UFC

Relatório - Ordenadores Estrutura de Dados

Professor: Atílio Gomes Luiz

Fábio Luz Duarte Filho (474027)

Alysson Alexandre de Oliveira Araújo (474084)

Ciência da Computação

Universidade Federal do Ceará - Quixadá/CE - Brasil

1 Objetivo

Este relatório tem como objetivo reportar como funcionam diferentes métodos de ordenação feitos em C++ pelos os alunos Fábio Luz e Alysson Araújo pedido pelo professor Atílio Gomes, no qual são feitos 18 testes com 6 algoritmos de ordenação (Bubble Sort, Selection Sort, Merge Sort, Heap Sort, Insertion Sort e Quick Sort). Cada um desses algoritmos foram implementados em três formas diferentes: iterativos e recursivos usando Vetores; e iterativos usando Lista Duplamente Encadeada Circular.

2 Divisão de Trabalho e Dificuldades Encontradas

A divisão de trabalho na dupla deu-se da seguinte maneira: Alysson tratou de alguns algoritmos em vetor e o realizado em lista, além de executar os teste em sua máquina. Fábio tratou também de alguns algoritmos em vetor, e da criação dos gráficos e deste relatório.

As dificuldades encontradas foram: a gestão do tempo para realização do trabalho, principalmente pela ocupação em período de fim de semestre; a divisão do trabalho em sim foi árdua; a comunicação entre a dupla; a execução dos testes (que é demorada, por volta de 1 dia inteiro para execução total) e da própria confecção dos algoritmos, principalmente nas mudanças entre

iterativos e recursivos.

3 Testes

Os testes são realizados em vetores/listas com números aleatoriamente gerados de tamanhos N: 1.000, 5.000, 10.000, 50.000, 100.000, 200.000, 300.000, 400.000, 500.000, 750.000 e 1.000.000. Para cada valor de N, são executadas 5 sementes diferentes.

4 Algoritmos em Vetores

4.1 Bubble Sort

O funcionamento do Bubble Sort é baseado em percorrer a estrutura diversas vezes fazendo, a cada passagem, o maior elemento "flutuar"para o topo, como bolhas num tanque.

4.1.1 Iterativo

```
1  void iterativeBubbleSort (int *array, int size) {
2  bool swapped;
3  for (int i = 0; i < size-1; i++) {
4      swapped = false;
5      for (int j = size-1; j > i; j--)
6          if (array[j] < array[j-1]) {
7              swap(&array[j], &array[j-1]);
8              swapped = true;
9          }
10         if (swapped == false)
11             break;
12     }
13 }
```

4.1.2 Recursivo

```
1  void recursiveBubbleSort (int *array, int size) {
2      if (size < 1)
3          return;
4      bool swapped = false;
5      for (int i = 0; i < size-1; i++)
6          if (array[i] > array[i+1]) {
7              swap(&array[i], &array[i+1]);
```

```
8               swapped = true;
9           }
10      if (swapped == false)
11          return;
12      recursiveBubbleSort(array, size-1);
13  }
```

Obs.: Os testes executados no algoritmo recursivo resultam em Segmentation Fault para entradas a partir de 200.000 devido ao grande número de chamadas de funções recursivas, fato não suportado pela máquina de teste.

4.2 Insertion Sort

Sobre o funcionamento do Insertion Sort: dado uma estrutura, constrói uma matriz final com um elemento de cada vez, uma inserção por vez. Como organiza-se cartas de baralho na mão. Percorre-se as posições da estrutura, começando com pela a[1]. Cada nova posição é como uma nova carta recebida, que será inserida no lugar correto na subestrutura ordenada à esquerda daquela posição.

4.2.1 Iterativo

```
1  void iterativeInsertionSort (int *array, int size) {
2      for (int i = 1; i < size; i++) {
3          int aux = array[i];
4          int j = i-1;
5          while (j >= 0 && array[j] > aux) {
6              array[j+1] = array[j];
7              j--;
8          }
9          array[j+1] = aux;
10     }
11 }
```

4.2.2 Recursivo

```
1  void recursiveInsertionSort (int* array, int size) {
2      if(size <= 1)
3          return;
4      recursiveInsertionSort(array, size-1);
5
6      int k = size-2;
```

```
7       int aux = array[size-1];
8
9       for(; array[k] > aux && k >= 0; k--) {
10          array[k+1] = array[k];
11      }
12      array[k+1] = aux;
13  }
```

Obs.: Os testes executados no algoritmo recursivo resultam em Segmentation Fault para entradas a partir de 100.000 devido ao grande número de chamadas de funções recursivas, fato não suportado pela máquina de teste.

4.3 Selection Sort

O funcionamento do Selection Sort baseia-se em percorrer a estrutura movendo o menor elemento para a posição a[0], o segundo menor para a[1] e assim sucessivamente para os n-1 elementos, como no algoritmo iterativo. Outra forma, é mover o maior para a[n-1], o segundo maior para a[n-2] e assim sucessivamente, como no algoritmo recursivo.

4.3.1 Iterativo

```
1  void iterativeSelectionSort (int *array, int size) {
2      for (int i = 0; i < size; i++) {
3          int minor = i;
4          for (int j = i+1; j < size; j++) {
5              if (array[j] < array[minor]) minor = j;
6          }
7          if (array[i] != array[minor])
8              swap(&array[i], &array[minor]);
9      }
10 }
```

4.3.2 Recursivo

```
1  void recursiveSelectionSort (int* array, int size) {
2      if (size <= 1)
3          return;
4
5      int major = 0;
6      for (int i = 1; i < size; i++) {
7          if (array[i] > array[major])
```

```
8               major = i;
9       }
10      if (array[size-1] != array[major])
11          swap (&array[size-1], &array[major]);
12
13      recursiveSelectionSort(array, size-1);
14  }
```

Obs.: Os testes executados no algoritmo recursivo resultam em Segmentation Fault para entradas a partir de 100.000 devido ao grande número de chamadas de funções recursivas, fato não suportado pela máquina de teste.

4.4 Merge Sort

O Merge Sort é um algoritmo do tipo Dividir para Conquistar. Consiste em dividir a estrutura em metades até que haja 1 ou 2 elementos nas subestruturas. Essa é a Divisão. Com isso, é chamada uma função (Merge) para unir essas subestruturas de modo que as ordena. Essa é a Conquista. Ao final de tudo, tem-se a estrutura ordenada.

4.4.1 Função auxiliar: Merge

```
1   void merge (int *array, int begin, int middle, int end) {
2       int iAux, jAux, kAux;
3       int *aux_array = new int[end-begin+1];
4       iAux = begin; jAux = middle+1; kAux = 0;
5
6       while (iAux <= middle && jAux <= end) {
7           if (array[iAux] <= array[jAux])
8               aux_array[kAux++] = array[iAux++];
9           else
10              aux_array[kAux++] = array[jAux++];
11      }
12      while (iAux <= middle) aux_array[kAux++] = array[iAux++];
13      while (jAux <= end) aux_array[kAux++] = array[jAux++];
14
15      for (iAux = begin; iAux <= end; iAux++)
16          array[iAux] = aux_array[iAux-begin];
17
18      delete[] aux_array;
19  }
```

4.4.2 Iterativo

```
1 void iterativeMergeSort (int *array, int size) {
2    for (int actual_size = 1; actual_size <= size-1; actual_size *=
        2) {
3        for (int left = 0; left < size-1; left += 2*actual_size) {
4            int middle = (left+actual_size-1 < size-1) ? left+
                actual_size-1 : size-1;
5            int right = (left+2*actual_size-1 < size-1) ? left+2*
                actual_size-1 : size-1;
6            merge(array, left, middle, right);
7        }
8    }
9 }
```

4.4.3 Recursivo

```
1 void recursiveMergeSort (int *array, int begin, int end) {
2    if (begin < end) {
3        int middle = (begin + end)/2;
4        recursiveMergeSort(array, begin, middle);
5        recursiveMergeSort(array, middle + 1, end);
6        merge(array, begin, middle, end);
7    }
8 }
```

4.5 Heap Sort

O Heap Sort é provavelmente o que possui a ideia mais complexa dentre os algoritmos desse trabalho .Tal ideia é próxima a do SelectionSort, que é buscar o maior elemento dentro do vetor e colocá-lo no fim do vetor. No entanto para fazer isso é necessário transformar o vetor a ser ordenado em um Heap, podendo ser o Max-Heap ou Min-Heap. O usado foi o Max-Heap.

Deve-se ver o Heap como uma árvore binária completa, mesmo que não seja realmente implementada um. No Heap vetor as posições dos elementos vão de A[1, ...,n]. Voltando a ideia da árvore binária completa com o Heap, na sua raiz ficará o maior valor que está presente no vetor e no Heap, os nós do último nível estão o mais a esquerda possível. Ela funciona da seguinte maneira: a raiz ou pai irá verificar qual dos seus filhos é maior que ele, caso um deles seja será feita a troca de valores do filho com o pai que eles possuem. O objetivo dele é fazer isso até que os pais e a raiz sejam maiores que os seus filhos.

Depois de ser feita a comparação e troca dos valores com todos (caso seja necessário), todos os pais serão maiores que os seus filhos e (é importante saber que a posição da raiz é a[1] do Heap-Vetor), é feita uma troca com o valor que está na posição a[n](no caso o último elemento do vetor) com o valor da raiz e, em seguida, a posição a[n] não será mais comparada com ninguém e para que isso ocorra, a visão do vetor deve ser diminuída fazendo que a[n] não seja mais visto pelo o algoritmo e quando for feita a organização dos valores para todos os pais serem maiores que seus filhos, será trocado o valor da raiz com o elemento que está na posição a[n-1]. Depois disso começa tudo novamente e o valor do elemento que a raiz irá trocar será a[n-2], a[n-3]... até que n-k seja igual a 1, significando que o vetor foi ordenado.

4.5.1 Funções auxiliares: Build Heap e Sieve

```
1 void buildHeap (int *array, int size) {
2     for (int l = 1; l < size; l++)
3         for(int a = l + 1; a > 1 && array[a/2] < array[a]; a = a/2)
4             swap(&array[a/2], &array[a]);
5 }
```

```
1 void sieve (int* array, int size){
2     for(int k = 2; k <= size; k = k*2) {
3         if (array[k] < array[k+1] && k < size)
4             k++;
5         if (array[k/2] >= array[k])
6             break;
7         swap(&array[k/2], &array[k]);
8     }
9 }
```

4.5.2 Iterativo

```
1 void iterativeHeapSort (int* array, int size){
2     buildHeap(array,size);
3     int x = size;
4     while(x >= 2){
5         swap(&array[1], &array[x]);
6         sieve(array, x-1);
7         x--;
8     }
9 }
```

4.5.3 Recursivo

A dupla não foi capaz de produzir o Heap Sort Recursivo a tempo da entrega do trabalho.

4.6 Quick Sort

O Quick Sort, como o Merge Sort, é um algoritmo de ordenação do tipo Dividir para Conquistar, porém, com uma outra lógica. A Divisão é feita dividindo a estrutura em uma parte com elementos menores e outra com elementos maiores do que um certo pivô (elemento qualquer). Isso é feito até que se tenha as menores partições possíveis. Logo, vem a Consquista, a união coordenada destas partições.

4.6.1 Função auxiliar: Partitionate

```
1  int partitionate (int *array, int begin, int end) {
2      int pivot = array[end];
3      int i = begin-1;
4      for (int j = begin; j <= end-1; j++) {
5          if (array[j] < pivot) {
6              i++;
7              swap(&array[i], &array[j]);
8          }
9      }
10     swap(&array[i+1], &array[end]);
11     return i+1;
12 }
```

4.6.2 Iterativo

```
1  void iterativeQuickSort(int *array, int begin, int end) {
2      int stack[end-begin+1];
3      int top = 0;
4      stack[top] = begin;
5      top++;
6      stack[top] = end;
7
8      while (top >= 0) {
9          end = stack[top];
10         top--;
11         begin = stack[top];
12         top--;
```

```
13
14          int middle = partitionate(array, begin, end);
15
16          if (middle-1 > begin) {
17              top++;
18              stack[top] = begin;
19              top++;
20              stack[top] = middle-1;
21          }
22
23          if (middle+1 < end) {
24              top++;
25              stack[top] = middle+1;
26              top++;
27              stack[top] = end;
28          }
29      }
30  }
```

4.6.3 Recursivo

```
1  void recursiveQuickSort (int *array, int begin, int end) {
2      while (begin < end) {
3          int middle = partitionate(array, begin, end);
4          if (middle-begin < end-middle) {
5              recursiveQuickSort(array, begin, middle-1);
6              begin = middle+1;
7          }
8          else {
9              recursiveQuickSort(array, middle+1, end);
10             end = middle-1;
11         }
12     }
13 }
```

5 Algoritmos em Listas

5.1 Bubble Sort

```
1  void List::bubbleSort() {
2      Node* list = head;
3      if(list->next == list && list->prev == list)
4          return;
5
6      Node* last = list->prev;
7      Node* aux = list->next;
8      while(aux != list) {
9          Node* aux2 = last;
10         while(aux2 != aux) {
11             if((aux2->prev)->key > aux2->key)
12                 swap((aux2->prev)->key,aux2->key);
13             aux2 = aux2->prev;
14         }
15         aux = aux->next;
16     }
17 }
```

5.2 Insertion Sort

A dupla não foi capaz de produzir o Insertion Sort em Lista a tempo da entrega do trabalho.

5.3 Selection Sort

A dupla não foi capaz de produzir o Selection Sort em Lista a tempo da entrega do trabalho.

5.4 Merge Sort

A dupla não foi capaz de produzir o Merge Sort em Lista a tempo da entrega do trabalho.

5.5 Heap Sort

A dupla não foi capaz de produzir o Heap Sort em Lista a tempo da entrega do trabalho.

5.6 Quick Sort

A dupla não foi capaz de produzir o Quick Sort em Lista a tempo da entrega do trabalho.

6 Cálculos e Complexidades

Nestes tópicos são dispostas as complexidades e os cálculos de: O número de comparações realizadas e o número de cópias feitas pelos algoritmos.

6.1 Bubble Sort

6.1.1 Complexidades

No pior caso, o Bubble Sort possui complexidade $O(n^2)$
No melhor caso, o Bubble Sort possui complexidade $O(n)$

6.1.2 Número de comparações

Melhor caso:

$$I(n) = (n-1) + 1 = n$$

Pior caso:

$$I(n) = \sum_{i=1}^{n-1}(n-i+1) = \frac{n(n+1)}{2}$$

6.1.3 Número de cópias

Melhor caso:

$$C(n) = 0$$

Pior caso:

$$C(n) = \sum_{i=1}^{n-1}(n-i) = \frac{n(n+1)}{2} - n = \frac{n(n+1)-2n}{2} = \frac{n(n+1-2)}{2} = \frac{n(n-1)}{2}$$

6.2 Insertion Sort

6.2.1 Complexidades

No pior caso, o Insertion Sort possui complexidade $O(n^2)$
No melhor caso, o Insertion Sort possui complexidade $O(n)$

6.2.2 Número de comparações

Melhor caso:

$$I(n) = n - 1$$

Pior caso:

$$I(n) = \sum_{i=1}^{n-1}(n-i) = \frac{n(n+1)}{2} - n = \frac{n(n+1)-2n}{2} = \frac{n(n+1-2)}{2} = \frac{n(n-1)}{2}$$

6.2.3 Número de cópias

Melhor caso:

$$C(n) = 0$$

Pior caso:

$$C(n) = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 = \frac{n(n+1)-2}{2} = \frac{n^2+n-2}{2} = \frac{(n-1)(n+2)}{2}$$

6.3 Selection Sort

6.3.1 Complexidades

No pior caso, o Selection Sort possui complexidade $O(n^2)$
No melhor caso, o Selection Sort possui complexidade $O(n^2)$

6.3.2 Número de comparações

Melhor caso:

$$I(n) = \sum_{i=1}^{n-1}(n-i+1) = \frac{n(n+1)}{2}$$

Pior caso:

$$I(n) = \sum_{i=1}^{n-1}(n-i+1) = \frac{n(n+1)}{2}$$

6.3.3 Número de cópias

Melhor caso:

$$C(n) = 0$$

Pior caso:

$$C(n) = \lfloor \frac{n}{2} \rfloor$$

6.4 Merge Sort

6.4.1 Complexidades

No pior caso, o Merge Sort possui complexidade $O(nlgn)$
No melhor caso, o Merge Sort possui complexidade $O(nlgn)$

6.4.2 Número de comparações

Melhor caso:

$$I(n) = \lfloor \frac{n}{2} \rfloor \lg n$$

Pior caso:

$$I(n) = \lfloor \frac{n}{2} \rfloor \lg n$$

6.4.3 Número de cópias

Melhor caso:

$$C(n) = 0$$

Pior caso:

$$C(n) = 2(nlgn + 2^{\lg n+1} - (2^{\lg n+2} - 2n))$$

6.5 Heap Sort

6.5.1 Complexidades

No pior caso, o Heap Sort possui complexidade $O(nlgn)$
No melhor caso, o Heap Sort possui complexidade $O(nlgn)$

6.5.2 Número de comparações

Melhor caso:

$$I(n) = n - 2$$

Pior caso:

$$I(n) = 3n \lg n$$

6.5.3 Número de cópias

Melhor caso:

$$C(n) = 0$$

Pior caso:

$$C(n) = 2n - 2 + (n-1) \lg n$$

6.6 Quick Sort

6.6.1 Complexidades

No pior caso, o Quick Sort possui complexidade $O(n^2)$
No melhor caso, o Quick Sort possui complexidade $O(nlgn)$

6.6.2 Número de comparações

Melhor caso:

$$I(n) = \lfloor \frac{n}{2} \rfloor + n$$

Pior caso:

$$I(n) = \sum_{i=0}^{n-1} (n-i) = \frac{n(n+1)}{2}$$

6.6.3 Número de cópias

Melhor caso:

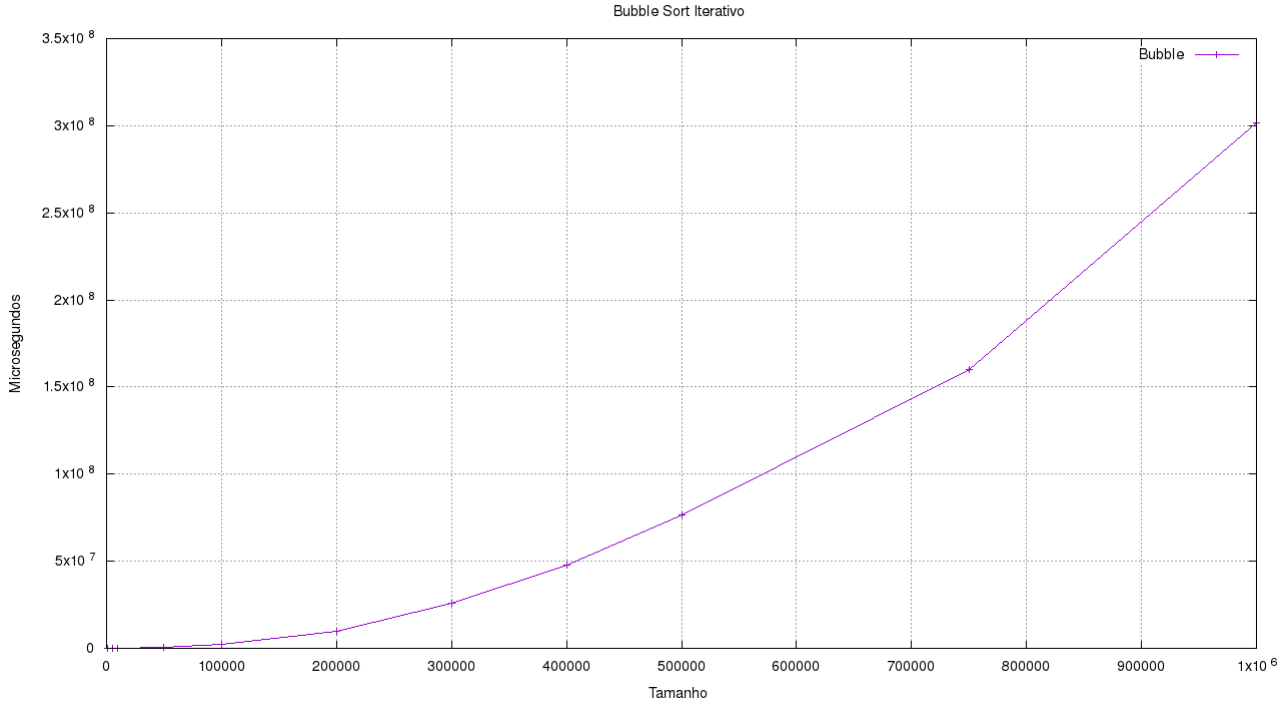
$$C(n) = 0$$

Pior caso:

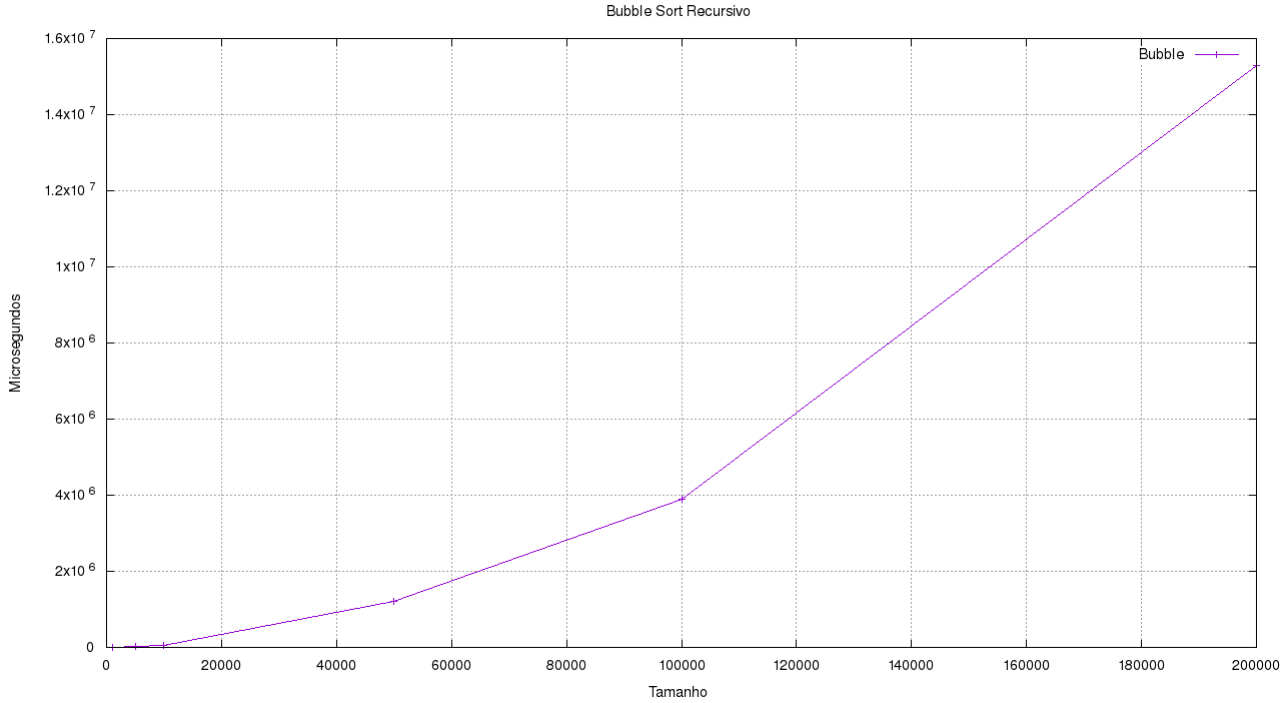
$$C(n) = \sum_{i=0}^{n-1}(n-i) = \frac{n(n+1)}{2}$$

7 Gráficos

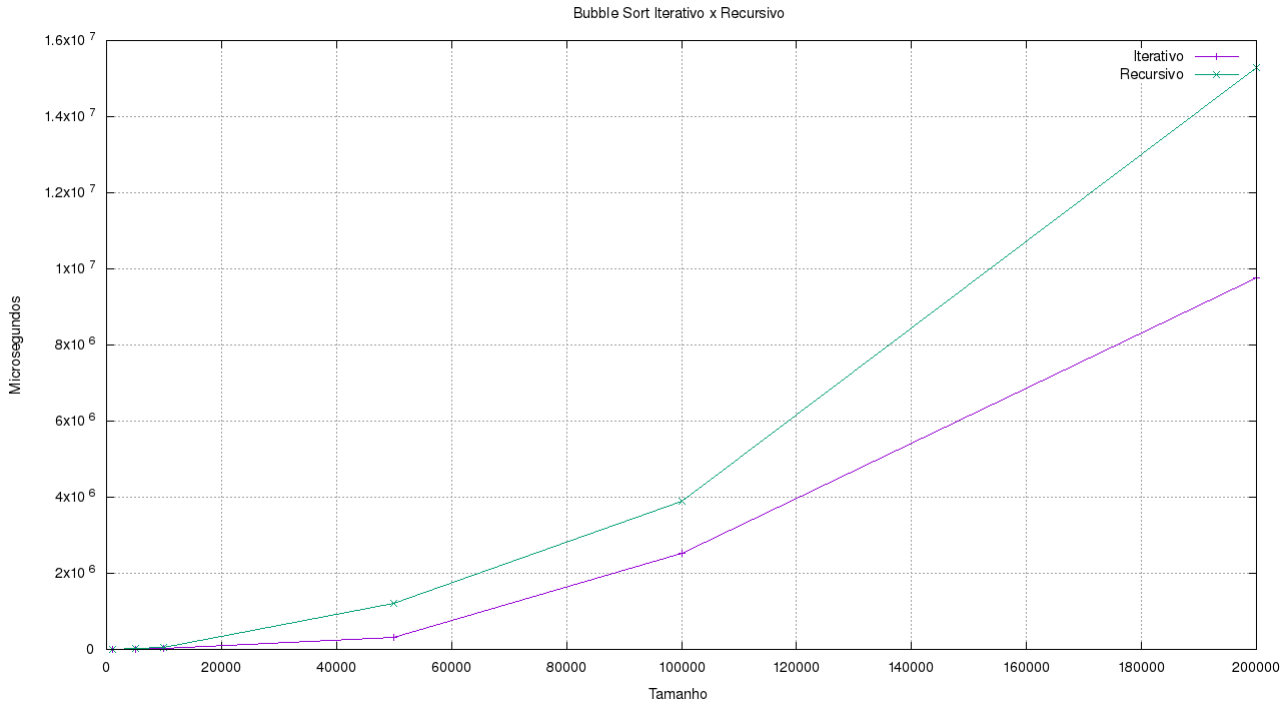
7.1 Bubble Sort Iterativo



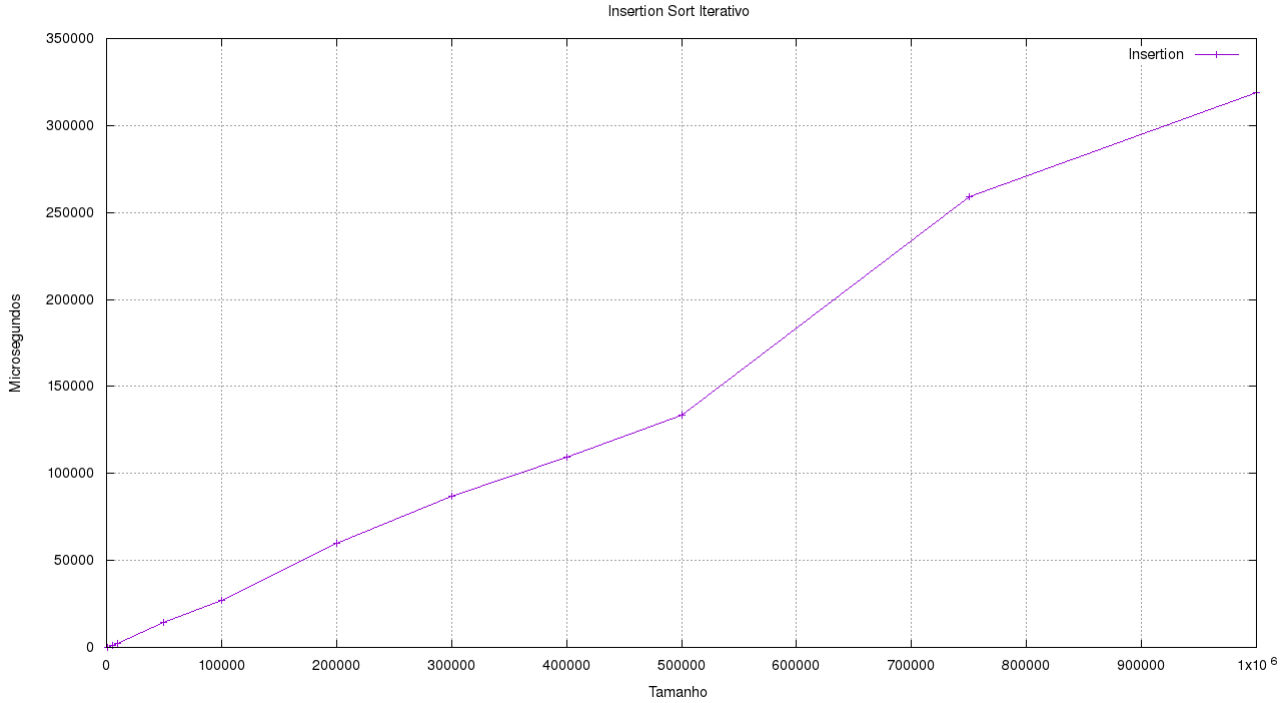
7.2 Bubble Sort Recursivo



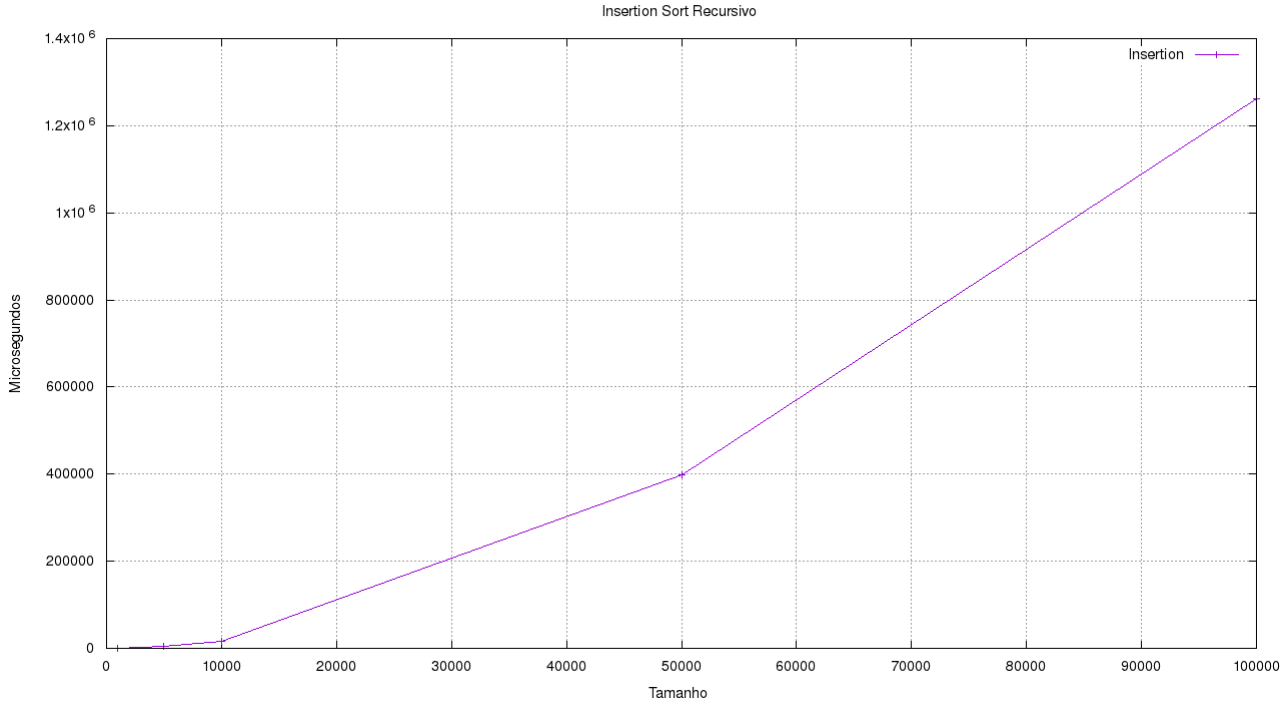
7.3 Comparação entre os Bubble Sort Iterativo e Recursivo



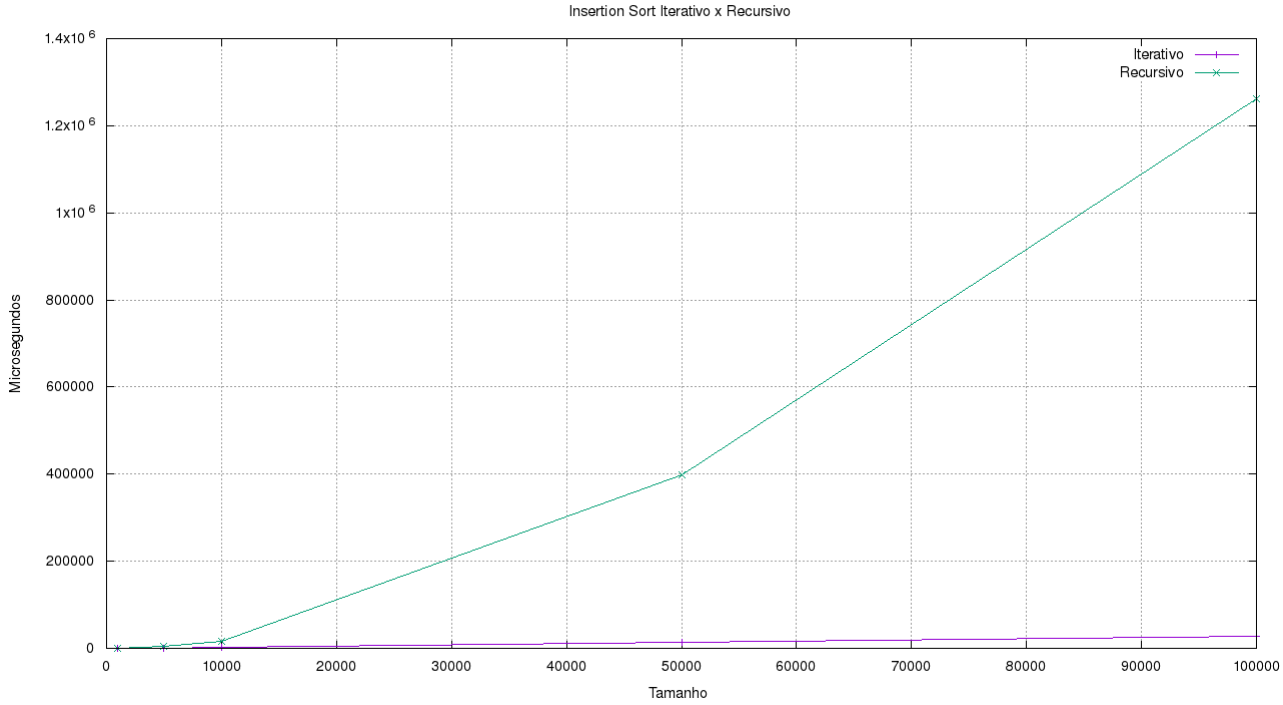
7.4 Insertion Sort Iterativo



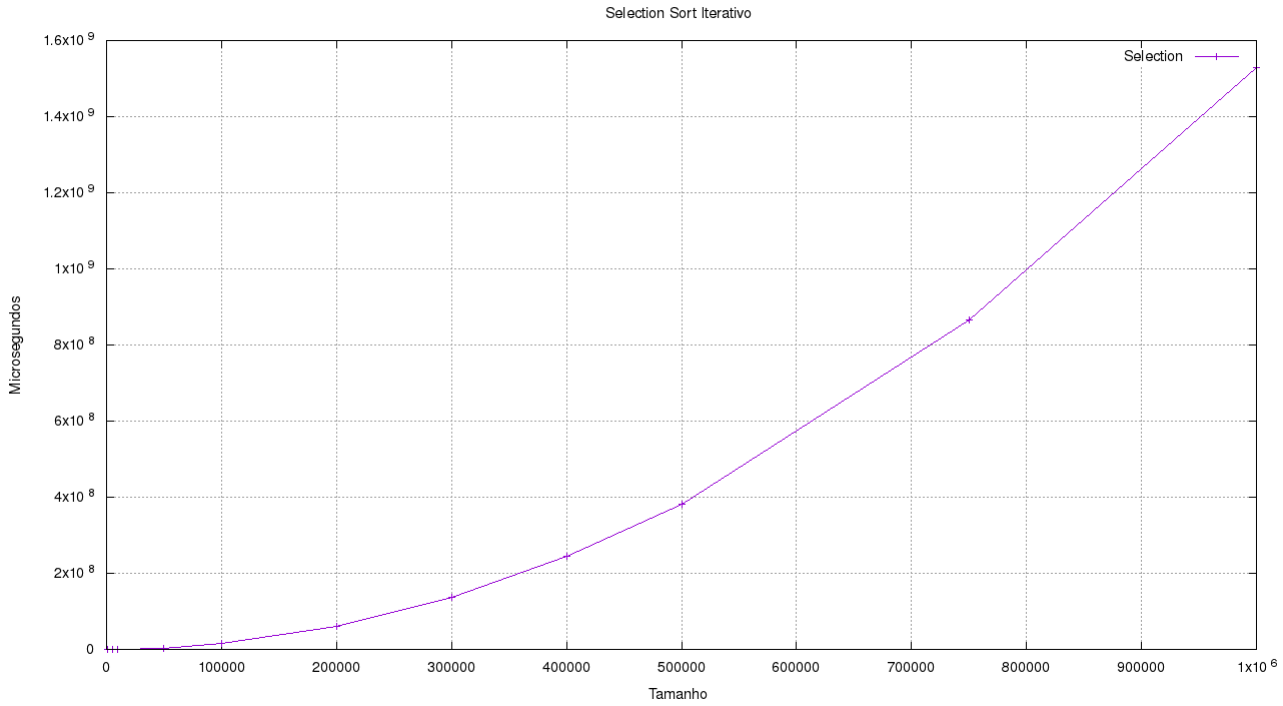
7.5 Insertion Sort Recursivo



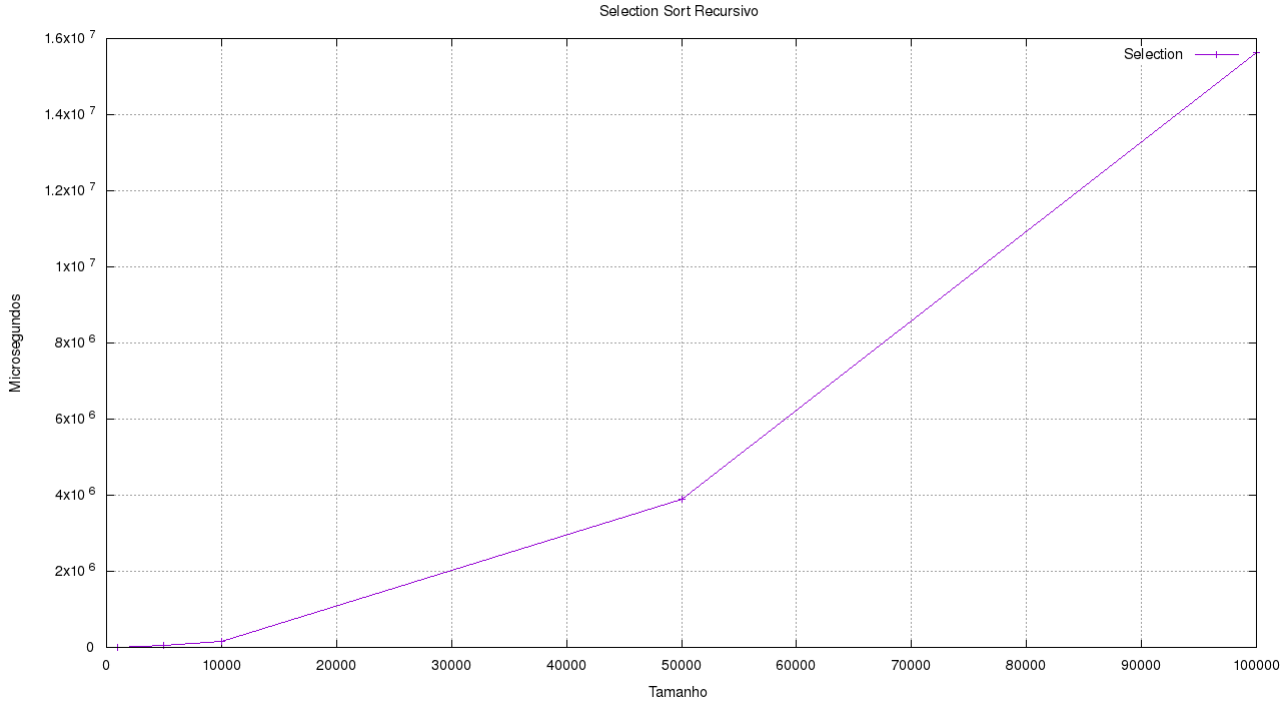
7.6 Comparação entre os Insertion Sort Iterativo e Recursivo



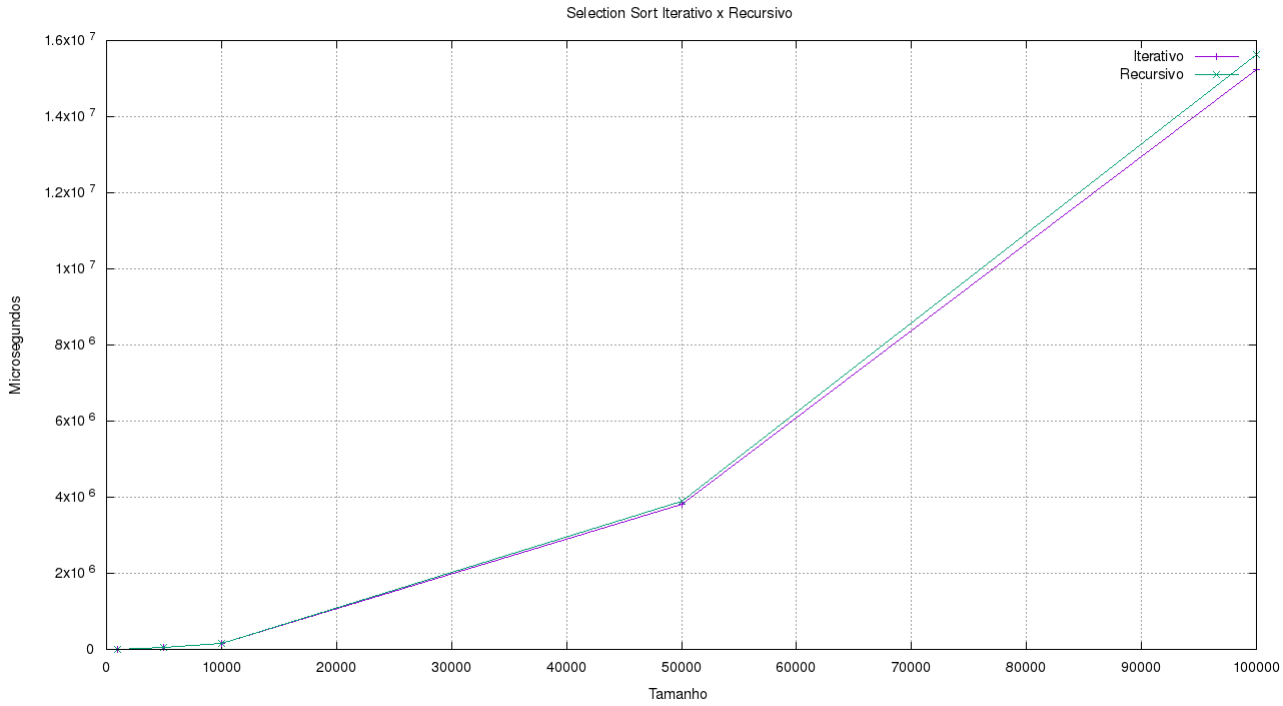
7.7 Selection Sort Iterativo



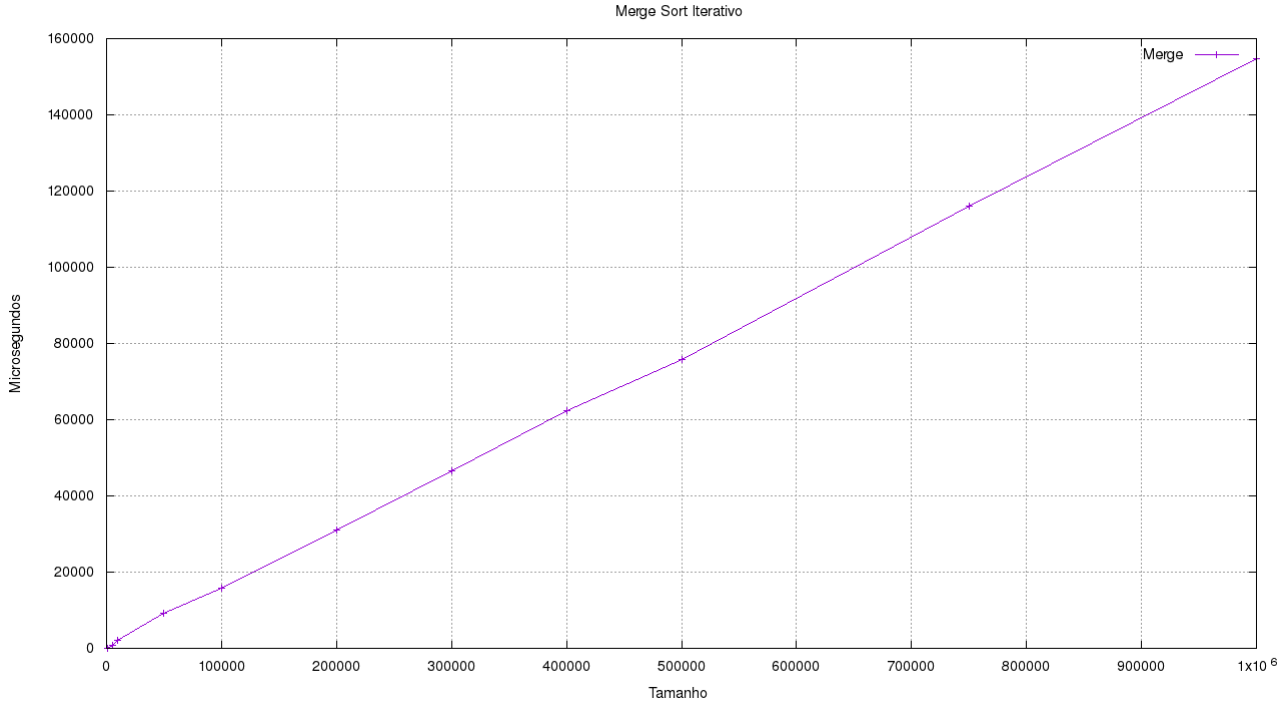
7.8 Selection Sort Recursivo



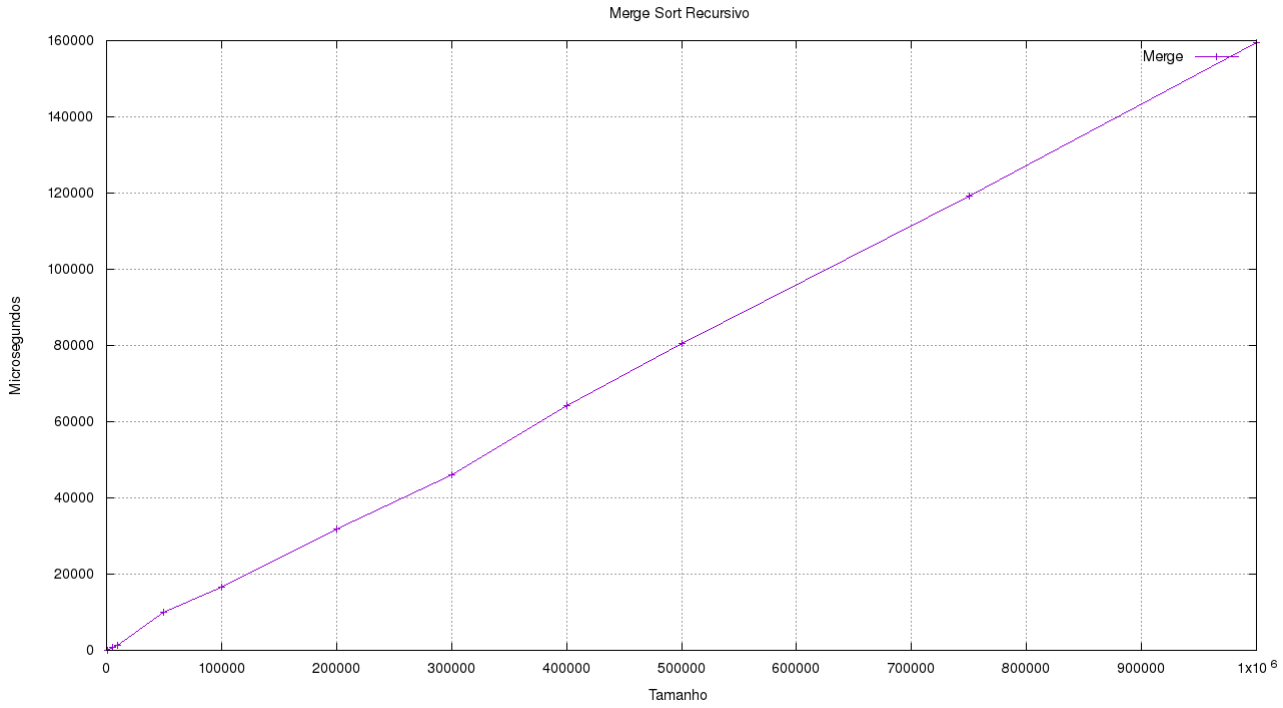
7.9 Comparação entre os Selection Sort Iterativo e Recursivo



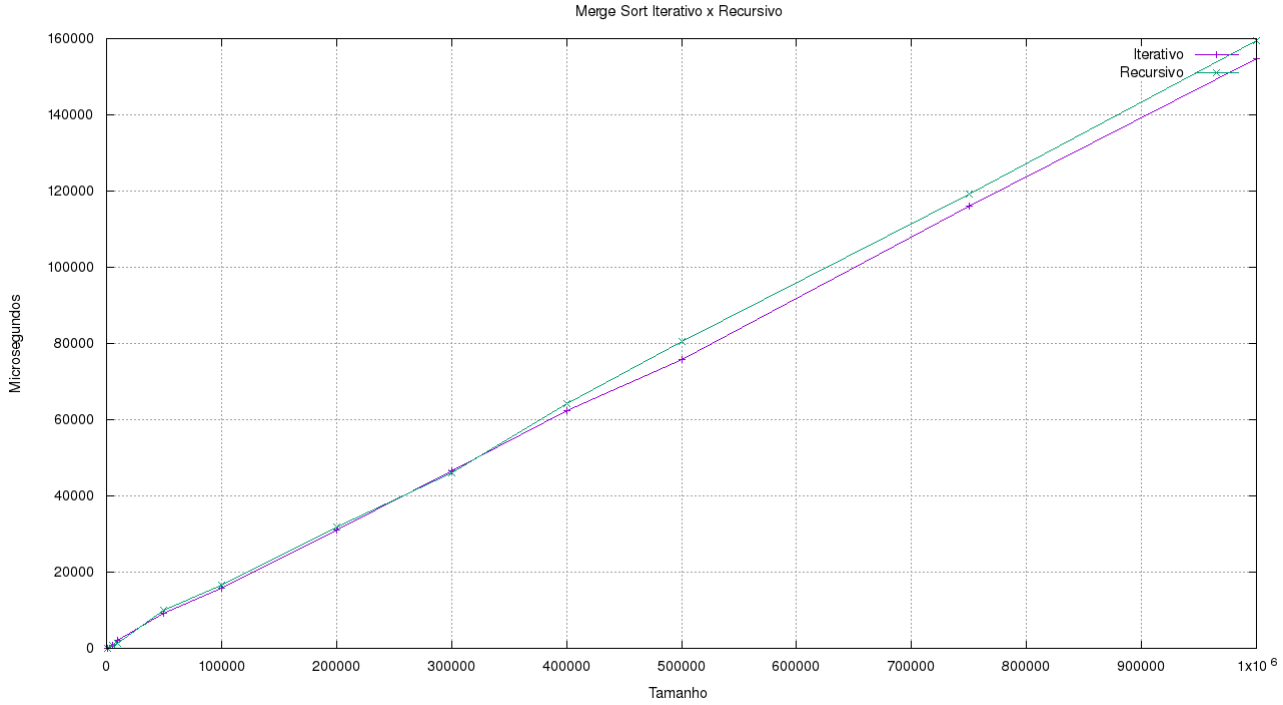
7.10 Merge Sort Iterativo



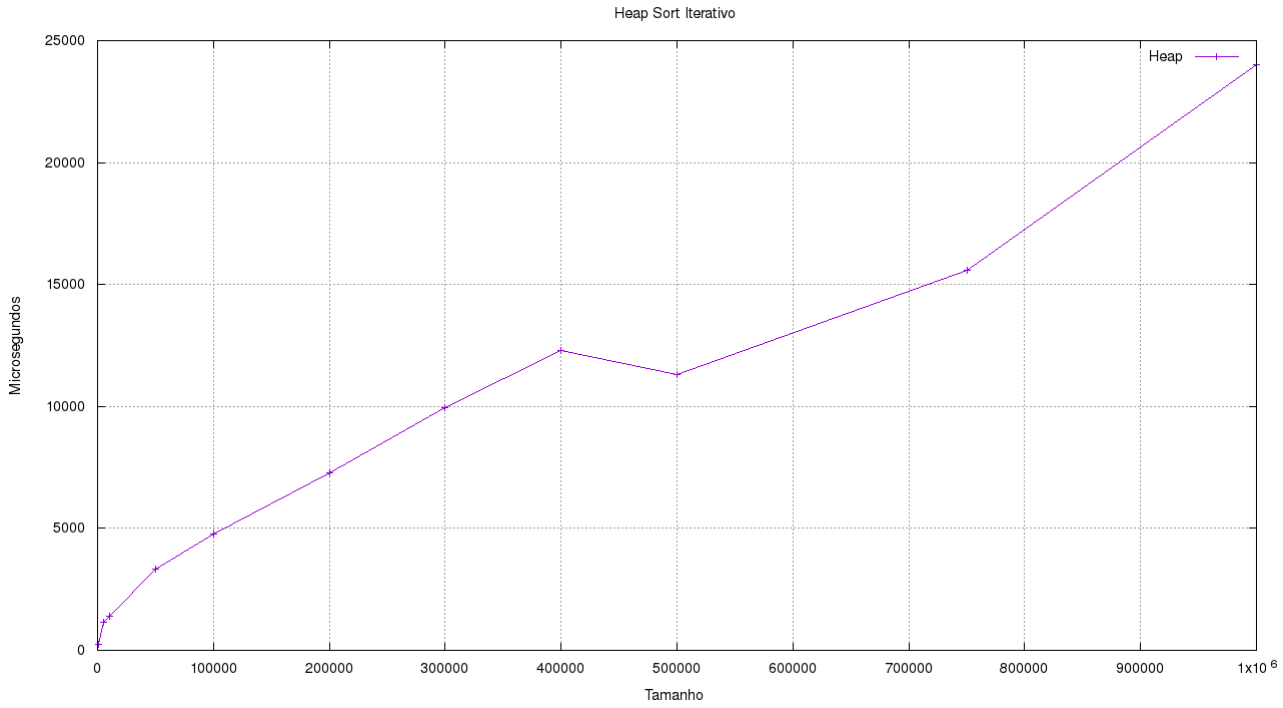
7.11 Merge Sort Recursivo



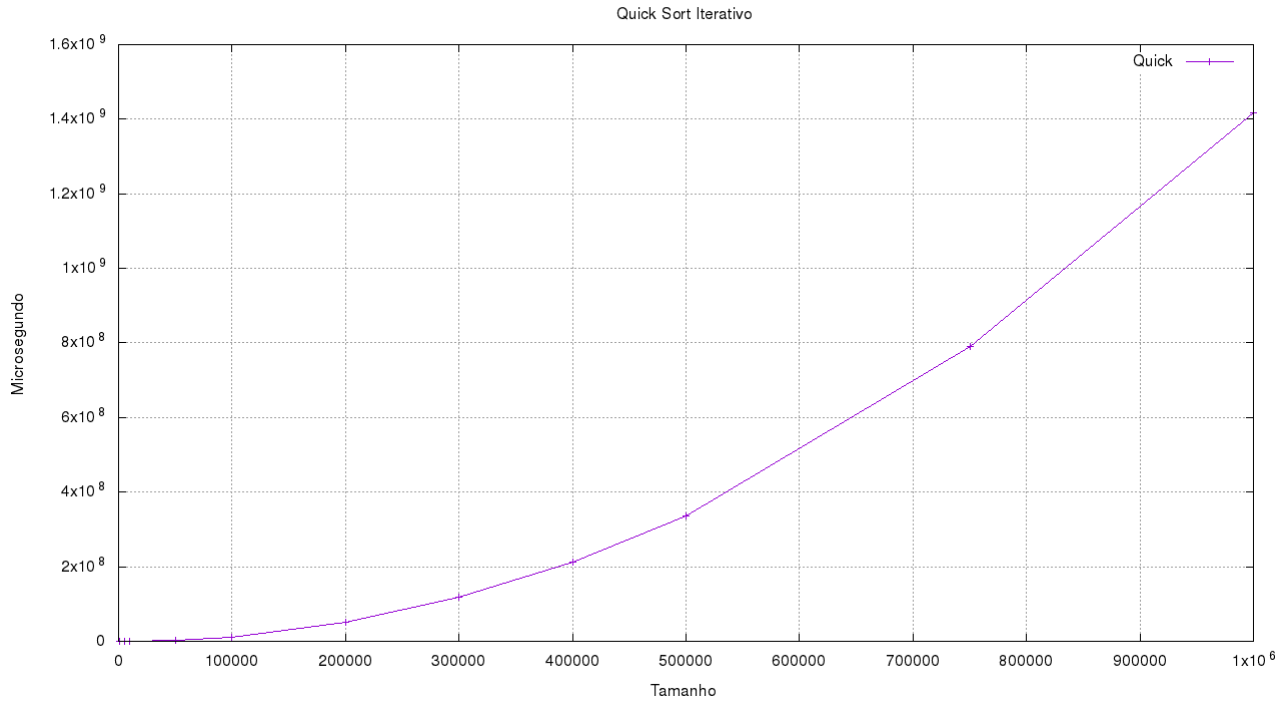
7.12 Comparação entre os Merge Sort Iterativo e Recursivo



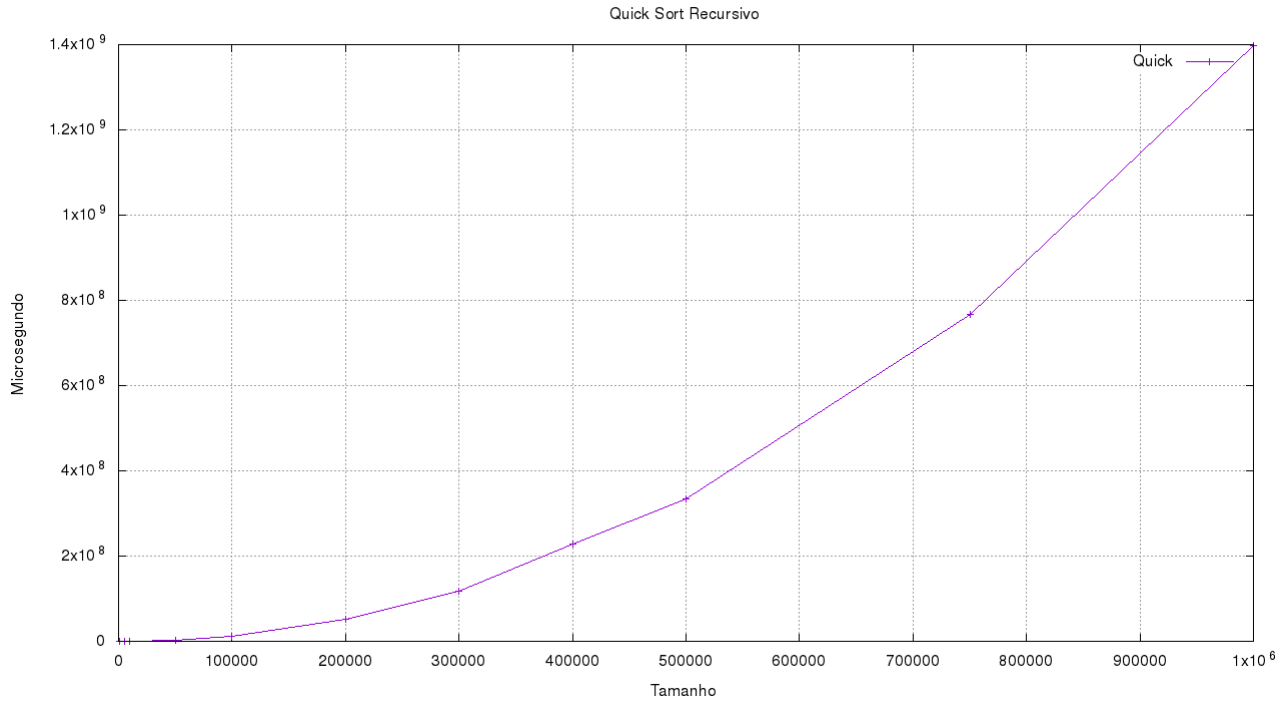
7.13 Heap Sort Iterativo



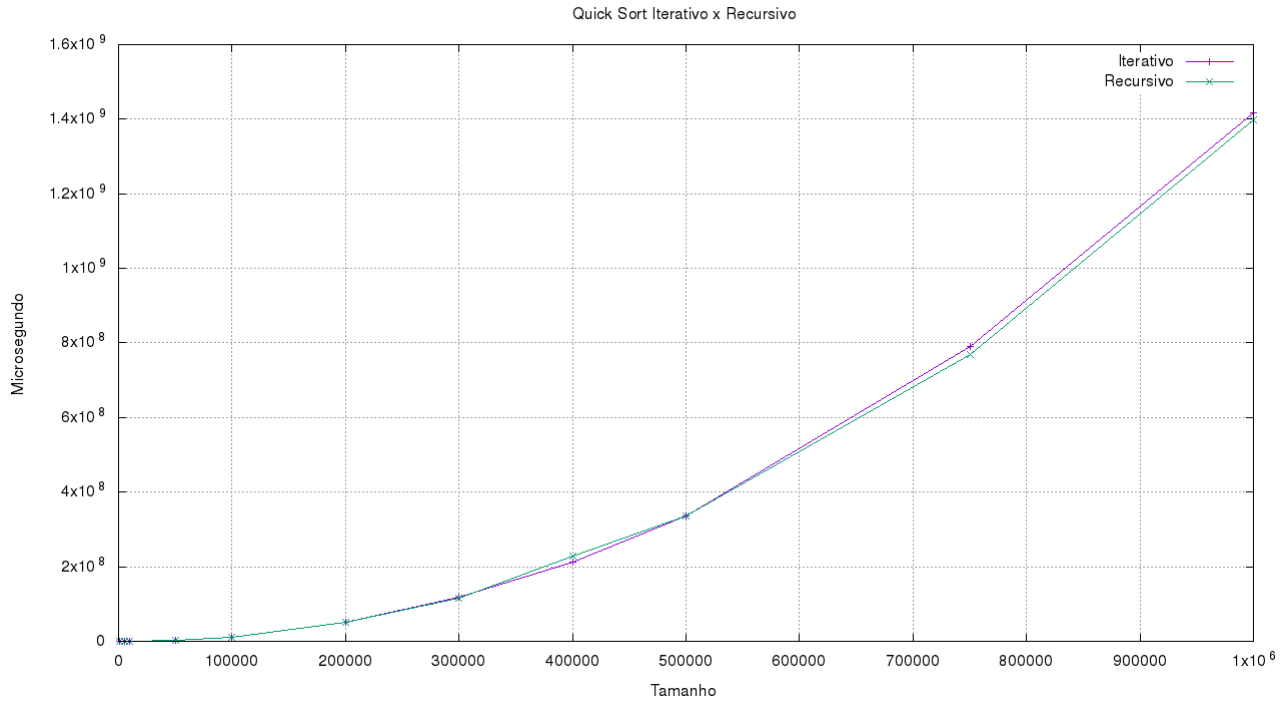
7.14 Quick Sort Iterativo



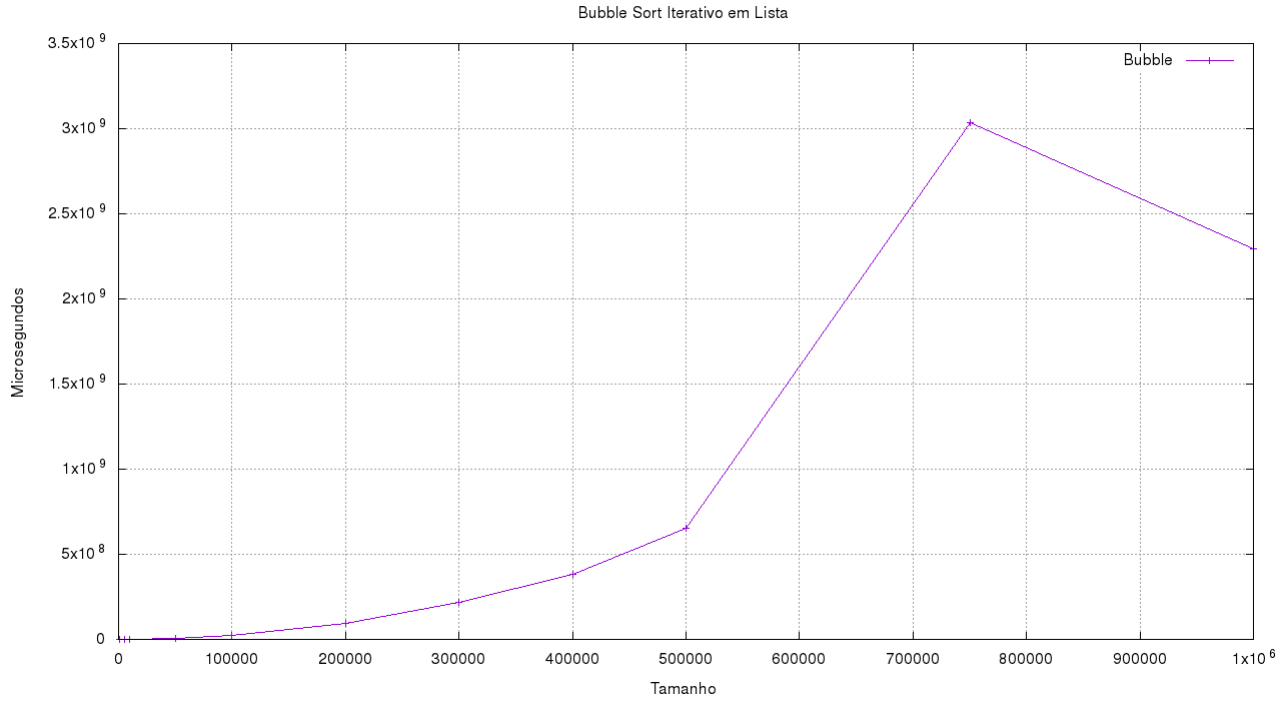
7.15 Quick Sort Recursivo



7.16 Comparação entre os Quick Sort Iterativo e Recursivo



7.17 Bubble Sort Iterativo em Lista



7.18 Comparação entre os Bubble Sort em Vetor e em Lista