Universidade Federal do ABC
Programa de Pós-graduação em Ciência da Computação

Análise dos Algoritmos de Ordenação

BubbleSort, InsertionSort, SelectionSort, MergeSort, QuickSort e HeapSort

Davidson de Faria
RA 21201931103
davidson.faria@ufabc.edu.br
Professores: Denise Goya e Luiz Rozante

Santo André, 28 de Novembro de 2019

1 Introdução

A ordenação de elementos consiste em organizá-los em ordem crescente (ou decrescente) de forma a facilitar a recuperação, busca ou visualização dos dados. O assunto já foi amplamente discutido em computação, fazendo com que se atingisse as cotas inferiores de complexidade da maioria dos algoritmos de ordenação.

A maioria das linguagens de programação já possuem os métodos de ordenação implementados, mas há casos que esses métodos não atingem a eficiência desejada e, portanto, o estudo dos algoritmos se torna importante para um cientista da computação. Tal estudo ocorre através da análise dos algoritmos para determinar qual será a quantidade de recursos, de tempo e espaço, necessária para a execução de uma entrada de tamanho arbitrário.

1.1 Objetivo

Esse projeto tem como objetivo aplicar os conhecimentos adquiridos sobre os seguintes algoritmos de ordenação: BubbleSort, InsertionSort, SelectionSort, MergeSort, QuickSort e HeapSort. O projeto irá explorar o melhor, pior e o caso médio de cada algoritmo e compará-los em questão de tempo.

2 BubbleSort

O primeiro algoritmo a ser discutido é o *Bubble Sort*, que segue a ideia de 'bolhas' subindo embaixo d'água. Tal algoritmo consiste em comparar os últimos termos de uma lista com os primeiros e realizar a troca dos elementos de acordo com a ordem desejada. Por percorrer a lista duas vezes - uma para percorrer os n elementos e outra para comparar os $n-1$ elementos - o algoritmo tem complexidade de ordem quadrática, O(n^2). Segue o pseudo-código do *BubbleSort*:

[ ]:
```
Função: BubbleSort(lista):
    Para cada elemento i da lista:
        Para cada elemento j < i da lista:
            Se lista[j] > lista[j+1]:
                Troca lista[j] <-> lista[j+1]
```

Abaixo, a implementação do código em linguagem C:

[1]:
```c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include "/home/davidson/Algoritmos/Ordenacao/bibaux.h"

#define tam 100
```

```c
void bubblesort(int *v, int t){
    int i, j, aux;
    for(i=t-1; i>=1; i--){
        for(j=0; j<i; j++){
            if(v[j] > v[j+1]){
                aux = v[j];
                v[j] = v[j+1];
                v[j+1] = aux;
            }
        }
    }
}

int main(){
    int v[tam];
    int i;

    vetorAleatorio(v, tam);

    printf("Vetor Inicial de Tamanho: %d\n", tam);
    printArray(v, tam);

    clock_t t1 = clock();
    bubblesort(v, tam);
    contaTempo(t1);

    printf("Vetor Ordenado:\n");
    printArray(v, tam);

    return 0;
}
```

```
Vetor Inicial de Tamanho: 100
88 88 65 16 47 88 11 47 96 48 65 32 11 41 81 17 96 23 42 34 47 82 71 22 44␣
 ↪98 90
94 4 77 97 44 18 62 61 17 51 24 17 99 72 82 84 36 23 65 53 71 40 47 58 87␣
 ↪29 29
9 26 80 52 20 36 81 69 81 51 32 94 69 35 18 86 34 42 20 70 78 95 36 31 19␣
 ↪76 31
77 16 60 58 25 38 38 77 11 27 11 80 8 62 12 54 31 47 72

0.000050 segundos.

Vetor Ordenado:
```

```
4 8 9 11 11 11 11 12 16 16 17 17 17 18 18 19 20 20 22 23 23 24 25 26 27 29␣
↪29 31
31 31 32 32 34 34 35 36 36 36 38 38 40 41 42 42 44 44 47 47 47 47 47 48 51␣
↪51 52
53 54 58 58 60 61 62 62 65 65 65 69 69 70 71 71 72 72 76 77 77 77 78 80 80␣
↪81 81
81 82 82 84 86 87 88 88 88 90 94 94 95 96 96 97 98 99
```

3 InsertionSort

O algoritmo *Insertion Sort* percorre a lista desde o início e, para cada índice i, ele percorre a sublista de tamanho $i-1$ comparando o seus elementos com o elemento do índice. A ordenação ocorre ao reinserir o elemento do índice i na posição correta da sublista. Assim como o algoritmo *Bubble Sort*, o *Insertion Sort* também percorre a lista e sublistas, portanto, sua complexidade é de ordem quadrática, $O(n^2)$. O pseudocódigo do *Insertion Sort*:

[ ]:
```
Função: InsertionSort(lista):
    Para cada elemento i da lista:
        escolhido <- lista[i]
        j <- i-1
        Enquanto j >=0 E lista[j] > escolhido:
            lista[j+1] <- v[j]
            j <- j-2
        lista[j+1] <- escolhido
```

A implementação do algoritmo *Insertion Sort* realizada em linguagem C pode ser vista abaixo para um vetor aleatório de tamanho 100.

[2]:
```c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include "/home/davidson/Algoritmos/Ordenacao/bibaux.h"

#define tam 100

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

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

int main(){
    int v[tam];
    int i;

    vetorAleatorio(v, tam);

    printf("Vetor Inicial de Tamanho: %d\n", tam);
    printArray(v, tam);

    clock_t t1 = clock();
    insertionSort(v, tam);
    contaTempo(t1);

    printf("Vetor Ordenado:\n");
    printArray(v, tam);

    return 0;
}
```

```
Vetor Inicial de Tamanho: 100
94 27 39 96 82 13 23 43 39 73 32 42 66 21 35 97 44 80 67 47 52 30 43 28 15␣
 ↪94 90
17 33 93 59 80 20 50 28 54 63 51 97 54 76 81 96 94 3 83 43 99 16 10 98 68␣
 ↪92 41
48 7 35 38 24 21 83 35 1 55 85 29 9 0 80 58 54 8 40 3 2 95 38 45 94 54 7 93␣
 ↪23
52 86 23 59 22 14 36 43 97 71 44 5 57 25 14 9 57

0.000015 segundos.

Vetor Ordenado:
0 1 2 3 3 5 7 7 8 9 9 10 13 14 14 15 16 17 20 21 21 22 23 23 23 24 25 27 28␣
 ↪28
29 30 32 33 35 35 35 36 38 38 39 39 40 41 42 43 43 43 43 44 44 45 47 48 50␣
 ↪51 52
52 54 54 54 54 55 57 57 58 59 59 63 66 67 68 71 73 76 80 80 80 81 82 83 83␣
 ↪85 86
90 92 93 93 94 94 94 94 95 96 96 97 97 97 98 99
```

4 SelectionSort

O algoritmo *Selection Sort* percorre a lista de modo a selecionar o índice do menor elemento - ou o maior, se for decrescente - da sublista de elementos restante e trocar com o índice atual. Sua complexidade é quadrática, $O(n^2)$, por percorrer duas vezes os elementos da lista. Segue o pseudocódigo do *Selection Sort*.

[ ]:
```
Função: SelectionSort(lista):
    Para cada elemento i da lista:
        menor = i
        Para cada elemento j>=i+1 da lista:
            Se (lista[j]< lista[menor])
                menor = j
        Se i != menor:
            Troca lista[i] <-> lista[menor]
```

A implementação abaixo mostra o comportamento do algoritmo *Selection Sort* para um vetor aleatorio de tamanho 100.

[3]:
```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include "/home/davidson/Algoritmos/Ordenacao/bibaux.h"

#define tam 100

void selectionSort(int *v, int t){
    int i, j, aux;
    for(i=0; i<tam-1; i++){
        aux = i;
        for(j=i+1; j<tam; j++){
            if(v[j] < v[aux]){
                aux = j;
            }
        }
        if(i!=aux){
            v[i] = v[i] + v[aux];
            v[aux] = v[i] - v[aux];
            v[i] = v[i] - v[aux];
        }
    }
}

int main(){
    int v[tam];
```

```c
    int i;

    vetorAleatorio(v, tam);

    printf("Vetor Inicial de Tamanho: %d\n", tam);
    printArray(v, tam);

    clock_t t1 = clock();
    selectionSort(v, tam);
    contaTempo(t1);

    printf("Vetor Ordenado:\n");
    printArray(v, tam);

    return 0;
}
```

```
Vetor Inicial de Tamanho: 100
41 94 76 31 4 21 14 84 28 55 3 33 39 24 99 14 96 58 54 58 46 82 24 66 56 44␣
 ↪92
71 77 20 44 71 66 72 54 70 45 69 54 73 24 9 58 15 33 57 29 81 67 36 39 14␣
 ↪18 15
80 74 59 24 45 37 96 41 8 14 13 62 84 58 83 90 83 60 99 41 27 84 51 57 65␣
 ↪70 93
56 84 63 72 16 37 31 93 34 20 41 76 28 56 41 43 40 52 78

0.000023 segundos.

Vetor Ordenado:
3 4 8 9 13 14 14 14 14 15 15 16 18 20 20 21 24 24 24 24 27 28 28 29 31 31␣
 ↪33 33
34 36 37 37 39 39 40 41 41 41 41 41 43 44 44 45 45 46 51 52 54 54 54 55 56␣
 ↪56 56
57 57 58 58 58 58 59 60 62 63 65 66 66 67 69 70 70 71 71 72 72 73 74 76 76␣
 ↪77 78
80 81 82 83 83 84 84 84 84 90 92 93 93 94 96 96 99 99
```

5 MergeSort

O algoritmo *Merge Sort* se baseia no método de Divisão e Conquista e consiste em ordenar a lista a partir de duas oturas listas ordenadas. Para tal, o algoritmo divide a lista original até que suas sublistas se tornem pares. Então, esses pares são ordenados e combinados com outros pares, formando novas sublistas até que toda a lista seja ordenada. De forma

sucinta, o algoritmo utiliza de três passos:

```
1. Dividir: Dividir os dados em subsequências pequenas;
2. Conquistar: A ordenação dos elementos é realizada através de comparação;
3. Combinar: Intercala as sublistas ordenadas.
```

A complexidade do *Merge Sort* é $\Theta(nlogn)$ para todos os casos.

A seguir, o pseudocódigo do *Merge Sort* recursivo.

[ ]:
```
Função: MergeSort(lista, esquerda, direita):
    Se direita > esquerda:
        meio <- (esquerda+direita)/2
        MergeSort(lista, esquerda, meio)
        MergeSort(lista, meio+1, direita)
        Intercala(lista, esquerda, meio, direita)

Função Intercala(lista, esquerda, meio, direita):
    tamEsq <- meio - esquerda + 1
    tamDir <- direita - meio
    L <- lista[1...tamEsq + 1]
    R <- lista[1...TamDir + 1]
    i <- 1
    j <- 1
    Para k de esquerda até direita:
        Se L[i] <= R[j]:
            lista[k] <- L[i]
            i < i + 1
        Senão:
            lista[k] <- R[j]
            j <- j + 1
```

A implementação do *Merge Sort* em linguagem C é realizada abaixo para um vetor aleatório de tamanho 100.

[4]:
```c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include "/home/davidson/Algoritmos/Ordenacao/bibaux.h"

#define tam 100

void intercala(int v[], int l, int m, int r){
    int i, j, k;
    int tam1 = m-l+1;
    int tam2 = r-m;
```

```
    int v1[tam1], v2[tam2];

    for(i=0; i<tam1; i++)
        v1[i] = v[l+i];
    for(j=0; j<tam2; j++)
        v2[j] = v[m+1+j];

    j=0;
    i=0;
    k=l;

    while (i<tam1 && j<tam2){
        if(v1[i] <= v2[j]){
            v[k] = v1[i];
            i++;
        }
        else{
            v[k] = v2[j];
            j++;
        }
        k++;
    }

    while(i<tam1){
        v[k] = v1[i];
        i++;
        k++;
    }

    while(j<tam2){
        v[k] = v2[j];
        j++;
        k++;
    }
}

void mergeSort(int v[], int l, int r){
    if(l<r){
        int m;

        m = l+(r-l)/2;

        mergeSort(v, l, m);
```

```
        mergeSort(v, m+1, r);

        intercala(v, l, m, r);
    }

}

int main(){
    int v[tam];
    int i;

    vetorAleatorio(v, tam);

    printf("Vetor Inicial de Tamanho: %d\n", tam);
    printArray(v, tam);

    clock_t t1 = clock();
    mergeSort(v, 0, tam-1);
    contaTempo(t1);

    printf("Vetor Ordenado:\n");
    printArray(v, tam);

    return 0;
}
```

```
Vetor Inicial de Tamanho: 100
86 50 45 68 52 98 52 44 84 67 18 55 95 97 66 1 54 41 21 76 47 59 57 25 99␣
 ↪15 28
84 7 68 95 93 18 40 61 23 91 13 67 75 32 85 30 28 34 48 81 41 42 2 69 89 13␣
 ↪27
66 13 94 94 49 1 63 44 94 81 37 55 4 28 20 23 3 5 60 85 85 95 85 66 88 27␣
 ↪21 57
68 34 84 35 99 78 29 48 79 44 45 25 26 34 80 82 62 1

0.000022 segundos.

Vetor Ordenado:
1 1 1 2 3 4 5 7 13 13 13 15 18 18 20 21 21 23 23 25 25 26 27 27 28 28 28 29␣
 ↪30
32 34 34 34 35 37 40 41 41 42 44 44 44 45 45 47 48 48 49 50 52 52 54 55 55␣
 ↪57 57
59 60 61 62 63 66 66 66 67 67 68 68 68 69 75 76 78 79 80 81 81 82 84 84 84␣
 ↪85 85
```

```
85 85 86 88 89 91 93 94 94 94 95 95 95 97 98 99 99
```

6 QuickSort

Assim como o *Merge Sort*, o *Quick Sort* utiliza do método de Divisão e Conquista para realizar a ordenação. Esse algoritmo toma um elemento como pivô de forma que seja maior que os elementos da lista à sua esquerda e menor que os elementos à sua direita. A lista é, então, particionada e um novo pivô é selecionado recursivamente até que a lista esteja ordenada. Segue o pseudocódigo do *Quick Sort*:

[ ]:
```
Função: QuickSort(lista, esq, dir):
    Se esq < dir:
        pivo <- particao(lista, esq, dir)
        QuickSort(lista, esq, pivo-1)
        QuickSort(lista, pivo+1, dir)

Função Particao(lista, esq, dir):
    pivo <- lista[dir]
    esqP <- esq - 1
    Para j de esq até dir - 1:
        Se lista[j] < pivo:
            esqP <- esqP + 1
            Troca lista[i] <-> lista[j]
    Troca lista[i+1] <-> V[r]
    Retorna i+1
```

O *Quick Sort* tem complexidade O($nlogn$) para o melhor e o caso médio, e complexidade O(n^2) para o pior caso, que ocorre quando a lista já está ordenada. Sua implementação para um vetor aleatório de tamanho 100 está descrita abaixo.

[6]:
```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include "/home/davidson/Algoritmos/Ordenacao/bibaux.h"

#define tam 100

int particao(int *v, int l, int r){
    int p = v[r];
    int i = (l-1);
    int j;

    for(j=l; j<r-1; j++){
        if(v[j] < p){
```

```
            i++;
            troca(&v[i], &v[j]);
        }
    }
    troca(&v[i+1], &v[r]);

    return (i+1);
}

void quickSort(int *v, int l, int r){
    if(l<r){
        int p = particao(v, l, r);

        quickSort(v, l, p-1);
        quickSort(v, p+1, r);
    }
}

int main(){
    int v[tam];
    int i;

    vetorAleatorio(v, tam);

    printf("Vetor Inicial de Tamanho: %d\n", tam);
    printArray(v, tam);

    clock_t t1 = clock();
    quickSort(v, 0, tam-1);
    contaTempo(t1);

    printf("Vetor Ordenado:\n");
    printArray(v, tam);

    return 0;
}
```

```
Vetor Inicial de Tamanho: 100
11 2 48 31 77 56 38 72 76 93 84 47 17 13 34 16 12 2 26 27 62 25 41 15 94 75␣
 ↪29
69 96 44 46 60 46 94 43 75 3 81 47 31 74 83 79 43 49 13 60 61 15 86 40 30␣
 ↪63 82
97 57 9 26 78 5 70 24 17 68 71 60 43 74 93 43 5 20 26 36 63 27 50 75 88 17␣
 ↪13 29
```

```
47 76 63 45 85 72 71 63 29 94 88 47 62 11 59 6 37 53

0.000010 segundos.

Vetor Ordenado:
2 3 2 5 5 6 9 12 11 13 13 13 15 16 15 17 17 25 24 26 26 26 20 27 27 17 29␣
 ↪29 29
30 31 36 31 38 34 40 41 43 43 43 43 44 46 46 45 47 47 47 48 47 49 50 11 53␣
 ↪57 60
60 60 56 62 62 61 63 63 63 68 69 70 63 71 72 72 74 74 71 75 75 75 76 76 77␣
 ↪78 79
81 59 82 83 84 85 86 88 88 93 94 94 94 93 96 97 37
```

7 HeapSort

O *Heap Sort* é um algoritmo de ordenação baseado em comparação utilizando a estrutura *Heap Binário*. Essa estrutura é definida por uma árvore binária completa - ou quase completa se o faltar nós no último nível - e onde todos os nós estão à mais esquerda possível. Um *Heap* também pode ser representado por uma pilha de forma que cada nó pai p tenha os nós filhos posicionados nos índices $2p + 1$ à esquerda e $2p + 2$ à direita. O *Heap* também deve garantir que os nós pais sempre sejam maior que os nós filhos.

O algoritmo consiste na construção de um *Heap* a partir de um lista, então é realizada a troca da raíz - maior elemento - com o elemento mais a direita do nível mais baixo - menor elemento - e, se o *Heap* perder sua estrutrua, deve ser construído novamente. Esse processo deve ser repetivo até que todos os elementos sejam retirados do *Heap*. O pseudocódigo do *Heap Sort* é descrito abaixo.

[ ]:
```
Função: HeapSort(lista, tam):
    Para i de tam/2 - 1 até 0:
        ConstoiHeap(lista, tam, i)
    Para i de tam-1 até 0:
        Troca lista[0] <-> lista[i]
        ConstoiHeap(lista, i, 0)

Função: ConstroiHeap(lista, tam, i):
    maior <- i
    esq <- 2*i + 1
    dir <- 2*i + 2
    Se (esq < tam E lista[esq] > v[maior]):
        maior = esq

    Se (dir < tam E lista[dir] > v[maior]):
        maior = dir
```

```
    Se maior != i:
        Troca lista[i] <-> lista[maior]
        ConstroiHeap(lista, tam, maior)
```

A complexidade do algortimo para a construção do *Heap* é logarítmica, O($logn$), entretante como o *Heap Sort* necessita percorrer todos os elementos, então a complexidade leva uma termo n relativo ao tamanho da lista, portanto, O($nlogn$). A implementação do *Heap Sort* para um vetor aleatório de tamanho 100 é descrita abaixo.

[9]:
```c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include "/home/davidson/Algoritmos/Ordenacao/bibaux.h"

#define tam 100

void heapify(int *v, int n, int i){
    int m = i;
    int l = 2*i+1;
    int r = 2*i+2;

    if (l < n && v[l] > v[m])
        m = l;

    // If right child is larger than largest so far
    if (r < n && v[r] > v[m])
        m = r;

    if (m != i) {
        troca(&v[i], &v[m]);

        heapify(v, n, m);
    }
}

void heapSort(int *v, int n){
    int i;
    for(i=n/2 -1; i >= 0; i--)
        heapify(v, n, i);

    for (i=n-1; i>=0; i--)
    {
        troca(&v[0], &v[i]);
```

```
        heapify(v, i, 0);
    }
}

int main(){
    int v[tam];
    int i;

    vetorAleatorio(v, tam);

    printf("Vetor Inicial de Tamanho: %d\n", tam);
    printArray(v, tam);

    clock_t t1 = clock();
    heapSort(v, tam);
    contaTempo(t1);

    printf("Vetor Ordenado:\n");
    printArray(v, tam);

    return 0;
}
```

```
Vetor Inicial de Tamanho: 100
16 85 86 87 35 41 60 92 42 25 88 50 51 22 20 48 98 10 76 52 39 90 55 23 59␣
 ↪40 45
93 14 15 20 82 52 6 69 87 99 29 79 94 6 19 96 10 42 68 58 92 79 86 44 70 77␣
 ↪52
93 88 44 38 33 58 6 54 40 58 12 61 46 64 42 77 58 48 97 54 58 91 22 68 83 1␣
 ↪7 80
72 36 32 17 24 76 56 10 86 14 16 78 72 80 39 70 44 33

0.000016 segundos.

Vetor Ordenado:
1 6 6 6 7 10 10 10 12 14 14 15 16 16 17 19 20 20 22 22 23 24 25 29 32 33 33␣
 ↪35
36 38 39 39 40 40 41 42 42 42 44 44 44 45 46 48 48 50 51 52 52 52 54 54 55␣
 ↪56 58
58 58 58 58 59 60 61 64 68 68 69 70 70 72 72 76 76 77 77 78 79 79 80 80 82␣
 ↪83 85
86 86 86 87 87 88 88 90 91 92 92 93 93 94 96 97 98 99
```

8 Análise dos Algoritmos

Para analisar o comportamento dos algoritmos com entradas de vetores aleatórios de tamanhos diferentes, foram realizados testes com o seguinte programa:

[ ]:
```c
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include "/home/davidson/Algoritmos/Ordenacao/bibaux.h"
#include "/home/davidson/Algoritmos/Ordenacao/algoritmos.h"

#define max 7

int main(){
    int i, j;
    int *v;
    int vetTam[max] = {100, 1000, 5000, 10000, 30000, 50000, 100000};
    double res[max][6];
    clock_t t1;

    FILE *arq;
    arq = fopen("analise.txt", "w");

    for(i=0; i<max; i++){
        v = (int *) malloc(vetTam[i] * sizeof(int));

        vetorAleatorio(v, vetTam[i]);
        t1 = clock();
        bubblesort(v, vetTam[i]);
        res[i][0] = contaTempoR(t1);

        vetorAleatorio(v, vetTam[i]);
        t1 = clock();
        insertionSort(v, vetTam[i]);
        res[i][1] = contaTempoR(t1);

        vetorAleatorio(v, vetTam[i]);
        t1 = clock();
        selectionSort(v, vetTam[i]);
        res[i][2] = contaTempoR(t1);

        vetorAleatorio(v, vetTam[i]);
        t1 = clock();
        mergeSort(v, 0, vetTam[i]-1);
```

```
        res[i][3] = contaTempoR(t1);

        vetorAleatorio(v, vetTam[i]);
        t1 = clock();
        quickSort(v, 0, vetTam[i]-1);
        res[i][4] = contaTempoR(t1);

        vetorAleatorio(v, vetTam[i]);
        t1 = clock();
        heapSort(v, vetTam[i]);
        res[i][5] = contaTempoR(t1);

        free(v);
    }

    printf("\n");
    for(i=0; i<max; i++){
        fprintf(arq, "%d\t", vetTam[i]);
        printf("%d\t", vetTam[i]);
        for(j=0; j<6; j++){
            fprintf(arq, "%f\t", res[i][j]);

            printf("%f\t", res[i][j]);
        }
        fprintf(arq, "\n");
        printf("\n");
    }
    fclose(arq);
}
```

A tabela a seguir mostra os tempos, em milisegundos, necessários para cada algoritmo executar as tarefas desejadas.

Tamanho	***BubbleSort***	***InsertionSort***	***SelectionSort***	***MergeSort***	***QuickSort***	***HeapSort***
100	0.036000	0.009000	0.018000	0.014000	0.008000	0.014000
1000	2.439000	1.141000	2.028000	0.206000	0.138000	0.255000
5000	78.750000	17.883000	34.747000	0.734000	0.728000	1.092000
10000	325.241000	84.315000	126.196000	1.516000	2.094000	2.262000
30000	3061.186000	640.764000	1141.172000	4.859000	14.872000	8.541000
50000	8788.488000	1830.865000	3452.357000	8.162000	37.136000	14.264000
100000	35727.199000	7990.090000	13893.826000	17.476000	149.848000	40.105000

Pela a análise da tabela, é possível perceber que os algoritmos quadráticos levam muito mais tempo para realizar a mesma tarefa. A figura 1 apresenta o comportamento das curvas e o ajuste quadrático para seus pontos.

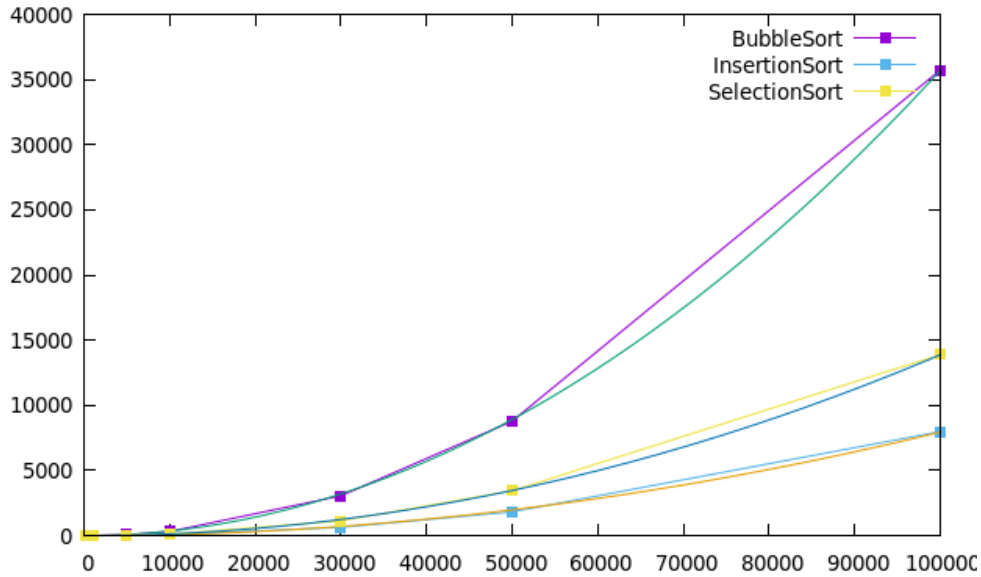

Figure 1: Tempo de execução dos algoritmos de complexidade quadrática.

Em contrapartida, os algoritmos de complexidade linearitmica não levarem nem 1 segundo para executarem as mesmas tarefas. A figura 2 apresenta o comportamento das curvas do tempo de execução de cada tarefa.

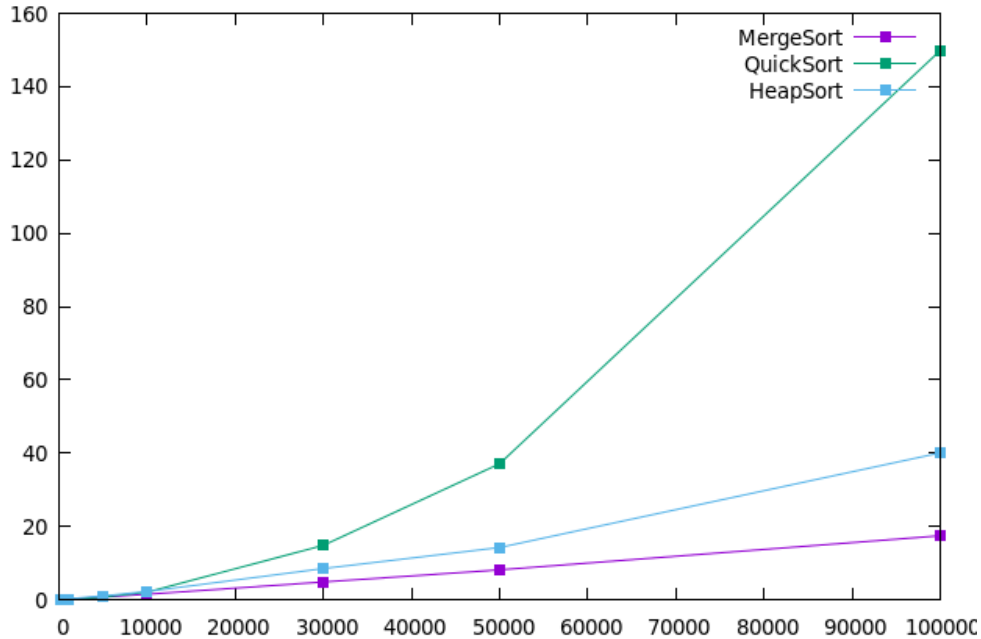

Figure 2: Tempo de execução dos algoritmos de complexidade linearitmica.

9 Conclusão

Pela a análise dos algoritmos estudados, é possível perceber uma clara diferença na eficiência dos algoritmos. Dessa forma, um cientista da computação é, então, capaz de compreender quais as melhoras maneiras de realizar uma tarefa de forma evitar cálculos desnecessários que aumentem o tempo de execução do algoritmo.

Como sugestões para complementar o trabalho, seria interessante analisar os todos os casos de cada algoritmo, aumentar o tamanho das listas e utilizar computadores diferentes.

Referências

1. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009.

Anexo

Este trabalho se encontra no repositório:
https://github.com/davidsondefaria/Ordenacao.git