



**Curso de Ciência da Computação  
UNIVERSIDADE PAULISTA**

**Guilherme Augusto Sbizero Correa RA:F235289**

**Maria Fernanda Mamani Huarcasi RA:N656999**

**Moisés da Silva Freitas RA:F1610J2**

**DESENVOLVIMENTO DE SISTEMA PARA ANÁLISE DE PERFORMANCE DE  
ALGORITMOS DE ORDENAÇÃO DE DADOS**

**SÃO PAULO – SP**

**2021**

## **Sumário**

<b>1. Introdução .....</b>	<b>1</b>
<b>2. Objetivo do trabalho .....</b>	<b>2</b>
<b>3. Algoritmo de ordenação.....</b>	<b>3</b>
3.1 Bubble Sort.....	3
3.2 Selection Sort.....	4
3.3 Insertion Sort .....	5
3.4 Quick Sort.....	6
<b>4. Desenvolvimento .....</b>	<b>8</b>
<b>5. Resultados e Discussão.....</b>	<b>13</b>
<b>6. Considerações Finais.....</b>	<b>17</b>
<b>7. Bibliografia.....</b>	<b>19</b>
<b>8. Código Fonte.....</b>	<b>20</b>
<b>9. FICHA DE ATIVIDADES PRÁTICAS SUPERVISIONADAS .....</b>	<b>23</b>

## 1. Introdução

Com os avanços tecnológicos, o mundo passou a usar mais dados digitais do que físicos e com isso, foi necessário criar métodos de ordenação, busca, implementação entre outras coisas que agilizam a organização e o uso dos dados virtuais.

Com essas necessidades, foram desenvolvidos incontáveis algoritmos de ordenação, como o Bubble Sort, Selection Sort, Insertion Sort, Quick Sort e etc. Também existe uma variação desses métodos, onde um pode fazer ordenação mais rápida em determinadas ocasiões e pode fazer ordenação de caracteres também; várias possibilidades, que agilizam cada processo de organização de dados e estamos em uma era que os dados estão se tornando bigdatas (arquivos enormes) que envolve inteligência artificial, dados de uma empresa gigante etc. E com isso, métodos de ordenação se torna uma das coisas mais importantes, porque sem ele, a organização de dados e busca do mesmo se torna extremamente difícil e trabalhoso.

Existem várias razões para se ordenar uma sequência, uma delas é a possibilidade de acessar seus dados com mais eficiência, tornando a busca possível e mais rápida, cada método de ordenação tem suas vantagens e desvantagens, como por exemplo o Quick Sort que é um método poderoso e extremamente rápido para ordenação de vetores grandes, já em contra partida, ele não é totalmente estável; já o Bubble Sort, tem como sua principal vantagem a simplicidade de entendimento e uma facilidade para implementar o algoritmo, mas tem um número muito grande de movimentações de elementos, ou seja, não é uma boa ideia utilizar esse método em ordenação complexa.

## **2. Objetivo do trabalho**

O objetivo deste trabalho é trazer um programa totalmente feito com a linguagem C, onde este programa terá como objetivo fazer ordenações de vetores com o método de sorteamento escolhido pelo usuário. Ao fim ele mostrará o tempo de demora de cada sorteamento, dando assim a base para a análise de estruturas.

Junto há a análise teórica de cada método de sorteamento, falando de como funciona a análise lógica deles e como funciona matematicamente. No fim o usuário terá a conclusão de por que certos métodos são mais demorados e outros mais rápidos, tudo baseado na lógica de como cada um tem.

### 3. Algoritmo de ordenação

Algoritmo de ordenação na ciência da computação tem o significado de um algoritmo (programa) que organiza os elementos em uma ordem que foi colocada no algoritmo pelo programador, as mais utilizadas são: ordens crescentes e decrescentes.

O objetivo da ordenação é facilitar buscas, recuperação, edição, entre outras coisas. Deixando os dados organizados e com operações eficientes.

Existem diversos algoritmos de ordenação, umas mais rápidas e outras mais lentas, ambas têm vantagens e desvantagens...

#### 3.1 Bubble Sort

Bubble sort é um algoritmo de ordenação simples, sua principal maneira de ordenar é pegando o primeiro valor e testar com o segundo para ver qual é o maior, e fazendo teste lógico em cada posição do vetor, visando trocar o maior número com o menor caso ele esteja fora de posição, com o objetivo de colocar o maior número na última posição possível. Depois desse teste, ele vai fazer a próxima interação, onde ele vai procurar o segundo maior número e colocar na penúltima ou na segunda posição (caso seja decrescente) e vai fazendo mais interações até ordenar todos os valores do vetor.

A quantidade de interações é a quantidade de valores dentro de um vetor  $-1$  ( $N-1$ ), ou seja, esse vetor do exemplo a seguir que contém 5 valores, terá 4 interações.

Neste exemplo, o vermelho representa a comparação dos números (Exemplo, o número 35 é maior que 55? Se sim, troca. Senão, não troca e passa para o número seguinte), já o azul, representa a cor dos números ordenados.

Tabela 1 - Simulação do método de trocas do Bubble Sort

Primeira interação					Segunda interação					Terceira interação				
0	1	2	3	4	0	1	2	3	4	0	1	2	3	4
35	55	12	21	4	35	12	21	4	55	12	21	4	35	55
35	55	12	21	4	12	35	21	4	55	12	21	4	35	55
35	12	55	21	4	12	21	35	4	55	12	4	21	35	55
35	12	21	55	4	12	21	4	35	55	12	4	21	35	55
35	12	21	4	55	12	21	4	35	55	12	4	21	35	55

Quarta interação				
0	1	2	3	4
12	4	21	35	55
4	12	21	35	55
4	12	21	35	55
4	12	21	35	55
4	12	21	35	55

Fonte: Guilherme Augusto, 2021.

### 3.2 Selection Sort

O Selection Sort é baseado em se passar sempre o menor valor do vetor para a primeira posição (ou o maior, dependendo da lógica escrita pelo programador), depois o segundo menor valor para a segunda posição, o terceiro menor valor para a terceira posição, usando a fórmula  $n-1$  em elementos restantes até os últimos dois elementos restantes.

O método de ordenação contém dois laços de repetição, onde um é chamado de interno e outro de externo, o interno percorre todo o vetor para achar o menor valor, enquanto o externo continua na posição 0 até que o laço interno compare todos os vetores e armazena em uma variável o menor número achado e troca se achar um menor que ele mesmo, depois disso o laço externo avança uma casa e o interno também, ignorando o vetor anterior e procurando o próximo menor valor que contém no vetor.

Nessa tabela, o roxo representa a variável do menor número, onde o primeiro número menor que a casa do vetor, ele armazena e compara as próximas casa do vetor com o número armazenado nessa variável e caso ache, ele troca o menor número e depois troca com o laço de repetição externo, ordenando o número e indo para a próxima casa do vetor. O vermelho representa a comparação inicial e o azul escuro representa o vetor ordenado.

Tabela 2 - Simulação do método de trocas do Selection Sort

Primeira interação					Segunda interação					Terceira interação				
0	1	2	3	4	0	1	2	3	4	0	1	2	3	4
35	55	12	21	4	4	55	12	21	35	4	12	55	21	35
35	55	12	21	4	4	55	12	21	35	4	12	55	21	35
35	55	12	21	4	4	55	12	21	35	4	12	21	55	35
35	55	12	21	4	4	12	55	21	35					
4	55	12	21	35										

Quarta interação				
0	1	2	3	4
4	12	21	55	35
4	12	21	35	55

Fonte: Guilherme Augusto, 2021.

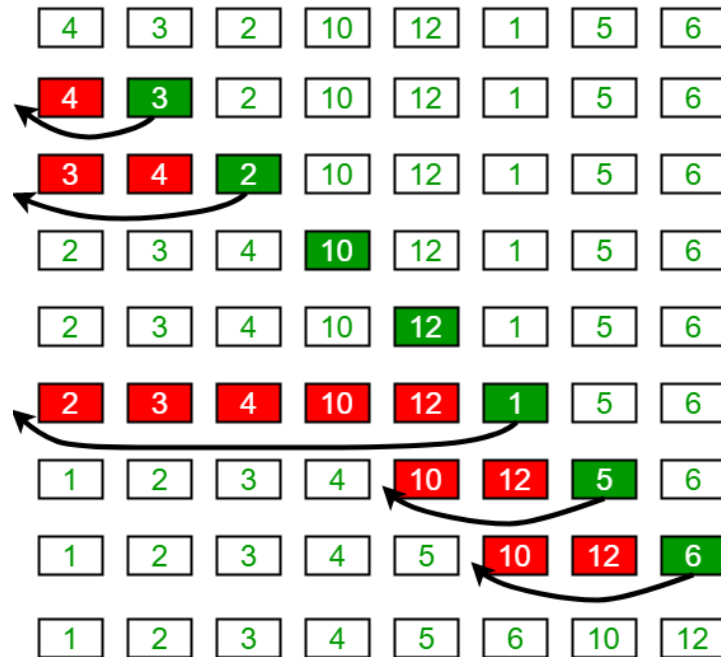
### 3.3 Insertion Sort

O Insertion Sort é um algoritmo de ordenação que, dando certa estrutura ele constrói uma matriz com elementos de cada vez, uma inserção por vez, além de ser bem eficiente é usado geralmente para problemas com pequenas entradas.

Um exemplo de insertion sort é que temos 6 pessoas cada uma segurando uma plaquinha com algum número e estão desordenado, o insertion irá checar de 2 em 2 e sempre quando o valor da direita for menor que o da esquerda ele irá trocar os valores os ordenando de forma crescente e se for preciso os números checados anteriores que não foram trocados ele irá repetir o processo e checa de novo se é preciso troca os valores que mudaram de posição.

Esta é uma ideia por trás da ordenação por inserção, que percorre as posições do array, começando do índice 1, a cada nova posição você precisa inseri-lo no lugar correto no subway ordenado à esquerda daquela posição.

Tabela 3 - Simulação do método de trocas do Insertion Sort



Fonte: GeeksforGeeks, 2016.

### 3.4 Quick Sort

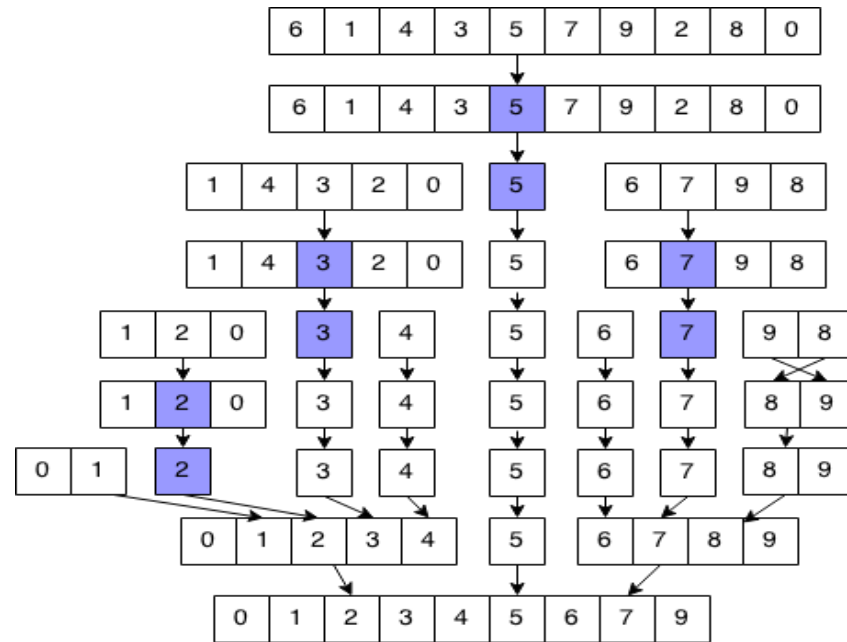
Método de ordenação rápido e eficiente criado em 1960 por C.A.R Hoare, após tentar traduzir um dicionário de inglês para russo ordenando palavras.

O quicksort adota a estratégia de divisão e conquista, consiste em organizar as chaves de forma que as chaves menores precedam as chaves maiores logo em seguida é ordenado as sublistas de chaves menores e maiores até que se encontre ordenado.

Simplesmente ele escolhe um elemento da lista que será o pivô e o marca e testa todos até que todos anteriores ao marcado sejam menores e todos os elementos posteriores sejam maiores, repetindo esse processo até que todos os elementos estejam ordenados.



Tabela 4 - Simulação do método de trocas do Quick Sort



Fonte: CS Handbook, 2016.

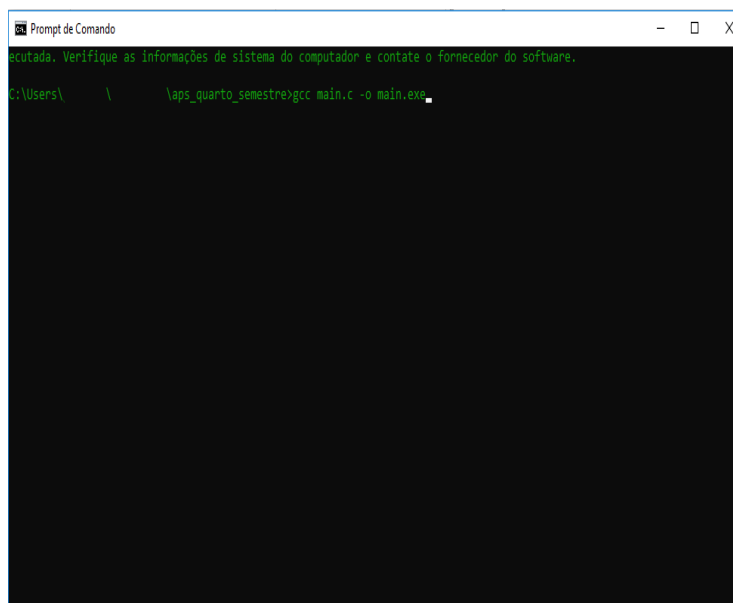
Com o elemento pivô selecionado ele irá iniciar a trocas onde as anteriores ao pivô sejam menores e todos os elementos posteriores ao pivô sejam maiores.

#### 4. Desenvolvimento

O desenvolvimento do programa foi primeiramente baseado em desenvolver os métodos de ordenação que iriam ser aplicados. Em princípio seriam apenas três métodos, porém, com o avanço do curso, foi-se aplicado a utilização do Quicksort, pelo fato de apresentar método de recursividade. Além de aplicar essa funcionalidade extra, também a ideia do projeto é desenvolver um programa em C que possa ser executado em todos os sistemas operacionais (distribuições Linux, Windows e MacOS), assim trazendo bibliotecas que pudessem funcionar 100% em cada um.

Algumas partes do programa foram tanto compiladas e executadas pela IDE DevC++ como também foi feito testes no editor de texto Visual Studio Code, isso ficou baseado na preferência de cada um do grupo. Porém todos os executáveis foram criados através do famoso compilador GCC (GNU Compiler Collection), aplicado através de uma compilação via terminal.

Imagem 1 – Terminal CMD



Fonte: Guilherme Augusto – 2021.

Também, para melhor rapidez de inserção de informações tanto no código do programa como também na parte escrita, este trabalho contou com a utilização do Git e Github para que o repositório sempre tivesse atualizado em qualquer alteração, assim todos do projeto ficariam cientes de quem fez o que nele. Também será mantido o projeto neste repositório para que qualquer pessoa possa ter acesso ao código e a parte escrita dele sempre sujeito aos direitos autorais.

Com o projeto organizado, com cada pasta e arquivos desenvolvidos, foi criado o arquivo main.c, onde foi aplicada todo o código. Na montagem do programa houve sempre a ideia de rebuscar ao máximo o que a linguagem C tem há proporcionar. Então para cada método de sorteamento seria feito através de uma struct (estrutura), com quatro vetores que iriam passar por cada método de sorteamento. Este struct foi alterado seu tipo primitivo apenas para ser chamado de Vetores.

Após a criação da struct Vetores, foram criados os métodos de inserção de valores dos vetores. Esse método possui um laço de repetição para inserção de valores baseado no tamanho máximo dos vetores, a variável "TAM". Nisso, para que haja todos os valores possíveis de "TAM" (que são 20000), a função que insere valores aleatórios e a função rand. Para que pudesse inserir valores de 0 a 20000 era preciso chamar a função rand com a porcentagem (resto) de "TAM". Sendo que dentro do laço, esse valor aleatório era espelhado em cada um dos vetores. Por fim, se retornava o struct Vetores com os valores inseridos.

Após a inserir valores nos vetores, foi criado todos os métodos de sorteamento. O Bubblesort, Insertionsort, Selectionsort e Quicksort. Eles são totalmente a peça-chave do programa para a execução. Cada um desses métodos irá receber futuramente um dos vetores da struct Vetores na função "OpcaoSort", sendo que em todos os métodos de ordenação, recebe-se um ponteiro de tipo inteiro para esses vetores. Como C é uma linguagem de médio nível e sabe mexer bem com a memória do computador, passar um vetor de tamanho N para um ponteiro é apenas passar uma grande "fita" da sua memória para o ponteiro.

O Bubblesort apresenta uma lógica de dois laços de repetição, onde o laço inferior (o que está dentro do laço superior), sempre pegara a posição do laço superior e somar mais um (se o laço superior estiver na posição 0, o inferior estará na posição 1), onde sempre verifica se caso a posição do vetor do laço superior for maior que a posição do vetor no laço inferior haverá uma troca. Isso é um processo demorado pois ele tem que passar pelo laço inferior várias vezes.

Já o Selectionsort apresenta uma lógica também com um laço superior e inferior, onde segue também a mesma regra que o Bubblesort que o laço inferior soma mais um em relação ao laço superior. Porém, ele apresenta uma variável chamada "min" que pega a posição do vetor na interação do laço superior. Quando ele entra no laço inferior, há a verificação se o a posição do vetor em "min" é maior

que a posição do vetor no índice do laço inferior. Se isso ocorrer, o “min” pega o índice do laço inferior. Em seguida, no laço superior ocorrem as trocas, assim o método não precisa ficar fazendo várias trocas igual ao Bubblesort, apenas tendo uma troca por cada índice do laço superior.

Agora, com o método Insertionsort há uma mudança relativa aos outros códigos, onde há também um laço superior e um inferior, porém, o inferior apresenta já mais de uma condicional (tanto o Bubble como o Selection usavam laços de repetição for que apenas há uma condicional se o índice delas for menor que tal valor esse índice deve ser incrementado mais uma vez). No método há uma variável chave, onde ela apenas pega o valor do vetor no índice do laço superior. Logo em seguida a variável “j” (o índice do laço inferior) recebe o valor do índice do laço superior decrementado por um. Dentro do laço inferior há a verificação se o índice “j” é maior que zero e se “j” é maior que zero e se o vetor na posição “j” é maior que a chave. Se as condições forem atendidas no inferior, o vetor na posição “j” incrementado com mais um recebe o vetor na posição “j”, em seguida “j” é decrementado por um. Por fim, ao sair do laço inferior, o vetor na posição “j” incrementado com mais um recebe a chave.

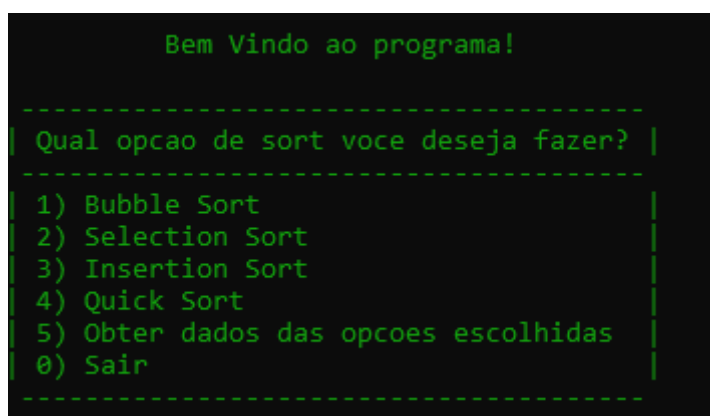
Agora o último método de sorteamento, o Quicksort, é com certeza o mais diferente dos outros. Ele precisa da posição primaria do vetor (sempre em todos os casos 0) e a última. Após saber isso há os índices “i” e “j” onde “i” recebe o início e “j” o fim, e uma variável que recebe o valor do meio do vetor (já sabendo o início e o fim, é apenas necessário pegar a soma deles e dividir por dois). Após isso há um laço que verifica se “i” é inferior ou igual a “j”, e dentro dele há dois laços inferiores onde o primeiro é se o valor do vetor na posição “i” é inferior a valor da posição do meio do vetor e se “i” é menor que o fim, caso ele entre nesse laço, “i” é incrementado por mais um. Já no segundo laço, ele verifica se o valor vetor na posição “j” é superior a posição do vetor do meio e se “j” é maior que o início, onde caso passe pelas condições, “j” é decrementado por um. Após esses dois laços há uma condicional que vê se “i” é inferior ou igual a “j”, onde se atender há as trocas do vetor e “i” será incrementado por um e “j” decrementado por um. Após a saída do laço superior há duas condicionais, onde na primeira se verifica se “j” é superior ao início. Atendendo há o grande macete do Quicksort, onde há a chamada do próprio método, onde ele passa o nos parâmetros o vetor, a posição início novamente e a posição final será “j” incrementado. Agora a segunda condicional verifica se “i” é

inferior ao fim, onde passando pela condicional, ocorre também novamente uma chamada do próprio método com os parâmetros do vetor, “i” sendo a variável inicial e o fim se mantendo novamente como a posição final.

A função “MostrarTela” para mostrar que o vetor foi corretamente preenchido. A função “Interface” para deixar a parte front-end mais user-friendly, e por final, a função “OpcaoSort” que é basicamente o core do programa. Ele irá servir tanto para pegar os resultados de desempenho entre as ordenações, assim como a execução de cada função que foi criada no projeto. Se fosse Orientação a objetos, poderia-se chamá-la de classe pai basicamente, já que é ela que está gerenciando os ponteiros e as funções do programa. Por final, temos as funções de ordenação e a função de busca binária, no qual é o foco principal do trabalho. Durante o desenvolvimento foi adicionado um código um pouco “diferente” como experimento. O nosso convidado ilustre é o código espaguete, utilizado na linha 210 e 285 da fonte do código. Ele foi utilizado para ter um aumento de desempenho. Pois graças a ele, não houve necessidade de criar um método, ou repetir a mesma string várias vezes para apenas dizer que a opção já tinha sido escolhida.

Com os métodos organizados, há a compilação do programa via terminal. Esse programa executável é tanto em exe para a plataforma Windows como out para Linux e MacOS.

Imagem 2 – Interface do programa

A screenshot of a terminal window with a black background and green text. At the top, it says "Bem Vindo ao programa!". Below that, a dashed line separates the header from the menu. The menu is titled "Qual opcao de sort voce deseja fazer?" and is enclosed in a box made of dashed lines. The options listed are: 1) Bubble Sort, 2) Selection Sort, 3) Insertion Sort, 4) Quick Sort, 5) Obter dados das opcoes escolhidas, and 0) Sair.

```
Bem Vindo ao programa!  
-----  
| Qual opcao de sort voce deseja fazer? |  
-----  
| 1) Bubble Sort                        |  
| 2) Selection Sort                    |  
| 3) Insertion Sort                   |  
| 4) Quick Sort                      |  
| 5) Obter dados das opcoes escolhidas |  
| 0) Sair                            |  
-----
```

Fonte: Guilherme Augusto – 2021.

Ao executar, aparece-se a interface de interação para o usuário, onde nela se pode escolher cada um dos métodos de sorteamento. Onde ao escolher há todo o preenchimento da tela com os números numa grande velocidade sendo incapaz do usuário saber quais deles foram sorteados aleatoriamente. Após isso, a tela é limpa e aparece uma pergunta “Entre com o inteiro a ser pesquisado”, onde o usuário deve inserir um número aleatório que possa estar no vetor.

Após a inserção que pode ou não achar o número da busca, volta-se a interface do programa com as outras funções de sorteamento, porém o que foi escolhido anteriormente não pode ser executado novamente. E esse ciclo ocorre até o usuário resolver sair. Quando houver a execução de todos os métodos de sorteamento apenas restará as opções de obter os resultados dos sorteamentos e sair, que apenas mostra em quantos milissegundo demorou cada tipo de sorteamento.

Com o resultado dá para se analisar que cada método de sorteamento tem um tempo de demora decorrente a estrutura de seu algoritmo, e em quase todos os todas as execuções, o método Quicksort se mostra sempre o mais ágil e rápido por seu modo de separar os problemas em partes baseados na recursividade.

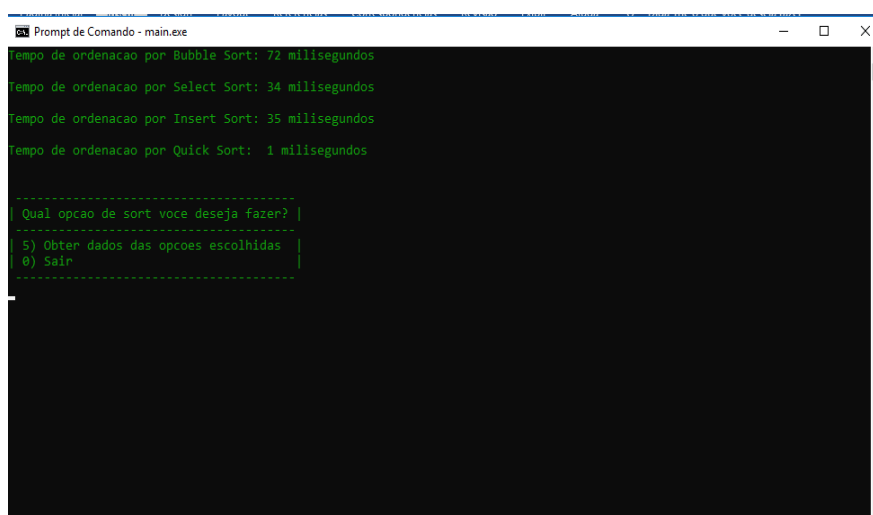
Como dito, cada método de sorteamento tem seu algoritmo próprio baseado em uma troca de valores de um dos vetores da estrutura Vetores, pois como cada um dos vetores teve valores aleatórios inseridos em uma ordem diferente, não se pode saber qual número estará em cada posição. Assim cada um dos algoritmos, baseado em sua lógica, atuarão de maneiras diferentes de análise em suas posições sempre verificando baseado em sua maior parte baseado em dois ou três laços de repetições.

## 5. Resultados e Discussão

Com o desenvolvimento do programa, foram feitos testes de quantos milissegundos cada um dos métodos tem em diferentes tamanhos de vetores. Assim foram usados tamanhos de cinco mil, dez mil, quinze mil, vinte mil para os vetores para que se pudesse fazer uma análise de cada método.

Com cinco mil foi obtido os resultados de 72 milissegundos para o Bubblesort, 34 milissegundos para o Selectionsort, 35 segundos para o Insertionsort e 1 milissegundo para o Quicksort. Mas ao rodar novamente o programa, o Insertionsort abaixou para 18 milissegundos, podendo ter essa variação decorrente ao quando executar o programa.

Imagem 2 – Resultados dos métodos com 5000 posições



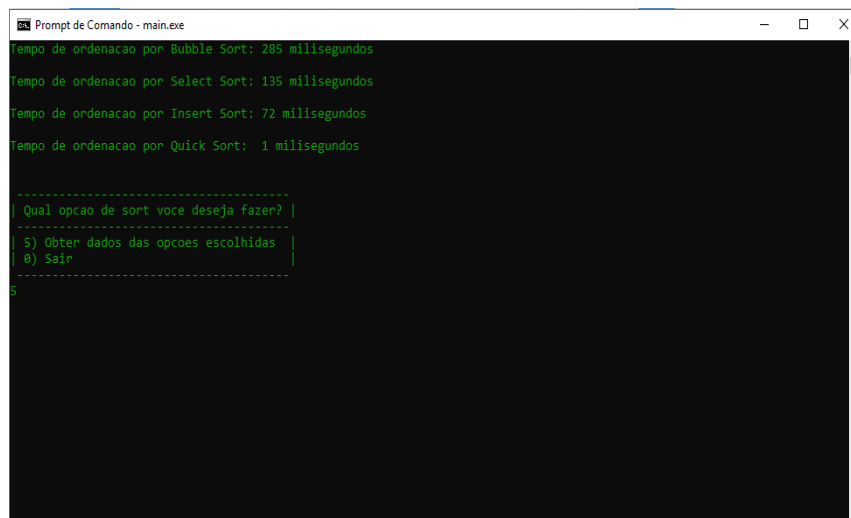
```
Prompt de Comando - main.exe
Tempo de ordenacao por Bubble Sort: 72 milissegundos
Tempo de ordenacao por Select Sort: 34 milissegundos
Tempo de ordenacao por Insert Sort: 35 milissegundos
Tempo de ordenacao por Quick Sort: 1 milissegundos

Qual opcao de sort voce deseja fazer?
5) Obter dados das opcoes escolhidas
0) Sair
```

Fonte: Guilherme Augusto– 2021.

Já com o uso de dez mil, houve já uma mudança enorme de performance em cada um, onde o Bubblesort quase triplicou-o com o resultado de cinco mil, onde com dez mil já foi para quase 282 milissegundos. Já o Selectionsort quadruplicou seu valor com cinco mil e ficou com 135 milissegundos. O Insertionsort deu uma duplicada e ficou com 72 milissegundos e o Quicksort se manteve novamente com apenas 1 milissegundo. No segundo teste feito com o mesmo valor, a única alteração foi com o Bubblesort que aumentou apenas 3 milissegundos, ficando com 185 milissegundos.

Imagem 3 – Resultados dos métodos com 10000 posições



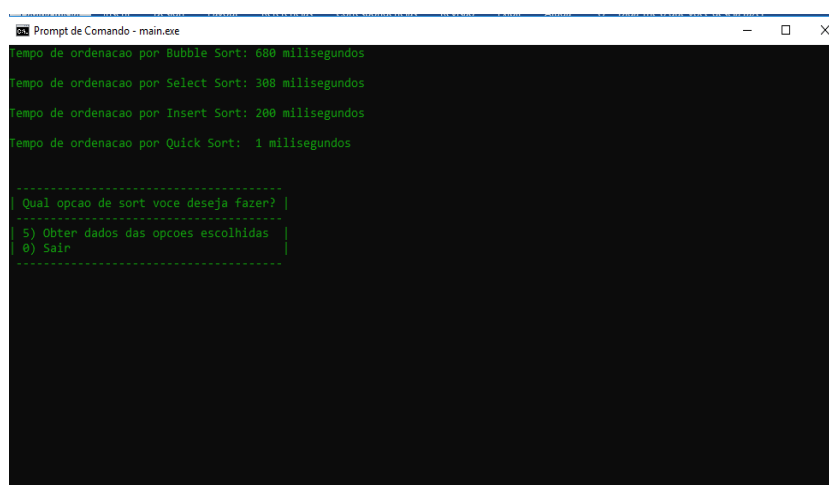
```
Prompt de Comando - main.exe
Tempo de ordenacao por Bubble Sort: 285 milissegundos
Tempo de ordenacao por Select Sort: 135 milissegundos
Tempo de ordenacao por Insert Sort: 72 milissegundos
Tempo de ordenacao por Quick Sort: 1 milissegundos

-----
| Qual opcao de sort voce deseja fazer? |
-----
| 5) Obter dados das opcoes escolhidas |
| 0) Sair                               |
-----
5
```

Fonte: Guilherme Augusto- 2021.

Com o teste de quinze mil o Bubblesort teve o resultado de 671 milissegundos, onde triplicou novamente em relação ao teste anterior. O Selectionsort ficou com o valor de 308 milissegundos, o Insertionsort ficou com 180 milissegundos e o Quicksort agora mudou e dobrou para 2 milissegundos. Agora quando houve a segunda execução do programa, o Bubblesort saiu de 671 para 680 milissegundos, o Selectionsort se manteve o mesmo, o Insertionsort saiu de 180 e foi para 200 milissegundos e o Quicksort se manteve igual.

Imagem 4 – Resultados dos métodos com 15000 posições



```
Prompt de Comando - main.exe
Tempo de ordenacao por Bubble Sort: 680 milissegundos
Tempo de ordenacao por Select Sort: 308 milissegundos
Tempo de ordenacao por Insert Sort: 200 milissegundos
Tempo de ordenacao por Quick Sort: 1 milissegundos

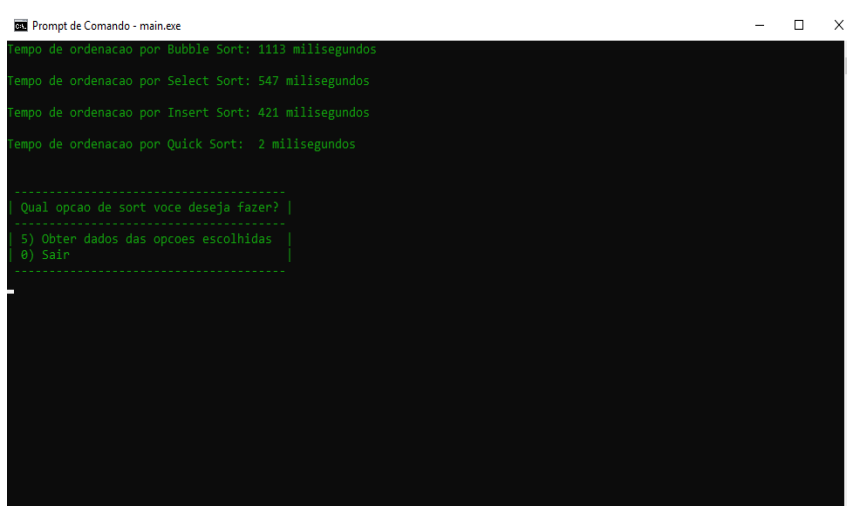
-----
| Qual opcao de sort voce deseja fazer? |
-----
| 5) Obter dados das opcoes escolhidas |
| 0) Sair                               |
-----
```

Fonte: Guilherme Augusto– 2021.



Com o teste de vinte mil posições deu para se perceber uma pequena lentidão do programa em relação aos testes anteriores, principalmente com o Bubblesort. Neste teste o Bubblesort apresentou a mudança de 671 para 1141 milissegundos, o Selectionsort foi de 308 para 553 milissegundos, o Insertionsort foi de 180 para 366 milissegundos e o Quicksort agora foi de 2 para 3 milissegundos. Agora com a segunda execução houve invés de um aumento de tempo para cada vetor, houve uma diminuição, onde o Bubblesort agora ficou com 1113 milissegundos, o Selectionsort ficou com 547 milissegundos, o Insertion ficou com 421 milissegundos e o Quicksort ficou com 2 milissegundos.

Imagem 5 – Resultado dos métodos com 20000 posições



```
Prompt de Comando - main.exe
Tempo de ordenacao por Bubble Sort: 1113 milissegundos
Tempo de ordenacao por Select Sort: 547 milissegundos
Tempo de ordenacao por Insert Sort: 421 milissegundos
Tempo de ordenacao por Quick Sort: 2 milissegundos

Qual opcao de sort voce deseja fazer?
5) Obter dados das opcoes escolhidas
0) Sair
```

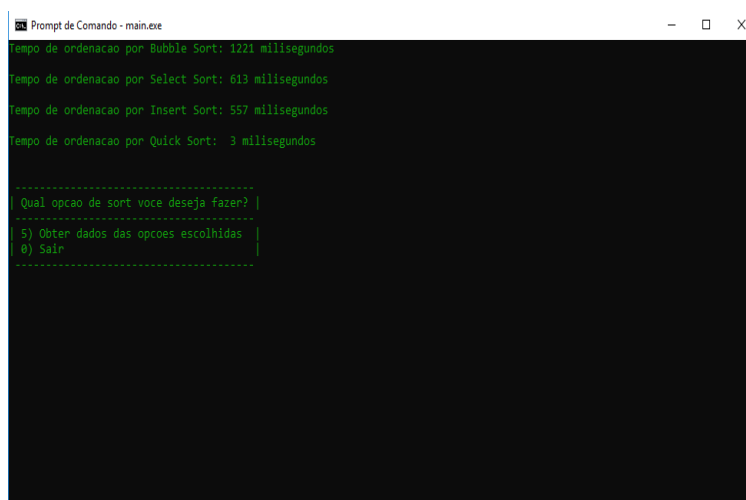
Fonte: Guilherme Augusto – 2021.

Olhando todos os resultados deu para perceber de totalmente a performance de cada métodos. Onde o Quicksort se demonstrou sempre o mais rápido e eficiente, onde seu modo de recursão tornam o método não só uma análise de posições como uma separação de problemas para se saber como ordenar os vetores.

Agora o Bubblesort por sempre fazer várias análises no vetor, faz com que isso degaste tanto do processador quanto da memória, fazendo assim o método ser totalmente lento. Por isso deve-se evitar o uso de várias interações de vetores em laços de repetição.

Os resultados que foram feitos mostram que em quase todos os casos o método Quicksort supera todos os outros por seu modo de separar o problema em partes. Isso garante que o programa não faça várias verificações e mudando toda hora as posições do vetor, fazendo com que com apenas algumas condicionais e chamadas recursivas o método não houvesse toda hora uma nova troca no método, analisando apenas os índices que deveriam ser trocados.

Imagem x.x – Resultados dos métodos



```
Prompt de Comando - main.exe
Tempo de ordenacao por Bubble Sort: 1221 milisegundos
Tempo de ordenacao por Select Sort: 613 milisegundos
Tempo de ordenacao por Insert Sort: 557 milisegundos
Tempo de ordenacao por Quick Sort: 3 milisegundos

-----
Qual opcao de sort voce deseja fazer? |
-----
5) Obter dados das opcoes escolhidas |
0) Sair |
-----
```

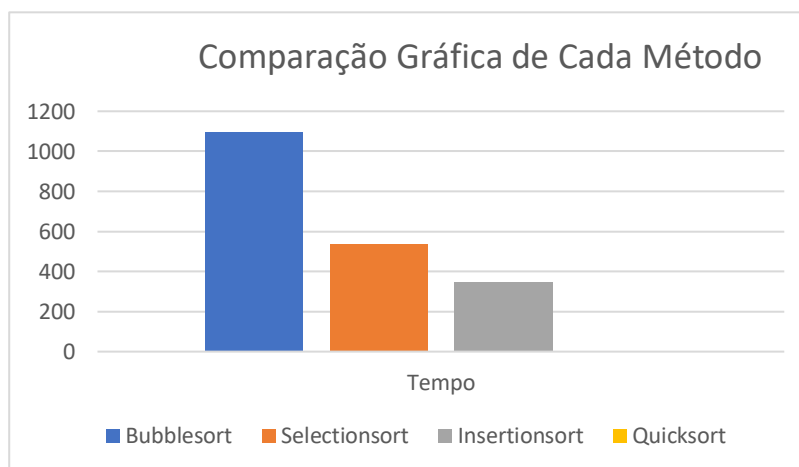
Fonte: Guilherme Augusto – 2021.

Analisando os códigos de sorteamento e a suas lógicas, deu para se perceber que aqueles que há um laço superior e inferior fazendo várias verificações se tornam os mais demorados. Ficar fazendo num laço inferior várias verificações e lá fazer as trocas causa uma demora de performance.

Porém, dentro desses, o Bubblesort se tornou o mais lento por toda hora que há a entrada na condicional do laço inferior há várias trocas e desnecessárias já que ele precisa percorrer todas as posições do vetor e verificar se a posição do vetor no índice do laço inferior é menor que a posição do laço superior. Isso é algo demorado e faz muito processamento de memória quanto de CPU.

## 6. Considerações Finais

Imagem: Gráfico de Cada Método



Autor: Guilherme Augusto – 2021.

Ao observar os gráficos é quase impossível ver o tempo do Quicksort em milissegundos comparado aos demais métodos. Onde mesmo que mais rápidos que o Bubblesort, o Insertionsort e Selectionsort ainda tem bastante uso de memória e processamento para fazer os sorteamentos e alinhar todo o vetor. Mostrando que aqueles que tiveram um laço superior e um inferior, onde dentro do inferior há uma análise baseada no índice do laço superior há uma perda de performance pois aí que ocorre a constante análise de posições e trocas do vetor;

Vendo esses algoritmos conseguimos perceber como a lógica de programação aplicada em cada um dos métodos é algo essencial para a performance do programa. Isso é muito devido ao uso da memória RAM do computador que armazenando variáveis em seus espaços e com o processador executando as operações dos algoritmos. Vemos que quando há o uso de poucos processos de laços de repetições e ainda separar os processos a performance se torna mais rápida. Já, quando há muito o uso do laço de repetição, o programa ficará abusando da memória com suas várias trocas desnecessárias e muito processo da CPU.

Como a linguagem C é uma linguagem de médio nível e sua criação foi muito baseada no uso de memória (devido a criação do sistema UNIX), saber criar algoritmos tanto de sorteamento, busca ou entre outros exerce que os programadores sejam capazes de elaborar soluções com a linguagem e não desgastem demais o computador.

Também foi possível analisar que laços de repetição causam a maior parte dos problemas de performance do programa. Já que eles funcionam de uma maneira que analisa posição por posição, e que laços com laços internos (laços inferiores) causam mais análise ainda e é baseado no percorrer do laço superior, isso causa lentidão do programa e deve ser evitado esse tipo de processo.

Concluindo, vimos que cada método apresenta uma vantagem em relação ao outro baseado no meio de ordenar o vetor e no tempo que ele faz as trocas. Bubblesort se apresentou como o método mais demorado por seu excesso de trocas e já o Quicksort se demonstrou o mais rápido por saber dividir em partes o problema a ser resolvido e não abusar dos laços de repetição.

## 7. Bibliografia

LINGUAGEM C PROGRAMAÇÃO DESCOMPLICADA.  
<https://www.youtube.com/user/progdescomplicada/>. **Youtube**, 2014. Disponível em: [https://www.youtube.com/watch?v=qU8N\\_bmebQ4](https://www.youtube.com/watch?v=qU8N_bmebQ4)>. Acesso em: 14 Novembro 2018.

LINGUAGEM C PROGRAMAÇÃO DESCOMPLICADA.  
<https://www.youtube.com/user/progdescomplicada/playlists>. **Youtube**, 2014. Disponível em: [https://www.youtube.com/watch?v=qU8N\\_bmebQ4](https://www.youtube.com/watch?v=qU8N_bmebQ4)>. Acesso em: 15 Novembro 2018.

ISIDRO, P. <https://www.youtube.com/user/fmassetto/playlists>. **Youtube**, 2017. Disponível em: <https://www.youtube.com/watch?v=KiZ1vT-tEtU>>. Acesso em: 22 Outubro 2018.

YOUNG, M. Quick Sort. **Thecshandbook**, 201? Disponível em: [http://www.thecshandbook.com/Quick\\_Sort](http://www.thecshandbook.com/Quick_Sort)>. Acesso em: 23 Outubro 2018.

CHITRANAYAL, P. Insertion-Sort. **Geeksforgeeks**, 201? Disponível em: <https://www.geeksforgeeks.org/insertion-sort/>>. Acesso em: 24 Outubro 2018.

## 8. Código Fonte

```
#include <stdio.h> #include <stdlib.h> #include <string.h> #include <time.h>
#include <stdbool.h> //Definir tamanho dos vetores #define tam 20000 //apontadores
para os laços de repetição int i, j; bool a, b, c, d; //Verificadores para saber se a
opção já foi escolhida typedef struct { //Vetores para guardar valores aleatórios e
logo então, serem sortados int vetor1[tam]; int vetor2[tam]; int vetor3[tam]; int
vetor4[tam]; //Guarda informação do tempo de duração do método int bubble; int
select; int insert; int quick; } Vetores; /*Preenche vetores aleatoriamente e faz um
espelho dos vetores restantes para obter maior precisão ao comparar entre os
métodos*/ Vetores preencher(Vetores v) { for (i = 0; i < tam; i++) { v.vetor1[i] = rand()
% tam; v.vetor2[i] = v.vetor1[i]; v.vetor3[i] = v.vetor1[i]; v.vetor4[i] = v.vetor1[i]; } return
v; } //Mostra no console todos os valores dentro do vetor desejado void
mostrarTela(int *vetor, int numeroVetor) { printf("\n"); for (i = 0; i < tam; i++) {
printf("%d \t", vetor[i]); } printf("\n"); printf("VETOR %d \n", numeroVetor);
system("cls"); } //Faz uma busca binária para achar um determinado valor dentro do
vetor desejado void buscaBinaria(int *vetor) { int achou = 0, inicio = 0, fim = tam - 1,
meio, busca; printf("\nEntre com o inteiro a ser pesquisado: "); scanf("%d", &busca);
while (inicio <= fim) { meio = (inicio + fim) / 2; if (vetor[meio] == busca) achou = 1; if
(busca < vetor[meio]) fim = meio - 1; else inicio = meio + 1; } system("cls"); if (achou
== 1) printf("\nAchou o valor %d \n", busca); else printf("\n Nao achou o valor \n"); }
//Método de sorteamento void bubbleSort(int *vetor) { int aux; for (i = 0; i < tam - 1;
i++) { for (j = i + 1; j < tam; j++) { if (vetor[i] > vetor[j]) { aux = vetor[i]; vetor[i] = vetor[j];
vetor[j] = aux; } } } //Método de sorteamento void selectionSort(int *vetor) { int min,
aux; for (i = 0; i < tam - 1; i++) { min = i; for (j = i + 1; j < tam; j++) { if (vetor[j] <
vetor[min]) { min = j; } } aux = vetor[i]; vetor[i] = vetor[min]; vetor[min] = aux; } }
//Método de sorteamento void insertSort(int *vetor) { int chave; for (i = 1; i < tam; i++)
{ chave = vetor[i]; j = i - 1; while ((j >= 0) && (vetor[j] > chave)) { vetor[j + 1] = vetor[j];
j = j - 1; } vetor[j + 1] = chave; } } //Método de sorteamento void quicksort(int *vetor,
int began, int end) { int i, j, pivo, aux; i = began; j = end - 1; pivo = vetor[(began +
end) / 2]; while (i <= j) { while (vetor[i] < pivo && i < end) { i++; } while (vetor[j] > pivo
&& j > began) { j--; } if (i <= j) { aux = vetor[i]; vetor[i] = vetor[j]; vetor[j] = aux; i++; j--; }
} if (j > began) quicksort(vetor, began, j + 1); if (i < end) quicksort(vetor, i, end); }
/*Interface do programa. Serve tanto para saber se a opção já foi escolhida, como
```

```

também para obter dados de todas as opções escolhidas.*/ void interface(bool a,
bool b, bool c, bool d) { char string[20] = " (JA ESCOLHIDO)"; printf("\n -----
-----\n"); printf("| Qual opcao de sort voce deseja fazer? |"); printf("\n -----
-----\n"); if(!a || !b || !c || !d) { if (!a) printf("| 1) Bubble Sort
\n"); else printf("| 1) Bubble Sort%s\n", string); if (!b) printf("| 2) Selection Sort
\n"); else printf("| 2) Selection Sort%s\n", string); if (!c) printf("| 3) Insertion Sort
\n"); else printf("| 3) Insertion Sort%s\n", string); if (!d) printf("| 4) Quick Sort
\n"); else printf("| 4) Quick Sort%s\n", string); } printf("| 5) Obter dados das
opcoes escolhidas\n"); printf("| 0) Sair\n"); printf(" -----
-----\n"); } //Seleciona uma determinada opção para fazer a consulta de
sort escolhida a partir do Switch void opcaoSort(Vetores v) { int opcao=-1; //Opção
do SwitchCase. Está como -1 para não fechar o loop do laço de repetição while
(opcao != 0) { clock_t inicio, fim; //Pega o tick da máquina para verificar performance
interface (a, b, c, d); //Interface do programa recebe os verificadores de opção
scanf("%d", &opcao); char str[20] = ""; //Variável para realizar uma reescrita.
Utilizado apenas para teste. switch (opcao) { case 1: /* Se opção já foi escolhida>
opção do Case = 600 e ir para linha X (Código espaguete) o código espaguete foi
realizado aqui para economizar algumas linhas de código do programa. */ if(a)
{opcao = 500; goto LINE96532;} inicio = clock(); //pega o tick do programa e salva na
variável inicio bubbleSort(v.vetor1); //Executa o método BubbleSort na variável
desejada fim = clock(); //pega o tick do programa e salva na variável fim
mostrarTela(v.vetor1, 1); //Mostra na tela os valores de um vetor X strcpy(str,
"Bubble Sort"); //Reescrita da string str. v.bubble = fim - inicio; // subtrai fim - início
para ter uma estimativa do tempo de execução do Sort buscaBinaria(v.vetor1);
//Realiza uma busca binária a partir do valor escolhido printf("\nTempo de ordenacao
por %s: %d milisegundos\n", str, v.bubble); //Teste de concatenação de string a=true;
//Verificador para saber se essa opção já foi escolhida break; //case 2, 3 e 4 segue o
mesmo padrão do case 1 case 2: if(b) {opcao = 500; goto LINE96532;} inicio =
clock(); selectionSort(v.vetor2); fim = clock(); mostrarTela(v.vetor2, 2); strcpy(str,
"Select Sort"); v.select = fim - inicio; buscaBinaria(v.vetor2); printf("\nTempo de
ordenacao por %s: %d milisegundos\n", str, v.select); b=true; break; case 3: if(c)
{opcao = 500; goto LINE96532;} inicio = clock(); insertSort(v.vetor3); fim = clock();
mostrarTela(v.vetor3, 3); strcpy(str, "Insert Sort"); v.insert = fim - inicio;
buscaBinaria(v.vetor3); printf("\nTempo de ordenacao por %s: %d milisegundos\n",

```


```

str, v.insert); c=true; break; case 4: if(d) {opcao = 500; goto LINE96532;} inicio =
clock(); quicksort(v.vetor4, 0, tam); fim = clock(); mostrarTela(v.vetor4, 4); strcpy(str,
"Quick Sort"); v.quick = fim - inicio; buscaBinaria(v.vetor4); printf("\nTempo de
ordenacao por %s: %d milisegundos\n", str, v.quick); d=true; break; case 5:
system("cls"); //Apagar o que está escrito no console if(!a&&!b&&!c&&!d)//Se alguma
opção de sort já foi executada, mostrar a performance de determinado Sort. {
printf("\nNenhuma opcao ainda foi executada!\n"); } if(a) printf("Tempo de ordenacao
por Bubble Sort: %d milisegundos\n\n", v.bubble); if(b) printf("Tempo de ordenacao
por Select Sort: %d milisegundos\n\n", v.select); if(c) printf("Tempo de ordenacao por
Insert Sort: %d milisegundos\n\n", v.insert); if(d) printf("Tempo de ordenacao por
Quick Sort: %d milisegundos\n\n", v.quick); break; case 0: return; LINE96532:
//Código espaguete para não repetir código, ou criar um método apenas para
verificar se a opção já foi escolhida // (ver linha 210 para mais informações) case
500: system("cls"); printf("Essa opcao ja foi executada! \n"); break; default:
system("cls"); printf("Opcao Invalida! \n"); break; } } } int main() { printf("\n Bem
Vindo ao programa!\n"); Vetores v; //Inicializa o struct Vetores (Linha 10) v =
preencher(v); //Preenche e guarda os valores no ponteiro. opcaoSort(v); //Método
para fazer teste de performance dos sorts system("pause"); //Pausa o console para
não fechar na cara do fulano return 0; }

```



## 9. FICHA DE ATIVIDADES PRÁTICAS SUPERVISIONADAS



UNIVERSIDADE PAULISTA

FICHA DAS ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS

NOME:Guilherme Augusto Sbizero CorreaTURMA: CC4A33RA: F235289

CURSO: Ciências da ComputaçãoCAMPUS: TatuapéSEMESTRE: 4º SemestreTURNO: Manhã

CÓDIGO DA ATIVIDADE: 77B1SEMESTRE: 4º semestreANO GRADE: 2021

DATA DA ATIVIDADE	DESCRIÇÃO DA ATIVIDADE	TOTAL DE HORAS	ASSINATURA DO ALUNO	HORAS ATRIBUÍDAS (1)	ASSINATURA DO PROFESSOR
14/nov	Confecção do projeto detinado a Atividade Prática Supervisionada	75 Horas	Guilherme Augusto Sbizero Correa	75 Horas	

TOTAL DE HORAS ATRIBUÍDAS: 75 Horas

AVALIAÇÃO: Aprovado ou Reprovado

NOTA:

DATA: / /

CARIMBO E ASSINATURA DO COORDENADOR DO CURSO

UNIP UNIVERSIDADE PAULISTA					
FICHA DAS ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS					
NOME: Maria Fernanda Mamani Huarasi		TURMA: CC4A33	RA: N656999		
CURSO: Ciências da Computação		CAMPUS: Tatuapé	SEMESTRE: 4º Semestre	TURNOS: Manhã	
CÓDIGO DA ATIVIDADE: 7781		SEMESTRE: 4º semestre	ANO GRADE: 2021		
DATA DA ATIVIDADE	DESCRIÇÃO DA ATIVIDADE	TOTAL DE HORAS	ASSINATURA DO ALUNO	HORAS ATRIBUÍDAS (1)	ASSINATURA DO PROFESSOR
14/nov	Confecção do projeto detinado a Atividade Prática Supervisionada	75 Horas	Maria Fernanda Mamani Huarasi	75 Horas	
			TOTAL DE HORAS ATRIBUÍDAS: 75 Horas		
			AVALIAÇÃO: Aprovado ou Reprovado		
			NOTA: _____		
			DATA: ____/____/____		
			CARIMBO E ASSINATURA DO COORDENADOR DO CURSO		

UNIP UNIVERSIDADE PAULISTA					
FICHA DAS ATIVIDADES PRÁTICAS SUPERVISIONADAS - APS					
NOME: Moisés da Silva Freitas		TURMA: CC4A33	RA: F161012		
CURSO: Ciências da Computação		CAMPUS: Tatuapé	SEMESTRE: 4º Semestre	TURNOS: Manhã	
CÓDIGO DA ATIVIDADE: 7781		SEMESTRE: 4º semestre	ANO GRADE: 2021		
DATA DA ATIVIDADE	DESCRIÇÃO DA ATIVIDADE	TOTAL DE HORAS	ASSINATURA DO ALUNO	HORAS ATRIBUÍDAS (1)	ASSINATURA DO PROFESSOR
14/nov	Confecção do projeto detinado a Atividade Prática Supervisionada	75 Horas	Moisés da Silva Freitas	75 Horas	
			TOTAL DE HORAS ATRIBUÍDAS: 75 Horas		
			AVALIAÇÃO: Aprovado ou Reprovado		
			NOTA: _____		
			DATA: ____/____/____		
			CARIMBO E ASSINATURA DO COORDENADOR DO CURSO		

Link para repositório da APS no Google Drive e no GitHub

Google Drive:

[https://drive.google.com/drive/folders/1B4NagZc7nGSCszU\\_XXS68S5ah1gcK69J?usp=sharing](https://drive.google.com/drive/folders/1B4NagZc7nGSCszU_XXS68S5ah1gcK69J?usp=sharing)

GitHub: [https://github.com/GuilhermeSbizero0804/APS\\_4-Semestre](https://github.com/GuilhermeSbizero0804/APS_4-Semestre)