Análise de algoritmos de ordenação

Nome: Jeferson Gonçalves Noronha Soriano
Matrícula: 471110
Curso: Engenharia de software

1. Algoritmos

a. SelectionSort

Basicamente o selection sort funciona pegando o menor valor q ele encontra no vetor e coloca na posição dele ai depois o segundo menor na posição dele

o código tem um laço mais externo com a variável i que vai aumentando de um em um, o i representa a posição do menor valor que estamos analisando naquele momento, tem um variável “min” que recebe a posição i, então quando o laço mais externo tiver no i = 0, ele vai pegar a primeiro posição como referência e quando o i = 1 ele vai pegar referência da segundo posição e assim por diante, o laço mais interno vai passar por todas as posições do vetor que vai analisando se o vetor na posição j e menor q o vetor na posição min se ele for o min passa até como referência aquela posição j, quando o laço mais interno acabar, o próximo passo é a realização de um swap entre o vetor na posição min e vetor na posição i, isso garante que todo número à esquerda estará ordenado

código SelectionSort.

```c
void selectionSort(int *vetor, int tam){
    int i, j, min;
    for(i = 0; i < tam-1; i++){
        min = i;

        for(j = i+1; j < tam; j++){
            if(vetor[j] < vetor[min]){
                min = j;
            }
        }

        int aux = vetor[min];
        vetor[min] = vetor[i];
        vetor[i] = aux;
    }
}
```

b. InsertionSort

Esse algoritmo é bem simples, uma analogia bem conhecida para explicar o insertion é de uma baralho, tu quer manter as cartas ordenadas na mão então a medida que chega uma carta nova você tem essa carta como referência e procura o lugar dela.

no código é feito da seguinte maneira, pegamos esse valor que será o chave e armazenamos em uma variável aux que é o valor do vetor na posição i, que é controlado pelo laço mais externo que passa pelo vetor todo, que vai ser analisado com os valores a sua esquerda, enquanto esse valor for menor que os valores a esquerda, o vetor J+1 receberá o valor do vetor j, quando o aux for maior que vetor na posição j o ciclo não entra e o vetor na posição j+1 recebe o aux, isso é controlado com por um ciclo mais interno que o j recebe o i + 1 e vai decrementando enquanto a condição do laço for verdadeira.

Código InsertionSort.

```c
void insertionSort(int *vetor, int tam){
    int i, j, aux;
    for(i = 1; i < tam; i++){
        aux = vetor[i];

        for(j = i-1; j >= 0 && vetor[j] > aux; j--){
            vetor[j+1] = vetor[j];
        }

        vetor[j+1] = aux;
    }
}
```

c. MergeSort

O merge Sort segue a técnica de divisão e conquista para a ordenação, o algoritmo consiste em dividir pela metade o vetor em sub vetores até que o sub vetor tenha tamanho 1, onde é possível fazer as comparação, isso de forma recursiva, então quando as chamadas das funções retornam a parte de merge acontece, então vem juntando e ordenando, uma analogia para a junção é que tem duas pilhas de baralho e você quer organizar em uma única pilha, então você a cartas do topo de cada pilha e ver qual a menor que vai para a pilha principal e a maior volta para o topo, quando um dos montes acaba, pega o resto que sobrou do outro monte e coloca na pilha principal.

No código essa divisão do vetor é feita de uma maneira lógica, com variavais que representa o início o meio e fim do vetor, o que seria a pilha principal no exemplo das cartas será um vetor aux, um laço acontece enquanto um dessa “pilhas de cartas” não acabar, quando uma delas acaba, o vetor aux só recebe o resto da “pilha” que falta, aí no fim de tudo você passa o vetor aux já organizado para o vetor principal, aí tem a lista organizada

Código MergeSort.

```
void mergeSort(int *vetor, int p, int r){
    if(p < r){
        int q = (p+r)/2;

        //dividir
        mergeSort(vetor, p, q);
        mergeSort(vetor, q+1, r);

        //alocando um vetor aux
        int *vetAux = new int[r-p+1];
        int i = p;
        int j = q+1;
        int k = 0;

        //intercalar no vetor vetor[p..q](a primeira metade) e vetor[q+1...r](segunda metade)
        while(i <= q && j <= r){
            if(vetor[i] <= vetor[j]){
                vetAux[k] = vetor[i];
                i++;
                k++;
            }else{
                vetAux[k] = vetor[j];
                j++;
                k++;
            }
        }

        //se sobrar alguma parte, copia para o vetor aux
        while(i <= q){
            vetAux[k] = vetor[i];
            i++;
            k++;
        }

        while(j <= r){
            vetAux[k] = vetor[j];
            j++;
            k++;
        }

        //agora copia o vetAux( ja q ele ta ordenado) para vetor
        for(i = p; i <= r; i++){
            vetor[i] = vetAux[i-p];
        }

        //por fim liberar a memoria alocada do vetAux
        delete[] vetAux;
    }
```

d. QuickSort

O quick sort usa a mesma ideia de divisão e conquista do merge sort, com a diferença que a parte de divisão do quick acontece tudo, na "descida" que as coisas acontecem no quick, ele vai tudo aí só depois se chama de forma recursiva.

A lógica está escolher um elemento para ser o pivô, que no nosso algoritmo sempre será o último elemento do vetor, e deixar todos os números que são menores que esse pivô para esquerda dele e todos que forem maior deixar para a direita do dele, repetindo esse passo algumas vezes e garantindo essa prioridade chegará um momento que o vetor estará organizado

‘ **Código QuickSort.**

```
void quickSort(int *vetor, int p, int r){

    if(p < r){

       //pivo sempre sera o ultimo elemento do vetor
        int pivo = vetor[r];
        int j = p;

        for(int k = p; k < r; k++){
            if(vetor[k] <= pivo){
                int aux = vetor[k];
                vetor[k] = vetor[j];
                vetor[j] =  aux;

                j++;
            }
        }
        vetor[r] = vetor[j];
        vetor[j] = pivo;

        int i = j;
        //dividir
        quickSort(vetor, p, i-1);
        quickSort(vetor, i+1, r);
    }
}
```

2. Algoritmo escolhido por mim

a. ShellSort

O shellSort é um “insertion sort melhorado”, porque permite realizar a troca de elementos distantes um do outro. Realiza a troca em um espaço(h) calcular qual seria o valor desse h, o que dita se a troca será eficiente, vamos usar esse exemplo que não e o h mais eficiente, supondo que h = tam/2 e ao longo do ciclo o h ia sendo dividido por 2 até h = 1.

Quando o h = 1 é a garantia que o vetor será ordenado, e o algoritmo realiza um insertion sort comum, mas como o shell sort é melhor que o insertion se ele realiza o algoritmo do insertion quando o h = 1??

o insertion sort é muito eficiente quando o vetor está quase organizado, e é o que acontece antes do h = 1, o vetor vai ficando em um estado excelente, quando o insertion acontece, é quando ele tá em um estado bom para um insertion ser o mais rápido.

mas o h = tam/2 não é o melhor h de todos para um âmbito geral(pode ter um caso que ele seja a melhor opção) o melhor h proposto por Knuth o criador do algoritmo é o h = 3 * h + 1
no código realizamos um while que roda enquanto o h < tam

Código shellSort.

```
void shellSort(int *vetor, int tam){
    int i, j, aux;
    int h = 1;
    //primeiro vamos calcular os saltos q será o H
    while(h < tam){
        h *= 3 + 1;
    }
    //vai dividir os saltos por 3. ate q o salto seja 1 e realize um insetionSort normal
    for(; h > 0; h /= 3){

        for(i = h; i < tam; i++){
            aux = vetor[i];

            for(j = i-h; j >= 0 && vetor[j] > aux; j -= h){
                vetor[j+h] = vetor[j];
            }

            vetor[j+h] = aux;
        }

    }
}
```

3. Gráficos

a. explicação de como foi pego os dados

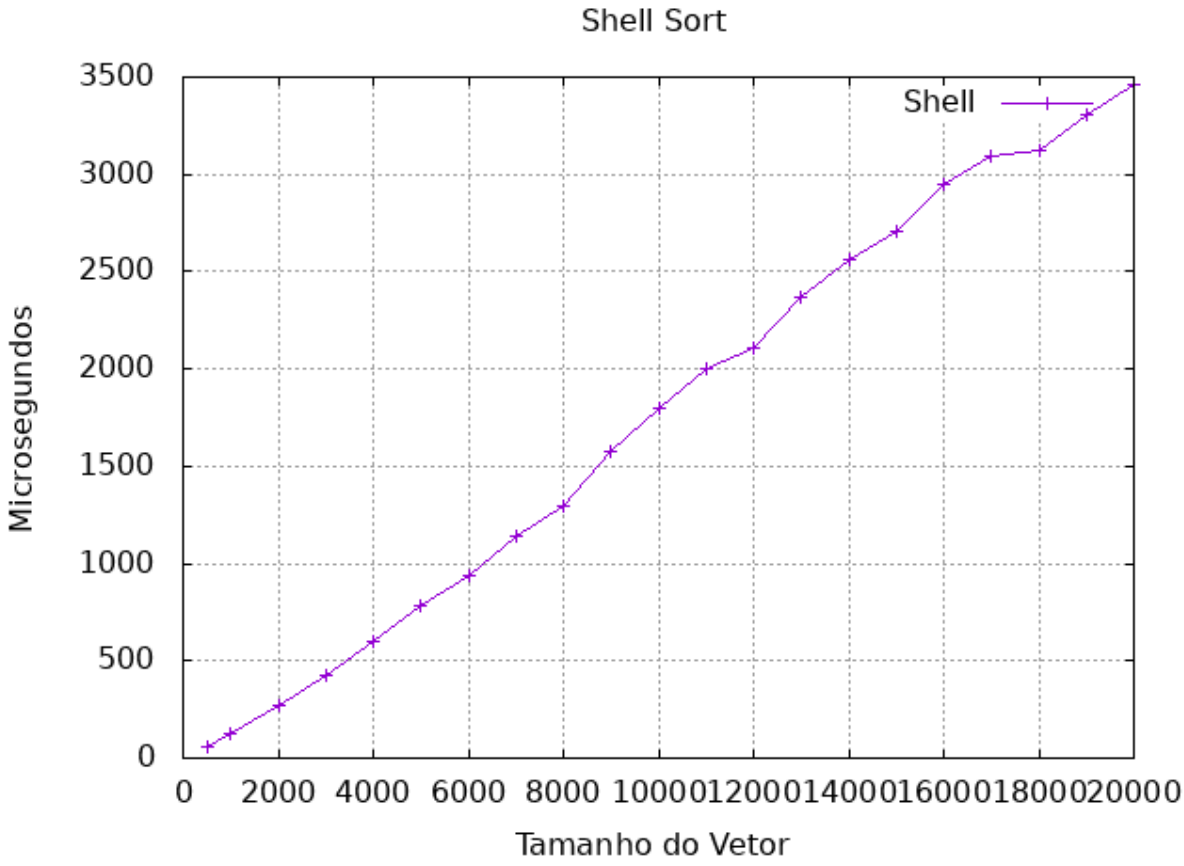
Tem uma função que gera os dados de forma aleatória para cada vetor de tamanho N, no nosso código gera 20 arquivos para cada vetor, e armazena em um arquivo .dat que posteriormente será lido para pegar esses dados.

quando a função de calcular o tempo de execução é chamada, passamos um vetor contendo os tamanhos de vetores, o nome do arquivo que será responsável por guardar o tempo médio que o algoritmo de ordenação executou para um vetor tamanho N com elementos aleatórios

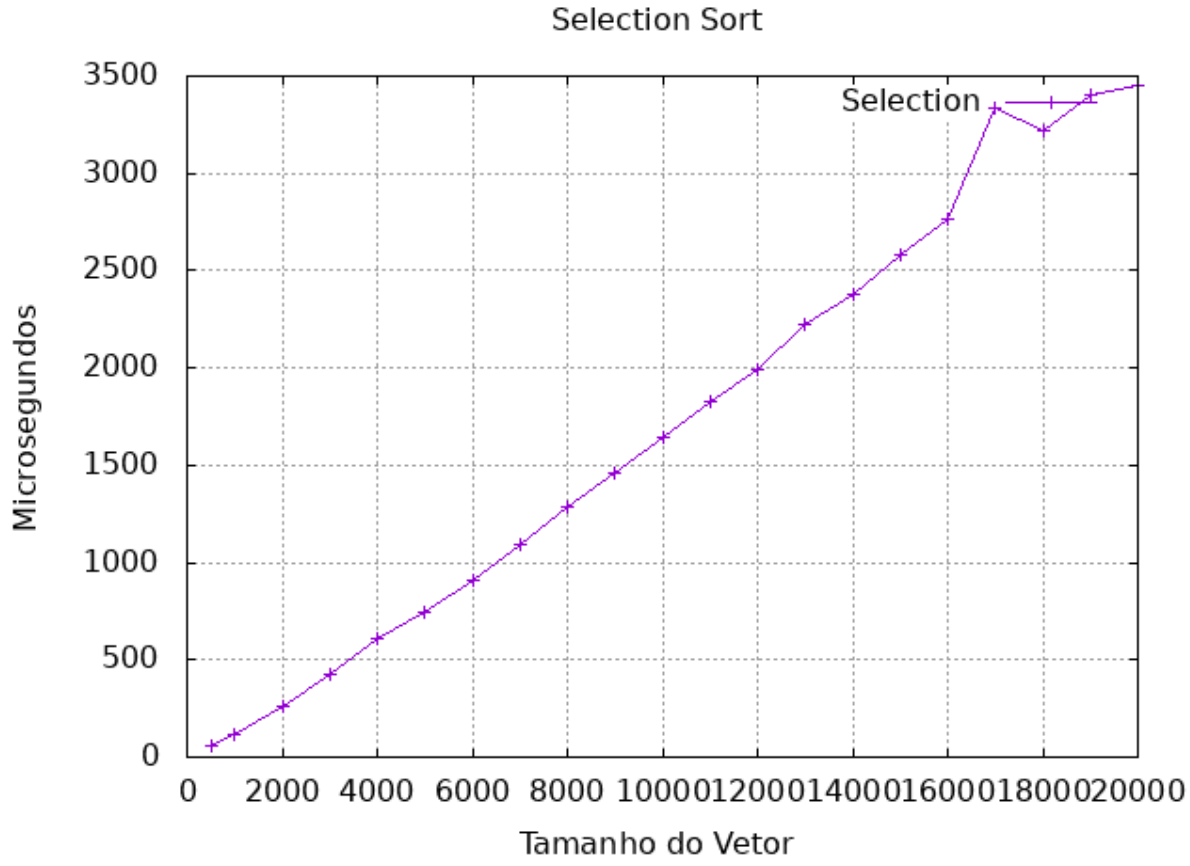
Então para termos o tempo de execução do código de ordenação criamos uma variável INICIAL que pega o clock antes da função de

ordenação ser chamada e um variável FINAL que pega o clock no momento que a chamada da função acaba, então o FINAL - INICIAL temos o tempo de execução

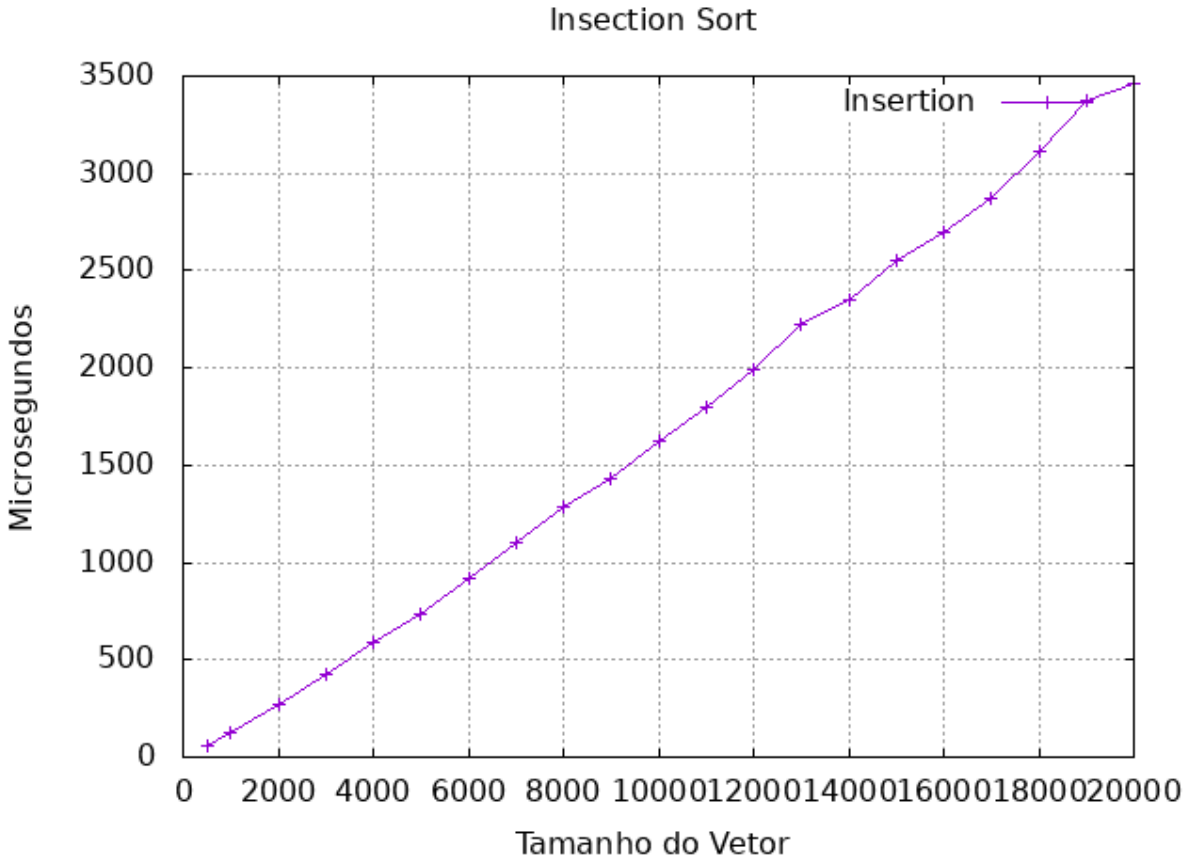
b. ShellSort(vetor)



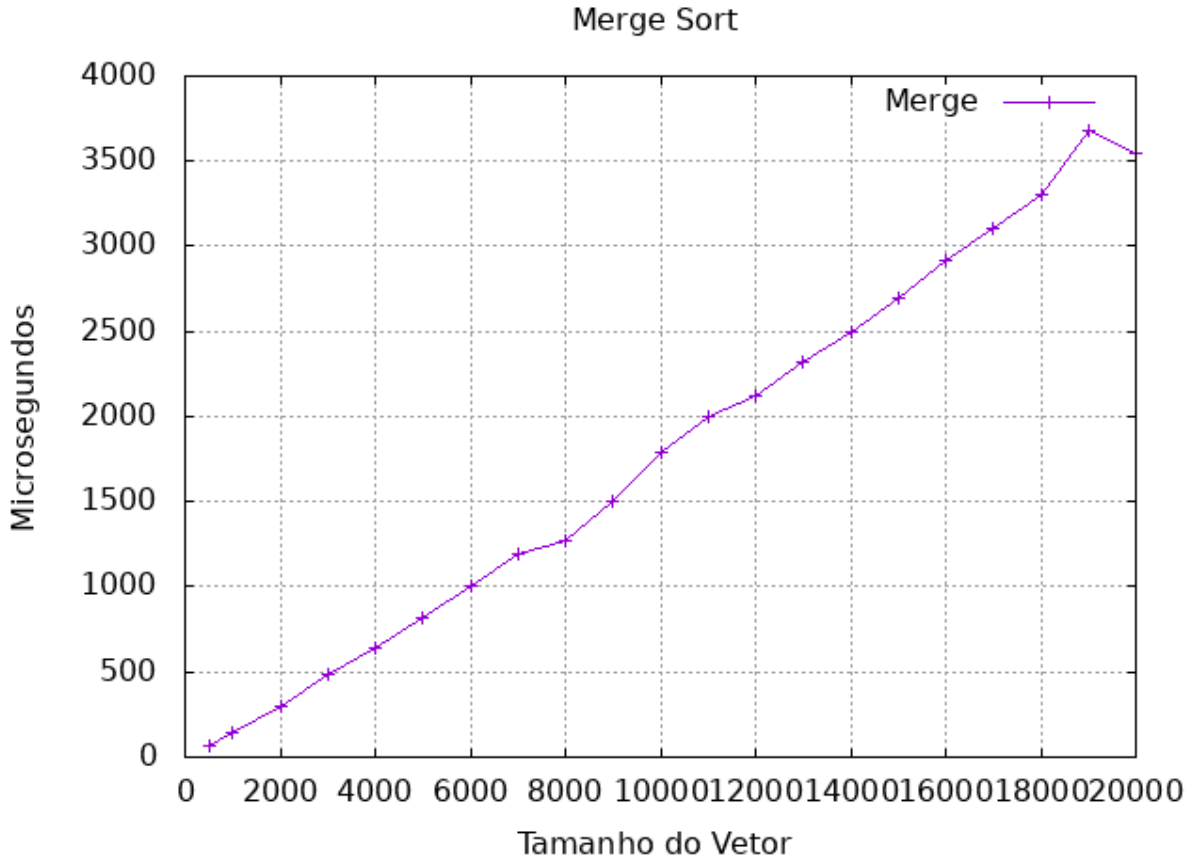
c. SelectionSort (vetor)



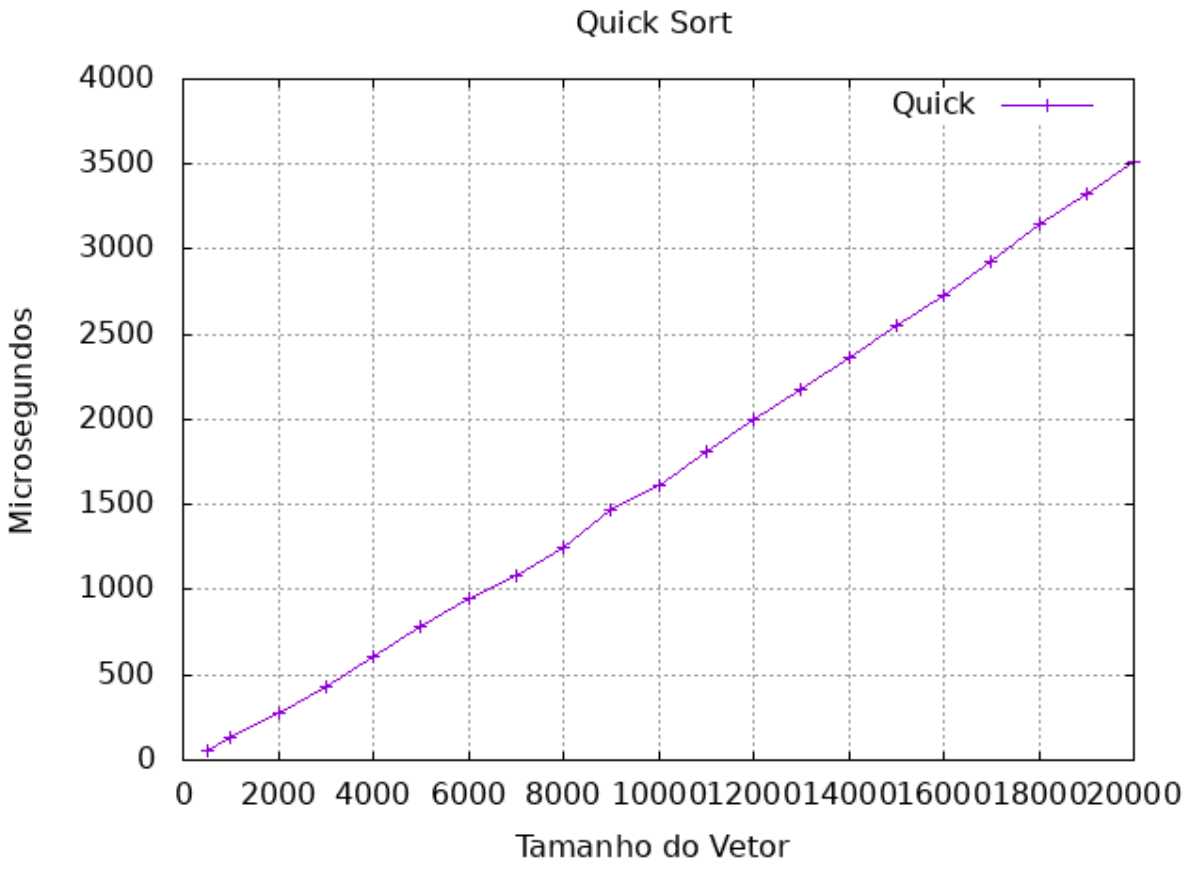
d. InsertionSort (vetor)



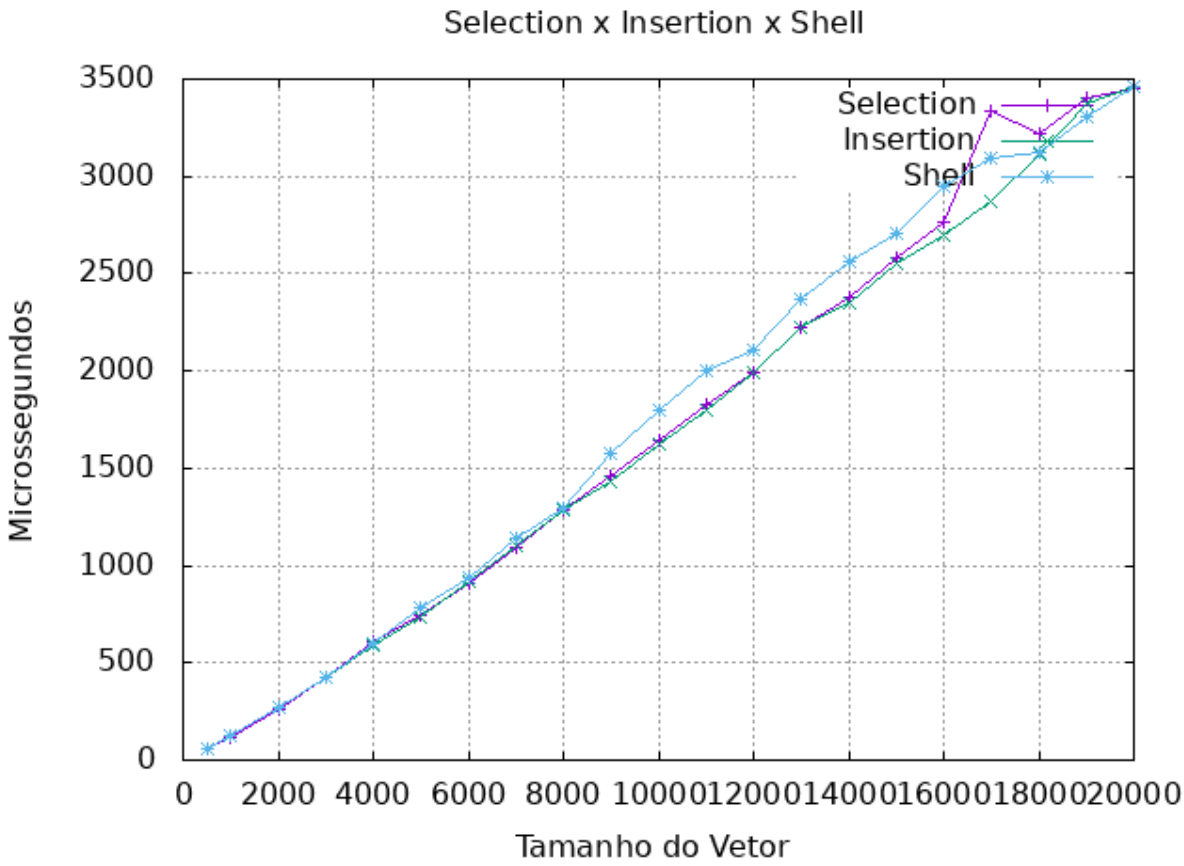
e. MergeSort (vetor)



f. QuickSort (vetor)



4. Análise do gráfico dos métodos simples



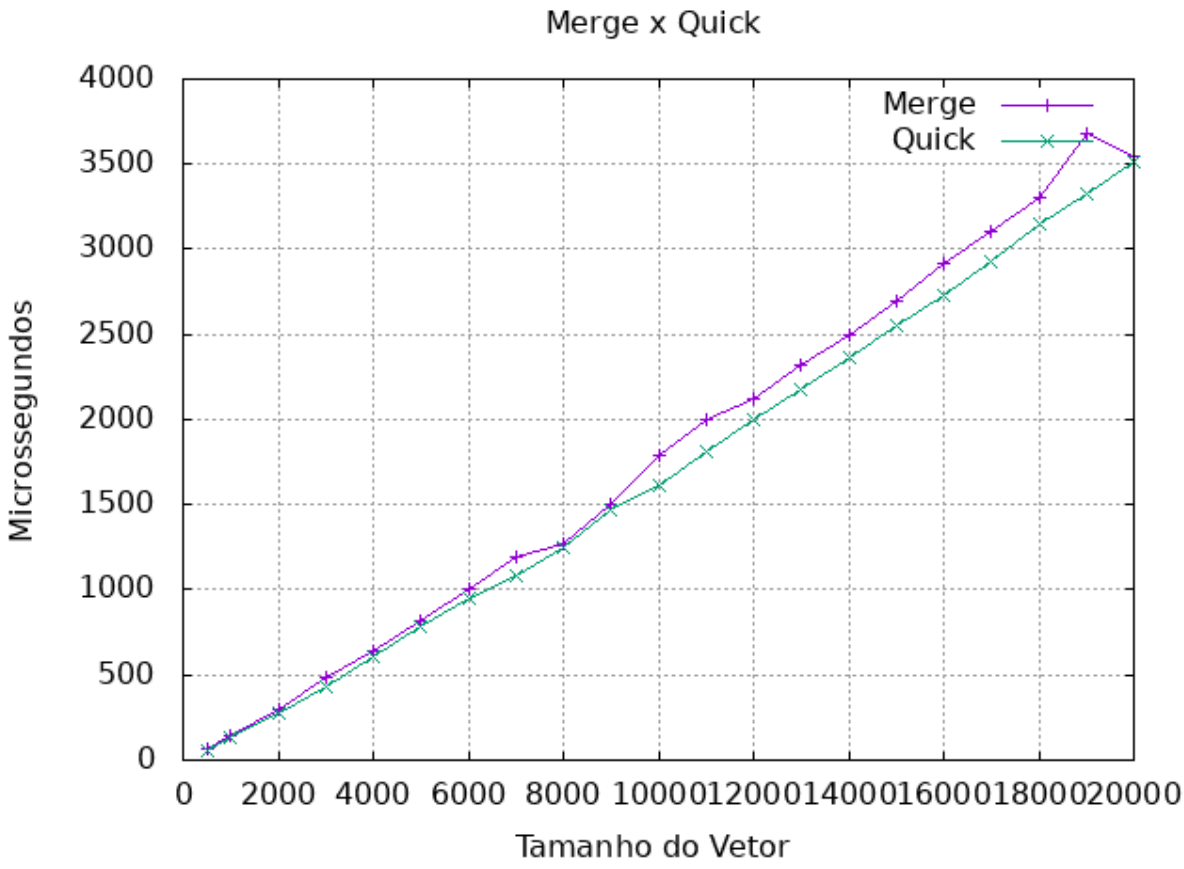
Podemos ver que quando o vetor tem tamanho pequeno os algoritmos são bem eficientes e praticamente tem o mesmo tempo até o tamanho 8000.

como os valores foram gerados de forma aleatória, acredito que foram gerados arquivos mais próximo de vetores que seriam o pior caso, ali no tamanho de vetor 17000, o que podemos ver o comportamento do selection sort, que tem um pico muito elevado no tempo de execução, enquanto os outros continuam meio que na mesma, sem esse pico repentino, o que mostra que em vetores mais desorganizado, onde os valores menores estão mais próxima do fim do vetor, o selection sort é o menos eficiente

Também podemos reparar na possível progressão do shell, o que indica que quanto maior o tamanho do vetor ele ta tendo um tendência a ficar com um tempo menor que o insertion sort, ali no tamanho de vetor 18000 ele ta ficando menor em tempo de execução em relação ao outros.

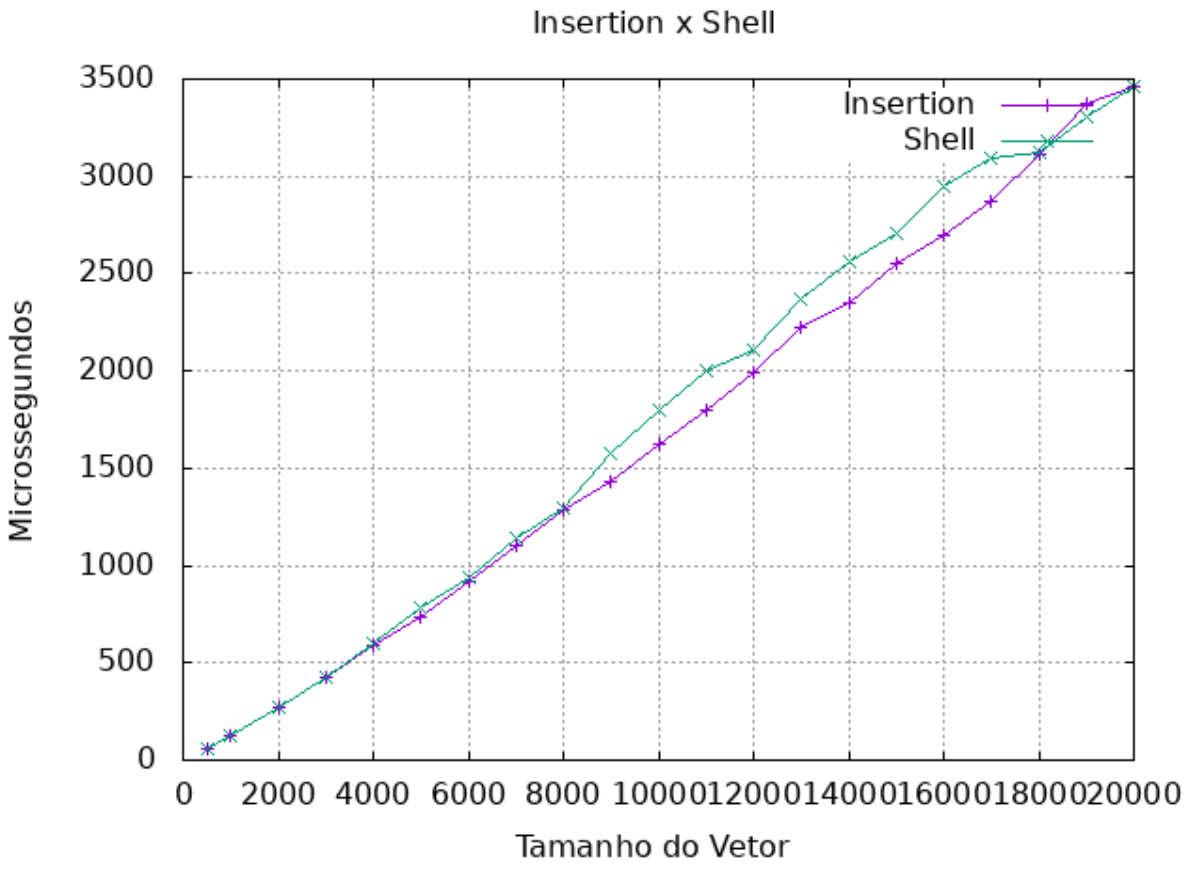
O insertion sort quando não teve praticamente o mesmo tempo de execução que os outros algoritmos, ele estava com os melhores tempo, podemos dizer que os insertion é a melhor escolha entre esse métodos simples de ordenação, é o melhor quando o vetor está preenchido de forma aleatória

5. Análise do gráfico dos métodos Sofisticados



Como foi dito, a complexidade de pior caso do quicksort pode até ser pior que a do merge, mas na prática, ele é bem melhor, até com o tamanho do vetor 9000 o tempo dos dois é bem similar, mas daí em diante é perceptível que o quick é mais eficiente, até parece que não é muita coisa, pq são alguns milésimos, mas isso rodando várias vezes, esse milésimos tem muita diferença no final das contas

6. Gráfico do ShellSort X InsertionSort



Até o tamanho de vetor 4000 os dois tem o mesmo tempo, até o tamanho 18000 é mais vantajoso usar o insertion sort, ele é consideravelmente mais rápido, mas ali nos tamanho 200000 a tempo do shell dá uma diminuída boa e do insertion continua a aumenta, o que indica que o shell para tamanhos maior que esse seja bem mais vantajoso.

quando o vetor é pequeno é muito rápido e não faz diferença qual método de ordenação irá usar

8.Fontes

https://pt.wikipedia.org/wiki/Algoritmo_de_ordena%C3%A7%C3%A3o

https://pt.wikipedia.org/wiki/Insertion_sort

https://pt.wikipedia.org/wiki/Selection_sort

https://pt.wikipedia.org/wiki/Merge_sort

https://pt.wikipedia.org/wiki/Quicksort

https://pt.wikipedia.org/wiki/Shell_sort

https://www.youtube.com/watch?v=v0AKijIlcXQ

https://www.youtube.com/watch?v=5prE6Mz8Vh0

https://www.youtube.com/watch?v=wx5juM9bbFo