Otavio Baroni De Moura
Rafael Matias Da Silva
TSI - 2º Semestre
ESTRUTURA DE DADOS

INSTITUTO FEDERAL
São Paulo
Campus Araraquara

RELATÓRIO TÉCNICO

Exercício Avaliativo 1: Métodos de classificação

O presente estudo apresenta a análise de algoritmos de ordenação, que direcionam para a ordenação, ou reordenação, de valores apresentados em uma dada sequência, para que os dados possam ser acessados posteriormente de forma mais eficiente. Neste relatório serão apresentados, de forma breve, os algoritmos de ordenação que fazem parte deste estudo seus respectivos códigos em Linguagem C, assim como os dados coletados nos testes, bem como uma breve análise e, por fim, serão apresentadas as conclusões acerca da análise dos dados. Como referência para desenvolver e avaliar os algoritmos de ordenação, utilizamos o artigo "Algoritmos de Ordenação: Um estudo comparativo", que orientou nossa análise de desempenho em diferentes cenários (melhor, médio e pior caso). Seguindo essa abordagem, estruturamos o código para uma exibição única e final dos resultados, visando clareza e precisão.

Em uma visão geral o código realiza testes de desempenho em algoritmos de ordenação com um array de tamanho definido pela constante “SIZE”, permitindo avaliar cada algoritmo em três cenários: pior caso (ordem decrescente), melhor caso (ordem crescente) e caso médio (aleatório). A função copiar_array é usada para replicar o array inicial para cada algoritmo, garantindo que todos operem sobre os mesmos valores e viabilizando a comparação direta de desempenho. Para alternar entre os cenários de teste, basta comentar ou descomentar as funções de geração de arrays (gerar_decrescente, gerar_crescente e gerar_aleatorio). A estrutura está organizada para facilitar a alteração entre os tipos de entrada, mantendo sempre os mesmos dados de entrada para consistência nos testes.

O arquivo métodos.c é responsável pelas implementações dos métodos de ordenação e pelo registro das métricas (tempo, número de trocas e comparações), essenciais para análise detalhada do desempenho de cada método de acordo com o cenário. Vale ressaltar que a linha (#define TAM neste arquivo, define o tamanho dos arrays, assegurando uma única exibição de resultados ao final dos métodos de ordenação, evitando saídas repetitivas e proporcionando uma análise direta do desempenho final. Ainda assim, algumas questões utilizadas necessitaram de estudos mais específicos, e pesquisas para que o código conseguisse atingir o melhor desempenho.

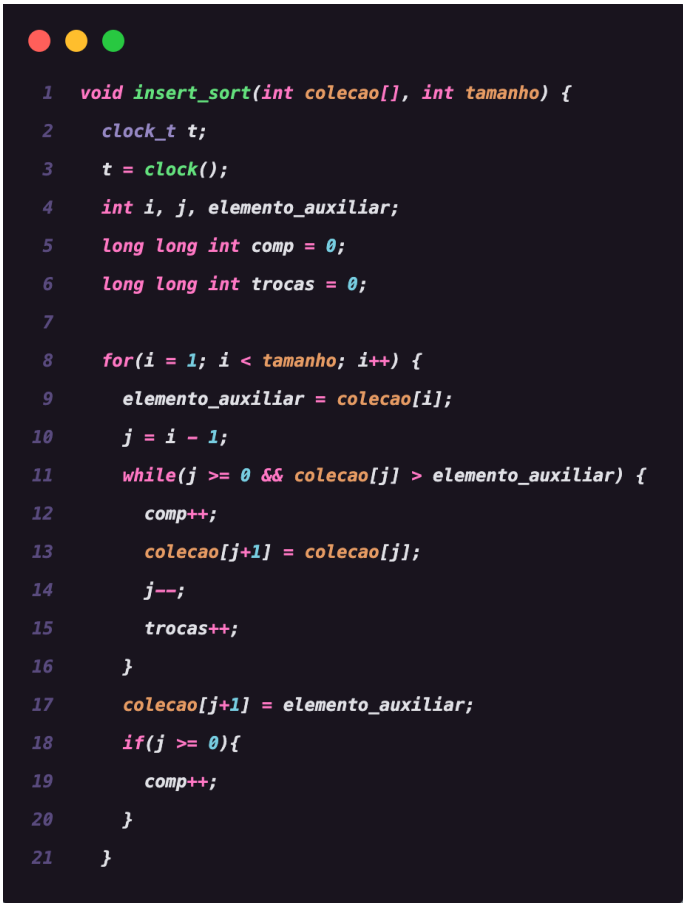
Bubble Sort:
Este algoritmo é um dos classificadores mais simples, ao qual o algoritmo percorre a lista de chaves sequencialmente várias vezes. Cada passagem consistem em comparar cada elemento na lista com seu sucessor e
trocar os dois elementos.

Figura 1 - Função Bubble Sort em C

```c
void bubble_sort(int colecao[], int tamanho){
  clock_t t;
    t = clock();

  int i, j, elemento_auxiliar;
  int trocou = true;
  long long int comp = 0;
  long long int trocas = 0;

  for(i=0; i < tamanho && trocou; i++){
    trocou = false;
    for(j=0; j < tamanho-(1+i); j++){
      comp++;
      if(colecao[j] > colecao[j+1]){
        elemento_auxiliar = colecao[j];
        colecao[j] = colecao[j+1];
        colecao[j+1] = elemento_auxiliar;
        trocou = true;
        trocas++;
      }
    }
  }
```

Insertion Sort: O Insertion Sort é um algoritmo de ordenação que constrói a lista final de forma ordenada, inserindo cada elemento de uma sequência, um a um, na posição correta. A cada iteração, o algoritmo remove um elemento da lista de entrada e o insere na posição correta da lista ordenada. Dessa forma, ele percorre a lista n vezes e, a cada iteração, insere o próximo elemento na posição correta, comparando-o com os elementos já ordenados até encontrar o lugar adequado.

Figura 2 - Função Insertion Sort em C

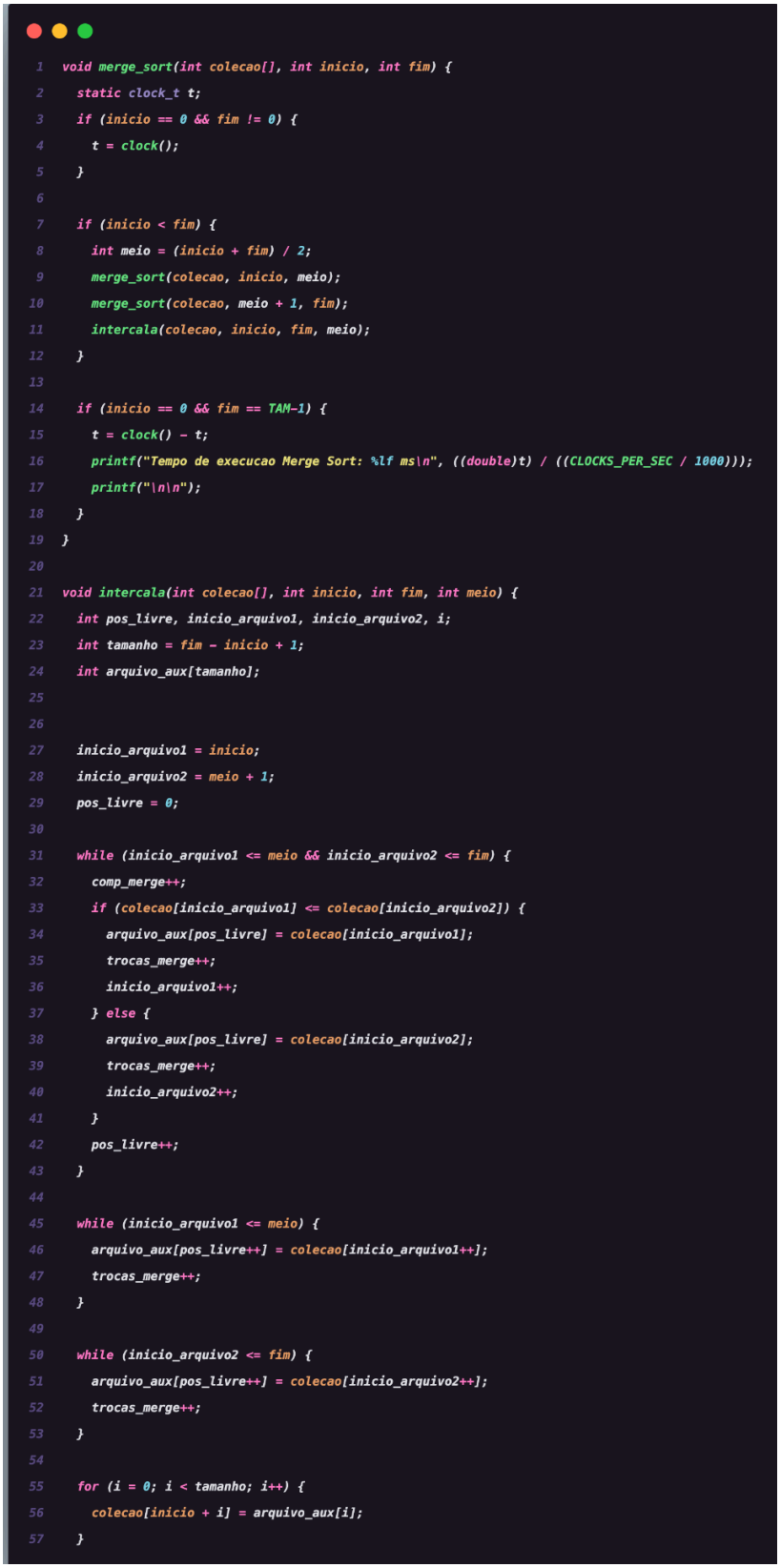


```c
void insert_sort(int colecao[], int tamanho) {
  clock_t t;
  t = clock();
  int i, j, elemento_auxiliar;
  long long int comp = 0;
  long long int trocas = 0;

  for(i = 1; i < tamanho; i++) {
    elemento_auxiliar = colecao[i];
    j = i - 1;
    while(j >= 0 && colecao[j] > elemento_auxiliar) {
      comp++;
      colecao[j+1] = colecao[j];
      j--;
      trocas++;
    }
    colecao[j+1] = elemento_auxiliar;
    if(j >= 0){
      comp++;
    }
  }
```

Merge Sort:

O Merge Sort é um algoritmo que usa a técnica de divisão e conquista para dividir a lista em duas metades e ordená-las separadamente, antes de juntá-las em uma sequência final ordenada. A lista é dividida recursivamente até que cada sub-lista tenha um elemento. Em seguida, os elementos são combinados em pares, ordenando-se conforme são mesclados, até obter uma lista final ordenada.

Figura 3 - Função Merge Sort em C

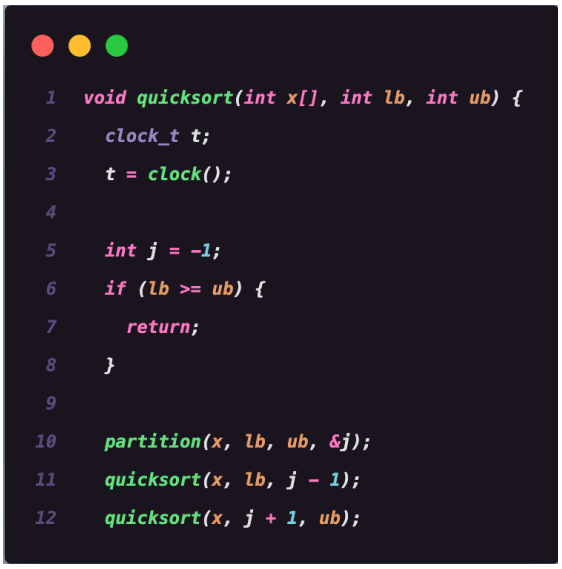


```c
void merge_sort(int colecao[], int inicio, int fim) {
  static clock_t t;
  if (inicio == 0 && fim != 0) {
    t = clock();
  }

  if (inicio < fim) {
    int meio = (inicio + fim) / 2;
    merge_sort(colecao, inicio, meio);
    merge_sort(colecao, meio + 1, fim);
    intercala(colecao, inicio, fim, meio);
  }

  if (inicio == 0 && fim == TAM-1) {
    t = clock() - t;
    printf("Tempo de execucao Merge Sort: %lf ms\n", ((double)t) / ((CLOCKS_PER_SEC / 1000)));
    printf("\n\n");
  }
}

void intercala(int colecao[], int inicio, int fim, int meio) {
  int pos_livre, inicio_arquivo1, inicio_arquivo2, i;
  int tamanho = fim - inicio + 1;
  int arquivo_aux[tamanho];


  inicio_arquivo1 = inicio;
  inicio_arquivo2 = meio + 1;
  pos_livre = 0;

  while (inicio_arquivo1 <= meio && inicio_arquivo2 <= fim) {
    comp_merge++;
    if (colecao[inicio_arquivo1] <= colecao[inicio_arquivo2]) {
      arquivo_aux[pos_livre] = colecao[inicio_arquivo1];
      trocas_merge++;
      inicio_arquivo1++;
    } else {
      arquivo_aux[pos_livre] = colecao[inicio_arquivo2];
      trocas_merge++;
      inicio_arquivo2++;
    }
    pos_livre++;
  }

  while (inicio_arquivo1 <= meio) {
    arquivo_aux[pos_livre++] = colecao[inicio_arquivo1++];
    trocas_merge++;
  }

  while (inicio_arquivo2 <= fim) {
    arquivo_aux[pos_livre++] = colecao[inicio_arquivo2++];
    trocas_merge++;
  }

  for (i = 0; i < tamanho; i++) {
    colecao[inicio + i] = arquivo_aux[i];
  }
```

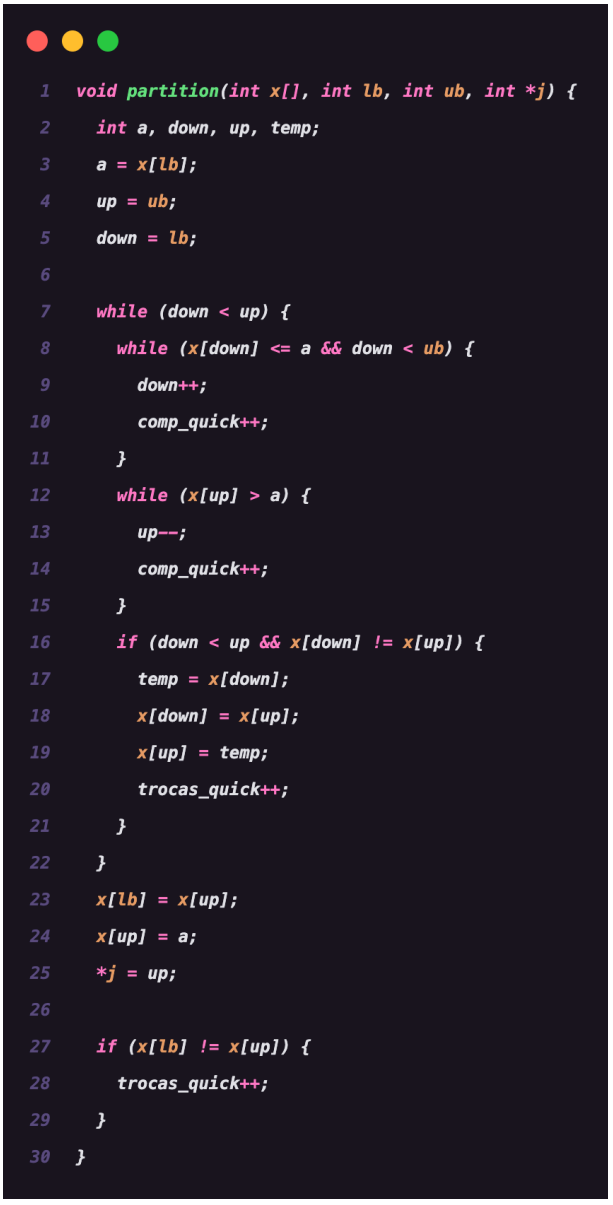
Quick Sort:

O Quick Sort é outro algoritmo baseado na técnica de divisão e conquista. Ele seleciona um elemento da lista, chamado pivô, e reordena a lista para que todos os elementos menores que o pivô fiquem à sua esquerda, enquanto os maiores fiquem à direita. O processo se repete recursivamente para as sub-listas formadas até que toda a lista esteja ordenada.

Figura 4 - Primeira Função QuickSort em C



```c
void quicksort(int x[], int lb, int ub) {
  clock_t t;
  t = clock();

  int j = -1;
  if (lb >= ub) {
    return;
  }

  partition(x, lb, ub, &j);
  quicksort(x, lb, j - 1);
  quicksort(x, j + 1, ub);
```



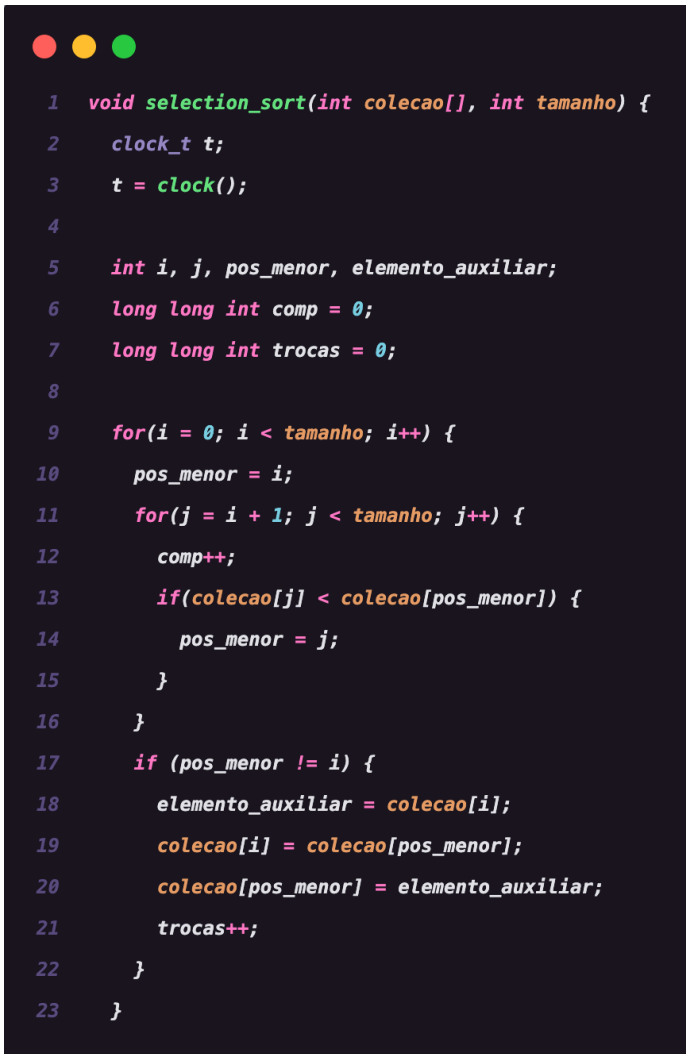
```c
void partition(int x[], int lb, int ub, int *j) {
  int a, down, up, temp;
  a = x[lb];
  up = ub;
  down = lb;

  while (down < up) {
    while (x[down] <= a && down < ub) {
      down++;
      comp_quick++;
    }
    while (x[up] > a) {
      up--;
      comp_quick++;
    }
    if (down < up && x[down] != x[up]) {
      temp = x[down];
      x[down] = x[up];
      x[up] = temp;
      trocas_quick++;
    }
  }
  x[lb] = x[up];
  x[up] = a;
  *j = up;

  if (x[lb] != x[up]) {
    trocas_quick++;
  }
}
```

Figura 5 - Segunda Função QuickSort em C

Selection Sort:

O Selection Sort é um algoritmo de ordenação que funciona encontrando o menor (ou maior) elemento da lista e o movendo para a primeira posição. Em seguida, busca-se o segundo menor elemento e move-se para a segunda posição, e assim por diante, até que a lista esteja ordenada. Esse algoritmo é simples e fácil de implementar, mas não é ideal para listas grandes devido à sua complexidade de tempo, pois envolve muitas trocas e comparações.

Figura 6 - Função Selection Sort em C



```c
void selection_sort(int colecao[], int tamanho) {
  clock_t t;
  t = clock();

  int i, j, pos_menor, elemento_auxiliar;
  long long int comp = 0;
  long long int trocas = 0;

  for(i = 0; i < tamanho; i++) {
    pos_menor = i;
    for(j = i + 1; j < tamanho; j++) {
      comp++;
      if(colecao[j] < colecao[pos_menor]) {
        pos_menor = j;
      }
    }
    if (pos_menor != i) {
      elemento_auxiliar = colecao[i];
      colecao[i] = colecao[pos_menor];
      colecao[pos_menor] = elemento_auxiliar;
      trocas++;
    }
  }
```

Análise com vetor de 100 Elementos:

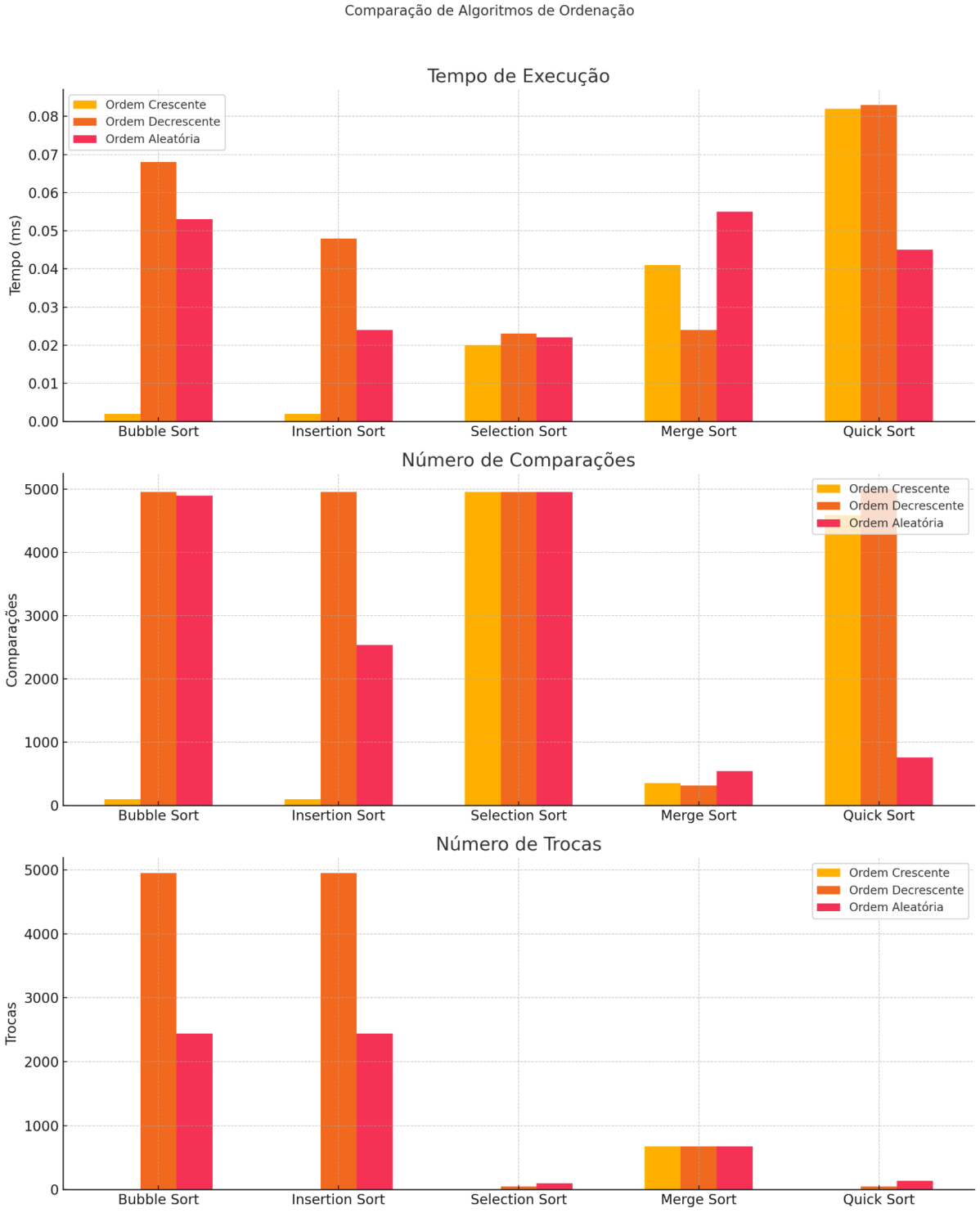
Para este estudo, utilizamos um vetor com 100 elementos em todos os testes, seguindo as especificações do exercício. Através deste conjunto limitado, foi possível observar com clareza as diferenças entre os métodos de ordenação: enquanto algoritmos mais simples, como o Bubble Sort e o Selection Sort, executaram as operações de forma sequencial e com maior número de trocas, algoritmos mais eficientes como o Merge Sort e o Quick Sort apresentaram abordagens mais complexas, mas com maior eficiência. Essa escolha de dados é adequada para entender as características e diferenças entre cada algoritmo antes de aplicá-los em contextos mais desafiadores com vetores de maior dimensão.

Análise com Vetor de 100 Elementos:

VETOR [100]

Lista	Ordem Crescente			Ordem Decrescente			Ordem Aleatória		
Algoritmo	**Tempo(ms)**	**Comp.**	**Trocas**	**Tempo(ms)**	**Comp.**	**Trocas**	**Tempo(ms)**	**Comp.**	**Trocas**
Bubble Sort	0.0020	99	0	0.0680	4950	4950	0.0530	4895	2438
Insertion Sort	0.0020	99	0	0.0480	4950	4950	0.0240	2533	2438
Selection Sort	0.0200	4950	0	0.0230	4950	50	0.0220	4950	96
Merge Sort	0.0410	356	672	0.0240	316	672	0.0550	542	672
Quick Sort	0.0820	4588	0	0.0830	4999	50	0.0450	760	135

Figura 7 - Gráfico para 100 elementos



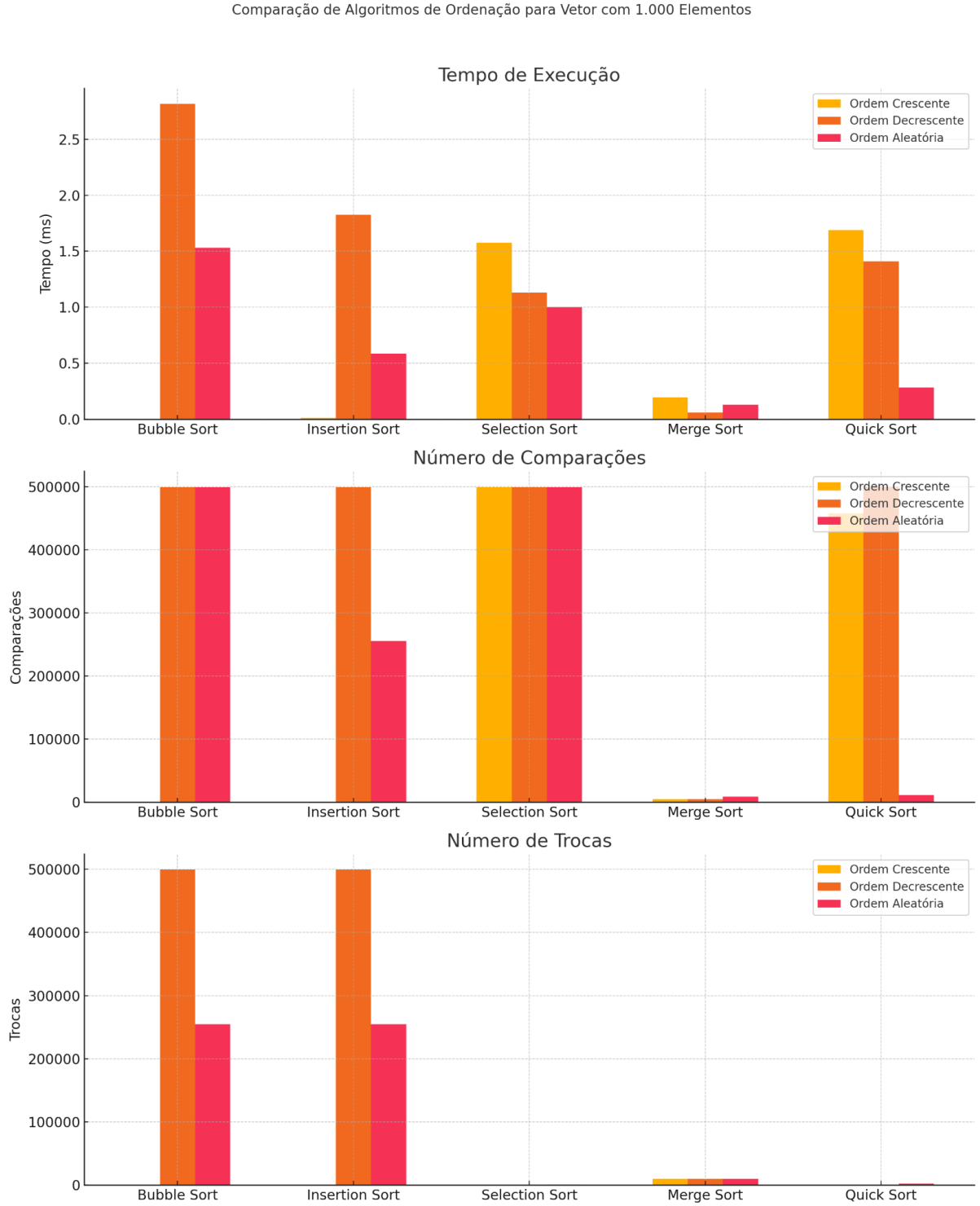
Nesse gráfico, foi possível observar que para listas pequenas e já ordenadas, os algoritmos simples funcionam muito bem, enquanto os mais complexos desempenham maior trabalho, sendo possível perceber sua eficiência para listas maiores. Seguindo a análise, os algoritmos Bubble Sort e Insertion Sort apresentam tempos de execução muito baixos em listas ordenadas de forma crescente, pois não necessitam de trocas. Em contrapartida, para listas em ordem decrescente, esses algoritmos sofrem um aumento considerável no tempo de execução devido ao grande número de comparações e trocas. O Merge Sort e o Quick Sort, por outro lado, mantêm tempos de execução consistentemente baixos, especialmente em listas aleatórias, mostrando-se mais eficazes para qualquer disposição inicial dos elementos.

Análise com Vetor de 1.000 Elementos:

VETOR [1.000]

Lista	Ordem Crescente			Ordem Decrescente			Ordem Aleatória		
Algoritmo	**Tempo(ms)**	**Comp.**	**Trocas**	**Tempo(ms)**	**Comp.**	**Trocas**	**Tempo(ms)**	**Comp.**	**Trocas**
Bubble Sort	0.005	999	0	2.816	499500	499500	1.5310	499472	254648
Insertion Sort	0.0100	999	0	1.826	499500	499500	0.5860	255639	254648
Selection Sort	1.5760	499500	0	1.129	499500	500	1.0020	499500	995
Merge Sort	0.1940	5044	9976	0.060	4932	9976	0.1300	8682	9976
Quick Sort	1.6890	457674	0	1.409	499999	500	0.2840	11496	2092

Figura 8 - Gráfico para 1000 elementos



Para o vetor de 1.000 elementos, observa-se que, em ordem crescente, algoritmos como Bubble Sort e Insertion Sort apresentaram tempos de execução baixos, refletindo a ausência de trocas. Na ordem decrescente, esses algoritmos exigiram um número elevado de comparações e trocas, o que resultou em tempos bem mais altos. Já o Merge Sort e o Quick Sort destacaram-se como os mais eficientes em listas aleatórias, exibindo tempos baixos e um número de operações consideravelmente menor.

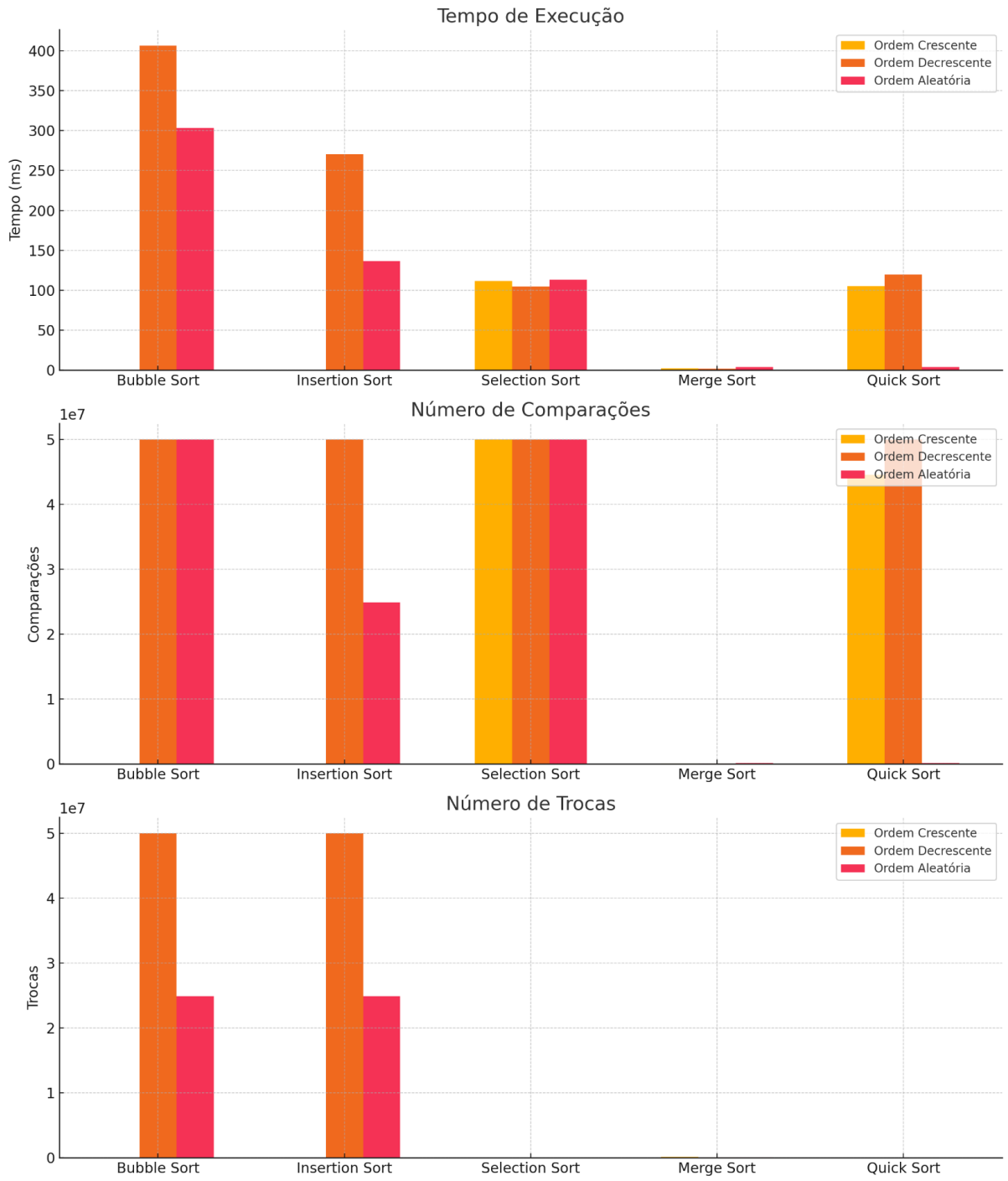
Análise com Vetor de 10.000 Elementos:

VETOR [10.000]

Lista	Ordem Crescente	Ordem Decrescente	Ordem Aleatória

Algoritmo	Tempo(ms)	Comp.	Trocas	Tempo(ms)	Comp.	Trocas	Tempo(ms)	Comp.	Trocas
Bubble Sort	0.0380	9999	0	406.525	49995000	49995000	303.364	49994535	24883340
Insertion Sort	0.1240	9999	0	270.523	49995000	49995000	136.676	24893334	24883340
Selection Sort	111.596	49995000	0	104.634	49995000	5000	112.974	49995000	9992
Merge Sort	2.4480	69008	133616	1.9060	64608	64608	3.611	120537	59071
Quick Sort	105.2110	44554107	0	119.860	49999999	5000	3.795	170064	28322

Figura 9 - Gráfico para 10000 elementos



Neste gráfico, foi analisado o vetor de 10.000 elementos, e notamos um comportamento bem semelhante. Bubble Sort e Insertion Sort mantiveram um desempenho razoável para listas em ordem crescente, mas foram substancialmente mais lentos em listas decrescentes devido ao grande número de trocas e comparações. O Merge Sort e o Quick Sort

novamente mostraram-se mais eficazes em ordem aleatória, consolidando-se como os algoritmos mais adequados para manipulação de listas maiores.

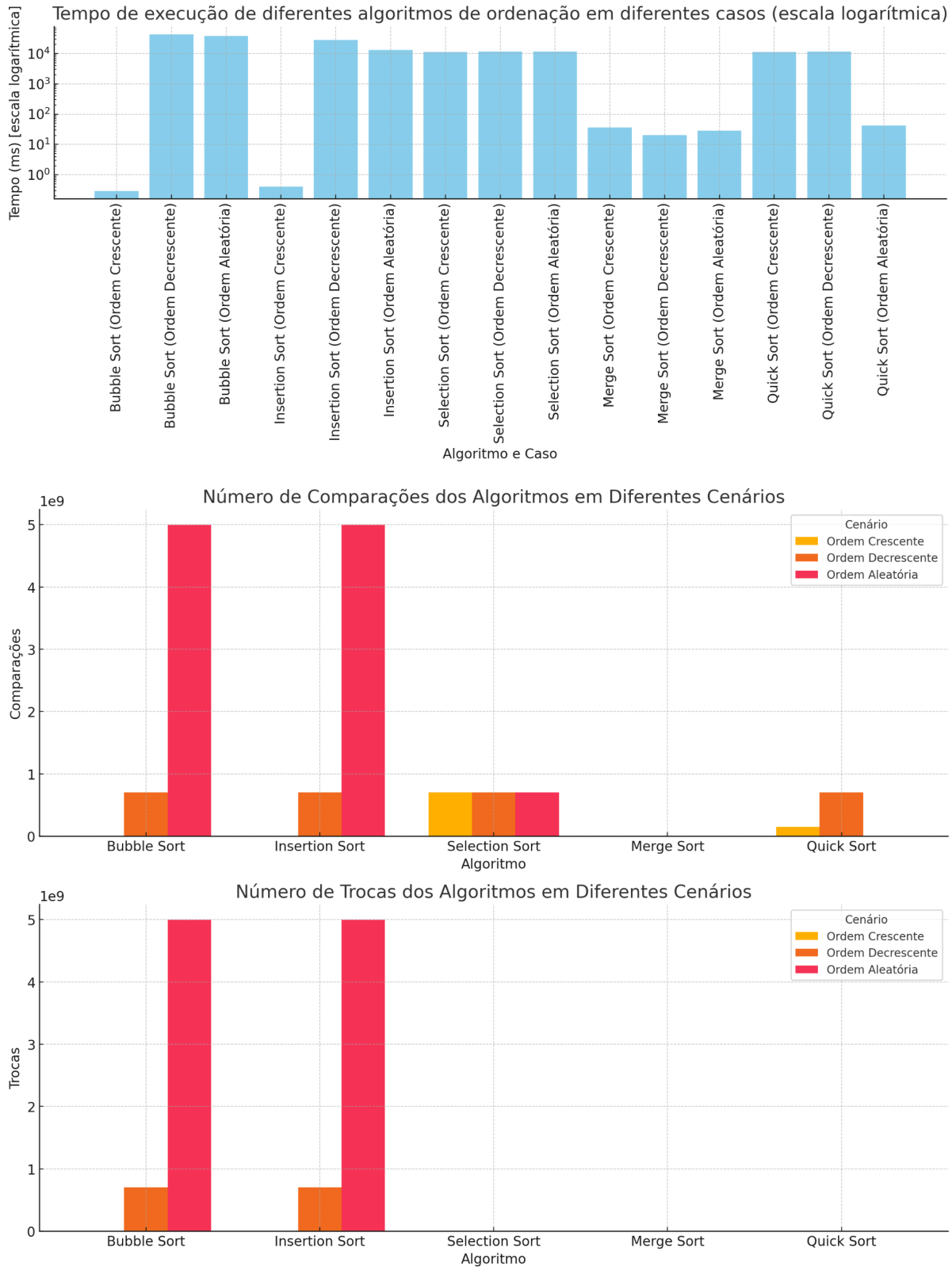
Análise com Vetor de 100.000 Elementos:

Durante o desenvolvimento de algoritmos de ordenação, encontramos um problema ao contar o número de trocas em Bubble Sort e Insertion Sort para arrays com mais de 100.000 elementos. As variáveis de contagem, declaradas como int, estavam suscetíveis a overflow, resultando em valores negativos. Para solucionar isso, estudamos e decidimos utilizar o tipo de dado long long int, que permite armazenar números muito maiores. Com essa mudança, conseguimos registrar corretamente as trocas e comparações.
Além disso, para exibir esses valores corretamente no printf, utilizamos o especificador %lld. Essa experiência nos ensinou a importância da escolha adequada de tipos de dados ao lidar com grandes volumes de dados em algoritmos, garantindo medições precisas e a integridade dos resultados.

VETOR [100.000]

Lista	Ordem Crescente			Ordem Decrescente			Ordem Aleatória		
Algoritmo	**Tempo(ms)**	**Comp.**	**Trocas**	**Tempo(ms)**	**Comp.**	**Trocas**	**Tempo(ms)**	**Comp.**	**Trocas**
Bubble Sort	0.2940	99999	0	42918,816	704982704	704982704	38206.216	4999950000	4999950000
Insertion Sort	0.4100	99999	0	27616,588	704982704	704982704	13059.116	4999950000	4999950000
Selection Sort	11119.36	704982704	0	11503,123	704982704	50000	11641.921	704982704	99983
Merge Sort	36.424	853904	69008	20,321	815024	815024	28.123	1536329	759820
Quick Sort	11178.247	149669707	0	11564,118	705032703	99999	41.986	2074073	389928

Figura 10 - Gráfico para 100.000 elementos



No gráfico acima, é possível perceber que em ordem crescente, os algoritmos de ordenação como Bubble Sort e Insertion Sort mostraram tempos de execução significativamente menores devido à ausência de necessidade de trocas. Na ordem decrescente, esses mesmos algoritmos tiveram tempos muito elevados devido ao aumento no número de comparações e trocas. O Merge Sort e Quick Sort foram os mais eficientes em ordem aleatória, com tempos mais baixos e complexidade reduzida em comparação com os outros algoritmos.

Limitações de Execução para 1 Milhão de Elementos:

Durante o desenvolvimento deste estudo, tentamos executar o programa de ordenação com uma lista de 1 milhão de elementos, no entanto, devido às limitações de processamento e memória das máquinas utilizadas, não foi possível concluir a execução para esse volume de dados. Essa limitação se tornou evidente principalmente nos algoritmos com complexidade, como o Bubble Sort e o Selection Sort, que requerem um número elevado de comparações e trocas, gerando grande carga.
Além disso, até mesmo algoritmos mais eficientes como Merge Sort e Quick Sort, com complexidade, acabam sobrecarregando as máquinas em alguns casos. Essa experiência ressalta a importância de utilizar recursos de hardware compatíveis ao processar conjuntos de dados de tamanhos enormes.
Apesar de tudo, buscamos várias possíveis soluções para que o teste se realizasse, estudando métodos para possibilitar viabilizar o teste com um milhão de elementos.
Com isso, encontramos o comando (ulimit -s unlimited) foi utilizado no ambiente de execução para remover o limite padrão da pilha de memória (stack) de um processo. Pois em sistemas Unix e Linux, o limite padrão da pilha costuma restringir a alocação de grandes arrays em memória. Quando aumentamos esse limite, garantimos que o programa possa manipular arrays de grande tamanho, como os definidos pela constante de 1.000.000, sem enfrentar um erro de "stack overflow" ou "segmentation fault".

Figura 11. Ordem Aleatória (Caso médio)

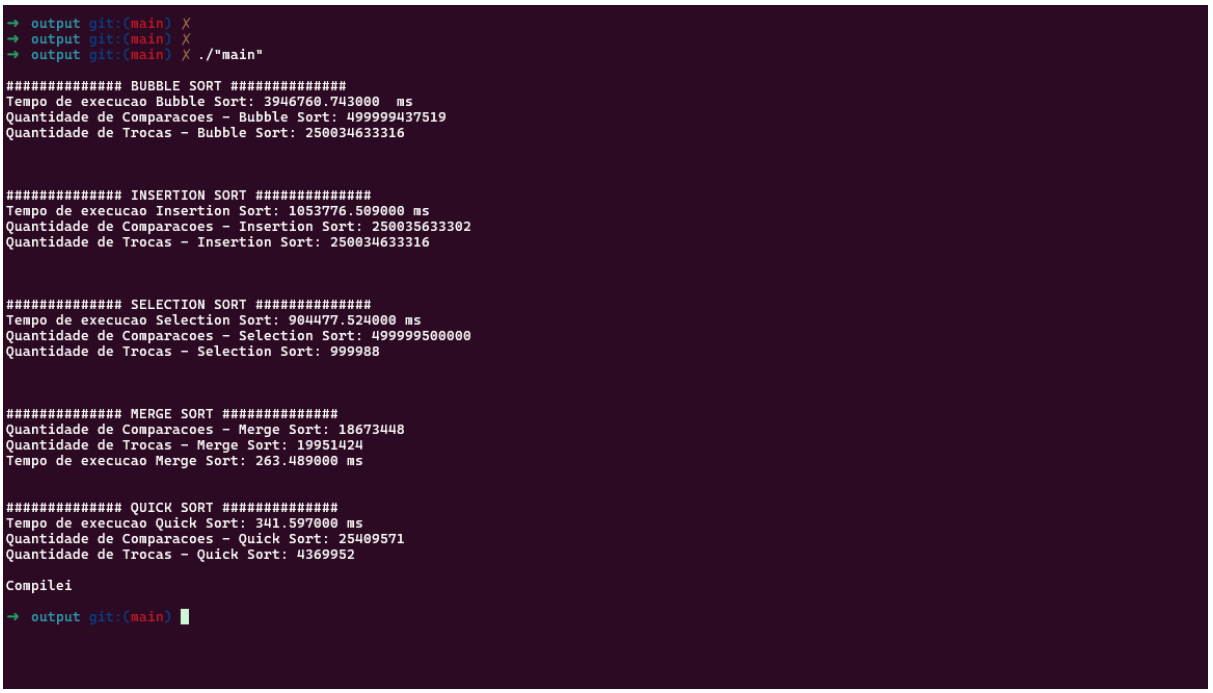


Figura 12. Ordem Crescente (Melhor caso)

```
➜  output git:(main) X ./"main"
➜  output git:(main) X ./"main"
➜  output git:(main) X ./"main"

############### BUBBLE SORT ###############
Tempo de execucao Bubble Sort: 2.252000  ms
Quantidade de Comparacoes - Bubble Sort: 999999
Quantidade de Trocas - Bubble Sort: 0



############### INSERTION SORT ###############
Tempo de execucao Insertion Sort: 3.058000 ms
Quantidade de Comparacoes - Insertion Sort: 999999
Quantidade de Trocas - Insertion Sort: 0



############### SELECTION SORT ###############
Tempo de execucao Selection Sort: 922532.553000 ms
Quantidade de Comparacoes - Selection Sort: 499999500000
Quantidade de Trocas - Selection Sort: 0



############### MERGE SORT ###############
Quantidade de Comparacoes - Merge Sort: 10066432
Quantidade de Trocas - Merge Sort: 19951424
Tempo de execucao Merge Sort: 153.720000 ms


############### QUICK SORT ###############
Tempo de execucao Quick Sort: 865907.920000 ms
Quantidade de Comparacoes - Quick Sort: 444199699619
Quantidade de Trocas - Quick Sort: 0

Compilei

➜  output git:(main) X
```

Figura 13. Ordem Decrescente (Pior caso)

```
➜  ESDD cd metodos-de-classificacao
➜  metodos-de-classificacao git:(main) X cd output
➜  output git:(main) X ulimit -s unlimited
➜  output git:(main) X ./"main"

############# BUBBLE SORT #############
Tempo de execucao Bubble Sort: 3533926.470000  ms
Quantidade de Comparacoes - Bubble Sort: 499999500000
Quantidade de Trocas - Bubble Sort: 499999500000



############# INSERTION SORT #############
Tempo de execucao Insertion Sort: 2076709.011000 ms
Quantidade de Comparacoes - Insertion Sort: 499999500000
Quantidade de Trocas - Insertion Sort: 499999500000



############# SELECTION SORT #############
Tempo de execucao Selection Sort: 1158457.329000 ms
Quantidade de Comparacoes - Selection Sort: 499999500000
Quantidade de Trocas - Selection Sort: 500000



############# MERGE SORT #############
Quantidade de Comparacoes - Merge Sort: 9884992
Quantidade de Trocas - Merge Sort: 19951424
Tempo de execucao Merge Sort: 162.919000 ms


############# QUICK SORT #############
Tempo de execucao Quick Sort: 1035408.308000 ms
Quantidade de Comparacoes - Quick Sort: 499999999999
Quantidade de Trocas - Quick Sort: 500000

Compilei

➜  output git:(main) X
```

Por fim, após diversos métodos pesquisados e várias horas de compilação, pelo excesso de dados, os testes foram finalizados, além da análise e tratamento dos dados foram feitas através do presente relatório.

O Repositório encontra-se disponível no GitHub em:
https://github.com/RafaelMatiass/metodos-de-classificacao

Artigo referência:
https://periodicos.ufersa.edu.br/ecop/article/view/7082/6540