UNIVERSIDADE PAULISTA

CIÊNCIA DA COMPUTAÇÃO

ANDRÉ CARVALHO CERQUEIRA – D27057-5

ERICK WILLIAM LIMA RODRIGUES – N13795-9

GUSTAVO GARCIA LEAL ALVES DE MOURA – N130EA-8

THIAGO OLIVEIRA SANTOS – D227FC-2

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

SÃO PAULO – SP

2018

ANDRÉ CARVALHO CERQUEIRA – D27057-5

ERICK WILLIAM LIMA RODRIGUES – N13795-9

GUSTAVO GARCIA LEAL ALVES DE MOURA – N130EA-8

THIAGO OLIVEIRA SANTOS – D227FC-2

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

Orientador: Alan.

SÃO PAULO – SP

2018

**Sumário**

[1. Objetivo do trabalho 4](#_Toc529734522)

[2. Introdução 5](#_Toc529734523)

[3. Algoritmo de ordenação 6](#_Toc529734524)

[3.1 Bubble sort 6](#_Toc529734525)

[3.2 Select sort 6](#_Toc529734526)

[3.3 Insertion sort 6](#_Toc529734527)

[3.4 Quick sort 6](#_Toc529734528)

[4. Desenvolvimento 7](#_Toc529734529)

[5. Resultados e Discussão 9](#_Toc529734530)

[6. Considerações Finais 10](#_Toc529734531)

[7. Bibliografia 11](#_Toc529734532)

[8. Código Fonte 12](#_Toc529734533)

# 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 analise teórica de cada método de sorteamento, falando de como funciona a analise 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.

# Introdução

# 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.

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

## 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   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 35 | 55 | 12 | 21 | 4 | | 35 | 55 | 12 | 21 | 4 | | 35 | 12 | 55 | 21 | 4 | | 35 | 12 | 21 | 55 | 4 | | 35 | 12 | 21 | 4 | 55 | | Segunda interação   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 35 | 12 | 21 | 4 | 55 | | 12 | 35 | 21 | 4 | 55 | | 12 | 21 | 35 | 4 | 55 | | 12 | 21 | 4 | 35 | 55 | | 12 | 21 | 4 | 35 | 55 |   Quarta interação   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 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 | | Terceira interação   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 12 | 21 | 4 | 35 | 55 | | 12 | 21 | 4 | 35 | 55 | | 12 | 4 | 21 | 35 | 55 | | 12 | 4 | 21 | 35 | 55 | | 12 | 4 | 21 | 35 | 55 | |

Fonte: Thiago Oliveira Santos, 2018.

## Select sort

## Insertion sort

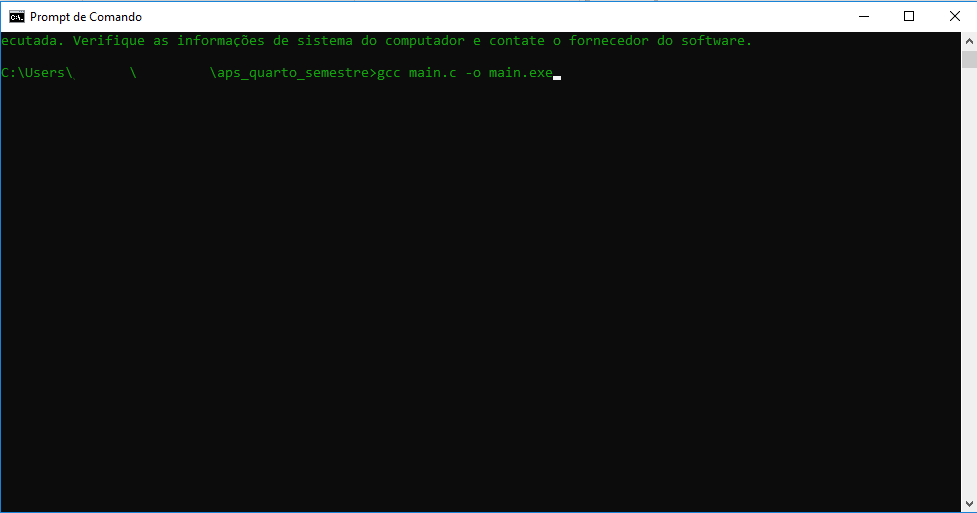
## Quick sort

# 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 x.x – Terminal CMD



Fonte: Gustavo Moura – 2018

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 há ideia de rebuscar ao máximo o que a linguagem C tem a 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 à 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 lanços 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.

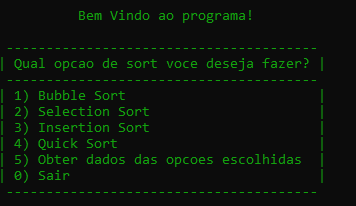
Agora, com o método Insertionsort há uma mudança relativa aos outros códigos, onde há também um laço superior e um inferior, porém o inferior apresenta já mais de uma condicional (tanto o Bubble como o Selection usavam laços de repetição for que apenas há uma condicional se o índice delas for menor que 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 ultimo 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 inicio 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 inicio 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 chama-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 novo 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 x.x – Interface do programa

  
 Fonte: Gustavo Moura – 2018

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.

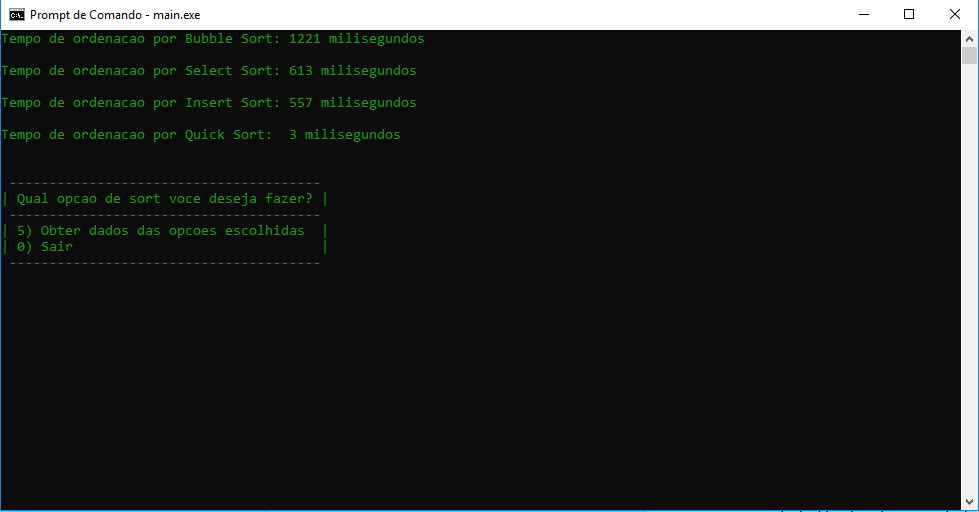
Com o resultado dá para se analisar que cada método de sorteamento tem um tempo de demora decorrente a estrutura de seu algoritmo, e em quase todos os todas as execuções, o método Quicksort se mostra sempre o mais ágil e rápido por seu modo de separar os problemas em partes baseados na recursividade.

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 analise em suas posições sempre verificando baseado em sua maior parte baseado em dois ou três laços de repetições.

# Resultados e Discussão

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



Fonte: Gustavo Moura - 2018

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.

Imagem: Gráfico de Cada Método

Autor: Gustavo Moura – 2018

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 ai que ocorre a constante analise 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 dá 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 analise ainda e também 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.

# Considerações Finais

# Bibliografia

**Não há fontes bibliográficas no documento atual.**

# Código Fonte