UNIVERSIDADE FEDERAL DE VIÇOSA
CAMPUS DE RIO PARANAÍBA
SISTEMAS DE INFORMAÇÃO

GLEIDSON VINÍCIUS GOMES BARBOSA – 6331

ANÁLISE DE ALGORITMOS DE ORDENAÇÃO - INSERTION SORT, MERGE SORT E QUICK SORT

RIO PARANAÍBA
19 de novembro de 2022

GLEIDSON VINÍCIUS GOMES BARBOSA – 6331

ANÁLISE DE ALGORITMOS DE ORDENAÇÃO - INSERTION SORT, MERGE SORT E QUICK SORT

Trabalho apresentado para obtenção de créditos na disciplina SIN213 - Projeto de Algoritmos da Universidade Federal de Viçosa - Campus de Rio Paraníba, ministrada pelo Professor Pedro Moisés de Souza.

Orientador: Pedro Moisés de Souza

RIO PARANAÍBA

19 de novembro de 2022

Resumo

Os algoritmos de ordenação apresentam grande importância no estudo teórico de algoritmos, bem como em aplicações rotineiras e práticas do cotidiano. Diante dessa grande importância ao longo dos anos surgem alguns algoritmos de ordenação na literatura, onde são analisados casos em que cada algoritmo se enquadra. Sendo assim se torna necessário um estudo para analisar o comportamento desses algoritmos, bem como suas complexidades para indicar qual é o mais adequado para cada tipo de problema. Neste trabalho serão estudados os algoritmos Insertion Sort, Merge Sort e Quick Sort com três quatro abordagens para seleção de seu pivô, onde são realizados testes de e analisadas suas complexidades com base na teoria apresentada em sala de aula e vistas em livros texto além de ferramentas encontradas na internet.

Palavras-chaves: Insertion Sort, Merge Sort, Quick Sort, algoritmos de ordenação, Projeto de Algoritmos, Divisão e Conquista.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Neste trabalho, objetiva-se analisar os algoritmos Insertion Sort, Merge Sort e Quick Sort com três diferentes abordagens para seleção de pivô realizando testes com entradas para melhor caso, pior caso e caso médio e comparar seus desempenhos analisando sua complexidade e observar se os resultados práticos condizem com a teoria apresentada. Será comparado um algoritmo em "lento"em teoria e será feita a comparação com algoritmos que utilizam a abordagem de divisão e conquista para demonstrar a gigantesca diferença de desempenho. Apesar de existirem diversas ferramentas na web como (TOPTAL, 2022) e (USF, 2022) onde podemos ver um pouco da teoria além de um exemplo do funcionamento dos algoritmos, para este trabalho desenvolvemos um software em linguagem C conforme a teoria apresentada em sala de aula, vista também em (CORMEN et al., 2002) e no já citado Toptal. Neste software geramos as entradas compatíveis com cada caso para 10, 100, 1.000, 10.000, 100.000 e 1.000.000 de instâncias, executamos testes e gravamos os dados em arquivos, tanto as entradas quanto saída destes dados ordenados e o tempo de execução do algoritmo e analisamos os resultados com base na teoria.

2 Insertion Sort

2.1 Análise e Complexidade do Algoritmo

Conforme visto em sala de aula, o algoritmo usa é apresentado utilizando uma sequência A de n elementos, observamos isso na tabela a seguir:

Insertion Sort(A,n)	**Custo**	**Vezes**
1 for j <- 2 to comprimento	C1	n
2 do chave < - A[j]	C2	n-1
3 "Inserir A[j] na sequência ordenada A[1.2.....j-1]"	C3 = 0	n-1
4 i <- j-1	C4	n-1
5 while i>0 e A[i] >chave	C5	$\sum_{t=2}^{n} tj$
6 do A[i+1] <- A[i]	C6	$\sum_{t=2}^{n}(tj-1)$
7 i <- i-1	C7	$\sum_{t=2}^{n}(tj-1)$
8 A[i+1] <- chave	C8	n-1

Fonte: Algoritmo visto em sala de aula

Ao observar o algoritmo apresentado, nota-se que é possível calcular sua complexidade multiplicando o custo de cada linha de sua execução por pela quantidade de vezes que esta é executada, assim chegamos na seguinte equação:

$$C1n+C2(n-1)+C4(n+1)+C5[\sum_{t=02}^{n} tj]+C6[\sum_{t=02}^{n}(tj-1)]+C7[\sum_{t=02}^{n}(tj-1)]+C8(n-1) \quad (2.1)$$

Sendo tj o número de vezes que o loop foi executado.Observa-se que a linha 3 tem custo 0 visto que apenas apresenta um comentário. Para chegar ao melhor caso, pior caso e caso médio, basta desenvolver tal equação utilizando os seguintes valores:

2.1.1 Melhor Caso

$$tj = 1 \quad (2.2)$$

Neste caso teremos que o nosso tj = 1 e na linha 5 a condição do loop é falsa, logo as linhas 6 e 7 não são executadas. Assim ao desenvolver a equação da complexidade do algoritmo temos:

$$(C1 + C2 + C3 + C4 + C5 + C8)n - (C2 + C4 + C5 + C8) \quad (2.3)$$

que podemos simplificar para:

$$An - B \tag{2.4}$$

chegando assim a uma função linear de n que é o melhor caso, ou $\Omega(n)$.

2.1.2 Pior Caso

$$tj = j \tag{2.5}$$

Neste caso, o vetor inicial estará em ordem decrescente, o que nos fará entrar em nosso loop e comparar cada elemento do nosso vetor principal com cada elemento do subarranjo ordenado, neste caso teremos que o nosso tj = j para todo jogo de 2 até n. O que nos leva a desenvolver utilizando propriedades matemáticas os nossos somatórios, chegando a seguintes igualdades:

$$\sum_{t=02}^{n} tj = \sum_{t=2}^{n} j = \frac{n(1+n)}{2} - 1 \tag{2.6}$$

$$\sum_{t=02}^{n} (tj - 1) = \sum_{t=2}^{n} tj - \sum_{t=2}^{n} 1 = \frac{n(1+n)}{2} - 1 - (n-1) = \frac{n(n-1)}{2} \tag{2.7}$$

Assim, ao desenvolver a equação de complexidade do algoritmo aplicando esta condição temos:

$$(\frac{C5 + C6 + C7}{2})n^2 + (C1 + C2 + C4 + C8 + \frac{C5 - C6 - C7}{2})n + (-C2 - C4 - C5 - C8) \tag{2.8}$$

que podemos simplificar para:

$$An^2 + Bn + C \tag{2.9}$$

Chegando assim a uma função quadrática de n que é o pior caso, ou $\mathcal{O}(n^2)$.

2.1.3 Caso Médio

$$tj = \frac{j}{2} \tag{2.10}$$

No caso médio, assumimos que teremos nossa condição para entrar no loop cumprida em média 50% das vezes, logo temos tj = j/2, utilizando as propriedades matemáticas chegamos nas seguintes igualdades:

$$\sum_{t=02}^{n} tj = \sum_{t=2}^{n} \frac{j}{2} = \frac{1}{2}(\frac{n(1+n)}{2} - 1) = \frac{n(1+n)}{4} - \frac{1}{2} \tag{2.11}$$

$$\sum_{t=02}^{n}(tj-1)=\sum_{t=2}^{n}tj-\sum_{t=2}^{n}1=\frac{n(1+n)}{4}-\frac{1}{2}-(n-1)=\frac{n(1+n)}{4}-n+\frac{1}{2} \tag{2.12}$$

Assim, ao desenvolver a equação de complexidade do algoritmo aplicando esta condição temos:

$$(\frac{C5+C6+C7}{4})n^2+(C1+C2+C4+C8-C6-C7+\frac{C5-C6-C7}{4})n \tag{2.13}$$

$$+(-C2-C4-C5-C8-\frac{C5-C6-C7}{2}) \tag{2.14}$$

que podemos simplificar para:

$$An^2+Bn+C \tag{2.15}$$

Chegando assim a uma função quadrática de n que é o caso médio, observe que é igual ao nosso pior caso, assim observamos $\theta(n^2)$.

2.2 Tabela e Gráfico do Algoritmo

O Algoritmo foi implementado conforme apresentado em sala de aula, apenas com adaptações para a linguagem solicitada conforme a imagem a seguir:

```
void insertionSort(int *vetor, int tamanho)
{
    int i, j;
    for(i = 1; i < tamanho; i++)
    {
        int x = vetor[i];
        j = (i-1);
        while(j >= 0 && vetor[j] > x)
        {
            vetor[j+1] = vetor[j];
            vetor[j] = x;
            j--;
        }
    }
}
```

Figura 1 – Código Insertion Sort

Fonte: Autoria própria

Foram realizados testes para cada entrada que variam entre 10, 100, 1.000. 10.000, 100.000, 1.000.000 instâncias e são ordenados de três formas: Crescente, Decrescente e Aleatório. Para cada combinação de quantidades de instâncias e tipo ordenação foi realizada uma execução do algoritmo. Os resultados da ordenação e o tempo de execução são gravados em arquivos diferentes para que se possa observar o resultado da ordenação e o tempo de execução do algoritmo. Os resultados destes testes são apresentados na tabela e gráfico a seguir:

Tabela 1 – Insertion Sort: Tempos por tipo e tamanho das entradas.

	10	**100**	**1.000**	**10.000**	**100.000**	**1.000.000**
Aleatório	0.0140	0.0140	0.0160	0.0760	4.9940	562.4980
Crescente	0.0130	0.0150	0.0140	0.0150	0.0140	0.0140
Decrescente	0.0170	0.0140	0.0150	0.1310	9.8410	1084.3570

Fonte: Autoria própria

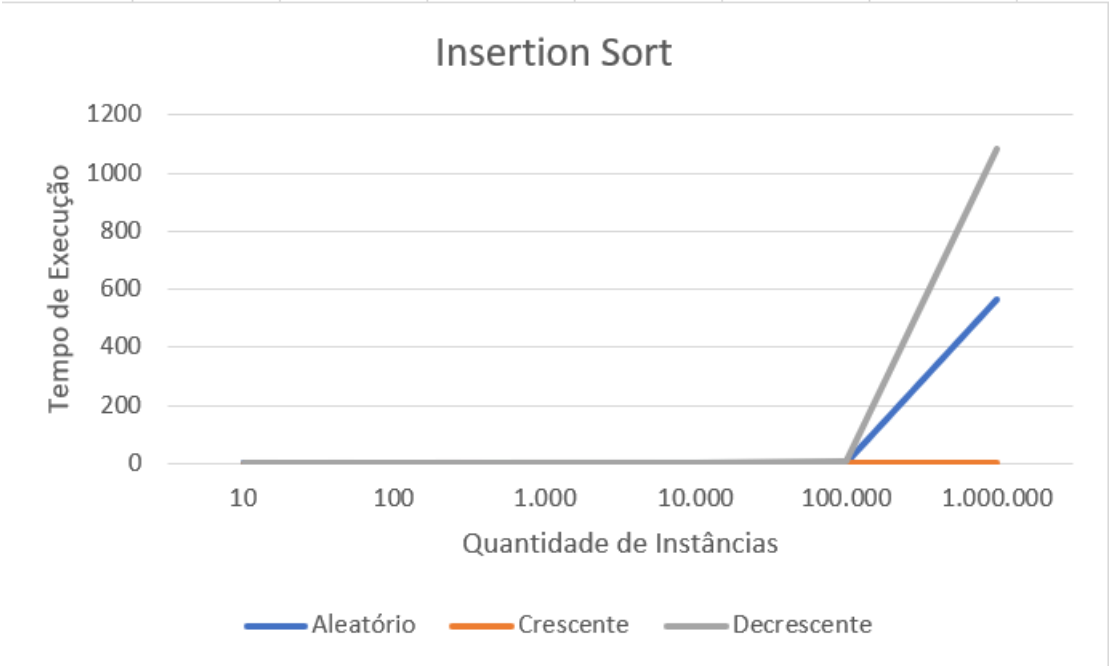


Figura 2 – Gráfico de tempos de execução Insertion Sort

Fonte: Autoria própria

Assim é possível observar que conforme a teoria nos propõe com a análise da complexidade, os resultados dos testes apresentam desempenho compatível com aquilo que foi proposto.

3 Merge Sort

3.1 Análise e Complexidade do Algoritmo

Conforme visto em sala de aula, o algoritmo usa uma abordagem de divisão e conquista e é apresentado utilizando uma sequência A de n elementos e ele é dividido em dois algoritmos, onde um é recursivo responsável por dividir a sequência e o outro é responsável por unir "Conquistar", observamos isso nas tabelas a seguir:

Merge Sort(A,n)		**t(n)**
1	Se p < r then $q = \frac{p+r}{2}$	$\mathcal{O}(1)$
2	Ordene Recursivamente	$t(\frac{n}{2}) + t(\frac{n}{2})$
3	Merge	$\mathcal{O}(n)$

Fonte: Algoritmo visto em sala de aula

Ao observar o algoritmo Merge Sort, nota-se que é possível calcular sua complexidade desenvolvendo um cálculo somando suas linhas:

$$t(n) = \mathcal{O}(1) + t(\frac{n}{2}) + t(\frac{n}{2}) + \mathcal{O}(n) \tag{3.1}$$

Ou

$$t(n) = 2t(\frac{n}{2}) + \mathcal{O}(n) \tag{3.2}$$

Observamos que esse cálculo resultou em uma recorrência, ao calculá-la é possível chegar a complexidade do referido algoritmo.

Merge		**Custo**	**Vezes**
1	while(i < q+1 and j < r+1)	C1	n
2	if(v[i] <= v[j])	C2	n-1
3	w[k++] = v[i++]	C3	n-1
4	else	C4	n-1
5	w[k++] = v[j++]	C5	n-1
6	while(i < q)	C6	n
7	w[k++] = v[i++]	C7	n-1
8	while(j < r)	C8	n
9	w[k++] = v[j++]	C9	n-1
10	for(i = p, i < r, ++i)	C10	n
11	v[i] = w[i-p]	C11	n-1

Fonte: Algoritmo visto em sala de aula

Ao observar o algoritmo Merge, nota-se que é possível calcular sua complexidade multiplicando o custo de cada linha de sua execução por pela quantidade de vezes que esta é executada, assim chegamos na seguinte equação:

$$C1n + C2(n-1) + C3(n-1) + C4(n-1) + C5(n-1) \tag{3.3}$$

$$+C6n + C7(n-1) + C8n + C7(n-1) + C10n + C11(n-1) \tag{3.4}$$

3.1.1 Melhor Caso, Pior Caso e Caso Médio

$$tj = j \tag{3.5}$$

Neste Algoritmo observa-se um caso interessante, teremos sempre uma recorrência, teremos que o Melhor Caso, Pior Caso e Caso Médio são iguais. A seguir é desenvolvida a recorrência proposta e a provada pelo método da indução:

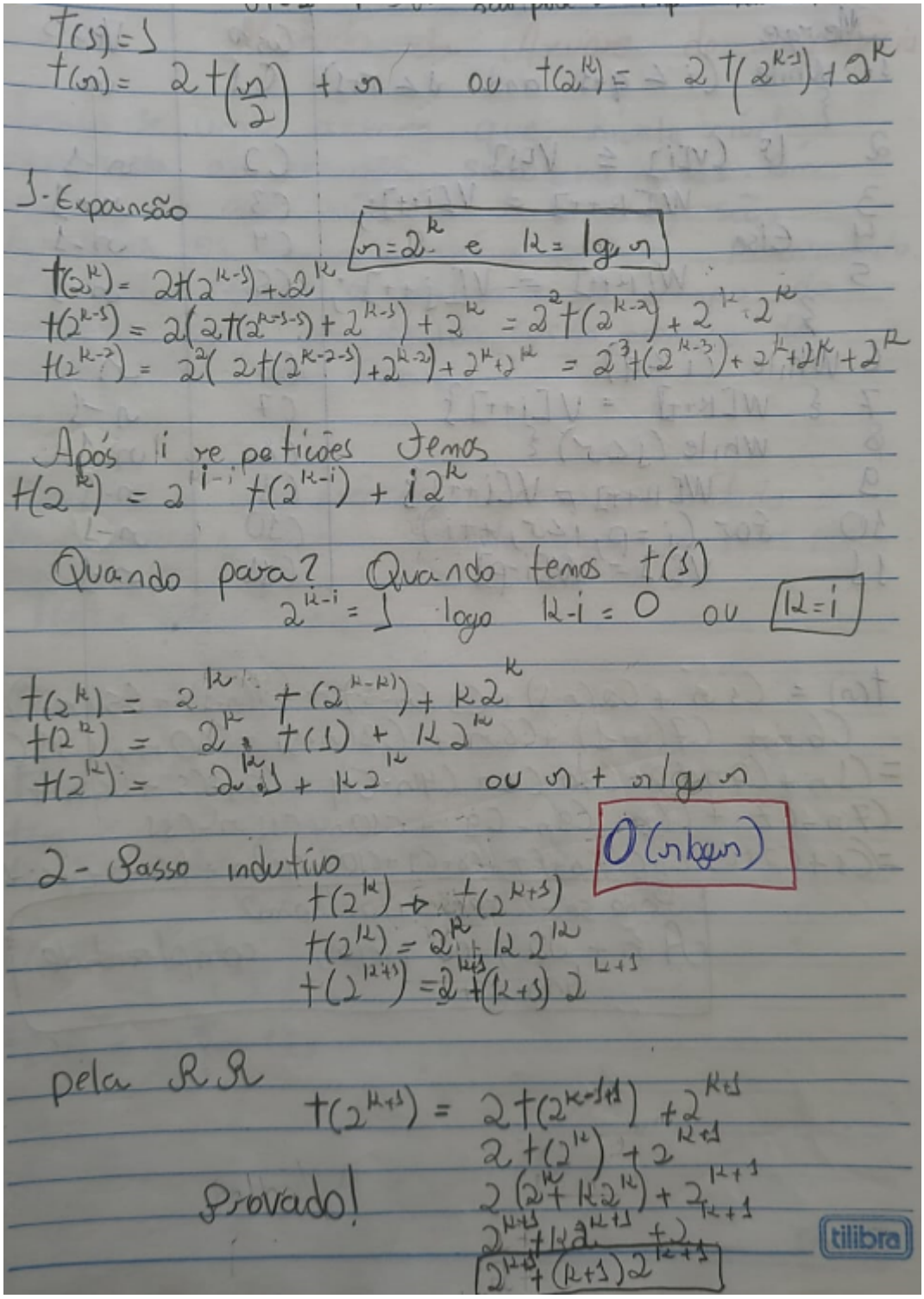


Figura 3 – Recorrência Merge Sort

Fonte: Autoria própria

Chegando assim em $n + n \log n$ uma complexidade linear logarítmica $\theta(n \log n)$.

3.2 Tabela e Gráfico do Algoritmo

O Algoritmo foi implementado conforme apresentado em sala de aula e seguindo também pseudo códigos já apresentados pelo professor Clausius Duque Reis em períodos passados, apenas com adaptações para a linguagem solicitada, ele foi dividido em três funções, sendo a função mergeSort a auxiliar para iniciar a divisão e esta por sua vez inicia a conquista conforme a imagens a seguir:

```c
//Merge Sort
void mergeSort(int *vetor, int n)
{
  int *c = malloc(sizeof(int) * n);
  dividir(vetor, c, 0, n - 1);
  free(c);
}

/*
  Dado um vetor de inteiros e dois inteiros i e f, ordena o vetor[i..f] em ordem crescente.
  O vetor c é utilizado internamente durante a ordenação.
*/
void dividir(int *vetor, int *c, int i, int f)
{
  if (i >= f)
  {
    return;
  }

  int m = (i + f) / 2;

  dividir(vetor, c, i, m);
  dividir(vetor, c, m + 1, f);

  /* Se vetor[m] <= vetor[m + 1], então vetor[i..f] já está ordenado. */
  if(vetor[m] <= vetor[m + 1])
  {
    return;
  }

  conquistar(vetor, c, i, m, f);
}
```

Figura 4 – Código da função mergeSort e Conquistar

Fonte: Autoria própria

```c
void conquistar(int *vetor, int *c, int i, int m, int f)
{
  int z, iv = i, ic = m + 1;

  for (z = i; z <= f; z++)
  {
    c[z] = vetor[z];
  }

  z = i;

  while (iv <= m && ic <= f)
  {
    /* Invariante: vetor[i..z] possui os valores de vetor[iv..m] e vetor[ic..f] em ordem crescente. */

    if (c[iv] <= c[ic])
    {
      vetor[z++] = c[iv++];
    }
    else
    {
      vetor[z++] = c[ic++];
    }
  }

  while (iv <= m)
  {

    vetor[z++] = c[iv++];
  }

  while (ic <= f)
  {
    vetor[z++] = c[ic++];
  }
}
```

Figura 5 – Código da função Conquistar

Fonte: Autoria própria

Foram realizados testes para cada entrada que variam entre 10, 100, 1.000. 10.000, 100.000, 1.000.000 instâncias e são ordenados de três formas: Crescente, Decrescente e Aleatório. Para cada combinação de quantidades de instâncias e tipo ordenação foi realizada uma execução do algoritmo. Os resultados da ordenação e o tempo de execução são gravados em arquivos diferentes para que se possa observar o resultado da ordenação e o tempo de execução do algoritmo. Os resultados destes testes são apresentados na tabela e gráfico a seguir:

Tabela 2 – Bubble Sort: Tempos por tipo e tamanho das entradas.

	10	100	1.000	10.000	100.000	1.000.000
Aleatório	0.0200	0.0210	0.0200	0.0300	0.0400	0.2540
Crescente	0.0210	0.0210	0.0220	0.0210	0.0220	0.0300
Decrescente	0.0200	0.0200	0.0220	0.0300	0.0300	0.1380

Fonte: Autoria própria

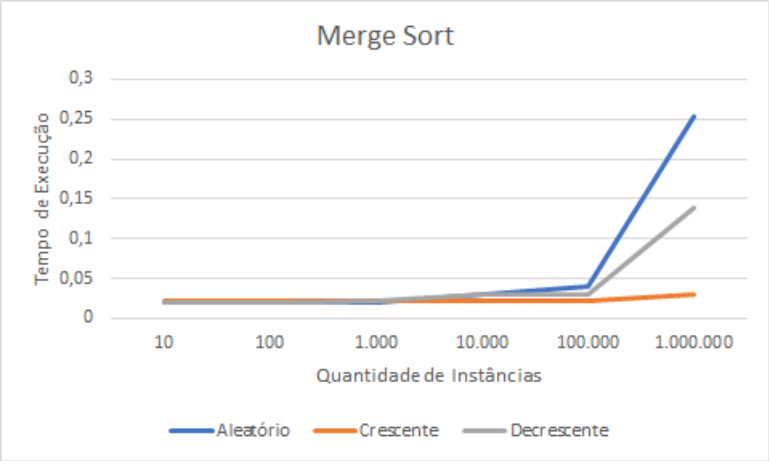


Figura 6 – Gráfico de tempos de execução Merge Sort

Fonte: Autoria própria

Assim é possível observar que conforme a teoria vista em sala de aula nos propõe com a análise da complexidade, os resultados dos testes apresentam desempenho compatível com aquilo que foi proposto, apesar de existirem pequenas diferenças, todos os casos levam tempos baixos, mantendo o padrão linear logarítmico, então temos $\theta(n \log n)$.

4 Quick Sort

O quick sort tem muitas semelhanças com o merge sort, é possível observar algumas dessas principalmente no melhor caso, isso ocorre por ambos os algoritmos se tratarem de algoritmos de divisão e conquista. Porém o quick sort pode variar conforme a escolha de seu pivô, então seu melhor pior e caso médio variam de acordo com a escolha do pivô. O algoritmo quickSort se manterá igual para todos os casos apresentados, apenas teremos variações na escolha dos pivôs.

4.1 Análise e Complexidade do Algoritmo

Quick Sort(A, p, r)	
1	If p < r-1 then q<-particiona(A,p,r)
2	quickSort(A, p, q-1)
3	quickSort(A, q+1, r)

Fonte: Algoritmo visto em sala de aula

Particiona		**Custo**	**Vezes**
1	x <- A[p]	C1	1
2	i <- A[p-1]	C2	1
3	j <- A[r-1]	C3	1
4	while true	C4	n
5	repeat i <- i+1	C5	n-1
6	until A[i] >= x	C6	n-1
7	repeat j <- j-1	C7	n-1
8	until A[j] <= x	C8	n-1
9	if i<j	C9	n-1
10	then troca A[i] <-> A[j]	C10	n-1
11	troca A[p] <-> A[j]	C11	n-1

Fonte: Algoritmo visto em sala de aula

Este algoritmo como já citado, também trabalha com divisão e conquista, assim vemos que tal qual merge sort tem dois algoritmos onde um trabalha com uma recorrência e outro de complexidade linear onde faz a união das partes dividas. Suas complexidades dependem da forma de escolher o pivô como veremos nas sessões seguintes.

4.1.1 Melhor Caso

Nosso melhor caso ocorre quando as partições são balanceadas, ou seja, tem o mesmo tamanho, assim temos sempre a metade das comparações em cada chamada, assim

podemos assumir uma recorrência tal qual a do Merge sorte, logo podemos resolver a mesma e chegaremos no mesmo melhor caso.

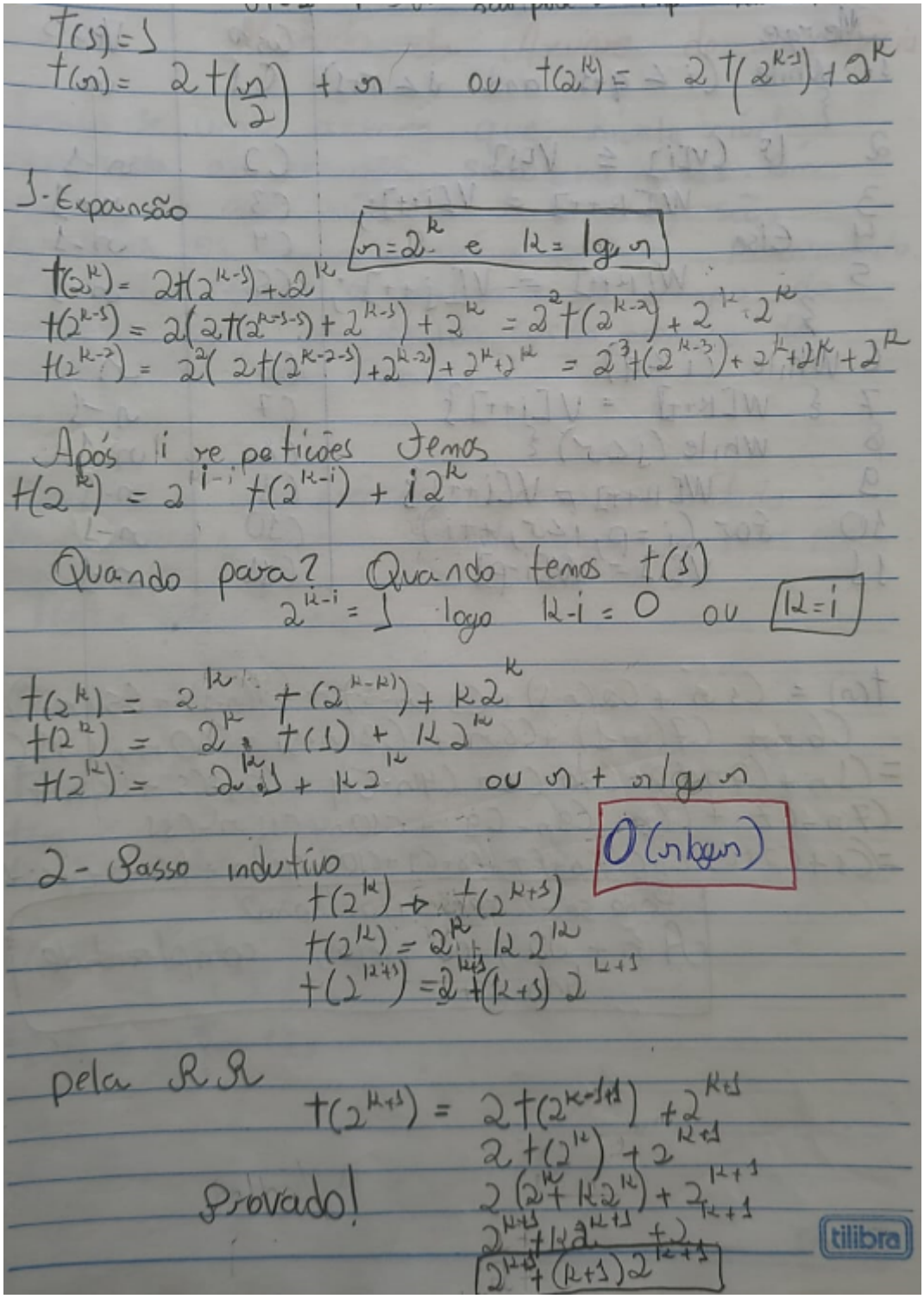


Figura 7 – Recorrência Merge Sort

Fonte: Autoria própria

Chegando assim em $n + n \log n$ uma complexidade linear logarítmica $\Omega(n \log n)$.

4.1.2 Pior Caso

Nosso pior caso ocorre quando a escolha do pivô é inadequada, como exemplo podemos citar uma escolha sistemática dos extremos de um arquivo já ordenado, onde ele será chamado recursivamente n vezes eliminando apenas um item em cada chamada, chegando assim em um pior caso quadrático, ou seja, $\mathcal{O}(n^2)$.

4.1.3 Caso Médio

Nosso caso médio é adotado com base em (SEDGEWICK; FLAJOLET, 1996) onde ele apresenta que o número de comparações é

$$t(n) = 1.386nlogn - 0.846n \tag{4.1}$$

logo observa-se um caso médio $\theta(nlogn)$.

4.2 Tabela e Gráfico do Algoritmo

Para todos os algoritmos utilizaremos uma função auxiliar para chamar o algoritmo, variando apenas o nome com base na seleção de seu pivô. Os códigos das funções padrão e com seleção de pivô por media foram baseadas nas funções apresentadas em (ZIVIANI et al., 2004) enquanto as funções com seleção de pivô aleatóriamente e por mediana de três foram baseadas em códigos apresentados por Henrique Felipe em (CYBERINI, 2022).

```
void quickSort(int *vetor,int n)
{
    ordena(vetor,0,n-1);
}
```

Figura 8 – Código Quick Sort

Fonte: Autoria própria

Para os métodos de seleção do pivô aleatório e por mediana também foi utilizada uma função auxiliar para troca dos vetores, a função swap.

```c
void swap(int vetor[], int i, int j)
{
    int temp = vetor[i];
    vetor[i] = vetor[j];
    vetor[j] = temp;
}
```

Figura 9 – Código função swap

Fonte: Autoria própria

A partir da função quickSort chamamos a função ordena, esta varia de acordo com cada método de seleção do pivô como vemos a seguir:

```c
void ordena(int *vetor, int esq, int dir)
{
    int i,j;
    particao(vetor,esq,dir,&i,&j);
    if(esq < j)
    {
        ordena(vetor,esq,j);
    }
    if(i< dir){
        ordena(vetor,i,dir);
    }
}
```

Figura 10 – Código função ordena para o método (ZIVIANI et al., 2004)

Fonte: Autoria própria

```
void ordenaMEDIA(int *vetor, int esq, int dir)
{
    int i,j;
    particaoMEDIA(vetor,esq,dir,&i,&j);
    if(esq < j)
    {
        ordenaMEDIA(vetor,esq,j);
    }
    if(i<dir)
    {
        ordenaMEDIA(vetor,i,dir);
    }
}
```

Figura 11 – Código função ordena para o método da média

Fonte: Autoria própria

```
void ordenaRANDOM(int vetor[], int esq, int dir)
{
    if (esq < dir)
    {
        int q = particaoRANDOM(vetor, esq, dir);
        ordenaRANDOM(vetor, esq, q - 1);
        ordenaRANDOM(vetor, q + 1, dir);
    }
}
```

Figura 12 – Código função ordena para o método aleatório

Fonte: Autoria própria

```
void ordenaMEDIANA(int vetor[], int esq, int dir)
{
    if (esq < dir)
    {
        int q = particaoMEDIANA(vetor, esq, dir);
        ordenaMEDIANA(vetor, esq, q - 1);
        ordenaMEDIANA(vetor, q + 1, dir);
    }
}
```

Figura 13 – Código função ordena para o método da mediana

Fonte: Autoria própria

Por fim, a função partição é onde se concentra as maiores diferenças entre os três métodos, pois nela ocorre a seleção do pivô e a ordenação.

```
void particao(int* vetor, int esq,int dir , int *i,int *j)
{
    int x,w;
    *i = esq;
    *j = dir;
    x = vetor[(*i + *j) / 2];
    do
    {
        while( x > vetor[*i])(*i)++;
        while( x < vetor[*j])(*j)--;
        if(*i <= *j)
        {
            w = vetor[*i];
            vetor[*i] = vetor[*j];
            vetor[*j] = w;
            (*i)++;
            (*j)--;
        }
    }while(*i <= *j);
}
```

Figura 14 – Código função particao para o método (ZIVIANI et al., 2004)

Fonte: Autoria própria

```c
void particaoMEDIA(int* vetor, int esq,int dir , int *i,int *j){
    int x,w;
    *i = esq;
    *j = dir;
    //Calculo de media
    int result,t,minpos=0,maxpos=0;
    int aux[3];
    aux[0]=*i+(rand()%(*j-*i));
    aux[1]=*i+(rand()%(*j-*i));
    aux[2]=*i+(rand()%(*j-*i));

    for(t=0;t<3;t++)
    {
        if(vetor[aux[t]]< vetor[minpos])
        {
            minpos=t;
        }
        if(vetor[aux[t]]>vetor[maxpos])
        {
            maxpos=t;
        }
    }


    for(t=0;t<3;t++)
    {
        if(t!= minpos && t != maxpos)
        {
            result =aux[t];
        }
    }

    x = vetor[result];
    do
    {
        while( x > vetor[*i])
            (*i)++;
        while( x < vetor[*j])
            (*j)--;
        if(*i <= *j){
            w = vetor[*i];
            vetor[*i] =vetor[*j];
            vetor[*j] = w;
            (*i)++;
            (*j)--;
        }
    }while(*i <= *j);
}
```

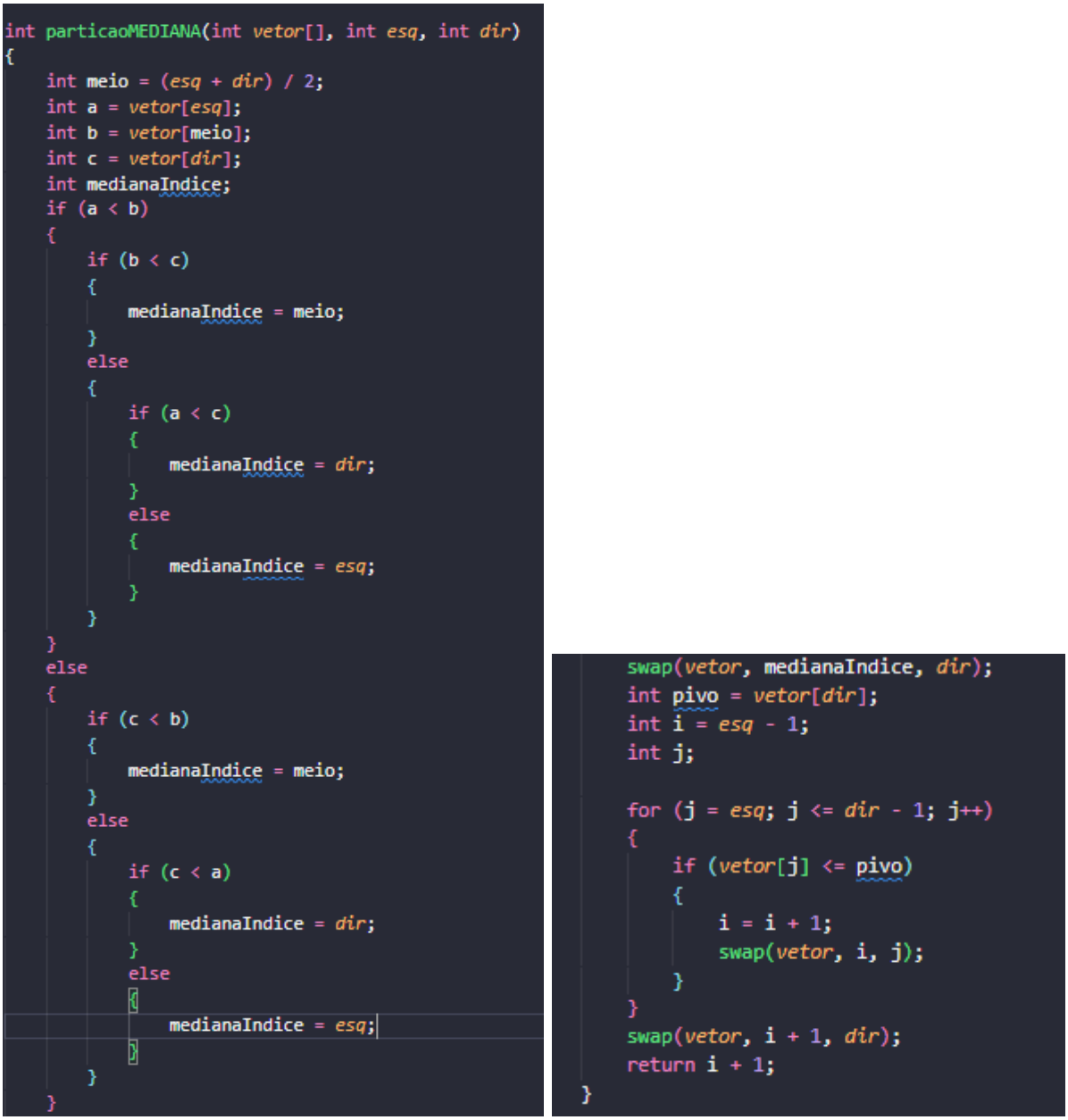
Figura 15 – Código função particao para o método da média

Fonte: Autoria própria

```
int particaoRANDOM(int vetor[], int esq, int dir)
{
    int k;
    double d;
    d = (double) rand () / ((double) RAND_MAX + 1);
    k = d * (dir - esq + 1);
    int randomIndex = esq + k;
    swap(vetor, randomIndex, dir);
    int pivo = vetor[dir];
    int i = esq - 1;
    int j;
    for (j = esq; j <= dir - 1; j++)
    {
        if (vetor[j] <= pivo)
        {
            i = i + 1;
            swap(vetor, i, j);
        }
    }
    swap(vetor, i + 1, dir);
    return i + 1;
}
```

Figura 16 – Código função particao para o método aleatório

Fonte: Autoria própria



```
int particaoMEDIANA(int vetor[], int esq, int dir)
{
    int meio = (esq + dir) / 2;
    int a = vetor[esq];
    int b = vetor[meio];
    int c = vetor[dir];
    int medianaIndice;
    if (a < b)
    {
        if (b < c)
        {
            medianaIndice = meio;
        }
        else
        {
            if (a < c)
            {
                medianaIndice = dir;
            }
            else
            {
                medianaIndice = esq;
            }
        }
    }
    else
    {
        if (c < b)
        {
            medianaIndice = meio;
        }
        else
        {
            if (c < a)
            {
                medianaIndice = dir;
            }
            else
            {
                medianaIndice = esq;
            }
        }
    }
    swap(vetor, medianaIndice, dir);
    int pivo = vetor[dir];
    int i = esq - 1;
    int j;

    for (j = esq; j <= dir - 1; j++)
    {
        if (vetor[j] <= pivo)
        {
            i = i + 1;
            swap(vetor, i, j);
        }
    }
    swap(vetor, i + 1, dir);
    return i + 1;
}
```

Figura 17 – Código função particao para o método da mediana

Fonte: Autoria própria

Foram realizados testes para cada entrada que variam entre 10, 100, 1.000. 10.000, 100.000, 1.000.000 instâncias e são ordenados de três formas: Crescente, Decrescente e Aleatório. Para cada combinação de quantidades de instâncias e tipo ordenação foi realizada uma execução do algoritmo. Os resultados da ordenação e o tempo de execução são gravados em arquivos diferentes para que se possa observar o resultado da ordenação e o tempo de execução do algoritmo. Os resultados destes testes separados por cada método de seleção do pivô são apresentados na tabelas e gráficos a seguir:

Tabela 3 – Quick Sort: Tempos por tipo e tamanho das entradas.

	10	100	1.000	10.000	100.000	1.000.000
Aleatório	0.0150	0.0150	0.0160	0.0170	0.0280	0.1650
Crescente	0.0150	0.0140	0.0150	0.0160	0.0180	0.0550
Decrescente	0.0380	0.0140	0.0140	0.0150	0.0190	0.0530

Fonte: Autoria própria

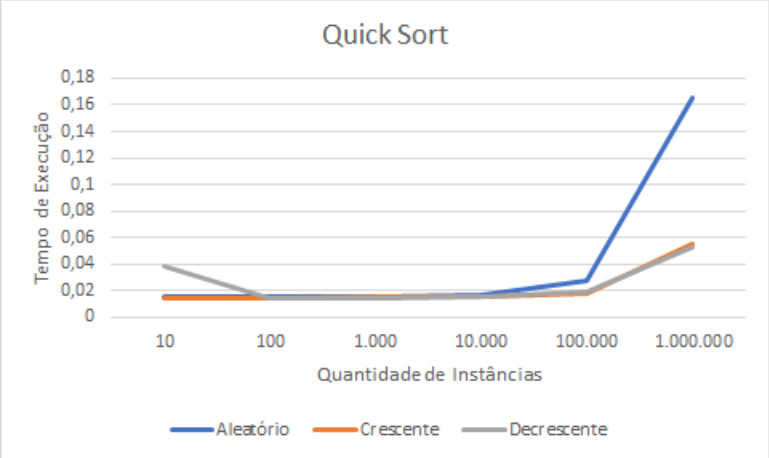


Figura 18 – Gráfico de tempos de execução Quick Sort

Fonte: Autoria própria

Tabela 4 – Quick Sort Pivô por Média: Tempos por tipo e tamanho das entradas.

	10	100	1.000	10.000	100.000	1.000.000
Aleatório	0.0150	0.0150	0.0150	0.0160	0.0340	0.2200
Crescente	0.0160	0.0130	0.0150	0.0170	0.0250	0.1560
Decrescente	0.0150	0.0150	0.0150	0.0150	0.0250	0.1760

Fonte: Autoria própria

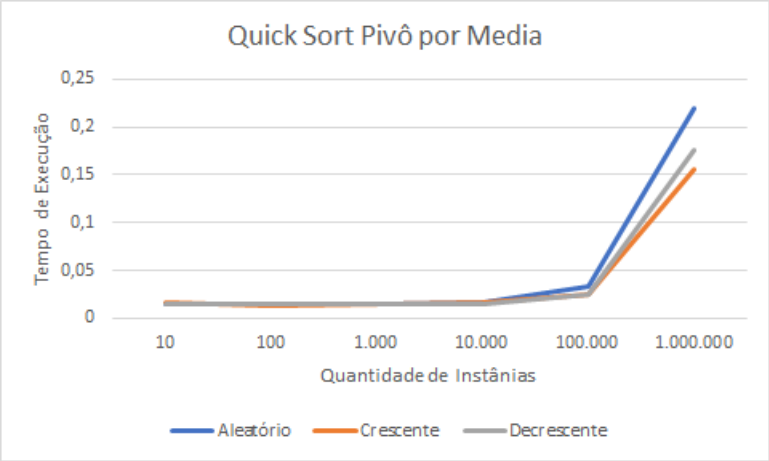


Figura 19 – Gráfico de tempos de execução Quick Sort com seleção de pivô por média

Fonte: Autoria própria

Tabela 5 – Quick Sort Pivô Aleatório: Tempos por tipo e tamanho das entradas.

	10	100	1.000	10.000	100.000	1.000.000
Aleatório	0.0210	0.0220	0.0210	0.0230	0.0480	0.2950
Crescente	0.0210	0.0200	0.0240	0.0250	0.0360	0.1980
Decrescente	0.0200	0.0230	0.0240	0.0240	0.0380	0.2060

Fonte: Autoria própria

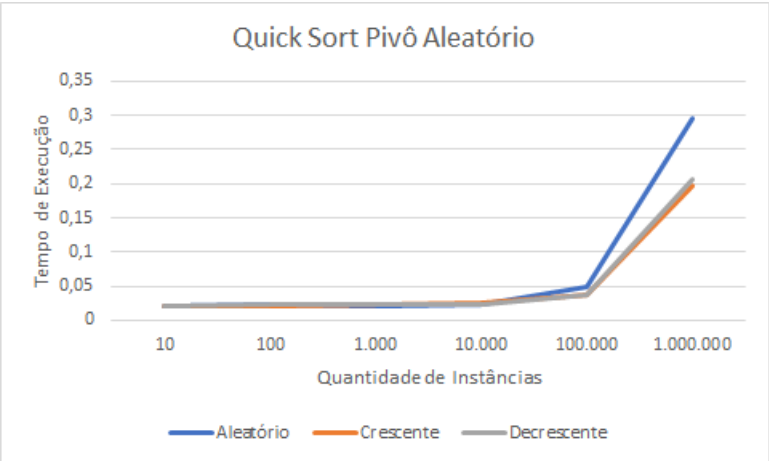


Figura 20 – Gráfico de tempos de execução Quick Sort com seleção de pivô de forma aleatória

Fonte: Autoria própria

Tabela 6 – Quick Sort Pivô por Mediana: Tempos por tipo e tamanho das entradas.

	10	100	1.000	10.000	100.000	1.000.000
Aleatório	0.0210	0.0210	0.0210	0.0250	0.0420	0.1790
Crescente	0.0220	0.0220	0.0290	0.0220	0.0310	0.1320
Decrescente	0.0200	0.0200	0.0230	0.0230	0.0460	0.2580

Fonte: Autoria própria

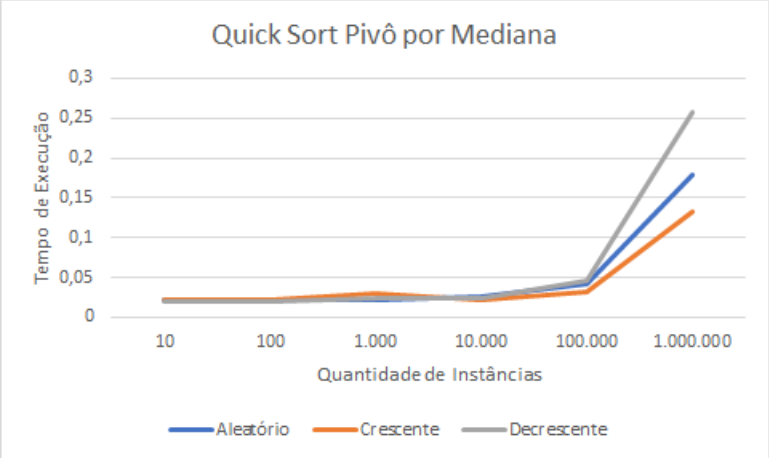


Figura 21 – Gráfico de tempos de execução Quick Sort com seleção de pivô por Mediana

Fonte: Autoria própria

Assim é possível observar que que apesar de baixa houve sim diferença entre os métodos de seleção do pivô é visível uma leve desvantagem de tempo para onde o pivô não foi selecionado de uma maneira balanceada, como no método de seleção aleatório. Ainda assim obteve-se um desempenho próximo para as versões vistas em (ZIVIANI et al., 2004) e as versões vistas em (CYBERINI, 2022).

5 Visão Geral dos Algoritmos

A seguir temos a tabela apresentando as complexidades para todos os casos dos algoritmos vistos juntamente com o gráfico comparando os tempos de execuções para cada um deles. Observa-se que as complexidades citadas são baseadas em conhecimento adquirido através das aulas da disciplina e na literatura anteriormente já citada. Na tabela o Quick Sort é representado por apenas uma linha, visto que sua complexidade varia com base na seleção do seu pivô, sendo o melhor caso um pivô balanceado e o pior caso um pivô próximo as extremidades.

Tabela 7 – Complexidade dos algoritmos

Algoritmo	**Melhor Caso (Ω)**	**Pior Caso ($\mathcal{O}$)**	**Caso Médio (θ)**
Insertion Sort	$\Omega(n)$	$\mathcal{O}(n^2)$	$\theta(n^2)$
Merge Sort	$\Omega(nlogn)$	$\mathcal{O}(nlogn)$	$\theta(nlogn)$
Quick Sort	$\Omega(nlogn)$	$\mathcal{O}(n^2)$	$\theta(nlogn)$

Fonte: Autoria própria

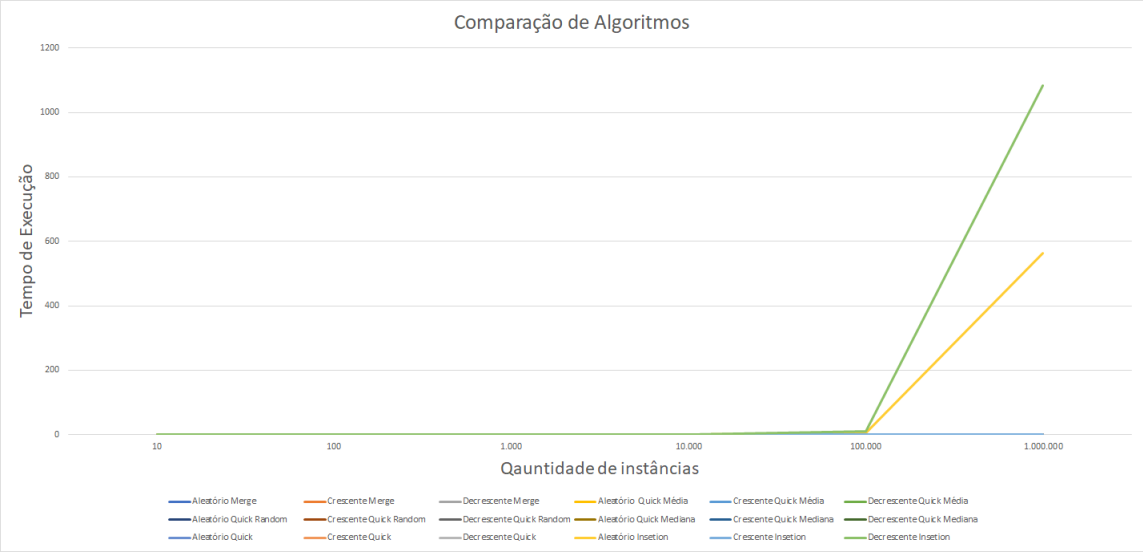


Figura 22 – Gráfico de tempos de execução para todos os algoritmos

Fonte: Autoria própria

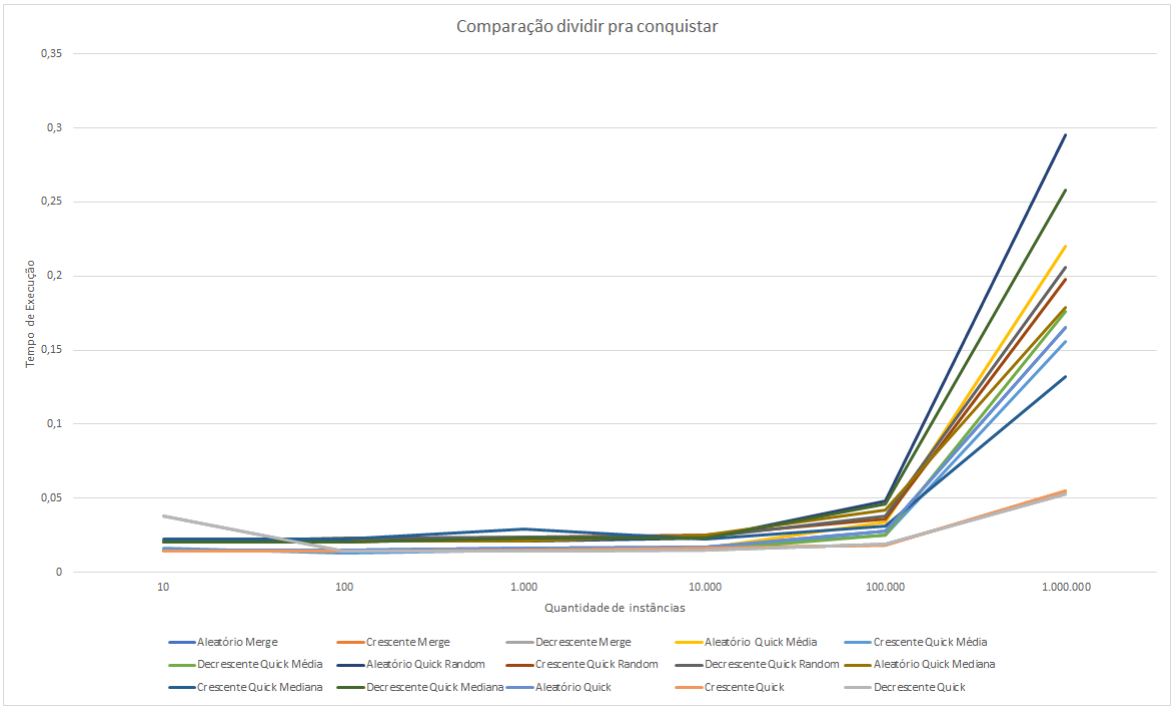


Figura 23 – Gráfico de tempos de execução para algoritmos de divisão e conquista

Fonte: Autoria própria

Ao observar as complexidades de todos os algoritmos vistos até o momento, é ao comparar com o primeiro apresentado é possível verificar que a complexidade difere de forma extrema, de tal forma que não é possível nem visualizar as linhas dos algoritmos de divisão e conquista, porém ao observar apenas algoritmos desta categoria vemos que estes não diferem muito em seus tempos de execução, demonstrando assim a efetividade da abordagem dos mesmos.

6 Conclusão

Portanto, levando em consideração os estudos e os testes feitos métodos de Ordenação Merge Sort e Quick Sort com suas variações ao longo deste trabalho pode-se perceber que em uma ordenação com 10, 100 e 1.000 instâncias, o algoritmo possui o mesmo desempenho, tanto para crescente, decrescente e randômico, a partir de 10.000 instâncias já se pode perceber uma pequena variação no tempo de execução que logo se torna uma grande diferença com 100.000 e 1.000.000 de instâncias. No caso do algoritmo Insertion Sort, observa-se uma extrema diferença em seus resultados exceto para o seu melhor caso, onde não é necessário ordenar, sendo este o menos efetivo e lento algoritmo apresentado neste artigo. Assim confirma-se que conforme a teoria os desempenhos alcançam desempenho compatível a complexidade apresentada para todos os casos.

Referências

CORMEN, T. H. et al. Algoritmos: teoria e prática. **Editora Campus**, v. 2, p. 296, 2002. Citado na página 6.

CYBERINI. **Blod Cyberini**. 2022. Disponível em: <https://www.blogcyberini.com/>. Acesso em: 19 nov. 2022. Citado 2 vezes nas páginas 19 e 28.

SEDGEWICK, R.; FLAJOLET, P. **Introduction to algorithm analysis**. [S.l.]: Addison-Wesley Reading, MA, 1996. Citado na página 19.

TOPTAL. **Toptal**. 2022. Disponível em: <https://www.toptal.com/developers/sorting-algorithms/>. Acesso em: 28 sep. 2022. Citado na página 6.

USF. **USF Computer Science Department**. 2022. Disponível em: <https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html>. Acesso em: 28 sep. 2022. Citado na página 6.

ZIVIANI, N. et al. **Projeto de algoritmos: com implementações em Pascal e C**. [S.l.]: Thomson Luton, 2004. v. 2. Citado 6 vezes nas páginas 3, 19, 20, 22 e 28.