SERVIÇO PÚBLICO FEDERAL · MINISTÉRIO DA EDUCAÇÃO UNIVERSIDADE FEDERAL DE VIÇOSA · UFV CAMPUS FLORESTAL

Trabalho 3 - AEDS 1

Análise do desempenho de diferentes variações do algoritmo de ordenação QuickSort

Miguel Antônio Ribeiro e Silva [EF04680]
Alan Gabriel Martins Silva [EF04663]
Vitor Vasconcelos de Melo Pontes [EF04255]

Florestal - MG 2022

Sumário

1. Introdução

O projeto consiste em analisar diversas variações do algoritmo de ordenação QuickSort. Essa análise compara os algoritmos considerando três métricas de desempenho: número de comparações de chaves, o número de cópias de registros realizadas, e o tempo total gasto para ordenação. As variações do Quicksort implementadas e analisadas são:

- **Quicksort Recursivo**: este é o Quicksort recursivo apresentado em sala de aula[4].

- **Quicksort Mediana(k)**: esta variação do Quicksort recursivo escolhe o pivô para partição como sendo a mediana de k elementos do vetor, aleatoriamente escolhidos. Utilize k = 3 e k = 5.

- **Quicksort Insercao(m)**: esta variação modifica o Quicksort Recursivo para utilizar o algoritmo de inserção para ordenar partições (isto é, pedaços do vetor) com tamanho menor ou igual a m. Experimente com m = 10 e m = 100.

- **Quicksort Empilha Inteligente()**: esta variação otimizada do QuicksortRecursivo processa primeiro o lado menor da partição.

- **Quicksort Iterativo**: esta variação escolhe o pivô como o elemento do meio (como apresentado em sala de aula), mas não é recursiva. Em outras palavras, esta é uma versão iterativa do Quicksort apresentado em sala de aula.

Realizaram-se experimentos com as cinco variações, considerando vetores aleatoriamente gerados, com tamanho N = 1000, 5000, 10000, 50000, 100000, 500000 e 1000000 e para cada valor de N, utilizamos 5 sementes diferentes a fim avaliar os valores médios do tempo de execução, do número de comparações de chaves e do número de cópias de registros. Para cada semente, as medições foram devidamente armazenadas em um arquivo de saída, passado, juntamente com a semente, pela linha de comando.

2. Organização

Na Figura 1 é possível visualizar a organização do projeto. Na pasta **Algoritmos/** está a implementação do projeto, separada em módulos. Cada variação do QuickSort recebeu um arquivo .h (contendo o protótipo das funções) e um .c (desenvolvimento das funções).

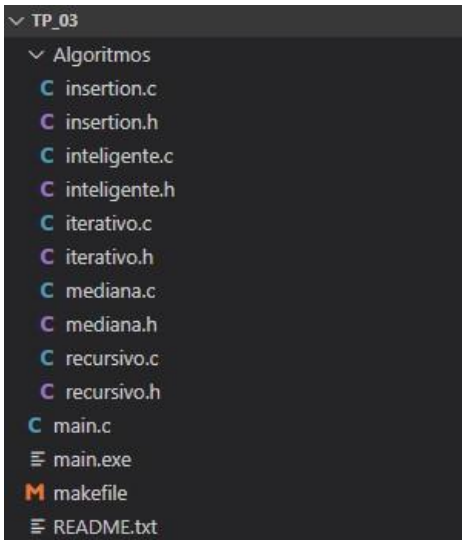


Figura 1 - Repositório do projeto.

Para executar o projeto, foi utilizado um arquivo **Makefile** com os comandos necessários para compilar os códigos. O arquivo **README.txt** detalha os comandos necessários para executar o programa da maneira correta. A função **main** contém a chamada das bibliotecas referentes a cada variação de QuickSort, declaração de variáveis necessárias para os testes e a realização e análise dos mesmos.

3. Desenvolvimento

Primeiramente, implementamos as variações do QuickSort na linguagem C seguindo à risca, todos os passos descritos no trabalho e exemplificados na Introdução. Todos os algoritmos descritos a seguir, foram implementados baseando-se na literatura de referência da disciplina (**Ziviani, N., "Projeto de Algoritmos"**)[1], e no material didático da disciplina.

3.1 QuickSort Recursivo:

A versão mais tradicional do QuickSort, como o nome descreve, funciona recursivamente, e escolhe o pivô como elemento do meio. A função QuickSort_Recursivo () passa como parâmetro o vetor A, aleatoriamente gerado, o valor inteiro N e o vetor B, onde é armazenado o número de comparações, cópias e o tempo, medido em ms. As funções utilizadas no algoritmo estão especificadas no arquivo de cabeçalho **recursivo.h** e implementadas no arquivo **recursivo.c**.

3.2 QuickSort Mediana(k):

Esta variação do Quicksort recursivo escolhe o pivô para partição como sendo a mediana de k elementos do vetor, aleatoriamente escolhidos. A escolha do pivô foi feita da seguinte maneira: na função Particao_Mediana(), foi declarado o vetor de inteiros **vet_median**, este vetor ficará responsável por armazenar os elementos, aleatoriamente escolhidos, do **vetor A**, que queremos ordenar, em cada partição. Após escolher os elementos, esse vetor será ordenado usando o algoritmo **Insertion Sort**, e depois, será escolhida a mediana desse vetor (**vet_median**) como sendo o pivô desta partição, exemplificado, na **figura 1**. O restante do algoritmo, funciona de forma similar ao **QuickSort Recursivo**. A função QuickSort_Mediana() passa como parâmetro o vetor A, gerado aleatoriamente, o valor N, o valor k (k=3 ou k=5), e o vetor B, onde será armazenado o número de comparações, cópias e o tempo. As funções utilizadas no algoritmo estão especificadas no arquivo de cabeçalho **mediana.h** e implementadas no arquivo **mediana.c**.

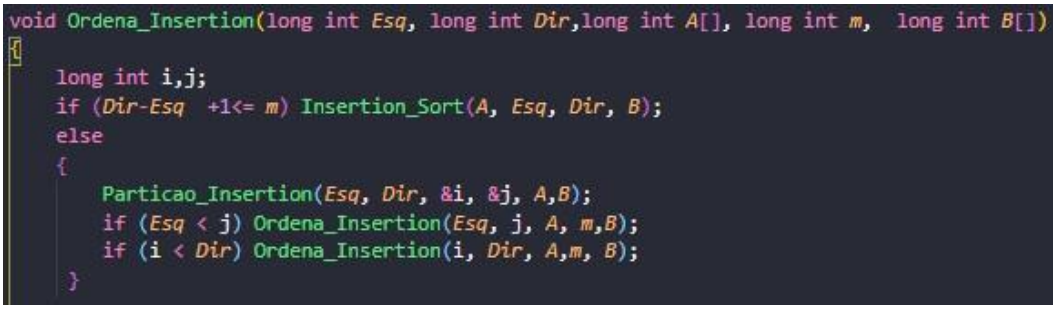
```
long int *vet_median; //declara o vetor
vet_median = (long int *) malloc(sizeof(long int) * k); //aloca dinamicamente
*i = Esq; *j = Dir;
for(long int c=0; c<k; c++){
    num = (rand() % (Dir - Esq + 1)) + Esq ; //gera os elementos aleatórios de
    vet_median[c] = A[num];
}
for(long int p=1; p<k; p++){ //ordena o vet_median
    aux = vet_median[p];
    long int h = p-1;

    while((h >= 0) && (aux < vet_median[h])){
        vet_median[h+1] = vet_median[h];
        h--;
    }
vet_median[h+1]  = aux;
}
long int f = (k / 2); //calcula a mediana
pivo = vet_median[f]; //escolhe o pivô
```

Figura 1

3.3 QuickSort Insercao(m):

Esta variação modifica o **Quicksort Recursivo** para utilizar o algoritmo de inserção para ordenar partições (isto é, pedaços do vetor) com tamanho **menor ou igual a m**. Ele funciona da seguinte forma: na função Ordena_Insertion(), o algoritmo verifica se esse “pedaço” tem tamanho menor ou igual a m. Se a verificação for contemplada, esta chama a função Insertion_Sort(), que utiliza o método de inserção para ordenar (**Figura 2**). O restante do algoritmo, funciona de forma similar ao **QuickSort Recursivo**. A função QuickSort_Insercao(), passa como parâmetro o vetor A, o valor N, o valor m (m=10 ou m=100) e o vetor B. As funções utilizadas no algoritmo estão especificadas no arquivo de cabeçalho **insertion.h** e implementadas no arquivo **insertion.c**.

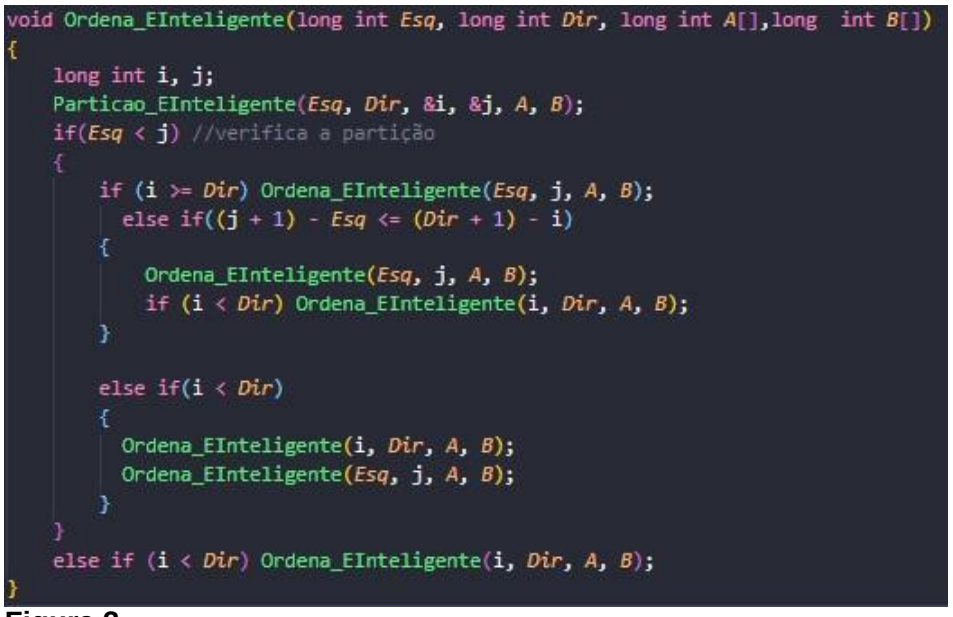


```
void Ordena_Insertion(long int Esq, long int Dir,long int A[], long int m,  long int B[])
{
    long int i,j;
    if (Dir-Esq  +1<= m) Insertion_Sort(A, Esq, Dir, B);
    else
    {
        Particao_Insertion(Esq, Dir, &i, &j, A,B);
        if (Esq < j) Ordena_Insertion(Esq, j, A, m,B);
        if (i < Dir) Ordena_Insertion(i, Dir, A,m, B);
    }
```

Figura 2

3.4 QuickSort Empilha Inteligente():

Esta variação modifica o **QuickSort Recursivo**, a fim de processar primeiro o lado menor da partição. Ele funciona da seguinte forma: na função Ordena_EInteligente(), o algoritmo verifica qual o lado menor da partição, e o ordena primeiro, recursivamente (**Figura 3**). Após completar a ordenação, ele executa o lado maior, e assim, sucessivamente, até ordenar completamente o vetor. O restante do algoritmo, funciona de forma similar ao **QuickSort Recursivo**. A função QuickSort_EInteligente(), passa como parâmetro o vetor A, o valor N e o vetor B. As funções utilizadas no algoritmo estão especificadas no arquivo de cabeçalho **inteligente.h** e implementadas no arquivo **inteligente.c**.



```
void Ordena_EInteligente(long int Esq, long int Dir, long int A[],long  int B[])
{
    long int i, j;
    Particao_EInteligente(Esq, Dir, &i, &j, A, B);
    if(Esq < j) //verifica a partição
    {
        if (i >= Dir) Ordena_EInteligente(Esq, j, A, B);
          else if((j + 1) - Esq <= (Dir + 1) - i)
        {
            Ordena_EInteligente(Esq, j, A, B);
            if (i < Dir) Ordena_EInteligente(i, Dir, A, B);
        }

        else if(i < Dir)
        {
          Ordena_EInteligente(i, Dir, A, B);
          Ordena_EInteligente(Esq, j, A, B);
        }
    }
    else if (i < Dir) Ordena_EInteligente(i, Dir, A, B);
}
```

Figura 3

3.5 QuickSort Iterativo:

Esta variação modifica o **QuickSort Recursivo**, a fim de torna-lo iterativo, da seguinte forma: ele usa uma pilha para iterar os marcadores **Esq e Dir** e o **pivô** da partição. A variável **topo_pilha** é iniciada com –1 e a pilha, na posição pilha[topo_pilha++] recebe **Esq e Dir**. Após isso, um loop while é chamado e enquanto **topo_pilha** for igual ou maior que zero, ele executa a função Particao_Iterativo(), ordenando o vetor e os marcadores Esq e Dir (**Figura 4**)[2]. O restante do algoritmo, incluindo a função Particao_Iterativo(), funciona de forma similar ao **QuickSort Recursivo**. A função QuickSort_Iterativo(), passa como parâmetro o vetor A, o valor N e o vetor B. As funções utilizadas no algoritmo estão especificadas no arquivo de cabeçalho **iterativo.h** e implementadas no arquivo **iterativo.c**.

```c
void Ordena_Iterativo(long int A[], long int Esq, long int Dir, long int B[])
{
    long int i, j;
    long int* pilha; //usa uma pilha para ordenar os elementos
    pilha = (long int*) malloc (Dir - Esq + 1 * sizeof(long int)); //aloca a pilha
    long int topo_pilha= -1; //define topo_pilha = -1
    pilha[++topo_pilha] = Esq;//++topo_pilha recebe esq e dir
    pilha[++topo_pilha] = Dir;
    while (topo_pilha >= 0) { //enquanto o topo pilha não for 0, ele executa a função partição, ordenando
        Dir = pilha[topo_pilha--];
        Esq = pilha[topo_pilha--];
        Particao_Iterativo(A, Esq, Dir, &i, &j, B);
        if (j > Esq) {
            pilha[++topo_pilha] = Esq; //aqui é onde ele "itera" e ordena
            pilha[++topo_pilha] = j;
        }
        if (i < Dir) {
            pilha[++topo_pilha] = i;
            pilha[++topo_pilha] = Dir;
        }
    }
```

Figura 4

3.6 main():

A função **main** é o ponto de partida para a execução do programa. Na linha de comando, no momento da execução, são passados o valor da semente e o nome do arquivo de saída, onde serão armazenadas as análises de cada algoritmo[3]. Primeiramente, é criado o **vetor B**, onde serão armazenadas as contagens, de tempo, comparações e cópias, em seguida, o **vetor A**, que será ordenado, é alocado e as variáveis, definidas. Após isso, um loop for executa 7 vezes, uma vez para cada valor de N e faz as análises de todos os algoritmos descritos acima.

4. Resultados

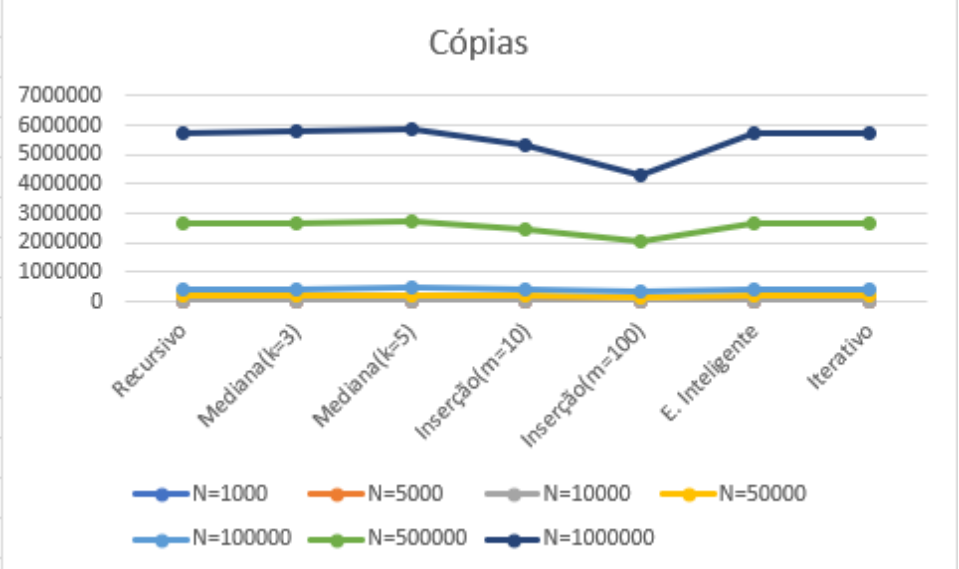
Em cada um dos algoritmos analisados**,** as comparações e movimentações foram computadas da seguinte forma: a cada iteração **do-while**, na função **Particao()**, foi contabilizada uma movimentação, as comparações, por sua vez, foram contadas conforme os elementos do vetor forem comparados entre si, na mesma função. No algoritmo **QuickSort Insercao(m)**, além das contagens descritas anteriormente, contabiliza as movimentações e comparações quando a parte a ser ordenada é menor que m, porém usando o algoritmo **Insertion Sort**. Já o algoritmo **QuickSort Mediana(k)**, apesar de usar o algoritmo **Insertion Sort** para definir o pivô, as contagens foram contabilizadas apenas na iteração **do-while**, para uma análise mais precisa, pois os cálculos (movimentações e comparações) na escolha do pivô, não impactaria tanto no resultado final. O tempo gasto na execução de cada algoritmo, foi medido em ms[4].

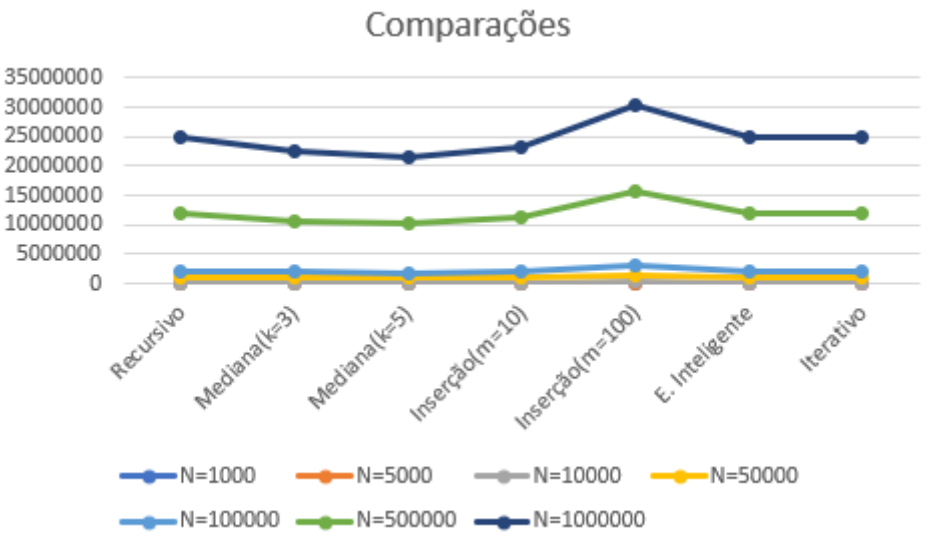
Após executar o programa cinco vezes, com cinco sementes diferentes, usamos o Excel para processar os valores obtidos em cada arquivo de saída, inserimos os resultados em uma tabela, para cada algoritmo QuickSort implementado, e calculamos a média para cada valor de N. Os resultados obtidos estão descritos nos gráficos e tabelas a seguir, gerados pelo Excel, após concluir os cálculos.

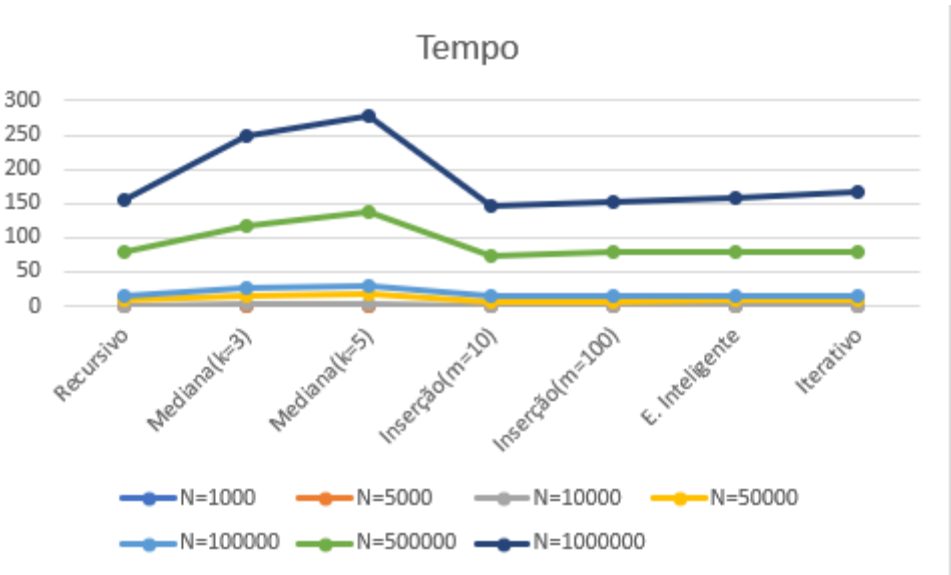
Tempo							
	Recursivo	Mediana(k=3)	Mediana(k=5)	Inserção(m=10)	Inserção(m=100	E. Inteligente	Iterativo
N=1000	0,2	0	0,2	0,2	0	0,6	0
N=5000	0,4	1,2	1,6	0,8	0,2	0,8	1
N=10000	1,2	3,4	3,4	1,2	1,4	1,4	1,6
N=50000	8	14,2	16,8	7	7,2	8	7,6
N=100000	16	26,6	30,4	13,4	14	14,8	15,4
N=500000	79,4	118	137	73,2	79,2	77,8	78
N=1000000	156,6	247,6	277	146,2	152,4	157,2	167,4

Comparações							
	Recursivo	Mediana(k=3)	Mediana(k=5)	Inserção(m=10)	Inserção(m=100	E. Inteligente	Iterativo
N=1000	12781,4	11800	11366,8	11715,2	22496	12781,4	12781,4
N=5000	81481,6	73028,6	70375,8	76268,8	131161	81481,6	81481,6
N=10000	173532,2	158152,8	151654,6	162954,6	270455,6	173532,2	173532,2
N=50000	1027504,4	921470,8	882034,2	974265	1507630	1027504,4	1027504,4
N=100000	2171408,8	1957409,4	1887134,8	2059891,8	3136194,8	2171408,8	2171408,8
N=500000	11974494	10646159,8	10255052,8	11276105	15870919,6	11974494	11974494
N=1000000	24899034,6	22461779,4	21472693,4	23387229,4	30363115,4	24899034,6	24899034,6

Cópias							
	Recursivo	Mediana(k=3)	Mediana(k=5)	Inserção(m=10)	Inserção(m=100	E. Inteligente	Iterativo
N=1000	2684,8	2780,2	2804,6	2569,4	1920,2	2684,8	2684,8
N=5000	16100,8	16649	16781,6	15495,6	12253,8	16100,8	16100,8
N=10000	34730,6	35717,6	35988,2	33479,4	27037,2	34730,6	34730,6
N=50000	201680	207854	209820,8	194091	161319	201680	201680
N=100000	436127,8	447298	451186,2	415393,8	345770,4	436127,8	436127,8
N=500000	2643809	2697113,2	2720218,8	2476716,6	2018931,8	2643809	2643809
N=1000000	5728289,4	5831268,8	5880075,4	5355119,8	4310082,4	5728289,4	5728289,4


Cópias
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
Recursivo
Mediana(k=3)
Mediana(k=5)
Inserção(m=10)
Inserção(m=100)
E. Inteligente
Iterativo
N=1000
N=5000
N=10000
N=50000
N=100000
N=500000
N=1000000


Comparações
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
Recursivo
Mediana(k=3)
Mediana(k=5)
Inserção(m=10)
Inserção(m=100)
E. Inteligente
Iterativo
N=1000
N=5000
N=10000
N=50000
N=100000
N=500000
N=1000000


Tempo
300
250
200
150
100
50
0
Recursivo
Mediana(k=3)
Mediana(k=5)
Inserção(m=10)
Inserção(m=100)
E. Inteligente
Iterativo
N=1000
N=5000
N=10000
N=50000
N=100000
N=500000
N=1000000

5. Conclusão:

Após a realização dos testes, para todos os algoritmos, à medida que N vai aumentando:

- Foi constatado que o algoritmo **QuickSort Insercao(m=10)** desempenhou melhor em relação ao tempo, pois para pedaços do vetor com tamanho menor ou igual a m, ele ordena usando o algoritmo **Insertion Sort,** que para valores menores, desempenha melhor que o **QuickSort Recursivo**. Já os **QuickSort Mediana(k)** obtiveram os piores tempos. Isso ocorre pois o algoritmo além de escolher aleatoriamente os elementos do vetor ele precisa ordena-los para obter a mediana que será o pivô para a partição.

- Percebemos que o algoritmo **QuickSort Insercao(m=100)** obteve mais comparações e menos cópias em relação aos outros algoritmos, devido ao uso do algoritmo **Insertion Sort** para ordenar pedaços do vetor de tamanho menor ou igual a 100. Já o restante dos algoritmos não tiveram muitas discrepâncias entre si em relação a comparações e cópias.

- O algoritmo **QuickSort Mediana(k=3)** desempenhou melhor em relação ao **QuickSort Mediana(k=5)**, pois o tempo gasto para a execução foi menor, devido a escolha do pivô para a partição. Já o número de comparações e cópias foram praticamente os mesmos quando comparados entre si.

- Os algoritmos **QuickSort Inserção(m)** tiveram desempenhos similares em relação ao tempo. Apesar disso, o **QuickSort Inserção(m=100)** faz mais comparações que o **QuickSort Inserção(m=10)**. Em questão de cópias, ambos os algoritmos apresentam resultados abaixo da média, quando comparados com o restante das variações, com o **QuickSort Inserção(m=100)** desempenhando melhor.

  Em contrapartida, para valores de N menores que 10000, todos os algoritmos desempenham de forma bastante similar. Por isso julgamos mais interessante a analise de desempenho dos algoritmos com valores de N maiores.

6. Referências:

[1] Ziviani, N., "Projeto de Algoritmos."

[2] Disponível em: https://stackoverflow.com/questions/12553238/quicksort-iterative-or-recursive Último acesso em: 15 de março de 2022

[3] Disponível em: https://www.thegeekstuff.com/2013/01/c-argc-argv/ Último acesso em 21 de março 2022

[4] Disponível em: https://www.clubedohardware.com.br/forums/topic/1031279-resolvido-medir-tempo-de-execu%C3%A7%C3%A3o-em-c/ Último acesso em: 17 de março 2022