Seminário sobre os Algoritmos Bubble Sort e Quicksort

Ciro Guilherme Nass
Alexandre Raphael Marques dr Freitas

16 de setembro de 2025

Resumo

Este relatório apresenta dois algoritmos de ordenação: Bubble Sort e Quicksort. São abordadas suas ideias, implementações, complexidade computacional, análises empíricas, uso de memória, estabilidade, versões recursivas e iterativas. Ao final, é feita uma comparação geral entre ambos.

Sumário

1 Bubble Sort **3**
- 1.1 Introdução . . . 3
- 1.2 Ideia do Algoritmo . . . 3
- 1.3 Implementação . . . 3
  - 1.3.1 Pseudocódigo . . . 3
  - 1.3.2 Implementação em C . . . 3
- 1.4 Complexidade Computacional . . . 4
- 1.5 Complexidade Empírica . . . 4
- 1.6 Uso de Memória . . . 4
- 1.7 Estabilidade . . . 4
- 1.8 Versões Recursiva e Iterativa . . . 5
  - 1.8.1 Iterativa . . . 5
  - 1.8.2 Recursiva . . . 5
- 1.9 Resumo do Método . . . 5

2 Quicksort **5**
2.1 Introdução 5
2.2 Ideia do Algoritmo 6
2.3 Implementação 6
2.3.1 Pseudocódigo 6
2.3.2 Implementação em C 7
2.4 Complexidade Computacional 7
2.5 Complexidade Empírica 8
2.6 Uso de Memória 8
2.7 Estabilidade 8
2.8 Versões Recursiva e Iterativa 8
2.8.1 Recursiva 8
2.8.2 Iterativa 9
2.9 Resumo do Método 9

3 Comparação entre Bubble Sort e Quicksort **10**
3.1 Comparação Empírica 10

4 Bibliografia **11**

1 Bubble Sort

1.1 Introdução

Algoritmos de ordenação são fundamentais em ciência da computação. O Bubble Sort, apesar de pouco eficiente em termos práticos, é amplamente usado em contextos didáticos devido à sua simplicidade.

1.2 Ideia do Algoritmo

O Bubble Sort percorre repetidamente a lista, comparando elementos adjacentes e trocando-os se estiverem fora de ordem. O processo se repete até que nenhuma troca seja necessária, indicando que a lista está ordenada.

1.3 Implementação

1.3.1 Pseudocódigo

```
procedure BubbleSort(A)
    n := comprimento(A)
    para i de 1 até n-1 faça
        para j de 0 até n-i-1 faça
            se A[j] > A[j+1] então
                trocar A[j] e A[j+1]
```

1.3.2 Implementação em C

```c
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int trocou = 0;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                trocou = 1;
            }
        }
        if (!trocou) break;
    }
}
```

1.4 Complexidade Computacional

- Melhor caso: $O(n)$ (quando a lista já está ordenada).
- Pior caso: $O(n^2)$ (quando a lista está em ordem inversa).
- Caso médio: $O(n^2)$.

Situação	Comparações	Trocas	Complexidade
Melhor caso	$n-1$	0	$O(n)$
Pior caso	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	$O(n^2)$
Caso médio	$\approx \frac{n^2}{2}$	$\approx \frac{n^2}{4}$	$O(n^2)$

Tabela 1: Complexidade do Bubble Sort

1.5 Complexidade Empírica

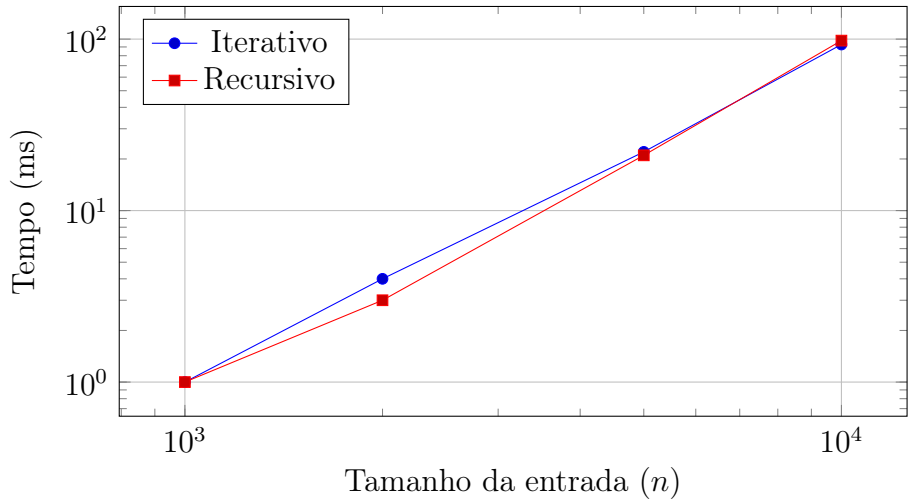

Figura 1: Desempenho do Bubble Sort em C (Iterativo vs Recursivo).

1.6 Uso de Memória

O algoritmo utiliza apenas uma variável auxiliar para realizar as trocas. Logo, o uso de memória é $O(1)$.

1.7 Estabilidade

O Bubble Sort é estável, pois não altera a ordem relativa de elementos iguais.

1.8 Versões Recursiva e Iterativa

1.8.1 Iterativa

Já apresentada no código acima.

1.8.2 Recursiva

```
void bubbleSortRecursivo(int arr[], int n) {
    if (n == 1) return;

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

1.9 Resumo do Método

- Complexidade: $O(n^2)$ no pior e caso médio, $O(n)$ no melhor caso.
- Memória: $O(1)$.
- Estabilidade: Estável.
- Implementação: Iterativa e recursiva.

2 Quicksort

2.1 Introdução

O Quicksort é um dos mais eficientes algoritmos de ordenação por comparação, sendo amplamente utilizado na prática devido ao seu bom desempenho médio.

2.2 Ideia do Algoritmo

O Quicksort segue a estratégia de *divisão e conquista*: escolhe-se um elemento pivô, particiona-se o vetor em dois subvetores (um com elementos menores que o pivô e outro com os maiores), e aplica-se recursivamente o mesmo processo aos subvetores até que estejam ordenados.

2.3 Implementação

2.3.1 Pseudocódigo

```
procedure QuickSort(A, inicio, fim)
    se inicio < fim então
        p := Particiona(A, inicio, fim)
        QuickSort(A, inicio, p-1)
        QuickSort(A, p+1, fim)

procedure Particiona(A, inicio, fim)
    pivô := A[fim]
    i := inicio - 1
    para j de inicio até fim-1 faça
        se A[j] <= pivô então
            i := i + 1
            trocar A[i] e A[j]
    trocar A[i+1] e A[fim]
    retornar i+1
```

2.3.2 Implementação em C

```c
int particiona(int arr[], int baixo, int alto) {
    int pivo = arr[alto];
    int i = (baixo - 1);
    for (int j = baixo; j < alto; j++) {
        if (arr[j] <= pivo) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[alto];
    arr[alto] = temp;
    return (i + 1);
}

void quickSort(int arr[], int baixo, int alto) {
    if (baixo < alto) {
        int pi = particiona(arr, baixo, alto);
        quickSort(arr, baixo, pi - 1);
        quickSort(arr, pi + 1, alto);
    }
}
```

2.4 Complexidade Computacional

- Melhor caso: $O(n \log n)$ (quando as partições são equilibradas).
- Pior caso: $O(n^2)$ (quando sempre se escolhe o pior pivô).
- Caso médio: $O(n \log n)$.

Situação	Comparações	Trocas	Complexidade
Melhor caso	$\approx n \log n$	$\approx n \log n$	$O(n \log n)$
Pior caso	$\frac{n(n-1)}{2}$	$\frac{n(n-1)}{2}$	$O(n^2)$
Caso médio	$\approx n \log n$	$\approx n \log n$	$O(n \log n)$

Tabela 2: Complexidade do Quicksort

2.5 Complexidade Empírica

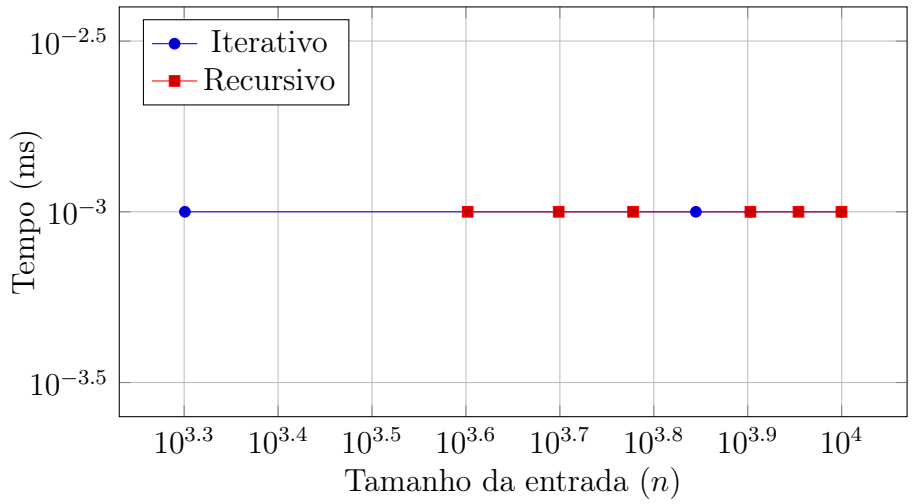

Figura 2: Desempenho do Quick Sort em C (Iterativo vs Recursivo).

2.6 Uso de Memória

O Quicksort é um algoritmo *in-place*, pois requer apenas memória extra para a pilha de chamadas recursivas. Assim, o uso de memória é $O(\log n)$ em média.

2.7 Estabilidade

O Quicksort **não é estável**, pois elementos iguais podem ter sua ordem relativa alterada durante o particionamento.

2.8 Versões Recursiva e Iterativa

2.8.1 Recursiva

Já apresentada no código acima.

2.8.2 Iterativa

```
void quickSortIterativo(int arr[], int baixo, int alto) {
    int pilha[alto - baixo + 1];
    int topo = -1;

    pilha[++topo] = baixo;
    pilha[++topo] = alto;

    while (topo >= 0) {
        alto = pilha[topo--];
        baixo = pilha[topo--];

        int p = particiona(arr, baixo, alto);

        if (p - 1 > baixo) {
            pilha[++topo] = baixo;
            pilha[++topo] = p - 1;
        }
        if (p + 1 < alto) {
            pilha[++topo] = p + 1;
            pilha[++topo] = alto;
        }
    }
}
```

2.9 Resumo do Método

- Complexidade: $O(n \log n)$ no melhor e caso médio, $O(n^2)$ no pior caso.
- Memória: $O(\log n)$ em média.
- Estabilidade: Não é estável.
- Implementação: Recursiva e iterativa.

3 Comparação entre Bubble Sort e Quicksort

Critério	Bubble Sort	Quicksort
Complexidade (melhor caso)	$O(n)$	$O(n \log n)$
Complexidade (caso médio)	$O(n^2)$	$O(n \log n)$
Complexidade (pior caso)	$O(n^2)$	$O(n^2)$
Uso de memória	$O(1)$	$O(\log n)$
Estabilidade	Estável	Não estável
Aplicação prática	Didática	Ordenação eficiente em geral

Tabela 3: Comparação geral entre Bubble Sort e Quicksort

3.1 Comparação Empírica

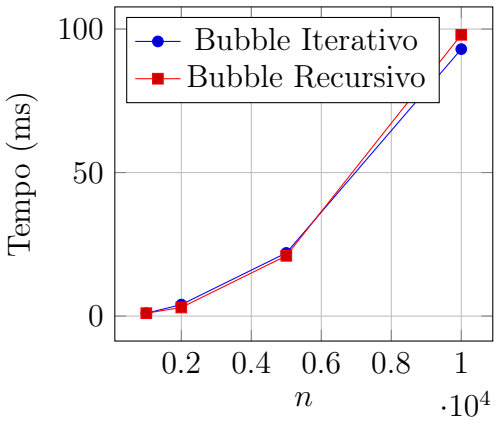

Figura 3: Bubble Sort

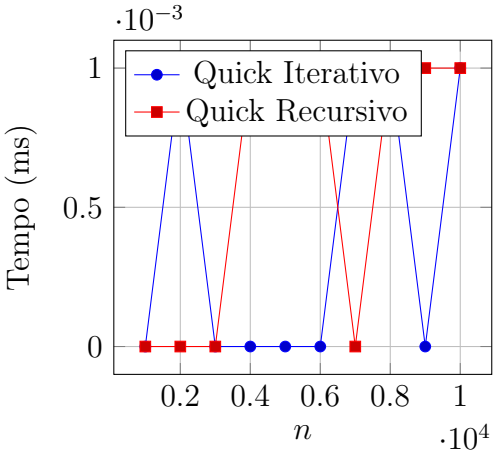

Figura 4: Quicksort

O Bubble Sort é mais simples e indicado para fins didáticos, mas é ineficiente em termos de desempenho. Já o Quicksort é amplamente utilizado na prática, pois apresenta bom desempenho médio ($O(n \log n)$) e baixa sobrecarga de memória.

4 Bibliografia

- GeeksforGeeks. Bubble Sort. Disponível em: `https://www.geeksforgeeks.org/bubble-sort/`.
- GeeksforGeeks. Quicksort. Disponível em: `https://www.geeksforgeeks.org/quick-sort/`.