Universidade de São Paulo

Faculdade de Filosofia, Ciências e Letras de Ribeirão Preto

Introdução à Computação II - Trabalho I

Francisco Eduardo Fontenele, NºUSP 15452569
Davi Gabriel Domingues, NºUSP 15447497

Novembro de 2024

Link para o repositório: SortingAlgorithms

Sumário

1 Introdução

O objetivo deste trabalho é implementar, testar e analisar algoritmos de ordenação clássicos em diferentes cenários. Para isso, utilizamos a linguagem C++ e avaliamos sete algoritmos: Inserção Direta, Inserção Binária, Seleção, Bubblesort, Heapsort, Merge Sort e QuickSort. A análise envolve a contagem de comparações e movimentações realizadas pelos algoritmos, bem como o tempo de execução para diferentes tamanhos e tipos de vetores.

O relatório apresenta a metodologia empregada, os resultados obtidos em forma de gráficos e análises críticas para comparar a eficiência de cada algoritmo. O trabalho também discute os resultados com base em análises assintóticas e propõe recomendações práticas para o uso de cada algoritmo.

2 Metodologia

Para realizar o trabalho, seguimos os seguintes passos:

1. **Implementação:** Os algoritmos de ordenação foram implementados em arquivos de código-fonte separados, utilizando boas práticas de programação em C++.

2. **Testes Iniciais:** Cada algoritmo foi testado com o vetor fixo {45, 56, 12, 43, 95, 19, 8, 67}, fornecido pelo professor, para verificar sua correta funcionalidade. Todos os algoritmos ordenaram corretamente o vetor.

3. **Avaliação:** Para análise de desempenho, os algoritmos foram executados em três cenários diferentes:

   - Vetores em ordem crescente.
   - Vetores em ordem decrescente.
   - Vetores com valores aleatórios.

   Em cada caso, calculamos o número de comparações, movimentações e o tempo de execução.

4. **Geração de Gráficos:** Os dados coletados foram usados para criar gráficos comparativos.

5. **Análise:** Os resultados foram analisados com base nas complexidades assintóticas e características de cada algoritmo.

3 Resultados

Abaixo são apresentados os resultados obtidos para cada tipo de vetor, incluindo comparações, movimentações e tempos de execução:

Resultados para Vetores em Ordem Crescente

Os vetores em ordem crescente apresentaram resultados consistentes com as expectativas teóricas. Algoritmos como 'Insertion Sort' e 'Binary Insertion Sort' obtiveram um desempenho excelente, com poucas comparações e movimentações, uma vez que o vetor já estava quase completamente ordenado. Por outro lado, algoritmos como 'Selection Sort' e 'Bubble Sort' apresentaram maior número de operações devido à sua abordagem menos eficiente para vetores pré-ordenados. O tempo de execução do 'Merge Sort' e 'Quicksort' permaneceu constante e competitivo, destacando-se pelo seu desempenho assintótico $O(n \log n)$.

Resultados para Vetores em Ordem Decrescente

Os vetores em ordem decrescente representaram o pior cenário para a maioria dos algoritmos baseados em inserção, como o 'Insertion Sort', devido à necessidade de reordenar completamente o vetor. Neste caso, 'Merge Sort' e 'Heapsort' se destacaram, apresentando tempos de execução muito mais eficientes devido à sua robustez para lidar com grandes quantidades de desordem. O 'Quicksort', embora eficiente na média, apresentou uma leve degradação no desempenho devido à necessidade de particionar os elementos de forma menos favorável.

Resultados para Vetores Aleatórios

Os vetores aleatórios evidenciaram o equilíbrio geral entre os algoritmos. O 'Quicksort' demonstrou excelente desempenho, consolidando-se como uma das melhores opções práticas para vetores desordenados. O 'Merge Sort' também apresentou resultados consistentes e previsíveis, enquanto algoritmos mais simples, como o 'Bubble Sort', evidenciaram sua limitação em termos de eficiência quando comparados aos demais. O tempo de execução dos algoritmos foi diretamente proporcional ao número de operações realizadas, destacando as vantagens dos algoritmos $O(n \log n)$ sobre os $O(n^2)$.

Mudanças implementadas no código final

Para o trabalho desenvolvido, foram aplicados alguns princípios de otimização e de conformidade aos conceitos discutidos em sala de aula. Como mudanças e aplicações de paradigmas, em relação aos pseudocódigos e ao conteúdo apresentados em sala de aula, foi - se aplicado (para):

- InsertionSort: Presença de condicional, antes da criação do sentinela, gerando economia no total de movimentações efetivas.
- BubbleSort: Presença do comando break, caso o vetor já esteja ordenado, gerando economia no total de comparações efetivas.
- MergeSort: Aplicação do princípio de recursão para o particionamento do vetor.
- HeapSort: Tratamento da ordenação sem o auxílio de funções a parte (heap construído primeiramente, seguida da ordenação propriamente dita).
- QuickSort: Ausência das condicionais após o loop do...while.

- Tratamento da ordenação começando pelo índice 1: O primeiro conteúdo do vetor de interesse a ser tratado é o presente no índice 1. O tratamento vai até n, para um n inteiro qualquer e positivo. Tem - se presente nos algoritmos de ordenação, as lógicas devidas para o tratamento do conteúdo presente no índice 0, evitando, assim, o erro "OutOfBondsException".

4 Discussão (Item d)

Os resultados demonstram diferenças significativas no desempenho de cada algoritmo conforme o tipo de vetor utilizado. Conforme esperado: - Vetores ordenados favorecem algoritmos como 'Insertion Sort' devido à menor quantidade de movimentações e comparações. - Vetores inversamente ordenados favorecem algoritmos como 'Merge Sort' e 'Heapsort' devido à eficiência $O(n \log n)$. - Vetores aleatórios mostram o equilíbrio do 'Quicksort' para a maioria das situações práticas, evidenciando seu excelente desempenho médio.

5 Conclusão (Item e)

Com base nos resultados: - Para vetores **crescentes**, 'Insertion Sort' e 'Binary Insertion Sort' são os mais indicados devido à baixa complexidade computacional neste caso. - Para vetores **decrescentes**, 'Merge Sort' e 'Heapsort' apresentam os melhores tempos de execução, sendo as opções recomendadas. - Para vetores **aleatórios**, 'Quicksort' destaca-se como o mais eficiente na maioria dos cenários.

A Gráficos

Resultados com Vetores Ordenados Crescentemente

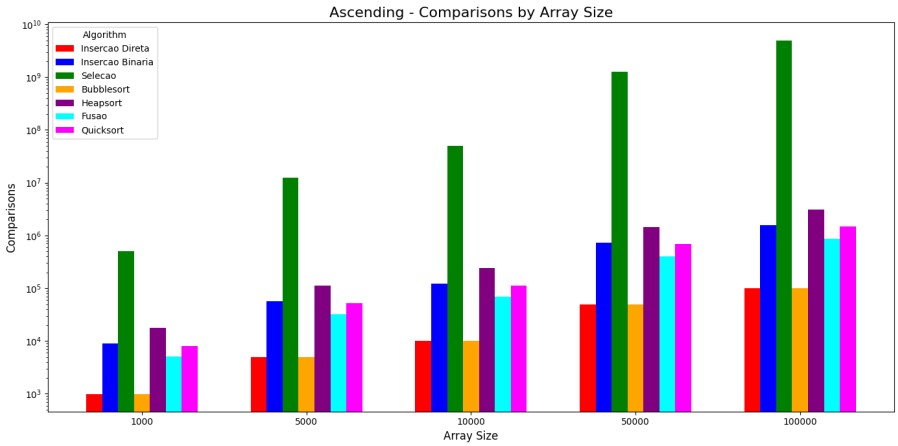

Figura 1: Número de comparações para vetores ordenados crescentemente.

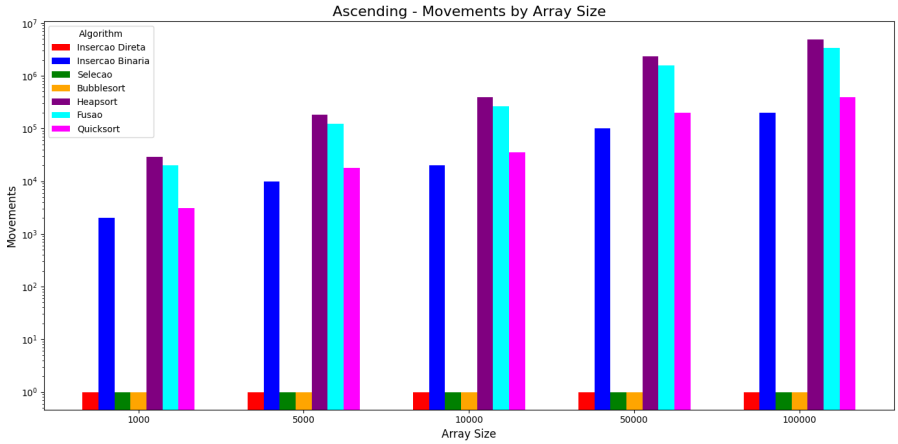

Figura 2: Número de movimentações para vetores ordenados crescentemente.

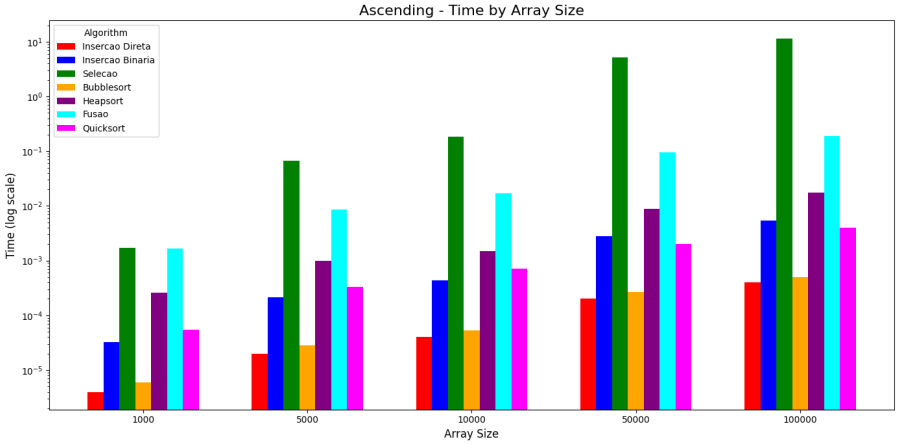

Figura 3: Tempo de execução para vetores ordenados crescentemente.

Resultados com Vetores Ordenados Decrescentemente

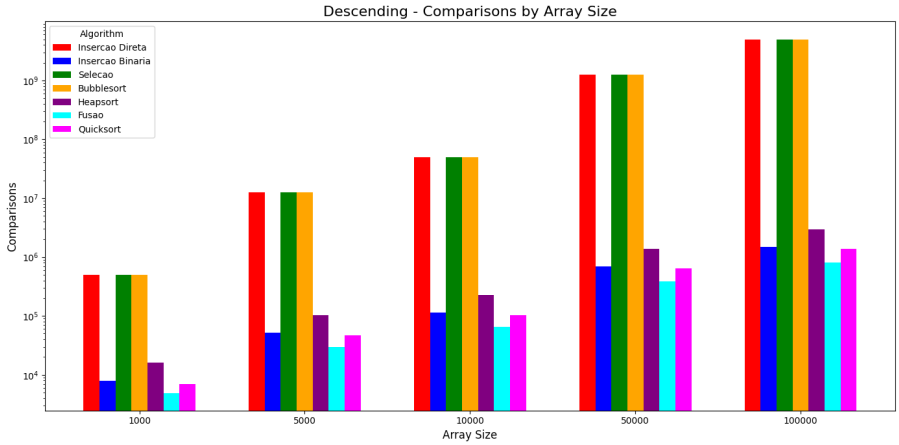

Figura 4: Número de comparações para vetores ordenados decrescentemente.

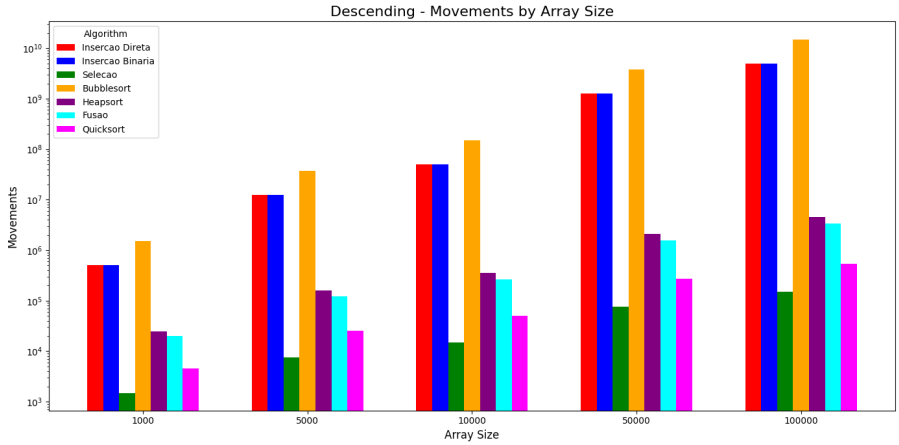

Figura 5: Número de movimentações para vetores ordenados decrescentemente.

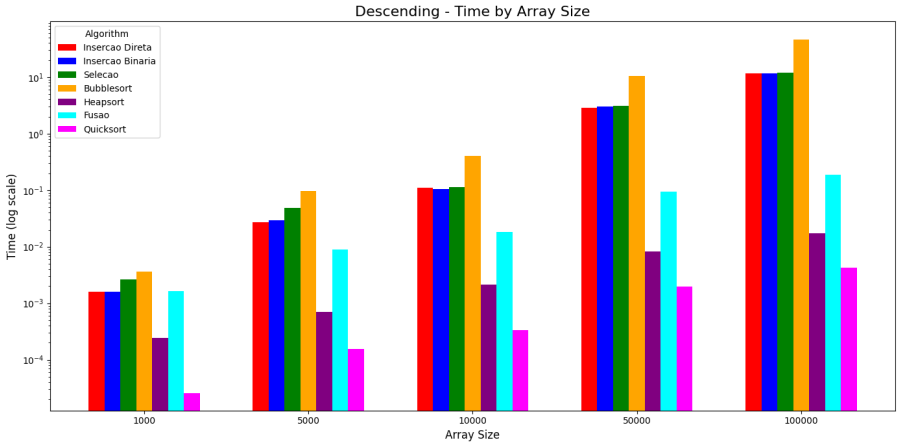

Figura 6: Tempo de execução para vetores ordenados decrescentemente.

Resultados com Vetores Aleatórios

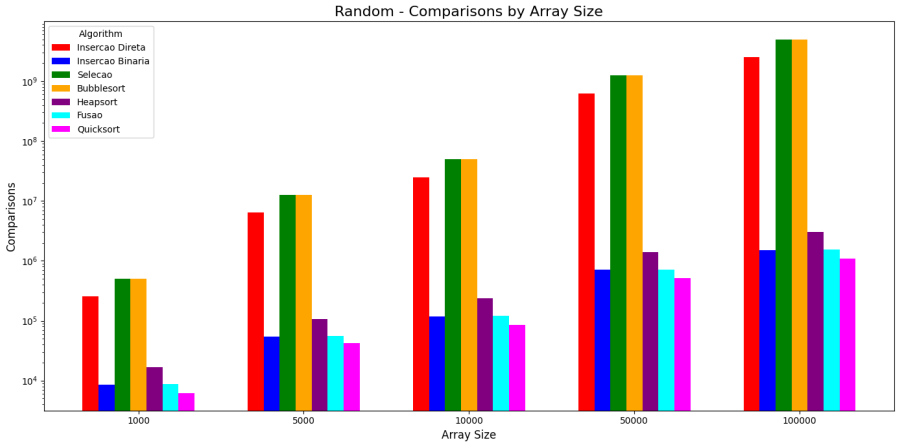

Figura 7: Número de comparações para vetores aleatórios.

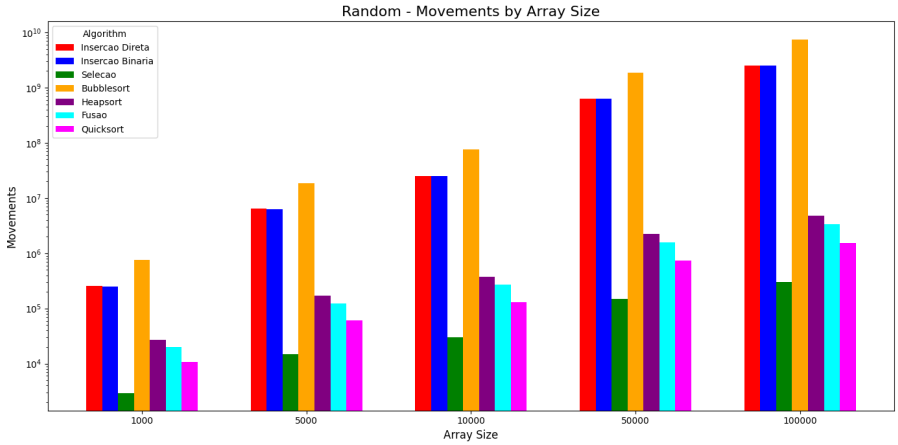

Figura 8: Número de movimentações para vetores aleatórios.

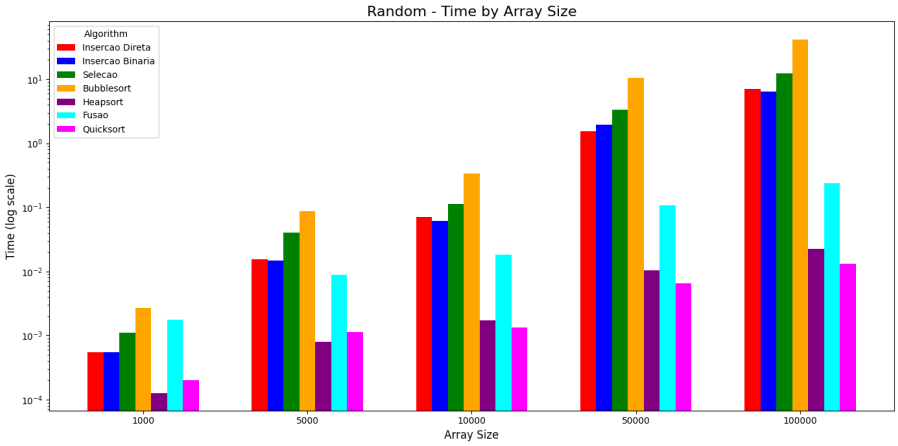

Figura 9: Tempo de execução para vetores aleatórios.

B Códigos Implementados

B.1 Arquivo main.cpp

```cpp
#include "sorting.h"
#include "utils.h"
#include <iostream>
#include <vector>
#include <fstream>
#include <chrono>
#include <filesystem>

using namespace std;

void testAlgorithm(const string& name, void (*sortFunc)(int[], int),
                   const vector<int>& sizes, const string& arrayType) {
    vector<sorting::SortStats> stats;
    cout << "  " << name << "... " << flush;

    for (int size : sizes) {
        int* arr = new int[size + 1];

        if (arrayType == "ascending")
            utils::generateAscendingArray(arr, size);
        else if (arrayType == "descending")
            utils::generateDescendingArray(arr, size);
        else
            utils::generateRandomArray(arr, size);

        sorting::resetStats();

        auto start = chrono::high_resolution_clock::now();
        if (name == "Quicksort") {
            sorting::quickSort(arr, 1, size);
        } else {
            sortFunc(arr, size);
        }
        auto end = chrono::high_resolution_clock::now();

        sorting::SortStats stat;
        stat.comparisons = sorting::getComparisons();
        stat.movements = sorting::getMovements();
        stat.timeInSeconds = chrono::duration<double>(end - start).count();

        stats.push_back(stat);
        delete[] arr;
    }

```

```
45      utils::exportResults(name, sizes, stats, arrayType, "../data/
            " + arrayType + "_resultados.csv");
46      cout << "OK\n" << flush;
47  }
48
49  int main() {
50      int choice;
51      cout << "=== Programa de Ordenacao ===\n";
52      cout << "1 - Teste com entrada manual (itens a e b)\n";
53      cout << "2 - Analise de desempenho (item c)\n";
54      cout << "Escolha uma opcao: ";
55      cin >> choice;
56
57      if (choice == 1) {
58          int n;
59          cout << "Digite o tamanho do vetor: ";
60          cin >> n;
61
62          int* arr = new int[n + 1];
63          cout << "Digite os " << n << " elementos do vetor:\n";
64
65          cin.ignore();
66
67          for (int i = 1; i <= n; i++) {
68              cout << "Elemento " << i << ": ";
69              cin >> arr[i];
70          }
71
72          cout << "\nVetor original: ";
73          utils::printArray(arr, n);
74          cout << "\nResultados das ordenacoes:\n\n";
75
76          int* temp = new int[n + 1];
77
78          cout << "Insercao Direta: ";
79          utils::copyArray(arr, temp, n);
80          sorting::insertionSort(temp, n);
81          utils::printArray(temp, n);
82
83          cout << "Insercao Binaria: ";
84          utils::copyArray(arr, temp, n);
85          sorting::binaryInsertionSort(temp, n);
86          utils::printArray(temp, n);
87
88          cout << "Selecao: ";
89          utils::copyArray(arr, temp, n);
90          sorting::selectionSort(temp, n);
91          utils::printArray(temp, n);
92
93          cout << "Bubblesort: ";
94          utils::copyArray(arr, temp, n);
```

```
95          sorting::bubbleSort(temp, n);
96          utils::printArray(temp, n);
97
98          cout << "Heapsort: ";
99          utils::copyArray(arr, temp, n);
100         sorting::heapSort(temp, n);
101         utils::printArray(temp, n);
102
103         cout << "Mergesort: ";
104         utils::copyArray(arr, temp, n);
105         sorting::mergeSort(temp, n);
106         utils::printArray(temp, n);
107
108         cout << "Quicksort: ";
109         utils::copyArray(arr, temp, n);
110         sorting::quickSort(temp, 1, n);
111         utils::printArray(temp, n);
112
113         delete[] arr;
114         delete[] temp;
115     }
116
117     else if (choice == 2) {
118         filesystem::create_directory("../data");
119         vector<string> fileNames = {
120             "../data/ascending_resultados.csv",
121             "../data/descending_resultados.csv",
122             "../data/random_resultados.csv"
123         };
124
125         for (const auto& fileName : fileNames) {
126             ofstream clearFile(fileName, ios::trunc);
127             clearFile.close();
128         }
129
130         vector<int> sizes = {1000, 5000, 10000, 50000, 100000};
131         //vector<int> sizes = {10, 20, 30, 40, 50};
132         vector<string> arrayTypes = {"ascending", "descending", "
                random"};
133
134         for (const auto& type : arrayTypes) {
135             cout << "\nTestando arrays " << type << ":\n";
136             testAlgorithm("Insercao Direta", sorting::
                insertionSort, sizes, type);
137             testAlgorithm("Insercao Binaria", sorting::
                binaryInsertionSort, sizes, type);
138             testAlgorithm("Selecao", sorting::selectionSort,
                sizes, type);
139             testAlgorithm("Bubblesort", sorting::bubbleSort,
                sizes, type);
```

```
140             testAlgorithm("Heapsort", sorting::heapSort, sizes,
                    type);
141             testAlgorithm("Fusao", sorting::mergeSort, sizes,
                    type);
142             testAlgorithm("Quicksort", nullptr, sizes, type);
143         }
144
145         cout << "\nTodos os testes foram concluidos!\n";
146         cout << "Os resultados foram exportados para os arquivos
                'ascending_resultados.csv', 'descending_resultados.csv'
                e 'random_resultados.csv'.\n";
147     }
148
149     return 0;
150 }
```

B.2 Arquivo sorting.cpp

```
1  #include "sorting.h"
2  #include <algorithm>
3
4  namespace sorting {
5      long long comparisons = 0;
6      long long movements = 0;
7
8      long long getComparisons() { return comparisons; }
9      long long getMovements() { return movements; }
10     void resetStats() { comparisons = 0; movements = 0; }
11
12     void insertionSort(int arr[], int size) {
13         for (int i = 2; i <= size; i++) {
14             comparisons++;
15             if (arr[i] < arr[i-1]) {
16                 int key = arr[i];
17                 int j = i - 1;
18                 movements++;
19
20                 while (j >= 1 && arr[j] > key) {
21                     comparisons++;
22                     arr[j + 1] = arr[j];
23                     movements++;
24                     j--;
25                 }
26                 arr[j + 1] = key;
27                 movements++;
28             }
29         }
30     }
31
32     void binaryInsertionSort(int arr[], int size) {
33         for (int i = 2; i <= size; i++) {
34             int key = arr[i];
```

```
35              int left = 1;
36              int right = i - 1;
37              movements++;
38
39              while (left <= right) {
40                  int mid = (left + right) / 2;
41                  comparisons++;
42                  if (arr[mid] > key)
43                      right = mid - 1;
44                  else
45                      left = mid + 1;
46              }
47
48              for (int j = i - 1; j >= left; j--) {
49                  arr[j + 1] = arr[j];
50                  movements++;
51              }
52              arr[left] = key;
53              movements++;
54          }
55      }
56
57      void selectionSort(int arr[], int size) {
58          for (int i = 1; i < size; i++) {
59              int min_idx = i;
60
61              for (int j = i + 1; j <= size; j++) {
62                  comparisons++;
63                  if (arr[j] < arr[min_idx]) {
64                      min_idx = j;
65                  }
66              }
67
68              if (min_idx != i) {
69                  int temp = arr[i];
70                  arr[i] = arr[min_idx];
71                  arr[min_idx] = temp;
72                  movements += 3;
73              }
74          }
75      }
76
77      void bubbleSort(int arr[], int size) {
78          for (int i = 1; i <= size-1; i++) {
79              bool swapped = false;
80              for (int j = 1; j <= size-i; j++) {
81                  comparisons++;
82                  if (arr[j] > arr[j + 1]) {
83                      std::swap(arr[j], arr[j + 1]);
84                      movements += 3;
85                      swapped = true;
```

```
86                  }
87              }
88              if (!swapped) break;
89          }
90      }
91
92      void heapSort(int arr[], int size) {
93          // Construir heap
94          for (int i = size/2; i >= 1; i--) {
95              int root = i;
96              while (2*root <= size) {
97                  int child = 2*root;
98                  if (child + 1 <= size) {
99                      comparisons++;
100                     if (arr[child + 1] > arr[child])
101                         child++;
102                 }
103                 comparisons++;
104                 if (arr[root] < arr[child]) {
105                     std::swap(arr[root], arr[child]);
106                     movements += 3;
107                     root = child;
108                 } else break;
109             }
110         }
111
112         for (int i = size; i > 1; i--) {
113             std::swap(arr[1], arr[i]);
114             movements += 3;
115             int root = 1;
116             while (2*root < i) {
117                 int child = 2*root;
118                 if (child + 1 < i) {
119                     comparisons++;
120                     if (arr[child + 1] > arr[child])
121                         child++;
122                 }
123                 comparisons++;
124                 if (arr[root] < arr[child]) {
125                     std::swap(arr[root], arr[child]);
126                     movements += 3;
127                     root = child;
128                 } else break;
129             }
130         }
131     }
132
133     void merge(int arr[], int left, int mid, int right) {
134         int* temp = new int[right - left + 2];
135         int i = left, j = mid + 1, k = 0;
136
```

```
137         while (i <= mid && j <= right) {
138             comparisons++;
139             if (arr[i] <= arr[j]) {
140                 temp[k++] = arr[i++];
141             } else {
142                 temp[k++] = arr[j++];
143             }
144             movements++;
145         }
146 
147         while (i <= mid) {
148             temp[k++] = arr[i++];
149             movements++;
150         }
151         while (j <= right) {
152             temp[k++] = arr[j++];
153             movements++;
154         }
155 
156         for (i = 0; i < k; i++) {
157             arr[left + i] = temp[i];
158             movements++;
159         }
160 
161         delete[] temp;
162     }
163 
164     void mergeSortHelper(int arr[], int left, int right) {
165         if (left < right) {
166             int mid = (left + right) / 2;
167             mergeSortHelper(arr, left, mid);
168             mergeSortHelper(arr, mid + 1, right);
169             merge(arr, left, mid, right);
170         }
171     }
172 
173     void mergeSort(int arr[], int size) {
174         mergeSortHelper(arr, 1, size);
175     }
176 
177     void quickSort(int arr[], int left, int right) {
178         if (left >= right) return;
179         int mid = (left + right) / 2;
180         int pivot = std::max(std::min(arr[left], arr[mid]), std::
               min(std::max(arr[left], arr[mid]), arr[right]));
181         movements += 3;
182 
183         int i = left, j = right;
184         while (i <= j) {
185             while (arr[i] < pivot) {
186                 i++;
```

```
187                 comparisons++;
188             }
189             while (arr[j] > pivot) {
190                 j--;
191                 comparisons++;
192             }
193             if (i <= j) {
194                 std::swap(arr[i], arr[j]);
195                 movements += 3;
196                 i++;
197                 j--;
198             }
199         }
200 
201         quickSort(arr, left, j);
202         quickSort(arr, i, right);
203     }
204 }
```

B.3 Arquivo sorting.h

```
1  #ifndef SORTING_H
2  #define SORTING_H
3  
4  namespace sorting {
5      struct SortStats {
6          long long comparisons;
7          long long movements;
8          double timeInSeconds;
9      };
10 
11     // Declaração das variáveis globais de comparações e
       movimentações
12     extern long long comparisons;
13     extern long long movements;
14 
15     // Funções para acessar as estatísticas
16     long long getComparisons();
17     long long getMovements();
18     void resetStats();
19 
20     // Funções de ordenação
21     void insertionSort(int arr[], int size);
22     void binaryInsertionSort(int arr[], int size);
23     void selectionSort(int arr[], int size);
24     void bubbleSort(int arr[], int size);
25     void heapSort(int arr[], int size);
26     void mergeSort(int arr[], int size);
27     void quickSort(int arr[], int left, int right);
28 }
29 
30 #endif
```

B.4 Arquivo utils.cpp

```cpp
#include "utils.h"
#include <iostream>
#include <chrono>
#include <random>
#include <fstream>
#include <iomanip>

using namespace std;

namespace utils {
    void printArray(const int arr[], int size) {
        cout << "[ ";
        for (int i = 1; i <= size; i++) {
            cout << arr[i] << " ";
        }
        cout << "]" << endl;
    }

    void generateRandomArray(int arr[], int size, int max_value)
       {
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<> dis(1, max_value);

        for (int i = 1; i <= size; i++) {
            arr[i] = dis(gen);
        }
    }

    void generateAscendingArray(int arr[], int size) {
        for (int i = 1; i <= size; i++) {
            arr[i] = i;
        }
    }

    void generateDescendingArray(int arr[], int size) {
        for (int i = 1; i <= size; i++) {
            arr[i] = size - i + 1;
        }
    }

    void copyArray(const int source[], int dest[], int size) {
        for (int i = 1; i <= size; i++) {
            dest[i] = source[i];
        }
    }

    sorting::SortStats testSort(void (*sortFunc)(int[], int), int
        arr[], int size) {
```

```
48          sorting::SortStats stats = {0, 0, 0.0};
49          sorting::resetStats();
50
51          int* arrCopy = new int[size + 1];
52          copyArray(arr, arrCopy, size);
53
54          try {
55              auto start = chrono::high_resolution_clock::now();
56              sortFunc(arrCopy, size);
57              auto end = chrono::high_resolution_clock::now();
58
59              stats.timeInSeconds = chrono::duration<double>(end -
                    start).count();
60              stats.comparisons = sorting::getComparisons();
61              stats.movements = sorting::getMovements();
62          }
63          catch (...) {
64              delete[] arrCopy;
65              throw;
66          }
67
68          delete[] arrCopy;
69          return stats;
70      }
71
72      sorting::SortStats testQuickSort(int arr[], int size) {
73          sorting::SortStats stats = {0, 0, 0.0};
74
75          sorting::resetStats(); // Reseta os contadores antes de
                começar
76
77          auto start = chrono::high_resolution_clock::now();
78          sorting::quickSort(arr, 1, size);
79          auto end = chrono::high_resolution_clock::now();
80
81          chrono::duration<double> diff = end - start;
82          stats.timeInSeconds = diff.count();
83
84          stats.comparisons = sorting::getComparisons();
85          stats.movements = sorting::getMovements();
86
87          return stats;
88      }
89
90      void exportResults(const std::string& algorithm_name,
91                         const std::vector<int>& sizes,
92                         const std::vector<sorting::SortStats>&
                              stats,
93                         const std::string& array_type,
94                         const std::string& filename) {
95          static bool headers_written = false;
```

```
96          std::ofstream file(filename, std::ios::app);
97
98          if (!headers_written) {
99              file << "Algorithm,ArrayType,Size,Comparisons,
                    Movements,Time\n";
100             headers_written = true;
101         }
102
103         for (size_t i = 0; i < sizes.size(); i++) {
104             file << algorithm_name << ","
105                  << array_type << ","
106                  << sizes[i] << ","
107                  << std::fixed << stats[i].comparisons << ","
108                  << std::fixed << stats[i].movements << ","
109                  << std::fixed << std::setprecision(6) << stats[i
                        ].timeInSeconds
110                  << "\n";
111         }
112
113         file.close();
114     }
115 }
```

B.5 Arquivo utils.h

```
1  #ifndef UTILS_H
2  #define UTILS_H
3
4  #include "sorting.h"
5  #include <vector>
6  #include <string>
7
8  namespace utils {
9      void generateRandomArray(int arr[], int size, int max_value =
           1000000);
10     void generateAscendingArray(int arr[], int size);
11     void generateDescendingArray(int arr[], int size);
12     void copyArray(const int source[], int dest[], int size);
13     void printArray(const int arr[], int size);
14
15     sorting::SortStats testSort(void (*sortFunc)(int[], int), int
           arr[], int size);
16     sorting::SortStats testQuickSort(int arr[], int size);
17
18     void exportResults(const std::string& algorithm_name,
19                        const std::vector<int>& sizes,
20                        const std::vector<sorting::SortStats>&
                              stats,
21                        const std::string& array_type,
22                        const std::string& filename);
23 }
24
```

```
25 #endif
```