Relatório

Nome: João Victor Dias Barroso
Matrícula: 474110
Professor: Atílio Gomes

1 Introdução

Este trabalho consiste em implementar 4 algoritmos de ordenação: InsertionSort, SelectionSort, QuickSort e MergeSort de forma iterativa e recursiva para cada algoritmo. Além disso, analisar a ordem de complexidade de cada algoritmo, gerar gráficos. Para a geração dos gráficos foi usada uma ferramenta de linha de comanado chamada gnuplot, que é uma ferramenta que plota gráficos de acordo com um arquivo de texto.

2 Grupo

Por questões práticas resolvi fazer sozinho. Então todas as tarefas de implementação, documentação foram feitas exclusivamente por mim.

3 Metodologia

Todos os algoritmos foram prototipados em um arquivo chamado vetorOrdenado.h, implementados em um arquivo chamado vetorOrdenado.cpp e testados em uma main.cpp. Para cada implementação seja esta iterativa ou recursiva, foram gerados valores aleatórios de 500 até 20000, afim de testar o tempo de execução das mesmas. As funções de gerar os dados e ler os dados são totais implementações do arquivo main.cpp enviado pelo professor, só que adaptando para os meus algoritmos.

4 Insertion Sort - Ordenação por inserção

4.1 Iterativo

Na implementação deste algoritmo de forma iterativa:

```
1  void iteractive_insertion_sort(int vet[], int size) {
2      int i = 0, j = 0, key = 0, index = 0;
3      for(i = 1; i < size; i++) {
4          key = vet[i];
5          index = i
6          for(j = i - 1; j >= 0; j--) {
7              if(key < vet[j]) {
8                  vet[j+1] = vet[j];
9                  index = j;
10             }
11         }
12         vet[index] = key;
13     }
14 }
```

É criada uma key que recebe o vetor na posição atual e corresponde ao valor que é necessário alterar a posição. No segundo *for* é que começa da posição em que i está atualmente, é criado uma verificação se a chave é menor que os valores que estão a sua esquerda, se sim, estes valores são deslocados a direita, até que key seja maior ou igual aos valores da esquerda. No fim a posição atualizada (index) recebe a key que foi modificada. Este é uma algoritmo que pode ser considerada *in loco* por não usar estruturas de dados adicionais. O tempo de execução deste algoritmo é: considerando que o primeiro *for* vai de 1 a *size* - 1 e que na i-ésima iteração o segundo laço rodará i vezes, então tem-se que:

$$
\begin{aligned}
C + 1 + C + 2 + C + 3 + ... + C + size - 1 = C \cdot \sum_{i=1}^{size} i \\
C \cdot \sum_{i=1}^{size} i = \sum_{i=1}^{size} i \\
\sum_{i=1}^{size} i = (1 + size) \cdot \frac{(size)}{2} \\
C \cdot (1 + size) \cdot \frac{(size)}{2} = C \cdot \left(\frac{size^2 + size}{2} \right)
\end{aligned}
\tag{1}
$$

Esta equação resultante tem maior grau o quadrático, o que indica que a ordem de complexidade é $\mathbf{O}(Cn^2)$, onde n=size e C é uma constante que representa todas as atribuições e criação de novas variáveis que foram realizadas. O que fica explícito no gráfico abaixo que apresenta uma curva crescendo gradativamente.

4.2 Recursiva

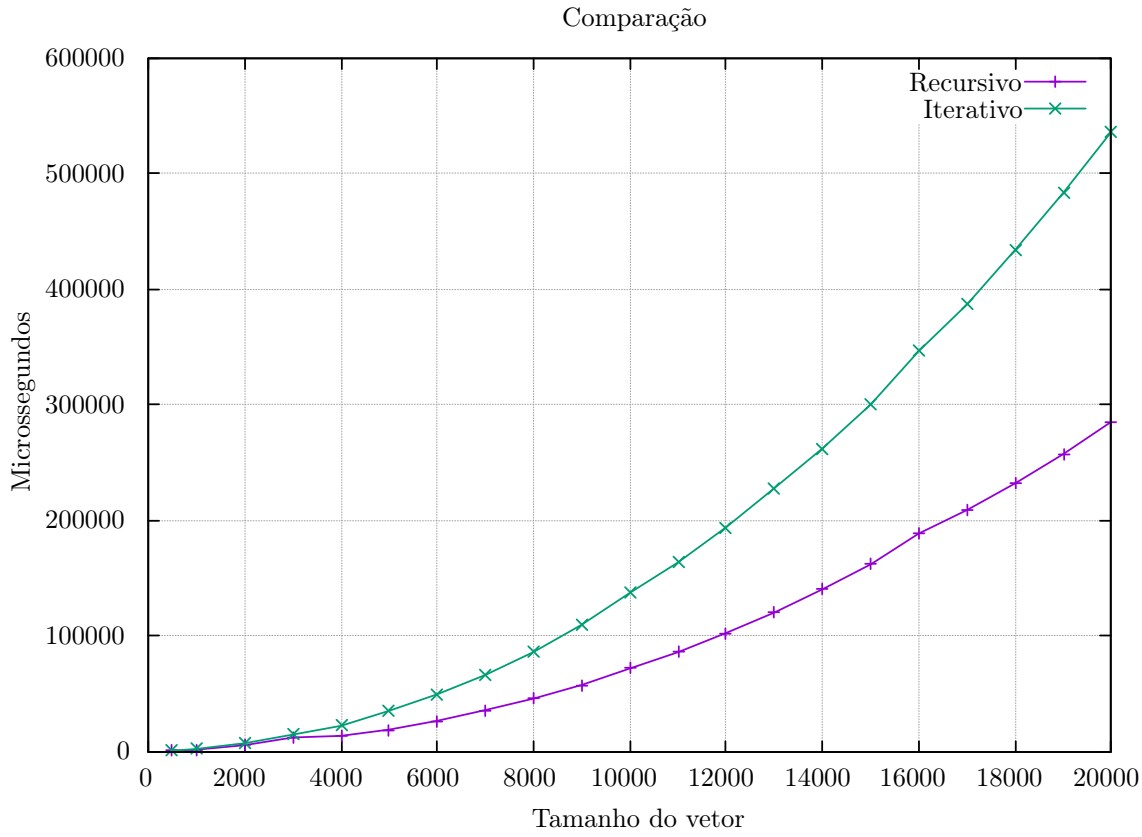
```
1  void recursive_insertion_sort(int vet[], int size, int index
       , int key, int current) {
2      while(index >= 0 && key < vet[index]) {
3          vet[index+1] = vet[index];
4          index--;
5      }
6      vet[index+1] = key;
7      if(current < size-1) {
8          recursive_insertion_sort(vet, size, current, vet[
               current+1], current+1);
9      }else return;
10 }
```

Nesta implementação recursiva é verificado se a chave escolhida a cada chamada recursiva é menor que os valores que estão a sua esquerda, se sim assim como na implementação iterativa vista acima, estes valores maiores que as chaves são deslocados para direita, conclui-se que sempre os valores a esquerda de qualquer chave a partir da segunda chamada recursiva vão estar ordenados. Assim como na sua versão iterativa a ordem de complexidade é algo próximo de $\mathbf{O}(n^2)$, onde n = size pois:

$$
\begin{aligned}
1 + 2 + 3 + ... + size - 1 &= \sum_{=1}^{size} i \\
\sum_{i=1}^{size} i &= (1 + size) \cdot \frac{(size)}{2} \\
C + (1 + size) \cdot \frac{(size)}{2} &= C + \left(\frac{size^2 + size}{2} \right)
\end{aligned}
\tag{2}
$$

Como visto no cálculo acima, a constante ao invés de multiplicada é somada, pois na implementação recursiva existem menos atribuições. No gráfico acima fica nítido esse termo quadrático pelo gráfico apresentar uma curva, embora que pequena.

4.3 Comparação Recursiva e Iterativa



Como pode-se perceber no gráfico acima a implementação recursiva por ter menos atribuições deu resultados melhores que o mesmo algoritmo, só que iterativo. Ou seja, a recursão acabou ordenando todos os vetores em menor tempo. Como explicitado nos cálculos destes dois algoritmos, a versão que usa estritamente laços, leva aproximadamente o dobro do tempo que na versão usando recursão.

5 Selection Sort - Ordenação por seleção

5.1 Iterativo

Implementação:

```
1  void iteractive_selection_sort(int vet[], int size) {
2      int menor = 0;
3      int index = 0;
4      for(int i = 0; i < size-1; i++) {
5          index = i;
6          menor = vet[i];
7          for(int j = i; j < size; j++) {
8              if(menor > vet[j]) {
9                  menor = vet[j];
10                 index = j;
11             }
12         }
13         int tmp = vet[i];
14         vet[i] = vet[index];
15         vet[index] = tmp;
16     }
17 }
```

A ideia da ordenação por seleção é achar o menor valor e, se este não estiver na posição correta, trocá-lo com a primeira posição, depois achar o segundo menor e colocá-lo na segunda posição e assim sucessivamente até que o vetor esteja completamente ordenado. É isso que está sendo feito nesta implementação. Fazendo a análise da complexidade.

Tabela 1: Índices

i	j
0	0 até **size-1**
1	1 até **size-1**
2	2 até **size-1**
.	.
.	.
.	.
size-2	**size-2** até **size-1**

Como é possível perceber na tabela cada vez que o índice i aumenta a quantidade de iterações do laço mais interno diminui até que este índice alcance o valor de size-2. Considerando-se, então um *size*=5, vão ocorrer 5+4+3+2, 14 execuções.

Colocando em termos mais gerais e substituindo size por n:

$$\left(\sum_{i=1}^{n} i\right) - 1 = (n+1) \cdot \frac{n}{2} - 1$$
$$(n+1) \cdot \frac{n}{2} - 1 = \frac{n^2+n}{2} - 1 \tag{3}$$

Então conclui-se que a ordem de complexidade é $\mathbf{O}(n^2)$.

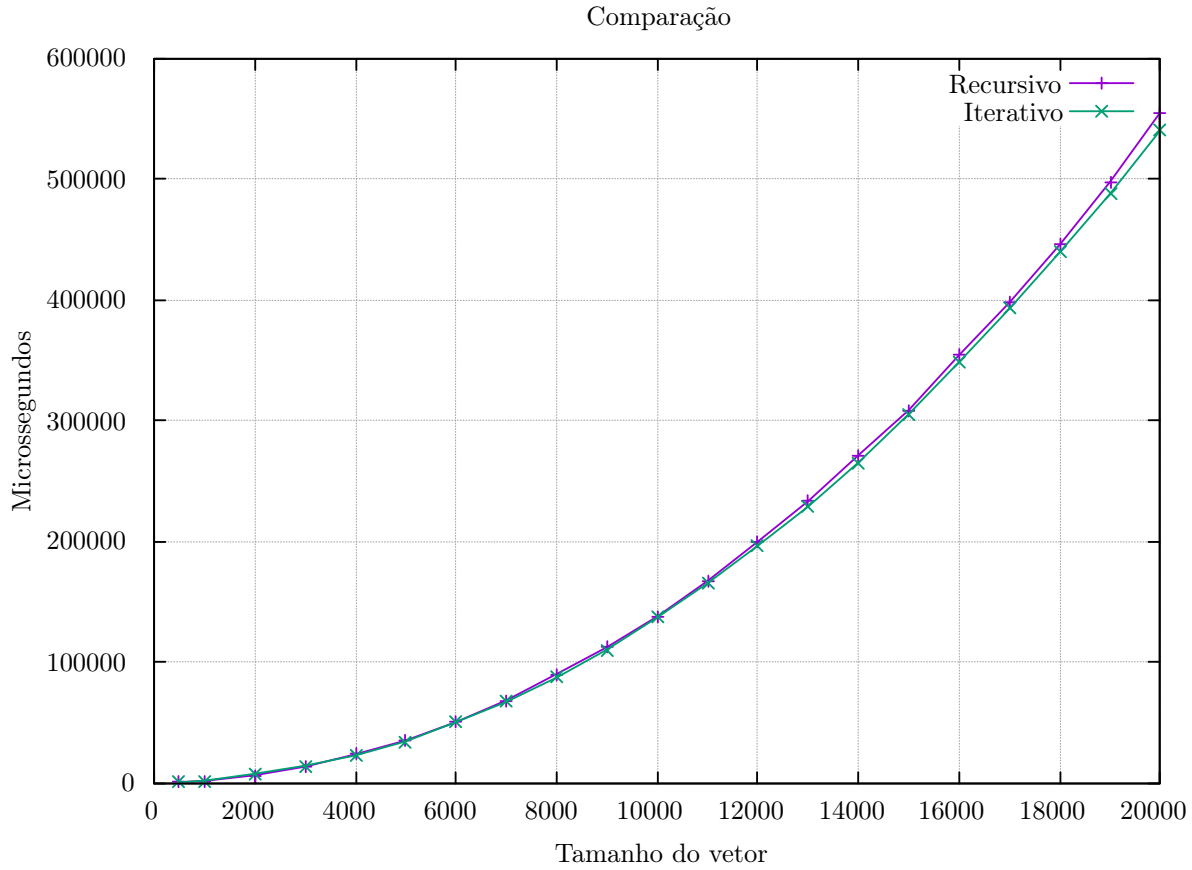
5.2 Recursivo

```
1  void recursive_selection_sort(int vet[], int size, int index
       , int min, int current_position) {
2      int i = current_position;
3      for(; i < size; i++) {
4          if(min > vet[i]) {
5              min = vet[i];
6              index = i;
7          }
8      }
9      int tmp = vet[current_position];
10     vet[current_position] = vet[index];
11     vet[index] = tmp;
12
13     if(current_position < size-1) {
14         recursive_selection_sort(vet, size, current_position
               +1, vet[current_position+1], current_position+1);
15
16     }else return;
17 }
```

Na implementação recursiva deste algoritmo foi optado por usar laço, pois na implementação sem o seu uso o algoritmo apresentou muita ineficiência, por que não conseguia ordenar nem 900 valores. Então, para uma análise mais igualitária foi feita desta forma. Pelo fato de os algoritmos apresentarem quase o mesmo número de atribuições acaba que eles apresentam um tempo de ordenação praticamente igual como pode ser visto no gráfico de comparação do selection sort. Ou seja, esta implementação recursiva vai apresentar uma eficiência de $\mathbf{O}(n^2)$, mas como esta usa recursos da pilha da memória, vai apresentar um tempo um pouco maior.

5.3 Comparação SelectionSort Recursivo e Iterativo

Comparando a versão recursiva com a iterativa é perceptível que apresentam quase sempre o mesmo tempo para o mesmo tamanho de vetor.



6 Merge Sort - Ordenação por intercalação

Este algoritmo usa a ideia do dividir para conquistar. Basicamente, se trata de dividir o vetor e ordenar os sub-vetores existentes. É um algoritmo extremamente bom, pois como será visto na análise o mergeSort consegue uma eficiência de $\mathbf{O}(n \log_2 n)$, que comparado aos outros dois vistos é bem mais rápido. Para tanto, é necessário uma função que intercala o vetor.

```
1  void merge(int vet[], int begin, int middle, int end) {
2      int size = end-begin+1;
3      int *vetAux = new int[size];
4      int beginAux = begin, midAux = middle+1;
5      int current_position = 0;
6
7      while(beginAux <= middle && midAux <= end) {
8          if(vet[beginAux] < vet[midAux]) {
9              vetAux[current_position++] = vet[beginAux++];
10         }else{
11             vetAux[current_position++] = vet[midAux++];
12         }
13     }
14     while(beginAux <= middle)
15         vetAux[current_position++] = vet[beginAux++];
16
17     while(midAux <= end)
18         vetAux[current_position++] = vet[midAux++];
19
20     for(int i = begin; i <= end; i++)
21         vet[i] = vetAux[i-begin];
22
23
24     delete[] vetAux;
25 }
```

6.1 Recursivo

A implementação recursiva acaba sendo um código bem simples, que basicamente começa pela primeira metade do vetor ordena totalmente em ordem crescente, depois vai para segunda metade e o ordena completamente e por fim ordena o vetor inteiro.

```
1  void recursive_merge_sort(int vet[], int begin, int end) {
2      if (begin < end) {
3          int middle = (end+begin)/2;
4
5          recursive_merge_sort(vet, begin, middle);
6          recursive_merge_sort(vet, middle+1, end);
7          merge(vet, begin, middle, end);
8      }
9  }
```

Este algoritmo pode ser pensado como uma árvore binária, pois a cada chamada recursiva o vetor é dividido mais e mais até resultar em um único elemento e retorná-lo. Então para facilitar os cálculos vão ser usadas entradas múltiplas de dois. Por exemplo 4 elementos:

```
                          C(n) [4, 1, 2, 3]
                               /            \
            2C(n/2)= C(n) [4, 1]          [2, 3]
                           /    \         /    \
         4C(n/4) = C(n) [4]     [1]     [2]    [3]
```

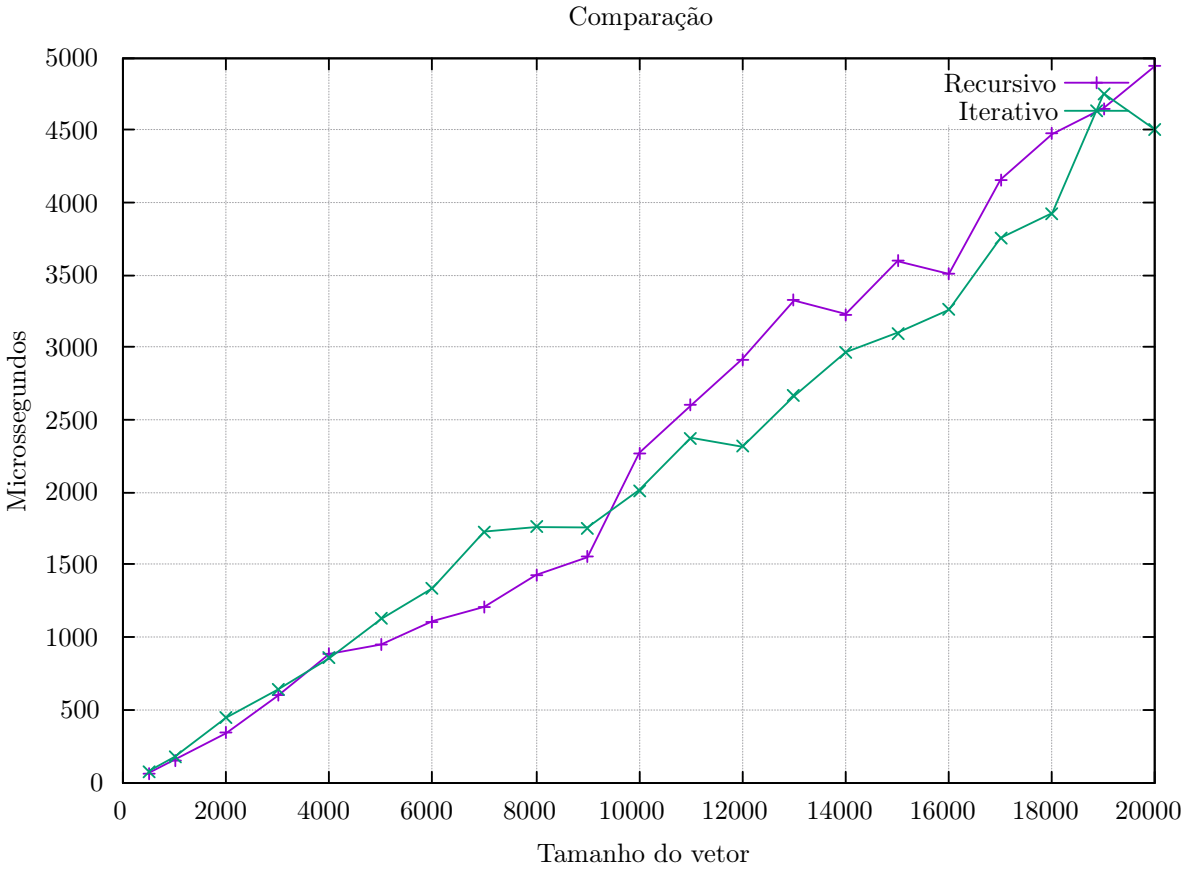
O C acima representa uma constante. Em termos gerais, a divisão é feita em tempo **O(1)**, mas restam dois sub-vetores que precisam ser resolvidos em tempo ágil, como para cada execução o vetor é dividido em dois, considerando l como nível da árvore, existiriam $l \cdot C \cdot n$ execuções. Como visto em árvores binárias l=$\log_2 n$+1. Por exemplo, se n=4, l=3, que é exatamente a quantidade de níveis apresentada na árvore acima. Descartando o +1, temos a notação igual a O(n $\log_2 n$). Como este algoritmo usa um vetor auxiliar na função de intercalação pode-se dizer que não é *in loco*.

6.2 Iterativo

```
void iteractive_merge_sort(int vet[], int begin, int end) {
    int middle = (begin+end)/2;
    for(int i = begin; i < middle; i++)
        merge(vet, begin, i, i+1);
    for(int i = middle; i < end; i++)
        merge(vet, middle+1, i, i+1);
    merge(vet, begin, middle, end);
}
```

A versão iterativa também apresenta um tempo de **O(n log n)**, pois o que é feito nele é um processo muito parecido com o que acontece em sua versão recursiva. A cada iteração, começando da primeira posição do vetor até a primeira metade. Depois da metade+1 até o fim. Após tudo isso a função merge é chamada, com um fim unicamente de ordenar o vetor inteiro. O iterativo nesta implementação vai ser um pouco melhor por não usar a pilha da memória, o que pode ser entendido no gráfico abaixo, na maior parte do tempo o iterativo ordenou o vetor em menos tempo.

6.3 Comparação MergeSort Recursivo e Iterativo



7 Quick Sort - Ordenação rápida

Este é um algoritmo que também é bem rápido. Ele usa uma estratégia de definir um pivô e fazer com sempre o elemento a esquerda do pivô seja menor e o elemento a direita maior. Por eu não precisar usar nenhum vetor auxiliar, diz-se que ele é *in loco*. Mas como escolher o pivô?

```
int partition(int vet[], int begin, int end) {
    int pivot = vet[begin];
    int left = begin, right = end-1;
    while(left < right) {
        while(vet[left] <= pivot)
            left++;
        while(vet[right] > pivot)
            right--;
        if(left < right)
            std::swap(vet[left], vet[right]);

    }
    vet[begin] = vet[right];
    vet[right] = pivot;
    return right; d
}
```

Este é um algoritmo que retorna nosso pivô, quando ele for retornado, é garantido que todos os elementos a esquerda dele são menores e os a direita maiores, embora eles estejam totalmente desordenados.

7.1 Recursiva

```
void recursive_quick_sort(int vet[], int begin, int end) {
    int pivot;
    if(begin < end) {
        pivot = partition(vet, begin, end);
        recursive_quick_sort(vet, begin, pivot);
        recursive_quick_sort(vet, pivot+1, end);
    }
}
```

Esta é a implementação recursiva do *Quick sort*, que basicamente ordena as partições que o pivô define. Na primeira chamada o pivô vai ser definido, exemplo:

```
vet = [3, 5, 9, 1, 0]
recursive_quick_sort(vet, 0, 5)
pivot = 2; // Este pivot representa a a posicao em que o
    pivo se encontra //
vet  = [1, 0, 3, 9, 5] // Vetor depois da primeira particao
    //
```

Observação: O pivô dentro da função recursiva sempre representará uma posição no vetor, já na função de partição ele vai receber um valor contido no vetor.

De posse do vetor da primeira partição e o início ainda menor que o fim, a primeira chamada recursiva acontece. Só que aqui ele vai particionar do início ao pivô.

```
vet = [1, 0, 3, 9, 5];
recursive_quick_sort(vet, 0, 2);
pivot = 1;
vet = [0, 1, 3, 9, 5] // Vetor apos a particao //
```

Quando a recursão é chamada pela primeira vez é como se realmente só a metade do vetor esteja passando, já que a função só terá acesso até o meio. Agora que não há mais como ordenar a primeira metade do vetor, ou seja, a pliha de execução dela encerrou, a próxima chamada recursiva é feita, lembrando que o pivô é a posição 2:

```
vet = [0, 1, 3, 9, 5];
recursive_quick_sort(vet, 3, 5);
pivot = 3;
vet = [0, 1, 3, 5, 9] ; // Vetor apos a particao //
```

Assim o vetor é ordenado, mas a chamada vai acontecer até que begin receba o tamanho do vetor para a recursão encerrar.

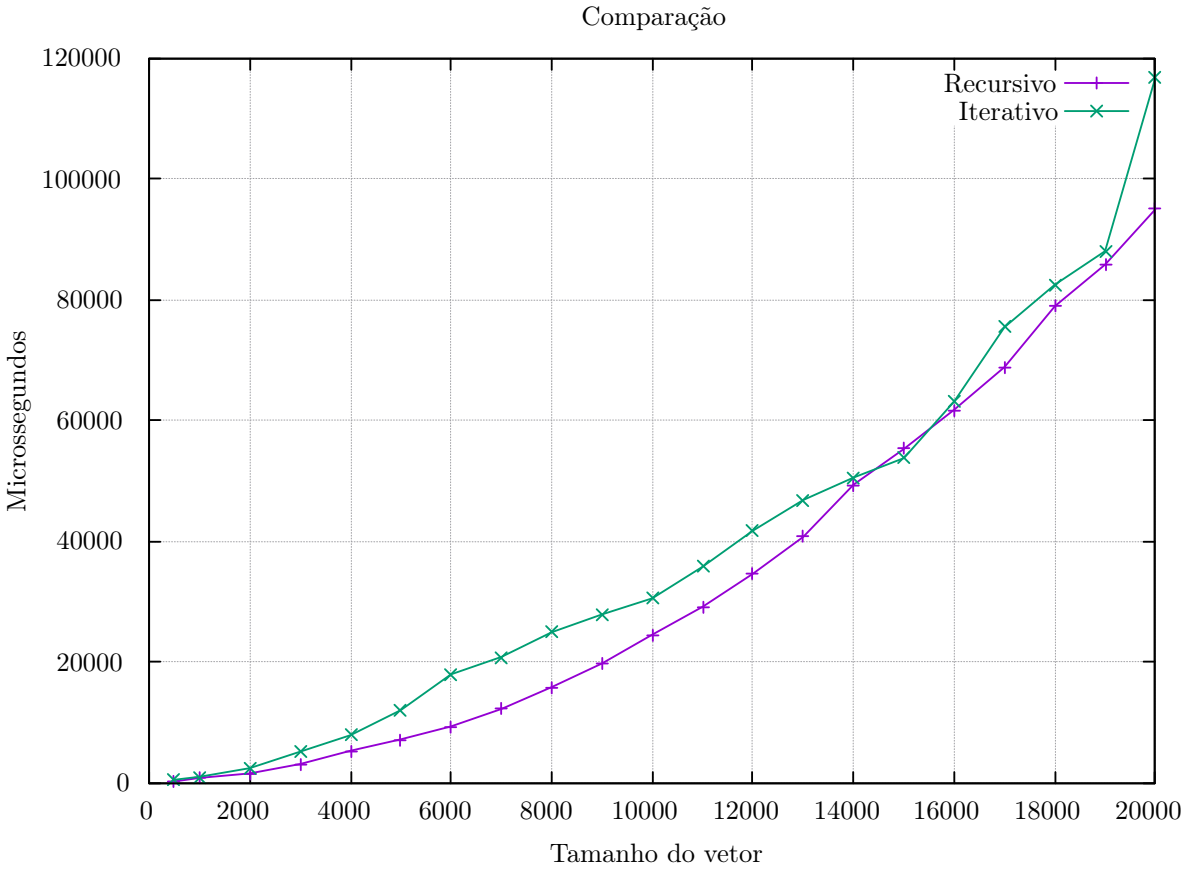
7.2 Iterativo

```
void iteractive_quick_sort(int vet[], int begin, int end) {
    std::stack<int> *st = new std::stack<int>[end-begin];
    st->push(begin);
    st->push(end);
    while (!st->empty()) {
        end = st->top();
        st->pop();
        begin = st->top();
        st->pop();
        int pivot = partition(vet, begin, end);
        if (pivot > begin) {
            st->push(begin);
            st->push(pivot);
        }
        if (pivot + 1 < end) {
            st->push(pivot + 1);
            st->push(end);
        }
    }

    delete[] st;

}
```

Na versão iterativa o uso de pilha facilita o processo de ordenação. A pilha é criada com intuito de simular o que ocorre na recursão que faz uso da pilha da

memória. Inicialmente uma nova pilha é alocada usando a biblioteca <stack> do próprio C++. Como primeiro passo os valores do inicio e fim do vetor são adicionados a pilha nesta ordem. Ocorre o laço que verifica se a pilha não está vazia, se estivesse o vetor estaria ordenado. O fim vai receber o valor contido no topo da pilha e logo após é removido, o mesmo acontece para o inicio logo em seguida. Agora ocorre a primeira partição e definição do pivô, agora acontece uma verificação se o pivô for maior que o início atual é adicionada na pilha o início e o pivô nesta ordem, assim como a posição posterior do pivô for menor que o fim. E assim acontece até que o vetor esteja ordenado, ou seja, pilha vazia.

7.3 Comparação Quick Sort Recursivo e Iterativo



A versão iterativa por usar uma função auxiliar ordena os elementos um pouco mais lento que a versão recursiva.