**Documentação Trabalho Prático Retornando para casa**

**Estruturas de Dados**

**Nome: Heloiza Aparecida dos Santos**

**Matrícula: 2017086953**

Departamento de Ciencia da Computação – Universidade Federal de Minas Gerais (UFMG) Belo Horizonte – MG – Brasil

heloizas@ufmg.br

1. **Introdução**

Este trabalho prático teve como objetivo desenvolver conhecimentos a cerca dos métodos de ordenação estudados nas aulas da disciplina Estruturas de Dados DCC205, incluindo os algoritmos de ordenação **Insertion Sort, Heap Sort, Quick Sort, Quick Sort Melhorado e Shell**. Foi solicitado a ordenação de arquivos contendo nomes de planetas e a distância deles em saltos hiperespaciais do mais distante para o mais perto (decrescente) da Terra.

1. **Implementação**

* A estratégia utilizada para desenvolver os algoritmos de ordenação, foi a implementação de classes para cada um dos métodos **Insertion Sort, Heap Sort, Quick Sort, Quick Sort Melhorado e Shell**. A classe principal é a Ordenacao, que deve receber 2 argumentos: *dados\_aleatorio.txt* ***ou*** *dados\_crescente.txt* ***ou*** *dados\_descrescente.txt* **e** *quantidadeLinhas*. A classe Ordenacao instancia a classe do método selecionado, que deve ser solicitado no Main.
* Os arquivos usados para desenvolvimento e análise foram os disponibilizados pelos monitores dia 05/03/2021, no arquivo *entradas-corrigidas.zip*.
* As classes estão organizadas da seguinte maneira:

|  |  |
| --- | --- |
| **Classes:** | **Principais responsabilidades:** |
| Vetor | Contém o nome e a distância do planeta. Possui duas funções:  - AdicionarNome(string):  - AdicionarDistancia(int): |
| Ordenacao | Recebe como argumento o arquivo com os dados e a quantidade de linhas a ser processada.  Responsabilidade e principais funções:  - GerarVetor(arquivo): Gera o vetor de cada linha do arquivo de acordo com a quantidade fornecida. Utiliza as funções AdicionarNome() e AdicionarDistancia() da classe Vetor.  - Instanciar funçção que chama a classe OrdenacaoHeap, OrdenacaoInsercao, OrdenacaoQuick, OrdenacaoQuickMelhorado **ou** OrdenacaoShell, de acordo com o algoritmo de seleção selecionado no Main.cc.  - ImprimirVetorOrdenado(Vetor): Imprime as 7 primeiras linhas do arquivo ordenado. |
| OrdenacaoInsercao | Método de ordenação Insertion Sort. Principal função:  - Ordena(Vetor, quantidadeLinhas): Realiza o método da inserção em ordem decrescente. |
| OrdenacaoHeap | Método de ordenação Heap Sort. Principais funções:  - Constroi(Vetor, tamanhoHeap, aux): Transforma o vetor em Heap (árvore).  - Ordena(Vetor, quantidadeLinhas): Realiza o método Heap Sort em ordem decrescente. |
| OrdenacaoQuick | Método de ordenação Quick Sort. Principais funções:  - Particao(Vetor, int, int, int\*, int\*): Rearranja de modo que os elementos menores fiquem do lado esquerdo do vetor.  - QuickSort(Vetor, int, int): Realiza o método Quick Sort em ordem decrescente. |
| OrdenacaoQuickMelhorado | Método de ordenação Quick Sort Melhorado (Modificado). Principais funções:  - QuickSortMelhorado(Vetor, int, int): Utiliza um algoritmo simples (inserção) para partições de tamanho pequeno.  - Particao(Vetor, int, int, int\*, int\*): Rearranja de modo que os elementos menores fiquem do lado esquerdo do vetor.  - QuickSort(Vetor, int, int): Realiza o método Quick Sort em ordem decrescente. |
| OrdenacaoShell | Método de ordenação Shell.  Principal função:  - Ordena(Vetor, quantidadeLinhas): Realiza o método da ordenação Shell em ordem decrescente. |

No arquivo Main do programa um objeto da classe Ordenacao é instanciado: ordenacao(arquivo, qtdLinhas), e apenas **uma** das funções da classe ordenação deve ser chamada (Insercao, HeapSort, QuickSort, QuickSortMelhorado, Shell).

Como padrão, está selecionado o método Heap Sort.

Text

Description automatically generated

* O código está organizado seguindo a seguinte estrutura de pastas:

projeto

- bin: executáveis gerados após compilação.

- src: códigos .cc.

- obj: arquivos .o.

- include: headers .h.

Makefile

* O trabalho foi desenvolvido e testado utilizando a seguinte configuração:  
  Sistema Operacional: macOS Catalina 10.15.7  
  Linguagem de programação implementada: C++  
  Compilador utilizado: GCC  
  Dados do processador: Intel Iris Plus Graphics 640  
  Memória RAM: 8 GB

1. **Instruções de compilação e execução**

Na pasta principal do programa, executar o comando make. Na pasta bin rodar o programa fornecendo o arquivo e quantidade desejada na linha de comando.

Exemplo:

|  |
| --- |
| make  cd bin  ./run.out dados\_aleatorio.txt qtdLinhas\* |

\*qtdLinhas é um inteiro 1<=qtdLinhas>=200000.

1. **Análise de complexidade das principais funções:**

|  |  |
| --- | --- |
| **Funções:** | **Complexidade:** |
| Classe: Vetor  AdicionarNome(string) | **Complexidade de Tempo:**  Complexidade constante Θ(1). Apenas atribuição.  **Complexidade de Espaço:**  Complexidade constante Θ(1). |
| Classe: Vetor  AdicionarDistancia(int) | **Complexidade de Tempo:**  Complexidade constante Θ(1). Apenas atribuição.  **Complexidade de Espaço:**  Complexidade constante Θ(1). |
| Classe: Ordenacao  GerarVetor(arquivo) | **Complexidade de Tempo:**  Nessa função o arquivo é mapeado em um vetor. Essa é a função onde o custo de tempo é maior, pois contém um laço, cuja duração se baseia na iteração de toda a entrada possuindo duração Θ(n).  **Complexidade de Espaço:**  Complexidade constante Θ(1). |
| Classe: Ordenacao  Qualquer função que instancie a classe de ordenação desejada.  Ex: Insercao(), HeapSort()... | **Complexidade de Tempo:**  São operações de atribuição que chamam construtores com complexidade Θ(n²) para Inserção, e Θ(nlog(n)) para HeapSort, Quick Sort, Quick Sort Melhorado e Shell.  **Complexidade de Espaço:**  Complexidade de espaço constante Θ(1). |
| Classe: Ordenacao  ImprimirVetorOrdenado(Vetor) | **Complexidade de Tempo:**  Contém um laço, cuja duração se baseia na iteração de duração Θ(n).  **Complexidade de Espaço:**  Complexidade constante Θ(1). |
| Classe: OrdenacaoInsercao  Ordena(Vetor, int) | **Complexidade de Tempo:**  Complexidade Θ(n²). 2 laços aninhados.  **Complexidade de Espaço:**  Complexidade constante Θ(1). |
| Classe: OrdenacaoHeap  Constroi(Vetor, int) | **Complexidade de Tempo:**  Complexidade constante Θ(1). Apenas comparações e atribuições.  **Complexidade de Espaço:**  Complexidade constante Θ(1). |
| Classe: OrdenacaoHeap  Ordena(Vetor, int) | **Complexidade de Tempo:**  Complexidade Θ(nlog(n)). Algoritmo de divisão e conquista.  **Complexidade de Espaço:**  Complexidade constante Θ(1). |
| Classe: OrdenacaoQuick  Particao(Vetor, int, int, int\*, int\*) | **Complexidade de Tempo:**  Complexidade Θ(nlog(n)). Algoritmo de divisão e conquista.  **Complexidade de Espaço:**  Complexidade constante Θ(1). |
| Classe: OrdenacaoQuick  QuickSort(Vetor, int) | **Complexidade de Tempo:**  Complexidade constante Θ(1). Apenas comparações e atribuições.  **Complexidade de Espaço:**  Complexidade constante Θ(1). |
| Classe: OrdenacaoQuickMelhorado  QuickSortMelhorado(Vetor, int, int) | **Complexidade de Tempo:**  Complexidade Θ(nlog(n)). Algoritmo de divisão e conquista.  **Complexidade de Espaço:**  Complexidade constante Θ(1). |
| Classe: OrdenacaoShell  Ordena(Vetor, int) | **Complexidade de Tempo:**  Complexidade Θ(nlog(n)). Extensão do método de inserção, considerando vários segmentos.  **Complexidade de Espaço:**  Complexidade constante Θ(1). |

1. **Análise e Comparação**

Para cada um dos três arquivos disponibilizados para ordenação (*dados\_aleatorio.txt, dados\_crescente.txt e dados\_descrescente.txt,* disponibilizados dia 05/03/2021, foram realizados testes para cada um dos algoritmos implementados.

Todos os algoritmos tiveram o seguinte desempenho:

Melhor desempenho: *dados\_decrescente.txt*

Pior desempenho: *dados\_crescente.txt*

Médio desempenho: *dados\_aleatorio.txt*

* **Insertion Sort:**

Algoritmo de complexidade Θ(n²). Foi o algoritmo que mostrou pior desempenho na ordenação do arquivo *dados\_crescente.txt e dados\_aleatorio.txt.* Algoritmo estável.

* **Heap Sort:**

Algoritmo de complexidade Θ(nlog(n)) dividir para conquistar. Pior desempenho que Quick Sort e Quick Sort melhorado. Desempenho parecido com o Shell. Não estável.

* **Quick Sort:**

Algoritmo de complexidade Θ(nlog(n)) dividir para conquistar. Foi o algoritmo com melhor desempenho, perdendo apenas para o Quick Sort melhorado. Método não estável.

* **Quick Sort Melhorado:**

Algoritmo de complexidade Θ(nlog(n)) dividir para conquistar. A melhoria implementada foi usar um algoritmo simples (inserção) para partições de tamanho pequeno, menos de 10 elementos. Foi escolhido essa modificação pela facilidade de adaptação.Teve o melhor desempenho de todos os algoritmos para todas as entradas. Método não estável.

* **Shell:**

Algoritmo de complexidade Θ(nlog(n)). Esse algoritmo foi escolhido para ser implementado por ser uma implementação simples.

O algoritmo é conhecido por ser uma extensão do método de inserção. A diferença é que no lugar de considerar o array a ser ordenado como um único segmento, vários segmentos são considerados e é aplicado o método de inserção direta em cada um deles.

O tempo de execução é sensível à ordem inicial do arquivo, tendo melhor desempenho em arquivo decrescente. É um método não estável.

O tempo foi medido com o comando **Time**, que mede quanto tempo leva para um comando ser utilizado, e o programa Excel para consolidação dos dados e geração dos gráficos.

Exemplo de como o tempo foi medido:

|  |
| --- |
| time ./run.out dados\_aleatorio.txt 50000 |

Foi observado que o tempo de execução varia (minimamente) de execução para execução, portanto foi realizado três execuções e o tempo a ser considerado foi a **média desses três valores**.

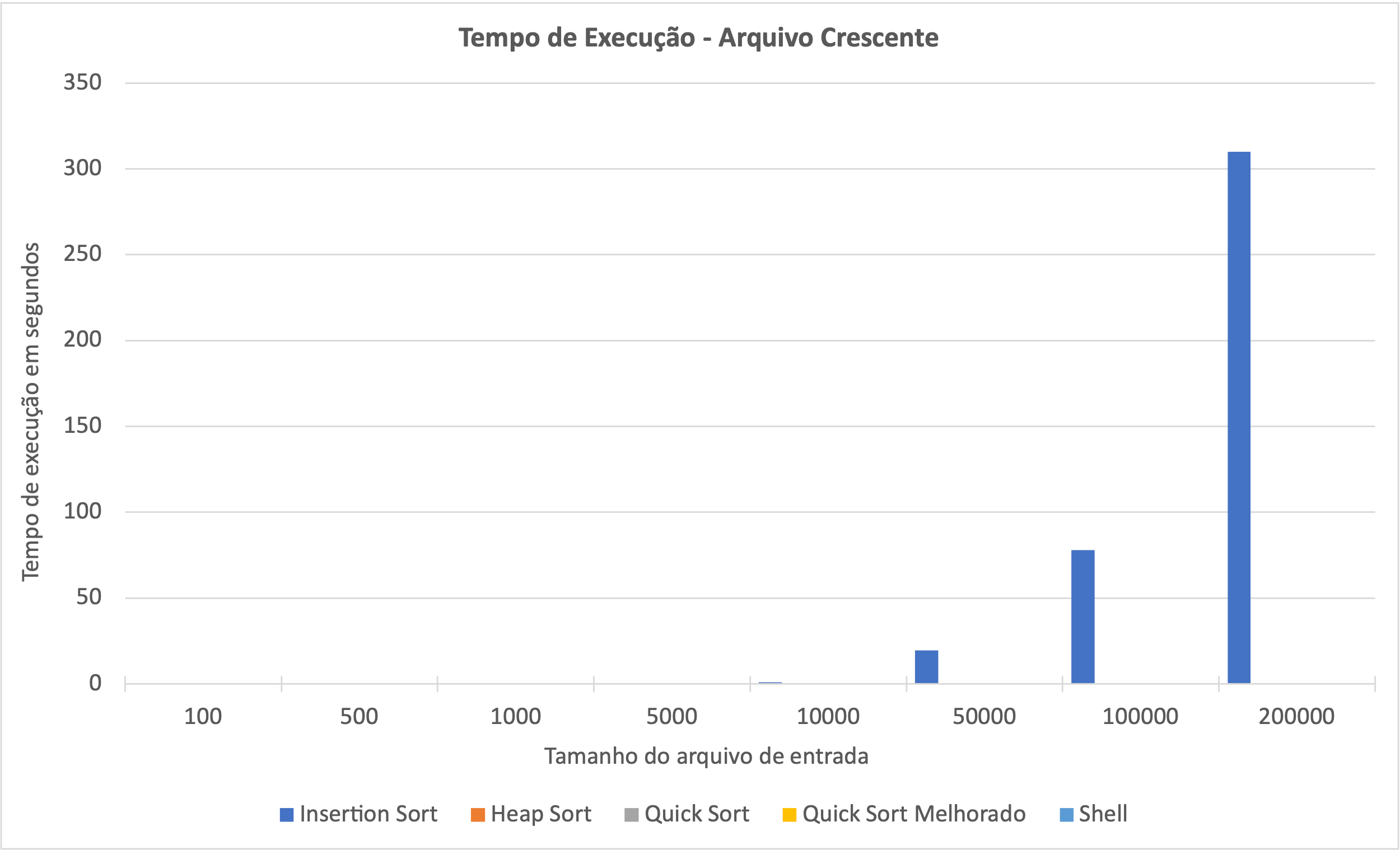
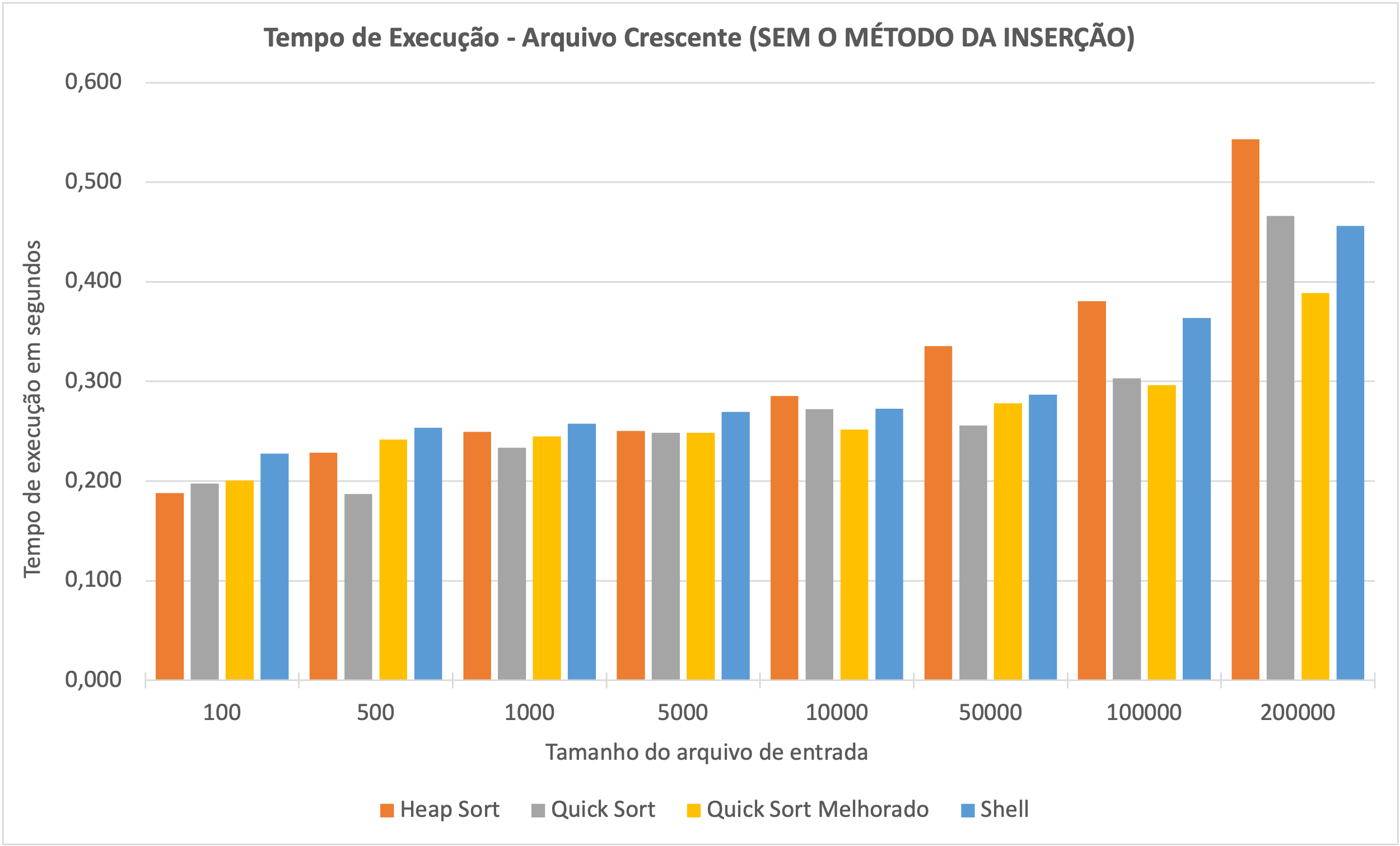
1. **Medição para o arquivo com dados crescentes:**

Tempo médio de três execuções:

Graphical user interface, application, table

Description automatically generated

O algoritmo de Inserção, foi o que teve o valor mais discrepante (fora de escala), portanto foi gerado dois gráficos, um gráfico considerando o algoritmo de inserção, e outro não.

Considerando método da inserção:  
Não considerando o método da inserção:

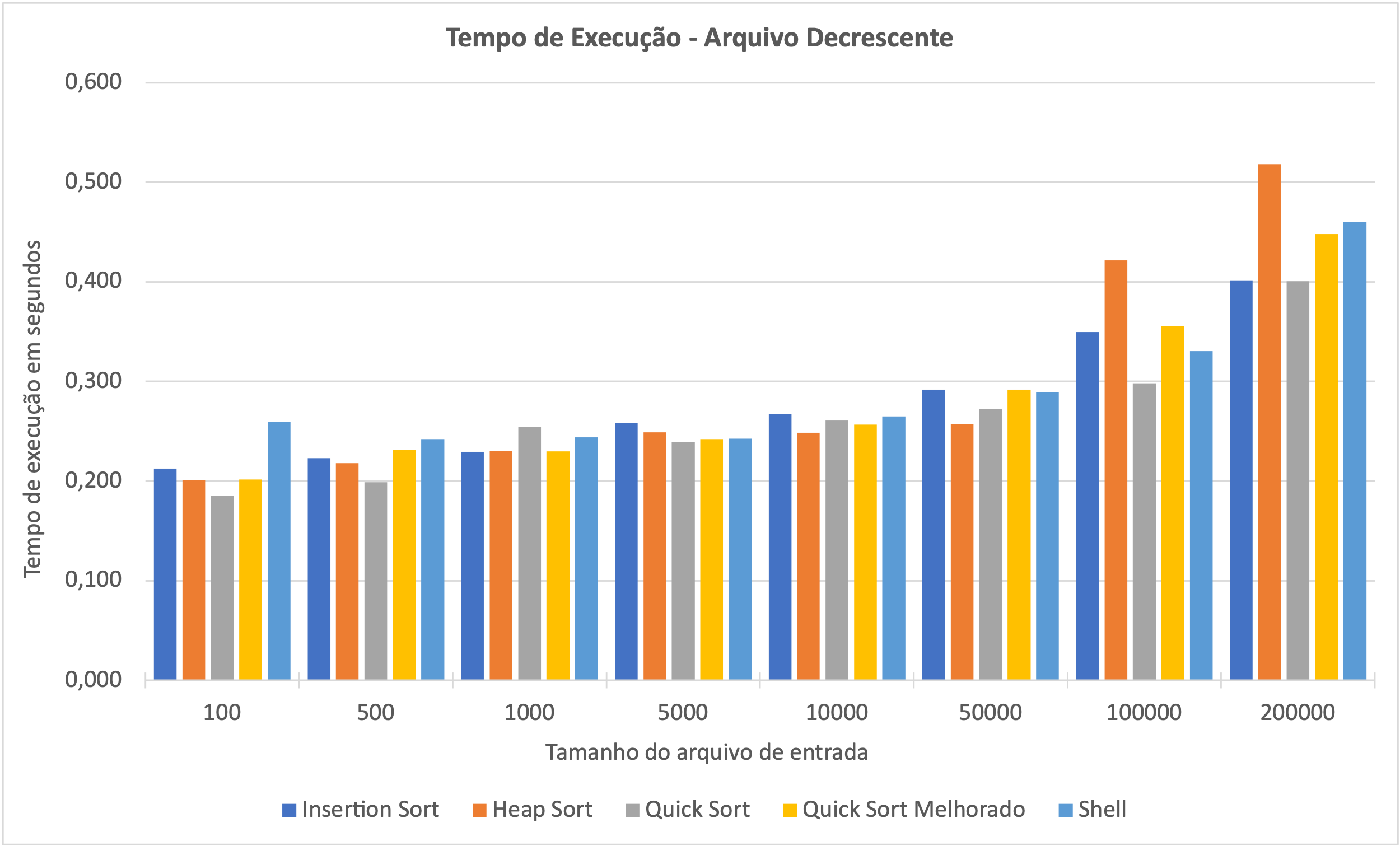
1. **Medição para o arquivo com dados decrescentes:**

Tempo médio de três execuções:

Table

Description automatically generated

Com o arquivo já ordenado (decrescente), todos os algoritmos tiveram melhor desempenho:



1. **Medição para o arquivo com dados aleatórios:**

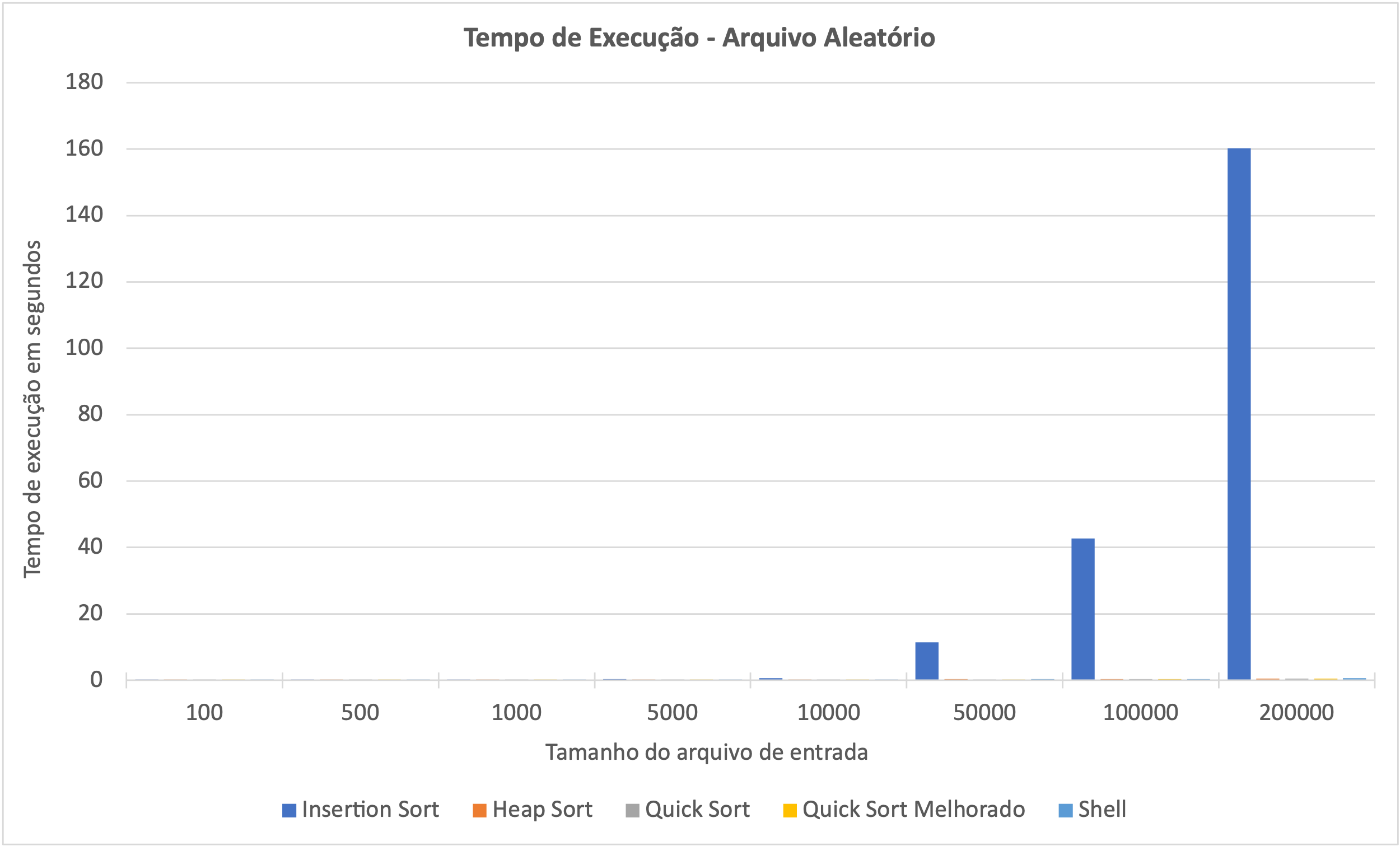
Tempo médio de três execuções:

Graphical user interface, table

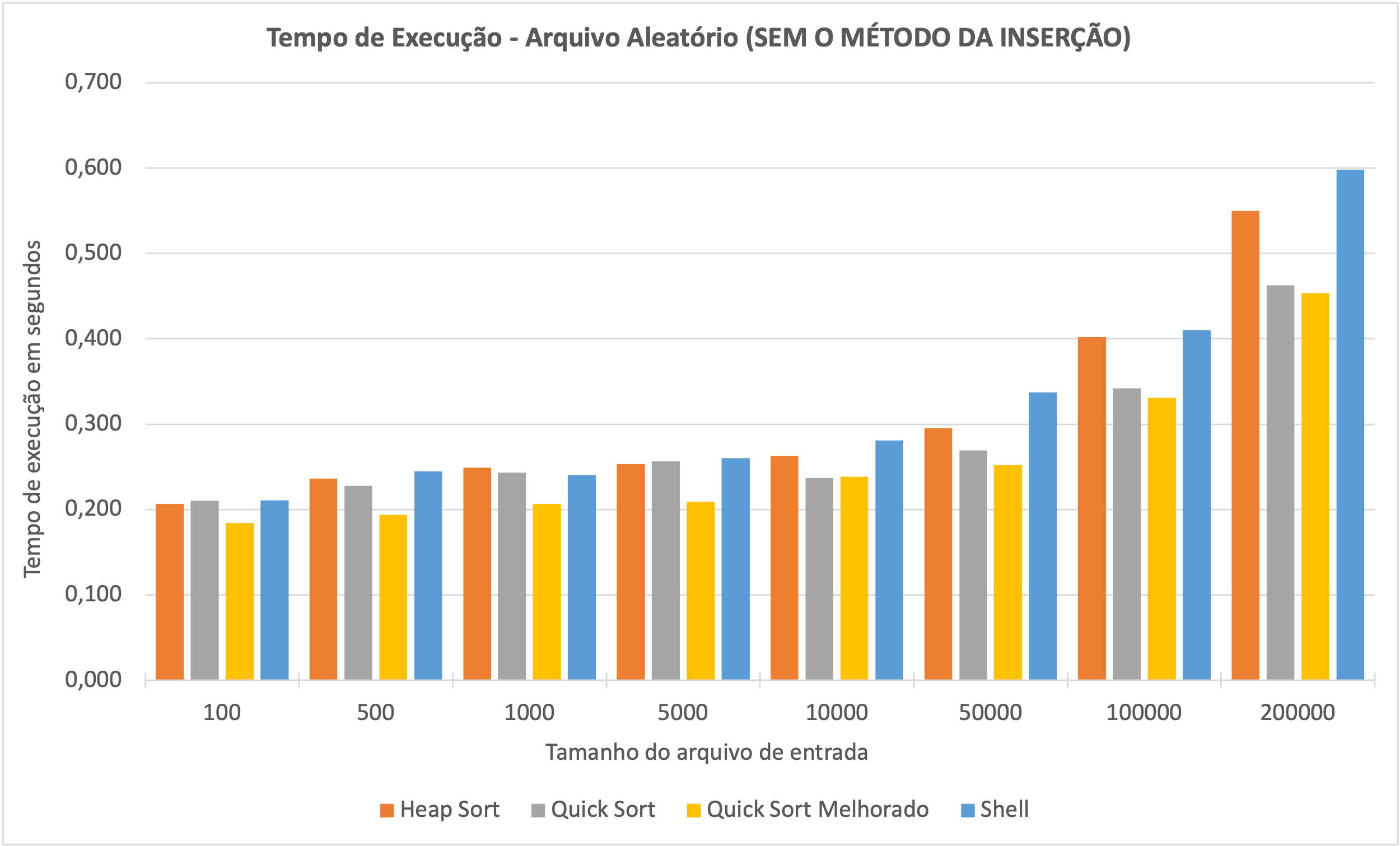
Description automatically generated

O algoritmo de Inserção, novamente teve o tempo mais discrepante (fora de escala), portanto foi gerado dois gráficos, um gráfico considerando o algoritmo de inserção, e outro não:

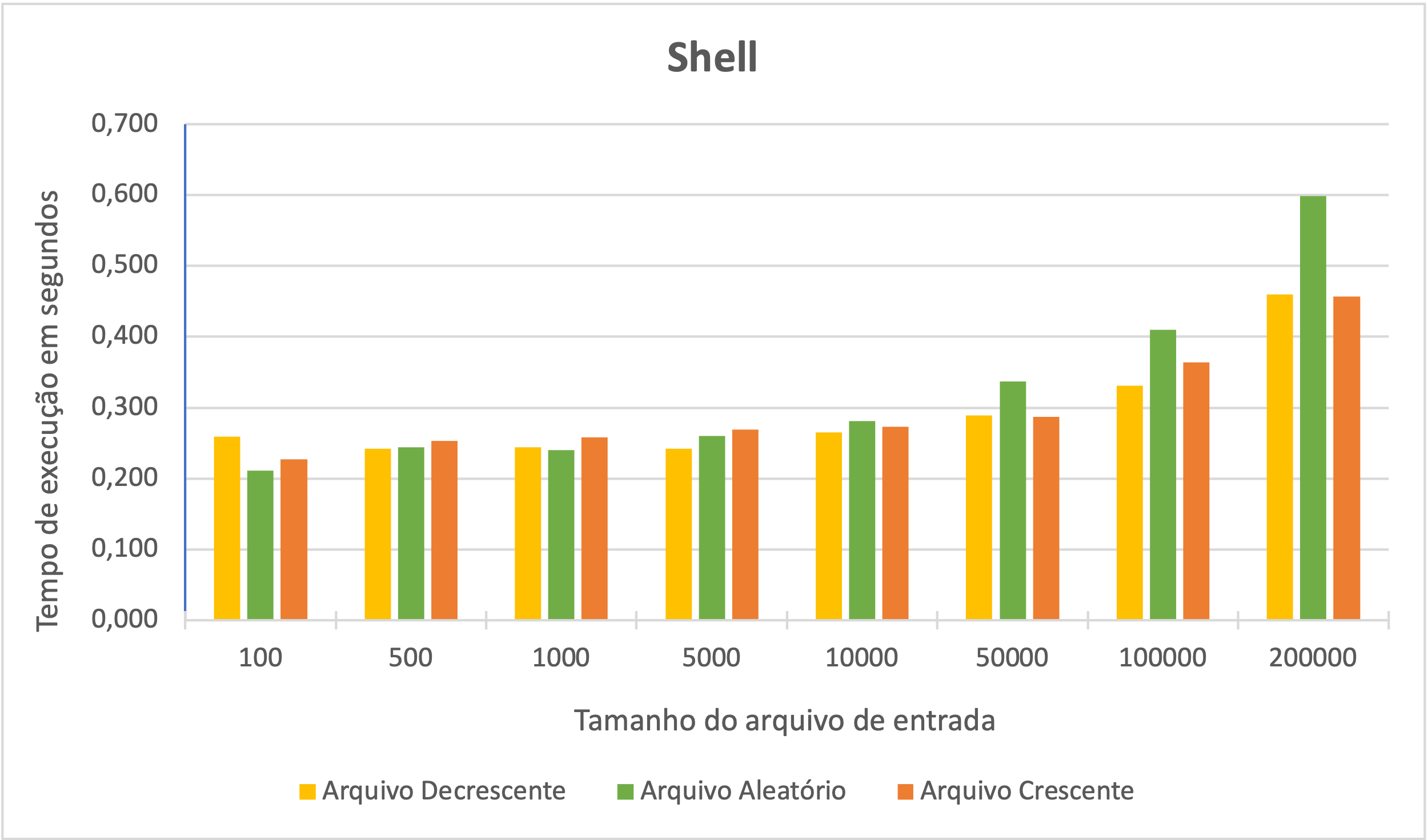
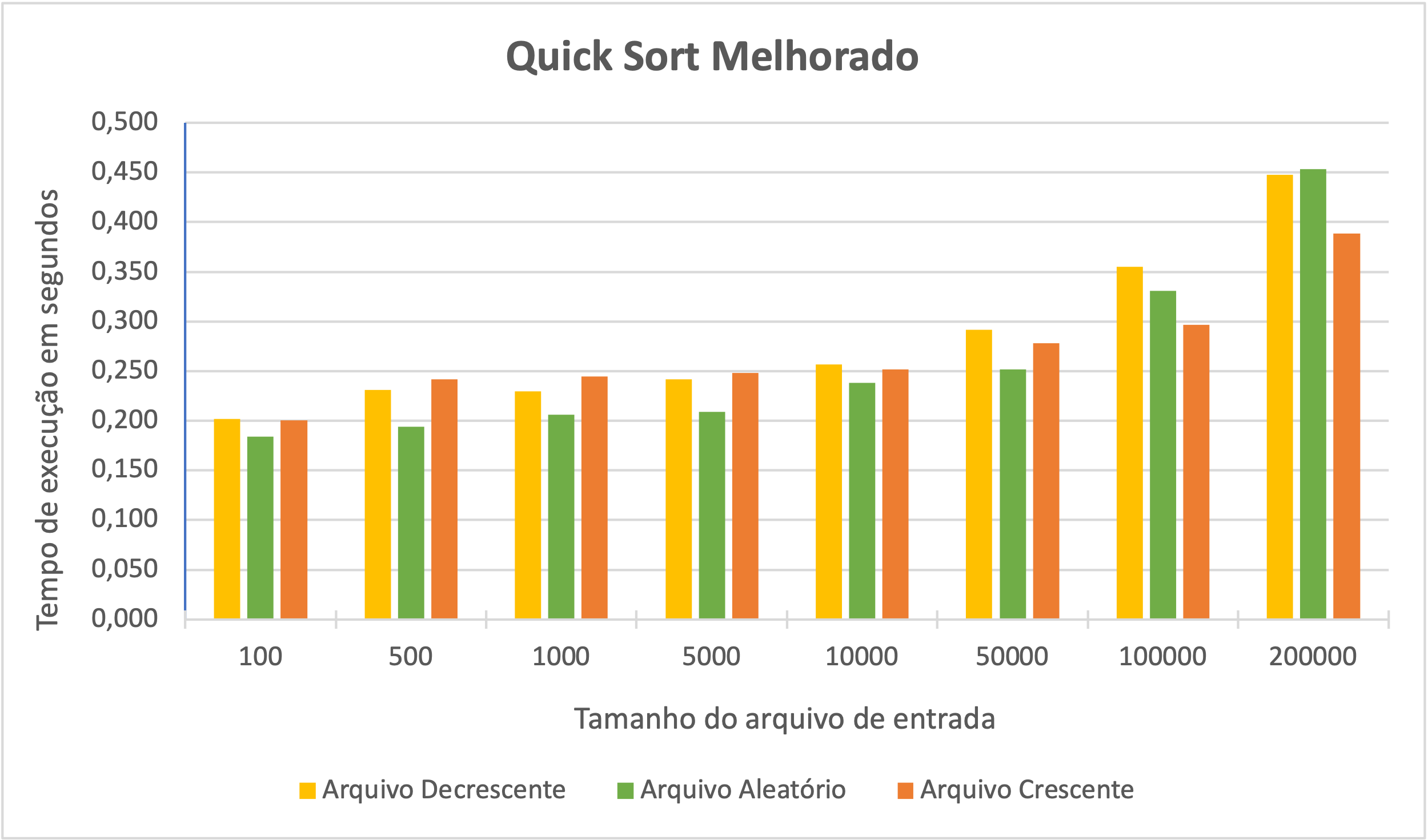
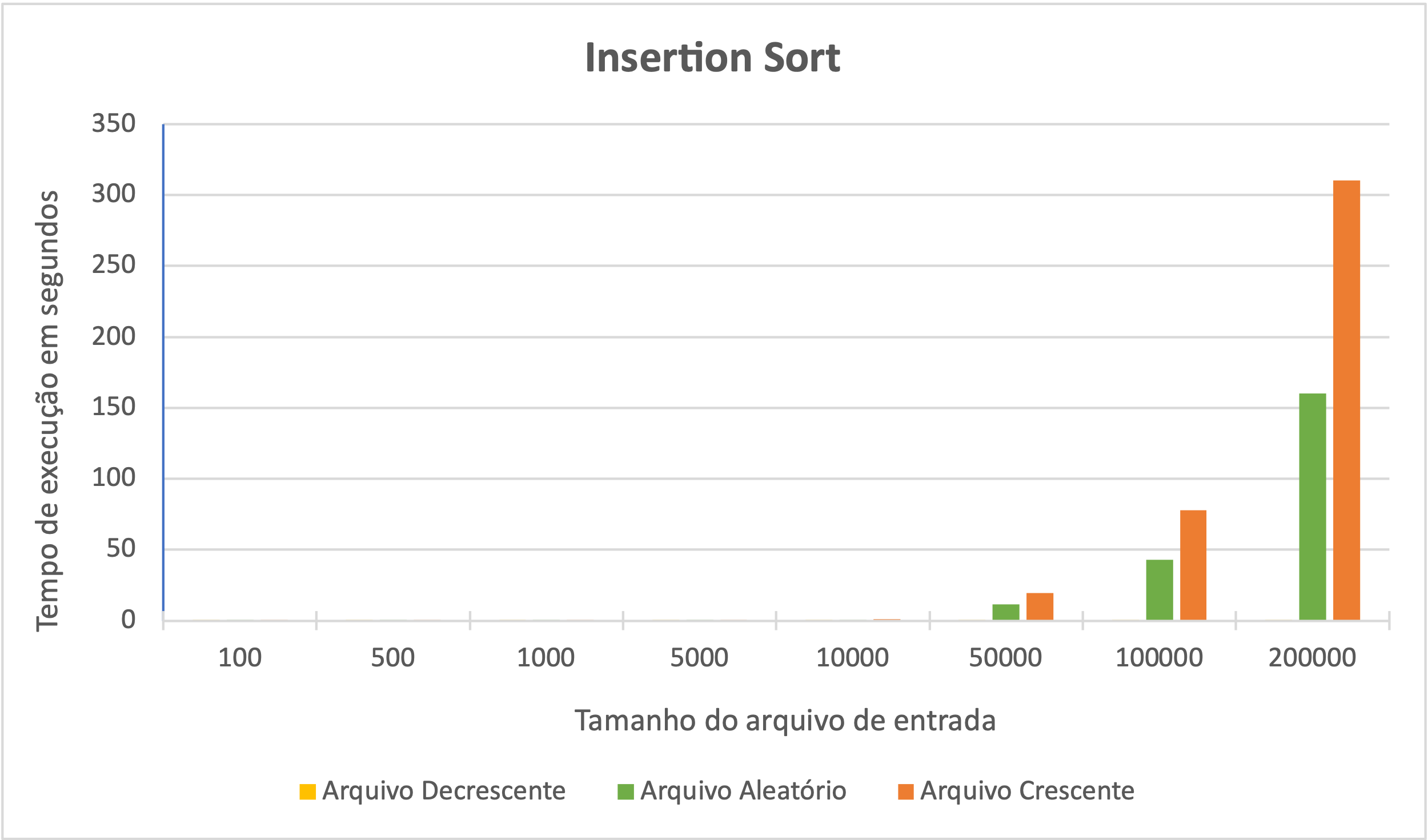
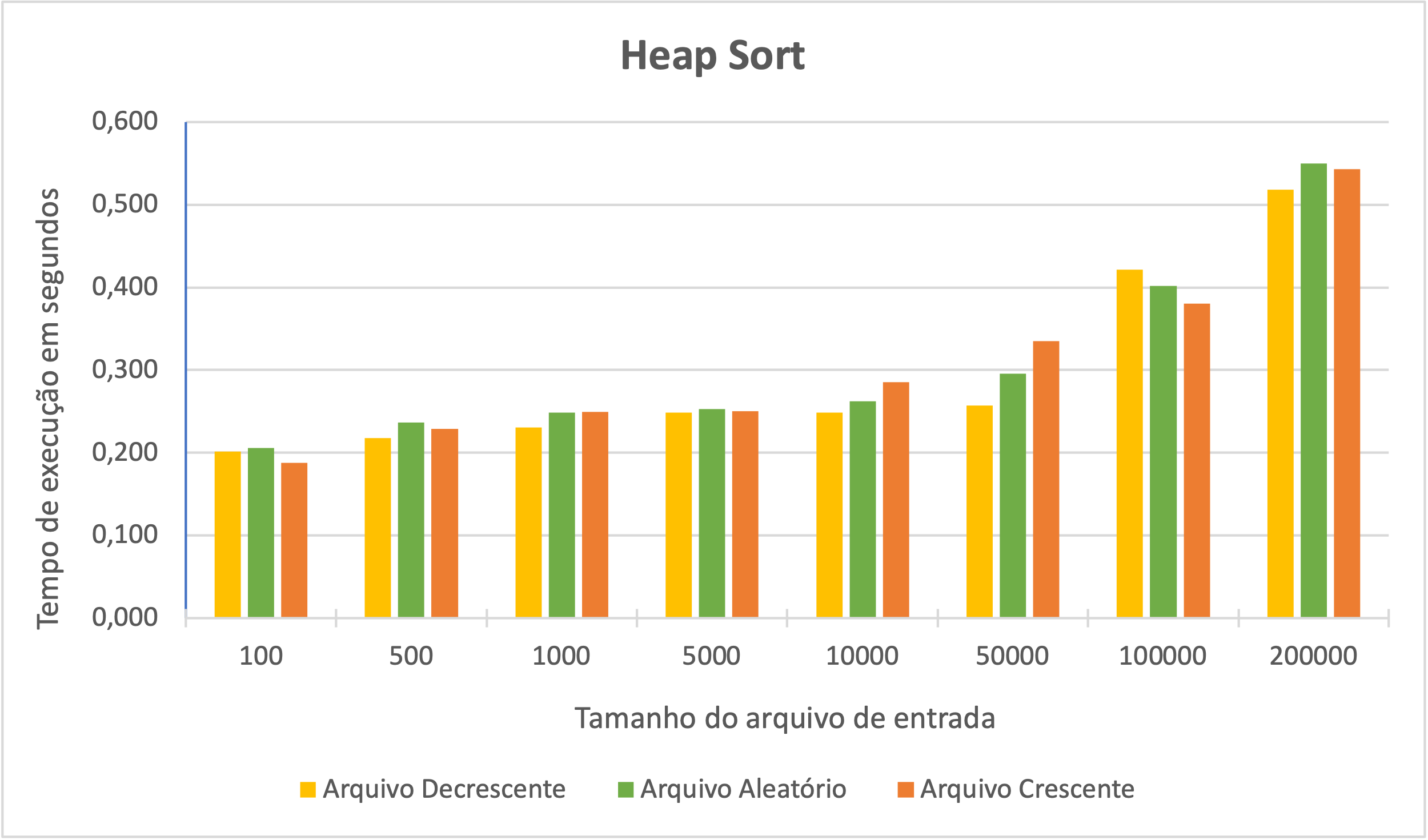
Considerando método da inserção:



Não considerando o método da inserção:



1. **Foi realizado uma comparação de como cada método se comporta levando em consideração o tamanho da entrada e a ordenação do arquivo:**

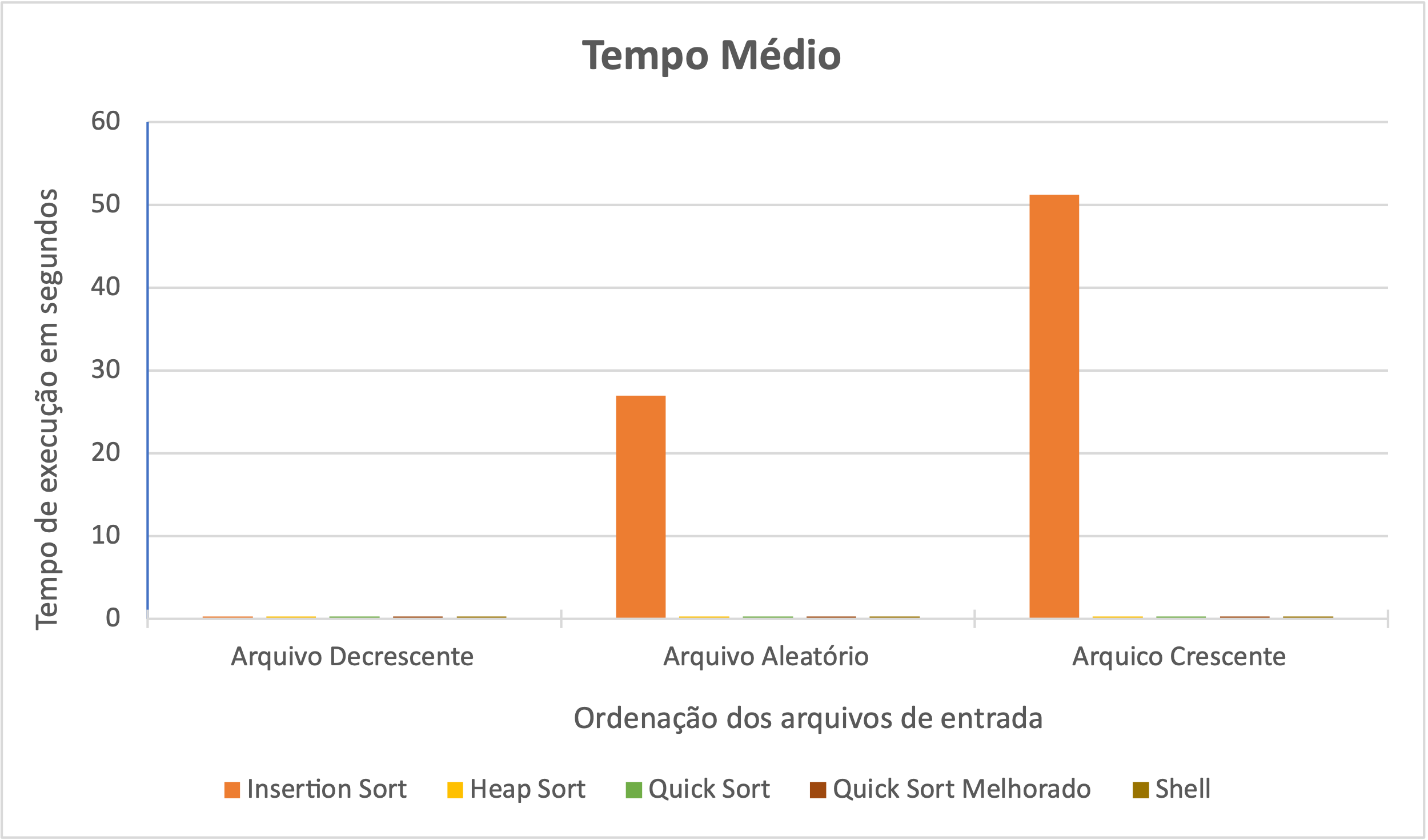


1. **Foi calculado também o tempo médio de execução dos 5 métodos para os três arquivos:**

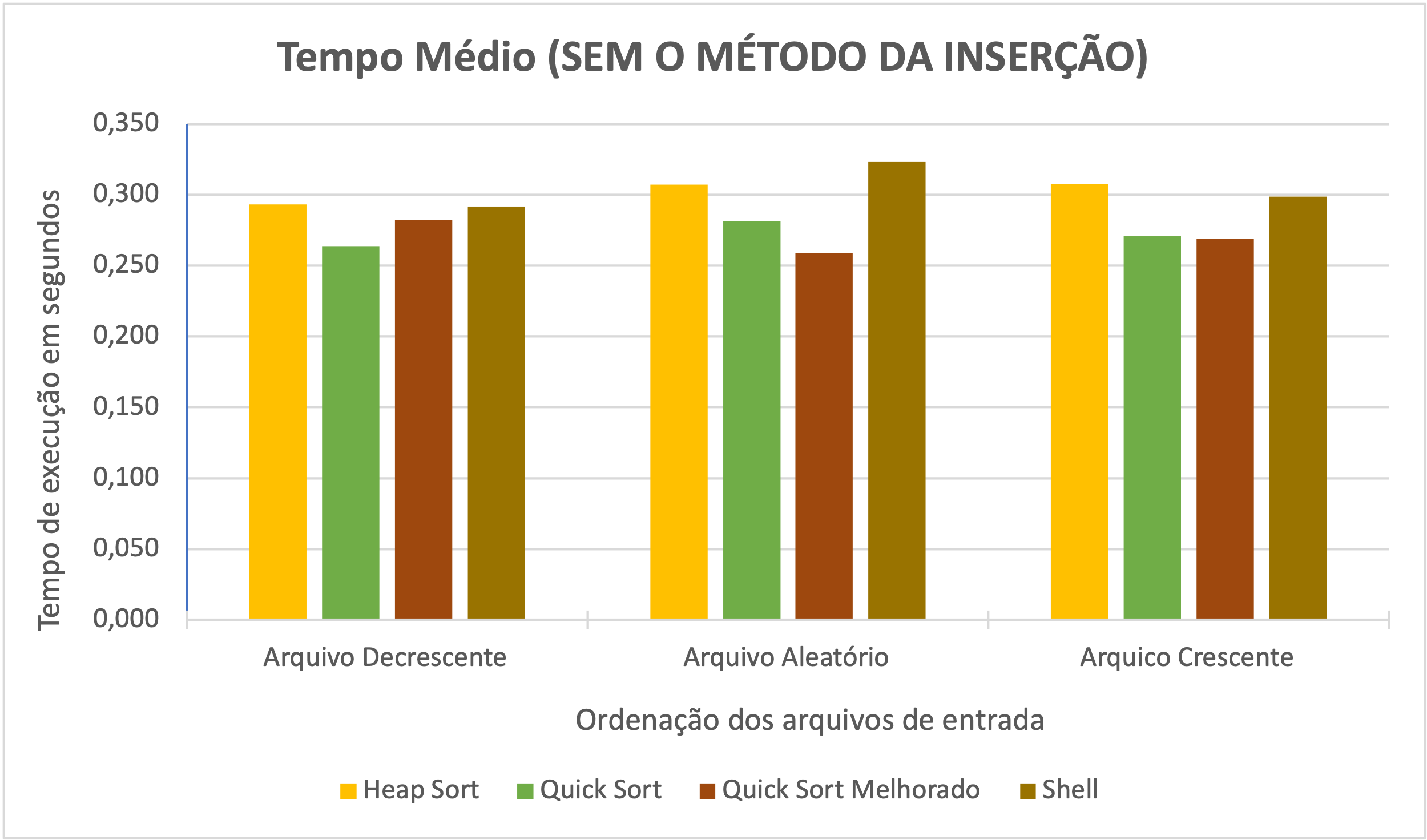
Table

Description automatically generated

Considerando método da inserção:



Não considerando o método da inserção:



1. **Conclusão**

Esse trabalho lidou com o problema de ordenar arquivos utilizando os métodos de ordenação aprendidos na disciplina. Durante a implementação da solução para o problema, houveram importantes desafios a serem superados, por exemplo o desenvolvimento correto de todos os métodos e o esforço de fazer a análise medindo o tempo um por um.

**Referências**

1. Chaimowicz, L. and Prates, R. (2021). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciencia da Computação. Universidade Federal de Minas Gerais. Belo Horizonte
2. <https://pt.wikipedia.org/wiki/Shell_sort>
3. <https://www.hostinger.com.br/tutoriais/comando-time-linux>
4. <http://www.cplusplus.com/reference/fstream/fstream/>
5. <http://www.cplusplus.com/reference/string/>