Estruturas de Dados e Análise de Algoritmos - EDAA
Proposta 2 – Métodos de Ordenação Quadráticos

Renan Tashiro[1], **Rodrigo Schmidt Nurmberg**[1]

[1]Programa de Pós Graduação em Ciência da Computação (PPGCOMP)
Universidade Estadual do Oeste do Paraná (UNIOESTE)
Rua Universitária, 2069 Bloco B – Bairro Universitário – 85819-110 – Cascavel – PR

renantashiro@hotmail.com, rodrigo.nurmberg@unioeste.br

Abstract. *This article investigates the efficiency of sorting algorithms, a fundamental operation in computing. Three algorithms with quadratic complexity in the average case and one with quadratic complexity in the worst case were chosen. Factors such as the state of the input data and the way they are sorted were also considered. The algorithms analyzed are Bubble Sort, Selection Sort, Insertion Sort, and Bucket Sort, implemented in the C language, and their behaviors were compared empirically in different testing scenarios. The analysis of these methods contributes to a deeper understanding of how the particularities of an algorithm's implementation affect its performance, even though they may have equivalent asymptotic complexity.*

Resumo. *Este artigo investiga a eficiência dos algoritmos de ordenação, uma operação fundamental na computação. Foram escolhidos três algoritmos de complexidade quadrática no caso médio e um com complexidade quadrática no pior caso. Também foram considerados fatores como o estado dos dados de entrada e a maneira como estão ordenados. Os algoritmos analisados são o Bubble Sort, Selection Sort, Insertion Sort e Bucket Sort, implementados na linguagem C, e seus comportamentos foram comparados empiricamente em diferentes cenários de testes. A análise desses métodos contribui para um entendimento mais profundo de como as particularidades de implementação de um algoritmo afetam seu desempenho, mesmo que possuam complexidade assintóticas equivalentes.*

1. Introdução

Ordenação é uma operação essencial na computação, por ser utilizada com frequência, mesmo pequenas otimizações podem gerar grandes impactos, principalmente ao operarem sobre grandes conjuntos de dados.

Na ordenação, para um dado conjunto de entrada $\langle a_1, a_2, ..., a_n \rangle$, gera-se uma permutação[1]. da sequência de entrada $\langle a_1^{'}, a_2^{'}, ..., a_n^{'} \rangle | a_i^{'} \leq a_{i+1}^{'} \forall i_1^{n-1}$.

A escolha do algoritmo de ordenação mais adequado, depende da natureza de cada aplicação e deve considerar, entre outros fatores, se os dados encontram-se ordenados, a forma como os dados estão armazenados (unidades de acesso aleatório ou sequencial,

[1]Também é possível ordenar os valores de forma decrescente: $\langle a_1^{'}, a_2^{'}, ..., a_n^{'} \rangle | a_i^{'} \geq a_{i+1}^{'} \forall i_1^{n-1}$

cada qual com sua velocidade), as estruturas de dados utilizadas (vetores, listas encadeadas, árvores, etc), quantidade de itens a ordenar e como estão distribuídos (uniformemente ou não) e também a arquitetura do computador (com diferentes operações e seus custos associados).

Este trabalho está organizado em seções. A Seção 2 apresenta os métodos de ordenação analisados neste trabalho, implementados na linguagem de programação C e acompanhados da análise assintótica dos respectivos códigos. Já a Seção 3 detalha os materiais e métodos empregados. Na Seção 4 são apresentados os resultados obtidos e por fim, na Seção 5 conclui-se o trabalho, fazendo-se uma breve discussão dos resultados.

2. Algoritmos de ordenação

Os algoritmos, de ordenação, recebem como como entrada o vetor a ser ordenado $arr_{i=1}^{n}\langle a_1, a_2, ..., a_n\rangle$, o tamanho do vetor n e a referência $swaps$, para o contador de trocas realizadas durante a ordenação do vetor. Como as implementações realizadas alteram o vetor de entrada, não é necessário retornar nenhum valor. Cabe ressaltar, que os algoritmos implementados realizam a ordenação crescente de valores numéricos.

Apesar da análise assintótica considerar que todas as operações possuem o mesmo custo, os algoritmos de ordenação costumam ser caracterizados pelo número de comparações (do tipo $>$ ou $<$) e/ou pelo número de trocas, entre os elementos do arranjo sendo ordenado, ainda que essas não sejam operações atômicas[2]. As operações de troca tendem a ser mais custosas que as operações de comparação.

As 4 subseções a seguir, apresentam os códigos-fonte dos algoritmos de ordenação analisados neste trabalho. Os códigos contém comentários, linha a linha, referentes à análise assintótica dos mesmos. Foram adotadas as abreviações da Tabela 1 na redação dos comentários. São feitas pequenas discussões, no corpo do texto, a respeito dos pontos mais relevantes das análises.

a: atribuição	s: subtração	d: divisão	c: chamada à função
i: indexação	ss: decremento	m: multiplicação	N: número de elementos do vetor
p: adição	pp: incremento	t: teste lógico	RQN: raiz quadrada de N ($\sqrt{N}$)

Tabela 1. Abreviações utilizadas nos comentários de análise sintática

2.1. Bubble Sort

O algoritmo de ordenação *Bubble Sort*, compara cada elemento com seus sucessores, caso este seja maior, realiza a troca, invertendo as posições dos elementos no arranjo. Esse processo é repetido do primeiro ao penúltimo elemento do vetor. A cada rodada n, o $n - ésimo$ maior elemento estará em sua posição final. O quadro Código 1, exibe a implementação utilizada neste trabalho.

A complexidade assintótica foi calculada considerando que no pior cenário, quando os dados encontram-se ordenados de forma decrescente, a comparação da linha 6 sempre será avaliada como verdadeira, executando assim as linhas 7 a 10, todas as vezes. Desse modo, para cada comparação será realizada também uma troca.

[2]Operações realizadas de forma indivisível, através de uma única operação/instrução.

```
1  void bubbleSort(int arr[], int n, long long *swaps) {
2    int temp;
3    *swaps = 0;                              // 1i + 1a
4    for (int i = 0; i < n - 1; i++)          // 1a + 1s + (N-1)(1t + 1pp) + 1t
5      for (int j = 0; j < n - i - 1; j++)    // (N-1)*(1a + 1s + 1t)+(N-1+1)(N-1)/2(1t + 1pp)
6        if (arr[j] > arr[j + 1]) {           // (N²-N)/2*(1i + 1t + 1i + 1p)
7          temp = arr[j];                     // (N²-N)/2*(1a + 1i)
8          arr[j] = arr[j + 1];               // (N²-N)/2*(1i + 1a + 1i + 1p)
9          arr[j + 1] = temp;                 // (N²-N)/2*(1i + 1p + 1a)
10         (*swaps)++;                        // (N²-N)/2*(1i + 1pp)
11       }
12 }                                          // 8,5N² - 3,5N
```

Código 1: Bubble Sort em C

O laço de repetição da linha 4, é executado $N-1$ vezes, pois apesar do valor iniciar em 0, utilizou-se a comparação $<$, contra o valor $N-1$. Já, para determinar o número de vezes que o laço da linha 5 é executado, calculou-se a soma da progressão aritmética (PA) na forma $S_a = \frac{(a_1+a_n)\times n}{2}$. Na primeira iteração, o laço executa $a_1 = N-1$ vezes e na última iteração, $a_n = 1$ vez, nas $n = N-1$ iterações do laço externo (linha 4). Chegando-se ao valor $\frac{N^2-N}{2}$.

As operações de atribuição (linha 3) e incremento (linha 10), sob a referência $swaps$, incluem uma operação adicional de indexação, correspondente à "desreferenciação"[3] do ponteiro. Para os laços de repetição cujo o teste lógico contém uma expressão aritmética que não se altera com a execução das iterações, como o *for* da linha 3, considerou-se que a expressão é resolvida uma única vez, na primeira execução do laço.

Seguindo o raciocino acima, cuja memória de cálculo encontra-se descrita na forma de comentários no código-fonte do quadro Código 1, chegou-se ao polinômio de custo $8,5N^2 - 3,5N$ (linha 12), implicando numa complexidade $O(n^2)$ para o método *Bubble Sort*.

2.2. Selection Sort

O objetivo do *Selection Sort* é reduzir o número de trocas, localizando, a cada iteração, do início para o fim do vetor, o menor valor ainda não ordenado, e trocando-o para sua posição correta. A implementação do *Selection Sort* utilizada neste trabalho, contendo comentários da análise assintótica, encontra-se no quadro Código 2.

Assim como no *Bubble Sort*, o laço de repetição da linha 4 é executado $N-1$ vezes. O número de execuções do laço da linha 6 segue a soma da PA de $a_1 = N-1$, $a_n = 1$ e $n = N-1$, resultando em $\frac{N^2-N}{2}$ iterações.

A complexidade assintótica foi calculada com base no pior cenário, quando os dados encontram-se ordenados, exceto pelo menor valor que está localizado na última posição do arranjo. Em cada iteração, o menor valor ainda não ordenado estará na última posição do arranjo, e ao ordená-lo, o menor valor subsequente não ordenado será movido para o final do vetor. Dessa forma, a comparação da linha 7 sempre será avaliada como

[3] Acesso à posição de memória referenciada pelo ponteiro.

```c
void selectionSort(int arr[], int n, long long *swaps) {
  int minIndex, temp;
  *swaps = 0;                          // 1i + 1a
  for (int i = 0; i < n - 1; i++) {    // 1a + 1t + 1s + (N-1)(1t +1pp)
    minIndex = i;                      // (N-1)*(1a)
    for (int j = i + 1; j < n; j++)    // (N-1)*(1a + 1t + 1p) + (N-1+1)(N-1)/2(1t + 1pp)
      if (arr[j] < arr[minIndex])      // (N²-N)/2*(1i + 1t + 1i)
        minIndex = j;                  // (N²-N)/2*(1a)
    if (minIndex != i) {               // (N-1)*(1t)
      temp = arr[minIndex];            // (N-1)*(1a + 1i)
      arr[minIndex] = arr[i];          // (N-1)*(1i + 1a + 1i)
      arr[i] = temp;                   // (N-1)*(1i + 1a)
      (*swaps)++;                      // (N-1)*(1i + 1a)
    }
  }
}                                      // 2N² + 15N - 11
```

Código 2: Selection Sort em C

verdadeira, executando a linha 8, e consequentemente o teste da linha 9 também sempre será avaliado como verdadeiro, incorrendo em N-1 trocas (linhas 10 a 13). Obteve-se assim, o polinômio de custo $2N^2 + 15N - 11$ (linha 16) e complexidade $O(n^2)$.

2.3. Insertion Sort

A estratégia do *Insertion Sort* baseia-se em encontrar a posição correta para um determinado elemento e só então realizar a troca, iniciando no segundo elemento e avançando até o último, compara-se o elemento com seus antecessores para determinar se este é menor que aqueles, e portanto deve ser trocado de posição.

```c
void insertionSort(int arr[], int n, long long *swaps) {
  int key, j;
  *swaps = 0;                          // 1i + 1a
  for (int i = 1; i < n; i++) {        // 1a + 1t + (N-1)(1t + 1pp)
    key = arr[i];                      // (N-1)*(1a + 1i)
    j = i - 1;                         // (N-1)*(1a + 1s)
    while (j >= 0 && arr[j] > key) {   // (N)*(1t + 1t + 1i + 1t)
      arr[j + 1] = arr[j];             // (N²-N)/2*(1i + 1p + 1a + 1i)
      (*swaps)++;                      // (N²-N)/2*(1i + 1pp)
      j = j - 1;                       // (N²-N)/2*(1a + 1s)
    }
    arr[j + 1] = key;                  // (N-1)*(1i + 1p + 1a)
  }
}                                      // 4N² + 9N - 5
```

Código 3: Insertion Sort em C

Assim como no *Selection Sort*, o laço de repetição da linha 4 é executado $N - 1$ vezes. Já o laço da linha 7, repete-se por $\frac{N^2-N}{2}$ rodadas, de acordo a soma da PA de $a_1 = 1$, $a_n = N - 1$ e $n = N - 1$. O pior caso do *Insertion Sort* ocorre ao ordenar vetores decrescentes, fazendo com que o elemento sob análise (linha 5) seja menor que todos seus

anteriores (linhas 7 a 11), sendo necessário deslocá-los, para liberar a posição correta, para a troca com o menor elemento (linha 12). Foi obtido o polinômio $4N^2 + 9N - 5$ (linha 14), de complexidade $O(n^2)$.

É possível encontrar na literatura, quem considere que o *Insertion Sort* faça apenas $O(n)$ trocas. As trocas realizadas para liberar a posição correta para o elemento sendo ordenado, são consideradas "atribuições", essas sim com ordem $O(n^2)$. Cabe ressaltar, que adotou-se o entendimento que as "atribuições" também constituem trocas, dessa foram foram contabilizadas na linha 9. Caso fosse adotado o entendimento de que as "atribuições" não são trocas, a contagem das trocas deveria ser movida para fora do laço da linha 7, deixando-a junto da troca, que é realizada na linha 12.

2.4. Bucket Sort

O *Bucket Sort* utiliza a abordagem dividir para conquistar. Os elementos são agrupados[4] em subconjuntos menores, denominados *buckets*, que por sua vez são ordenados, utilizando um método de ordenação qualquer, neste trabalho adotou-se o *Insertion Sort*, em seguida, os subconjuntos, individualmente ordenados, são concatenados gerando o arranjo ordenado.

Considerando que n é um número inteiro, positivo e maior que 0, a linha 12 do quadro Código 4, nunca será executada, assim como a linha 30, pois isso implicaria em ter, no arranjo um elemento, cujo valor seja maior que o máximo elemento do conjunto acrescido de 1. De forma que lhes foi atribuído o multiplicador 0 na análise assintótica.

Pela linha 11, assume-se que foram criados $\sqrt{N}$ *buckets*, assim, os laços de repetição regidos pelo número de *buckets* (linhas 22, 36 e 43), executam RQN iterações. Para determinar o número de repetições do laço da linha 41, partiu-se do raciocínio de que independentemente do número de *buckets*, e de seus tamanhos ($bucketSizes[i]$), o laço executará N vezes, pois seu propósito é concatenar todos *buckets* já ordenados, recriando o arranjo (arr) de forma ordenada.

A linha 39, faz RQN chamadas à função $insertionSort$, cujo o custo varia de acordo com o tamanho de cada *bucket*. No pior cenário, os dados encontram-se em ordem decrescente, e são alocados a apenas um *bucket*, dessa forma uma das chamadas ao *Insertion Sort* terá que ordenar todos N os elementos, as demais serão invocadas sob vetores vazios e custarão apenas a chamada à função, levando, assim, ao custo descrito na linha 38 $RQN + 4N^2 + 9N - 5$, implicando em uma complexidade $O(n^2)$. Neste cenário, o polinômio de custo total, para o *Bucket Sort* é aquele descrito na linha 50, $4N^2 + 26N + 18\sqrt{N} + 22$, sendo a complexidade quadrática determinada pelo *Insertion Sort*. Caso os dados fossem igualmente distribuídos nos $\sqrt{N}$ *buckets*, cada um contendo $\sqrt{N}$ elementos, as chamadas ao *Insertion Sort* teriam custo total de $\sqrt{N} \times (4\sqrt{N}^2 + 9\sqrt{N} - 5)$ e o comportamento do *Bucket Sort*, como um todo, teria complexidade $O(n)$.

[4]De acordo com a faixa de valores de cada subconjunto.

```c
void bucketSort(int arr[], int n, long long *swaps) {
  int i, j, k, range, numberOfBuckets;
  int max = arr[0];                                        // 1a + 1i
  *swaps = 0;                                              // 1i + 1a

  for (i = 1; i < n; i++)                                  // 1a + 1t + (N-1)(1t + 1pp)
    if (arr[i] > max)                                      // (N-1)*(1i + 1t)
      max = arr[i];                                        // (N-1)*(1a + 1i)

  numberOfBuckets = ceil(sqrt(n));                         // 1a + 1c + 1c
  if (numberOfBuckets == 0)                                // 1t
    numberOfBuckets = 1;                                   // 0*1a

  range = (max + 1) / numberOfBuckets;                     // 1a + 1p + 1d
  if (range == 0)                                          // 1t
    range = 1;                                             // 1a

  // 1a + 1c + 1c + 1m
  int **buckets = (int **)malloc(sizeof(int *) * numberOfBuckets);
  // 1a + 1c + 1c
  int *bucketSizes = (int *)calloc(numberOfBuckets, sizeof(int));

  for (i = 0; i < numberOfBuckets; i++) {                  // 1a + 1t + RQN*(1t + 1pp)
    buckets[i] = (int *)malloc(sizeof(int) * (n + 1));     // RQN*(1i + 1a + 1c + 1c)
    bucketSizes[i] = 0;                                    // RQN*(1i + 1a)
  }

  for (i = 0; i < n; i++) {                                // 1a + 1t + N(1t + 1pp)
    j = arr[i] / range;                                    // N*(1a + 1i + 1d)
    if (j >= numberOfBuckets)                              // N*(1t)
      j = numberOfBuckets - 1;                             // 0*N*(1a + 1s)
    buckets[j][bucketSizes[j]++] = arr[i];                 // N*(1i + 1i + 1pp + 1a + 1i)
  }

  k = 0;                                                   // 1a
  long long localSwaps;
  for (i = 0; i < numberOfBuckets; i++) {                  // 1a + 1t + RQN*(1t + 1pp)
    // RQN*(1c) + 4N² + 9N - 5
    insertionSort(buckets[i], bucketSizes[i], &localSwaps);
    *swaps += localSwaps;                                  // RQN*(1i + 1a + 1p)
    for (j = 0; j < bucketSizes[i]; j++)                   // RQN*(1a+1t+1i) + N(1t+1pp)
      arr[k++] = buckets[i][j];                            // N*(1i + 1pp + 1a + 1i + 1i)
  }

  for (i = 0; i < numberOfBuckets; i++)                    // 1a + 1t + RQN*(1t + 1pp)
    free(buckets[i]);                                      // RQN*(1c + 1i)

  free(buckets);                                           // 1c
  free(bucketSizes);                                       // 1c
}                                                          // 4N² + 26N + 18RQN + 22
```

Código 4: Bucket Sort em C

3. Materiais e Métodos

Os algoritmos da Seção 2, foram implementados[5] na linguagem C e comparados empiricamente na ordenação de diferentes conjuntos de dados, com tamanho variando entre 100 e 2.000.000 elementos, de acordo com os 4 cenários abaixo:

- Aleatório: Os elementos do arranjo estão aleatorizados;
- Decrescente: Os elementos do arranjo estão do maior valor para menor;
- Ordenados: Os elementos do arranjo já estão ordenados de forma crescente;
- Parcialmente Ordenados: Apenas alguns elementos estão fora de ordem.

Para cada execução, registrou-se o número de trocas ($a_i \rightleftharpoons a_j$) e o tempo de execução (em ms) de cada um dos quatro métodos de ordenação implementados. O Código 5 demonstra como foi calculado o tempo de execução, utilizando-se o método **clock** da biblioteca **time.h**.

```c
#include <time.h>

clock_t start_time = clock();
// chamada ao método de ordenação
clock_t end_time   = clock();

double time_spent = (double)(end_time - start_time) * 1000.0 / CLOCKS_PER_SEC;
```

Código 5: Temporização em C

Para compilar os códigos foi utilizado o compilador GCC e o seguinte comando.

```
gcc -o EXECUTAVEL NOME_ARQUIVO.c -lm
```

Os testes foram realizados utilizando um processador AMD Ryzen 5 3500x, com 16 GB de memória RAM, rodando o sistema operacional GNU/Linux, distribuição Ubuntu 22.04.02 LTS. O código foi compilado, sem otimizações, utilizando o compilador GCC 11.4.0.

4. Resultados

A Tabela 2 sumariza os resultados obtidos nos experimentos realizados em arranjos estáticos e está subdividida em quatro conjuntos de colunas, correspondentes aos cenários: aleatório, decrescente, ordenado e semi-ordenado. Na referida tabela, são expostos os tempos de execução em milissegundos e o número de permutações realizadas para cada algoritmo, subdivididas de acordo com o tamanho do arranjo.

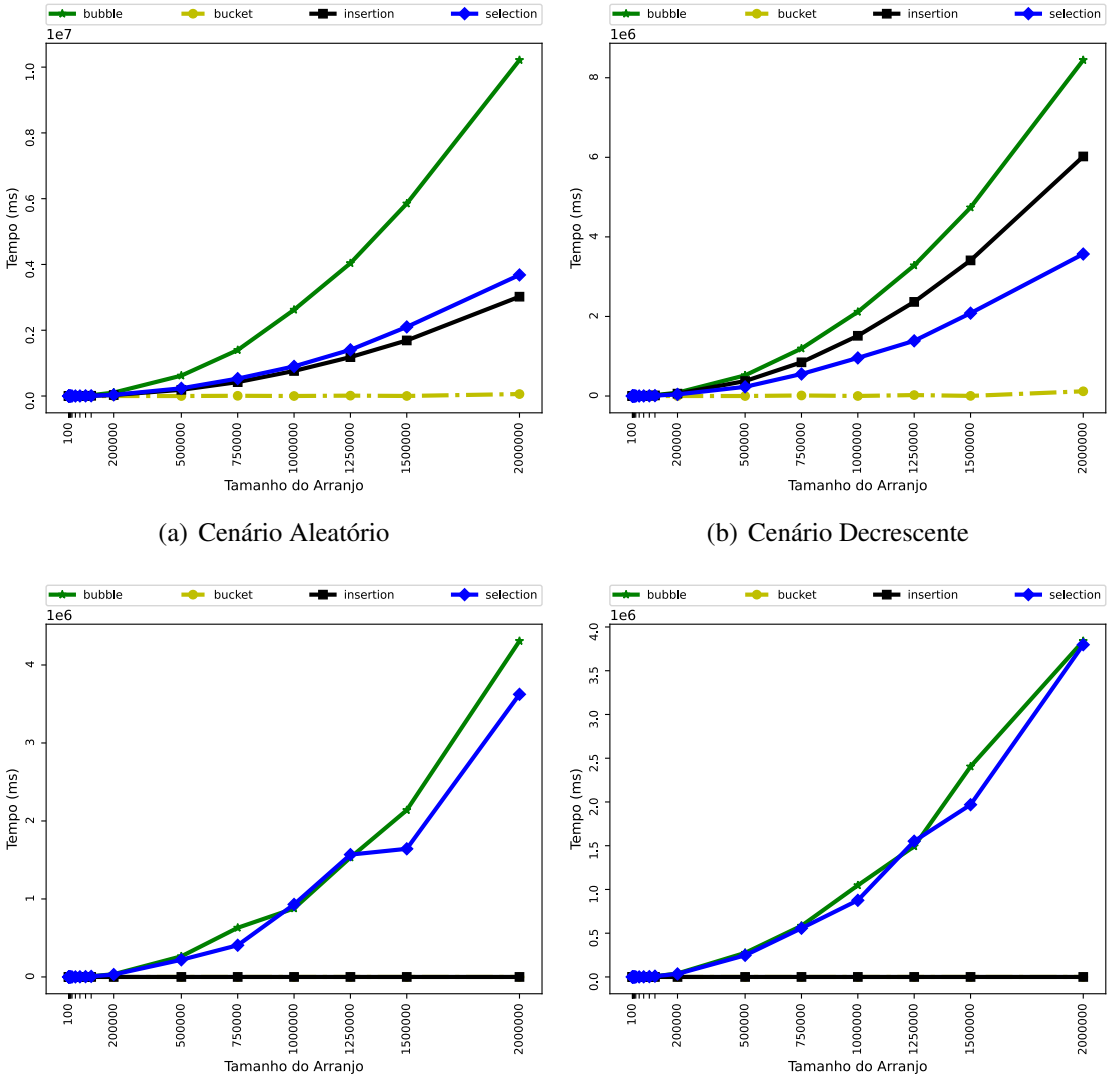
Analisando a Figura 1, constata-se que o *Bubble Sort* apresentou o desempenho menos eficiente no que concerne ao tempo de execução. A análise dos diferentes cenários

[5] O código-fonte completo, os dados de entrada e os resultados dos testes podem consultados no repositório deste trabalho em https://github.com/rsn86/edaa-aa

		Aleatório		Decrescente		Ordenado		Semi-Ordenado	
Tam.	Ordenação	Tempo (ms)	Trocas	Tempo (ms)	Trocas	Tempo (ms)	Trocas	Tempo (ms)	Trocas
100	Bubble	0,02	2.320	0,04	4.950	0,01	0	0,02	1.412
	Bucket	0,01	246	0,01	497	0,01	0	0,01	161
	Insertion	0,01	2.320	0,02	4.950	0,00	0	0,01	1.412
	Selection	0,01	96	0,01	50	0,01	0	0,01	26
200	Bubble	0,12	10.501	0,08	19.897	0,03	0	0,07	2.997
	Bucket	0,01	647	0,01	1.359	0,01	0	0,01	297
	Insertion	0,03	10.501	0,06	19.897	0,00	0	0,01	2.997
	Selection	0,04	197	0,03	102	0,03	0	0,03	25
500	Bubble	0,45	63.916	0,53	124.729	0,29	0	0,24	6.390
	Bucket	0,03	2.756	0,04	5.605	0,02	0	0,02	429
	Insertion	0,28	63.916	0,53	124.729	0,00	0	0,02	6.390
	Selection	0,20	491	0,21	261	0,23	0	0,26	28
1.000	Bubble	1,64	242.285	2,10	499.398	0,76	0	0,85	14.432
	Bucket	0,08	7.798	0,13	15.583	0,07	0	0,05	793
	Insertion	0,71	242.285	1,49	499.398	0,00	0	0,05	14.432
	Selection	0,78	991	0,76	543	0,89	0	0,88	27
2.000	Bubble	6,49	1.020.378	9,07	1.998.603	2,98	0	3,18	29.278
	Bucket	0,17	21.731	0,23	44.198	0,09	0	0,09	1.067
	Insertion	3,07	1.020.378	6,03	1.998.603	0,01	0	0,09	29.278
	Selection	4,10	1.991	3,89	1.153	2,80	0	3,56	30
5.000	Bubble	41,20	6.229.620	51,70	12.495.004	18,49	0	19,26	78.757
	Bucket	0,47	88.096	0,79	174.716	0,18	0	0,16	1.695
	Insertion	18,59	6.229.620	38,19	12.495.004	0,01	0	0,26	78.757
	Selection	17,93	4.989	20,10	3.080	22,20	0	17,16	35
7.500	Bubble	95,41	14.135.727	119,26	28.115.568	43,81	0	42,98	127.915
	Bucket	0,74	159.921	1,19	319.833	0,23	0	0,24	1.911
	Insertion	43,42	14.135.727	83,85	28.115.568	0,02	0	0,41	127.915
	Selection	41,27	7.488	53,56	4.838	39,64	0	40,09	39
10.000	Bubble	176,90	25.127.766	208,37	49.984.816	75,91	0	90,64	189.215
	Bucket	1,06	249.321	1,73	492.969	0,29	0	0,30	2.492
	Insertion	75,61	25.127.766	149,58	49.984.816	0,03	0	0,61	189.215
	Selection	72,85	9.987	87,37	6.599	70,52	0	86,90	50
15.000	Bubble	436,92	56.655.180	470,17	112.470.052	168,12	0	167,81	220.272
	Bucket	1,83	460.923	3,24	914.571	0,40	0	0,39	3.152
	Insertion	169,53	56.655.180	338,46	112.470.052	0,06	0	1,18	220.272
	Selection	160,71	14.987	248,21	10.168	158,68	0	197,86	57
30.000	Bubble	2.018,68	223.898.990	1.886,87	449.894.311	666,45	0	795,09	476.173
	Bucket	5,02	1.383.271	8,99	2.744.413	0,68	0	0,70	4.233
	Insertion	670,27	223.898.990	1.349,33	449.894.311	0,07	0	2,56	476.173
	Selection	634,04	29.985	814,27	21.197	628,38	0	622,24	106
50.000	Bubble	5.965,71	626.213.038	5.151,51	1.249.723.614	1.837,64	0	1.996,09	500.505
	Bucket	10,17	2.807.724	18,50	5.577.326	2,26	0	1,84	4.831
	Insertion	1.883,25	626.213.038	3.758,37	1.249.723.614	0,12	0	1,66	500.505
	Selection	1.757,76	49.976	2.185,04	36.245	2.207,27	0	1.712,39	161
75.000	Bubble	13.667,99	1.405.529.316	11.692,30	2.811.896.825	4.208,09	0	4.976,33	535.207
	Bucket	17,69	5.097.612	32,45	10.134.960	2,19	0	2,26	5.973
	Insertion	4.237,78	1.405.529.316	8.452,82	2.811.896.825	0,18	0	1,82	535.207
	Selection	5.417,60	74.975	5.637,65	54.624	3.941,92	0	3.849,98	270
100.000	Bubble	24.502,30	2.487.623.102	21.116,25	4.998.943.436	9.184,03	0	7.352,31	555.524
	Bucket	38,98	12.106.773	77,45	24.938.422	2,82	0	2,77	7.325
	Insertion	7.522,18	2.487.623.102	15.079,50	4.998.943.436	0,25	0	1,96	555.524
	Selection	7.333,68	99.973	10.169,83	73.653	7.013,86	0	8.660,75	287
200.000	Bubble	103.945,31	10.017.382.493	84.976,80	19.995.875.941	34.967,45	0	39.677,79	633.324
	Bucket	73,50	22.084.536	138,03	44.349.770	4,74	0	4,78	9.667
	Insertion	30.246,20	10.017.382.493	59.753,71	19.995.875.941	0,49	0	2,42	633.324
	Selection	35.358,97	199.952	40.886,56	148.452	30.862,35	0	34.687,91	497
500.000	Bubble	619.788,25	62.398.054.464	519.562,45	124.974.616.679	264.457,70	0	275.115,84	445.301
	Bucket	253,40	80.102.109	492,06	160.418.079	10,51	0	10,74	15.253
	Insertion	188.799,87	62.398.054.464	376.132,63	124.974.616.679	1,23	0	2,63	445.301
	Selection	234.755,68	499.885	230.505,37	374.968	219.714,45	0	247.054,54	1.474
750.000	Bubble	1.397.801,05	140.461.175.701	1.193.145,27	281.193.046.742	630.452,03	0	584.502,16	685.084
	Bucket	6.784,59	2.226.355.495	13.646,95	4.476.270.630	14,44	0	14,85	14.852
	Insertion	423.122,18	140.461.175.701	847.207,95	281.193.046.742	1,84	0	3,93	685.084
	Selection	529.791,24	749.853	553.308,81	563.761	406.344,05	0	556.724,12	1.750
1.000.000	Bubble	2.621.901,18	249.656.445.773	2.115.306,25	499.898.945.886	878.072,48	0	1.047.324,32	605.648
	Bucket	619,74	201.364.861	1.227,07	402.304.509	19,19	0	19,54	20.249
	Insertion	762.266,57	249.656.445.773	1.514.271,33	499.898.945.886	2,46	0	4,55	605.648
	Selection	901.002,03	999.834	957.461,79	750.863	930.081,52	0	875.865,12	2.654
1.250.000	Bubble	4.037.722,91	390.361.807.907	3.280.903,46	781.092.263.392	1.531.357,60	0	1.492.589,48	542.933
	Bucket	11.642,12	3.823.157.442	23.717,94	7.785.568.864	23,22	0	23,63	17.760
	Insertion	1.183.038,01	390.361.807.907	2.365.788,87	781.092.263.392	3,08	0	4,73	542.933
	Selection	1.405.304,34	1.249.781	1.387.109,57	937.928	1.568.541,89	0	1.551.613,82	2.967
1.500.000	Bubble	5.850.463,43	562.524.061.196	4.737.851,72	1.124.772.934.347	2.138.878,32	0	2.404.658,42	564.329
	Bucket	1.635,69	533.878.239	3.279,48	1.072.864.410	27,38	0	28,58	21.637
	Insertion	1.690.018,01	562.524.061.196	3.409.335,45	1.124.772.934.347	3,70	0	5,81	564.329
	Selection	2.100.262,86	1.499.729	2.082.929,71	1.122.889	1.642.914,73	0	1.968.687,38	3.900
2.000.000	Bubble	10.214.543,55	1.000.596.643.362	8.444.420,38	1.999.596.732.062	4.307.577,54	0	3.839.856,09	421.506
	Bucket	59.732,11	19.622.645.914	118.847,91	38.981.312.866	35,68	0	36,69	24.405
	Insertion	3.020.346,37	1.000.596.643.362	6.021.550,31	1.999.596.732.062	4,93	0	6,19	421.506
	Selection	3.681.204,26	1.999.613	3.569.111,36	1.487.419	3.624.027,24	0	3.797.275,57	5.641

Tabela 2. Desempenho dos métodos de ordenação

de teste revelou que os cenários aleatório e decrescente tiveram uma performance significativamente inferior comparada aos demais. Nos cenários ordenado e semi-ordenado, os métodos *Bubble Sort* e *Selection Sort* exibiram um desempenho similar, porém inferior aos demais algoritmos. Entre todos os métodos avaliados, o *Bucket Sort* manifestou o melhor desempenho em tempo de execução. Tal resultado pode ser atribuído à sua complexidade algorítmica inferior, a qual, em seu pior cenário, exibe um comportamento quadrático, algo que, durante os testes, não se manifestou, evidenciando empiricamente que tal cenário é atípico.



(a) Cenário Aleatório

(b) Cenário Decrescente

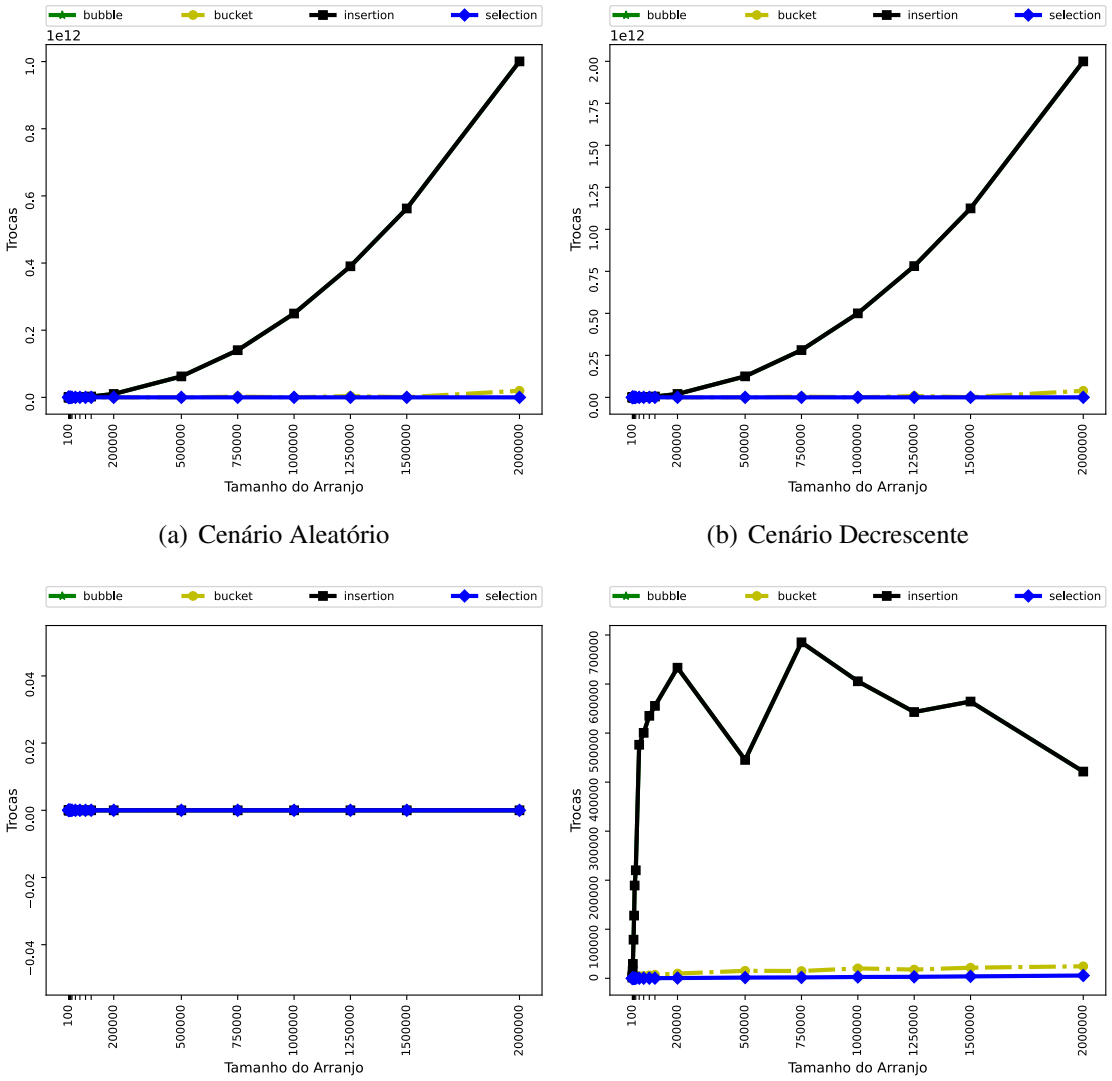
(c) Cenário Ordenado

(d) Cenário Semi-Ordenado

Figura 1. Tempo (ms) x Tamanho do Arranjo por Cenário

A Figura 2 oferece uma análise adicional acerca do número de permutações efetuadas por cada algoritmo. Através desta figura, torna-se evidente que os algoritmos *Insertion Sort* e *Bubble Sort* são aqueles que mais realizam permutações em comparação aos demais.

As Figuras 3 e 4 apresentam os resultados individuais por métodos de ordenação,



(a) Cenário Aleatório

(b) Cenário Decrescente

(c) Cenário Ordenado

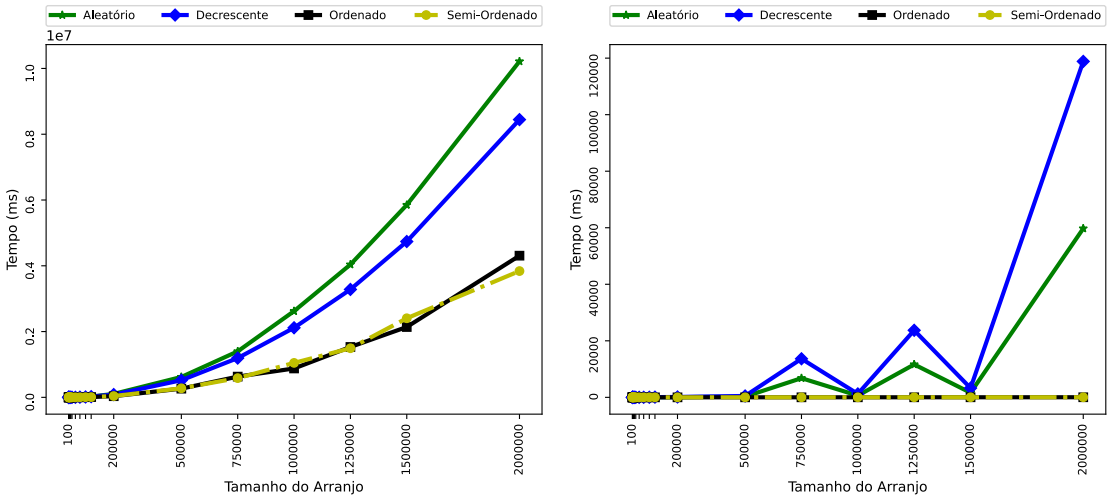
(d) Cenário Semi-Ordenado

Figura 2. Trocas x Tamanho do Arranjo por Cenário

ilustrando a curva de crescimento tanto no tempo de execução quanto no número de permutações para cada conjunto de entrada.

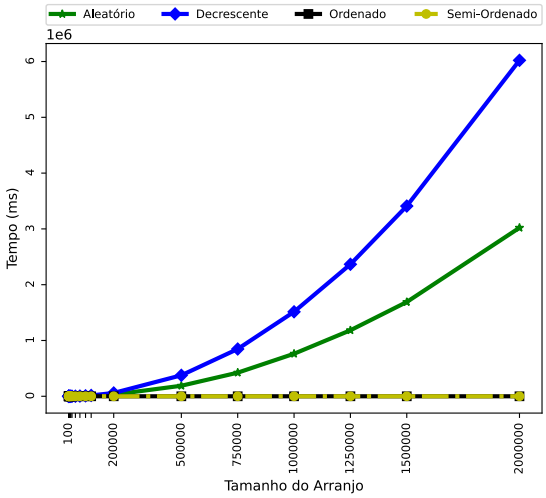
O método *Selection Sort*, quanto ao tempo de execução, conforme ilustrado na Figura 3(d), não apresentou variação significativa em relação ao conjunto de entrada, ao contrário dos demais métodos, que para os conjuntos ordenados e semi-ordenados, demonstraram tempo de execução substancialmente reduzido. Tanto o *Selection Sort* quanto o *Bubble Sort* (Figura 3(a)) exibiram comportamento quadrático em todos os cenários de teste. Ademais, é possível observar que os métodos *Insertion Sort* (Figura 3(c)) e *Bucket Sort* (Figura 3(b)) apresentaram performance notavelmente melhores nos cenários ordenados e semi-ordenados, comparando-se aos demais cenários, alcançando assim uma complexidade algorítmica linear.

O método *Bucket Sort* apresentou um comportamento distinto em relação aos demais métodos, com taxas de crescimento sublineares, nos casos aleatórios e decrescentes,

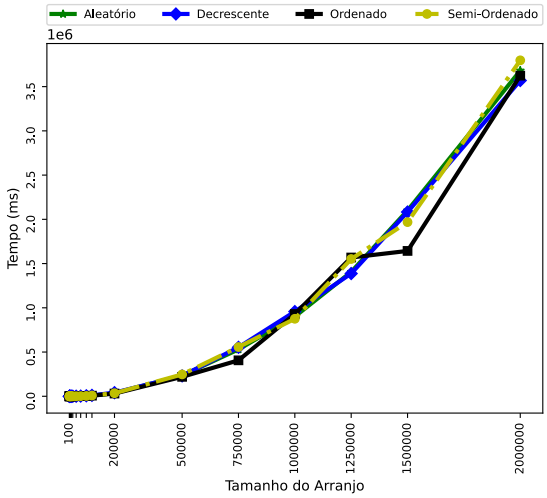


(a) Bubble Sort

(b) Bucket Sort



(c) Insertion Sort



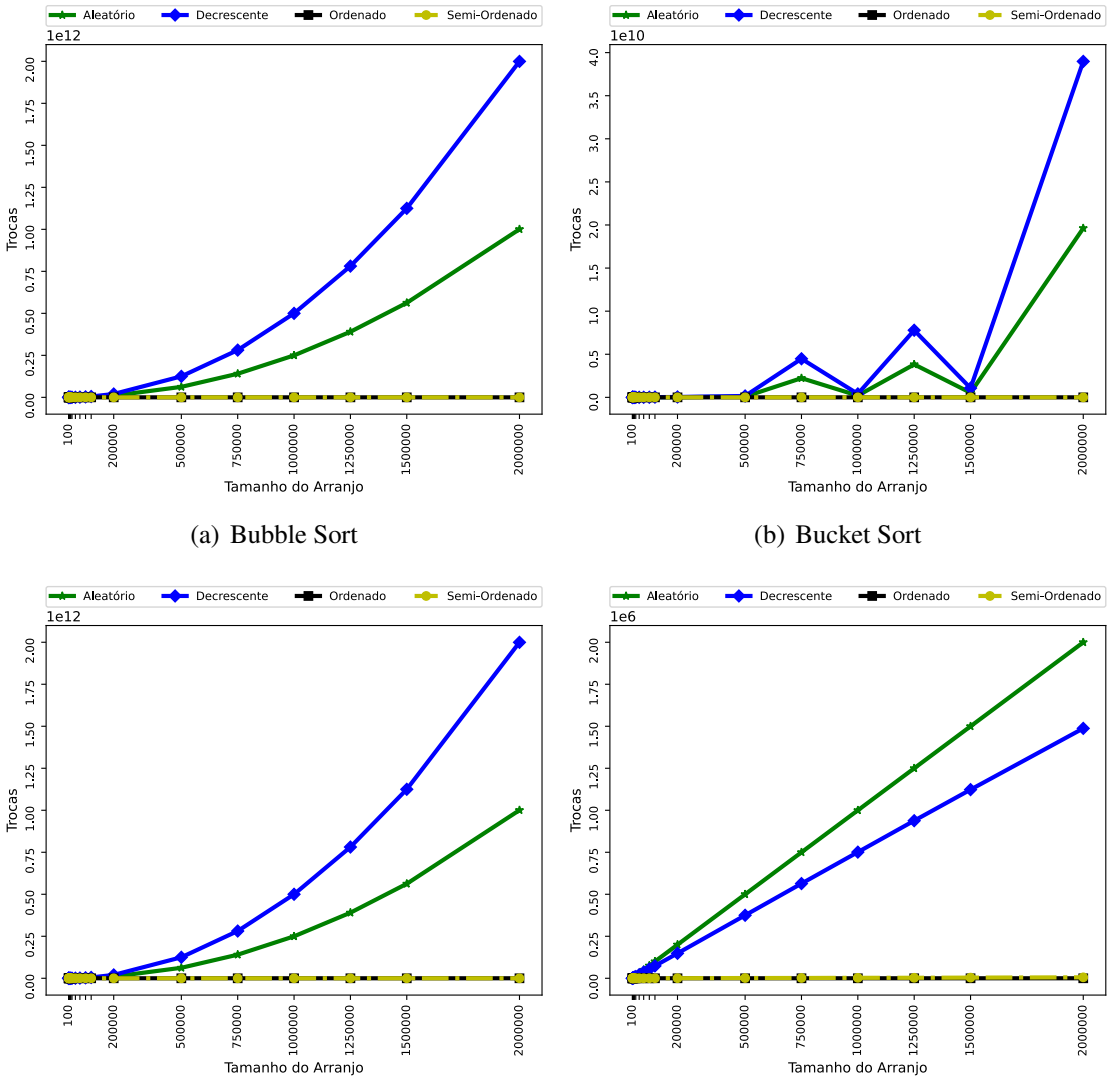
(d) Selection Sort

Figura 3. Tempo (ms) x Tamanho do Arranjo por Método

à medida que os arranjos estáticos aumentam de tamanho. Uma possível explicação para esse comportamento atípico pode estar relacionada à dimensão do *bucket*, que é proporcional à raiz quadrada do tamanho do arranjo de entrada.

Ao analisar a Figura 4, observa-se que os métodos realizam poucas ou nenhuma troca quando os dados encontram-se semi-ordenados ou ordenados. Pode-se observar também, conforme descrito na Seção 2, que o cenário decrescente constitui o pior caso para os algoritmos analisados, menos para *Selection Sort* (Figura 4(d)), cujo pior cenário seria um conjunto praticamente ordenado exceto pelo menor elemento, que estaria na última posição do arranjo, isso também pode explicar o comportamento observado na Figura 2(d). No cenário decrescente, o *Selection Sort* realiza aproximadamente $\frac{N}{2}$ trocas, pois a cada troca move tanto o menor quanto o maior elemento para suas posições corretas.

As aparentes inconsistências nos tempos de ordenação, do método *Bucket Sort*, para os cenários aleatório e decrescente, observadas na Figura 3(b), na verdade estão



(a) Bubble Sort

(b) Bucket Sort

(c) Insertion Sort

(d) Selection Sort

Figura 4. Trocas x Tamanho do Arranjo por Método

relacionadas aos dados em si, como pode-se observar pelo número de trocas efetuadas para o mesmo cenário na Figura 4(b).

5. Conclusões

A análise dos algoritmos de ordenação Bubble Sort, Selection Sort, Insertion Sort e Bucket Sort, implementados na linguagem C, mostrou que a complexidade quadrática não é o único fator a considerar ao avaliar o desempenho. As características dos dados de entrada e os detalhes de como os algoritmos são implementados podem afetar significativamente a eficiência. A compreensão dessas nuances pode ajudar engenheiros e programadores na seleção e implementação de algoritmos de ordenação, considerando não apenas a complexidade teórica, mas também os cenários de uso prático.

A desconexão observada entre o número de trocas (Figura 2) e o tempo de ordenação (Figura 1), para o método *Insertion Sort*, mais evidente no cenário semi-

ordenado, carece de investigações adicionais para determinar se esse comportamento se deve à entropia dos dados ou se está relacionado com a tese das ”atribuições”, abordada na Seção 2.3, que postula que o rearranjo dos dados, para liberação da posição correta no vetor, não se trata de trocas de fato, mas sim de operações menos custosas, denominadas ”atribuições”.