

# Table of Contents

Learn Sorting Algorithm .....	2
Selection Sort .....	13
Implementação .....	17
Por que o Selection Sort NÃO é Estável? .....	22
Vantagens vs. Desvantagens .....	27
Quando Usar? .....	28
Comparação com Outros Algoritmos .....	29
Variações do Selection Sort .....	30
Exercícios Práticos .....	31
Conclusão .....	41
Bubble Sort .....	42

# Learn Sorting Algorithm

## Entendendo Algoritmos de Ordenação

Algoritmos de ordenação funcionam como organizar uma bagunça - você pega uma lista desordenada (como anões de tamanhos diferentes) e os coloca na ordem correta (do menor para o maior ou vice-versa).

Imagine esta fila de anões:



Fila de anão

## Conceitos Fundamentais Antes de Começar

Antes de mergulharmos no algoritmo, vamos entender alguns conceitos básicos:

### O que é um Algoritmo?

Um **algoritmo** é simplesmente uma lista de instruções passo a passo para resolver um problema. É como uma receita de bolo - você segue os passos na ordem certa para chegar ao resultado desejado.



**Exemplo prático:** Para fazer um sanduíche, o algoritmo seria:

1. Pegue duas fatias de pão
2. Passe manteiga em uma fatia
3. Coloque o recheio
4. Feche com a outra fatia

## O que é Iteração?

**Iteração** é quando repetimos um conjunto de passos várias vezes. É como quando você escova os dentes - faz o mesmo movimento várias vezes até limpar todos os dentes.

Em programação, quando dizemos:

- "Iteração 1" = primeira vez que executamos os passos
- "Iteração 2" = segunda vez que executamos os passos
- E assim por diante...

## O que é Comparação?

**Comparar** é verificar qual elemento é maior, menor ou igual ao outro. É como quando você compara a altura de duas pessoas para saber quem é mais alto.

## O Básico do Algoritmo de Ordenação

Nosso algoritmo simples compara os elementos um a um, como quando você organiza suas roupas - pega cada peça e compara com as outras para ver qual é maior ou menor.

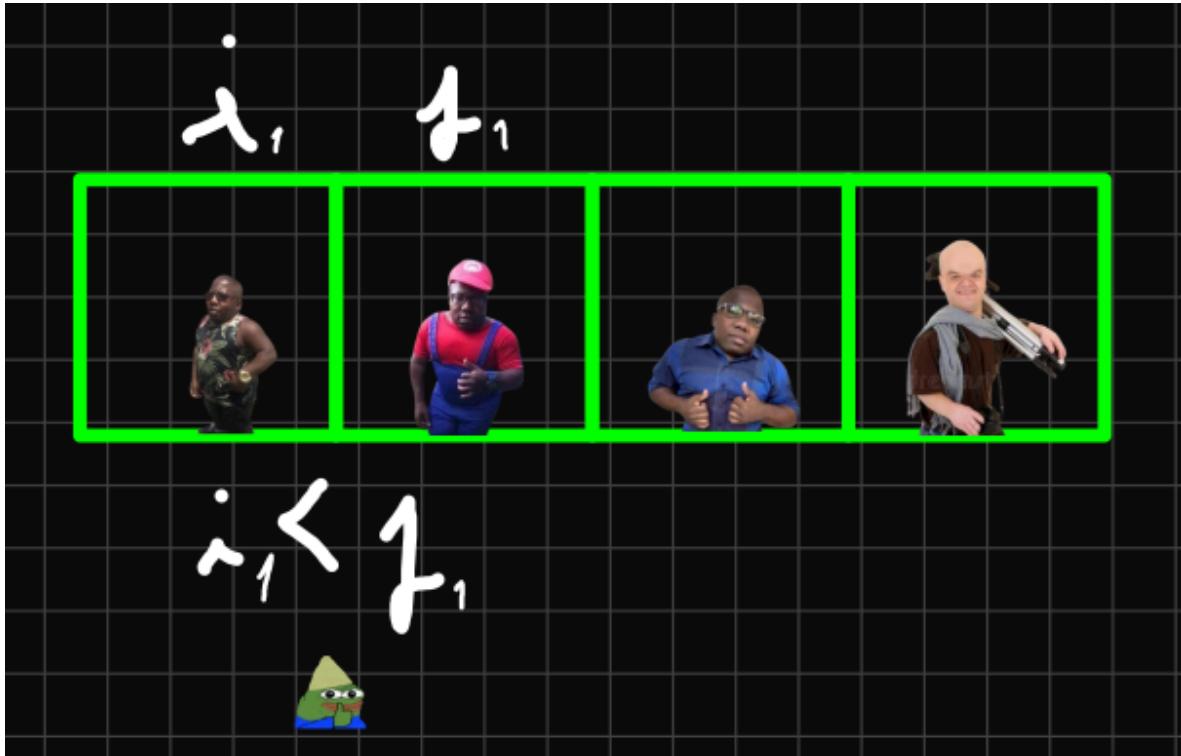
## Como Funciona na Prática

Agora vamos ver como o algoritmo trabalha **iteração por iteração** (ou seja, repetição por repetição):

### Primeira Iteração (1<sup>a</sup> Repetição)

Comparamos o primeiro anão ( $i_1$ ) com o segundo anão ( $j_1$ ).

- **Pergunta:** O primeiro é menor que o segundo?
- **Resposta:** Sim (verdadeiro) → Não preciso trocar nada!

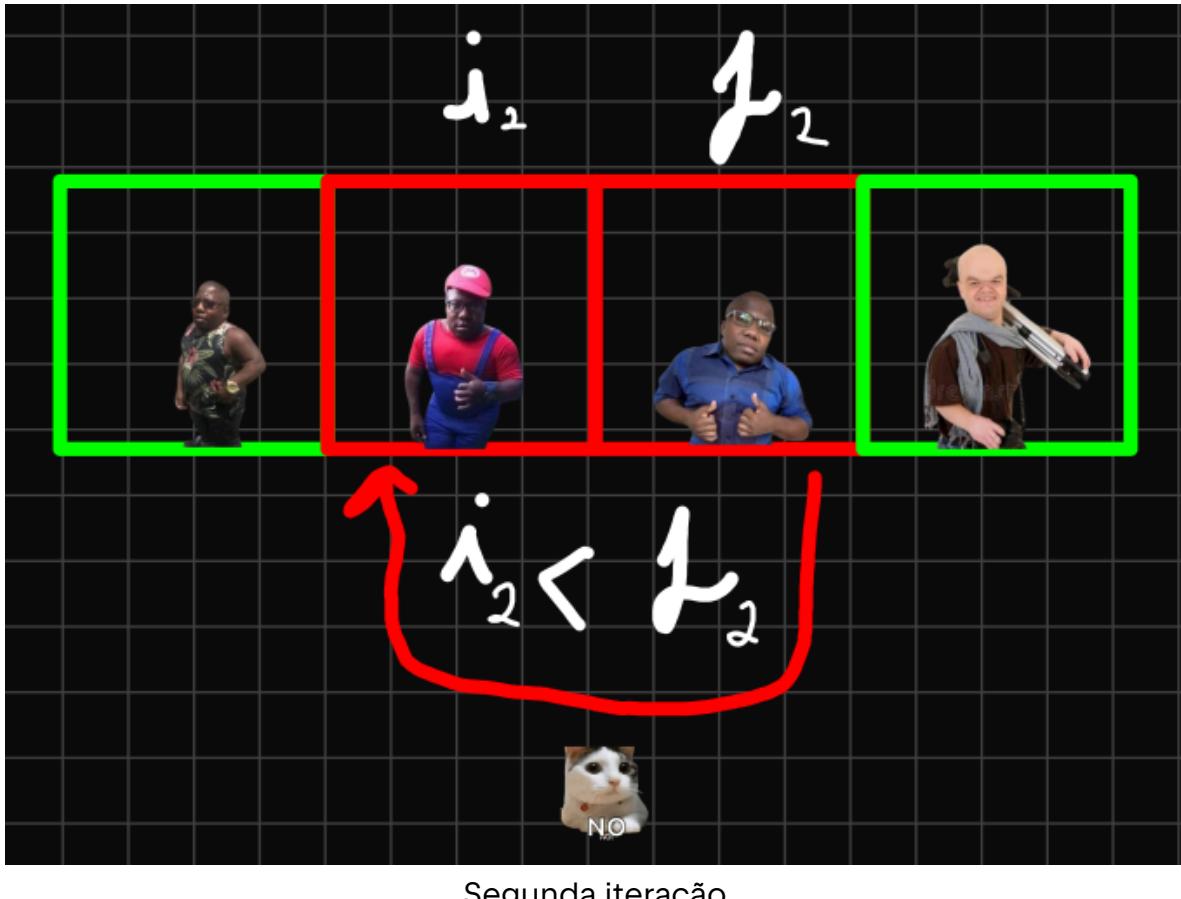


Primeira iteração

## Segunda Iteração (2<sup>a</sup> Repetição)

Agora comparo  $i_2$  com  $j_2$ .

- **Pergunta:** O primeiro é menor que o segundo?
- **Resposta:** Não (falso) → Preciso trocar eles de lugar!



Segunda iteração

### Terceira Iteração (3<sup>a</sup> Repetição)

Comparo  $i_3$  com  $j_3$ .

- **Pergunta:** O primeiro é menor que o segundo?
- **Resposta:** Sim (verdadeiro) → Não preciso trocar nada!

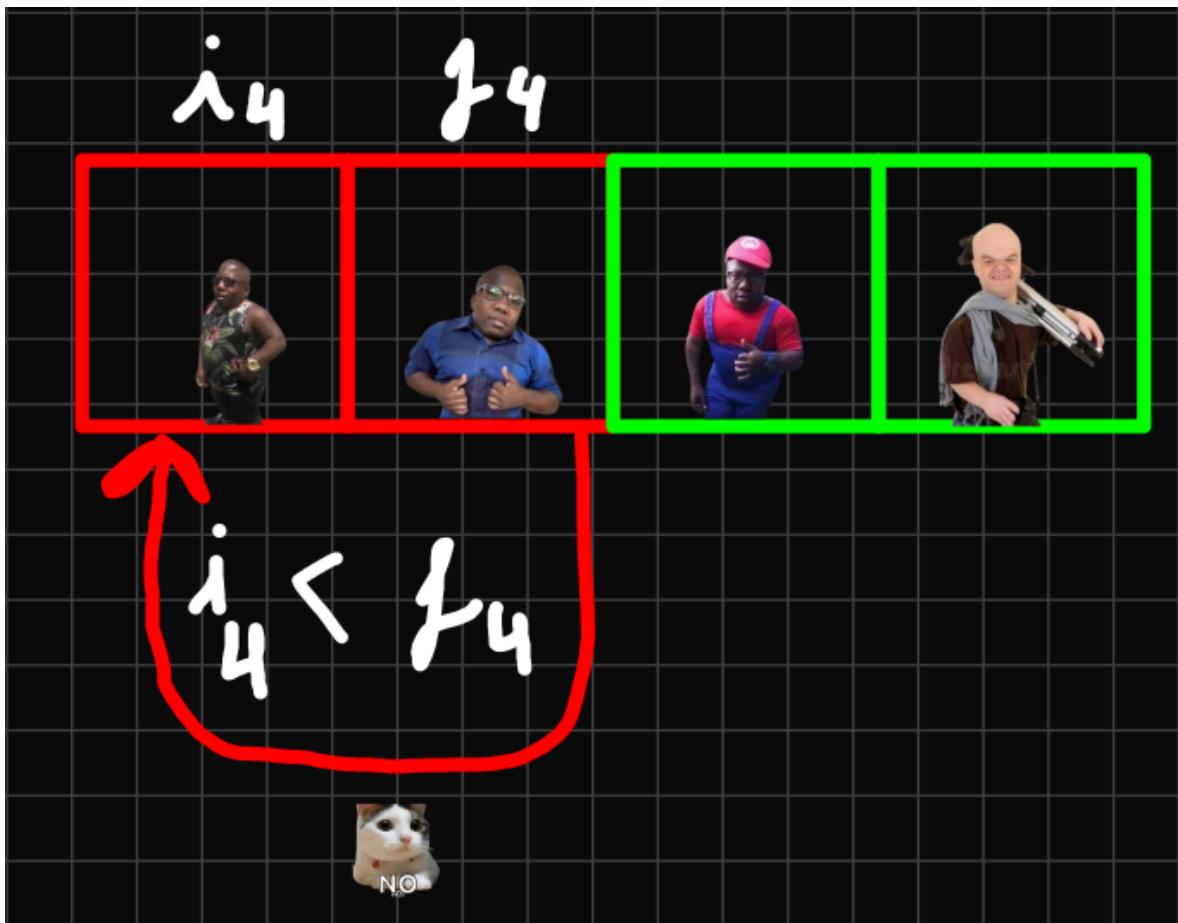


Terceira iteração

#### Quarta Iteração (4<sup>a</sup> Repetição)

Comparo  $i_4$  com  $j_4$ .

- **Pergunta:** O primeiro é menor que o segundo?
- **Resposta:** Não (falso) → Preciso trocar eles de lugar!



### Quinta Iteração (5<sup>a</sup> e Última Repetição)

Agora todos os elementos já estão na ordem correta!

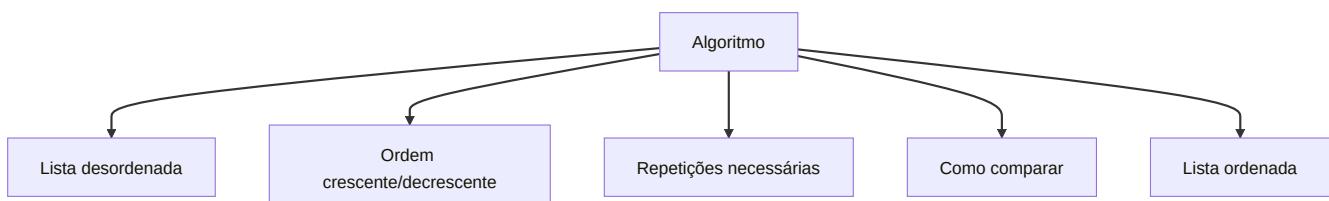


**⚠ Resumo:** A cada iteração, fazemos uma **pergunta** (comparação) e tomamos uma **decisão** (trocar ou não trocar). Repetimos isso até que tudo esteja organizado!

## O Que Todo Algoritmo de Ordenação Precisa

Resumindo, todo algoritmo de ordenação possui:

1. **Entrada:** A lista desordenada
2. **Tipo de ordem:** Crescente ou decrescente
3. **Iterações:** Quantidade de repetições do processo
4. **Comparações:** Critério para comparar os elementos
5. **Saída:** A lista organizada



# Partindo para o Código

Agora vamos ver como transformar nossa lógica em código C++. Vou explicar cada parte:

## Conceitos Básicos Antes do Código

### O que é um Array?

Um **array** é como uma fileira de caixas numeradas onde guardamos valores. Cada caixa tem um número (índice) e pode guardar um valor.

Array: [5] [2] [8] [1] [9]

Índice: 0 1 2 3 4

### O que é um Loop For?

Um **loop for** é uma forma de repetir uma ação várias vezes. É como dizer: "faça isso 10 vezes" ou "faça isso para cada elemento da lista".

```
for (int i = 0; i < 5; i++) {  
    // Este código vai executar 5 vezes  
    // i vai valer: 0, 1, 2, 3, 4  
}
```

## O Código Completo Explicado

```
#include <iostream> // Para usar cout e cin (entrada e saída)  
#include <cstdlib> // Para usar rand() (números aleatórios)  
#define MAX_SIZE 100 // Define o tamanho máximo do array  
using namespace std;  
  
int main()  
{  
    // PARTE 1: DECLARAÇÃO DAS VARIÁVEIS  
    int n; // Quantos números vamos ordenar  
    int array[MAX_SIZE]; // Nossa lista de números (o array)  
  
    // PARTE 2: ENTRADA DE DADOS  
    cout << "Digite quantos números você quer ordenar: " << endl;
```

```

cin >> n; // Lê a resposta do usuário

// PARTE 3: PREENCHIMENTO COM NÚMEROS ALEATÓRIOS
cout << "Gerando " << n << " números aleatórios..." << endl;
for (int i = 0; i < n; i++) {
    array[i] = rand() % 100; // Gera número entre 0 e 99
}

// PARTE 4: MOSTRA A LISTA BAGUNÇADA
cout << "\nLista original (bagunçada): ";
for (int i = 0; i < n; i++) {
    cout << "[" << array[i] << "] ";
}
cout << endl << endl;

// PARTE 5: O ALGORITMO DE ORDENAÇÃO (O CORAÇÃO DO PROGRAMA!)
cout << "Iniciando ordenação..." << endl;

// Loop externo: seleciona cada elemento (iteração principal)
for (int i = 0; i < n; i++) {

    // Loop interno: compara com os elementos seguintes
    for (int j = i + 1; j < n; j++) {

        // Mostra o que está sendo comparado
        cout << "Comparando: " << array[i] << " com " <<
array[j];

        // AQUI É ONDE A MÁGICA ACONTECE!
        // Se o primeiro for MENOR que o segundo, trocamos
        if (array[i] < array[j]) {
            cout << " → Trocando!" << endl;

            // PROCESSO DE TROCA (como trocar duas cartas de
lugar)
            int temp = array[i]; // Guarda o primeiro valor
            array[i] = array[j]; // Põe o segundo no lugar do
primeiro
    }
}
}

```

```

        array[j] = temp;      // Põe o primeiro no lugar
do segundo
    } else {
        cout << " → Não precisa trocar" << endl;
    }
}
}

// PARTE 6: MOSTRA O RESULTADO FINAL
cout << "\n" << "=" << endl;
cout << "Lista ordenada (maior para menor): ";
for (int i = 0; i < n; i++) {
    cout << "[" << array[i] << "] ";
}
cout << endl;

return 0; // Programa terminou com sucesso
}

```

## Pontos Importantes Para Entender:

### 1. Como Funciona o rand()

- `rand()` gera números aleatórios gigantes
- `rand() % 100` limita os números entre 0 e 99 (mais fácil de trabalhar)

### 2. Por que Dois Loops For?

- **Loop externo (i):** Escolhe cada elemento da lista, um por vez
- **Loop interno (j):** Compara esse elemento com todos os outros que vêm depois
- É como comparar cada anão com todos os outros anões

### 3. Como Funciona a Troca (Swap)?

Imagine que você tem duas cartas e quer trocar elas de lugar:

```
int temp = array[i]; // Guardo a primeira carta na mão
array[i] = array[j]; // Ponho a segunda carta no lugar da
primeira
array[j] = temp; // Ponho a primeira carta (que estava na
mão) no lugar da segunda
```

#### 4. Mudando a Ordem

- Para ordem DECRESCENTE (maior → menor): use if (`array[i] < array[j]`)
- Para ordem CRESCENTE (menor → maior): use if (`array[i] > array[j]`)

#### Importante Saber

Este algoritmo se chama **Selection Sort** (Ordenação por Seleção). Ele não é o mais rápido para listas muito grandes, mas é **perfeito para aprender** porque:

- É fácil de entender
- Mostra claramente como funciona a ordenação
- Usa conceitos básicos que você vai usar sempre



Dica: Execute o código algumas vezes com números pequenos (como 5 elementos) para ver exatamente como ele funciona!

# Selection Sort

## O que é Selection Sort?

O **Selection Sort** é um algoritmo de ordenação simples e intuitivo que funciona dividindo o array em duas partes:

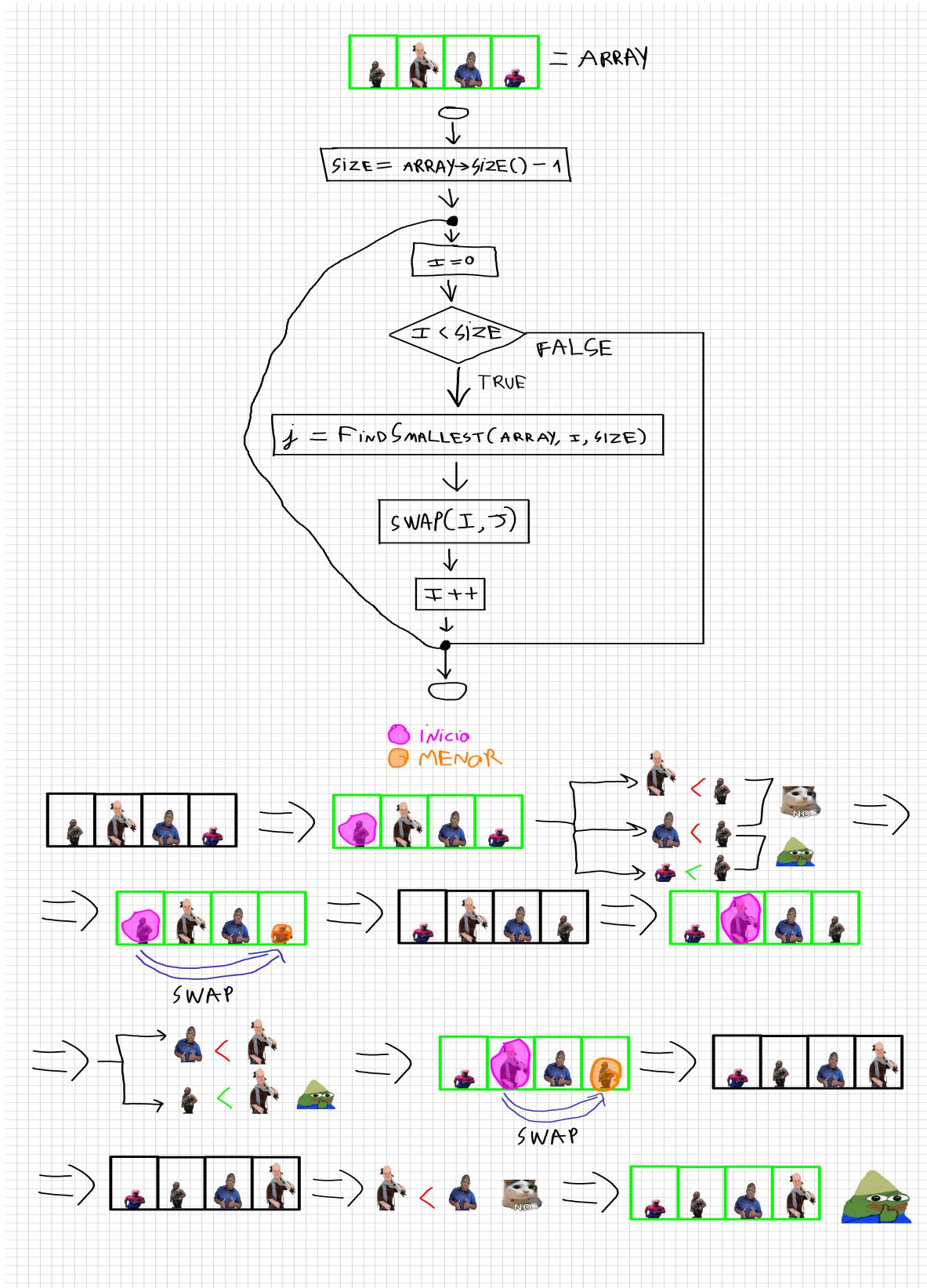
- **Parte ordenada:** Localizada no início do array (inicialmente vazia)
- **Parte não ordenada:** Localizada no final do array (inicialmente todo o array)

## Como funciona?

O algoritmo funciona da seguinte forma:

1. **Encontra o menor elemento** da parte não ordenada
2. **Troca** esse elemento com o primeiro elemento da parte não ordenada
3. **Expande** a parte ordenada em uma posição
4. **Repete** o processo até que todo o array esteja ordenado

## Diagramação



selectionSort\_annotated.png

# Exemplo Visual

Considerando o array: [64, 25, 12, 22, 11]

## Iteração 1:

- Parte ordenada: []
- Parte não ordenada: [64, 25, 12, 22, 11]
- Menor elemento: 11 (posição 4)
- Troca: 11 ↔ 64
- Resultado: [11, 25, 12, 22, 64]

## Iteração 2:

- Parte ordenada: [11]
- Parte não ordenada: [25, 12, 22, 64]
- Menor elemento: 12 (posição 2)
- Troca: 12 ↔ 25
- Resultado: [11, 12, 25, 22, 64]

## Iteração 3:

- Parte ordenada: [11, 12]
- Parte não ordenada: [25, 22, 64]
- Menor elemento: 22 (posição 3)
- Troca: 22 ↔ 25
- Resultado: [11, 12, 22, 25, 64]

## Iteração 4:

- Parte ordenada: [11, 12, 22]
- Parte não ordenada: [25, 64]
- Menor elemento: 25 (já na posição correta)
- Sem troca necessária
- Resultado: [11, 12, 22, 25, 64]

# Implementação

Nossa implementação educativa inclui saídas detalhadas para ajudar no aprendizado.

Você pode encontrar uma implementação completa e educativa do Selection Sort em:

## Função Principal - Selection Sort Algorithm

```
void selectionSortAlgorithm(int dataArray[], int arraySize)
{
    cout << "======" << endl;
    cout << "STARTING SELECTION SORT ALGORITHM" << endl;
    cout << "======" << endl;
    cout << "Initial array: ";
    for(int displayIndex = 0; displayIndex < arraySize;
displayIndex++) {
        cout << dataArray[displayIndex] << " ";
    }
    cout << endl << endl;

    int totalNumberOfSwaps = 0;

    for (int currentPosition = 0; currentPosition < arraySize - 1;
currentPosition++)
    {
        cout << ">>> ITERATION " << (currentPosition + 1) << "
<<<" << endl;
        cout << "Looking for smallest element for position " <<
currentPosition << endl;

        int smallestElementIndex =
findSmallestElementIndex(dataArray, currentPosition, arraySize -
1);

        if (currentPosition != smallestElementIndex) {
            cout << "Smallest element is at position " <<
smallestElementIndex << ", need to swap with position " <<
```

```

 currentPosition << endl;
         swapElements(dataArray, currentPosition,
smallestElementIndex);
         totalNumberOfSwaps++;
} else {
    cout << "Smallest element is already in correct
position (" << currentPosition << "), no swap needed" << endl;
}

cout << "Array state after iteration " << (currentPosition
+ 1) << ":" ;
for(int displayIndex = 0; displayIndex < arraySize;
displayIndex++) {
    if (displayIndex <= currentPosition) {
        cout << "[" << dataArray[displayIndex] << "] " ; // 
Already sorted elements
    } else {
        cout << dataArray[displayIndex] << " " ;           // 
Not yet sorted elements
    }
}
cout << endl;
cout << "Elements sorted: " << (currentPosition + 1) <<
"/" << arraySize << endl;
cout << "-----" << endl;
}

cout << "===== " << endl;
cout << "SELECTION SORT ALGORITHM COMPLETED!" << endl;
cout << "Total number of swaps performed: " <<
totalNumberOfSwaps << endl;
cout << "===== " << endl;
}

```

## Função para Encontrar o Menor Elemento

```

int findSmallestElementIndex(int dataArray[], int startIndex, int
endIndex)
{
    cout << " --- Searching for smallest element in range [" <<
startIndex << ", " << endIndex << "] ---" << endl;
    int currentIndex = startIndex;
    int smallestIndex = currentIndex;
    int numberofComparisons = 0;

    cout << "Initial element for comparison: dataArray[" <<
smallestIndex << "] = " << dataArray[smallestIndex] << endl;

    while (currentIndex <= endIndex)
    {
        cout << "Comparing dataArray[" << currentIndex << "] = "
<< dataArray[currentIndex] << " with current smallest dataArray["
<< smallestIndex << "] = " << dataArray[smallestIndex];
        numberofComparisons++;

        if (dataArray[currentIndex] < dataArray[smallestIndex])
        {
            cout << " -> " << dataArray[currentIndex] << " is
smaller! New smallest element found at position " << currentIndex
<< endl;
            smallestIndex = currentIndex;
        }
        else
        {
            cout << " -> " << dataArray[currentIndex] << " >= " <<
dataArray[smallestIndex] << ", keeping current smallest" << endl;
        }

        currentIndex++;
    }

    cout << "Smallest element found: " << dataArray[smallestIndex]
<< " at position " << smallestIndex << " (after " <<

```

```
    numberOfComparisons << " comparisons)" << endl;
    return smallestIndex;
}
```

## Função para Trocar Elementos

```
void swapElements(int dataArray[], int firstPosition, int
secondPosition)
{
    cout << " -> Swapping elements: " << dataArray[firstPosition]
<< " (position " << firstPosition << ") <-> " <<
dataArray[secondPosition] << " (position " << secondPosition <<
")" << endl;
    int temporaryValue = dataArray[firstPosition];
    dataArray[firstPosition] = dataArray[secondPosition];
    dataArray[secondPosition] = temporaryValue;
    cout << " -> After swap: position " << firstPosition << " = "
<< dataArray[firstPosition] << ", position " << secondPosition <<
" = " << dataArray[secondPosition] << endl;
}
```

## Características do Algoritmo

### Complexidade de Tempo

- **Melhor caso:**  $O(n^2)$  - Mesmo que o array já esteja ordenado
- **Caso médio:**  $O(n^2)$  - Comportamento típico
- **Pior caso:**  $O(n^2)$  - Array ordenado inversamente

### Complexidade de Espaço

- $O(1)$  - Algoritmo in-place, usa apenas memória constante adicional

### Propriedades Importantes

Propriedade	Valor
Estável	✗ Não
In-place	✓ Sim
Adaptivo	✗ Não
Comparações	$O(n^2)$
Trocas	$O(n)$

# Por que o Selection Sort NÃO é Estável?

## O que significa "Estabilidade" em algoritmos de ordenação?

Um algoritmo de ordenação é **estável** quando mantém a **ordem relativa** dos elementos que possuem valores iguais. Ou seja, se dois elementos têm o mesmo valor, aquele que aparece primeiro no array original deve aparecer primeiro no array ordenado.

## Exemplo Prático de Instabilidade

Considere um array de cartas onde cada carta tem um valor e um naipe:

Array inicial: [5♠, 3♦, 5♥, 2♣, 3♠]

Vamos ordenar por valor numérico apenas, ignorando o naipe:

Ordenação Estável (esperada):

[2♣, 3♦, 3♠, 5♠, 5♥]

- Note que 3♦ vem antes de 3♠ (mantém ordem original)
- E 5♠ vem antes de 5♥ (mantém ordem original)

Selection Sort (instável):

[2♣, 3♠, 3♦, 5♥, 5♠]

- 3♠ agora vem antes de 3♦ (ordem alterada!)
- 5♥ agora vem antes de 5♠ (ordem alterada!)

## Por que isso acontece no Selection Sort?

O Selection Sort **troca elementos distantes entre si**, o que pode "pular" sobre elementos iguais e alterar sua ordem relativa.

## Demonstração com números simples:

Array inicial: [4, 2, 4, 1, 3]

- Para distinguir, vamos chamar: [4a, 2, 4b, 1, 3]

### Execução do Selection Sort:

#### Iteração 1:

- Procura menor elemento: 1 (posição 3)
- Troca: 4a ↔ 1
- Resultado: [1, 2, 4b, 4a, 3]



**Observe:** 4b agora vem antes de 4a!

#### Iteração 2:

- Procura menor na parte não ordenada [2, 4b, 4a, 3]: 2 já está correto
- Sem troca
- Resultado: [1, 2, 4b, 4a, 3]

#### Iteração 3:

- Procura menor na parte não ordenada [4b, 4a, 3]: 3 (posição 4)
- Troca: 4b ↔ 3
- Resultado: [1, 2, 3, 4a, 4b]

Array final: [1, 2, 3, 4a, 4b]

## O Problema da "Troca Distante"

Array: [4a, 2, 4b, 1, 3]

↑                   ↑  
|\_\_\_\_\_|

Troca distante que "pula"  
sobre 4b, alterando a ordem!

Quando o Selection Sort encontra o menor elemento em uma posição distante, ele o troca diretamente com a posição atual, **pulando sobre todos os elementos intermediários**, incluindo aqueles que têm o mesmo valor.

## Impacto Prático da Instabilidade

A instabilidade pode ser problemática em situações reais:

### 1. Ordenação de registros de funcionários por salário:

- Se dois funcionários têm o mesmo salário, você pode querer manter a ordem original (ex: por data de contratação)

### 2. Classificação de produtos por preço:

- Produtos com mesmo preço podem ter diferentes prioridades de exibição

### 3. Ordenação de notas de alunos:

- Alunos com a mesma nota podem estar em ordem alfabética inicialmente

## Como tornar o Selection Sort estável?

É possível modificar o Selection Sort para ser estável, mas isso aumenta a complexidade:

```
// Versão estável (menos eficiente)
void stableSelectionSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int minIdx = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
```

```

    }

    // Em vez de trocar diretamente,
    // desloca todos os elementos entre i e minIdx
    int key = arr[minIdx];
    while (minIdx > i) {
        arr[minIdx] = arr[minIdx-1];
        minIdx--;
    }
    arr[i] = key;
}

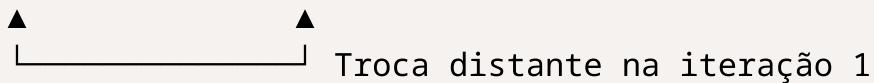
```

**⚠️ Mas isso aumenta a complexidade de trocas de  $O(n)$  para  $O(n^2)$ !**

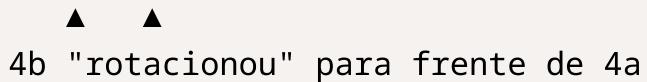
## Visualização da Instabilidade - "Rotatividade"

O conceito de "rotatividade" que você mencionou refere-se à forma como os elementos "giram" ou alteram suas posições relativas durante as trocas distantes:

Estado inicial: [4a, 2, 4b, 1, 3]



Após troca: [1, 2, 4b, 4a, 3]



Estado inicial: [A, B, C, D] (onde A = D em valor)

Após Selection: [D, B, C, A] (A e D trocaram posições!)



## Comparação Visual: Estável vs Instável

Algoritmo Estável (ex: Insertion Sort):

[3a, 1, 3b, 2] → [1, 2, 3a, 3b] Ordem mantida



3a ainda vem antes de 3b

Selection Sort (Instável):

[3a, 1, 3b, 2] → [1, 2, 3b, 3a] Ordem alterada!



3b agora vem antes de 3a

## Resumo: Por que Selection Sort é Instável

- **Trocas distantes:** Elementos são trocados através de grandes distâncias
- **Pula elementos:** Ignora elementos iguais no meio do caminho
- **Foco apenas no valor:** Não considera a posição original dos elementos iguais
- **Prioriza eficiência:** A versão estável seria muito menos eficiente
- **Rotatividade:** Elementos iguais podem "rotacionar" suas posições relativas

# Vantagens vs. Desvantagens

## Vantagens

- Simples de implementar e entender
- Poucas trocas: Máximo de  $n-1$  trocas
- In-place: Não requer memória adicional
- Funciona bem com arrays pequenos
- Eficiente quando operações de escrita são caras

## Desvantagens

- Complexidade  $O(n^2)$ : Ineficiente para arrays grandes
- Não é estável: Pode alterar a ordem relativa de elementos iguais
- Não é adaptivo: Não aproveita arrays parcialmente ordenados
- Sempre faz  $n-1$  passadas: Mesmo que o array já esteja ordenado

# Quando Usar?

O Selection Sort é adequado quando:

- **Arrays pequenos** (< 50 elementos)
- **Operações de escrita são caras** (ex: memória flash)
- **Simplicidade é mais importante que eficiência**
- **Memória é limitada** (algoritmo in-place)

# Comparação com Outros Algoritmos

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Estável	Trocas
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	✗	$O(n)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	✓	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	✓	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	✓	-
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	✗	$O(n \log n)$

# Variações do Selection Sort

## 1. Selection Sort Bidirecional

Encontra simultaneamente o menor e maior elemento em cada passada, colocando-os nas extremidades.

## 2. Selection Sort Recursivo

Implementação recursiva que ordena recursivamente subarray após colocar o menor elemento na primeira posição.

## 3. Selection Sort Estável

Modificação que mantém a estabilidade trocando elementos apenas quando necessário.

# Exercícios Práticos

## Exercício 1: Implementação Básica

Implemente o Selection Sort para ordenar um array de strings por ordem alfabética.

### Solução do Exercício 1

Código completo (null)

```
#include <iostream>
#include <string>
using namespace std;

// Função para encontrar o índice da menor string
int findSmallestStringIndex(string arr[], int start, int end) {
    int smallestIndex = start;

    for (int i = start + 1; i <= end; i++) {
        if (arr[i] < arr[smallestIndex]) {
            smallestIndex = i;
        }
    }

    return smallestIndex;
}

// Função para trocar duas strings
void swapStrings(string arr[], int pos1, int pos2) {
    string temp = arr[pos1];
    arr[pos1] = arr[pos2];
    arr[pos2] = temp;
}

// Selection Sort para strings
void selectionSortStrings(string arr[], int size) {
    cout << "Ordenando strings por ordem alfabética:" << endl;
```

```

for (int i = 0; i < size - 1; i++) {
    cout << "\nIteração " << (i + 1) << ":" << endl;

    int smallestIndex = findSmallestStringIndex(arr, i, size -
1);

    if (i != smallestIndex) {
        cout << "Trocando \" " << arr[i] << "\" com \" " <<
arr[smallestIndex] << "\" " << endl;
        swapStrings(arr, i, smallestIndex);
    } else {
        cout << "\"" << arr[i] << "\" já está na posição
correta" << endl;
    }

    // Mostrar estado atual
    cout << "Array atual: ";
    for (int j = 0; j < size; j++) {
        if (j <= i) {
            cout << "[" << arr[j] << "] ";
        } else {
            cout << arr[j] << " ";
        }
    }
    cout << endl;
}

}

// Exemplo de uso
int main() {
    string frutas[] = {"banana", "maçã", "laranja", "uva",
"abacaxi"};
    int tamanho = 5;

    cout << "Array inicial: ";
    for (int i = 0; i < tamanho; i++) {
        cout << frutas[i] << " ";
    }
}

```

```

    }

    cout << endl;

    selectionSortStrings(frutas, tamanho);

    cout << "\nArray final ordenado: ";
    for (int i = 0; i < tamanho; i++) {
        cout << frutas[i] << " ";
    }
    cout << endl;

    return 0;
}

```

### Pontos importantes:

- Usamos o operador `<` para comparação lexicográfica de strings
- A lógica é a mesma do Selection Sort para números
- Strings são ordenadas alfabeticamente (A-Z)

## Exercício 2: Contagem de Operações

Modifique o algoritmo para contar o número de comparações e trocas realizadas.

### Solução do Exercício 2

Código completo (null)

```

#include <iostream>
using namespace std;

// Estrutura para armazenar estatísticas
struct SortStatistics {
    int comparisons;
    int swaps;

    SortStatistics() : comparisons(0), swaps(0) {}
}

```

```

};

// Função para encontrar o menor elemento com contagem de
comparações
int findSmallestWithStats(int arr[], int start, int end,
SortStatistics& stats) {
    int smallestIndex = start;

    for (int i = start + 1; i <= end; i++) {
        stats.comparisons++; // Incrementa contador de comparações

        if (arr[i] < arr[smallestIndex]) {
            smallestIndex = i;
        }
    }

    return smallestIndex;
}

// Função para trocar elementos com contagem
void swapWithStats(int arr[], int pos1, int pos2, SortStatistics&
stats) {
    if (pos1 != pos2) {
        stats.swaps++; // Incrementa contador de trocas

        int temp = arr[pos1];
        arr[pos1] = arr[pos2];
        arr[pos2] = temp;

        cout << " Troca #" << stats.swaps << ":" << arr[pos1]
            << " <-> " << arr[pos2] << endl;
    }
}

// Selection Sort com estatísticas
void selectionSortWithStats(int arr[], int size) {
    SortStatistics stats;

```

```

cout << "==== SELECTION SORT COM ESTATÍSTICAS ===" << endl;
cout << "Array inicial: ";
for (int i = 0; i < size; i++) {
    cout << arr[i] << " ";
}
cout << endl << endl;

for (int i = 0; i < size - 1; i++) {
    cout << "Iteração " << (i + 1) << ":" << endl;

        int smallestIndex = findSmallestWithStats(arr, i, size -
1, stats);

        cout << " Menor elemento: " << arr[smallestIndex]
            << " (posição " << smallestIndex << ")" << endl;

        swapWithStats(arr, i, smallestIndex, stats);

        cout << " Comparações nesta iteração: "
            << (stats.comparisons - (i > 0 ? (size - i) : 0)) <<
endl;

        cout << " Array atual: ";
        for (int j = 0; j < size; j++) {
            if (j <= i) {
                cout << "[" << arr[j] << "] ";
            } else {
                cout << arr[j] << " ";
            }
        }
        cout << endl << endl;
    }

// Exibir estatísticas finais
cout << "===== ESTATÍSTICAS FINAIS =====" << endl;
cout << "Total de comparações: " << stats.comparisons << endl;
cout << "Total de trocas: " << stats.swaps << endl;
cout << "Eficiência de trocas: " << stats.swaps << "/" <<
(size - 1)

```

```

        << " (" << (stats.swaps * 100.0 / (size - 1)) << "%" <<
endl;
    cout << "Comparações teóricas O(n2): " << (size * (size - 1) /
2) << endl;
    cout << "Comparações reais vs teóricas: " << stats.comparisons
        << "/" << (size * (size - 1) / 2) << endl;
}

// Exemplo de uso
int main() {
    int numeros[] = {64, 34, 25, 12, 22, 11, 90};
    int tamanho = 7;

    selectionSortWithStats(numeros, tamanho);

    return 0;
}

```

### Análise das Estatísticas:

- **Comparações:** Sempre  $n(n-1)/2$  independente da entrada
- **Trocas:** No máximo  $n-1$ , pode ser menor se alguns elementos já estão no lugar
- **Eficiência:** Mostra quantas trocas foram realmente necessárias

## Exercício 3: Versão Estável

Implemente uma versão estável do Selection Sort.

### Solução do Exercício 3

Código completo (null)

```

#include <iostream>
using namespace std;

// Função para rotacionar elementos para manter estabilidade
void rotateRight(int arr[], int start, int end) {

```

```

int temp = arr[end];
for (int i = end; i > start; i--) {
    arr[i] = arr[i - 1];
}
arr[start] = temp;
}

// Selection Sort Estável
void stableSelectionSort(int arr[], int size) {
    cout << "==== SELECTION SORT ESTÁVEL ===" << endl;
    cout << "Array inicial: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl << endl;

    for (int i = 0; i < size - 1; i++) {
        cout << "Iteração " << (i + 1) << ":" << endl;

        // Encontrar o menor elemento na parte não ordenada
        int minIndex = i;
        for (int j = i + 1; j < size; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }

        cout << " Menor elemento: " << arr[minIndex]
            << " (posição " << minIndex << ")" << endl;

        // Em vez de trocar, rotacionar para manter estabilidade
        if (minIndex != i) {
            cout << " Rotacionando elementos da posição " << i
                << " até " << minIndex << endl;

            int minValue = arr[minIndex];
            rotateRight(arr, i, minIndex);
        }
    }
}

```

```

        cout << " Elemento " << minValue
            << " movido para posição " << i << " mantendo
ordem relativa" << endl;
    } else {
        cout << " Elemento já está na posição correta" <<
endl;
    }

    cout << " Array atual: ";
    for (int j = 0; j < size; j++) {
        if (j <= i) {
            cout << "[" << arr[j] << "] ";
        } else {
            cout << arr[j] << " ";
        }
    }
    cout << endl << endl;
}

cout << "Array final ordenado (estável): ";
for (int i = 0; i < size; i++) {
    cout << arr[i] << " ";
}
cout << endl;
}

// Versão para demonstrar estabilidade com pares (valor, índice
original)
struct Element {
    int value;
    int originalIndex;
};

void printElements(Element arr[], int size, string label) {
    cout << label;
    for (int i = 0; i < size; i++) {
        cout << "(" << arr[i].value << "," << arr[i].originalIndex
<< ") ";
    }
}

```

```

    }
    cout << endl;
}

void stableSelectionSortDemo(Element arr[], int size) {
    cout << "\n==== DEMONSTRAÇÃO DE ESTABILIDADE ===" << endl;
    printElements(arr, size, "Array inicial: ");

    for (int i = 0; i < size - 1; i++) {
        // Encontrar o menor elemento
        int minIndex = i;
        for (int j = i + 1; j < size; j++) {
            if (arr[j].value < arr[minIndex].value) {
                minIndex = j;
            }
        }

        // Rotacionar para manter ordem relativa
        if (minIndex != i) {
            Element minElement = arr[minIndex];
            for (int k = minIndex; k > i; k--) {
                arr[k] = arr[k - 1];
            }
            arr[i] = minElement;
        }

        cout << "Após iteração " << (i + 1) << ": ";
        printElements(arr, size, "");
    }
}

int main() {
    // Teste básico
    int numeros[] = {4, 2, 2, 8, 3, 3, 1};
    int tamanho = 7;

    stableSelectionSort(numeros, tamanho);
}

```

```
// Demonstração de estabilidade
Element elementos[] = {
    {4, 0}, {2, 1}, {2, 2}, {8, 3}, {3, 4}, {3, 5}, {1, 6}
};

stableSelectionSortDemo(elementos, 7);

return 0;
}
```

### Como funciona a versão estável:

1. **Rotação em vez de troca:** Em vez de trocar o menor elemento com o primeiro, rotacionamos todos os elementos
2. **Preserva ordem relativa:** Elementos iguais mantêm sua ordem original
3. **Complexidade:**  $O(n^2)$  tempo, mas  $O(n)$  operações de movimento por iteração
4. **Trade-off:** Mais operações de movimento, mas mantém estabilidade

### Diferença visual:

- **Selection Sort normal:**  $[2a, 2b] \rightarrow [2b, 2a]$  (não estável)
- **Selection Sort estável:**  $[2a, 2b] \rightarrow [2a, 2b]$  (estável)

# Conclusão

O Selection Sort é um excelente algoritmo para **aprender os conceitos de ordenação** devido à sua simplicidade e lógica intuitiva. Embora não seja eficiente para arrays grandes, tem seu lugar em situações específicas onde a simplicidade e o baixo número de trocas são importantes.

O algoritmo demonstra claramente os conceitos de:

- Divisão de problema em subproblemas
- Invariantes de loop
- Análise de complexidade
- Trade-offs entre diferentes métricas de performance

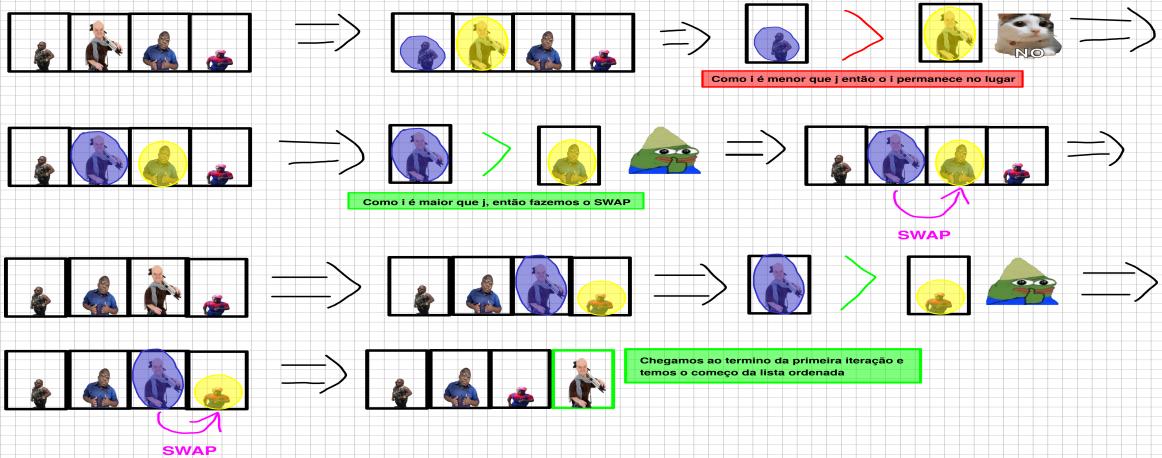
# Bubble Sort

Diagramação

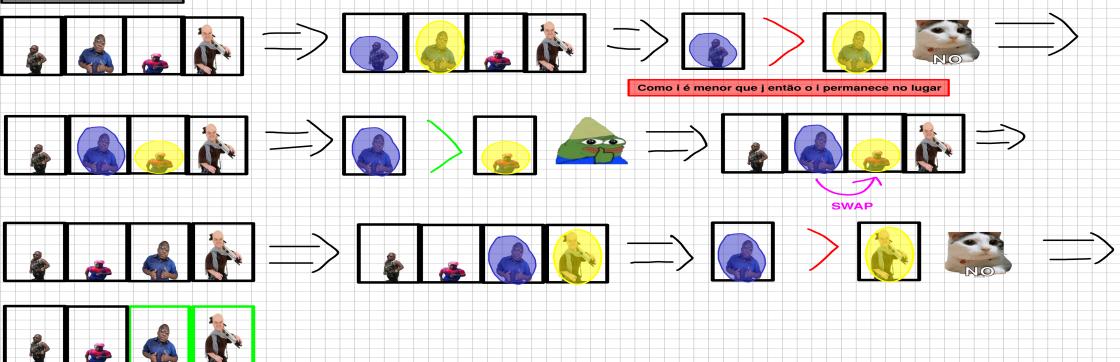
# BUBBLE SORT

  $\equiv$  

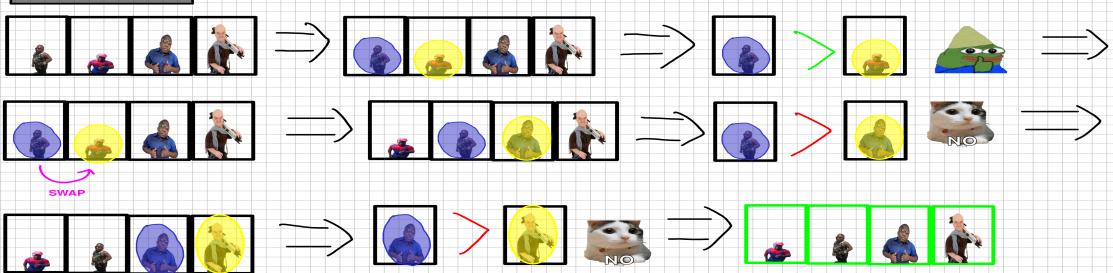
## 1° Iteração



## 2° Iteração



## 3° Iteração



Essa é uma das formas de criarmos o Bubble Sort, está seria uma das formas básicas e não tão eficientes, já que se você perceber eu estaria percorrendo o array muitas vezes e verificando elementos que já sofreram até SWAP para isso podemos verificar se foi trocado e dependendo podemos quebrar o loop que percorre o array

bubbleSort\_annotated.png