Relatório - Métodos de Ordenação
Estrutura de Dados

Equipe : Eliabe Soares, Henricky de Lima
Matriculas: 470820, 475075

Professor: Atílio Gomes

Departamento de Ciência da Computação
Universidade Federal do Ceará (UFC) – Quixadá, CE – Brasil

Dezembro 2019

1 Introdução

O problema consiste em implementar os algoritmos de ordenação, estudados em sala, de forma recursiva e interativa, e de forma que possam ordenar os elementos de uma lista endadeada ou vetor, tendo por objetivo fazer o estudo dos tempos dos mesmos.Os algoritmos estudados foram BubbleSort, InsertionSort, SelectionSort, QuickSort, MergeSort e HeapSort.

2 Divisão de Tarefas

A divisão das tarefas na resolução e implementação para o trabalho foi balanceada, onde os alunos dividiram as implementação do algoritmo de forma que os dois fizessem a mesma quantidade.

3 Metodologia

Tendo como objetivo implemetar e testar os tempos dos algoritmos para cada tamanho de vetor testado, foi necessário o uso da biblioteca **chrono**, para a obtenção dos tempos de execução dos algoritmos, a biblioteca **cstdlib**, para trabalhar com números aleatórios, e a biblioteca **fstream**, para trabalhar com arquivos, e foi elaborados métodos para otimizar o processo de geração dos dados em arquivo.

Para agilizar o processo de geração de dados foi utilizado o **Google Colab**, que permite o uso elevado de memória e processamento em nuvem, além da possibilidade de execução paralela. Contudo, foi necessário abrir mão do uso do mesmo vetor para todas os algoritmos, perdendo a precisão geral entre gráficos.

Para facilitar o processo de obtenção dos tempos e reutilização do vetor original foram criadas as funções do tipo **TimeBubbleRecursive** que recebem o vetor ou lista retorna um double, criam uma cópia do mesmo e apartir desta aplicam ao algoritmo. Desta forma, basta aplicar o método de obtenção do tempo, como pode-se ver no exemplo abaixo.

```
double TimeBubbleRecursive(int *v, int n){
        std::cout<<"\tBUBBLE_Recursive:\n";
        int vec[n];
        for(int i=0;i<n;i++){
                vec[i]=v[i];
        }
        auto ini = std::chrono::high_resolution_clock::now();

        bubbleSort_recursive(vec,n);

        auto fim = std::chrono::high_resolution_clock::now();
        auto d = std::chrono::duration_cast<std::chrono::microseconds>(fim - ini).count();
        std::cout<<"\n\ttime: "<<(double) d/1000000<<" seg\n";
        return (double)d;
}
```

Outro método utilizado, visando o uso no **Google Colab**, foi usar três notebooks separados para rodar alguns algoritmos por vez, isso trás um grande ganho de tempo. Para fazer isso foi necessário, a primeira vista, fazer **main** separadas para cada dupla de algoritmo (Interativo e recursivo), por isso foi feito uma grande quantidade **main**s, como no exemplo do Apêndice B.1.

Para a geração de gráficos foi utilizado o código do Apêndice B.2, onde utilizou-se **python**, com as bibliotecas numpy e matplotlib para gerar de forma fácil os gráficos para cada arquivo de dados.

Foi pedido para testar os algoritmos com vetores de tamanhos 1000, 5000, 10000, 50000,100000, 500000 e 1000000 elementos, gerando cada um de forma aleatória e tirando a média dos tempos de 5 testes. Um observação a ser feita seria que alguns algoritmos não tinham capacidade de aguentar a memória exigida pelo grande volume de processo necessário, por isso foi reduzido o tamanho dos números de operações de modo que caiba nas limitações do algoritmo.

4 Complexidade da funções

4.1 BubbleSort

Bubble Sort é um algoritmo de ordem em notação de Big $o(n^2)$ pois nele há dois laços de repetição aninhados sendo que as operações no segundo laço serão excutadas, (n-1) vezes, o void BubbleSort em listas encadeadas será mais demorado do que o normal pois ha mais instruções, sendo elas d trocas de ponteiro e percorrer uma lista que depende d funções externas.

No Pior caso e no pior caso será o(n^2)

A versão recursivca será complexidade de o(n^2) São usadas duas funcões uma auxliar que é chamada para fazer as comparações com todos os elementos e a que é chamada pelo usuario para é usada para enviar o elemeto da comparação, com há **n** elementos a função auxiliar é chamada n vezes juntamente com a função que a usa.

4.2 InsertionSort

o Inserction sort em notação Big O o(n^2),é proporcional ao numero de comparações de quem tem a o valor maior. Nessa parte de código while ($i >= 0$ && $A[i] > key$) vai roda o laço externo (for (j = 1; j ¡ n; j++)) mandar de numeros n, assim ficamos com o somatório n-1+n-2n-3+...+o = $\frac{n(n-1)}{2} = o(n^2)$.Na Versão recursiva não da pra afimar se é O(n^2), pois em cada funcao há uma chamada recurvisa para si mesma.

4.3 QuickSort

Quick é um algoritmo bom, ou seja faz $nlogn$ operções. Usa duas funções.
Ideia é usar um pivô e apartir dele se basear pios todos a direita serão maiores e esquerda menores

```
int particiona (int *A, int p, int r)
void quickSortRecursivo(int *A, int p, int r)
```

Função particiona é usada para escolher um pivô e retorna sua posição
Função quickrcursivo é usada para dividir o vetor, em subvetores
Tempo d Execução : Dependo da divisão tanto pode ser $O(n^2)$ como $O(nlogn)$ Se for uma divisão no melhor caso resolvendo a recorrencia $T(n) = 2T(n/2) + (n). = O(nlogn$
Pior caso $O(n^2)$

4.4 MergeSort

O merge é um algoritmo boom, faz operações em $(nlogn)$
É usado duas funções:

```
void intercala(int *A, int p ,int q, int r)
```

Função recebe um vetor dividido com posições de inicio e fim,logo após, cria um novo vetor e vai colocando em ordem crescente o elementos dos vetores intercalados e por fim coloca d volta no vetor original. Tempo de Execução :
o Vetor é divido mas é percorrido n veze na intercalaçao

```
void mergeSortRecursive(int *A, int p,int r)
```

a função principal acopla em si a função intercala, a função vai dividindo o vetor se chamando recursivamente até o caso base, quando supera o caso base chama a função intercala Tempo de Execução :
Como vai dividindo o vetor ao meio a pilha de execução da recursão fica bem parecida com uma árvore balanceada então no pior caso ela chamada a altura de uma árvore que é aproximdamente $logn$ Complexidade em geral = $O(nlogn)$

4.5 HeapSort

O HeapSort é um algoritmo bom, com tempo de execução n no pior caso.O Algoritmo usa de duas funções para se executar completamente :

```
void constroi_heap(int *A, int n):
```

A ideia por trás do heap é tratar a estrtura utilizada seja ela uma lista ou vetor,como uma árvore completa, nessa implementação o tipo de heap utilizado é de max heap ou seja cada tendo o vetor A e k(k variando de 0 a n), nele A[k/2] $\geq A[k](aposicão(k/2)posicãodopaidek)$.

$TempodeExecucão$:
$Comoavariávelfpresentenafunçãovaidesecendodenívelemcadaiteraçãopensandoemumaárvore, sendosendorepetidonvezes, sab$
$\lfloor logn \rfloor + 1$, nesse caso como são feitas n vezes a mudança de nível a complexidade fica do tipo $n \cdot logn = O(nlogn)$

```
void peneira(int *A,int n)
```

A função peneira vai descendo a árvore com o objetivo de ajustar um quase heap ou seja dentro do vetor há um $A[k/2] < A[k]$ nesse caso é necessário ajustar a posição do A[k/2],descendo pela a arvore Tempo de Execução:
Como a função desce o elemento que esteje nos padrões de um max heap, no ior caso até chegar no último nível $(1 + \lfloor logn \rfloor)$ vezes complexida da função fica aproximdamente $nlogn = O(nlogn)$

```
void heapsortInterative(int *A, int n)
```

A função heapsortInterative que é a principal chamando as outras duas anteriores para fazer as operações. Tempo de Execução :
É chamada apenas uma vez a função constroi heap de complexidade $nlogn$
A função peneira é chamada aproximadamente n vezes, a mesma tendo complexidade $logn$
Complexidade Geral fica $2nlogn = O(nlogn)$

Versão Recursiva

```
void heapsortRecursive(int  *A, int n, bool e = true)
```

Funcionamento : A função começa com n elementos, com o vetor , e uma flag, sendo essa usada para definir quando se usar a função constroi heap, na primeira execução com a flag esta com valor *true* usa a função coonstroi heap logo após troca o vetor na posição zero (A[0]) com vetor na posição (n-1)(A[n-1]), jogando o maior elemento para o final do vetor,logo após é usada função peneira para reajustar o heap, por fim a função é chamada recursivamente passando o vetor, falg com valor false, tamanho n-1. Tempo de execuçao :

função constroi_heap é chamada uma vez ($nlog(n)$)

funcao peneira é chamada n vezes ($logn$)

Complexidade Geral no pior caso $2nlogn = O(nlogn)$

A Figuras

Resultados obtidos dos dados.

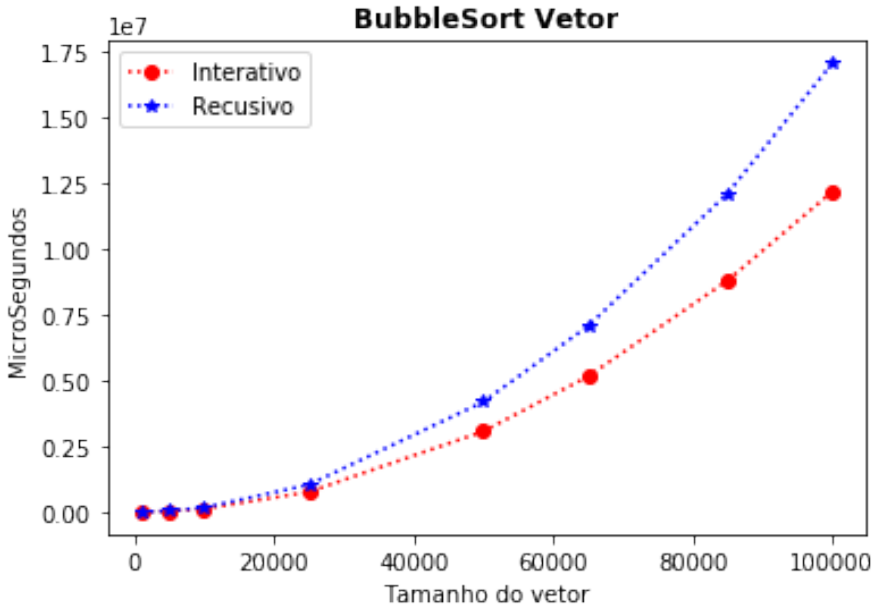

Figure 1: Gráfico do BubbleSort para Vetor

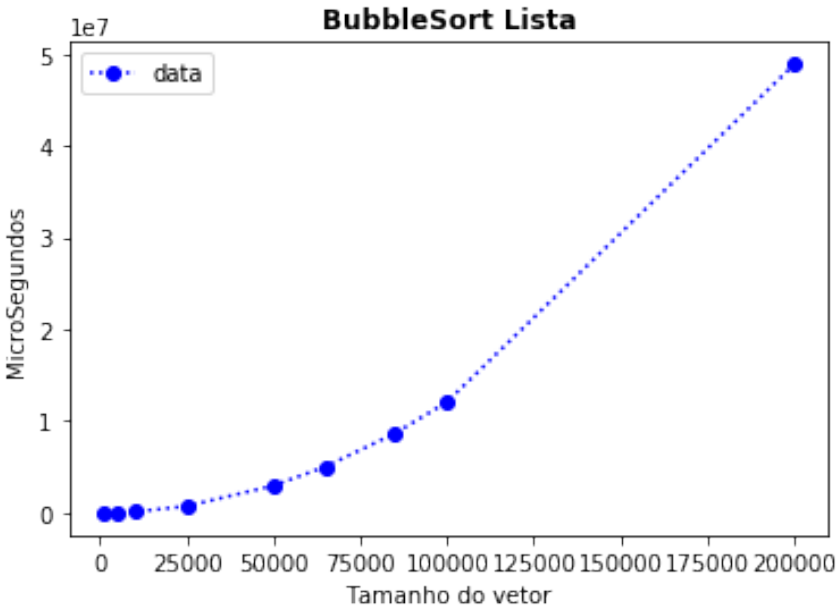

Figure 2: Gráfico do BubbleSort para Lista

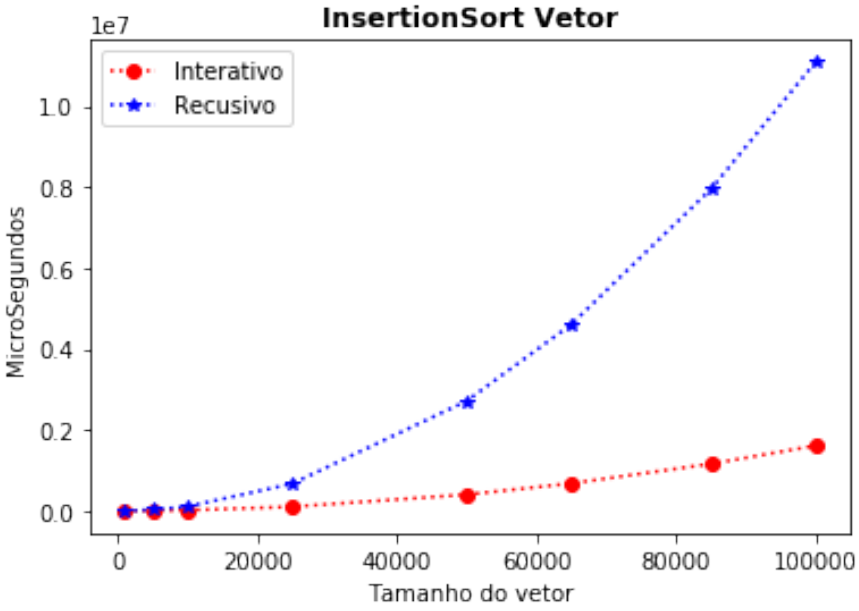

Figure 3: Gráfico do InsertionSort para Vetor

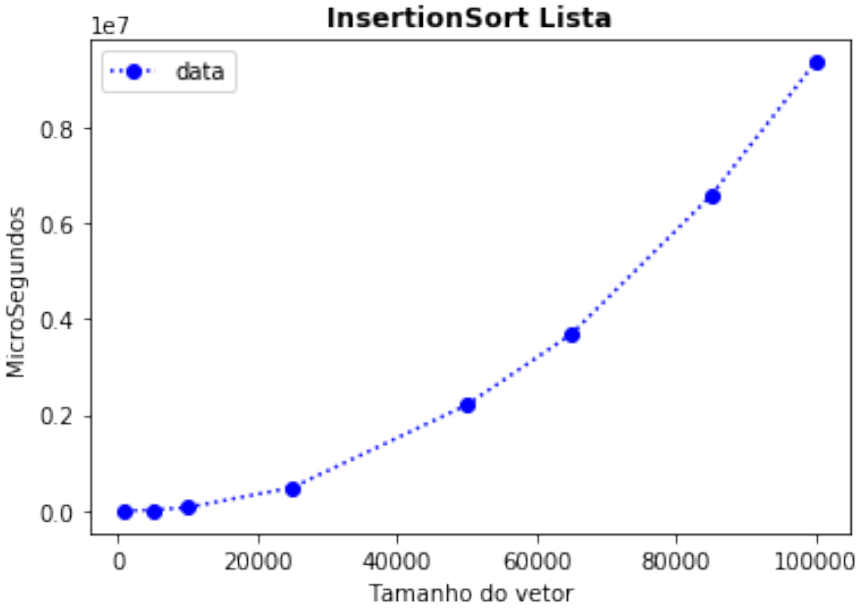

Figure 4: Gráfico do InsertionSort para Lista

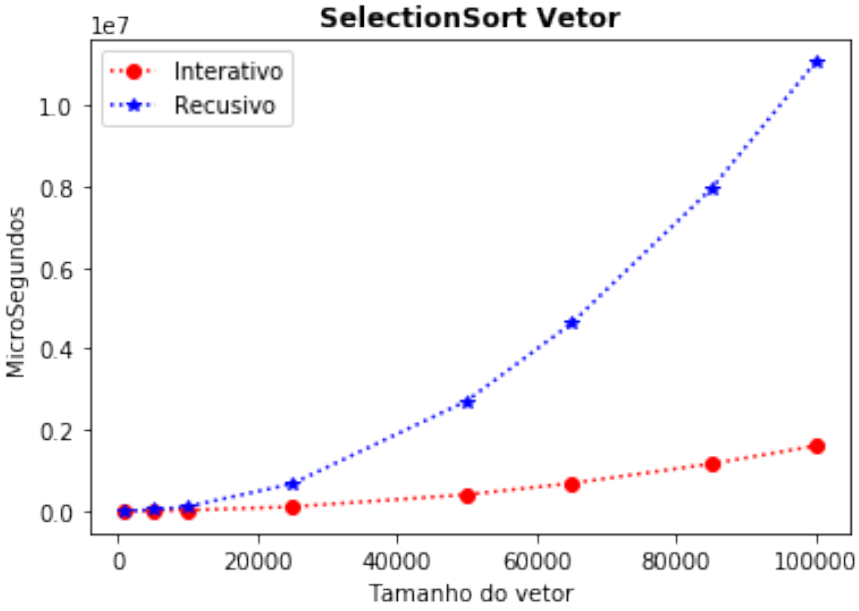

Figure 5: Gráfico do SelectionSort para Vetor

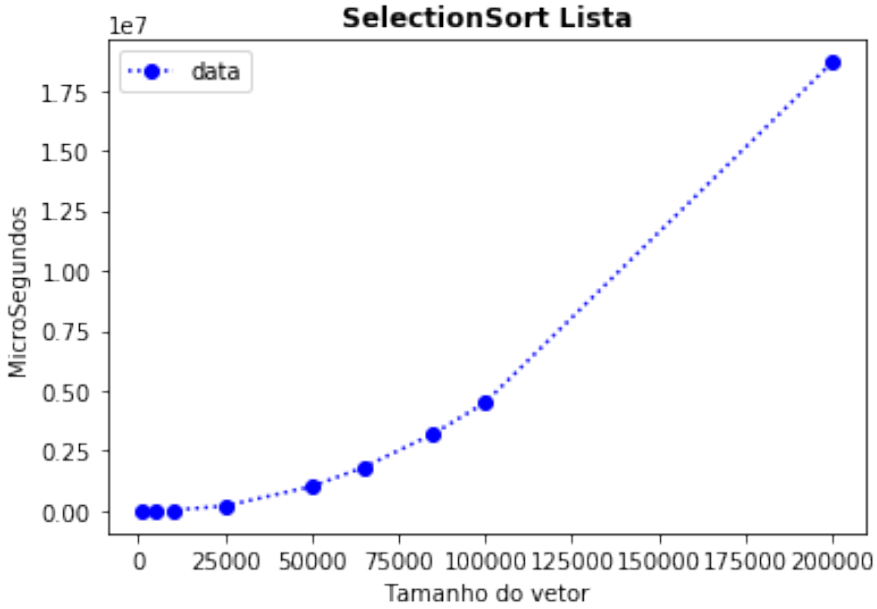

Figure 6: Gráfico do SelectionSort para Lista

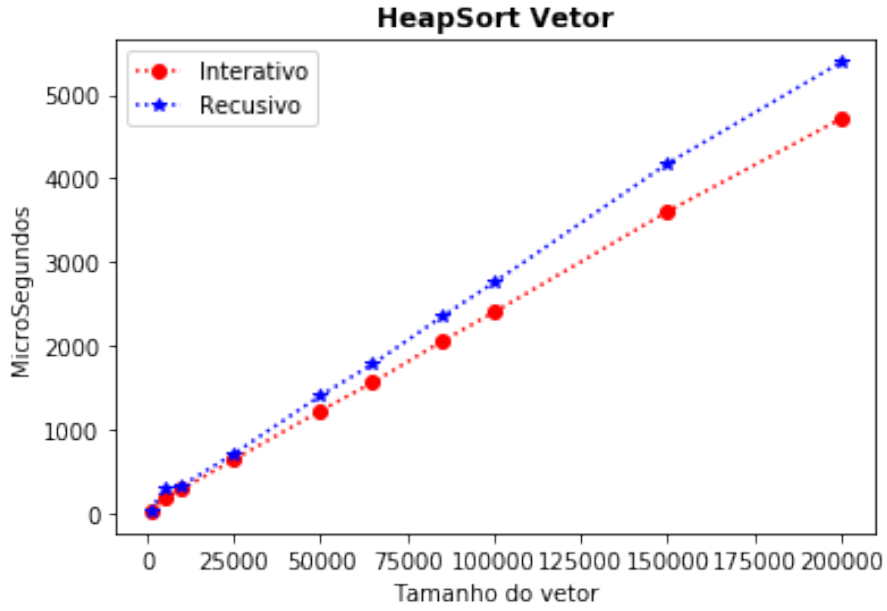

Figure 7: Gráfico do HeapSort para Vetor

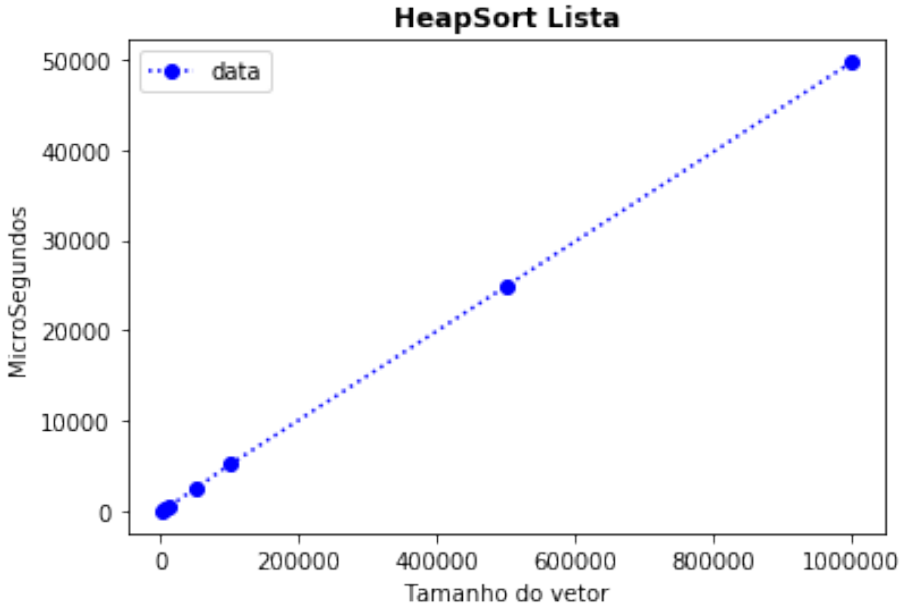

Figure 8: Gráfico do HeapSort para Lista

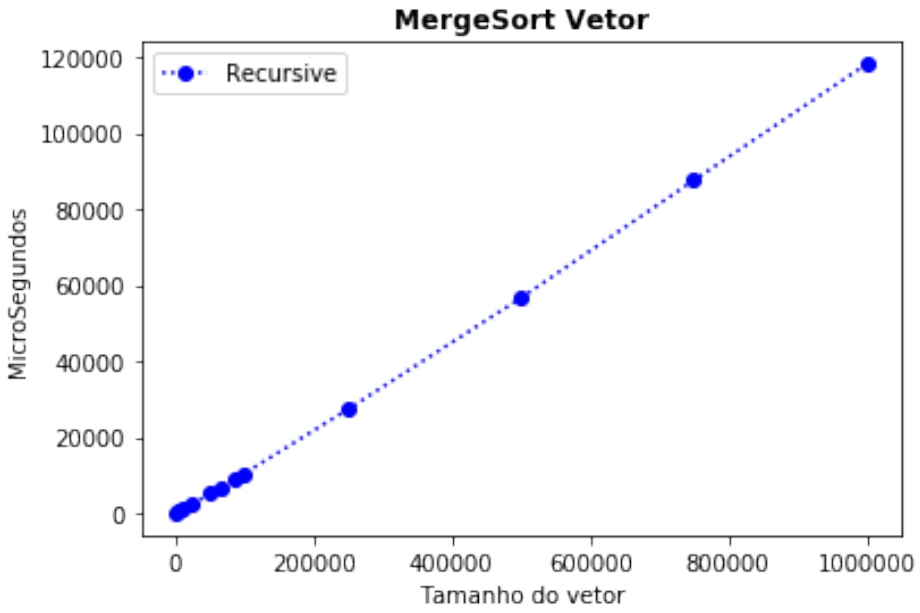

Figure 9: Gráfico do MergeSort para Vetor

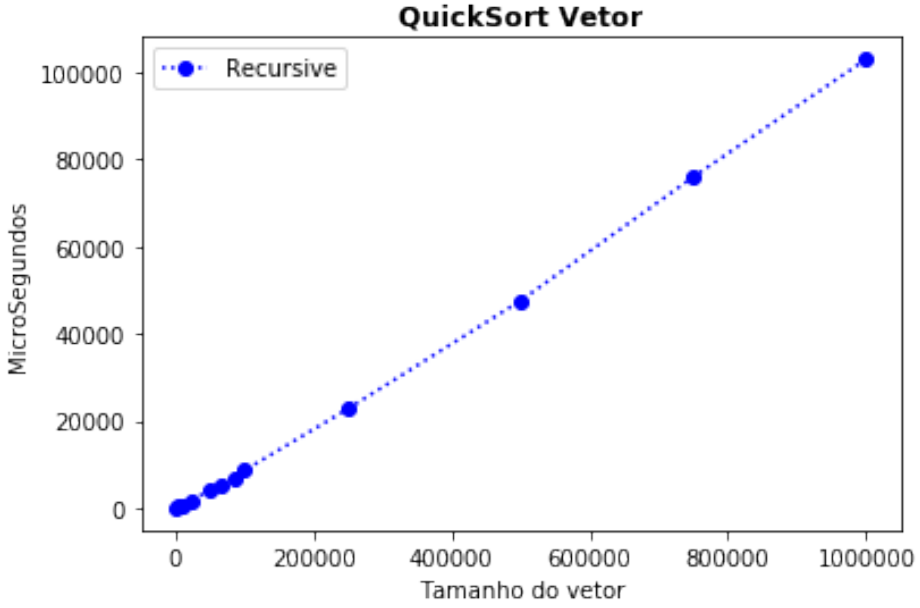

Figure 10: Gráfico do QuickSort para Vetor

B Códigos

B.1 Gerar Dados C++

```
#include <fstream> //arquivos
using namespace std;
//Global
const int tam[] = {1000,5000,10000,50000,100000,500000,1000000};
int TOTAL_N = sizeof(tam)/sizeof(tam[0]);
int main(){
        auto iniGeral = std::chrono::high_resolution_clock::now(); //tempo
        //Arquivo:
         ofstream arqOut;
         string linha; // string que vai armazenar as linhas do arquivo
         string aux =""; // string auxiliar para trabalhar com numero >9
         //definindo o arquivo no diretório
         arqOut.open("dadosQuick.txt",ios::out); //escrita
         //quantidade de testes para a média
        int N_media= 5;
        for(int k: tam){ //for percorrendo os tamanhos
                int somaI = 0, SomaR =0;
                int mediaI=0,MediaR=0;
                aux = "";
                int n = k;
                auto ini = std::chrono::high_resolution_clock::now();
                for(int l =0;l<N_media;l++){ //for para fazer as médias
                        std::cout<<"N: "<<n;
                        int v[n];
                        srand(time(NULL));
                        for(int i=0;i<n;i++){ // for para gerar os valores
                                v[i] = rand();
                        }
                        std::cout<<"-------------\n::QUICK_SORT::\n";
                        SomaR += TimeQuickRecursive(v,n);
                        if(l == N_media-1){
                                //rodou N vezes e tiro a média
                                MediaR = SomaR/N_media;
                                SomaR=0;//limpa
                        }
                }
                //pego a média
                MediaR = MediaR/N_media;
                //salvo
                aux = to_string(n)+"\t"+"\t"+to_string(MediaR);
                arqOut << aux <<"\n";
                auto fim = std::chrono::high_resolution_clock::now();
                auto d = std::chrono::duration_cast<std::chrono::microseconds>(fim - ini).count();
                std::cout<<"\n>>>Time to "<< k<<": "<<(double) d/1000000<<" seg\n";
        }
        arqOut.close();
}
```

B.2 Gerar Gráficos em python

```
import numpy as np
import matplotlib.pyplot as plt
def CriaGraf(path,Title, qtd=2,Legenda = "data"):
    M = np.loadtxt(path) #importar arquivos
    n,I,R = [],[],[]
    for i in range(0,len(M[:])): n.append(M[i][0])
    for i in range(0,len(M[:])): I.append(M[i][1])
    if(qtd ==2)://verificador
        for i in range(0,len(M[:])): R.append(M[i][2])
    if(qtd ==2):
        plt.plot(n,I,"r:o",label = "Interativo")
        plt.plot(n,R,"b:*",label = "Recusivo")
    else: plt.plot(n,I,"b:o",label = Legenda)

    plt.title(Title,fontweight="bold")
    plt.xlabel("Tamanho do vetor")
    plt.ylabel("MicroSegundos")
    plt.legend()
    plt.show()
# Usando
CriaGraf("Dados/dadosBubble.txt","BubbleSort Vetor")
CriaGraf("Dados/dadosBubbleList.txt","BubbleSort Lista",1)
```