ALGORITMO DE OPERAÇÃO BUBBLE SORT

Espaço de apresentação criado para alocar os materiais de recomendados sobre o conteúdo abordado

Antonio Hiroky, David do Carmo, Douglas Masuzzo

Praia Grande, 25 de setembro de 2025

SUMÁRIO

1. INTRODUÇÃO A ALGORITMO DE ORDENAÇÃO

1.1. O que é um Algoritmo?

Um algoritmo pode ser entendido como um procedimento computacional bem definido que, a partir de uma ou mais entradas, gera uma ou mais saídas. Em essência, trata-se de uma sequência de etapas que transforma dados de entrada em resultados. Nesse sentido, o algoritmo é uma ferramenta para a resolução de problemas computacionais bem especificados, nos quais se define a relação entre a entrada e a saída desejada. Um exemplo clássico é o problema da ordenação de números, que, além de ser recorrente na prática, serve de base para o estudo e aplicação de diversas técnicas de projeto e análise de algoritmos (CORMEN, 2012).

1.2. Algoritmo de Ordenação

Os algoritmos de ordenação são procedimentos computacionais que organizam elementos de uma lista ou array, em uma sequência específica, de ordem crescente ou decrescente. A eficiência de uma sequência ordenada pode impactar diretamente na manipulação de dados e no desempenho de sistemas que lidam com grandes volumes de informações.

2. ORDENAÇÃO DE BOLHA - BUBBLE SORT

2.1. Origem

Buscando compreender a popularidade do método, pesquisadores realizaram diversos estudos para identificar sua origem. A primeira descrição registrada data de 1956, mencionada por Donald Knuth, que analisou o funcionamento do algoritmo sob o nome de “ sorting by exchange “ (ordenação por troca).

Em 1962, a Association for Computing Machinery (ACM) publicou uma bibliografia em sua conferência, utilizando o mesmo termo. Essa denominação também aparece em materiais didáticos da época, como o livro Programming Business Computers, de Daniel McCracken, bem como em artigos da Journal of the ACM (1961–1962), onde era comumente referido como exchange sorting.

Acredita-se que o termo Bubble Sort tenha sido empregado pela primeira vez, em 1962, por Kenneth Iverson, em sua obra A Programming Language. Embora não haja consenso absoluto, essa é considerada a primeira menção documentada do nome.

Em 1963, o algoritmo foi registrado oficialmente no repositório da ACM como Algorithm 175, sob a denominação “ Shuttle Sort( ) “. Posteriormente, identificaram-se erros no código publicado, levando à proposta de uma versão corrigida, incluindo a otimização que permite a parada antecipada quando não ocorrem trocas. Esse mesmo procedimento já havia sido estudado em 1955, no computador ORDVAC, localizado no Aberdeen Proving Ground.

2.2. Descrição

O Bubble Sort, também conhecido como método da bolha, é um dos algoritmos de ordenação mais simples e didáticos utilizados em programação. Seu nome vem da ideia de que, a cada passagem pelo vetor, os maiores elementos “flutuam” para o final da lista, assim como bolhas que sobem à superfície da água.
Apesar de ser considerado pouco eficiente para grandes quantidades de dados, o método é utilizado em ambientes acadêmicos por sua facilidade de compreensão e implementação. Ajudando estudantes a entenderem conceitos de lógica de programação, como comparações, trocas, laços de repetição e a complexidade de algoritmos.

3. PROCESSO DE FUNCIONAMENTO

3.1. Mecanismos de Ordenação

O mecanismo é um algoritmo de ordenação simples baseado em comparações sucessivas entre elementos adjacentes de uma lista. Sua lógica consiste em percorrer a lista diversas vezes, comparando pares consecutivos e realizando trocas sempre que o elemento da esquerda for maior que o da direita.

Ao final da primeira iteração, o maior elemento da lista estará na última posição, ou seja, em seu lugar definitivo. Na segunda iteração, o segundo maior elemento também ocupará sua posição correta, e assim sucessivamente. Esse processo continua até que toda a lista esteja ordenada.

3.1.1. Etapas de funcionamento

- O Bubble Sort percorre o vetor várias vezes, comparando sempre dois elementos vizinhos (posição i e i+1).
- O Bubble Sort percorre o vetor várias vezes, comparando sempre dois elementos vizinhos (posição i e i+1).
- Ao final de cada passagem pelo vetor, o maior elemento daquela sequência “sobe” para o final, como uma bolha na água.
- Na próxima passagem, não é mais necessário considerar o último elemento, pois ele já está no lugar certo.
- Esse processo se repete até que não seja mais necessária nenhuma troca, o que significa que o vetor está ordenado.

3.2. Pseudocódigo

```
INÍCIO
   PROCEDIMENTO BubbleSort(VETOR A[1..N])
      PARA i ← 1 ATÉ N - 1 FAÇA
         trocou ← FALSO

         PARA j ← 1 ATÉ N - i FAÇA
            SE A[ j ] > A[ j + 1 ] ENTÃO
               temp ← A[ j ]
               A[j] ← A[ j + 1 ]
               A[ j + 1 ] ← temp
               trocou ← VERDADEIRO
            FIM SE
         FIM PARA

         SE trocou = FALSO ENTÃO
            SAIR DO LAÇO
         FIM SE
      FIM PARA
   FIM PROCEDIMENTO
        FIM
```

4. NÍVEIS DE COMPLEXIDADE

A complexidade de um algoritmo de ordenação é verificada de acordo com a rapidez e o espaço do processo de organização de uma lista de elementos. A complexidade é expressa em notação ao “ Big O ”, que descreve o tempo de execução em relação ao tamanho de entrada daquele vetor. Através dessa análise, estima-se que o desempenho geral apresentado pelo método é considerado ruim em comparação aos demais algoritmos de ordenação, que possuem uma classificação superior.

4.1. Melhor caso - Lista Ordenada

- A técnica pode apresentar uma complexidade linear O(n) se o método for implementado com uma otimização para o processo se nenhuma troca for feita em uma passagem.

4.2. Pior caso - Lista Desordenada

- O método apresenta uma complexidade de O(n²), pois são necessárias múltiplas passagens pela lista para ordenar todos os elementos, exigindo constantes comparações e trocas.

4.3. Análise de Complexidade de Algoritmos

	Constante	**Logaritmo**	**Linear**	**N-log(N)**	**Quadrante**	**Cúbico**	**Exponencial**
N	O(1)	O(logN)	O(N)	O(N logN)	O (N²)	O(N³)	O(2^N)
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4096	65.536
32	1	5	32	160	1024	32.768	4.294.967…

A análise de complexidade busca avaliar a eficiência de um algoritmo em termos de tempo de execução e uso de memória, considerando o crescimento do problema. O objetivo é prever como o desempenho se comporta à medida que o tamanho da entrada aumenta. Um dos principais instrumentos dessa análise é a Big O Notation, que descreve a ordem de crescimento do algoritmo, expressando o limite superior do número de operações realizadas. Ela permite comparar algoritmos independentemente da máquina ou linguagem usada.

- $O(1)$
  - Complexidade constante o tempo de execução do algoritmo independe do tamanho da entrada é bem rápido.

- $O(log(n))$
  - Complexidade logarítmica o tempo de execução pode ser considerado menor do que uma constante grande. É super rápido

- $O(n)$
  - Complexidade linear o algoritmo realiza um número fixo de operações sobre cada elemento da entrada

- $O(n\,log(n))$
  - Típico de algoritmos que dividem um problema em subproblemas, resolve cada subproblema de forma independente, e depois combina os resultados

- $O(n^2)$
  - Complexidade quadrática. Típico de algoritmos que operam sobre pares dos elementos de entrada

- $O(n^3)$
  - Complexidade cúbica que é útil para resolver problemas pequenos como multiplicação de matrizes

- $O(2n)$
  - Complexidade exponencial. Típicos de algoritmos que fazem busca exaustiva (força bruta) para resolver um problema

- $O(n!)$
  - Complexidade fatorial. É normalmente encontrado ao analisar a complexidade de algoritmos de força bruta, que tentam todas as possibilidades para problemas de otimização combinatória.

5. VANTAGENS E DESVANTAGENS

5.1. Vantagens

O algoritmo reforça a ágil implementação e a fluída clareza conceitual sendo aplicado em recursos didáticos pela facilidade de compreensão e exemplificação (através da codificação), pois a lógica algorítmica se baseia apenas na comparação entre os elementos adjacentes e a substituição destes valores quando excedem o espaço de ordem.
A técnica ocupa espaço constante na memória, pois as trocas ocorrem diretamente no array original, sem intervenção de estruturas auxiliares. Além disso, a sua capacidade de aproveitar listas pré-ordenadas possibilita alcançar o baixo nível de complexidade de O(n) em determinados casos.

5.2. Desvantagens

Ao contrário do que é apresentado como qualidade, o grande ponto que destaca sua ineficiência é o nível de desempenho em relação às diferentes ordenações. A complexidade do nível escalonável na tabela O(n^2) torna impraticável sua operação com grande conjuntos de dados, pois o número de comparações e substituições evolui quadraticamente em relação ao tamanho da lista.
Em comparação com os algoritmos como Merge Sort, Quick Sort e Insertion Sort, a eficiência torna-se complexa por considerar a aplicação corretamente devido a instabilidade de sua implementação.

6. APLICABILIDADE

6.1. Cenário Adequado

Adequado para cenários de aprendizagem inicial de conceitos algoritmos e programação, simulando breve cenários com o uso de pequenas limitações, ressaltando a simplicidade, o baixo nível de complexidade e contextos com restrições de uso de memória.

6.2. Cenário Inadequado

Inadequado para cenários com manipulação de grande volume de dados ou aplicações que requerem alta performance e eficiência em

tempo.

7. USO EM LINGUAGENS DE PROGRAMAÇÃO

O algoritmo de ordenação é uma tarefa funcional que manipula dados e organiza os elementos em uma determinada sequência. O método abordado (Bubble Sort) representa sua lógica de organização através da comparação e substituição entre os elementos adjacentes, permitindo compreender e visualizar o conceito de Estrutura de Repetição, por meio do Loop Aninhado e a Manipulação de Array/List.
A implementação desta função pode variar conforme a ferramenta utilizada, destacando uma leve mudança entre a sintaxe e os pequenos detalhes que consolidam a semântica da linguagem, reforçando a versatilidade e a adaptabilidade do pensamento algorítmico. Pode-se destacar a utilização desta abordagem para os níveis iniciais de aprendizagem, enfatizando a simplicidade e o fácil entendimento entre os conceitos fundamentais da Linguagem de Programação. Porém, sua funcionalidade não pode ser considerada eficiente em específicas aplicações com grande volume de dados.

7.1. Java

```java
import java.util.Scanner;
import java.util.Arrays;

public class OrdenacaoVetor {

    public static void bubbleSort(int[] vetor) {
        int n = vetor.length;
        int temp;

        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (vetor[j] > vetor[j + 1]) {
                    temp = vetor[j];
                    vetor[j] = vetor[j + 1];
                    vetor[j + 1] = temp;
                }
            }
        }
    }
```

```
    public static void main(String[] args) {
        final int TAMANHO = 8;
        int[] vetor = new int[TAMANHO];
        Scanner scanner = new Scanner(System.in);

        System.out.println("Digite 8 numeros inteiros:");

        for (int i = 0; i < TAMANHO; i++) {
            System.out.print("Elemento " + (i + 1) + ": ");
            vetor[i] = scanner.nextInt();
        }

        bubbleSort(vetor);

        System.out.println("Vetor Original: " + Arrays.toString(vetor));
        System.out.println("Vetor Ordenado: " + Arrays.toString(vetor));

        scanner.close();
    }
}
```

- Declaração
  - int[ ] vetor = new[ ] vetor;
- Entrada de dados
  - vetor[i] = scanner.nextInt();
- Loop Aninhado
  - Loop Externo
    - for (int i = 0; i < n; i ++)
  - Loop Interno
    - for (int j = 0; j< n - i - 1; j++)
- Saída
  - vetor[j + 1] = temp
  - bubble_sort(vetor)

7.2. Python

```python
def bubble_sort(vetor):
    """Implementa o algoritmo Bubble Sort para ordenar o vetor."""
    n = len(vetor)
    for i in range(n - 1):
        for j in range(0, n - i - 1):
            if vetor[j] > vetor[j + 1]:
                vetor[j], vetor[j + 1] = vetor[j + 1], vetor[j]  # Troca de forma
    return vetor

TAMANHO = 8
vetor = []

print("--- Ordenacao de Vetor em Python (Bubble Sort) ---")
print("Digite 8 numeros inteiros, um por linha:")

for i in range(TAMANHO):
    try:
        numero = int(input(f"Elemento {i + 1}: "))
        vetor.append(numero)
    except ValueError:
        print("Entrada invalida. Por favor, digite um numero inteiro.")
        i -= 1

vetor_original = list(vetor)

vetor_ordenado = bubble_sort(vetor)

print(f"\nVetor Original: {vetor_original}")
        print(f"Vetor Ordenado: {vetor_ordenado}")
```

- Declaração
    - def bubble_sort(vetor)
    - n = len(vetor)

- Entrada
    - numero = int (input (f” Elemento { i + 1 } “))

- Loop Externo
    - for i in range (n - 1)

- Loop Interno
    - for j in range (0, n - i - 1)

- Operação
    - vetor[j], vetor[j + 1] = vetor[j + 1], vetor[j]

- Saída
    - vetor_original = list(vetor)
    - vetor_ordenado = bubble_sort(vetor

7.3. C++

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    const int TAMANHO = 8;
    int vetor[TAMANHO];

      std::cout << "Digite 8 numeros inteiros para o vetor:" <<
std::endl;
    for (int i = 0; i < TAMANHO; ++i) {
        std::cin >> vetor[i];
    }

    // Bubble Sort
    for (int i = 0; i < TAMANHO - 1; ++i) {
        for (int j = 0; j < TAMANHO - i - 1; ++j) {
            if (vetor[j] > vetor[j+1]) {
                // Troca os elementos
                int temp = vetor[j];
                vetor[j] = vetor[j+1];
                vetor[j+1] = temp;
            }
        }
    }

    std::cout << "\nVetor ordenado:" << std::endl;
    for (int i = 0; i < TAMANHO; ++i) {
        std::cout << vetor[i] << " ";
    }
    std::cout << std::endl;

    return 0;
```

```
}
```

- Declaração
  - int vetor [tamanho]
- Entrada
  - std :: cin >> vetor[i]
- Operação
  - int temp = vetor[i]
  - vetor[j] = vetor[j + 1]
  - vetor[j + 1] = temp
- Saída
  - std :: cout << vetor[i]

8. EXPLICAÇÃO

8.1. Estrutura do Algoritmo

8.1.1. Entrada de Dados

- Cria um vetor de tamanho fixo (8 posições)

```
const int TAMANHO = 8;
```

- Usuário digita os valores que serão armazenados no vetor

```
std::cin >> vetor[i];
```

8.1.2. Processamento de Ordenação (Bubble Sort)

8.1.2.1. Loop Aninhado

- Controla o número de passagens

```
for (int i = 0; i < TAMANHO - 1; ++i) {
```

- Controla os pares de elementos adjacentes

```
for (int j = 0; j < TAMANHO - i - 1; ++j) {
```

8.1.2.2. Troca de posição

- Verificador de posição

```
if (vetor[j] > vetor[j+1]) {}
```

8.1.3. Saída de Dados

- Exibe o vetor ordenado em ordem crescente

```
for (int i = 0; i < TAMANHO; ++i) {
    std::cout << vetor[i] << " ";
}
```

8.2. Processamento

8.2.1. Vetor Inicial

- Usuário fornece 8 números

```
int vetor[ 8 ] = { 7, 4, 6, 5, 0, 1, 3, 2 };
```

8.2.2. Primeira passada (i = 0)

- Comparar os demais valores

```
int vetor[ 8 ] = ( 7 > 4 ) , ( 7 > 6 ), ( 7 > 5 ).....
```

- O maior valor vai ser direcionado à última posição.

```
int vetor[ 8 ] = { 4, 6, 5, 0, 1, 3, 2, 7 };
```

8.2.3. Segunda passada (i = 1)

- Agora, 2º maior valor deve ser direcionado à última posição

```
int vetor[ 8 ] = { 4, 5, 0, 1, 3, 2, 6, 7 };
```

8.2.4. Passada seguinte

- Cada passa fixa o próximo maior elemento

```
int vetor[ 8 ] = { 4, 0, 1, 3, 2, 5, 6, 7 };
```

8.2.5. Fim do processo

- Vetor totalmente ordenado

```
int vetor[ 8 ] = { 0, 1, 2, 3, 4, 5, 6, 7 };
```

9. CONCLUSÃO

O estudo do método ressalta a dualidade presente entre a simplicidade e sua limitação. O algoritmo apresenta uma ineficiência em termos de desempenho ao comparar com os demais métodos de ordenação, apesar de sua implementação ser clara e intuitiva para o desenvolvimento cognitivo de programação. Entende-se que, diferentemente de sua efetividade em cenários que demandam praticidade sob grande porte, o Bubble Sort cumpre sua responsabilidade ao possibilitar a compreensão de estruturas de

repetição, comparações sucessivas e os conceitos de complexidade algorítmica, utilizando como um recurso didático ao introduzir simples conceitos de ordenação.

Assim, entende-se que a relevância não se define apenas a sua aplicação prática, mas em sua contribuição aos conceitos acadêmicos ao unir a simplicidade de implementação e a capacidade de exemplificar os fundamentos que sustentam a ciência da computação. (ASTRACHAN, 2003).

10. REFERÊNCIAS BIBLIOGRÁFICAS

ASTRACHAN, O. An Archaeological Algorithmic Analysis. Duham: Computer Science Department, Duke University. 2003. Disponível em: https://users.cs.duke.edu/~ola/papers/bubble.pdf. Acesso em: 26 set, 2025.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. Algoritmos: Teoria e Prática. 3. ed. Tradução de Arlete Simille Marques. Rio de Janeiro: Elsevier, 2012. Acesso em: 26 set. 2025.

ELEMARJR. O que é e como funciona o Bubble Sort. [s. d.] Disponível em: https://elemarjr.com/clube-de-estudos/artigos/o-que-e-e-como-funciona-o-bubblesort/. Acesso em: 26 set, 2025

GATTO, Elaine C. Algoritmos de Ordenação: Bubble Sort. 16 ago, 2017. Disponível em: https://embarcados.com.br/algoritmos-de-ordenacao-bubble-sort/#Sobre-o-Bubble-Sort. Acesso em: 26 set, 2025.

SILVA, W. Análise de Complexidade de Algoritmos. 8 ago, 2020. Disponível em: https://growthcode.com.br/java/analise-de-complexidade-de-algoritmos/. Acesso em: 26 set, 2025.

SMART IA SOLUTIONS. O que é: Algoritmos de Ordenação. 28 set. 2024. Disponível em: https://smartiasolutions.com.br/glossario/o-que-e-algoritmos-de-ordenacao/. Acesso em: 26 set. 2025.

11. SLIDES

https://www.canva.com/design/DAG0DemsY1o/3HQCZVe2ojfzgB_IXE_45Q/edit?utm_content=DAG0DemsY1o&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton