

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
Implementação	46
Por que o Bubble Sort É Estável?	52
Vantagens vs. Desvantagens	56
Quando Usar?	57
Comparação com Outros Algoritmos	58
Variações do Bubble Sort	59
Exercícios Práticos	62
Conclusão	77

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.

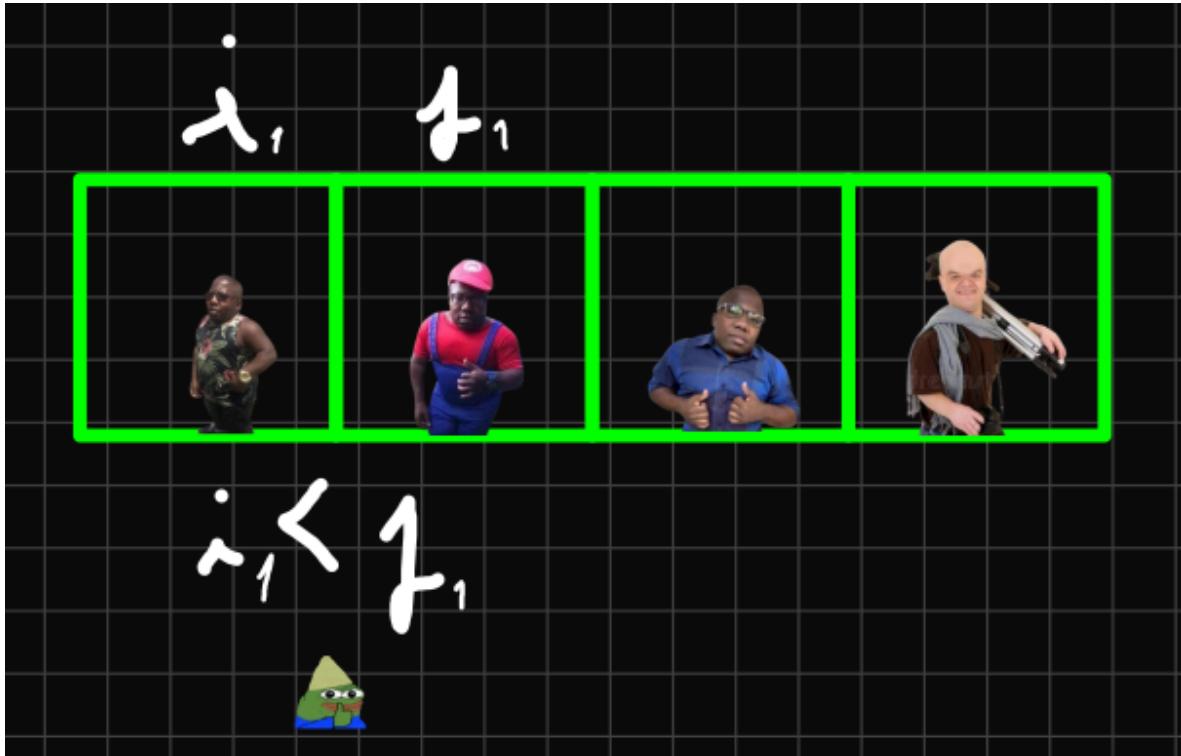
Como Funciona na Prática

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

Primeira Iteração (1^a 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!

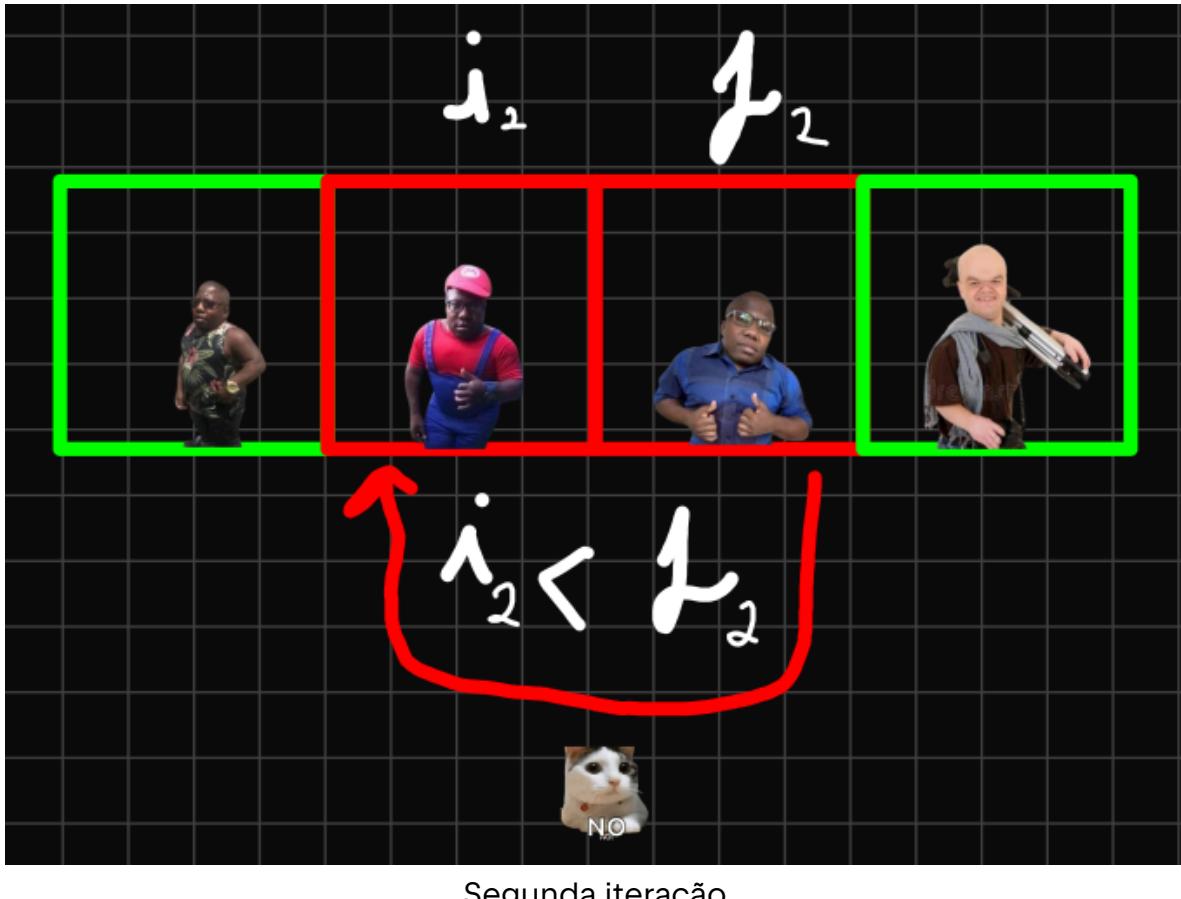


Primeira iteração

Segunda Iteração (2^a 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^a Repetição)

Comparo i_3 com j_3 .

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

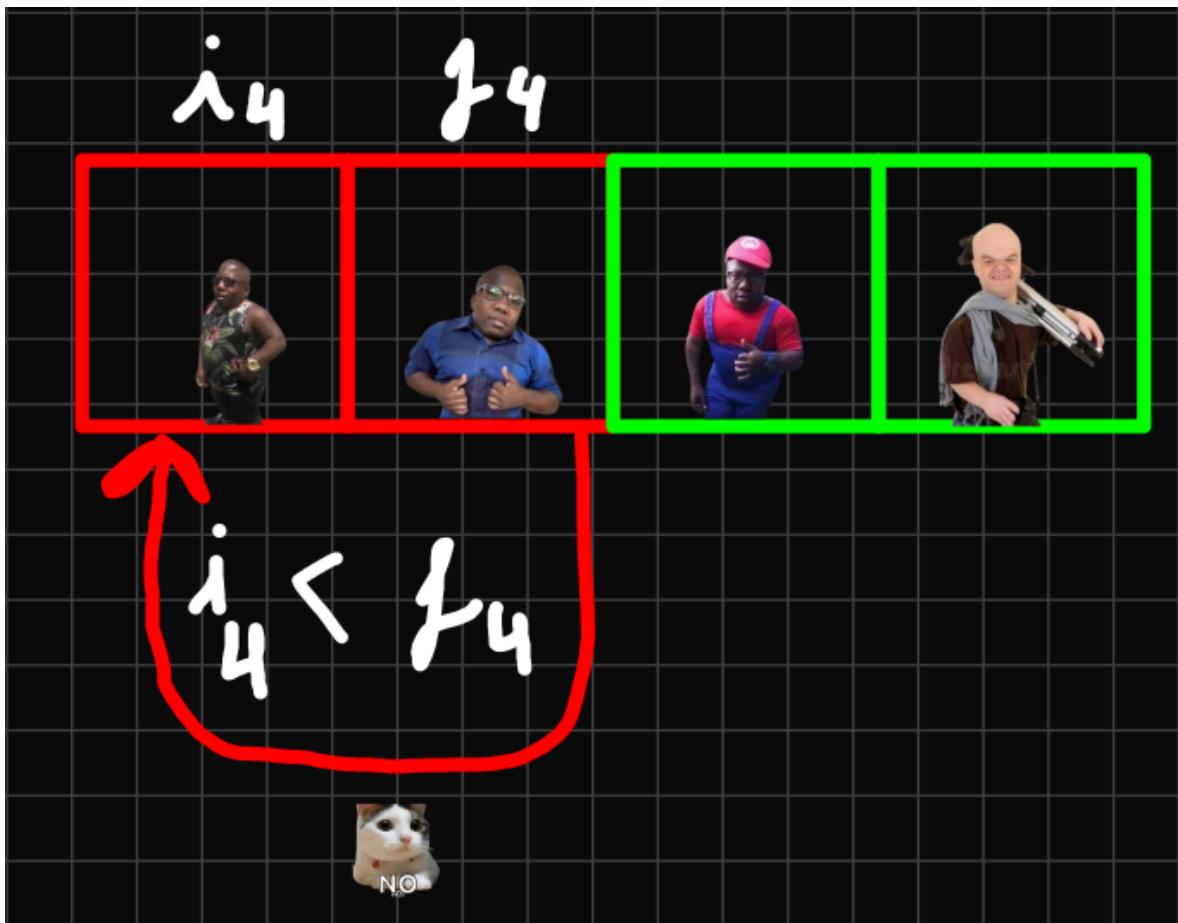


Terceira iteração

Quarta Iteração (4^a 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^a e Última Repetição)

Agora todos os elementos já estão na ordem correta!

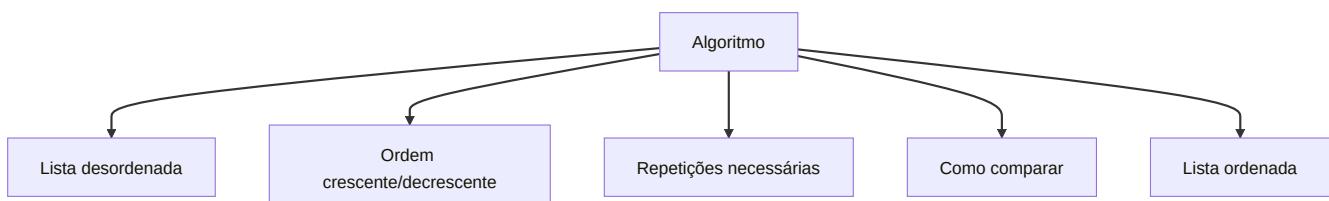


⚠ Resumo: A cada iteração, fazemos uma **pergunta** (comparação) e tomamos uma **decisão** (trocar ou não trocar). Repetimos isso até que tudo esteja organizado!

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:

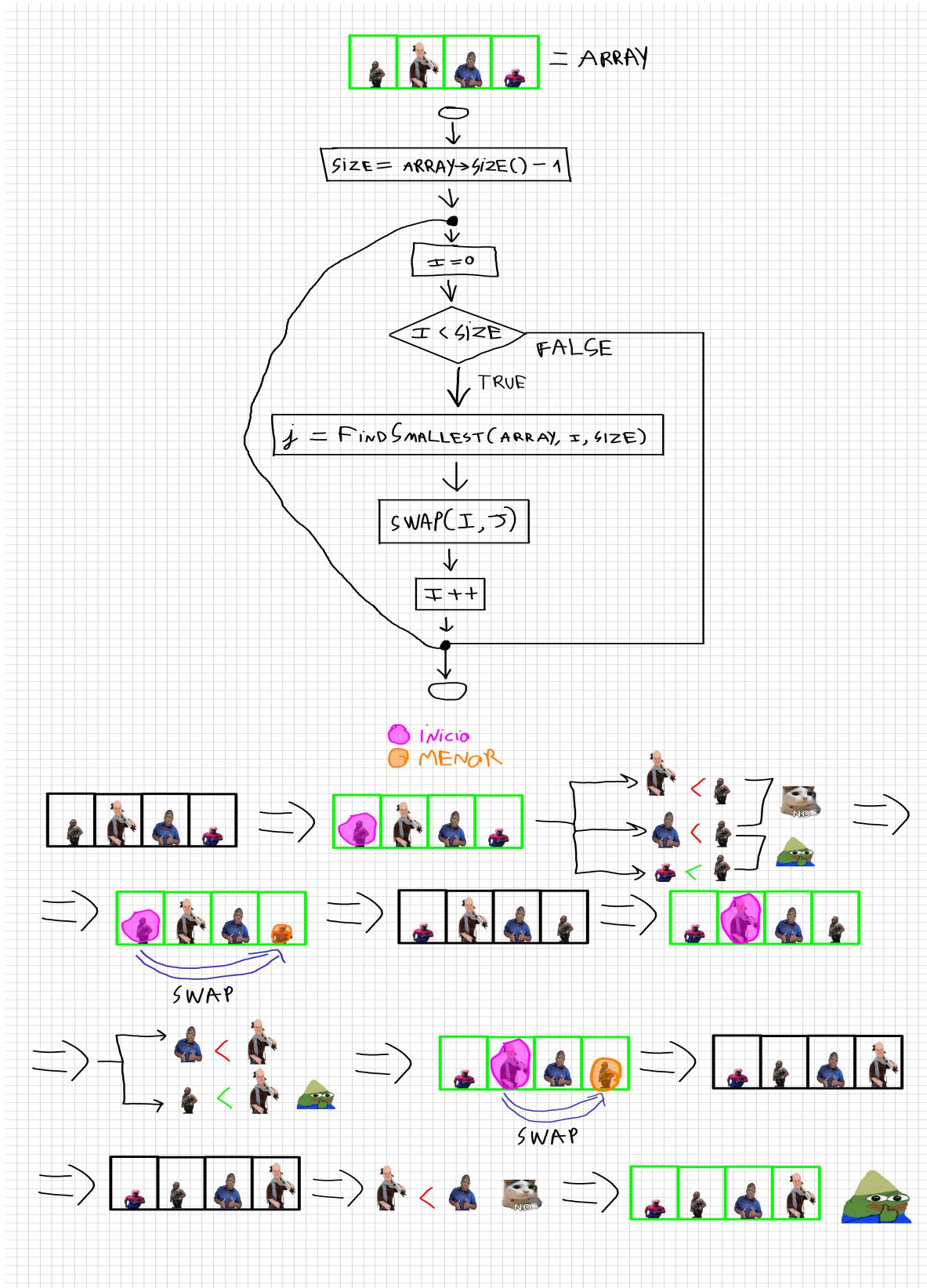
- **Parte ordenada:** Localizada no início do array (inicialmente vazia)
- **Parte não ordenada:** Localizada no final do array (inicialmente todo o array)

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;
}

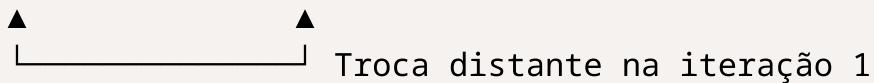
```

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

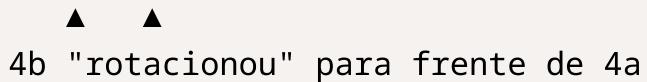
Visualização da Instabilidade - "Rotatividade"

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

O que é Bubble Sort?

O **Bubble Sort** é um algoritmo de ordenação simples que funciona comparando elementos adjacentes e trocando-os se estiverem na ordem errada. O nome vem do fato de que os elementos menores "borbulham" para o início da lista, assim como bolhas de ar sobem para a superfície da água.

Como funciona?

O algoritmo funciona da seguinte forma:

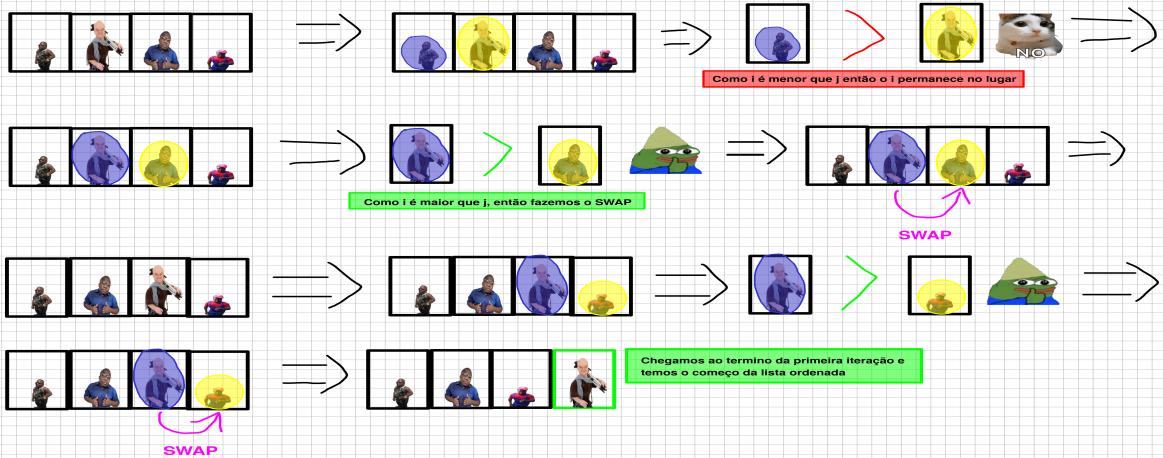
1. **Compara** elementos adjacentes no array
2. **Troca** os elementos se estiverem na ordem errada (o maior vai para a direita)
3. **Repete** o processo para todos os pares adjacentes
4. **Continua** as iterações até que nenhuma troca seja necessária

Diagramação

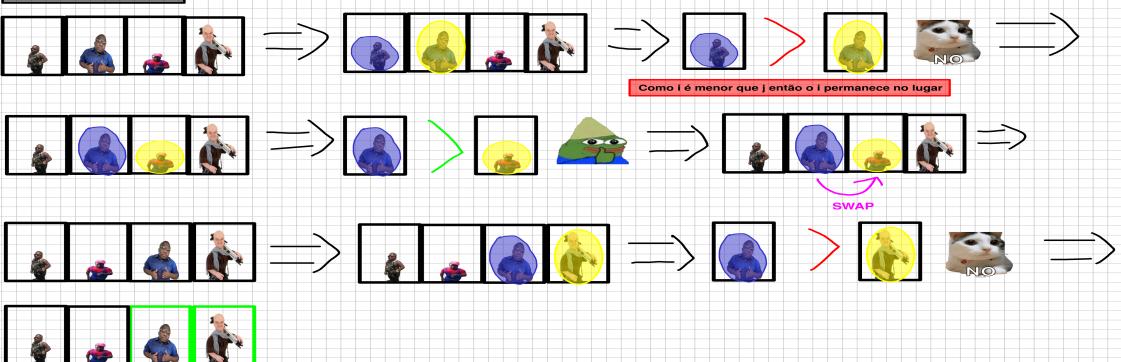
BUBBLE SORT

 \equiv 

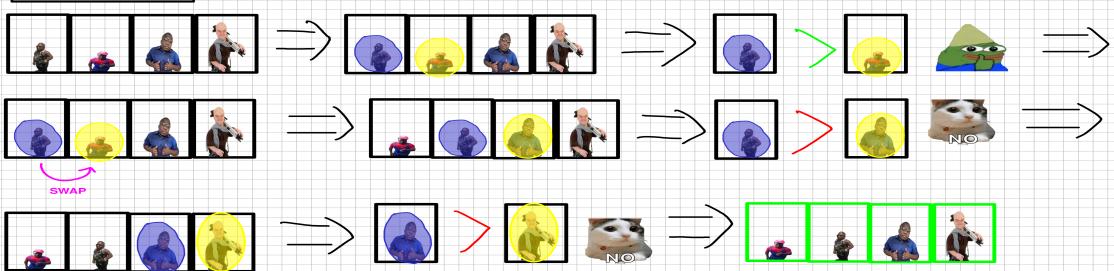
1° Iteração



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

Exemplo Visual

Considerando o array: [64, 34, 25, 12, 22, 11, 90]

Iteração 1:

- Compara 64 e 34: 64 > 34 → Troca: [34, 64, 25, 12, 22, 11, 90]
- Compara 64 e 25: 64 > 25 → Troca: [34, 25, 64, 12, 22, 11, 90]
- Compara 64 e 12: 64 > 12 → Troca: [34, 25, 12, 64, 22, 11, 90]
- Compara 64 e 22: 64 > 22 → Troca: [34, 25, 12, 22, 64, 11, 90]
- Compara 64 e 11: 64 > 11 → Troca: [34, 25, 12, 22, 11, 64, 90]
- Compara 64 e 90: 64 < 90 → Sem troca: [34, 25, 12, 22, 11, 64, 90]
- **Resultado:** O maior elemento (90) está na posição correta

Iteração 2:

- Compara 34 e 25: 34 > 25 → Troca: [25, 34, 12, 22, 11, 64, 90]
- Compara 34 e 12: 34 > 12 → Troca: [25, 12, 34, 22, 11, 64, 90]
- Compara 34 e 22: 34 > 22 → Troca: [25, 12, 22, 34, 11, 64, 90]
- Compara 34 e 11: 34 > 11 → Troca: [25, 12, 22, 11, 34, 64, 90]
- Compara 34 e 64: 34 < 64 → Sem troca
- **Resultado:** Os dois maiores elementos estão nas posições corretas

Iteração 3:

- Compara 25 e 12: 25 > 12 → Troca: [12, 25, 22, 11, 34, 64, 90]
- Compara 25 e 22: 25 > 22 → Troca: [12, 22, 25, 11, 34, 64, 90]
- Compara 25 e 11: 25 > 11 → Troca: [12, 22, 11, 25, 34, 64, 90]
- Compara 25 e 34: 25 < 34 → Sem troca

O processo continua até que nenhuma troca seja necessária, resultando em: [11, 12, 22, 25, 34, 64, 90]

Implementação

Nossa implementação educativa inclui saídas detalhadas para ajudar no aprendizado.

Você pode encontrar uma implementação completa e educativa do Bubble Sort em:
bubbleSort.cpp

Função Principal - Bubble Sort Algorithm

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

    int totalNumberOfSwaps = 0;
    int totalNumberOfComparisons = 0;
    bool hasSwapped = true;
    int iterationCount = 0;

    while (hasSwapped && iterationCount < arraySize - 1)
    {
        iterationCount++;
        hasSwapped = false;

        cout << "">>>> ITERATION " << iterationCount << " <<<" <<
endl;
        cout << "Comparing adjacent elements..." << endl;

        for (int currentIndex = 0; currentIndex < arraySize -
iterationCount; currentIndex++)
```

```

    {
        int nextIndex = currentIndex + 1;
        totalNumberOfComparisons++;

        cout << "Comparing dataArray[" << currentIndex << "] =
" << dataArray[currentIndex]
            << " with dataArray[" << nextIndex << "] = " <<
dataArray[nextIndex];

        if (dataArray[currentIndex] > dataArray[nextIndex]) {
            cout << " -> " << dataArray[currentIndex] << " > "
<< dataArray[nextIndex]
                << ", need to swap!" << endl;
            swapElements(dataArray, currentIndex, nextIndex);
            hasSwapped = true;
            totalNumberOfSwaps++;
        } else {
            cout << " -> " << dataArray[currentIndex] << " <=
" << dataArray[nextIndex]
                << ", no swap needed" << endl;
        }
    }

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

```

```

        if (!hasSwapped) {
            cout << "No swaps performed in this iteration - array
is sorted!" << endl;
        }
        cout << "-----" << endl;
    }

    cout << "===== " << endl;
    cout << "BUBBLE SORT ALGORITHM COMPLETED!" << endl;
    cout << "Total number of iterations: " << iterationCount <<
endl;
    cout << "Total number of comparisons: " <<
totalNumberOfComparisons << endl;
    cout << "Total number of swaps performed: " <<
totalNumberOfSwaps << endl;
    cout << "===== " << endl;
}

```

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;
}

```

Versão Otimizada do Bubble Sort

```
void optimizedBubbleSortAlgorithm(int dataArray[], int arraySize)
{
    cout << "======" << endl;
    cout << "STARTING OPTIMIZED BUBBLE SORT ALGORITHM" << endl;
    cout << "======" << endl;

    int totalNumberOfSwaps = 0;
    int totalNumberOfComparisons = 0;

    for (int iteration = 0; iteration < arraySize - 1;
iteration++)
    {
        bool hasSwapped = false;

        cout << ">>> ITERATION " << (iteration + 1) << " <<<" <<
endl;

        for (int currentIndex = 0; currentIndex < arraySize -
iteration - 1; currentIndex++)
        {
            totalNumberOfComparisons++;

            if (dataArray[currentIndex] > dataArray[currentIndex +
1]) {
                swapElements(dataArray, currentIndex, currentIndex +
1);
                hasSwapped = true;
                totalNumberOfSwaps++;
            }
        }

        // Early termination if no swaps occurred
        if (!hasSwapped) {
            cout << "No swaps in this iteration - array is already
sorted!" << endl;
        }
    }
}
```

```

        break;
    }

    cout << "Array after iteration " << (iteration + 1) << ":";
    for(int displayIndex = 0; displayIndex < arraySize;
displayIndex++) {
        cout << dataArray[displayIndex] << " ";
    }
    cout << endl << "-----" <<
endl;
}

cout << "Total comparisons: " << totalNumberOfComparisons <<
endl;
cout << "Total swaps: " << totalNumberOfSwaps << endl;
}

```

Características do Algoritmo

Complexidade de Tempo

- **Melhor caso:** $O(n)$ - Array já ordenado (com otimização)
- **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	<input checked="" type="checkbox"/> Sim
In-place	<input checked="" type="checkbox"/> Sim
Adaptivo	<input checked="" type="checkbox"/> Sim (com otimização)
Comparações	$O(n^2)$
Trocas	$O(n^2)$

Por que o Bubble Sort 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 Estabilidade

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:

Bubble Sort (estável):

[2♣, 3♦, 3♠, 5♠, 5♥]

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

Por que o Bubble Sort mantém a estabilidade?

O Bubble Sort mantém a estabilidade porque:

1. Compara apenas elementos adjacentes
2. Só troca elementos se o da esquerda for MAIOR que o da direita
3. Nunca troca elementos iguais

Demonstração com números simples:

Array: [4, 2a, 2b, 1] (onde 2a e 2b têm o mesmo valor, mas origens diferentes)

Iteração 1:

- Compara 4 e 2a: 4 > 2a → Troca: [2a, 4, 2b, 1]
- Compara 4 e 2b: 4 > 2b → Troca: [2a, 2b, 4, 1]
- Compara 4 e 1: 4 > 1 → Troca: [2a, 2b, 1, 4]

Iteração 2:

- Compara 2a e 2b: 2a == 2b → **Sem troca** (preserva ordem!)
- Compara 2b e 1: 2b > 1 → Troca: [2a, 1, 2b, 4]

Iteração 3:

- Compara 2a e 1: 2a > 1 → Troca: [1, 2a, 2b, 4]

Resultado final: [1, 2a, 2b, 4] Ordem original mantida!

Exemplo Prático com Dados Reais

```
struct Pessoa {
    string nome;
    int idade;
    int numeroChegada; // Para identificar ordem original
};

// Array inicial (ordenado por chegada):
// 1. João, 25 anos
// 2. Maria, 30 anos
// 3. Pedro, 25 anos
// 4. Ana, 20 anos

// Após Bubble Sort por idade:
// 1. Ana, 20 anos
// 2. João, 25 anos      <- João mantém prioridade sobre Pedro
// 3. Pedro, 25 anos     <- Pedro vem depois (ordem original)
```

```
preservada)  
// 4. Maria, 30 anos
```

Importância da Estabilidade

A estabilidade é crucial quando:

- **Ordenação múltipla:** Primeiro por um campo, depois por outro
- **Preservação de contexto:** Manter informações sobre ordem original
- **Interfaces de usuário:** Comportamento previsível para o usuário
- **Dados com metadados:** Timestamps, IDs, etc.

Comparação com Algoritmos Instáveis

Algoritmo	Estável?	Motivo
Bubble Sort	✓ Sim	Só troca adjacentes se forem diferentes
Selection Sort	✗ Não	Troca elementos distantes
Insertion Sort	✓ Sim	Insere mantendo ordem
Quick Sort	✗ Não	Particionamento pode alterar ordem
Merge Sort	✓ Sim	Merge preserva ordem quando iguais

Implementação que Garante Estabilidade

```
void stableBubbleSort(int arr[], int n) {  
    for (int i = 0; i < n - 1; i++) {  
        for (int j = 0; j < n - i - 1; j++) {  
            // CRUCIAL: usar > e não >=
```

```
        if (arr[j] > arr[j + 1]) { // Não troca elementos
iguais!
            swap(arr[j], arr[j + 1]);
        }
    }
}
```



🔑 **Ponto-chave:** Use `>` (maior que) e nunca `>=` (maior ou igual) na condição de troca para manter a estabilidade!

Vantagens vs. Desvantagens

Vantagens

- Simples de implementar e entender
- Estável: Mantém a ordem relativa de elementos iguais
- In-place: Não requer memória adicional
- Adaptivo: Com otimização pode detectar arrays já ordenados
- Funciona bem com arrays pequenos
- Detecta facilmente se o array já está ordenado

Desvantagens

- Complexidade $O(n^2)$: Ineficiente para arrays grandes
- Muitas trocas: Pode fazer até $O(n^2)$ trocas no pior caso
- Lento na prática: Mesmo entre algoritmos $O(n^2)$
- Não recomendado para dados em produção
- Elementos pequenos "borbulham" lentamente para o início

Quando Usar?

O Bubble Sort é adequado quando:

- **Arrays muito pequenos** (< 10 elementos)
- **Estabilidade é crucial** (manter ordem relativa de elementos iguais)
- **Detecção de ordenação** é importante (pode parar cedo se já ordenado)
- **Fins educacionais** (aprender conceitos de ordenação)
- **Simplicidade extrema** é mais importante que eficiência
- **Protótipos rápidos** onde performance não é crítica

Comparação com Outros Algoritmos

Algoritmo	Melhor Caso	Caso Médio	Pior Caso	Estável	Trocas
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	✓	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	✗	$O(n)$
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 Bubble Sort

1. Bubble Sort Otimizado (com Flag)

Adiciona uma flag para detectar quando o array já está ordenado e para antecipadamente.

```
void optimizedBubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        bool hasSwapped = false;

        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                hasSwapped = true;
            }
        }

        // Se não houve trocas, array está ordenado
        if (!hasSwapped) {
            break;
        }
    }
}
```

2. Cocktail Sort (Bubble Sort Bidirecional)

Funciona em ambas as direções alternadamente, melhorando a performance em alguns casos.

```
void cocktailSort(int arr[], int n) {
    bool hasSwapped = true;
    int start = 0;
    int end = n - 1;

    while (hasSwapped) {
```

```

hasSwapped = false;

// Esquerda para direita
for (int i = start; i < end; i++) {
    if (arr[i] > arr[i + 1]) {
        swap(arr[i], arr[i + 1]);
        hasSwapped = true;
    }
}
end--;

if (!hasSwapped) break;

// Direita para esquerda
for (int i = end; i > start; i--) {
    if (arr[i] < arr[i - 1]) {
        swap(arr[i], arr[i - 1]);
        hasSwapped = true;
    }
}
start++;
}

}

```

3. Bubble Sort Recursivo

Implementação recursiva que "borbulha" o maior elemento e ordena recursivamente o restante.

```

void recursiveBubbleSort(int arr[], int n) {
    // Caso base
    if (n == 1) return;

    // Uma passada para colocar o maior elemento no final
    for (int i = 0; i < n - 1; i++) {
        if (arr[i] > arr[i + 1]) {
            swap(arr[i], arr[i + 1]);

```

```

    }

    // Chama recursivamente para os primeiros n-1 elementos
    recursiveBubbleSort(arr, n - 1);
}

```

4. Odd-Even Sort (Brick Sort)

Variação que compara elementos em posições ímpares/pares alternadamente.

```

void oddEvenSort(int arr[], int n) {
    bool isSorted = false;

    while (!isSorted) {
        isSorted = true;

        // Compara elementos em posições ímpares
        for (int i = 1; i <= n - 2; i += 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }

        // Compara elementos em posições pares
        for (int i = 0; i <= n - 2; i += 2) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                isSorted = false;
            }
        }
    }
}

```

Exercícios Práticos

Exercício 1: Implementação Básica

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

Solução do Exercício 1

Código completo (bubbleSortString.cpp)

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

// Função para trocar duas strings
void swapStrings(string arr[], int pos1, int pos2) {
    cout << " -> Trocando \"" << arr[pos1] << "\" com \""
    arr[pos2] << "\"" << endl;
    string temp = arr[pos1];
    arr[pos1] = arr[pos2];
    arr[pos2] = temp;
}

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

    for (int i = 0; i < size - 1; i++) {
        cout << "\n>>> ITERAÇÃO " << (i + 1) << " <<<" << endl;
        bool hasSwapped = false;

        for (int j = 0; j < size - i - 1; j++) {
            cout << "Comparando \"" << arr[j] << "\" com \""
            arr[j + 1] << "\"";

            if (arr[j] > arr[j + 1]) {
```

```

        cout << " -> \" " << arr[j] << "\\" > \" " << arr[j +
1] << "\", precisa trocar!" << endl;
        swapStrings(arr, j, j + 1);
        hasSwapped = true;
    } else {
        cout << " -> \" " << arr[j] << "\\" <= \" " << arr[j
+ 1] << "\", sem troca" << endl;
    }
}

cout << "Array após iteração " << (i + 1) << ":" ;
for (int k = 0; k < size; k++) {
    if (k >= size - i - 1) {
        cout << "[\" " << arr[k] << "\"] ";
    } else {
        cout << "\" " << arr[k] << "\\" ";
    }
}
cout << endl;

if (!hasSwapped) {
    cout << "Nenhuma troca realizada - array já está
ordenado!" << endl;
    break;
}
}

int main() {
    const int SIZE = 6;
    string nomes[SIZE] = {"Maria", "João", "Ana", "Pedro",
"Carlos", "Beatriz"};

    cout << "Array inicial: ";
    for (int i = 0; i < SIZE; i++) {
        cout << "\" " << nomes[i] << "\\" ";
    }
    cout << endl << endl;
}

```

```

bubbleSortStrings(nomes, SIZE);

cout << "\n======" << endl;
cout << "RESULTADO FINAL:" << endl;
cout << "Array ordenado: ";
for (int i = 0; i < SIZE; i++) {
    cout << "\" " << nomes[i] << "\" ";
}
cout << endl;
cout << "======" << endl;

return 0;
}

```

Exercício 2: Bubble Sort com Contador de Operações

Modifique o algoritmo Bubble Sort para contar e exibir o número total de comparações e trocas realizadas.

Solução do Exercício 2

Código completo (bubbleWithStats.cpp)

```

#include <iostream>
using namespace std;

struct BubbleSortStats {
    int comparisons;
    int swaps;
    int iterations;
};

void bubbleSortWithStats(int arr[], int size, BubbleSortStats& stats) {
    stats.comparisons = 0;
    stats.swaps = 0;
    stats.iterations = 0;
}

```

```

cout << "Bubble Sort com estatísticas detalhadas:" << endl;

for (int i = 0; i < size - 1; i++) {
    stats.iterations++;
    cout << "\n>>> ITERAÇÃO " << stats.iterations << " <<<" <<
endl;
    bool hasSwapped = false;

    for (int j = 0; j < size - i - 1; j++) {
        stats.comparisons++;
        cout << "Comparação " << stats.comparisons << ":" " <<
arr[j] << " vs " << arr[j + 1];

        if (arr[j] > arr[j + 1]) {
            // Realizar troca
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;

            stats.swaps++;
            hasSwapped = true;
            cout << " -> Troca " << stats.swaps << "
realizada!" << endl;
        } else {
            cout << " -> Sem troca necessária" << endl;
        }
    }

    cout << "Array após iteração " << stats.iterations << ":" ;
}

for (int k = 0; k < size; k++) {
    cout << arr[k] << " ";
}
cout << endl;

if (!hasSwapped) {
    cout << "Otimização: Array já ordenado, parando

```

```

    antecipadamente!" << endl;
        break;
    }
}
}

int main() {
    int numeros[] = {64, 34, 25, 12, 22, 11, 90};
    int tamanho = sizeof(numeros) / sizeof(numeros[0]);
    BubbleSortStats estatisticas;

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

    bubbleSortWithStats(numeros, tamanho, estatisticas);

    cout << "\n======" << endl;
    cout << "ESTATÍSTICAS FINAIS:" << endl;
    cout << "Iterações realizadas: " << estatisticas.iterations << endl;
    cout << "Total de comparações: " << estatisticas.comparisons << endl;
    cout << "Total de trocas: " << estatisticas.swaps << endl;
    cout << "Array final: ";
    for (int i = 0; i < tamanho; i++) {
        cout << numeros[i] << " ";
    }
    cout << endl;
    cout << "======" << endl;

    return 0;
}

```

Exercício 3: Bubble Sort Bidirecional (Cocktail Sort)

Implemente uma variação do Bubble Sort que funciona nas duas direções.

Solução do Exercício 3

Código completo (cockTailSort.cpp)

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

void cocktailSort(int arr[], int size) {
    cout << "Implementando Cocktail Sort (Bubble Sort Bidirecional):" << endl;

    bool hasSwapped = true;
    int start = 0;
    int end = size - 1;
    int iteration = 0;

    while (hasSwapped) {
        iteration++;
        hasSwapped = false;

        cout << "\n>>> ITERAÇÃO " << iteration << " - DIREÇÃO →"
<< endl;
        // Passada da esquerda para direita
        for (int i = start; i < end; i++) {
            cout << "Comparando " << arr[i] << " com " << arr[i + 1];
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                hasSwapped = true;
                cout << " -> Trocado!" << endl;
            } else {
                cout << " -> Sem troca" << endl;
            }
        }
    }
}
```

```

        if (!hasSwapped) {
            break;
        }

        end--;
    }

    cout << "\n>>> ITERAÇÃO " << iteration << " - DIREÇÃO <="
    << endl;

    // Passada da direita para esquerda
    for (int i = end; i > start; i--) {
        cout << "Comparando " << arr[i] << " com " << arr[i - 1];
        if (arr[i] < arr[i - 1]) {
            swap(arr[i], arr[i - 1]);
            hasSwapped = true;
            cout << " -> Trocado!" << endl;
        } else {
            cout << " -> Sem troca" << endl;
        }
    }

    start++;
}

cout << "Array após iteração " << iteration << ": ";
for (int j = 0; j < size; j++) {
    cout << arr[j] << " ";
}
cout << endl;
}

int main() {
    int numeros[] = {5, 1, 4, 2, 8, 0, 2};
    int tamanho = sizeof(numeros) / sizeof(numeros[0]);

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

```

```

        cout << numeros[i] << " ";
    }
    cout << endl;

    cocktailSort(numeros, tamanho);

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

    return 0;
}

```

Exercício 4: Análise de Performance

Compare o desempenho do Bubble Sort com e sem otimização de parada antecipada.

Desafio Extra

Implemente uma versão do Bubble Sort que:

1. Conta operações (comparações e trocas)
2. Para automaticamente quando detecta que está ordenado
3. Mostra estatísticas detalhadas no final
4. Funciona com diferentes tipos de dados (int, float, string)

Solução do Desafio Extra

Código completo (advancedBubbleSort.cpp)

```
#include <iostream>
#include <string>
#include <vector>
#include <chrono>
```

```

#include <iomanip>
using namespace std;
using namespace std::chrono;

// Estrutura para armazenar estatísticas de ordenação
struct SortingStats {
    int comparisons = 0;
    int swaps = 0;
    int iterations = 0;
    double timeElapsed = 0.0;
    bool optimizedExit = false;
};

// Template para Bubble Sort genérico com estatísticas
template<typename T>
void advancedBubbleSort(T arr[], int size, SortingStats& stats) {
    auto start = high_resolution_clock::now();

    cout << "\n======" << endl;
    cout << "ADVANCED BUBBLE SORT WITH STATISTICS" << endl;
    cout << "======" << endl;

    stats = {0, 0, 0, 0.0, false}; // Reset statistics

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

    bool hasSwapped = true;

    while (hasSwapped && stats.iterations < size - 1) {
        hasSwapped = false;
        stats.iterations++;

        cout << ">>> ITERAÇÃO " << stats.iterations << " <<<" <<
endl;
}

```

```

for (int i = 0; i < size - stats.iterations; i++) {
    stats.comparisons++;

    cout << "Comparação " << stats.comparisons << ":" "
        << arr[i] << " vs " << arr[i + 1];

    if (arr[i] > arr[i + 1]) {
        // Realizar troca
        T temp = arr[i];
        arr[i] = arr[i + 1];
        arr[i + 1] = temp;

        stats.swaps++;
        hasSwapped = true;
        cout << " -> Troca " << stats.swaps << "
realizada!" << endl;
    } else {
        cout << " -> Sem troca necessária" << endl;
    }
}

cout << "Estado do array: ";
for (int i = 0; i < size; i++) {
    if (i >= size - stats.iterations) {
        cout << "[" << arr[i] << "] ";
    } else {
        cout << arr[i] << " ";
    }
}
cout << endl;

if (!hasSwapped) {
    stats.optimizedExit = true;
    cout << "🔴 OTIMIZAÇÃO: Array já ordenado! Parando
antecipadamente." << endl;
}
cout << "-----" << endl;

```

```

}

auto end = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(end - start);
stats.timeElapsed = duration.count() / 1000.0; // Convert to
milliseconds

cout << "\n======" << endl;
cout << "ESTATÍSTICAS DETALHADAS" << endl;
cout << "======" << endl;
cout << "Elementos no array: " << size << endl;
cout << "Iterações realizadas: " << stats.iterations << endl;
cout << "Total de comparações: " << stats.comparisons << endl;
cout << "Total de trocas: " << stats.swaps << endl;
cout << "Tempo de execução: " << fixed << setprecision(3)
    << stats.timeElapsed << " ms" << endl;
cout << "Otimização ativada: " << (stats.optimizedExit ? "✅"
Sim" : "❌ Não") << endl;
cout << "Eficiência: " << fixed << setprecision(2)
    << ((double)stats.swaps / stats.comparisons * 100) << "% das comparações resultaram em trocas" << endl;

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

// Função para testar com diferentes tipos de dados
void testWithIntegers() {
    cout << "\n1 2 3 4 TESTE COM NÚMEROS INTEIROS" << endl;
    int numeros[] = {64, 34, 25, 12, 22, 11, 90};
    int tamanho = sizeof(numeros) / sizeof(numeros[0]);
    SortingStats stats;

    advancedBubbleSort(numeros, tamanho, stats);
}

```

```

}

void testWithFloats() {
    cout << "\n1 23 4 TESTE COM NÚMEROS DECIMAIS" << endl;
    float decimais[] = {3.14f, 2.71f, 1.41f, 1.73f, 0.57f};
    int tamanho = sizeof(decimais) / sizeof(decimais[0]);
    SortingStats stats;

    advancedBubbleSort(decimais, tamanho, stats);
}

void testWithStrings() {
    cout << "\n📝 TESTE COM STRINGS" << endl;
    string nomes[] = {"Maria", "João", "Ana", "Pedro", "Carlos"};
    int tamanho = sizeof(nomes) / sizeof(nomes[0]);
    SortingStats stats;

    advancedBubbleSort(nomes, tamanho, stats);
}

void testWithAlreadySorted() {
    cout << "\n✓ TESTE COM ARRAY JÁ ORDENADO (Demonstração de
Otimização)" << endl;
    int ordenado[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int tamanho = sizeof(ordenado) / sizeof(ordenado[0]);
    SortingStats stats;

    advancedBubbleSort(ordenado, tamanho, stats);
}

void testWithReverseSorted() {
    cout << "\n✗ TESTE COM ARRAY INVERSAMENTE ORDENADO (Pior
Caso)" << endl;
    int reverso[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
    int tamanho = sizeof(reverso) / sizeof(reverso[0]);
    SortingStats stats;

    advancedBubbleSort(reverso, tamanho, stats);
}

```

```

}

// Função para comparar performance
void performanceComparison() {
    cout << "\n<img alt='chart icon' data-bbox='288 164 311 181' style='vertical-align: middle; height: 1em;"/> COMPARAÇÃO DE PERFORMANCE" << endl;
    cout << "===== " << endl;

    // Teste com diferentes tamanhos
    vector<int> sizes = {5, 10, 15};

    for (int size : sizes) {
        cout << "\n🔍 Testando com " << size << " elementos:" <<
endl;

        // Criar array aleatório
        int* arr = new int[size];
        for (int i = 0; i < size; i++) {
            arr[i] = rand() % 100;
        }

        SortingStats stats;
        advancedBubbleSort(arr, size, stats);

        cout << "Resultado: " << stats.iterations << " iterações,
"
            << stats.comparisons << " comparações, "
            << stats.swaps << " trocas em "
            << stats.timeElapsed << " ms" << endl;

        delete[] arr;
    }
}

int main() {
    cout << "===== "
<< endl;
    cout << "BUBBLE SORT AVANÇADO - DESAFIO EXTRA COMPLETO" <<
endl;
}

```

```

cout << "====="
<< endl;

// Teste com diferentes tipos de dados
testWithIntegers();
testWithFloats();
testWithStrings();

// Testes especiais para demonstrar otimizações
testWithAlreadySorted();
testWithReverseSorted();

// Comparaçāo de performance
performanceComparison();

cout << "\n🎉 Todos os testes foram concluídos com sucesso!"
<< endl;
cout << "📚 Este exemplo demonstra:" << endl;
cout << "    ✓ Contagem de operações" << endl;
cout << "    ✓ Parada antecipada (otimização)" << endl;
cout << "    ✓ Estatísticas detalhadas" << endl;
cout << "    ✓ Suporte a diferentes tipos de dados" << endl;
cout << "    ✓ Medição de tempo de execução" << endl;
cout << "    ✓ Análise de eficiência" << endl;

return 0;
}

```

Explicação da Solução

Esta solução avançada implementa todos os requisitos do desafio:

1. Contagem de Operações

- Struct `SortingStats` armazena comparações, trocas, iterações e tempo
- Cada operação é contada e exibida em tempo real

2. Parada Antecipada

- Flag `hasSwapped` detecta quando não há mais trocas
- Para automaticamente, economizando iterações desnecessárias

3. Estatísticas Detalhadas

- Número total de operações realizadas
- Tempo de execução em milissegundos
- Percentual de eficiência (trocas/comparações)
- Indicação se a otimização foi ativada

4. Suporte a Diferentes Tipos

- Template genérico funciona com `int`, `float`, `string`
- Testes demonstram funcionamento com cada tipo

5. Recursos Extras

- Visualização do processo de ordenação
- Testes com casos especiais (já ordenado, inverso)
- Comparação de performance com diferentes tamanhos
- Medição precisa de tempo de execução

Esta implementação é ideal para estudos avançados de algoritmos e análise de performance!

Conclusão

O Bubble Sort é um algoritmo fundamental para **aprender os conceitos de ordenação** devido à sua simplicidade conceitual e facilidade de implementação. Embora não seja eficiente para arrays grandes, é excelente para fins educacionais e situações específicas onde a estabilidade é crucial.

O algoritmo demonstra claramente os conceitos de:

- Comparação de elementos adjacentes
- Algoritmos estáveis vs. instáveis
- Otimizações algorítmicas (parada antecipada)
- Análise de complexidade no melhor e pior caso
- Trade-offs entre simplicidade e eficiência

Quando usar Bubble Sort:

- Arrays muito pequenos (< 10 elementos)
- Situações educacionais
- Quando a estabilidade é essencial
- Como base para entender algoritmos mais complexos