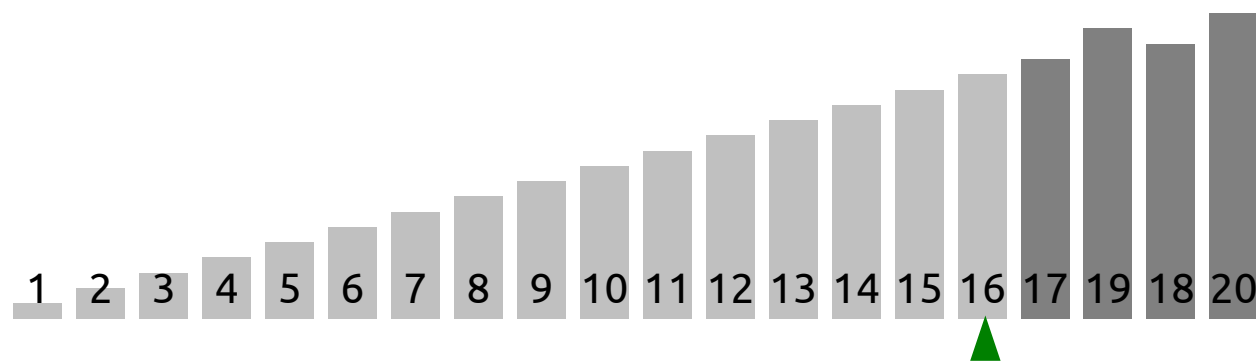
Passo 15:



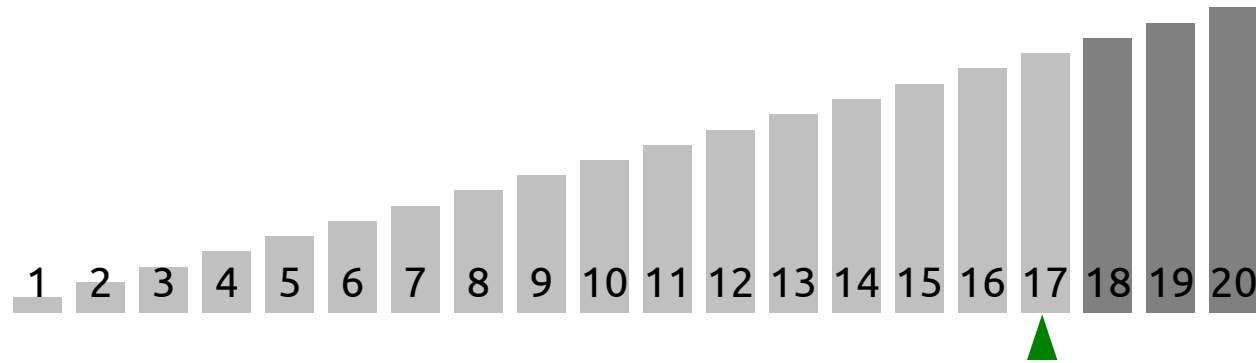
12-15

BubbleSort

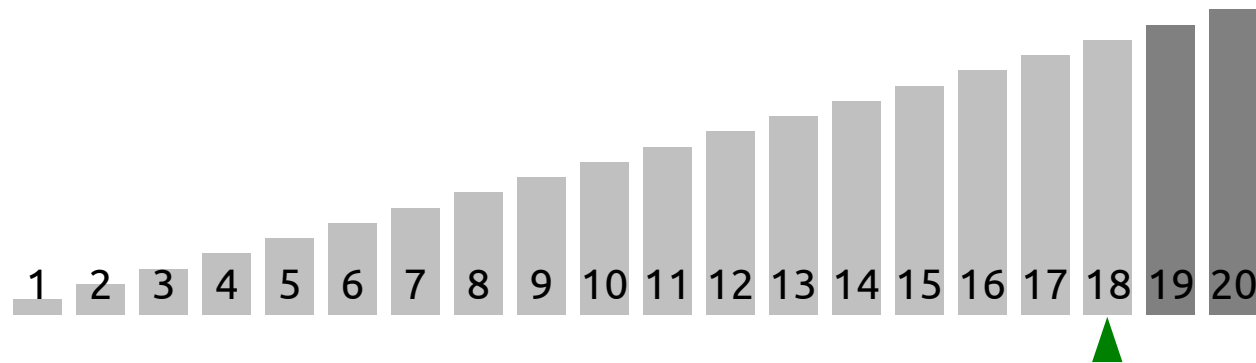
Passo 16:



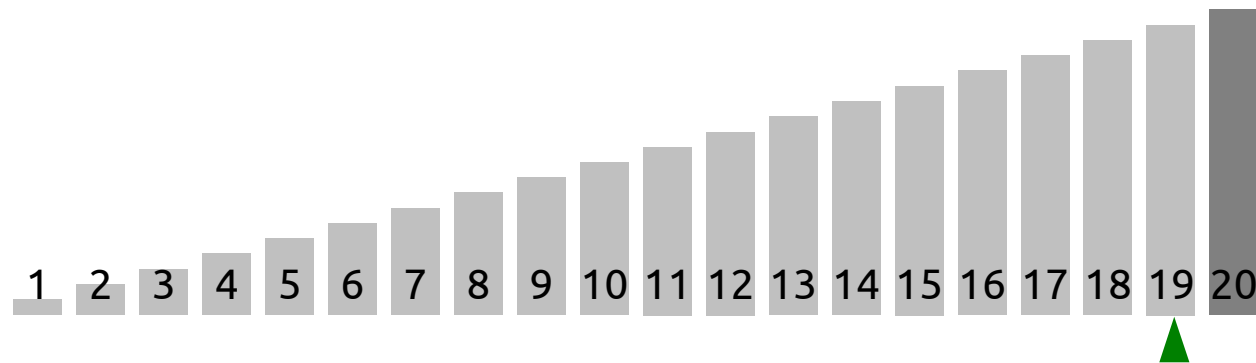
Passo 17:



Passo 18:



Passo 19:



12-15

Bubble Sort

- Quantas comparações são executadas?
- Quantas trocas são executadas?
- É estável?
- Quantidade de memória?

```
template <class Item>
void bubble(Item vetor[], int n)
{
    bool swapped;
    int i = 0;
    do {
        swapped = false;
        for( int j = n-1; j > i; j--)
        {
            if ( vetor[j-1] > vetor[j])
            {
                swap(vetor[j-1], vetor[j]);
                swapped = true;
            }
        }
        i++;
    } while( swapped && i < n );
}
```


Bubble Sort

- Melhor caso: vetor já ordenado
 - Comparações: **linear**
 - Trocas: **constante**

Apenas $i=0$

$j = n-1, n-2, \dots, 1$

Nunca é executado!

```
template <class Item>
void bubble(Item vetor[], int n)
{
    bool swapped;
    int i = 0;
    do {
        swapped = false;
        for( int j = n-1; j > i; j--)
        {
            if ( vetor[j-1] > vetor[j] )
            {
                swap(vetor[j-1], vetor[j]);
                swapped = true;
            }
        }
        i++;
    } while( swapped && i < n );
}
```

Bubble Sort

- Pior caso: vetor inversamente ordenado
 - Comparações: **quadrático**
 - Trocas: **quadrático**

$i=0,1,2,3,\dots,n-1$

$n-1,n-2,n-3,\dots,1$

comparação
sempre verdadeira

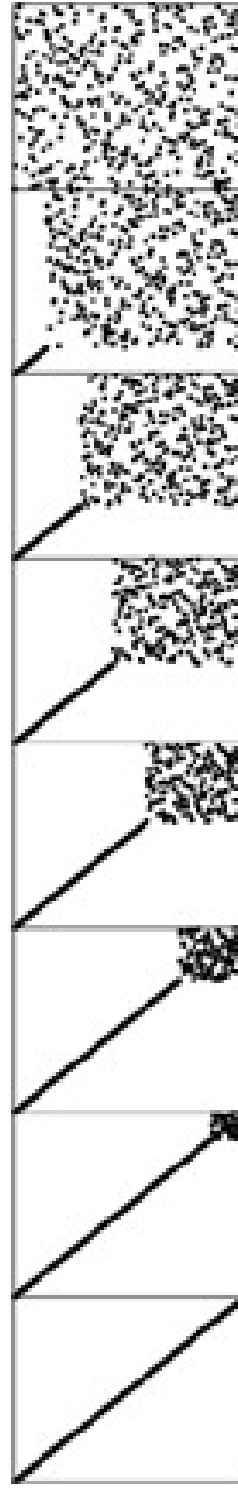
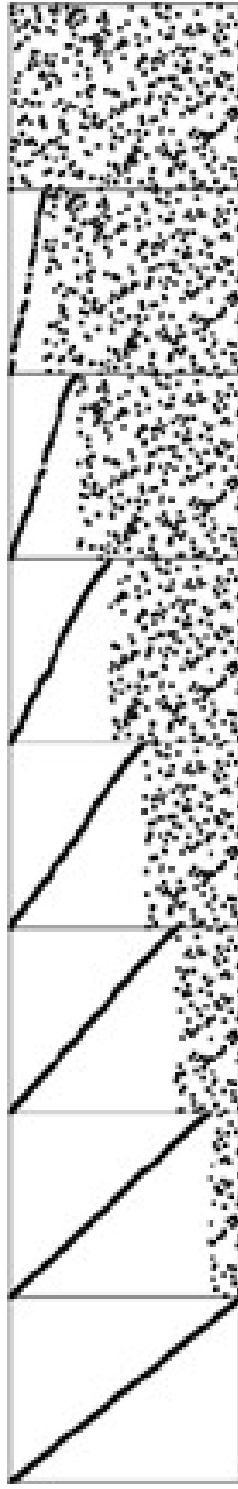
```
template <class Item>
void bubble(Item vetor[], int n)
{
    bool swapped;
    int i = 0;
    do {
        swapped = false;
        for( int j = n-1; j > i; j--)
        {
            if ( vetor[j-1] > vetor[j])
            {
                swap(vetor[j-1], vetor[j]);
                swapped = true;
            }
        }
        i++;
    } while( swapped && i < n );
}
```

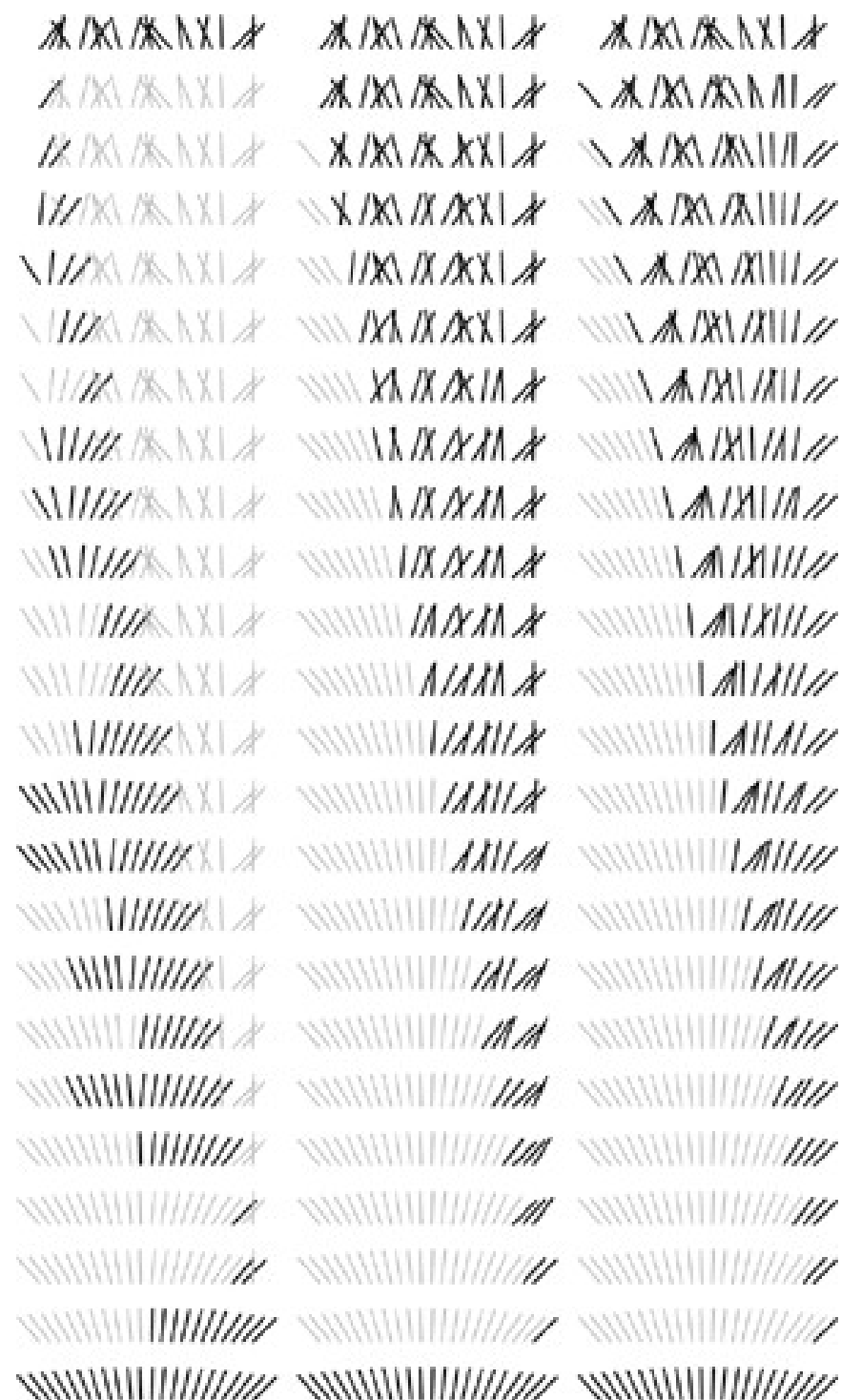
Bubble Sort

- Os valores dos dados interferem na execução do algoritmo
- Crescimento do número de comparações em relação ao tamanho de entrada:
 - **linear** (no melhor caso)
 - **quadrático** (no pior caso)
- Crescimento do número de trocas em relação ao tamanho de entrada:
 - **constante** (no melhor caso)
 - **quadrático** (no pior caso)
- Crescimento do uso de memória em relação ao tamanho da entrada: **constante**
- E o caso médio?
- O algoritmo é estável?

Bubble Sort

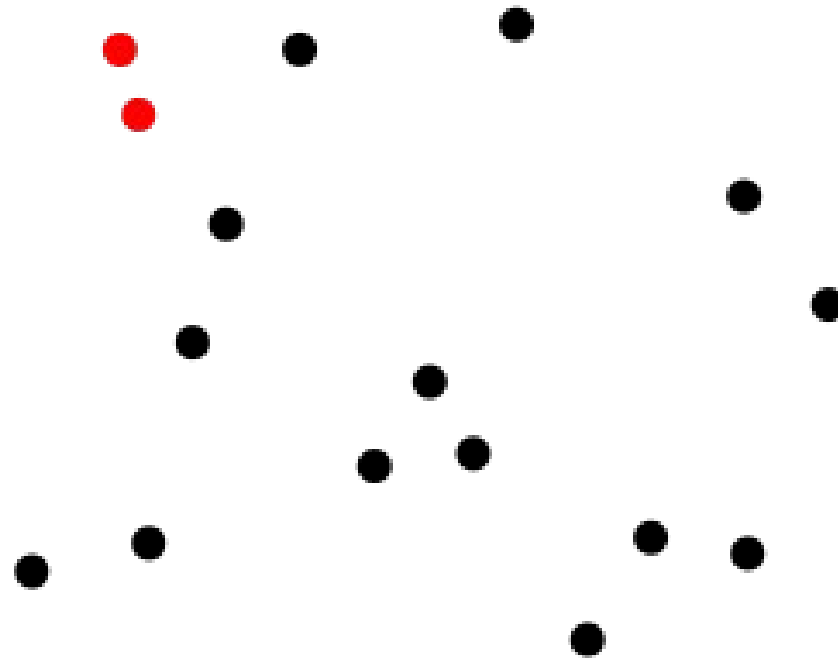
- Se não conhecemos nada das possíveis entradas (valores aleatórios)
 - Assumimos a média, logo:
 - Comparações: **quadrático**
 - Trocas: **quadrático**
- Se conhecemos: depende!
 - Imagine entradas quase sempre ordenadas ...
- O algoritmo é estável? Sim! (Porque?)





Exemplo 4

- Pontos mais próximos:



Exemplo 4

- Pontos mais próximos:

ALGORITHM *BruteForceClosestPair(P)*

//Finds distance between two closest points in the plane by brute force

//Input: A list P of n ($n \geq 2$) points $p_1(x_1, y_1), \dots, p_n(x_n, y_n)$

//Output: The distance between the closest pair of points

$d \leftarrow \infty$

for $i \leftarrow 1$ **to** $n - 1$ **do**

for $j \leftarrow i + 1$ **to** n **do**

$d \leftarrow \min(d, \text{sqrt}((x_i - x_j)^2 + (y_i - y_j)^2))$ //sqrt is square root

return d

