



UNIVERSIDADE FEDERAL DA GRANDE DOURADOS
Faculdade de Ciências Exatas e Tecnologias - FACET

GEILSO FARIA RODRIGUES DA SILVA

**RELATÓRIO DA AVALIAÇÃO DE ALGORITMOS DE ORDENAÇÃO
EM FUNÇÃO DO TEMPO E TAMANHO DOS VETORES**
Algoritmos e Estruturas de Dados II

Dourados - MS
2021

GEILSO FARIA RODRIGUES DA SILVA

**RELATÓRIO DA AVALIAÇÃO DE ALGORITMOS DE ORDENAÇÃO
EM FUNÇÃO DO TEMPO E TAMANHO DOS VETORES**

Relatório de análise do tempo gasto para ordenação de vetores totalmente desordenados e de diferentes tamanhos para os algoritmos de ordenação *mergesort*, *quicksort*, *selection sort*, *insertion sort* e *bubble sort*, em linguagem C.

**Dourados - MS
2021**

Resumo

Este relatório mostra em detalhes um estudo comparativo entre 5 algoritmos de ordenação, em que ambos foram expostos às mesmas condições de tamanho e complexidade, após isso foi realizada a aferição do tempo em milissegundos através da biblioteca `time.h` em linguagem de programação C. Através de sua metodologia leva o leitor a entender os pontos fracos e fortes de cada método de ordenação.

Palavras - chave: Algoritmos de Ordenação, Linguagem C e Comparação.

SUMÁRIO

1.0- Introdução	5
2.0 - Desenvolvimento	5
2.0.0 - Algoritmos de Ordenação	6
2.0.1- Bubble Sort	6
2.0.2 - Insertion Sort	7
2.0.3 - Selection Sort	7
2.0.4 - Quicksort	8
2.0.5 - Mergesort	9
2.1.0 - Testes	11
2.1.1 - Como foram realizados?	11
2.1.2 - Padronização dos testes	12
2.1.3 - Vetor de 10 posições	13
2.1.4 - Vetor de 100 posições	16
2.1.5 - Vetor de 1000 posições	19
2.1.6 - Vetor de 10000 posições	23
2.1.7 - Vetor de 100000 posições	28
3.0 - Conclusão	32
4.0 - Bibliografia	34

1.0-INTRODUÇÃO

Segundo o estudo “A Universe of Opportunities and Challenges”, realizado pela consultoria EMC, que apontou que desde os anos de 2006 a 2010, a quantidade de dados digitais, que antes era de 166 Exabytes, passou para 988 Exabytes, sextuplicando o volume de dados eletrônicos existentes. Sendo assim, se essas informações fossem armazenadas de forma aleatória, causariam grandes perdas tempo e recursos. Com o intuito de contribuir para a resolução deste problema, o uso de algoritmos que possam ordenar esses dados, de modo a facilitar o ganho de tempo e gasto de recursos são indispensáveis.

Eventualmente, diante da disponibilidade de diversos algoritmos de ordenação, surge a dúvida de qual é o preferível?. Pois, tratando de recursos computacionais sabemos que é oportuno resolver o problema da melhor maneira possível e com um menor gasto de tempo e recursos, para que assim não sejam usados recursos desnecessários. Assim sendo, este trabalho se propõe a comparar 5 algoritmos de ordenação, sendo eles: *mergesort*, *quicksort*, *selection sort*, *insertion sort* e *bubble sort*.

Para a realização deste experimento foi utilizado da biblioteca *time.h* em linguagem de programação C, e de sua função *Clock*. Em seguida, foram realizados a compilação dos algoritmos e o cálculo de tempo, para tal foram utilizados vetores de 10,100,1000,10000 e 100000 posições, em seus piores casos (totalmente desordenados).

Logo, para facilitar a compreensão do leitor, este relatório de experimento foi dividido em dois capítulos, sendo que cada capítulo corresponde a uma parte da amostragem (algoritmos de ordenação e testes). Cada capítulo conta com subcapítulos, que mostram em detalhes os algoritmos de ordenação envolvidos e todas as etapas do experimento.

2.0- DESENVOLVIMENTO

2.0.0-ALGORITMOS DE ORDENAÇÃO

2.0.1.BUBBLE SORT

Segundo Schildt (1996) “A ordenação por bolhas é uma ordenação por trocas. Ela envolve repetidas comparações e, se necessário, a troca de elementos adjacentes. Os elementos são como bolhas em um tanque de água, cada uma procura seu próprio nível”. Devido a essas repetidas comparações, este algoritmo apesar de ser um dos mais famosos, é também um dos piores algoritmos em termos de recursos computacionais, sendo conhecido como “o demônio das trocas”.

Aplicação em Linguagem C

```
void bubble_sort(int v[], int n)
{
    int i, j, aux;
    for(i=n-1; i>=1; i--)
    {
        for(j=0; j<i; j++)
        {
            if(v[j] > v[j+1])
            {
                aux = v[j];
                v[j] = v[j+1];
                v[j+1] = aux;
            }
        }
    }
}
```

Fonte: Algoritmo disponibilizado em sala de aula.

2.0.2- INSERTION SORT

Do mesmo modo, o *insertion sort* possui similar facilidade de aplicação do *bubble sort*, com sua execução baseada na comparação e inserção direta dos elementos. Sendo que, cada elemento selecionado previamente é comparado com o seu posterior, e se o elemento selecionado for maior é realizado a troca, em seguida o elemento que era o posterior ao selecionado anteriormente é comparado com o anterior do que estava na posição previamente selecionada e se este for menor é realizado novamente a troca de posição, até que cada elemento esteja na sua ordenação correta. Como no *bubble sort*, este também causa um maior gasto computacional.

Aplicação em Linguagem C

```
void insertionSort(int lista[], int quantidade)
{
    int i, j, aux, menorID;
    for(int i = 1; i < quantidade; i++)
    {
        aux = lista[i];
        for(j = i - 1; j >= 0 && lista[j] > aux; j--)
        {
            lista[j+1] = lista[j];
        }
        lista[j+1] = aux;
    }
}
```

Fonte: Algoritmo disponibilizado em sala de aula.

2.0.3 - SELECTION SORT

Novamente, como no *bubble sort* e o *insertion sort*, o *selection sort* possui a mesma facilidade de elaboração. Sua característica dominante se baseia em mudar o elemento com o menor valor para a

primeira posição, o segundo elemento de menor valor para a segunda posição, e assim por diante, sendo que sempre os elementos a esquerda sempre será menor que os a direita (SOUZA, RICARTE E LIMA, 2017).

Aplicação em Linguagem C

```
void selectionSort(int lista[], int quantidade)
{
    int i, j, aux, menorID;
    for(int i = 0; i < quantidade-1; i++)
    {
        menorID = i;
        for(int j = i + 1; j < quantidade; j++)
        {
            if(lista[j] < lista[menorID])
                menorID = j;
        }
        aux = lista[i];
        lista[i] = lista[menorID];
        lista[menorID] = aux;
    }
}
```

Fonte: Algoritmo disponibilizado em sala de aula.

2.0.4 - QUICKSORT

O comportamento deste algoritmo tem como base a ideia de “dividir para conquistar”. Do qual, previamente é selecionado um elemento pivô qualquer, e realizada a divisão da estrutura de modo a ordenar os valores. Portanto, ao fim de sua execução os elementos à esquerda/direita do pivô selecionado são maiores/menores, do que o pivô. Posteriormente, através da recursão os subconjuntos que foram divididos um antes do pivô e outro após o pivô são ordenados. Por este motivo, este algoritmo é considerado como um dos melhores algoritmos de ordenação.

Aplicação em Linguagem C

```
void quick_sort(int v[], int left, int right)
{
    int i, j, centro, aux;
    i = left;
    j = right;
    centro = v[(left+right)/2];
    while(i<=j)
    {
        while(v[i]<centro && i<right){i++;}
        while(v[j]>centro && j>left){j--;}
        if(i<=j)
        {
            aux = v[i];
            v[i] = v[j];
            v[j] = aux;
            i++;
            j--;
        }
    }
    if(j>left) {quick_sort(v, left, j);}
    if(i<right) {quick_sort(v, i, right);}
}
```

Fonte: Algoritmo disponibilizado em sala de aula.

2.0.5 - MERGE SORT

De acordo com (SOUZA, RICARTE E LIMA, 2017). “Este algoritmo tem como objetivo a reordenação de uma estrutura linear por meio da quebra, intercalação e união dos n elementos existentes. Em outras palavras, a estrutura a ser reordenada será, de forma recursiva, subdividida em estruturas menores até que não seja mais possível

fazê-lo. Em seguida, os elementos serão organizados de modo que cada subestrutura ficará ordenada ”.

Logo, como no *quicksort*, o princípio fundamental é a ideia de “dividir para conquistar”. Além do mais, ambos possuem funcionamento semelhante em termos de recursos computacionais. Entretanto, uma de suas diferenças é o que no *mergesort* a estrutura é dividida em outras muitas menores até que não seja mais possível e após isso o vetor é reordenado, já no *quicksort* a estrutura é dividida em duas metades e reordenadas.

Aplicação em Linguagem C

```
void mergeSort(int v[], int inicio, int fim)
{
    int meio;
    if(inicio < fim)
    {
        meio = floor((inicio+fim)/2);
        mergeSort(v,inicio,meio);
        mergeSort(v,meio+1,fim);
        merge(v,inicio,meio,fim);
    }
}

void merge(int v[], int inicio, int meio, int fim)
{
    int *temp, p1, p2, tamanho, i, j, k;
    int fim1 = 0, fim2 = 0;
    tamanho = fim-inicio+1;
    p1 = inicio;
    p2 = meio+1;
    temp = (int *) malloc(tamanho*sizeof(int));
    if(temp != NULL)
    {
        for(i=0; i<tamanho; i++)
        {
            if(!fim1 && !fim2)
```

```

{
    if(v[p1] < v[p2])
        temp[i]=v[p1++];
    else
        temp[i]=v[p2++];

    if(p1>meio) fim1=1;
    if(p2>fim) fim2=1;
}
else
{
    if(!fim1)
        temp[i]=v[p1++];
    else
        temp[i]=v[p2++];
}
}
for(j=0, k=inicio; j<tamanho; j++, k++)
    v[k]=temp[j];
}
free(temp);
}

```

2.1.0-TESTES

2.1.0. COMO FORAM REALIZADOS?

Para a realização dos experimentos, como já citado, foi utilizado da biblioteca time.h e da função clock em linguagem C, sendo o princípio de funcionamento da função clock baseado em contar a quantidade de pulsos de clock realizados em um segundo. Feito isso, para uma melhor visualização foi realizado a conversão para milissegundos, no qual é dado pela divisão do tempo em segundos por 1000.

1 Pulso de clock em eletrônica digital



Fonte: <https://integrada.minhabiblioteca.com.br/#/books/9788536530390>

Vários Pulsos de clock em eletrônica digital



A função clock trabalha contando esses pulsos.

Fonte: <https://integrada.minhabiblioteca.com.br/#/books/9788536530390>

2.1.1. PADRONIZAÇÃO DOS TESTES

Com a finalidade de produzir resultados confiáveis, para a execução dos testes de comparação foram adotadas medidas de precauções, sendo elas:

1. As testagens foram iniciadas após 5 minutos do computador ligado, a fim de se evitar processos iniciando, que de algum modo poderiam comprometer os resultados.
2. Apenas a IDE do *CodeBlocks* estava aberta, e executando apenas o código de teste.
3. Foram realizados 3 testes, a fim de obter-se o valor médio de tempo, esta medida foi tomada devido a variação do tempo de compilação e execução.
4. Os vetores eram os mesmos para os seus tamanhos equivalentes, e todos em seus piores casos.
5. Todos o experimento foi executado no mesmo computador, com as seguintes configurações:

- Processador: Intel(R) Core(TM) i3-7020 U CPU @ 2.30GHz
- Memória RAM: 4,00GB
- Arquitetura do Sistema operacional: 64 bits
- Arquitetura do Processador: baseado em x64
- Sistema Operacional: Windows 10 Home Single Language

6. O dispositivo não estava carregando, ação tomada a fim de evitar o aquecimento dos componentes que pudessem influenciar nos resultados.

2.1.2 - VETOR DE 10 POSIÇÕES

Primeiramente, com um vetor de 10 posições, o tempo permaneceu em 0.00 para todos algoritmos de ordenação. Indício este, de que em vetores pequenos a diferença de tempo é a mesma, não importando o método usado.

Função Main Vetor de 10 posições

```
int main(void)
{
    int v[9]={};
    int i,j;
    float tempo=0;
    clock_t t;
    for(i=0,j=10;i<10,j>0;i++,j--)
    {
        v[i]=j;
    }
    t = clock();
    mergeSort(v,0,9);
    t = clock()- t;
    printf("Tempo de execucao em MILESSEGUNDOS MERGE SORT: %f\n",
    ((float)t)/(CLOCKS_PER_SEC/1000);
    for(i=0,j=10;i<10,j>0;i++,j--)
    {
        v[i]=j; }
    t = clock();
    quick_sort(v,0,9);
    t = clock() - t;
    printf("Tempo de execucao em MILESSEGUNDOS QUICK SORT: %f\n",
    ((float)t)/(CLOCKS_PER_SEC/1000);
```


```

    for(i=0,j=10;i<10,j>0;i++,j--)
    {
        v[i]=j;
    }
    t = clock();
    insertionSort(v,10);
    t = clock() - t;
    printf("Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);
    for(i=0,j=10;i<10,j>0;i++,j--)
    {
        v[i]=j;
    }
    t = clock();
    selectionSort(v,10);
    t = clock() - t;
    printf("Tempo de execucao em MILESSEGUNDOS SELECTION SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);
    for(i=0,j=10;i<10,j>0;i++,j--)
    {
        v[i]=j;
    }
    t = clock();
    bubble_sort(v,10);
    t = clock() - t;
    printf("Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);
    return 0;
}

```

Fonte: Geilso Faria Rodrigues da Silva

1ª execução


 "C:\Users\geils\OneDrive\ALG2\P2\p2 do 10.exe"

```
Tempo de execucao em MILESSEGUNDOS MERGE SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 0.000000

Process returned 0 (0x0)   execution time : 1.040 s
Press any key to continue.
```

Fonte:Geilso Faria Rodrigues da Silva

2ª execução


 "C:\Users\geils\OneDrive\ALG2\P2\p2 do 10.exe"

```
Tempo de execucao em MILESSEGUNDOS MERGE SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 0.000000

Process returned 0 (0x0)   execution time : 0.660 s
Press any key to continue.
```

Fonte:Geilso Faria Rodrigues da Silva

3ª execução

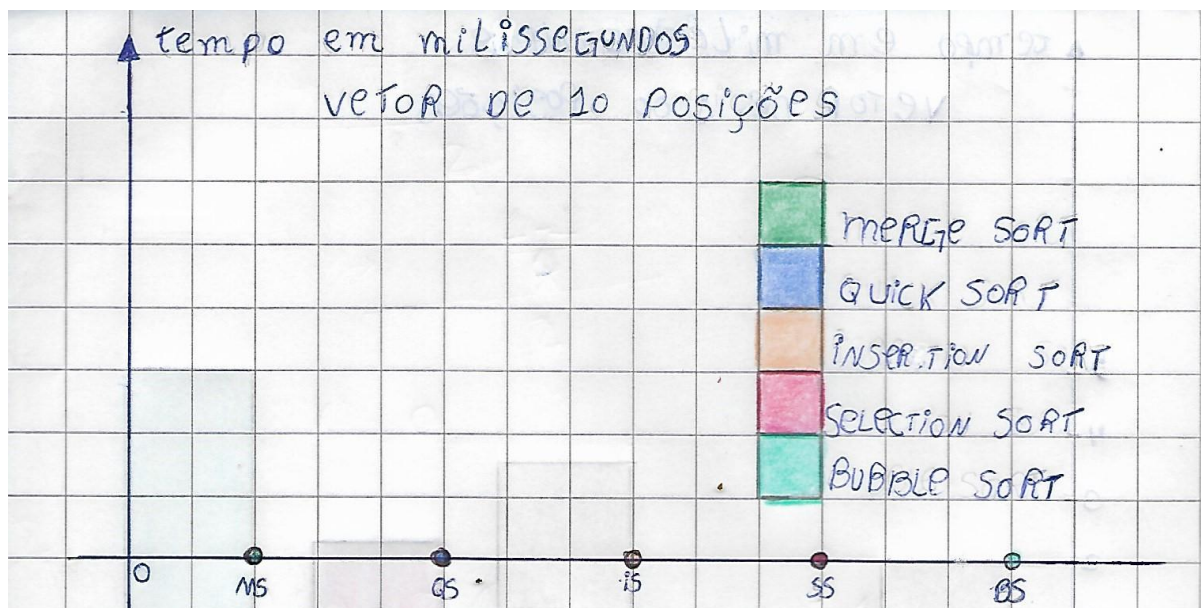
 "C:\Users\geils\OneDrive\ALG2\P2\p2 do 10.exe"

```
Tempo de execucao em MILESSEGUNDOS MERGE SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 0.000000

Process returned 0 (0x0)   execution time : 0.379 s
Press any key to continue.
```

Fonte:Geilso Faria Rodrigues da Silva

Gráfico em função do tempo de execução



Fonte: Geilso Faria Rodrigues da Silva

2.1.3 - VETOR DE 100 POSIÇÕES

Logo após, para o vetor de 100 posições, apesar da maioria dos algoritmos não realizarem nenhum pulso de clock, o *bubble sort* foi o único em que a sua média de tempo foi acima de 0, fato esse que pode ser explicado devido a sua grande quantidade de comparações realizadas.

Função Main Vetor de 100 posições

```
int main(void)
{
    int v[99]={};
    int i,j;
    float tempo=0;
    clock_t t;

    for(i=0,j=100;i<100,j>0;i++,j--)
    {
        v[i]=j;
    }
    t = clock();
```



```

mergeSort(v,0,99);

t = clock()- t;

printf("Tempo de execucao em MILESSEGUNDOS MERGE SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);


for(i=0,j=100;i<100,j>0;i++,j--)
{
    v[i]=j;
}

t = clock();

quick_sort(v,0,99);

t = clock() - t;

printf("Tempo de execucao em MILESSEGUNDOS QUICK SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);

for(i=0,j=100;i<100,j>0;i++,j--)
{
    v[i]=j;
}

t = clock();

insertionSort(v,100);

t = clock() - t;

printf("Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);

for(i=0,j=100;i<100,j>0;i++,j--)
{
    v[i]=j;
}

t = clock();

selectionSort(v,100);

t = clock() - t;

printf("Tempo de execucao em MILESSEGUNDOS SELECTION SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);

```


```

    for(i=0,j=100;i<100,j>0;i++,j--)
    {
        v[i]=j;}
    t = clock();
    bubble_sort(v,100);
    t = clock() - t;
    printf("Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);
    return 0;
}

```

Fonte: Geilso Faria Rodrigues da Silva

1ª execução

 "C:\Users\geils\OneDrive\ALG2\P2\p2 do 100.exe"

```


Tempo de execucao em MILESSEGUNDOS MERGE SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 1.000000

Process returned 0 (0x0)   execution time : 0.413 s
Press any key to continue.

```

Fonte:Geilso Faria Rodrigues da Silva

2ª execução

 "C:\Users\geils\OneDrive\ALG2\P2\p2 do 100.exe"

```

Tempo de execucao em MILESSEGUNDOS MERGE SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 0.000000

Process returned 0 (0x0)   execution time : 0.626 s
Press any key to continue.

```

Fonte:Geilso Faria Rodrigues da Silva

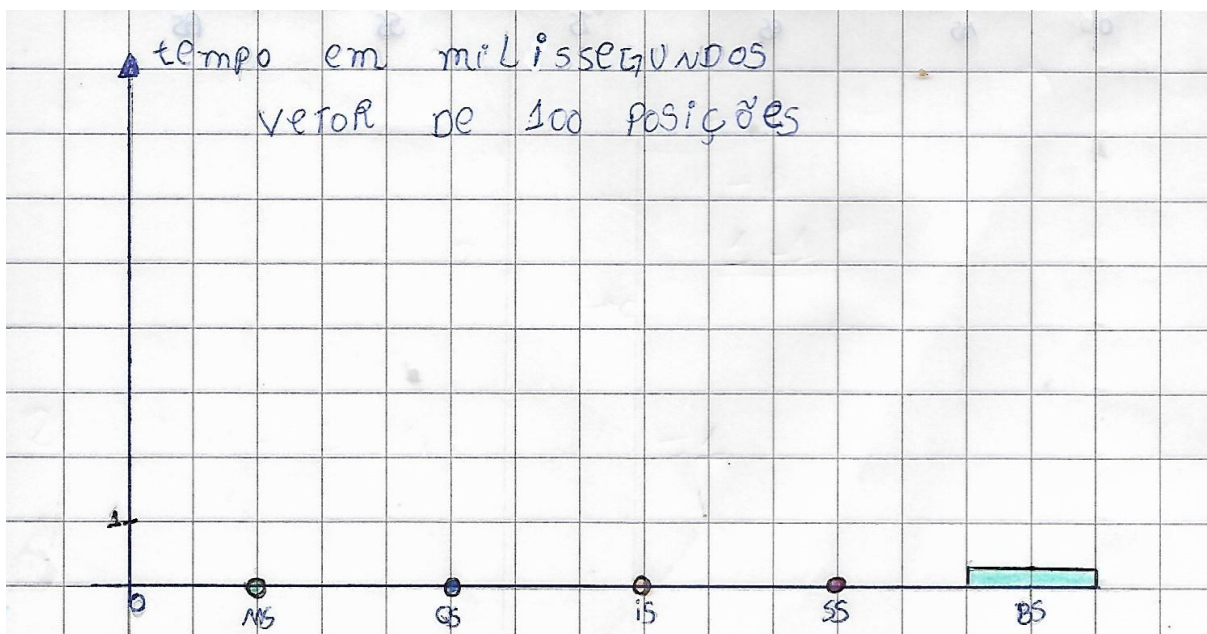
3ª execução

```
"C:\Users\geils\OneDrive\ALG2\P2\p2 do 100.exe"
Tempo de execucao em MILESSEGUNDOS MERGE SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 0.000000

Process returned 0 (0x0)   execution time : 0.732 s
Press any key to continue.
```

Fonte:Geilso Faria Rodrigues da Silva

Gráfico em função do tempo de execução



Fonte:Geilso Faria Rodrigues da Silva

2.1.4 - VETOR DE 1000 POSIÇÕES

Em seguida, para o vetor de 1000 posições a diferença dos algoritmos de ordenação já pôde ser percebida sutilmente. Algoritmos mais eficazes como o *mergesort* e *quicksort* já apresentaram tempos inferiores comparados ao restante. Tratando particularmente da diferença de tempo entre mergesort (2 milissegundos) e quicksort (0.66 milissegundos), que apesar de ser pequena, pode ser explicada devido ao particionamento minucioso do *mergesort*, levando assim mais tempo em relação ao *quicksort*.

Agora, tratando dos métodos menos eficazes estes já apresentaram tempo ligeiramente maior, sendo o *insertion sort* (3.66 milissegundos), *selection sort* (2.33 milissegundos), e em particular no caso do *bubble sort* (5 milissegundos) já se destacando por ser o maior tempo dentre todos os algoritmos.

Função Main Vetor de 1000 posições

```
int main(void)
{
    int v[999]={};
    int i,j;
    float tempo=0;
    clock_t t;

    for(i=0,j=1000;i<1000,j>0;i++,j--)
    {
        v[i]=j;
    }
    t = clock();
    mergeSort(v,0,999);
    t = clock()- t;
    printf("Tempo de execucao em MILESSEGUNDOS MERGE SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);
    for(i=0,j=1000;i<1000,j>0;i++,j--)
    {
        v[i]=j;
    }
    t = clock();
    quick_sort(v,0,999);
    t = clock() - t;
    printf("Tempo de execucao em MILESSEGUNDOS QUICK SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);
    for(i=0,j=1000;i<1000,j>0;i++,j--)
    {
        v[i]=j;
```

```

    }

    t = clock();
    insertionSort(v,1000);
    t = clock() - t;
    printf("Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);
    for(i=0,j=1000;i<1000,j>0;i++,j--)
    {
        v[i]=j;
    }
    t = clock();
    selectionSort(v,1000);
    t = clock() - t;
    printf("Tempo de execucao em MILESSEGUNDOS SELECTION SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);
    for(i=0,j=1000;i<1000,j>0;i++,j--)
    {
        v[i]=j;
    }
    t = clock();
    bubble_sort(v,1000);
    t = clock() - t;
    printf("Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);

    return 0;
}

```

Fonte: Geilso Faria Rodrigues da Silva

1ª execução

```
"C:\Users\geils\OneDrive\ALG2\P2\p2 do 1000.exe"
Tempo de execucao em MILESSEGUNDOS MERGE SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 1.000000
Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: 10.000000
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 3.000000
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 1.000000

Process returned 0 (0x0)   execution time : 0.632 s
Press any key to continue.
```

Fonte:Geilso Faria Rodrigues da Silva

2ª execução

```
"C:\Users\geils\OneDrive\ALG2\P2\p2 do 1000.exe"
Tempo de execucao em MILESSEGUNDOS MERGE SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 1.000000
Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: 8.000000
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 1.000000
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 10.000000

Process returned 0 (0x0)   execution time : 0.606 s
Press any key to continue.
```

Fonte:Geilso Faria Rodrigues da Silva

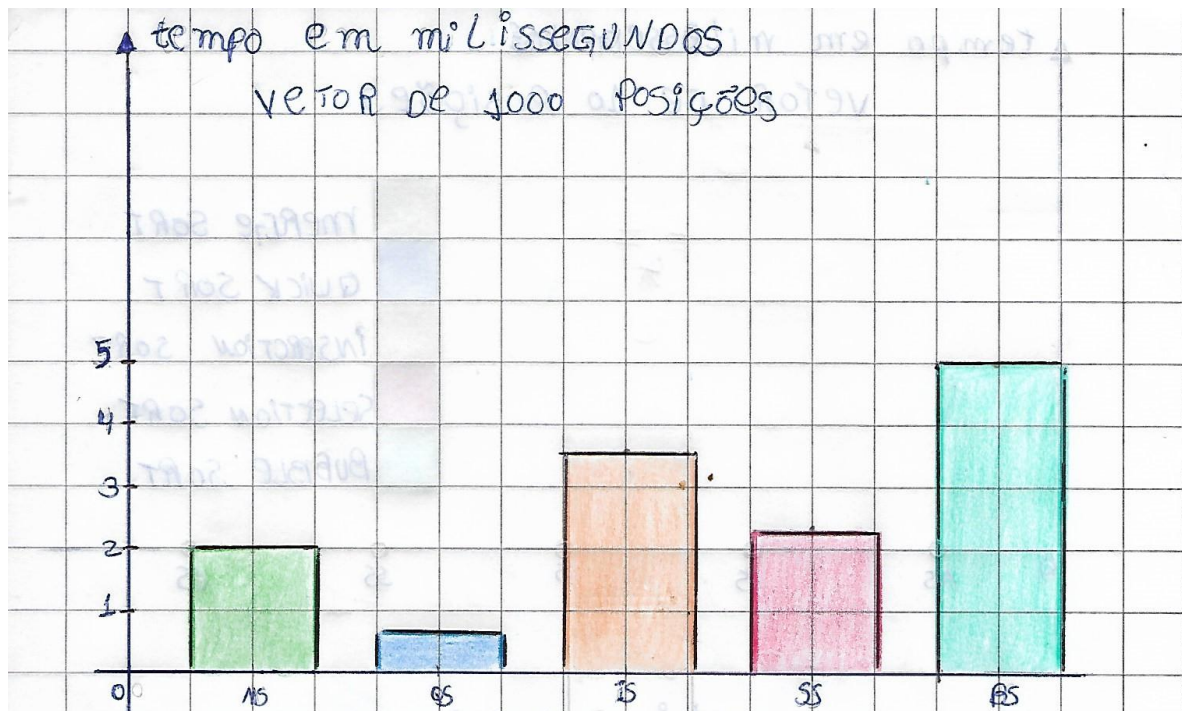
3ª execução

```
"C:\Users\geils\OneDrive\ALG2\P2\p2 do 1000.exe"
Tempo de execucao em MILESSEGUNDOS MERGE SORT: 6.000000
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: 2.000000
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 3.000000
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 4.000000

Process returned 0 (0x0)   execution time : 1.270 s
Press any key to continue.
```

Fonte:Geilso Faria Rodrigues da Silva

Gráfico em função do tempo de execução



Fonte: Geilso Faria Rodrigues da Silva

2.1.5 - VETOR DE 10000 POSIÇÕES

A partir de 10000 posições já foi possível notar claramente a diferença entre os algoritmos superiores como o *mergesort* (1.3 milissegundos) e *quicksort* (1 milissegundo). Por outro lado, o valor médio do tempo para o *insertion sort* foi de 335 milissegundos, *selection sort* com 226.3 milissegundos, e *bubble sort* novamente sendo o mais demorado de todos com 418.6 milissegundos. No mundo real apesar desta diferença parecer pequena, no mercado computacional esta já representaria perda de tempo, recursos computacionais e financeiros.

Função Main Vetor de 1000 posições

```
int main(void)
{
    int v[9999]={};
    int i,j;
    float tempo=0;
    clock_t t;
    for(i=0,j=10000;i<10000,j>0;i++,j--)
    {
        v[i]=j;
    }
}
```

```

}

t = clock();
mergeSort(v,0,9999);
t = clock() - t;
printf("Tempo de execucao em MILESSEGUNDOS MERGE SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);
for(i=0,j=10000;i<10000,j>0;i++,j--)
{
    v[i]=j;
}
t = clock();
quick_sort(v,0,9999);
t = clock() - t;
printf("Tempo de execucao em MILESSEGUNDOS QUICK SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);
for(i=0,j=10000;i<10000,j>0;i++,j--)
{
    v[i]=j;
}
t = clock();
insertionSort(v,10000);
t = clock() - t;
printf("Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);
for(i=0,j=10000;i<10000,j>0;i++,j--)
{
    v[i]=j;
}
t = clock();
selectionSort(v,10000);
t = clock() - t;
printf("Tempo de execucao em MILESSEGUNDOS SELECTION SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);
for(i=0,j=10000;i<10000,j>0;i++,j--)

```



```

{
    v[i]=j;
}
t = clock();
bubble_sort(v,10000);
t = clock() - t;
printf("Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000));
return 0;
}

```

Fonte: Geilso Faria Rodrigues da Silva

1ª execução

```

"C:\Users\geils\OneDrive\ALG2\P2\p2 do 10000.exe"
Tempo de execucao em MILESSEGUNDOS MERGE SORT: 4.000000
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 1.000000
Tempo de execucao em MILESSEGUNDOS INSERTION SORT: 264.000000
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 239.000000
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 392.000000

Process returned 0 (0x0)   execution time : 1.571 s
Press any key to continue.

```

Fonte:Geilso Faria Rodrigues da Silva

2ª execução

```

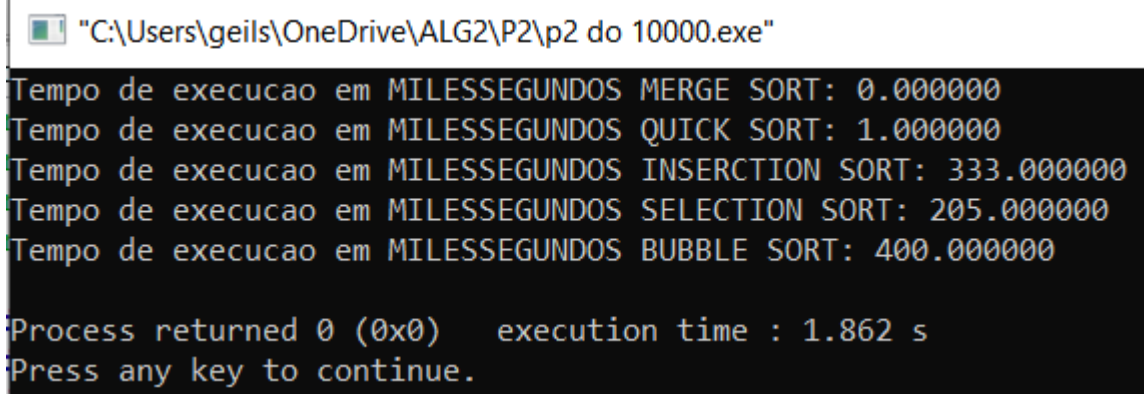
"C:\Users\geils\OneDrive\ALG2\P2\p2 do 10000.exe"
Tempo de execucao em MILESSEGUNDOS MERGE SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 1.000000
Tempo de execucao em MILESSEGUNDOS INSERTION SORT: 408.000000
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 235.000000
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 464.000000

Process returned 0 (0x0)   execution time : 1.904 s
Press any key to continue.

```

Fonte:Geilso Faria Rodrigues da Silva

3ª execução

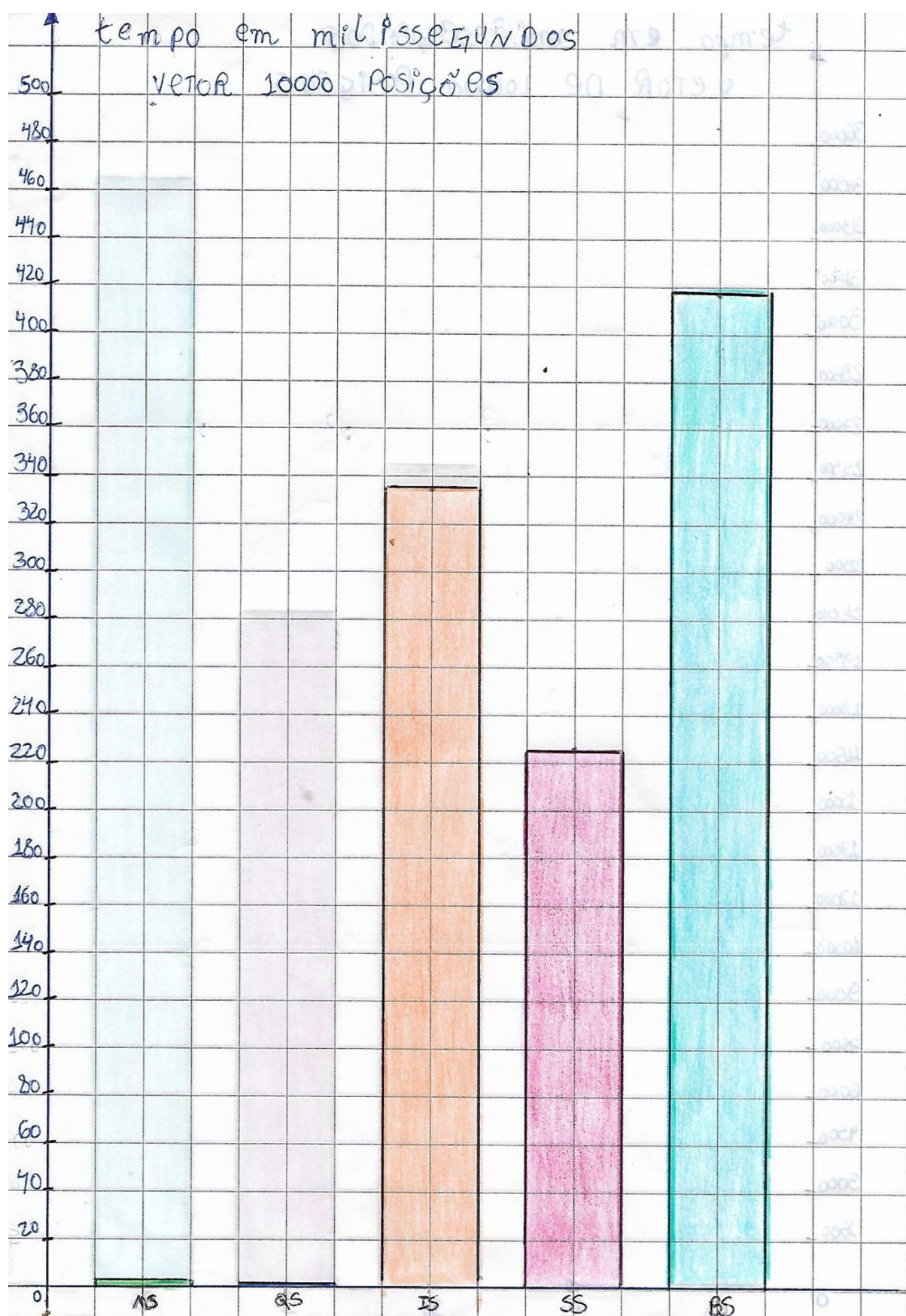


```
"C:\Users\geils\OneDrive\ALG2\P2\p2 do 10000.exe"
Tempo de execucao em MILESSEGUNDOS MERGE SORT: 0.000000
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 1.000000
Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: 333.000000
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 205.000000
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 400.000000

Process returned 0 (0x0)   execution time : 1.862 s
Press any key to continue.
```

Fonte:Geilso Faria Rodrigues da Silva

Gráfico em função do tempo de execução



Fonte: Geilso Faria Rodrigues da Silva

2.1.6 - VETOR DE 100000 POSIÇÕES

Como último experimento, em um vetor de 10000 posições, apenas confirmou que para vetores grandes a diferença de tempo é gritante entre algoritmos mais eficientes e os não tão compensativos. Para se ter ideia, o tempo médio do *mergesort* foi de 106.3 milissegundos, ainda mais surpreendente o *quicksort* com apenas 11 milissegundos . Por outro lado, para o *insertion sort* (25600 milissegundos), *selection sort* (21570.3 milissegundos) e “o demônio das trocas”, *bubble sort*, foi de 34932.6 milissegundos. Fato interessante sobre esta diferença, é que para a visualização desses valores em um gráfico chega ser até complicado, pois a escala em que mergesort e quicksort se encontram é muitas vezes menor que a de *selection sort*, *insertion sort* e *bubble sort*.

Função Main Vetor de 100000 posições

```
int main(void)
{
    int v[99999]={};
    int i,j;
    float tempo=0;
    clock_t t;

    for(i=0,j=100000;i<100000,j>0;i++,j--)
    {
        v[i]=j;
    }

    t = clock();
    mergeSort(v,0,99999);
    t = clock()- t;
    printf("Tempo de execucao em MILESSEGUNDOS MERGE SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);

    for(i=0,j=100000;i<100000,j>0;i++,j--)
    {
        v[i]=j;
    }

    t = clock();
```

```

    quick_sort(v,0,99999);

    t = clock() - t;

    printf("Tempo de execucao em MILESSEGUNDOS QUICK SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);

    for(i=0,j=100000;i<100000,j>0;i++,j--)
    {
        v[i]=j;
    }

    t = clock();

    insertionSort(v,100000);

    t = clock() - t;

    printf("Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);

    for(i=0,j=100000;i<100000,j>0;i++,j--)
    {
        v[i]=j;
    }

    t = clock();

    selectionSort(v,100000);

    t = clock() - t;

    printf("Tempo de execucao em MILESSEGUNDOS SELECTION SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);

    for(i=0,j=100000;i<100000,j>0;i++,j--)
    {
        v[i]=j;}

    t = clock();

    bubble_sort(v,100000);

    t = clock() - t;

    printf("Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: %f\n",
((float)t)/(CLOCKS_PER_SEC/1000);

    return 0;
}

```

Fonte: Geilso Faria Rodrigues da Silva

1ª execução

```
"C:\Users\geils\OneDrive\ALG2\P2\P2 do 100000.exe"  
Tempo de execucao em MILESSEGUNDOS MERGE SORT: 108.000000  
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 15.000000  
Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: 25494.000000  
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 21396.000000  
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 33851.000000  
  
Process returned 0 (0x0)   execution time : 81.369 s  
Press any key to continue.
```

Fonte:Geilso Faria Rodrigues da Silva

2ª execução

```
"C:\Users\geils\OneDrive\ALG2\P2\P2 do 100000.exe"  
Tempo de execucao em MILESSEGUNDOS MERGE SORT: 114.000000  
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 17.000000  
Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: 25408.000000  
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 20996.000000  
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 35130.000000  
  
Process returned 0 (0x0)   execution time : 82.907 s  
Press any key to continue.
```

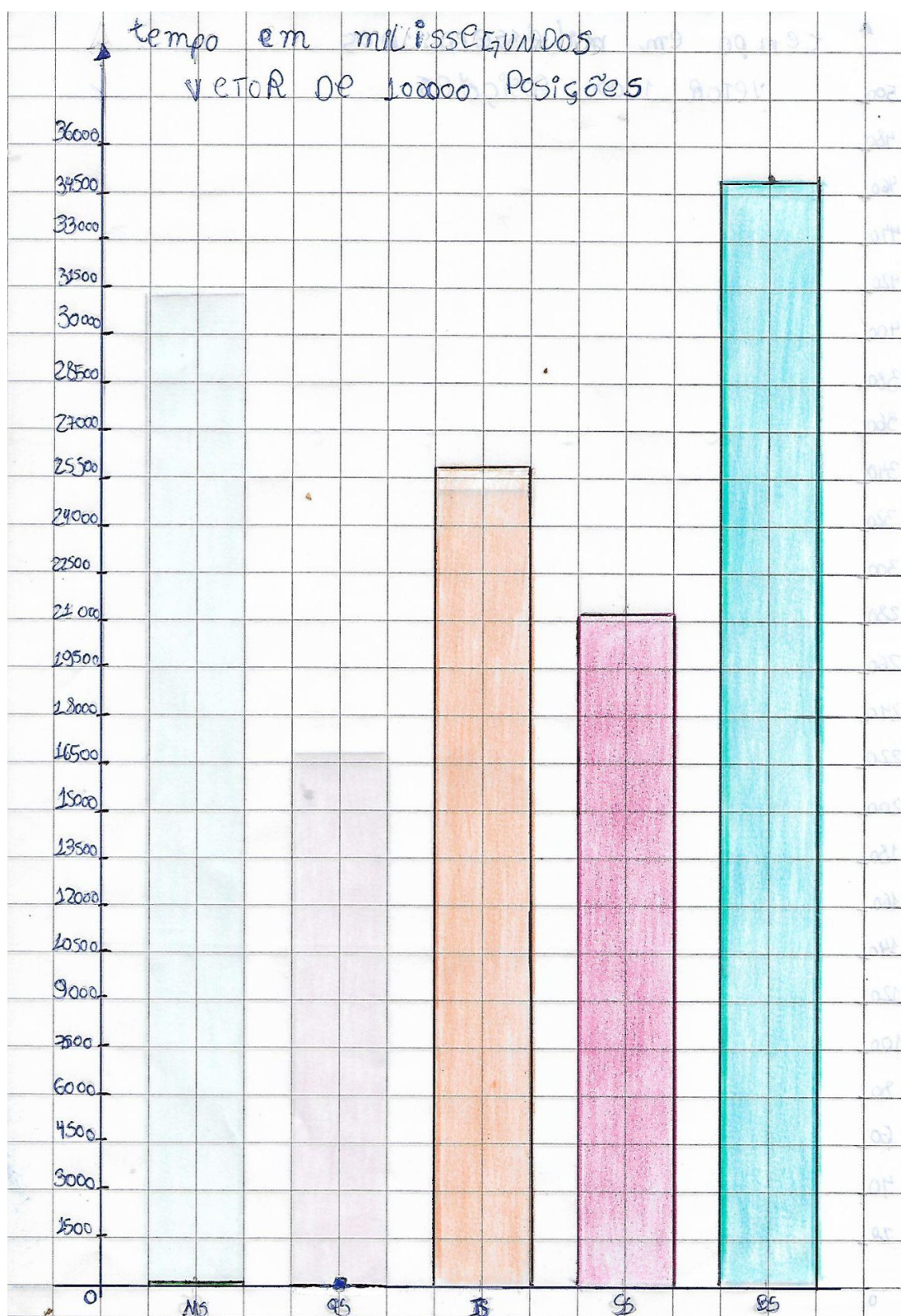
Fonte:Geilso Faria Rodrigues da Silva

3ª execução

```
"C:\Users\geils\OneDrive\ALG2\P2\P2 do 100000.exe"  
Tempo de execucao em MILESSEGUNDOS MERGE SORT: 97.000000  
Tempo de execucao em MILESSEGUNDOS QUICK SORT: 1.000000  
Tempo de execucao em MILESSEGUNDOS INSERCTION SORT: 25898.000000  
Tempo de execucao em MILESSEGUNDOS SELECTION SORT: 22319.000000  
Tempo de execucao em MILESSEGUNDOS BUBBLE SORT: 35817.000000  
  
Process returned 0 (0x0)   execution time : 85.551 s  
Press any key to continue.
```

Fonte:Geilso Faria Rodrigues da Silva

Gráfico em função do tempo de execução



Fonte: Geilso Faria Rodrigues da Silva

3.0 - CONCLUSÃO

Dessa forma, é perceptível que a ideia de “dividir para conquistar” torna seus algoritmos de excelentes no quesito ordenação. Prova disso, é o método *quicksort*, que se destacou por ser o mais eficiente entre todos, fato esse que se deve a seu funcionamento com a seleção de um pivô e a divisão em duas metades. Bem como, o *mergesort*, que apesar de ser um dos algoritmos mais eficientes, apresentou um desempenho inferior quando comparado ao *quicksort*, devido a sua divisão minuciosa em pequenas partes que consome mais tempo. No entanto, este não deixa de ser um dos métodos mais eficazes.

Por outro lado, métodos apoiados puramente na comparação e troca de elementos apresentaram maiores tempos de compilação e execução. Exemplificando, o *insertion sort*, apresentando o segundo maior tempo, visto que este realiza enormes volumes de comparações e trocas entre os elementos. Ademais, o *selection sort*, com o terceiro maior tempo, este que podemos dizer ser uma versão melhorada do método de ordenação por inserção, fundamentado na comparação e inserção direta de elementos, evitando em partes as quantidades de trocas e comparações que ocorrem no *insertion sort*.

E finalmente, podemos ver que o algoritmo *bubble sort*, apesar de ter fácil implementação, faz jus ao apelido que lhe é dado “demônio das trocas”, sendo o primeiro maior tempo dentre os cinco testados, com seu altíssimo número de comparações o que lhe torna inviável sua aplicação em vetores maiores.

Além disso, para vetores maiores foi possível notar claramente a diferença de tempo de algoritmos mais eficientes como *mergesort* e *quicksort*, que nos poupam grandes quantidades de tempo e recursos computacionais. Porém, em algoritmos como *insertion sort*, *selection sort* e *bubble sort* a quantidade de recursos exigidos foi amplamente maior, aumentando consideravelmente seus tempos médios.

Por outro lado, para vetores menores, curiosamente a diferença de tempo foi nula ou quase nula, sendo assim não importando qual método era implementado, que pode ser explicado pelas altas velocidades atuais de tempo de processamento, cada vez maiores.

Portanto, com este experimento é notável a importância da ordenação dos dados. Ora, o impacto negativo que algoritmos não viáveis causam no tempo de execução gasto, além de um maior consumo de recursos da máquina, e em casos de planejamentos mal

feitos até mesmo recursos financeiros. Ora, os benefícios de algoritmos mais ágeis, proporcionando menores tempos de processamentos e consumo de ferramentas. Logo, com o crescente volume de dados digitais existentes atualmente, o processo de ordenação se mostra indispensável.

4.0-Bibliografia

8 maneiras de medir o tempo de execução em C / C ++. Ichi Pro. [s.d]. Disponível

em:<<https://ichi.pro/pt/8-maneiras-de-medir-o-tempo-de-execucao-em-c-c-79591185633529>>.Acesso em 13/11/2021.

Gantz,John e Reinsel,David. The Digital Universe in 2020 :Big Data,Bigger Digital Shadows , and Biggest Growth in the Far East. EMC Corporation. Disponível

em:<<https://docplayer.net/14785-Executive-summary-a-universe-of-opportunities-and-challenges.html>>.Acesso em 13/11/2021.

Material Complementa.Linguagem C descomplicada,[s.d].Busca e Ordenação. Disponível

em:<<https://programacaodescomplicada.wordpress.com/complementar>>.Acesso em 13/11/2021.

SCHILDT, Herbert. C Completo e Total. 3ª edição, Ed. Makron Books,1997.

SOUZA, Jackson; RICARTE, João; LIMA, Náthalee. Algoritmos de ordenação: Um estudo comparativo.Universidade Federal Rural do Semi-Árido, Curso de Ciência e Tecnologia,2015. Disponível

em:<<https://periodicos.ufersa.edu.br/index.php/ecop/article/view/7082>>. Acesso em 13/11/2021.