



Curso: Ciência da Computação

Campus: Ribeirão Preto - Vargas

Atividade Prática Supervisionada - 4º semestre

N5812D6 – Leonardo Figueiredo do Nascimento

F24GGF0 – Davidson Ramos de Pádua

N550AE0 – Yago Rafael Vidal Vitor

F297IG2 – Matheus Barbosa de Oliveira

N622458 – Willian Ferreira de Oliveira

Novembro, 2021

Índice

1. OBJETIVO E MOTIVAÇÃO DO TRABALHO	3
2. INTRODUÇÃO	3
3. ALGORITMOS UTILIZADOS	4
4. PLANO DE DESENVOLVIMENTO	6
5. PROJETO(ESTRUTURA) DO PROGRAMA	11
6. RELATÓRIO COM AS LINHAS DE CÓDIGO	15
7. APRESENTAÇÃO DO PROGRAMA EM FUNCIONAMENTO	26
8. BIBLIOGRAFIA	30
9. FICHA DE ATIVIDADES PRÁTICAS SUPERVISIONADAS	31

Objetivo e Motivação do Trabalho

O objetivo como um todo em nosso trabalho é mostrar a diferença de tempo que cada algoritmo tem entre eles e entre si na linguagem de programação C, assim sendo os algoritmos, Ordenação por troca, BubbleSort (método da bolha), QuickSort (método da troca e partição), InsertionSort (método da inserção direta), BinaryInsertionSort (método da inserção direta binária), SelectionSort (método da seleção direta), HeapSort (método da seleção em árvore), MergeSort (método da intercalação), BucketSort (método da distribuição de chave).

Cada método listado acima demonstra um tempo diferente sendo que foram realizados 5 testes com tamanhos de vetores diferentes em cada algoritmo chegando assim na conclusão de que apesar da pouca diferença crescente o algoritmo apresenta pequenas variações de tempo de um para outro.

Introdução

Neste trabalho foram desenvolvidos algoritmos de ordenação usando a linguagem de programação C, com o uso do ambiente de desenvolvimento integrado (IDE) Dev-C++.

Vamos mostrar elementos utilizados no desenvolvimento do projeto, assim como o efeito desse trabalho na formação dos alunos e mostrar como algoritmos podem ser úteis nas demais disciplinas.

C é uma linguagem de programação estruturada de alto nível, que foi desenvolvida em 1972 por Dennis Ritchie com o propósito de desenvolver uma nova versão do sistema operacional Unix, seus programas precisam ser compilados pra se tornarem executáveis, a linguagem C influenciou muitas linguagens como C++, Java, C#, e muitas outras linguagens de programação tem sua sintaxe e estrutura influenciadas por C.

Algoritmos de ordenação são um algoritmos que pega os dados desordenados de uma sequência e os coloca em ordem correta, e estes possuem diferentes variações com

diferentes tempos de execução pra certos tamanhos de vetores, sendo alguns recomendados pra vetores pequenos e outros recomendados pra vetores com um grande número de dados.

A linguagem nos permite fazer diferentes aplicações, escritas em diferentes linguagens, interoperarem entre si. O trabalho irá mostra os algoritmos tendo em si Ordenação por troca, BubbleSort (método da bolha), QuickSort (método da troca e partição), InsertionSort (método da inserção direta), BinaryInsertionSort (método da inserção direta binária), SelectionSort (método da seleção direta), HeapSort (método da seleção em árvore), MergeSort (método da intercalação) e BucketSort (método da distribuição de chave). E nos ensina a avaliar a eficiência dos algoritmos que fazem uso dessas estruturas.

Para o cálculo do tempo de execução dos algoritmos em segundos, será utilizado uma biblioteca com funções e constantes que medem o tempo de execução tendo o início sendo antes da execução do algoritmo de ordenação e o fim logo após o seu término.

Para a pesquisa dos tempos de execução, cada algoritmo de ordenação foi testado com entradas de 1.000, 5.000, 10.000, 20.000 e 30.000 numeros gerados aleatoriamente pelo programa.

Portanto o trabalho sera apresentado na disciplina de estrutura de dados, mostrando a utilização de algoritmos em linguagem C.

Algoritmos Utilizados

Bubble Sort (método da bolha): O Bubble Sort é um dos algoritmos de ordenação mais simples, mas seu uso não é recomendado para aplicações que dependem de velocidade ou trabalhem com uma grande quantidade de dados, ele basicamente percorre o vetor repetidas vezes, comparando se a próxima posição é menor do que a posição atual e realiza a troca se ela for menor que a posição atual.

Quick Sort (método da troca e partição): O Quick Sort é um algoritmo que segue a estratégia de dividir para conquistar, é considerado um dos algoritmos de ordenação mais rapidos que tem, e ele funciona inicialmente escolhendo um elemento pivot sendo geralmente a primeira posição do vetor, então ele vai fazendo uma busca na esquerda a procura de um número maior que o pivot e uma busca na direita a procura de um elemento menor que o pivot, quando encontra realiza a troca, e após a ultima troca o

pivot é posicionado no meio do vetor de forma que os números menores que ele fiquem a esquerda e os maiores fiquem a direita, e esse processo vai ser repetido recursivamente na esquerda e na direita do pivot até que o vetor esteja completamente ordenado.

Insertion Sort (método da inserção direta): O Insertion Sort é um algoritmo que funciona como um jogo de cartas, você tem uma quantidade de cartas na mão e vai ordenando elas uma de cada vez e quando compra uma carta você irá percorrer todas as cartas e inserir ela na sua devida posição, a ideia se aplica a um vetor, onde você percorre ele e se encontrar um elemento menor que os anteriores, você irá voltar nas posições e realizar a inserção dele na posição certa.

Binary Insertion Sort (método da inserção direta binária): O Binary Insertion Sort é um algoritmo parecido com o Insertion Sort, a diferença é que o algoritmo seleciona um elemento do vetor e realiza uma busca binária pra saber qual deve ser sua posição ordenada, assim não sendo necessário fazer um número percorrer todo o vetor comparando números anteriores para ordená-lo.

Selection Sort (método da seleção direta): O Selection Sort é um algoritmo que funciona como o próprio nome diz, ele seleciona uma posição do vetor e vai percorrer o vetor em busca do menor número, se encontrar ele realiza a troca e continua a busca pra ver se não tem um número menor, com a busca finalizada o algoritmo vai pras próximas posições e realiza esse processo recursivamente até o vetor estar ordenado.

Heap Sort (método da seleção em árvore): O Heap Sort funciona como uma árvore binária, ele vai estruturar o vetor em uma árvore binária através de uma função heapify e vai buscar colocar o maior número do vetor na raiz da árvore pra depois removê-lo e inseri-lo no final da raiz recursivamente até o vetor estar ordenado.

Merge Sort (método da intercalação): O Merge Sort é um algoritmo que segue a estratégia de dividir para conquistar, ele divide o vetor desordenado na metade, e vai dividir recursivamente as metades até que cada elemento do vetor esteja separado, depois ele vai fundir os elementos do vetor de forma ordenada até que o vetor esteja completo

Bucket Sort (método da distribuição de chave): O Bucket Sort é um algoritmo que divide o vetor em baldes, os baldes tem um limite do menor e maior número que pode entrar neles, cada balde vai usar o Insertion Sort para ordenar os números presentes, e depois

da ordenação os baldes são concatenados para que o vetor fique ordenado de forma crescente.

Plano de Desenvolvimento

Para esta APS, a equipe decidiu que os algoritmos de ordenação seriam feitos separadamente com o intuito de acelerar o seu desenvolvimento e facilitar a compreensão dos seus códigos.

Cada algoritmo possui três bibliotecas em comum, a biblioteca <stdio.h> que serve para manipular a entrada e saída de dados, a biblioteca <stdlib.h> que serve para realizar alocações de memória, controle de processos, conversões e outras funções, e a biblioteca <time.h> que disponibiliza funções que permitem medir o tempo tanto em segundos quanto em milissegundos.

O vetor e seu tamanho são previamente definidos e não podem ser definidos pelo usuário.

Cada algoritmo possui uma imprimir() que vai imprimir o vetor na tela, sendo ele ordenado ou desordenado.

A função main() de cada algoritmo possui um loop de repetição que insere valores aleatórios no vetor que vai ser ordenado, depois a função imprimir() é chamada pra imprimir o vetor desordenado, depois disso a função clock() é chamada pra marcar o início da contagem do tempo de execução do algoritmo de ordenação, então o algoritmo de ordenação é chamado, depois disso a função clock() é chamada novamente pra marcar o fim da contagem do tempo de execução do algoritmo de ordenação, a função imprimir() é chamada novamente pra imprimir o vetor ordenado, depois disso o tempo de execução imprimido é o início menos o fim dividido por CLOCKS_PER_SEC para converter o tempo de execução de milissegundos para segundos.

- **BubbleSort**

O desenvolvimento do algoritmo Bubble Sort acabou por ser o mais fácil dentre todos os algoritmos, sendo que o algoritmo só precisa percorrer o vetor e comparar a posição atual com a seguinte e trocá-las se a posição seguinte for menor que a atual, para isso o algoritmo vai realizar um laço de repetição com a condição de que enquanto a variável

de continuação seja diferente de 0 o algoritmo vai continuar lendo o vetor e trocar se a posição seguinte for menor que o a atual, para isso dentro do laço de repetição tem um loop que vai percorrer todo o vetor e através de um condicional vai comparar a posição atual do vetor com a seguinte e se a posição seguinte for menor que a atual, vai realizar a troca e a variável de continuação vai receber a posição atual para que o laço de repetição possa continuar operando, e assim vai se realizar recursivamente até o vetor estar completamente ordenado..

- **QuickSort**

Para o desenvolvimento do algoritmo Quick Sort houve certas complicações para como seria a escolha do pivot e como elementos menores que ele seriam postos a esquerda e maiores a direita, ocasionando na criação de uma função particao() que vai receber o vetor, o seu início e seu fim, e através disso o pivot vai receber a primeira posição do vetor, a esquerda recebe o início do vetor e a direita recebe o fim do vetor, enquanto a esquerda for menor que a direita um laço de repetição vai fazer uma busca com um loop a procura de um elemento maior que o vetor a direita e um loop vai fazer a procura de um elemento menor que o pivot a direita, um condicional vai fazer a troca do elemento da esquerda com o elemento da direita se a esquerda for menor que a direita, após a última troca o elemento da direita troca de posição com o pivot. Para realizar a ordenação total, foi criado a função quickSort() que recebe o vetor, a sua esquerda e a sua direita, se a esquerda for menor que a direita ele recursivamente define o pivot chamando a particao() com o vetor, sua esquerda e sua direita passados no quickSort(), e depois vai realizar a partição da esquerda e da direita do pivot, para a esquerda o quickSort() é chamado novamente porém com a direita com uma posição a abaixo do pivot, para a direita o quickSort() é chamado novamente porém com a esquerda com uma posição acima do pivot, e assim vai se realizar recursivamente até o vetor estar completamente ordenado.

- **InsertionSort**

O desenvolvimento do algoritmo Insertion Sort também foi simples de desenvolver, uma função insertionSort() que recebe como parâmetros um vetor e seu tamanho foi criada para ordenar os valores do vetor, dentro dela tem um loop que vai percorrer o vetor e dentro deste loop tem uma variável auxiliar que vai guardar o valor da posição atual do vetor e outro loop com uma variável j que recebe a posição atual do vetor, que vai fazer uma busca pra ver se j é maior que 0 e se o auxiliar é menor que o valor da posição anterior de j, se for, o valor da posição j vai receber o valor da posição anterior, e após o loop o valor da posição de j recebe o auxiliar, e assim vai se realizar recursivamente até o vetor estar completamente ordenado.

- **BinaryInsertionSort**

Houve complicações para o desenvolvimento do algoritmo Binary Insertion Sort, tendo em mente que teria que implementar um Insertion Sort mas com uma busca binária para achar a posição correta para inserir o elemento do vetor, para isso foi criada a função que vai realizar a busca binária `buscaBinária()` que recebe como parâmetros o vetor, uma variável `i`, seu menor valor e seu maior valor, dentro dela é declarado um laço de repetição com a condição de que enquanto o menor valor for menor ou igual o maior valor ele vai continuar o ciclo, dentro desse laço de repetição uma variável `meio` é declarada, sendo que ela recebe o menor valor somado com o maior valor menos o menor valor dividido por 2, um condicional é declarado com a condição de que se a variável `i` for idêntica ao valor do `meio` do vetor então a função retorna `meio + 1`, ou se a variável `i` for maior que o valor do `meio` do vetor então o menor valor passa a ser a posição depois do `meio`, senão o maior valor passa a ser a posição antes do `meio`, após o fim do laço de repetição a função retorna o menor número. Para a ordenação do vetor, foi criada a função `binaryInsertionSort()` que recebe o vetor e seu tamanho como parâmetro, dentro dela é declarada 5 variáveis inteiras, sendo elas `i`, `j`, `k`, `loc` e `selected`, um loop vai ser criado para percorrer o vetor, dentro dele a variável `j` vai receber a posição anterior a posição atual, a variável `selected` vai receber o valor da posição atual, a variável `loc` vai achar a localização onde o `selected` deveria ser inserido, pra isso ele chama a função `buscaBinária()` tendo em seus parâmetros o vetor, o `selected`, o 0 e `j`, após isso um laço de repetição é declarado, tendo como condição a variável `j` sendo maior ou igual a `loc`, dentro desse laço o valor da próxima posição `j` recebe o valor da posição atual de `j`, depois `j` é decrementado, após o fim do ciclo o valor da próxima posição de `j` recebe o `selected`, e assim vai se realizar recursivamente até o vetor estar completamente ordenado.

- **SelectionSort**

Para o desenvolvimento do algoritmo Selection Sort foi desenvolvido a função `troca()` que recebe dois parâmetro, sendo o primeiro aquele vai ter o seu valor trocado pelo valor do segundo parâmetro, para a ordenação do vetor foi criada a função `selectionSort()` que recebe o vetor e seu tamanho, dentro dela há um loop que vai percorrer o vetor, dentro deste loop uma variável `menor` recebe a posição atual do vetor, então um segundo loop com uma variável `j` é criado percorrendo o vetor a partir da próxima posição, dentro deste loop há um condicional em que a variável `menor` vai receber a variável `j` se o valor de `j` for menor que o valor da variável `menor`, com o fim do primeiro loop vem um condicional que vai chamar a função `troca()` tendo como parâmetros o valor da posição atual e o valor da variável `menor` se a posição atual for diferente de `menor`, e assim vai se realizar recursivamente até o vetor estar completamente ordenado.

- **HeapSort**

Para o desenvolvimento do algoritmo Heap Sort foi desenvolvido a função `troca()` que recebe dois parâmetro, sendo o primeiro aquele vai ter o seu valor trocado pelo valor do

segundo parâmetro, para a estruturação do vetor em uma árvore binária foi feito a função `heapify()` que recebe como parâmetros um vetor, seu início e seu final, três variáveis são criadas, sendo elas pai, filhoE e filhoD, sendo o que pai(raiz) vai receber o final, o filhoE(filho da esquerda) vai receber o final multiplicado por 2 e somado com 1, e filhoD(filho da direita) vai receber o final multiplicado por 2 e somado com 2, a partir daí vem três condicionais, sendo que a primeira vai fazer com que o filho da esquerda vire o pai se o filhoE for menor que o tamanho do vetor e o valor do filhoE for maior que o valor do pai, a segunda vai fazer com que o filho da direita vire o pai se o filhoD for menor que o tamanho do vetor e o valor do filhoD for maior que o valor do pai, e a terceira vai chamar a função `troca()` tendo como parâmetros o valor do final e o valor do pai e vai chamar a função `heapify()` recursivamente nos filhos se o pai for diferente do final. Para a ordenação acontecer foi feito uma função `heapSort()` que recebe o vetor e seu tamanho como parâmetro, um loop de repetição vai ser criado tendo seu início no final da metade do tamanho do vetor menos 1 e enquanto ele percorre o vetor ao contrário ele vai chamar o `heapify()` tendo como parâmetros o vetor, seu tamanho e a posição atual para estruturar o vetor em uma árvore binária, ao terminar esse loop outro loop é criado, tendo seu início no final do vetor e enquanto ele percorre o vetor ao contrário ele vai chamar a função `troca()` para que o pai(raiz) seja jogado no final do vetor, após a troca a função `heapify()` é chamada de novo tendo como parâmetros o vetor, sua posição atual e o início para reestruturar a árvore binária após posicionar o pai no fim do vetor, e assim vai se realizar recursivamente até o vetor estar completamente ordenado.

- **MergeSort**

Para o desenvolvimento do algoritmo Merge Sort houve o uso da biblioteca `<math.h>`, o algoritmo possui uma função `merge()` que recebe como parâmetros o vetor, seu início, seu meio e seu fim, dentro da função são declaradas 9 variáveis, sendo elas `vAUX` que vai funcionar como um vetor auxiliar, `p1`, `p2`, `tamanho`, `i`, `j`, `k`, `fim1` e `fim2`, sendo que `p1` recebe o início, `p2` recebe o meio somado com 1, `fim1` e `fim2` recebem 0, `tamanho` recebe o início menos fim somado com 1, `vAUX` vai ter um espaço alocado na memória através do uso da função `malloc()` tendo como parâmetro a variável `tamanho`, se `vAUX` não for nulo, então um loop que vai percorrer o vetor enquanto `i` for menor que a variável `tamanho` é criado, sendo que `i` recebe 0 inicialmente, dentro dele há uma condicional cuja condição é que enquanto `fim1` e `fim2` forem 0 eles serão invertidos para que a condição seja verdadeira, dentro deste condicional há três condicionais, sendo que o primeiro é responsável pela fusão e ordenação do vetor, ele vai adicionar o valor da posição de `p1` na posição atual de `vAUX` se o valor da posição de `p1` for menor que a posição de `p2`, senão, o valor da posição de `p2` vai ser adicionado na posição atual de `vAUX`, o segundo condicional vai mudar o valor de `fim1` para 1 se `p1` for maior que o meio, o terceiro condicional vai mudar o valor de `fim2` para 1 se `p2` for maior que o fim, se `fim1` e `fim2` valem 1 então o vetor auxiliar `vAUX` vai receber uma cópia do que sobrar dos valores de `p1` se `fim1` for 1, senão vai receber uma cópia do que sobrar dos valores de `p2`, após o término do loop outro loop é criado, este vai copiar os valores do vetor auxiliar `vAUX` e mandar para o vetor original, nele a variável `j` recebe 0 e a variável `k`

recebe o início do vetor original, enquanto j for menor que a variável tamanho, o loop vai adicionando os valores da posição j do vetor auxiliar no valor da posição k do vetor original, após o término deste loop e do condicional, o espaço de memória alocado pelo vetor auxiliar vAUX é liberado pela função free(). Para este algoritmo realizar a divisão recursivamente e depois combinar os valores do vetor ordenado, foi criada uma função mergeSort() que recebe um vetor, seu início e seu fim, dentro dela é declarada uma variável meio, se o início for menor que o fim então a variável meio chama a função matemática floor() que vai receber início somado com o fim dividido por 2 e vai devolver o valor do meio arredondado, após isso a função mergeSort() vai ser chamada duas vezes pra começar a recursividade, sendo que a primeira recebe como parâmetros o vetor, seu início e o meio, já a segunda vai receber como parâmetros o vetor, o meio somado com 1 e o seu fim, depois disso a função merge() é chamada para combinar os valores, sendo seus parâmetros o vetor, seu início, seu meio e seu fim, e assim vai se realizar recursivamente até o vetor estar completamente ordenado.

- **BucketSort**

Houve complicações para o desenvolvimento do algoritmo Bucket Sort, sendo uma das principais como os baldes seriam feitos, para isso foi criada uma estrutura balde, que tem uma variável qtd inteira e um vetor valores do tipo inteiro que recebe o mesmo tamanho que o vetor original, uma função insertionSort() que recebe como parâmetros um vetor e seu tamanho foi criada para ordenar os valores do vetor dentro dos baldes, dentro dela tem um loop que vai percorrer o vetor e dentro deste loop tem uma variável auxiliar que vai guardar o valor da posição atual do vetor e outro loop com uma variável j que recebe a posição atual do vetor, que vai fazer uma busca pra ver se j é maior que 0 e se o auxiliar é menor que o valor da posição anterior de j, se for, o valor da posição j vai receber o valor da posição anterior, e após o loop o valor da posição de j recebe o auxiliar, e assim vai se realizar recursivamente até o vetor dentro do balde estar completamente ordenado. Para a criação dos baldes e a ordenação do vetor foi criada a função bucketSort() que recebe como parâmetros o vetor e seu tamanho, dentro dela 6 variáveis inteiras são declaradas, sendo elas i, j, maior, menor, nBaldes e pos, um ponteiro b que aponta para a estrutura balde é criado, inicialmente o maior e o menor recebem o valor do início do vetor, após isso um loop que vai percorrer o vetor é iniciado, dentro dele o maior vai receber o valor da posição atual se o valor da posição atual do vetor for maior que a variável maior e o menor vai receber o valor da posição atual se o valor da posição atual do vetor for menor que a variável menor, após esse loop a variável nBaldes que vai definir quantos baldes vão ser usados vai receber o maior menos o menor dividido por 100 somado a 1, após isso o ponteiro b é alocado na memória através da função malloc() tendo como tamanho a variável nBaldes, depois disso é feito um loop que vai inicializar os baldes, enquanto i for menor que nBaldes, dentro dele a variável qtd que representa a quantidade de elementos dentro do balde vai receber 0, após isso um loop que vai inserir os valores nos baldes é criado, ele vai percorrer o vetor e a variável pos que representa a posição do valor do vetor vai receber o valor da posição atual menos o menor dividido por 100, no balde da posição pos os valores da quantidade

do balde da posição pos recebem o valor da posição atual do vetor e após isso a quantidade do balde da posição pos é incrementada, após o término deste loop a variável pos recebe 0 e outro loop é criado para fazer a ordenação e depois a concatenação dos baldes, enquanto a variável i for menor que nBaldes ela vai chamar a função insertionSort() tendo como parâmetros os valores do balde da posição atual e sua quantidade qtd, após realizar a ordenação por inserção ele vai criar um outro loop que vai colocar os valores ordenados no vetor, para isso a variável j recebe 0, enquanto j for menor que o balde atual, a posição pos no vetor recebe os valores de j do balde atual e então j vai ser incrementado, após o vetor estar ordenado o espaço ocupado pelos baldes representado pelo ponteiro b vai ser liberado por meio da função free().

Estrutura do Projeto

- **BubbleSort**

O algoritmo BubbleSort consiste de uma função bubbleSort(), que vai realizar a ordenação enquanto o vetor tiver um numero maior que o próximo, uma função imprimir() que vai imprimir o vetor independente se ele estiver ordenado ou não, e a função main() vai definir valores aleatórios, imprimir o vetor desordenado, chamar a função bubbleSort() e imprimir o vetor ordenado e seu tempo de execução.

Seu tempo de execução para vetores de 1.000,5.000,10.000,20.000,30.000 números foi:

V[1.000] = 0,00200 segundos

V[5.000] = 0,05800 segundos

V[10.000] = 0,25300 segundos

V[20.000] = 1,10000 segundos

V[30.000] = 2,59300 segundos

- **QuickSort**

O algoritmo QuickSort consiste de uma função particao(), que escolhe o inicio como pivot e vai realizar a busca de um número maior que o pivo da esquerda pra direita, e de um número menor que o pivo da direita pra esquerda, e vai realizar a troca se a

esquerda for maior que a direita, após a última troca o pivot é colocado no meio do vetor, uma função quickSort(), que vai chamar a particao() e a realizar recursivamente a ordenação e divisão da esquerda e da direita do pivot até todos os números estarem ordenados, uma função imprimir() que vai imprimir o vetor independente se ele estiver ordenado ou não, e a função main() vai definir valores aleatórios, imprimir o vetor desordenado, chamar a função quickSort() e imprimir o vetor ordenado e seu tempo de execução.

Seu tempo de execução para vetores de 1.000,5.000,10.000,20.000,30.000 números foi:

V[1.000] = 0,00100 segundos

V[5.000] = 0,00100 segundos

V[10.000] = 0,00200 segundos

V[20.000] = 0,00300 segundos

V[30.000] = 0,00400 segundos

- **InsertionSort**

O algoritmo InsertionSort consiste de uma função insertionSort(), que vai realizar uma busca nas posições do vetor e se encontrar um número menor que os anteriores durante a busca, vai retornar nas posições e colocá-lo em sua devida posição, uma função imprimir() que vai imprimir o vetor independente se ele estiver ordenado ou não, e a função main() vai definir valores aleatórios, imprimir o vetor desordenado, chamar a função insertionSort() e imprimir o vetor ordenado e seu tempo de execução.

Seu tempo de execução para vetores de 1.000,5.000,10.000,20.000,30.000 números foi:

V[1.000] = 0,00100 segundos

V[5.000] = 0,01900 segundos

V[10.000] = 0,07000 segundos

V[20.000] = 0,26300 segundos

V[30.000] = 0,62600 segundos

- **BinaryInsertionSort**

O algoritmo BinaryInsertionSort consiste de uma função buscaBinaria(), que vai realizar uma busca binária no vetor e retornar o menor número, uma função binaryInsertionSort(), que vai chamar a função buscaBinaria() pra cada elemento do vetor e vai realizar a ordenação, uma função imprimir() que vai imprimir o vetor independente se ele estiver

ordenado ou não, e a função `main()` vai definir valores aleatórios, imprimir o vetor desordenado, chamar a função `binaryInsertionSort()` e imprimir o vetor ordenado e seu tempo de execução.

Seu tempo de execução para vetores de 1.000,5.000,10.000,20.000,30.000 números foi:

V[1.000] = 0,00100 segundos

V[5.000] = 0,01300 segundos

V[10.000] = 0,04800 segundos

V[20.000] = 0,17600 segundos

V[30.000] = 0,40000 segundos

- **SelectionSort**

O algoritmo SelectionSort consiste de uma função `troca()`, que vai realizar a troca baseado nos parâmetros que receber, uma função `selectionSort()`, que vai realizar uma busca no vetor atrás do menor número, se um número menor que o número selecionado for encontrado durante a busca então a função `troca()` é chamada e o número menor é posicionado no número selecionado, uma função `imprimir()` que vai imprimir o vetor independente se ele estiver ordenado ou não, e a função `main()` vai definir valores aleatórios, imprimir o vetor desordenado, chamar a função `selectionSort()` e imprimir o vetor ordenado e seu tempo de execução.

Seu tempo de execução para vetores de 1.000,5.000,10.000,20.000,30.000 números foi:

V[1.000] = 0,00100 segundos

V[5.000] = 0,03200 segundos

V[10.000] = 0,12200 segundos

V[20.000] = 0,47000 segundos

V[30.000] = 1,05900 segundos

- **HeapSort**

O algoritmo HeapSort consiste de uma função `troca()`, que vai realizar a troca baseado nos parâmetros que receber, uma função `heapify()` que vai estruturar o vetor como se fosse uma árvore binária, uma função `heapSort()` que vai chamar a função `heapify()` e depois ordenar o vetor colocando os maiores elementos nos final do vetor, uma função `imprimir()` que vai imprimir o vetor independente se ele estiver ordenado ou não, e a

função main() vai definir valores aleatórios, imprimir o vetor desordenado, chamar a função heapSort() e imprimir o vetor ordenado e seu tempo de execução.

Seu tempo de execução para vetores de 1.000,5.000,10.000,20.000,30.000 números foi:

V[1.000] = 0,00100 segundos

V[5.000] = 0,00100 segundos

V[10.000] = 0,00300 segundos

V[20.000] = 0,00500 segundos

V[30.000] = 0,00800 segundos

- **MergeSort**

O algoritmo MergeSort consiste de uma função merge(), que vai pegar metades de um vetor e juntá-las de forma ordenada, uma função mergeSort(), que vai dividir o vetor recursivamente e depois chamar a função merge() para combinar as metades de forma ordenada, uma função imprimir() que vai imprimir o vetor independente se ele estiver ordenado ou não, e a função main() vai definir valores aleatórios, imprimir o vetor desordenado, chamar a função mergeSort() e imprimir o vetor ordenado e seu tempo de execução.

Seu tempo de execução para vetores de 1.000,5.000,10.000,20.000,30.000 números foi:

V[1.000] = 0,00100 segundos

V[5.000] = 0,00200 segundos

V[10.000] = 0,00300 segundos

V[20.000] = 0,00500 segundos

V[30.000] = 0,00800 segundos

- **BucketSort**

O algoritmo BucketSort consiste de uma estrutura balde que vai , uma função insertionSort() que vai ordenar os elementos do vetor nos baldes, uma função bucketSort() que vai colocar os elementos do vetor nos baldes e então chamar a função insertionSort() pra ordená-los e depois concatenar eles e colocar de volta no vetor, uma função imprimir() que vai imprimir o vetor independente se ele estiver ordenado ou não, e a função main() vai definir valores aleatórios, imprimir o vetor desordenado, chamar a função bucketSort() e imprimir o vetor ordenado e seu tempo de execução.

Seu tempo de execução para vetores de 1.000,5.000,10.000,20.000,30.000 números foi:

V[1.000] = 0,00100 segundos

V[5.000] = 0,00100 segundos

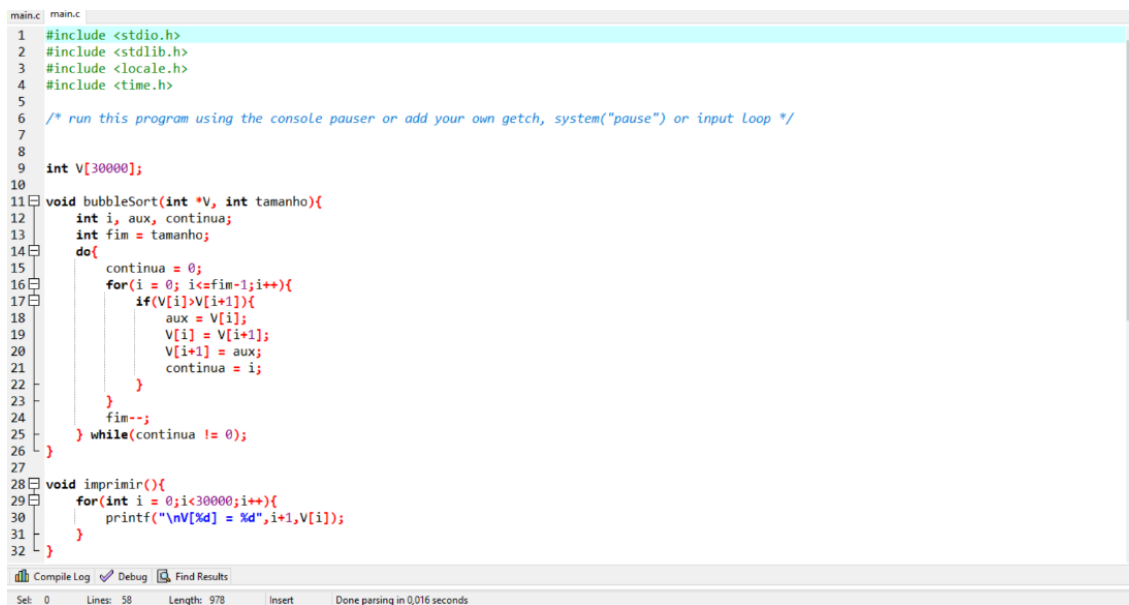
V[10.000] = 0,00100 segundos

V[20.000] = 0,00300 segundos

V[30.000] = 0,00400 segundos

Relatório com Linhas de Código

- BubbleSort**



```
main.c main.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <locale.h>
4  #include <time.h>
5
6  /* run this program using the console pauser or add your own getch, system("pause") or input loop */
7
8
9  int V[30000];
10
11 void bubbleSort(int *V, int tamanho){
12     int i, aux, continua;
13     int fim = tamanho;
14     do{
15         continua = 0;
16         for(i = 0; i<=fim-1;i++){
17             if(V[i]>V[i+1]){
18                 aux = V[i];
19                 V[i] = V[i+1];
20                 V[i+1] = aux;
21                 continua = i;
22             }
23         }
24         fim--;
25     } while(continua != 0);
26 }
27
28 void imprimir(){
29     for(int i = 0;i<30000;i++){
30         printf("\nV[%d] = %d",i+1,V[i]);
31     }
32 }
```

Compile Log Debug Find Results

Set: 0 Lines: 58 Length: 978 Insert Done parsing in 0,016 seconds

```
main.c main.c
27
28 void imprimir(){
29     for(int i = 0; i < 30000; i++){
30         printf("\nV[%d] = %d", i+1, V[i]);
31     }
32 }
33
34 int main(){
35     setlocale(LC_ALL, "Portuguese");
36
37     for(int i=0; i<30000; i++){
38         V[i] = rand() % 30000;
39     }
40
41     imprimir();
42
43     clock_t inicio = clock();
44
45     bubbleSort(V, 30000);
46
47     clock_t fim = clock();
48
49     printf("\n-----Valores ordenados-----\n");
50     imprimir();
51
52     printf("\nTempo de execucao: %f segundos\n", (double)(fim - inicio) / CLOCKS_PER_SEC);
53
54     system("pause");
55 }
56
57
58
Compile Log Debug Find Results
Sel: 0 Lines: 58 Length: 978 Insert Done parsing in 0.016 seconds
```

- QuickSort

```
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 /* run this program using the console pauser or add your own getch, system("pause") or input loop */
6
7 int V[1000];
8
9 int particao(int *V, int inicio, int fim){
10     int E, D, pivot, aux;
11     E = inicio;
12     D = fim;
13     pivot = V[inicio];
14
15     while(E < D){
16         //faz a busca pra encontrar se tem um numero maior que o pivot a sua esquerda
17         while(V[E] <= pivot && E <= fim){
18             E++;
19         }
20
21         //faz a busca pra encontrar se tem um numero menor que o pivot a sua direita
22         while(V[D] > pivot && D >= 0){
23             D--;
24         }
25
26         if(E < D){
27             aux = V[E];
28             V[E] = V[D];
29             V[D] = aux;
30         }
31     }
32 }
```



```

32
33 //Após a ultima troca, o elemento da direita troca de posição com o pivot
34 V[inicio] = V[D];
35 V[D] = pivot;
36 return D;
37 }
38
39 void quicksort(int *V, int E, int D){
40     int pivot;
41     if(E < D){
42         //acha o pivot
43         pivot = particao(V,E,D);
44         //ordena tudo da esquerda do pivot
45         quicksort(V,E,pivot-1);
46         //ordena tudo da direita do pivot
47         quicksort(V,pivot+1,D);
48     }
49 }
50
51 void imprimir(){
52     for(int i = 0; i < 1000; i++){
53         printf("\nV[%d]: %d", i+1, V[i]);
54     }
55 }
56 int main() {
57     for(int i = 0; i < 1000; i++){
58         V[i] = rand() % 1000;
59     }
60
61     imprimir();
62
63

```

Compile Log Debug Find Results

Sel: 0 Lines: 75 Length: 1468 Insert Done parsing in 0,016 seconds

```

44 //ordena tudo da esquerda do pivot
45 quicksort(V,E,pivot-1);
46 //ordena tudo da direita do pivot
47 quicksort(V,pivot+1,D);
48 }
49 }
50
51 void imprimir(){
52     for(int i = 0; i < 1000; i++){
53         printf("\nV[%d]: %d", i+1, V[i]);
54     }
55 }
56 int main() {
57     for(int i = 0; i < 1000; i++){
58         V[i] = rand() % 1000;
59     }
60
61     imprimir();
62
63     clock_t inicio = clock();
64     quicksort(V,0,1000);
65     clock_t fim = clock();
66
67     printf("\n-----Vetor ordenado-----\n");
68
69     imprimir();
70
71     printf("\nTempo de execucao: %f segundos\n", (double)(fim - inicio) / CLOCKS_PER_SEC);
72
73     return 0;
74 }
75

```

Compile Log Debug Find Results

- **InsertionSort**


```

main.c main.c main.c main.c main.c main.c main.c main.c main.c main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 /* run this program using the console pauser or add your own getch, system("pause") or input loop */
5 int V[10000];
6
7 int buscaBinaria(int *V, int i, int menor, int maior){
8     while (menor <= maior) {
9         int meio = menor + (maior - menor) / 2;
10        if (i == V[meio])
11            return meio + 1;
12        else if (i > V[meio])
13            menor = meio + 1;
14        else
15            maior = meio - 1;
16    }
17
18    return menor;
19 }
20
21 void binaryInsertionSort(int *V, int tamanho){
22     int i, loc, j, k, selected;
23
24     for (i = 1; i < tamanho; ++i) {
25         j = i - 1;
26         selected = V[i];
27
28         // acha a localização onde o selected deveria ser inserido
29         loc = buscaBinaria(V, selected, 0, j);
30
31         // move todos os elementos depois do selected pra dar espaço
32         while (j >= loc) {
33             V[j + 1] = V[j];
34             j--;
35         }
36         V[j + 1] = selected;
37     }
38 }
39
40 void imprime(V)

```

Sel: 0 Lines: 66 Length: 1462 Insert Done parsing in 0,031 seconds

```

main.c main.c main.c main.c main.c main.c main.c main.c main.c main.c
7 int buscaBinaria(int *V, int i, int menor, int maior){
8     while (menor <= maior) {
9         int meio = menor + (maior - menor) / 2;
10        if (i == V[meio])
11            return meio + 1;
12        else if (i > V[meio])
13            menor = meio + 1;
14        else
15            maior = meio - 1;
16    }
17
18    return menor;
19 }
20
21 void binaryInsertionSort(int *V, int tamanho){
22     int i, loc, j, k, selected;
23
24     for (i = 1; i < tamanho; ++i) {
25         j = i - 1;
26         selected = V[i];
27
28         // acha a localização onde o selected deveria ser inserido
29         loc = buscaBinaria(V, selected, 0, j);
30
31         // move todos os elementos depois do selected pra dar espaço
32         while (j >= loc) {
33             V[j + 1] = V[j];
34             j--;
35         }
36         V[j + 1] = selected;
37     }
38 }
39
40 void imprime(V)

```

Sel: 0 Lines: 66 Length: 1462 Insert Done parsing in 0,031 seconds

```
main.c main.c main.c main.c main.c main.c main.c main.c
35 }
36     V[j + 1] = selected;
37 }
38 }
39
40 void imprimir(){
41     for(int i = 0; i < 30000; i++){
42         printf("V[%d]: %d\n", i+1, V[i]);
43     }
44 }
45
46 int main() {
47     for(int i = 0; i < 30000; i++){
48         V[i] = rand() % 30000;
49     }
50
51     imprimir();
52
53     clock_t inicio = clock();
54
55     binaryInsertionSort(V, 30000);
56
57     clock_t fim = clock();
58
59     printf("\n-----Vetor ordenado-----\n");
60
61     imprimir();
62
63     printf("\nTempo de execucao: %f segundos\n", (double)(fim - inicio) / CLOCKS_PER_SEC);
64     system("pause");
65     return 0;
66 }
```

Compile Log Debug Find Results

Sel: 0 Lines: 66 Length: 1445 Insert Done parsing in 0,016 seconds

- **SelectionSort**

```
main.c main.c main.c main.c main.c main.c main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 /* run this program using the console pauser or add your own getch, system("pause") or input loop */
5 int V[30000];
6
7 void troca(int *a, int *b) {
8     int aux = *a;
9     *a = *b;
10    *b = aux;
11 }
12
13 void selectionSort(int *V, int tamanho){
14     int i, j, menor;
15     for(i = 0; i < tamanho-1; i++){
16         menor = i;
17         for(j = i+1; j < tamanho; j++){
18             if(V[j] < V[menor]){
19                 menor = j;
20             }
21         }
22         if(i != menor){
23             troca(&V[i], &V[menor]);
24         }
25     }
26 }
27
28 void imprimir(){
29     for(int i = 0; i < 30000; i++){
30         printf("V[%d]: %d\n", i+1, V[i]);
31     }
32 }
```

Compile Log Debug Find Results

```
main.c main.c main.c main.c main.c main.c
23     }
24     }
25     }
26 }
27
28 void imprimir(){
29     for(int i = 0; i < 30000; i++){
30         printf("V[%d]: %d\n", i+1, V[i]);
31     }
32 }
33
34 int main() {
35     for(int i = 0; i < 30000; i++){
36         V[i] = rand() % 30000;
37     }
38
39     imprimir();
40
41     clock_t inicio = clock();
42
43     selectionSort(V, 30000);
44
45     clock_t fim = clock();
46
47     printf("\n-----Vetor ordenado-----\n");
48
49     imprimir();
50
51     printf("\nTempo de execucao: %f segundos\n", (double)(fim - inicio) / CLOCKS_PER_SEC);
52     system("pause");
53     return 0;
54 }
```

- **HeapSort**

```
main.c main.c main.c main.c main.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  /* run this program using the console pauser or add your own getch, system("pause") or input loop */
5
6  int V[30000];
7
8  void troca(int *a, int *b) {
9      int aux = *a;
10     *a = *b;
11     *b = aux;
12 }
13
14 //Estruturar vetor como árvore binária
15 void heapify(int *V, int n, int i){
16     int pai = i;
17     int filhoE = 2 * i + 1;
18     int filhoD = 2 * i + 2;
19
20     //Se o filho da esquerda é maior que o pai
21     if (filhoE < n && V[filhoE] > V[pai])
22         pai = filhoE;
23
24     //Se o filho da direita é maior que o pai
25     if (filhoD < n && V[filhoD] > V[pai])
26         pai = filhoD;
27
28
29     if (pai != i) {
30         troca(&V[i], &V[pai]);
31         //faz o heapify recursivamente nos filhos
32         heapify(V, n, pai);
33     }
```

```
main.c main.c main.c main.c main.c
28
29 if (pai != i) {
30     troca(&V[pai], &V[i]);
31     //faz o heapify recursivamente nos filhos
32     heapify(V, n, pai);
33 }
34 }
35
36 void heapsort(int *V, int n){
37
38     for (int i = n / 2 - 1; i >= 0; i--)
39         heapify(V, n, i);
40
41     for (int i = n - 1; i >= 0; i--) {
42         troca(&V[0], &V[i]);
43         heapify(V, i, 0);
44     }
45 }
46 }
47
48 void imprimir(){
49     for(int i = 0; i<30000; i++){
50         printf("V[%d]: %d\n", i+1, V[i]);
51     }
52 }
53
54 int main() {
55     for(int i = 0; i<30000; i++){
56         V[i] = rand() % 30000;
57     }
58     imprimir();
59 }
```

Compile Log Debug Find Results

```
main.c main.c main.c main.c main.c
42     troca(&V[0], &V[i]);
43
44     heapify(V, i, 0);
45 }
46 }
47
48 void imprimir(){
49     for(int i = 0; i<30000; i++){
50         printf("V[%d]: %d\n", i+1, V[i]);
51     }
52 }
53
54 int main() {
55     for(int i = 0; i<30000; i++){
56         V[i] = rand() % 30000;
57     }
58     imprimir();
59
60     clock_t inicio = clock();
61
62     heapsort(V, 30000);
63
64     clock_t fim = clock();
65
66     printf("\n-----Vetor ordenado-----\n");
67
68     imprimir();
69
70     printf("\nTempo de execucao: %f segundos\n", (double)(fim - inicio) / CLOCKS_PER_SEC);
71     return 0;
72 }
73 }
```

Compile Log Debug Find Results

- MergeSort

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <math.h>
5 /* run this program using the console pauser or add your own getch, system("pause") or input Loop */
6
7 int V[30000];
8
9 void merge(int *V, int inicio, int meio, int fim){
10     int *vAUX, p1, p2, tamanho, i, j, k;
11     int fim1 = 0, fim2 = 0;
12     tamanho = fim - inicio + 1;
13     p1 = inicio;
14     p2 = meio + 1;
15     vAUX = (int *) malloc(tamanho * sizeof(int));
16     if(vAUX != NULL){
17         for(i = 0; i < tamanho; i++){
18             if(!fim1 && !fim2){
19                 if(V[p1] < V[p2]){
20                     vAUX[i] = V[p1++];
21                 } else{
22                     vAUX[i] = V[p2++];
23                 }
24                 if(p1 > meio) fim1 = 1;
25                 if(p2 > fim) fim2 = 1;
26             }
27             else{
28                 if(!fim1){
29                     vAUX[i] = V[p1++];
30                 }
31                 else{
32                     vAUX[i] = V[p2++];

```

```

20         vAUX[i] = V[p1++];
21     } else{
22         vAUX[i] = V[p2++];
23     }
24     if(p1 > meio) fim1 = 1;
25     if(p2 > fim) fim2 = 1;
26 }
27 else{
28     if(!fim1){
29         vAUX[i] = V[p1++];
30     }
31     else{
32         vAUX[i] = V[p2++];
33     }
34 }
35 }
36 for(j = 0, k = inicio; j < tamanho; j++, k++){
37     V[k] = vAUX[j];
38 }
39 }
40 free(vAUX);
41 }
42
43 void mergeSort(int *V, int inicio, int fim){
44     int meio;
45     if(inicio < fim){
46         meio = floor((inicio + fim) / 2);
47         mergeSort(V, inicio, meio);
48         mergeSort(V, meio + 1, fim);
49         merge(V, inicio, meio, fim);
50     }
51 }

```

Compile Log Debug Find Results

```
main.c main.c main.c main.c main.c
48     mergeSort(V,meio+1,fim);
49     merge(V,inicio,meio,fim);
50 }
51 }
52
53 void imprimir(){
54     for(int i = 0;i<30000;i++){
55         printf("V[%d]: %d\n",i+1,V[i]);
56     }
57 }
58
59 int main() {
60     for(int i = 0;i<30000;i++){
61         V[i] = rand() % 30000;
62     }
63
64     imprimir();
65
66     clock_t inicio = clock();
67
68     mergeSort(V,0,30000);
69
70     clock_t fim = clock();
71
72     printf("\n-----Vetor ordenado-----\n");
73
74     imprimir();
75
76     printf("\nTempo de execucao: %f segundos\n",(double)(fim - inicio) / CLOCKS_PER_SEC);
77     system("pause");
78     return 0;
79 }
```

- **BucketSort**

```
main.c main.c main.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  /* run this program using the console pauser or add your own getch, system("pause") or input loop */
5
6  struct balde{
7      int qtd;
8      int valores[30000];
9  };
10
11  int V[30000];
12
13  void insertionSort(int *V,int tamanho){
14      int i, j, aux;
15      for(i=1;i<tamanho;i++){
16          aux = V[i];
17          for(j=i;(j>0) && (aux < V[j - 1]);j--){
18              V[j] = V[j-1];
19          }
20          V[j] = aux;
21      }
22  }
23
24  void bucketSort(int *V,int tamanho){
25      int i,j,maior,menor,nBalde,pos;
26      struct balde *b;
27
28      maior = menor = V[0];
29      for(i=0;i<tamanho;i++){
30          if(V[i]>maior){
31              maior = V[i];
32          }
33      }
```



```
main.c main.c main.c
24 void bucketSort(int *V, int tamanho){
25     int i, j, maior, menor, nBaldes, pos;
26     struct balde *b;
27
28     maior = menor = V[0];
29     for(i=0; i<tamanho; i++){
30         if(V[i]>maior){
31             maior = V[i];
32         }
33         if(V[i]<menor){
34             menor = V[i];
35         }
36     }
37
38     nBaldes = (maior - menor) / 100 + 1;
39     b = (struct balde *) malloc(nBaldes * sizeof(struct balde));
40     for(i=0; i<nBaldes; i++){
41         b[i].qtd = 0;
42     }
43
44     for(i=0; i<tamanho; i++){
45         pos = (V[i] - menor)/100;
46         b[pos].valores[b[pos].qtd] = V[i];
47         b[pos].qtd++;
48     }
49     pos = 0;
50     for(i=0; i<nBaldes; i++){
51         insertionSort(b[i].valores, b[i].qtd);
52         for(j=0; j<b[i].qtd; j++){
53             V[pos] = b[i].valores[j];
54             pos++;
55         }
56     }
57 }
58
59 void imprimir(){
60     for(int i = 0; i<30000; i++){
61         printf("V[%d]: %d\n", i+1, V[i]);
62     }
63 }
64
65
```

Compile Log Debug Find Results

```
main.c main.c main.c
34     menor = V[i];
35 }
36 }
37
38 nBaldes = (maior - menor) / 100 + 1;
39 b = (struct balde *) malloc(nBaldes * sizeof(struct balde));
40 for(i=0; i<nBaldes; i++){
41     b[i].qtd = 0;
42 }
43
44 for(i=0; i<tamanho; i++){
45     pos = (V[i] - menor)/100;
46     b[pos].valores[b[pos].qtd] = V[i];
47     b[pos].qtd++;
48 }
49 pos = 0;
50 for(i=0; i<nBaldes; i++){
51     insertionSort(b[i].valores, b[i].qtd);
52     for(j=0; j<b[i].qtd; j++){
53         V[pos] = b[i].valores[j];
54         pos++;
55     }
56 }
57 free(b);
58 }
59
60 void imprimir(){
61     for(int i = 0; i<30000; i++){
62         printf("V[%d]: %d\n", i+1, V[i]);
63     }
64 }
65
```

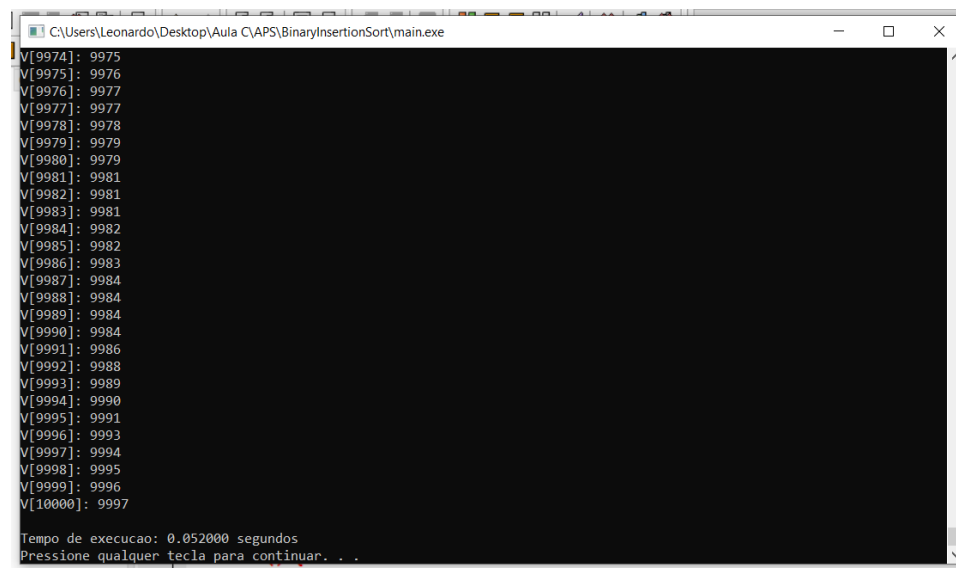
Compile Log Debug Find Results

Sel: 0 Lines: 86 Length: 1500 Insert Done parsing in 0.016 seconds

```
main.c main.c main.c
55 }
56 }
57 free(b);
58 }
59
60 void imprimir(){
61     for(int i = 0; i < 30000; i++){
62         printf("V[%d]: %d\n", i+1, V[i]);
63     }
64 }
65
66 int main() {
67     for(int i = 0; i < 30000; i++){
68         V[i] = rand() % 30000;
69     }
70
71     imprimir();
72
73     clock_t inicio = clock();
74
75     bucketSort(V, 30000);
76
77     clock_t fim = clock();
78
79     printf("\n-----Vetor ordenado-----\n");
80
81     imprimir();
82
83     printf("\nTempo de execucao: %f segundos\n", (double)(fim - inicio) / CLOCKS_PER_SEC);
84     system("pause");
85     return 0;
86 }
```

Funcionamento do Software

- **BinaryInsertionSort**



```
C:\Users\Leonardo\Desktop\Aula C\APS\BinaryInsertionSort\main.exe
V[9974]: 9975
V[9975]: 9976
V[9976]: 9977
V[9977]: 9977
V[9978]: 9978
V[9979]: 9979
V[9980]: 9979
V[9981]: 9981
V[9982]: 9981
V[9983]: 9981
V[9984]: 9982
V[9985]: 9982
V[9986]: 9983
V[9987]: 9984
V[9988]: 9984
V[9989]: 9984
V[9990]: 9984
V[9991]: 9986
V[9992]: 9988
V[9993]: 9989
V[9994]: 9990
V[9995]: 9991
V[9996]: 9993
V[9997]: 9994
V[9998]: 9995
V[9999]: 9996
V[10000]: 9997

Tempo de execucao: 0.052000 segundos
Pressione qualquer tecla para continuar. . .
```

- **BubbleSort**

```
C:\Users\Leonardo\Desktop\Aula C\APS\BubbleSort50\main.exe
V[9973] = 9974
V[9974] = 9975
V[9975] = 9975
V[9976] = 9976
V[9977] = 9977
V[9978] = 9977
V[9979] = 9978
V[9980] = 9979
V[9981] = 9979
V[9982] = 9981
V[9983] = 9981
V[9984] = 9981
V[9985] = 9982
V[9986] = 9982
V[9987] = 9983
V[9988] = 9984
V[9989] = 9984
V[9990] = 9984
V[9991] = 9984
V[9992] = 9986
V[9993] = 9988
V[9994] = 9989
V[9995] = 9990
V[9996] = 9991
V[9997] = 9993
V[9998] = 9994
V[9999] = 9995
V[10000] = 9996
Tempo de execucao: 0,287000 segundos
Pressione qualquer tecla para continuar. . .
```

- **QuickSort**

```
C:\Users\Leonardo\Desktop\Aula C\APS\QuickSort\main.exe
V[9976]: 9976
V[9977]: 9977
V[9978]: 9977
V[9979]: 9978
V[9980]: 9979
V[9981]: 9979
V[9982]: 9981
V[9983]: 9981
V[9984]: 9981
V[9985]: 9982
V[9986]: 9982
V[9987]: 9983
V[9988]: 9984
V[9989]: 9984
V[9990]: 9984
V[9991]: 9984
V[9992]: 9986
V[9993]: 9988
V[9994]: 9989
V[9995]: 9990
V[9996]: 9991
V[9997]: 9993
V[9998]: 9994
V[9999]: 9995
V[10000]: 9996
Tempo de execucao: 0.002000 segundos
-----
Process exited after 19.25 seconds with return value 0
Pressione qualquer tecla para continuar. . .
```

- **InsertionSort**

```
C:\Users\Leonardo\Desktop\Aula C\APS\InsertionSort\main.exe
V[9974]: 9975
V[9975]: 9976
V[9976]: 9977
V[9977]: 9977
V[9978]: 9978
V[9979]: 9979
V[9980]: 9979
V[9981]: 9981
V[9982]: 9981
V[9983]: 9981
V[9984]: 9982
V[9985]: 9982
V[9986]: 9983
V[9987]: 9984
V[9988]: 9984
V[9989]: 9984
V[9990]: 9984
V[9991]: 9986
V[9992]: 9988
V[9993]: 9989
V[9994]: 9990
V[9995]: 9991
V[9996]: 9993
V[9997]: 9994
V[9998]: 9995
V[9999]: 9996
V[10000]: 9997

Tempo de execucao: 0.075000 segundos
Pressione qualquer tecla para continuar. . .
```

- **SelectionSort**

```
C:\Users\Leonardo\Desktop\Aula C\APS\SelectionSort\main.exe
V[9974]: 9975
V[9975]: 9976
V[9976]: 9977
V[9977]: 9977
V[9978]: 9978
V[9979]: 9979
V[9980]: 9979
V[9981]: 9981
V[9982]: 9981
V[9983]: 9981
V[9984]: 9982
V[9985]: 9982
V[9986]: 9983
V[9987]: 9984
V[9988]: 9984
V[9989]: 9984
V[9990]: 9984
V[9991]: 9986
V[9992]: 9988
V[9993]: 9989
V[9994]: 9990
V[9995]: 9991
V[9996]: 9993
V[9997]: 9994
V[9998]: 9995
V[9999]: 9996
V[10000]: 9997

Tempo de execucao: 0.139000 segundos
Pressione qualquer tecla para continuar. . .
```

- **HeapSort**

```
C:\Users\Leonardo\Desktop\Aula C\APS\HeapSort\main.exe
V[9977]: 9977
V[9978]: 9978
V[9979]: 9979
V[9980]: 9979
V[9981]: 9981
V[9982]: 9981
V[9983]: 9981
V[9984]: 9982
V[9985]: 9982
V[9986]: 9983
V[9987]: 9984
V[9988]: 9984
V[9989]: 9984
V[9990]: 9984
V[9991]: 9986
V[9992]: 9988
V[9993]: 9989
V[9994]: 9990
V[9995]: 9991
V[9996]: 9993
V[9997]: 9994
V[9998]: 9995
V[9999]: 9996
V[10000]: 9997

Tempo de execucao: 0.003000 segundos
-----
Process exited after 17.77 seconds with return value 0
Pressione qualquer tecla para continuar. . . . .
```

- **MergeSort**

```
C:\Users\Leonardo\Desktop\Aula C\APS\MergeSort\main.exe
V[9974]: 9975
V[9975]: 9975
V[9976]: 9976
V[9977]: 9977
V[9978]: 9977
V[9979]: 9978
V[9980]: 9979
V[9981]: 9979
V[9982]: 9981
V[9983]: 9981
V[9984]: 9981
V[9985]: 9982
V[9986]: 9982
V[9987]: 9983
V[9988]: 9984
V[9989]: 9984
V[9990]: 9984
V[9991]: 9984
V[9992]: 9986
V[9993]: 9988
V[9994]: 9989
V[9995]: 9990
V[9996]: 9991
V[9997]: 9993
V[9998]: 9994
V[9999]: 9995
V[10000]: 9996

Tempo de execucao: 0.003000 segundos
Pressione qualquer tecla para continuar. . . . .
```

- **BucketSort**

```
C:\Users\Leonardo\Desktop\Aula C\APS\BucketSort\main.exe
V[9974]: 9975
V[9975]: 9976
V[9976]: 9977
V[9977]: 9977
V[9978]: 9978
V[9979]: 9979
V[9980]: 9979
V[9981]: 9981
V[9982]: 9981
V[9983]: 9981
V[9984]: 9982
V[9985]: 9982
V[9986]: 9983
V[9987]: 9984
V[9988]: 9984
V[9989]: 9984
V[9990]: 9984
V[9991]: 9986
V[9992]: 9988
V[9993]: 9989
V[9994]: 9990
V[9995]: 9991
V[9996]: 9993
V[9997]: 9994
V[9998]: 9995
V[9999]: 9996
V[10000]: 9997

Tempo de execucao: 0.001000 segundos
Pressione qualquer tecla para continuar.
```

Bibliografia

<https://www.geeksforgeeks.org/>

<https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao>

<https://www.devmedia.com.br/>

<https://henriquebraga92.medium.com/algoritmos-de-ordena%C3%A7%C3%A3o-i-bubble-sort-c162a67261ef>

<https://www.interviewkickstart.com/learn/binary-insertion-sort>

http://www.dsc.ufcg.edu.br/~pet/jornal/maio2013/materias/historia_da_computacao.html

<https://joaoarthurbm.github.io/eda/posts/selection-sort>

<https://pt.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/insertion-sort>

<http://prorum.com/?qa=4315/o-que-e-o-algoritmo-de-ordenacao-bucket-sort>

<https://joaoarthurbm.github.io/eda/posts/merge-sort/>

<https://joaoarthurbm.github.io/eda/posts/quick-sort/>

<https://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html>

Fichas da Atividade Prática Supervisionada

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, Iniciação Científica, trabalhos Individuais e em grupo, práticas de ensino e outras)

CAMPUS: Ribeirão Preto - Vargas **SEMESTRE:** 4º **TURNO:** Noturno

[illegible]

TOTAL DE HORAS: 86

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, Iniciação Científica, trabalhos individuais e em grupo, práticas de ensino e outras)

RA: F24GGF0 **CURSO:** Ciência da Computação

CAMPUS: Ribeirão Preto - Vargas **SEMESTRE:** 4º **TURNO:** Noturno

[illegible]

TOTAL DE HORAS: 86

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, iniciação científica, trabalhos individuais e em grupo, práticas de ensino e outras)

CAMPUS: Ribeirão Preto - Vargas **SEMESTRE:** 4º **TURNO:** Noturno

[illegible]

TOTAL DE HORAS: 86

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, iniciação Científica, trabalhos Individuais e em grupo, práticas de ensino e outras)

RA: N550AE0 **CURSO:** Ciência da Computação

CAMPUS: Ribeirão Preto - Vargaz

SEMESTRE:

4º

TURN0:

Noturno

[illegible]

TOTAL DE HORAS: 86

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, Iniciação Científica, trabalhos individuais e em grupo, práticas de ensino e outras)

RA: N622458 CURSO: Ciência da Computação

CAMPUS: Ribeirão Preto - Vargas **SEMESTRE:** 4º **TURNO:** Noturno

[illegible]

TOTAL DE HORAS: 86