Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação

Algoritmos de Ordenação

Introdução à Ciência da Computação II

Discentes:
Cleyton José Rodrigues Macedo
Kauã Benjamin Trombim Silva

Docente:
Marcelo Garcia Manzato

São Carlos
Novembro de 2025

1 Os Algoritmos de Ordenação

Abaixo apresentamos brevemente os algoritmos analisados neste relatório.

- **Bubble Sort:** Algoritmo de troca simples. Varre o vetor repetidamente comparando elementos adjacentes e trocando-os se estiverem fora de ordem, fazendo com que os maiores valores "flutuem" para o final.
- **Selection Sort:** Algoritmo de seleção direta. Percorre a parte não ordenada do vetor para encontrar o menor elemento e o troca com a primeira posição disponível dessa parte.
- **Insertion Sort:** Algoritmo de construção incremental. Percorre o vetor elemento por elemento, inserindo cada um na sua posição correta dentro da sublista já ordenada à esquerda.
- **Shell Sort:** Refinamento do Insertion Sort. Permite a troca de registros distantes uns dos outros usando um intervalo ("gap") que diminui progressivamente.
- **Quick Sort:** Algoritmo de "Dividir para Conquistar". Escolhe um elemento como pivô, particiona o vetor colocando os menores à esquerda e os maiores à direita do pivô, e ordena as sublistas recursivamente.
- **Heap Sort:** Algoritmo baseado em estrutura de dados (*Heap Máximo*). Organiza os dados em uma árvore binária onde o pai é maior que os filhos.
- **Merge Sort:** Algoritmo de "Dividir para Conquistar". Divide o vetor recursivamente pela metade até obter unidades indivisíveis e, em seguida, intercala (*merge*) os subvetores.
- **Counting Sort:** Algoritmo de ordenação linear (não comparativo). Conta a frequência de ocorrência de cada valor distinto para determinar a posição exata no vetor de saída.
- **Radix Sort:** Algoritmo de ordenação não comparativo. Ordena o vetor processando dígito por dígito (unidades, dezenas, etc.).

2 Os Dados Obtidos

2.1 Contando as Comparações

Para contabilizar as comparações e as trocas, utilizou-se um vetor de inteiros auxiliar, denominado `counts`, passado como argumento para cada função. O primeiro campo (`counts[0]`) armazena o número de comparações realizadas, e o segundo (`counts[1]`) o número de trocas.

Exemplo no Bubble Sort:

```
1 void bubble_sort(int* array, int n, long long int* counts){
2     for(int i = 0; i < n; i++){
3         for(int j = 0; j < n - i - 1; j++){
4             counts[0]++;
```

```
            if(array[j] > array[j + 1]){
                swap(&array[j], &array[j + 1]);
                counts[1]++;
            }
        }
    }
}
```

Listing 1: Exemplo de instrumentação no Bubble Sort

O ponteiro `int* counts` é incrementado sempre que ocorre uma operação relevante. Note que `counts[0]++` ocorre antes da comparação do `if` interno, e `counts[1]++` é acionado caso a função `swap` seja executada.

2.2 Medindo o Tempo

Para medir o tempo de execução, utilizou-se a biblioteca `time.h`, capturando o número de *clocks* do processador antes e depois da execução.

```
    clock_t inicio, fim;
    double tempo_cpu;
```

Listing 2: Variáveis para auxiliar a medição de tempo

```
    inicio = clock();
    switch(sort){
        case 1:
            bubble_sort(arr, n, counts);
            break;
    ...
        default:
            printf("Opcao invalida.\n");
    }

    fim = clock();
    tempo_cpu = ((double) (fim - inicio)) / CLOCKS_PER_SEC;
```

Listing 3: Cálculo do tempo de CPU

2.3 Gerando Vetores Aleatórios

Para a geração de vetores aleatórios, utilizou-se a função `get_random`, implementando um gerador do tipo *Xorshift*. Os testes foram realizados com 5 sementes (*seeds*) diferentes para garantir margem de confiança.

```
int get_random(int *state, int max) {
    uint32_t x = (uint32_t) *state;
    x ^= x << 13;
    x ^= x >> 17;
```

```
    x ^= x << 5;
    *state = x;
    return (int)((x % max) + 1);
}
```

Listing 4: Função pseudoaleatória Xorshift

3 Análise dos Resultados

3.1 Vetores Ordenados

- **Insertion Sort:** Foi o melhor algoritmo para o caso ordenado. A condição do laço interno `while (j >= 0)` falha na primeira comparação para cada elemento, resultando em complexidade linear $O(N)$.
- **Selection e Bubble Sort:** Foram os piores. Excluindo o Bubble Sort (que pode ser otimizado com *flag*), estes algoritmos são "cegos" à ordenação prévia, realizando sempre cerca de $\frac{N^2}{2}$ comparações.

3.2 Vetores Inversos

- **Contagem e Radix Sort:** Os melhores algoritmos neste caso. Como a ordem dos números não afeta o processamento dos dígitos ou a contagem de frequência, o desempenho permanece linear.
- **Insertion, Selection e Bubble Sort:** Apresentaram desempenho inviável (tempos acima de 10s para grandes entradas). O Insertion Sort sofre o pior caso, deslocando cada elemento por todo o vetor. O Bubble Sort realiza uma troca a cada comparação.

3.3 Vetores Aleatórios

- **Contagem e Radix Sort:** Mantêm-se como os melhores, pois a ordenação de inteiros simples favorece algoritmos não comparativos.
- **Quick Sort:** O melhor dentre os baseados em comparações. Devido às operações simples e boa localidade de cache, mantém um tempo médio de $O(N \log N)$.
- **Algoritmos Quadráticos:** Insertion, Selection e Bubble Sort tiveram o pior desempenho, confirmando a complexidade $O(N^2)$ no caso médio.

4 Tabelas de Resultados

Abaixo estão listadas as tabelas comparativas. Note que algoritmos não baseados em comparação podem apresentar valores no campo "Comparações" devido à busca auxiliar pelo maior valor do vetor (necessária para alocar memória).

Tabela 1: Resultados para Vetor **Ordenado** (Tamanho $N = 100$)

Algoritmo	**Comparações**	**Trocas**	**Tempo (s)**
Bubble Sort	4.950	0	0.000013
Selection Sort	4.950	0	0.000013
Insertion Sort	99	99	0.000003
Shell Sort	334	334	0.000006
Quick Sort	669	345	0.000010
Heap Sort	1.382	641	0.000021
Merge Sort	356	1.344	0.000009
Counting Sort	99	200	0.000003
Radix Sort	99	600	0.000008

Tabela 2: Resultados para Vetor **Ordenado** (Tamanho $N = 1.000$)

Algoritmo	**Comparações**	**Trocas**	**Tempo (s)**
Bubble Sort	499.500	0	0.001656
Selection Sort	499.500	0	0.001044
Insertion Sort	999	999	0.000009
Shell Sort	6.182	6.182	0.000023
Quick Sort	9.520	4.960	0.000074
Heap Sort	20.418	9.709	0.000245
Merge Sort	5.044	19.952	0.000092
Counting Sort	999	2.000	0.000017
Radix Sort	999	8.000	0.000068

Tabela 3: Resultados para Vetor **Ordenado** (Tamanho $N = 10.000$)

Algoritmo	**Comparações**	**Trocas**	**Tempo (s)**
Bubble Sort	49.995.000	0	0.113231
Selection Sort	49.995.000	0	0.104658
Insertion Sort	9.999	9.999	0.000033
Shell Sort	93.187	93.187	0.000321
Quick Sort	131.343	66.421	0.000545
Heap Sort	273.914	131.957	0.001831
Merge Sort	69.008	267232	0.000853
Counting Sort	9.999	20.000	0.000152
Radix Sort	99999	100.000	0.000812

Tabela 4: Resultados para Vetor **Ordenado** (Tamanho $N = 100.000$)

Algoritmo	**Comparações**	**Trocas**	**Tempo (s)**
Bubble Sort	4.999.950.000	0	11.418500
Selection Sort	4.999.950.000	0	10.481142
Insertion Sort	99.999	99.999	0.000324
Shell Sort	1.266.128	1.266.128	0.004290
Quick Sort	1.665.551	846.100	0.005643
Heap Sort	3.401.710	1.650.855	0.016890
Merge Sort	853.904	3337.856	0.010510
Counting Sort	99.999	200.000	0.001805
Radix Sort	99.999	1.200.000	0.007996

4.1 Vetores Inversos

Tabela 5: Resultados para Vetor **Inverso** (Tamanho $N = 100$)

Algoritmo	**Comparações**	**Trocas**	**Tempo (s)**
Bubble Sort	4.950	4.950	0.000025
Selection Sort	4.950	50	0.000035
Insertion Sort	4.950	5.049	0.000034
Shell Sort	575	650	0.000008
Quick Sort	868	556	0.000014
Heap Sort	1.134	517	0.000009
Merge Sort	316	1.344	0.000009
Counting Sort	99	200	0.000003
Radix Sort	99	600	0.000008

Tabela 6: Resultados para Vetor **Inverso** (Tamanho $N = 1.000$)

Algoritmo	**Comparações**	**Trocas**	**Tempo (s)**
Bubble Sort	499.500	499.500	0.002285
Selection Sort	499.500	500	0.001517
Insertion Sort	499.500	500.499	0.001579
Shell Sort	9.046	9.690	0.000035
Quick Sort	15.849	9.583	0.000135
Heap Sort	17.634	8.317	0.000103
Merge Sort	4.932	19.952	0.000080
Counting Sort	999	2.000	0.000017
Radix Sort	999	8.000	0.000091

Tabela 7: Resultados para Vetor **Inverso** (Tamanho $N = 10.000$)

Algoritmo	**Comparações**	**Trocas**	**Tempo (s)**
Bubble Sort	49.995.000	49.995.000	0.231528
Selection Sort	49.995.000	5.000	0.105361
Insertion Sort	49.995.000	50.004.999	0.130056
Shell Sort	130.764	136.317	0.000460
Quick Sort	234.923	139.308	0.000960
Heap Sort	243.394	116.697	0.001346
Merge Sort	64.608	267.232	0.000928
Counting Sort	9.999	20.000	0.000156
Radix Sort	9.999	100.000	0.000826

Tabela 8: Resultados para Vetor **Inverso** (Tamanho $N = 100.000$)

Algoritmo	**Comparações**	**Trocas**	**Tempo (s)**
Bubble Sort	4.999.950.000	4.999.950.000	22.923961
Selection Sort	4.999.950.000	50.000	10.649822
Insertion Sort	4.999.950.000	5.000.049.999	12.903007
Shell Sort	1.716.832	1.776.756	0.006472
Quick Sort	3.107.283	1.814.032	0.010976
Heap Sort	3.094.870	1.497.435	0.016492
Merge Sort	815.024	33.37.856	0.010206
Counting Sort	99.999	200.000	0.001705
Radix Sort	99.999	1.200.000	0.008140

4.2 Vetores Aleatórios

Tabela 9: Resultados para Vetor **Aleatório** (Tamanho $N = 100$)

Algoritmo	**Comparações**	**Trocas**	**Tempo (s)**
Bubble Sort	4.950	2.442	0.000039
Selection Sort	4.950	95	0.000020
Insertion Sort	2.497	2.501	0.000012
Shell Sort	721	760	0.000011
Heap Sort	1.263	581	0.000013
Quick Sort	745	416	0.000012
Merge Sort	545	1344	0.000013
Counting Sort	99	200	0.000003
Radix Sort	99	560	0.000008

Tabela 10: Resultados para Vetor **Aleatório** (Tamanho $N = 1.000$)

Algoritmo	**Comparações**	**Trocas**	**Tempo (s)**
Bubble Sort	499.500	248.870	0.002001
Selection Sort	499.500	991	0.001533
Insertion Sort	249.209	249.217	0.000841
Shell Sort	12.984	13.494	0.000107
Heap Sort	19.154	9.077	0.000169
Quick Sort	11.222	6.258	0.000081
Merge Sort	8.699	19.952	0.000136
Counting Sort	999	2.000	0.000017
Radix Sort	999	7.200	0.000063

Tabela 11: Resultados para Vetor **Aleatório** (Tamanho $N = 10.000$)

Algoritmo	**Comparações**	**Trocas**	**Tempo (s)**
Bubble Sort	49.995.000	24.847.170	0.215027
Selection Sort	49.995.000	9.988	0.105016
Insertion Sort	24.866.084	24.866.092	0.064353
Shell Sort	194.361	199.261	0.001572
Heap Sort	258.486	124.243	0.001913
Quick Sort	153.155	81.160	0.000894
Merge Sort	120.424	267.232	0.001585
Counting Sort	9.999	20.000	0.000159
Radix Sort	9.999	100.000	0.000804

Tabela 12: Resultados para Vetor **Aleatório** (Tamanho $N = 100.000$)

Algoritmo	**Comparações**	**Trocas**	**Tempo (s)**
Bubble Sort	4.999.950.000	2.496.075.884	27.062839
Selection Sort	4.999.950.000	99.987	10.423964
Insertion Sort	2.496.693.971	2.496.693.983	6.487136
Shell Sort	2.595.685	2.647.105	0.016229
Quick Sort	1.942.094	1.063.726	0.011184
Heap Sort	3.249.890	1.574.945	0.021238
Merge Sort	1.536.339	3.337.856	0.019250
Counting Sort	99.999	200.000	0.002068
Radix Sort	99.999	1.160.000	0.007748