Trabalho de Estrutura de Dados
Instituto Federal de Educação, Ciência e Tecnologia de São Paulo
Estrutura de Dados 1 - Professor Viana

Alunos:
João Gustavo dos Santos - SP3154211
Andrey Gustavo Segantin dos Santos - SP3155439

1. Introdução

Este trabalho tem como objetivo analisar o comportamento de diferentes algoritmos de ordenação. Ao final do relatório, será possível compreender o tempo de execução de cada algoritmo em comparação com os outros.

2.1 Metodologia utilizada

Para a criação deste relatório, desenvolvemos um código em linguagem C que lê o arquivo disponibilizado pelo professor, mede o tempo de execução de cada algoritmo de ordenação e gera um relatório comparativo como arquivo de texto. Os testes foram realizados com conjuntos de dados variando de 500 a 5000 elementos.

2.2 Explicação algoritmo Bubble Sort

Bubble Sort é um algoritmo de ordenação que se baseia no princípio de comparar os elementos adjacentes de um vetor, realizando trocas de posição caso estejam fora de ordem. O método percorre a estrutura diversas vezes, movendo os maiores valores para a última posição ao final de cada ciclo. Pode-se fazer um paralelo com “bolhas” subindo na água, o que inspirou seu nome.

Isso é possível porque, a cada passagem pelo vetor, pares de elementos vizinhos vão sendo comparados e trocados, se necessário. Desse modo, o maior elemento da parte não ordenada é movido para sua posição correta ao fim de cada ciclo. Esse processo se repete até que o vetor esteja completamente ordenado.

Complexidade do Bubble Sort:

- Melhor caso (O(n)): ocorre quando o vetor já está ordenado. Assim, o algoritmo percorre a estrutura apenas uma vez para verificar isso, não realizando trocas.
- Pior caso e médio ($O(n^2)$): ocorre quando o vetor está ordenado na ordem inversa, sendo necessário percorrer a estrutura diversas vezes e realizando várias trocas.

Vantagem do Bubble Sort:

- É um algoritmo fácil de ser entendido e implementado.

Desvantagem do Bubble Sort:

- Devido a seu tempo de complexidade de $O(n^2)$, o método bolha não é adequado para ordenar vetores com muitos elementos, sendo uma opção mais lenta para isso.

```c
void bubbleSort(int *vetor, int t){
  int i, j, x;

  for(i=0; i<t-1; i++) {
    for(j=0; j<t-i-1; j++) {
      if(vetor[j] > vetor[j+1]) {
        x = vetor[j];
        vetor[j] = vetor[j+1];
        vetor[j+1] = x;
      }
    }
  }
}
```

Algorítimo Bubble Sort

2.3 Explicação algorítimo Insertion Sort

O insertion Sort é um algoritimo de organização simples que percorre o vetor a partir do segundo elemento e o compara com os elementos anteriores, deslocando-os para frente caso sejam maiores que o elemento atual. Esse processo se repete até que todos os elementos sejam inseridos nas posições corretas.

```c
void insertionSort(int *vetor, int t) {
  int i=0, j, x;

  for (i=1; i<=t; i++) {
    x = vetor[i];
    for (j=i; j>0 && x<vetor[j-1]; j--) {
      vetor[j] = vetor[j-1];
    }
    vetor[j] = x;
  }
}
```

Algorítimo Insertion Sort

A complexidade do Insertion Sort depende da disposição inicial dos elementos no vetor:

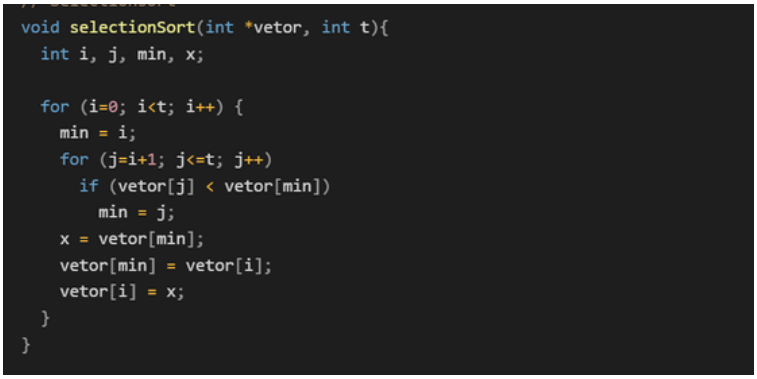
- Melhor caso (O(n)): Quando o vetor já está ordenado o algoritmo apenas percorre o vetor sem realizar trocas.
- Caso médio ($O(n^2)$): Em média, a complexidade do algoritmo é quadrática, uma vez que cada elemento precisa ser comparado e possivelmente movido várias vezes.
- Pior caso ($O(n^2)$): Acontece quando o vetor está em ordem decrescente, pois cada elemento precisa ser deslocado por todo o vetor antes de ser inserido na posição correta.

2.3 Explicação algoritmo Selection Sort

Selection Sort é um método de ordenação que verifica o menor elemento da parte não ordenada de um vetor, posicionando-o corretamente. Esse processo é repetido até que a estrutura esteja ordenada.

Inicialmente, o algoritmo percorre o vetor, realizando comparações entre seus elementos, com o objetivo de descobrir onde está o menor valor. Enquanto essa busca é realizada, a posição do menor elemento encontrado é salva em uma variável auxiliar.

Após essa busca, é realizada uma troca entre o menor elemento encontrado e o da primeira posição não ordenada do vetor. Esse processo se repete de forma que, ao final de cada ciclo, o menor valor ainda não ordenado seja trocado com a primeira posição ainda não ordenada. Assim, o menor valor é colocado na primeira posição, o segundo menor na segunda... até que a ordenação esteja completa.



```c
void selectionSort(int *vetor, int t){
  int i, j, min, x;

  for (i=0; i<t; i++) {
    min = i;
    for (j=i+1; j<=t; j++)
      if (vetor[j] < vetor[min])
        min = j;
    x = vetor[min];
    vetor[min] = vetor[i];
    vetor[i] = x;
  }
}
```

Algorítimo Selection Sort

Complexidade do Selection Sort:

Sempre apresenta complexidade de $O(n^2)$. Isso ocorre devido à sua estrutura de execução, já que o algoritmo utiliza dois laços aninhados, o que resulta em um número quadrático de operações.

- O primeiro laço percorre os n elementos, um por um = O(n)
- O segundo laço compara esse elemento com os outros elementos do vetor = O(n)
- Complexidade geral = O(n) * O(n) = O(n*n) = $O(n^2)$

Vantagem do Selection Sort:

- É um algoritmo intuitivo e fácil de ser aprendido, além de ser simples para implementar.
- Requer um número menor de trocas em comparação com o Bubble Sort, por exemplo.

Desvantagem do Selection Sort:

- Possui necessariamente um tempo de complexidade de $O(n^2)$, tornando-o uma opção mais lenta em comparação com outros métodos.

2.4 Explicação algoritmo Merge Sort

O Merge Sort é um algoritmo de ordenação que se baseia no paradigma da Divisão e Conquista. De forma simples, esse método funciona por meio de três etapas principais: dividir o vetor em duas metades (Divisão), ordenar cada uma das metades e combinar as metades ordenadas para remontar o vetor (Conquista).

Para que esse processo funcione, o algoritmo utiliza o conceito de recursividade para dividir o vetor original em subvetores menores, repetindo essas divisões até que cada subvetor possua um único elemento. Após essa etapa, ocorre a comparação de pares desses subvetores, trocando seus elementos se necessário.À medida que as funções recursivas retornam, os subvetores vão sendo combinados de forma ordenada. A cada combinação, os elementos que vieram dos dois subvetores são ordenados novamente no vetor combinado. Ao final da recursividade, o vetor original será reconstruído, mas de forma ordenada.

```c
void conquista(int *vetor, int esq, int dir, int meio) {
  int tamEsq = meio - esq + 1;
  int tamDir = dir - meio;
  int auxEsq[tamEsq], auxDir[tamDir];

  for (int i = 0; i < tamEsq; i++) {
        auxEsq[i] = vetor[i + esq];
  }
    for (int j = 0; j < tamDir; j++) {
        auxDir[j] = vetor[j + meio + 1];
  }

  int i = 0, j = 0, k = esq;

  while(i < tamEsq && j < tamDir) {
    if(auxEsq[i] < auxDir[j]) {
      vetor[k] = auxEsq[i];
      i++;
    }
    else {
      vetor[k] = auxDir[j];
      j++;
    }
    k++;
  }

    while (i < tamEsq) {
        vetor[k] = auxEsq[i];
        i++;
        k++;
    }

    while (j < tamDir) {
        vetor[k] = auxDir[j];
        j++;
        k++;
    }
}

void mergeSort(int *vetor, int esq, int dir) {
  if (esq < dir) {
    int meio = (esq + dir) / 2;

  mergeSort(vetor, esq, meio);
  mergeSort(vetor, meio + 1, dir);
  conquista(vetor, esq, dir, meio);
  }
}
```

Algorítimo Merge Sort

Complexidade do Merge Sort: O(n log n)

Vantagem do Merge Sort:

- Devido a seu tempo de complexidade, o Merge Sort pode ser extremamente mais eficiente do que outros métodos para ordenar vetores de muitos elementos

Desvantagem do Merge Sort:

- O algoritmo requer espaço extra, pois precisa armazenar temporariamente os subvetores durante a recombinação, sendo inadequado em situações com limitação de memória.

2.5 Explicação algoritmo Quick Sort

O Quick Sort é um algoritmo de ordenação que se baseia no paradigma da Divisão e Conquista. Diferente do Merge Sort, que realiza divisões para separar o vetor original em dois subvetores equilibrados, o Quick Sort utiliza um valor pivô como referência para a separação.

Inicialmente, o algoritmo seleciona algum elemento do vetor original como pivô (geralmente um valor de alguma das extremidades). Após isso, o restante dos elementos são comparados com o pivô: aqueles com valor menor são movidos para a esquerda e os maiores para a direita. Ao final dessa organização, o elemento utilizado como pivô já estará em sua posição correta no vetor e os elementos à sua esquerda e à sua direita formarão dois subvetores.

O processo de Divisão continua se repetindo de forma recursiva, sendo que, em cada novo subvetor, é escolhido um novo pivô para ser usado como referência para a repartição. Ao final das divisões, os elementos já estarão ordenados, portanto ocorre a fase de Conquista. Esta combina os subvetores recursivamente, remontando o vetor original de forma ordenada.

```c
void troca(int vetor[], int i, int j) {
  int aux = vetor[i];
  vetor[i] = vetor[j];
  vetor[j] = aux;
}

int divisao(int vetor[], int inicio, int fim) {
  int pivo = vetor[fim];
  int indicePivo = inicio;

  for(int i = inicio; i < fim; i++) {
    if(vetor[i] <= pivo) {
      troca(vetor, i, indicePivo);
      indicePivo++;
    }
  }

  troca(vetor, indicePivo, fim);
  return indicePivo;
}

void quickSort(int vetor[], int inicio, int fim) {
  if (inicio < fim) {
    int indicePivo = divisao(vetor, inicio, fim);
    quickSort(vetor, inicio, indicePivo - 1);
    quickSort(vetor, indicePivo + 1, fim);
  }
}
```

Algorítimo Quick Sort

Complexidade do Quick Sort:

- Melhor caso ($\Omega(n \log n)$): Ocorre quando o elemento pivô divide a matriz em duas metades iguais.
- Médio caso ($\theta(n \log n)$): Em média, o pivô divide a matriz em duas partes, mas não necessariamente iguais.
- Pior caso ($O(n^2)$): Ocorre quando o menor ou maior elemento é sempre escolhido como pivô (por exemplo, matrizes ordenadas).

Vantagem do Quick Sort:

- Devido a sua complexidade média, o Quick Sort é um algoritmo rápido e eficiente para vetores muito grandes.

Desvantagem do Quick Sort:

- Possui eficiência reduzida em situações que o pivô escolhido não é apropriado, como em vetores já ordenados, por exemplo.

2.6. Explicação algorítimo Heap Sort

O Heap Sort é um algoritmo de ordenação baseado na estrutura de dados Heap, que é uma árvore binária com duas propriedades:

- Balanceamento: se estiver incompleta os dados do último nível estão todas nas posições mais a esquerda.
- Estrutural: o valor armazenado em cada nó é maior que os de seus filhos.

Para a execução do Heap Sort foi necessário utilizar 3 funções: o “sift”, o “build” e o “heapSort”.

O “sift” tem o papel de fazer com que o vetor sempre respeite a propriedade estrutural do Heap, fazendo com que os nós pais sempre tenham valores maiores do que os filhos.

```c
void sift(int *vetor, int i, int t){
  int esq = 2 * i;
  int dir = 2 * i + 1;
  int maior = i;

  if (esq <= t && vetor[esq] > vetor[maior]) {
    maior = esq;
  }

  if (dir <= t && vetor[dir] > vetor[maior]) {
    maior = dir;
  }

  if (maior != i) {
    int aux = vetor[i];
    vetor[i] = vetor[maior];
    vetor[maior] = aux;
    sift(vetor, maior, t);
  }
}
```

Algorítimo Sift Heap Sort

A função “build” tem o papel de organizar toda a árvore chamando a função "sift".

```c
void build(int *vetor, int t) {
  for (int i = t/2; i>=0; i--) {
    sift(vetor, i, t);
  }
}
```

Algorítimo Build Heap Sort

A função principal, "heapSort", é responsável por ordenar o vetor removendo o elemento da raiz e colocá-lo no final, organizando a estrutura de forma crescente.

```c
void heapSort(int *vetor, int t) {
  build(vetor, t);
  for (int i = t; i>0; i--) {
    int aux = vetor[i];
    vetor[i] = vetor[0];
    vetor[0] = aux;
    sift(vetor, 0, i-1);
  }
}
```

Algorítimo Heap Sort

O Heap Sort tem um desempenho eficiente devido ao uso da estrutura de heap. Na construção do Heap o processo de construção (build) tem complexidade O(n). Na ordenação cada remoção do maior elemento envolve a restauração do heap, que tem complexidade O(log n). Como há n remoções, o tempo total é O(n log n).

- Melhor caso (O(n log n)): O vetor já está ordenado, mas o algoritmo ainda precisa construir o heap e ajustar os elementos.
- Caso médio (O(n log n)): Para vetores aleatórios, o tempo médio necessário para reorganizar o heap se mantém proporcional a n log n.
- Pior caso (O(n log n)): Ocorre quando o vetor está ordenado ao contrário, mas o Heap Sort mantém seu desempenho eficiente. por todo o vetor antes de ser inserido na posição correta.

2.7. Explicação do main

Na função main, são criados dois ponteiros: um para o arquivo contendo os números a serem ordenados e outro para o arquivo de texto onde será salvo o relatório com os tempos de execução dos algoritmos de ordenação.
Para contarmos o tempo utilizamos as funções:

- clock_t begin = clock();
- clock_t end = clock();
- time_spent += (double)(end - begin) / CLOCKS_PER_SEC;

Essas funções estão disponíveis na biblioteca “time.h".

```c
int main() {
  FILE * file;
  FILE * arqtxt;
  char nome[100];
  int *vetor = NULL;
  int t;
  double time_spent = 0.0;

  printf("Digite o endereço do arquivo: ");
  scanf("%99s", nome);

  remove("nome_algoritmo_n.txt");
  arqtxt = fopen("nome_algoritmo_n.txt", "a");

  file = fopen(nome, "r");

  if(file == NULL) {
    printf("Nao foi possivel abrir o arquivo.");
    return 1;
  }
```

Algorítimo main

O usuário deve inserir o endereço do arquivo que contém os números desordenados. Antes de iniciar o processamento, o arquivo de relatório é removido para garantir que, caso já exista, não haja sobrecarga de dados. Isso é necessário, pois, ao utilizar o modo "a" no fopen, se o código for executado mais de uma vez, os dados seriam acrescidos no arquivo em vez de substituir os existentes, o que não é o comportamento desejado.

```
printf("\n\n--------------- Bubblesort ---------------\n\n");
fprintf(arqtxt, "\n\n--------------- Bubblesort ---------------\n\n");

for(t = TAM; t <= 5000; t += TAM) {
  vetor = (int *) malloc (t * sizeof(int));
  if(vetor == NULL) {
    printf("Nao foi possível alocar memória para o vetor...");
    fclose(file);
    return 1;
  }
    for(int f=0; f<1000; f++) {
      leituraArq(vetor, file, t);

      clock_t begin = clock();
      bubbleSort(vetor, t);
      clock_t end = clock();

      time_spent += (double)(end - begin) / CLOCKS_PER_SEC;

    }
  printf("%d: %f\n", t, time_spent/1000);
  fprintf(arqtxt, "%d: %f\n", t, time_spent/1000);
  time_spent = 0;

  free(vetor);
}
```

Algorítimo main: Relatório Bubble Sort

Para cada tipo de ordenação, o tempo de execução é calculado e registrado tanto no relatório quanto exibido no terminal. O processo é repetido para os diferentes tamanhos de vetores, sendo eles: 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500 e 5000. O tempo médio de execução é calculado com base em 1000 execuções de cada tipo de ordenação.

4. Gráficos e Resultados

Os gráficos a seguir mostram quanto do tempo cada tipo de ordenação demorou para ordenar os vetores de diferentes tamanhos.

	Bubble	Selection	Insertion	Merge	Quick	Heap
500	0,000393	0,000408	0,000661	0,000102	0,000053	0,000168
1000	0,001425	0,001512	0,002767	0,000236	0,000094	0,000311
1500	0,007955	0,003287	0,006098	0,000380	0,000119	0,000578
2000	0,015581	0,005934	0,010790	0,000472	0,000285	0,000661
2500	0,024747	0,009133	0,016782	0,000521	0,000334	0,000874
3000	0,036718	0,013313	0,024213	0,000647	0,000311	0,001218
3500	0,051147	0,017822	0,032923	0,000862	0,000364	0,001346
4000	0,061989	0,023336	0,042630	0,001120	0,000489	0,001493
4500	0,088326	0,029525	0,046298	0,001256	0,000503	0,001821
5000	0,114947	0,036040	0,066984	0,001482	0,000577	0,001959

Tabela dos Gráficos (Quantidade de elementos / Segundos)

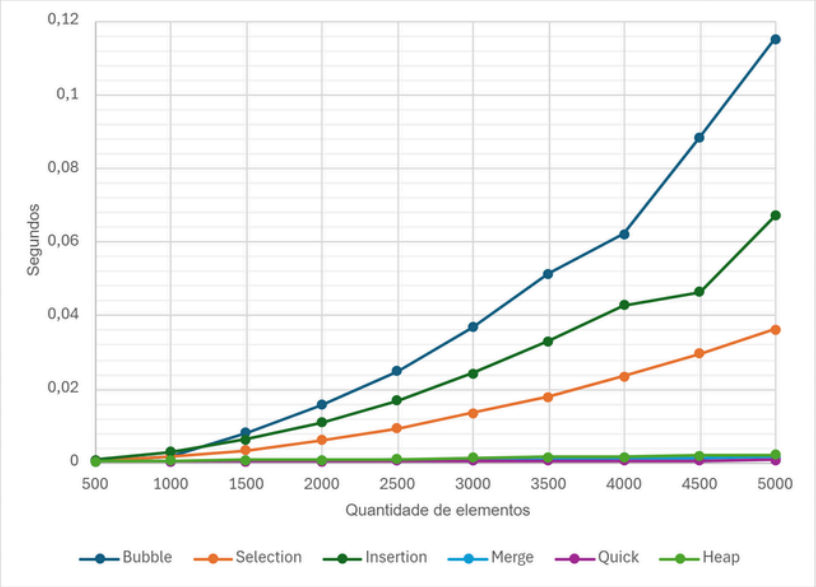


Gráfico 1: Comparação dos algoritmos dos tipos de ordenação: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort e Heap Sort.

Esse gráfico mostra que os algoritmos de complexidade quadrática, como Bubble Sort, Selection Sort e Insertion Sort, demonstram uma eficiência bem inferior à de Merge Sort, Quick Sort e Heap Sort, especialmente com o aumento do número de elementos. Entre os algoritmos quadráticos, o Insertion Sort foi o mais rápido. Isso porque ele realiza menos movimentações de elementos do que os outros dois, uma vez que desloca os valores diretamente para suas posições corretas, reduzindo o número de trocas. O Selection Sort ficou em segundo lugar. Embora também tenha complexidade $O(n^2)$, assim como o Insertion e o Bubble nos casos de vetores desordenados aleatoriamente, e realize um número fixo de comparações, ele faz menos trocas que o Bubble Sort, o que melhora um pouco sua eficiência. Já o Bubble Sort foi o mais lento dos três. Como ele faz várias passagens pelo vetor e realiza muitas trocas desnecessárias, seu tempo de execução acaba sendo o pior quando aplicado a uma lista desordenada.

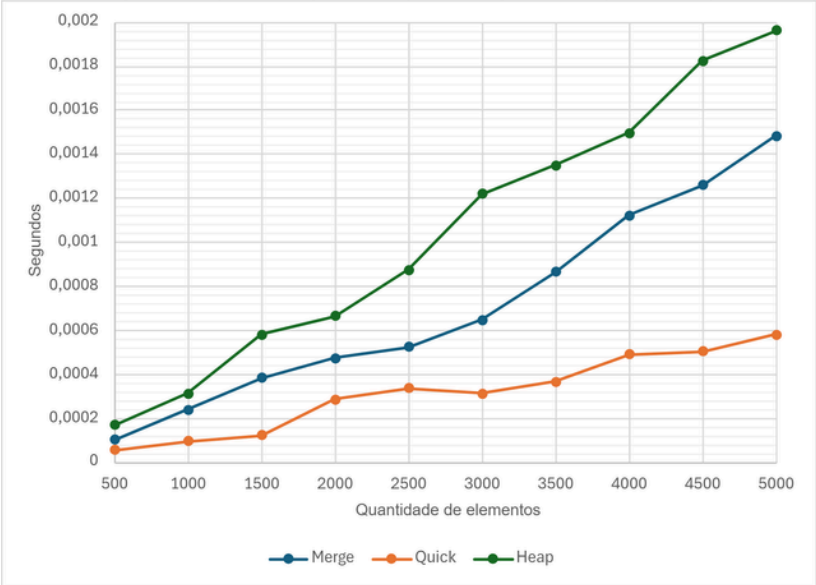


Gráfico 2: Comparação dos algoritmos dos tipos de ordenação: Merge Sort, Quick Sort e Heap Sort.

Nesse segundo gráfico, que compara os algorítimos Merge, Quick e Heap, é possível ver um desempenho muito próximo deles. Percebe-se que ao longo do teste o Quick Sort se mostrou o mais eficiente, o Merge o segundo e o Heap o terceiro.

Em comparação com o gráfico dos outros três algoritmos quadráticos (Bubble Sort, Selection Sort e Insertion Sort), é possível notar uma expressiva disparidade na eficiência para ordenar um vetor com muitos elementos. Enquanto os algoritmos quadráticos apresentam um crescimento exponencial do tempo de execução à medida que o número de elementos aumenta, os algoritmos Merge, Quick e Heap crescem de forma mais controlada, seguindo um padrão próximo a O(n log n), tornando-os significativamente mais escaláveis. Além disso, estes três outros métodos apresentaram um desempenho muito próximo entre si na situação analisada, diferente dos outros três que indicaram inclinações de curva com variações de crescimento mais distintas .

Entre esses três algoritmos, é possível notar que o mais lento para a ordenação dos 5000 elementos foi o Heap Sort. Como hipótese para isso ter ocorrido, pode-se citar o custo adicional de tempo, gasto para realizar a reestruturação da árvore binária após cada remoção do maior elemento, resultando em um maior número de trocas e comparações. Apesar disso, ainda se apresenta como uma opção mais eficiente do que Bubble, Selection e Insertion para a ordenação de um grande conjunto de dados. Ademais, pode ser uma opção mais viável do que Quick quando é necessário evitar a ocorrência do pior caso na ordenação de um vetor, pois possui uma garantia de complexidade O(n log n).

O segundo mais rápido entre os métodos testados foi o Merge Sort. Diferente do Heap, que é baseado na estrutura de árvore binária, os dois algoritmos mais rápidos compartilham a característica de adotarem como abordagem o paradigma da Divisão e Conquista. Dessa forma, o Merge consegue realizar a ordenação com um número menor de trocas e comparações do que os demais métodos anteriores. Mesmo que não tenha sido o mais rápido no teste, pode se apresentar como uma excelente opção para ordenar grandes vetores, pois possui uma garantia de complexidade O(n log n), sendo mais apropriado nos piores casos do que o método a seguir.

Por fim, o método que melhor performou nos testes foi o Quick Sort, também relacionado ao paradigma da Divisão e Conquista. Esse algoritmo se mostrou ser o mais eficiente para o caso, sendo apropriado para a ordenação de grandes vetores. Entretanto, é necessário ressaltar que essa eficiência não é garantida em todos os casos, posto que más escolhas de pivô para a ordenação podem resultar no aumento do número de trocas nesse método. No pior caso, o Quick Sort pode apresentar uma complexidade quadrática ($O(n^2)$), não tendo uma expressiva vantagem em relação ao outros métodos, se assemelhando ao Bubble, Selection e Insertion. Apesar disso, consegue se destacar nos casos médios e melhores, apresentando um ótimo desempenho em cenários como o do teste realizado.

5. Conclusão

Portanto, foi possível ver que o Quick Sort, de maneira geral, teve o melhor desempenho. No entanto, cada algoritmo tem seu próprio propósito e pode ser mais útil dependendo da situação. Alguns, como o Bubble Sort e o Insertion Sort, são mais fáceis de entender e implementar, enquanto outros, como o Merge Sort e o Quick Sort, são mais eficientes, mas exigem um nível maior de complexidade. Dessa forma, escolher o algoritmo certo é importante com base na situação, na quantidade de dados e nos recursos que você tem para execução.

6. Bibliografia

- https://www.geeksforgeeks.org/bubble-sort-algorithm/
- https://www.geeksforgeeks.org/selection-sort-algorithm-2/
- https://www.geeksforgeeks.org/merge-sort/
- https://www.geeksforgeeks.org/quick-sort-algorithm/
- https://www.alura.com.br/artigos/merge-quick-sort-qual-o-melhor-algoritmo?srsltid=AfmBOoqlvoOpXBIthb6
- https://www.devmedia.com.br/algoritmos-de-ordenacao-analise-e-comparacao/28261