Comparação de Algoritmos de Ordenação

Mateus Cardoso & Marco Bockoski
7° Período de Engenharia de Computação (UNITAU)

SELEÇÃO E IMPLEMENTAÇÃO DE ALGORITMOS

Os algoritmos de ordenação escolhidos para testes são: BubbleSort, SelectionSort, InsertionSort, CombSort, Shell Sort, QuickSort e Quick Sort Plus, pela justificativa da ocorrência de suas exposições em sala de aula, desde raciocínio inicial até o pseudocódigo completo.

A implementação do projeto de comparação de algoritmos de ordenação tem certas peculiaridades:

- Implementação em Java, devido à familiaridade dos integrantes do grupo com essa determinada linguagem;
- Utilização da IDE Java NetBeans para execução das linhas de código;
- A partir de funções da classe System própria da IDE NetBeans, foi desenvolvido uma classe chamada StopWatch, que é capaz de contar milissegundos ou segundos em que algo ocorre;
- Todo algoritmo de ordenação corresponde a duas funções, uma de ordenar de fato o vetor e outro para contabilizar número de trocas e comparações;
- Um vetor é gerado como modelo e suas cópias passam por processos de ordenamento tal como é contado o número de trocas e comparações que acontecem durante a ordenação;
- Existe uma função troca que altera os valores do vetor, sem a necessidade de declarar uma variável auxiliar na função de ordenação, não utilizada em determinados casos (InsertionSort);

BUBBLE SORT

O algoritmo BubbleSort (Ordenação em Bolha) se trata de estabelecer comparações entre dois elementos de um vetor, o maior é deslocado para direita (fim do vetor) por meio de trocas, se o elemento à esquerda é maior que o da direita, eles trocam e os novos elementos a serem comparados é o da direita e o da direita ao da direita.

Um exemplo para ilustrar, considere os elementos v[4] (esquerda) e v[5] (direita) com valores v[4] = 8 e v[5] = 4. O valor da v[4] é maior que o valor de [5], portanto seus valores vão trocar assim, v[4] = 4 e v[5] = 8. E a próxima troca será entre v[5] (direita) e v[6] (direita ao da direita).O efeito final será que os valores maiores estarão à direita e os menores à esquerda de modo ordenados de forma crescente.

Sua implementação em Java corresponde à seguinte linha de código:

Figura 1-Ordenação em Bolha (Sem Contadores)

```
public static void bubbleSort(int[] vetor){
    for(int j = 1; j < vetor.length; j++){
        for(int i = 0; i < vetor.length-j; i++){
            if(vetor[i] > vetor[i+1]){
                int aux = vetor[i];
                vetor[i] = vetor[i+1];
                vetor[i+1] = aux;
            }
        }
    }
}
```

Ela se trata dos dois loops, um (loop j) que escolhe a ultima posição possível que receberá o maior valor; e o segundo loop (loop i) responsável por fazer as devidas trocas entre os elementos dos vetores. O condicional “if” checa se o valor de fato é menor e no seu corpo contém a função de troca.

Existe uma função alternativa responsável por contabilizar o número de trocas e comparações:

Figura 2 - Ordenação em Bolha (Com Contadores)

```
public static void bubbleSortC(int[] vetor){
    for(int j = 1; j < vetor.length; j++){
        for(int i = 0; i < vetor.length-j; i++){
            if(vetor[i] > vetor[i+1]){
                int aux = vetor[i];
                vetor[i] = vetor[i+1];
                vetor[i+1] = aux;
                trocas++;
            }
            comparacoes++;
        }
    }
}
```

É responsável por realizar o mesmo processo, porém contém as variáveis trocas e comparações que são variáveis para instrumentação, servem para contabilizar quantidade de trocas e comparações.

Sua função de complexidade é determinada em partes:

- Considerando n o valor do tamanho do vetor e desconsiderando o número de trocas pelo seu caráter probabilístico.
- Loop j: $n - 1$
- Loop i: $[(n - 1) * n] / 2$
- Condicional if: $[(n - 1) * n] / 2$
- Trocas: Probabilística
- Soma das expressões, gera-se o Big-Oh:

$$(n - 1) + [(n - 1) * n] / 2 + [(n - 1) * n] / 2 = (n - 1) + (n - 1) * n$$
$$O(n^2) = n^2 - 1$$

SELECTION SORT

O algoritmo SelectionSort (Ordenação por Seleção) também se baseia em comparações para promover a troca entre elementos dos vetores, porém tem o intuito de reduzir a quantidade de trocas de elementos, assim, ele deixa fixo a troca no maior endereço de memória possível (v[vetor.length-j]) e identifica o maior valor possível do vetor, o endereço que detém esse valor é guardado por uma variável, responsável eventualmente para ser o segundo endereço da troca. Assim, como o BubbleSort, um loop é feito para buscar o maior valor e o segundo para realizar a devida troca.

Sua implementação em Java corresponde ao código da Figura 3:

Figura 3 - Ordenação por Seleção (Sem Contadores)

```
public static void selectSort(int[] vetor){
    for(int j = 1; j < vetor.length; j++){
        int maior = vetor.length-j;
        for(int i = 0; i < vetor.length-j; i++){
            if(vetor[i] > vetor[maior]) maior = i;
        }
        int aux = vetor[maior];
        vetor[maior] = vetor[vetor.length-j];
        vetor[vetor.length-j] = aux;
    }
}
```

O primeiro loop determina o índex do endereço de memória que receberá o maior valor (int maior) e, depois da execução do segundo loop que acha o índex que tem o maior valor; realiza a devida troca deixando o valor de v[vetor.length-j] = maior, o processo continua até deixar o vetor em ordem crescente.

Foi implementada uma função alternativa responsável por contabilizar o número de trocas e comparações:

Figura 4 - Ordenação por Seleção (Com Contadores)

```
public static void selectSortC(int[] vetor){
    for(int j = 1; j < vetor.length; j++){
        int maior = vetor.length-j;
        for(int i = 0; i < vetor.length-j; i++){
            if(vetor[i] > vetor[maior]) maior = i;
            comparacoes++;
        }
        int aux = vetor[maior];
        vetor[maior] = vetor[vetor.length-j];
        vetor[vetor.length-j] = aux;
        trocas++;
    }
}
```

As variáveis trocas e comparações são variáveis de instrumentação responsáveis por quantificar respectivos valores.

Sua função de complexidade é realizada pela análise de certos comandos:

- Considerando n o valor do tamanho do vetor
- Loop j: $n - 1$
- Loop i: $[(n - 1) * n] / 2$
- Condicional if: $[(n - 1) * n] / 2$
- Trocas: $n - 1$
- Soma das expressões, tem-se o Big-Oh:

$$(n - 1) + (n - 1) + \frac{[(n - 1) * n]}{2} + \frac{[(n - 1) * n]}{2} = 2 * (n - 1) + (n - 1) * n$$
$$= (2 + n) * (n - 1)$$

$$O(n^2) = n^2 + n - 2$$

INSERTION SORT

O algoritmo de ordenação por inserção (InsertionSort) considera que um vetor é separado em duas partes, parte ordenada e a parte não-ordenada. Um loop mais externo é responsável por selecionar qual valor (presente na parte não-ordenada) vai ser inserido na parte ordenada do vetor.

Enquanto existe um loop mais interno que é responsável pela devida inserção do valor na parte ordenada, cada iteração desse loop mais interno desloca os elementos que são maiores que o valor de inserção para o próximo endereço de memória, apesar disso não se perde o valor de inserção, pois ele é guardado em uma variável auxiliar que o colocará em uma posição apropriada no fim do loop interno.

A figura abaixo apresenta a implementação em Java do algoritmo:

Figura 5 - Ordenação por Inserção (Sem Contadores)

```
public static void insertionSort(int[] vetor){
    for (int i = 1; i < vetor.length; i++){
        int aux = vetor[i];
        int j = i;

        while ((j > 0) && (vetor[j-1] > aux)){
            vetor[j] = vetor[j-1];
            j -= 1;
        }
        vetor[j] = aux;
    }
}
```

O primeiro loop (for, loop exterior) é responsável por selecionar o elemento de inserção vetor[i]. A variável começa com o valor “1” pois já pressupõe que o elemento de índex 0 já é a parte ordenada, enquanto todo o resto é não-ordenado. A variável “j” é responsável por inserir apropriadamente o valor de inserção, já realizando o deslocamento dos maiores valores se necessário, por meio do segundo loop (while, loop interior).

O deslocamento dos maiores valores é feito no próprio loop interior, é testado se o valor de inserção é menor que um dos elementos da parte

ordenada, se for o caso, desloca esse elemento da parte ordenada para um endereço de índex sucessor, realizando até o valor de inserção chegar na posição apropriada, que terá seu valor trocado pelo último comando de atribuição no final do corpo do loop exterior.

Implementação alternativa com contadores:

Figura 6 - Ordenação por Inserção (Sem Contadores)

```
public static void insertionSortC(int[] vetor){
    for (int i = 1; i < vetor.length; i++){
        int aux = vetor[i];
        int j = i;


        while ((j > 0) && (vetor[j-1] > aux)){
            comparacoes++;
            vetor[j] = vetor[j-1];
            j -= 1;
        }
        comparacoes++;
        vetor[j] = aux;
        trocas++;
    }
}
```

As variáveis trocas e comparações são variáveis de instrumentação responsáveis por quantificar respectivos valores.

Sua função de complexidade é realizada pela análise de certos comandos:

- Considerando n o valor do tamanho do vetor
- Loop i (for): $n - 1$
- Loop j (while): $\begin{cases} n-1 & se\ ordenado\ (melhor\ caso) \\ \frac{n^2+n-2}{2} & se\ inversamente\ ordenado\ (pior\ caso) \end{cases}$
- Trocas: $n - 1$
- Soma das expressões, função de complexidade (Melhor caso):

$$(n-1)+(n-1)+(n-1)=3*(n-1)$$
$$O(n)=3*(n-1)$$

- Soma das expressões, função de complexidade (Pior caso):

$$(n-1)+\left(\frac{n^2+n-2}{2}\right)+(n-1)=\frac{n^2+n-2}{2}+\frac{4n}{2}-\frac{4}{2}$$
$$O(n^2)=\frac{n^2+5n-6}{2}$$

- Assim, todo caso intermediário corresponde à:

$$O(n^2)=\frac{n^2+5n-6}{2}\geq O\big(f(n)\big)\geq O(n)=3*(n-1)$$

COMB SORT

O CombSort corresponde a uma modificação do algoritmo de ordenação BubbleSort, sendo o BubbleSort Loop aninhados que escolhe a posição a ser ordenada por meio de trocas de vizinhos. No BubbleSort, separam-se dois tipos de valores, os de valores pequenos próximos ao final do vetor, são chamados de “turtles” pelo seu vagaroso deslocamento ao começo do vetor; enquanto os valores grandes aos arredores do começo do vetor são chamados de “rabbits”, pois rapidamente vão ao final do vetor, o CombSort é responsável por eliminar esse vagaroso deslocamento de valores pequenos ao começo do vetor.

Diferentemente do BubbleSort, o CombSort troca valores com certa diferença armazenada em uma variável denominada gap, se o valor mais a esquerda for maior que o mais a direita (seguindo a convenção que um vetor tem seu início no extremo esquerdo e fim no extremo direito), ocorre troca.

Essa variável gap que proporciona essas trocas garante a inexistência de “turtles” causando um efeito “rabbit” em valores de diferentes proporções, e ela e mensurada conforme o valor do vetor:

$$gap = \frac{vetor.length}{k}$$

E no decorrer das trocas e das varreduras pelo vetor, ocorre decaimento do valor da variável pela mesma proporção k:

$$gap = \frac{gap}{k}$$

Segundo estudos, o melhor valor para k é1.3, assim como os casos do gap assumir valores como 9 ou 10 serem substituídos pelo seu valor sendo 11. As trocas acontecem até o valor de gap corresponder a 1, assim, o vetor finaliza-se ordenado.

A primeira parte do código são as declarações de variáveis: constante, verificar ordenação e o valor de gap. Seguido do loop while não ordenado, ele testa condições do gap ser menor que 1 e casos de gap sendo 9 ou 10, se for o caso do gap ser menor que 1, considera-se ordenado. Caso contrário a ordenação se dá semelhantemente ao BubbleSort, como ocorre na implementação presente na figura 7:

Figura 7 - Algoritmo do Pente (Sem Contadores)

```
public static void combSort(int[] vetor){
    int gap = vetor.length;
    float cons =  (float)1.3;
    boolean ordenado = false;

    while(!ordenado){
        gap = (int)(((float) gap) / cons);

        if(gap<=1){
            gap = 1;
            ordenado = true;
        }else if (gap == 9 || gap == 10){
            gap = 11;
        }

        for(int i = 0; (i+gap) < vetor.length; i++){
            if(vetor[i]>vetor[i+gap]){
                int aux = vetor[i];
                vetor[i] = vetor[i+gap];
                vetor[i+gap] = aux;
                ordenado = false;
            }
        }
    }
}
```

A presença de contadores em sua implementação estão presentes dentro do loop for (para comparações) e dentro do condicional if (para trocas), ilustradas pela figura 8:

Figura 8 -Algoritmo do Pente (Com Contadores)

```
public static void combSortC(int[] vetor){
    int gap = vetor.length;
    float cons =  (float)1.3;
    boolean ordenado = false;

    while(!ordenado){
        gap = (int)(((float) gap) / cons);

        if(gap<=1){
            gap = 1;
            ordenado = true;
        }else if (gap == 9 || gap == 10){
            gap = 11;
        }

        for(int i = 0; (i+gap) < vetor.length; i++){
            comparacoes++;
            if(vetor[i]>vetor[i+gap]){
                int aux = vetor[i];
                vetor[i] = vetor[i+gap];
                vetor[i+gap] = aux;
                trocas++;
                ordenado = false;
            }
        }
    }
}
```

A função de complexidade tem sua demonstração trabalhosa, para fins de simplificação, será exposta apenas o valor final do Big-Oh para o melhor caso e o pior caso:

- Big-Oh do melhor caso (ordenado): $O(n \,.\log n)$
- Big-Oh do pior caso (inversamente ordenado): $O(n^2)$

SHELL SORT

Shell Sort é considerada um aprimoramento do InsertionSort, tendo sua modificação em transformar um vetor em subgrupos. Escolhe-se arbitrariamente um valorpara separar o vetor em subgrupos, de modo que todos os elementos de um subgrupo estão separados por um intervalo h entre si. Cada subgrupo é ordenado por Inserção, no final da ordenação dos subgrupos, reduz-se o valor de h e refaz a ordenação por inserção novamente. A ideia desse método de ordenação é semelhante ao CombSort, realizar a ordenação em intervalos maiores para garantir que valores grandes se mantenham próximos ao final ou se desloquem rapidamente para o final, o mesmo para o começo do vetor e os valores menores.

A implementação corresponde a uma parte para determinar o valor inicial de h e a outra parte realizar a ordenação por inserção por meio de loops aninhados, a utilização de h é o que garante a ordenação apenas aos subgrupos, finalizando quando $h = 1$.

Figura 9 - Ordenação de Shell (Sem Contadores)

```
public static void shellSort(int[] vetor){
    int h = 1;
    int n = vetor.length;

    while(h < n){
        h = h * 3 + 1;
    }

    h = h / 3;

    int aux, j;

    while(h > 0){
        for(int i = h; i < n; i++){
            aux = vetor[i];
            j = i;
            while(j >= h && vetor[j-h]> aux){
                vetor[j] = vetor[j-h];
                j = j - h;
            }
            vetor[j] = aux;
        }
        System.out.println("");
        h = h/2;
    }
}
```

A implementação com contadores é semelhante ao que acontece com o InsertionSort, dentro dos loops aninhados:

Figura 10 - Ordenação de Shell (Com Contadores)

```
public static void shellSortC(int[] vetor){
    int h = 1;
    int n = vetor.length;

    while(h < n){
        h = h * 3 + 1;
    }

    h = h / 3;
    int aux, j;

    while(h > 0){
        for(int i = h; i < n; i++){
            aux = vetor[i];
            j = i;
            while(j >= h && vetor[j-h]> aux){
                comparacoes++;
                vetor[j] = vetor[j-h];
                j = j - h;
                trocas++;
            }
            comparacoes++;
            vetor[j] = aux;
            trocas++;
        }
        h = h/2;
    }
}
```

A implementação da Shell Sort é variada conforme tamanho do vetor (na determinação dos intervalos) e por muitas trocas acontecerem de forma probabilística, as expressões das funções de complexidade são expostas abaixo:

- Big-Oh para o melhor caso (ordenado): $O(n \,.\log n)$
- Big-Oh para o pior caso(inversamente ordenado): $O(n^2)$

QUICK SORT

O método de ordenação por QuickSort consiste na chamada de método de partição e na recursividade do próprio QuickSort. O método de partição corresponde em adotar um valor como pivô (geralmente o primeiro) e determinar sua posição final no vetor, e todo valor menor ou igual ao pivô é colocado à sua esquerda e todo valor maior que o pivô é colocado à direita dele. O QuickSort faz o método de partição pela primeira vez, depois o resultado é um valor ordenado e dois vetores (de valores menores e iguais e outro de valores maiores) desordenados, assim, aplica-se o método QuickSort para cada um dos dois vetores recursivamente, o efeito causado é da ordenação do vetor inteiro pela partição.

A implementação do método de partição adota o pivô como o primeiro valor do vetor, estabelece valores de i e f para receber valores de início e fim, o valor de i só incrementa e o valor de f apenas decrementa, assim, um loop é responsável por deslocar os valores só encerrará quando i for maior que f. Os valores menores que o pivot se mantém nos lugares, caso se ache um valor maior em i, ele é trocado de posição com um valor menor que o pivô achado por f. A execução finaliza com o pivot ocupando sua posição final, valores menores e iguais a esquerda e maiores a direita e retorna o índice de posição do pivô.

Figura 11 - Método de Partição (Sem Contadores)

```
private static int separar(int[] vetor, int inicio, int fim) {
    int pivo = vetor[inicio];
    int i = inicio + 1, f = fim;
    while (i <= f) {
        if (vetor[i] <= pivo)  i++;
        else if (pivo < vetor[f])  f--;
        else {
            int troca = vetor[i];
            vetor[i] = vetor[f];
            vetor[f] = troca;
            i++;
            f--;
        }
    }
    vetor[inicio] = vetor[f];
    vetor[f] = pivo;
    return f;
}
```

A implementação do método QuickSort evoca o método de partição, por meio do índice do pivô, evoca-se dois métodos do QuickSort, uma com o vetor de menores que o pivô e outra com o vetor de maiores que o pivô. O processo ocorre recursivamente até o vetor ficar ordenado.

Figura 12 - Ordenação Rápida (Sem Contadores)

```
private static void quickSort(int[] vetor, int inicio, int fim) {
    if (inicio < fim) {
        int posicaoPivo = separar(vetor, inicio, fim);
        quickSort(vetor, inicio, posicaoPivo - 1);
        quickSort(vetor, posicaoPivo + 1, fim);
    }
}
```

As mesmas funções foram implementadas com variáveis de instrumentação, presentes nas figuras 13 e 14.

Figura 13 - Método de Partição (Com Contadores)

```
private static int separarC(int[] vetor, int inicio, int fim) {
    int pivo = vetor[inicio];
    int i = inicio + 1, f = fim;
    while (i <= f) {
        if (vetor[i] <= pivo){
            comparacoes++;
            i++;
        }
        else if (pivo < vetor[f]){
            comparacoes++;
            f--;
        }
        else {
           int troca = vetor[i];
           vetor[i] = vetor[f];
           vetor[f] = troca;
           trocas++;
           i++;
           f--;
        }
    }
    vetor[inicio] = vetor[f];
    vetor[f] = pivo;
    trocas++;
    return f;
}
```

Figura 14– Ordenação Rápida (Com contadores)

```
private static void quickSortC(int[] vetor, int inicio, int fim) {
    if (inicio < fim) {
        int posicaoPivo = separarC(vetor, inicio, fim);
        quickSortC(vetor, inicio, posicaoPivo - 1);
        quickSortC(vetor, posicaoPivo + 1, fim);
    }
}
```

As funções de complexidades desse algoritmo correspondemàs situações abaixo:

- Big-Oh para o melhor caso (ordenado): $O(n \,.\log n)$
- Big-Oh para o pior caso (inversamente ordenado): $O(n^2)$

QUICK SORT PLUS

Essa variação do Quick Sort é chama de Mediana de Três, que consiste em comparar qual é a mediana entre o primeiro valor do vetor, valor do meio do vetor e o último valor, dentre os três, qual tiver o valor intermediário, terá seu valor trocado pelo valor do meio e será considerado pivô do método de partição. A implementação está presente na Figura 15, dispensando explicações aprofundadas por proceder regularmente como um Quick Sort convencional. As Figuras 16, 17, 18 e 19 correspondem ao método de partição, Quick Sort Plus de Instrumentação e Método de Partição Plus de Instrumentação. O Big-Oh desse método é $O(n.\log n)$

Figura 15 - Ordenação Rápida (Mediana de Três)

```
private static void quickSortPlus(int[] vetor, int inicio, int fim) {
    if (inicio < fim) {
        int posicaoPivo = separarPlus(vetor, inicio, fim);
        quickSort(vetor, inicio, posicaoPivo - 1);
        quickSort(vetor, posicaoPivo + 1, fim);
    }
}
```

Figura 16 - Método de Partição (Mediana de Três)

```
private static int separarPlus(int[] vetor, int inicio, int fim) {
    int mid = inicio+fim/2;

    if (vetor[inicio] > vetor[mid]) {
        int aux = vetor[inicio];
        vetor[inicio] = vetor[mid];
        vetor[mid] = aux;
    }

    if (vetor[inicio] > vetor[fim]) {
        int aux = vetor[inicio];
        vetor[inicio] = vetor[fim];
        vetor[fim] = aux;
    }

    if (vetor[mid] > vetor[fim]) {
        int aux = vetor[mid];
        vetor[mid] = vetor[fim];
        vetor[fim] = aux;
    }

    int pivoIndex = mid;
    int pivo = vetor[mid];

    int aux1 = vetor[pivoIndex];
    vetor[pivoIndex] = vetor[fim];
    vetor[fim] = aux1;
    int res = inicio;

    for(int i = inicio; i < fim; i++){
        if(vetor[i] < pivo){
            int aux2 = vetor[i];
            vetor[i] = vetor[res];
            vetor[res] = aux2;
            res++;
        }
    }

    int aux2 = vetor[fim];
    vetor[fim] = vetor[res];
    vetor[res] = aux2;

    return res;
}
```

Figura 17 - Instrumentação na Ordenação Rápida

```
private static void quickSortPlusC(int[] vetor, int inicio, int fim) {
    if (inicio < fim) {
        int posicaoPivo = separarPlusC(vetor, inicio, fim);
        quickSort(vetor, inicio, posicaoPivo - 1);
        quickSort(vetor, posicaoPivo + 1, fim);
    }
}
```

Figura 18 - Instrumentação no Método de Partição (1/2)

```
private static int separarPlusC(int[] vetor, int inicio, int fim) {
    int mid = inicio+fim/2;

    if (vetor[inicio] > vetor[mid]) {
        int aux = vetor[inicio];
        vetor[inicio] = vetor[mid];
        vetor[mid] = aux;
        trocas++;
    }
    comparacoes++;

    if (vetor[inicio] > vetor[fim]) {
        int aux = vetor[inicio];
        vetor[inicio] = vetor[fim];
        vetor[fim] = aux;
        trocas++;
    }
    comparacoes++;

    if (vetor[mid] > vetor[fim]) {
        int aux = vetor[mid];
        vetor[mid] = vetor[fim];
        vetor[fim] = aux;
        trocas++;
    }
    comparacoes++;

    int pivoIndex = mid;
    int pivo = vetor[mid];
```

Figura 19 - Instrumentação no Método de Partição (2/2)

```
    int aux1 = vetor[pivoIndex];
    vetor[pivoIndex] = vetor[fim];
    vetor[fim] = aux1;
    trocas++;
    int res = inicio;

    for(int i = inicio; i < fim; i++){
        comparacoes++;
        if(vetor[i] < pivo){
            int aux2 = vetor[i];
            vetor[i] = vetor[res];
            vetor[res] = aux2;
            trocas++;
            res++;
        }
    }

    int aux2 = vetor[fim];
    vetor[fim] = vetor[res];
    vetor[res] = aux2;
    trocas++;

    return res;
}
```

FERRAMENTA DE MEDIÇÃO: CLASSE STOPWATCH

A implementação dessa classe corresponde ao uso de funções da System.java nativa da própria IDE. A classe contém duas variáveis, uma como valor do tempo inicial e a outra como valor do tempo final, por convenção, o tempo inicial é definido, também, na instanciação do objeto.

Figura 20 - Atributos e Construtor da Classe StopWatch

```
public class StopWatch {
    private long startTime = 0;
    private long stopTime = 0;

    public StopWatch()
    {
        startTime = System.currentTimeMillis();
    }
```

A função System.currentTimeMillis() fornece o valor de tempo do próprio computador. Métodos como start e stop dessa classe são responsáveis por exibir no console os valores de tempo.

Figura 21 - Métodos Start e Stop

```
public void start() {
    startTime = System.currentTimeMillis();
}

public void stop() {
    stopTime = System.currentTimeMillis();
    System.out.println("StopWatch: " + getElapsedTime() + " milliseconds.");
    System.out.println("StopWatch: " + getElapsedTimeSecs() + " seconds.");
}
```

A função stop evoca métodos como getElapsedTime() e getElapsedTimeSecs() que calculam os valores de tempo por meio de operações entre os dois atributos.

Figura 22 - Métodos de Cálculo de Tempo

```
public long getElapsedTime() {
    return stopTime - startTime;
}

//elaspsed time in seconds
public double getElapsedTimeSecs() {
    double elapsed;
    elapsed = ((double)(stopTime - startTime)) / 1000;
    return elapsed;
}
```

REALIZAÇÃO DE TESTES

Os testes realizados a seguir foram para fins de quantificar em números suas funções de complexidades e ressaltar elementos para conclusões, as seguintes observações são dadas para realização dos testes:

- O mesmo vetor é utilizado para as comparações de resposta entre os algoritmos de ordenação;
- Foram realizados testes com vetores ordenados, invertidos (ordenados inversamente) e randômicos;
- Os tamanhos dos vetores utilizados são de 1.000, 2.000, 4.000, 8.000 e 16.000;
- Após uma medição, as variáveis de comparação e troca são resetadas;
- A classe StopWatch foi utilizada para contabilizar o tempo de execução dos ordenamentos.
- A realização se deu em na máquinas correspondente às especificações da Tabela 1:

Tabela 1 - Configurações do Computador

	Modelo do Processador	Quantidade de Núcleos	Quantidade de Processadores Lógicos	RAM (GB)	Versão da IDE
Computador	Intel Core I3-9100F	4	4	8	13

ANALISE DE RESULTADOS

Utilizando os algoritmos apresentados acima, mensurados por meio das ferramentas já apresentadas. Cada experimento de vetor ordenado, inversamente ordenado e randômico.

- Os vetores gerados randomicamente são médias de uma amostra de 5 vetores randômicos.

- Toda característica de ordenação de vetor conta com uma tabela indicando a comparação entre os diferentes métodos de ordenação, juntamente da proporção em que eles crescem conforme aumenta o problema.

- A síntese da tabela é um gráfico de colunas, que utiliza representação gráfica para expressar visualmente a mensagem da tabela, uma observação a ser considerada é que a escala do gráfico é logarítmica decorrente da diferença de ordem de grandeza entre os valores mensurados
- A sequência de tabela e figura a se seguir se trata do caso em que o problema apresenta vetores já ordenados para a ordenação, utilizada para demonstrar como cada algoritmo de ordenação se comporta conforme diferentes contextos.

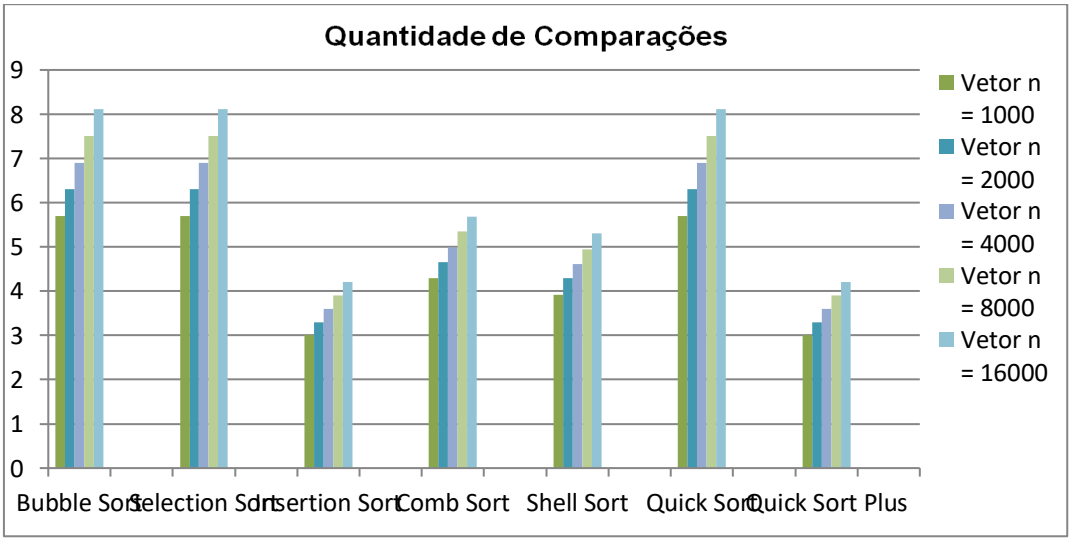
Observações e afirmações pertinentes são complementares à tabela, ao gráfico e à análise do próprio algoritmo em relação ao seu Big-Oh.
As tabelas abaixo correspondem aos vários testes já explicados abaixo

Tabela 2 - Vetor Ordenado (Comparações)

Vetor Ordenado (Comparações)	Vetor n = 1.000	Vetor n = 2.000	Vetor n = 4.000	Vetor n = 8.000	Vetor n = 16.000
Bubble Sort	499.500	1.999.000	7.998.000	31.996.000	127.992.000
Selection Sort	499.500	1.999.000	7.998.000	31.996.000	127.992.000
Insertion Sort	999	1.999	3.999	7.999	15.999
Comb Sort	19.706	45.373	98.719	221.383	474.742
Shell Sort	8.277	19.818	41.445	89.445	204.325
Quick Sort	499.500	1.999.000	7.998.000	31.996.000	127.992.000
Quick Sort Plus	1.002	2.002	4.002	8.002	16.002

O gráfico abaixo corresponde à comparação dos valores entre os diferentes tipos de ordenação, tal como evidenciar o aumento de passos computacionais em relação ao crescimento do tamanho do problema. O gráfico foi construído sob escala logarítmica decorrente das grandes diferenças de tamanho entre os valores contabilizados.

Gráfico 1 - Quantidade de Comparações (Ordenado)



A análise do gráfico 1 juntamente da tabela 2 é separada em 2 partes: comparação do aumento número de comparações contrastante ao crescimento do problema; comparação de eficiência entre algoritmos dado o contexto.

Algoritmos como Bubble Sort, Selection Sort e Quick Sort tem uma peculiaridade demonstrada acima com o Big-Oh sendo $O(n^2)$, exatamente correspondendo a $(n^2 - n)/2$, a proporção de crescimento é sempre quando dobra-se o problema, a complexidade temporal do algoritmo quadriplica.

Métodos de ordenação como Quick Sort Plus e Insertion Sort têm suas quantidades de comparações baseados bem diretamente ao tamanho do vetor, significando que são Big-Ohs $O(n)$.

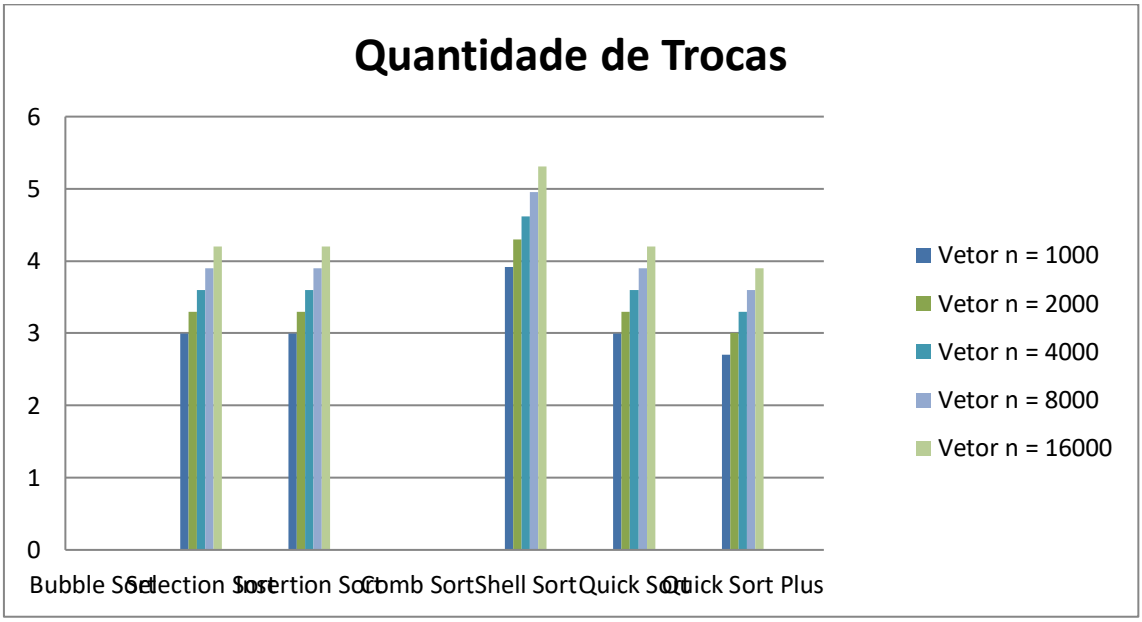
Assim, o crescimento das comparações é acentuado em Bubble, Selection e Quick Sort. Aproximadamente dobra de valores em Comb e Shell Sort e sua melhor proporção com a linearidade unitária de Quick Sort Plus e Insertion Sort.

Tabela 3 - Vetor Ordenado (Trocas)

Vetor Ordenado (Trocas)	Vetor n = 1.000	Vetor n = 2.000	Vetor n = 4.000	Vetor n = 8.000	Vetor n = 16.000
Bubble Sort	0	0	0	0	0
Selection Sort	999	1.999	3.999	7.999	15.999
Insertion Sort	999	1.999	3.999	7.999	15.999
Comb Sort	0	0	0	0	0
Shell Sort	8.277	19.818	41.445	89.445	204.325
Quick Sort	999	1.999	3.999	7.999	15.999
Quick Sort Plus	501	1.001	2.001	4.001	8.001

A tabela 3 é seguida do gráfico 2 em escala logarítmica:

Gráfico 2 - Quantidade de Trocas (Ordenado)



A quantidade de trocas realizadas pelos algoritmos Selection, Insertion, Quick e Quick Sort Plus em seu melhor cenário de ordenação (já ordenado) correspondente ao Big-Oh $O(n)$. Métodos como Bubble e seu aprimoramento, Comb Sort nem exigem trocas visto que suas quantidades de comparação já garantem esse estado ordenado. Shell Sort sendo o único caso de pior crescimento e resolução do caso por considerar um vetor em diferentes subvetores, executando algo semelhante ao Insertion Sort porém com mais passos computacionais.

Juntamente das conclusões das quantidades de comparações, o algoritmo Comb Sort lida bem com dados já ordenados, visto que a quantidade de comparações e trocas é menor em relação à manipulação do vetor. Outro algoritmo bem eficiente para casos já ordenados é o Quick Sort Plus, que faz muitas poucas comparações e trocas, que em geral englobam mais a determinação do pivô ao invés de trocas efetivas no vetor. Insertion Sort e Shell Sort tem a mesma quantidade de trocas e comparações, porém os valores de Insertion Sort são melhores para o vetor ordenado.

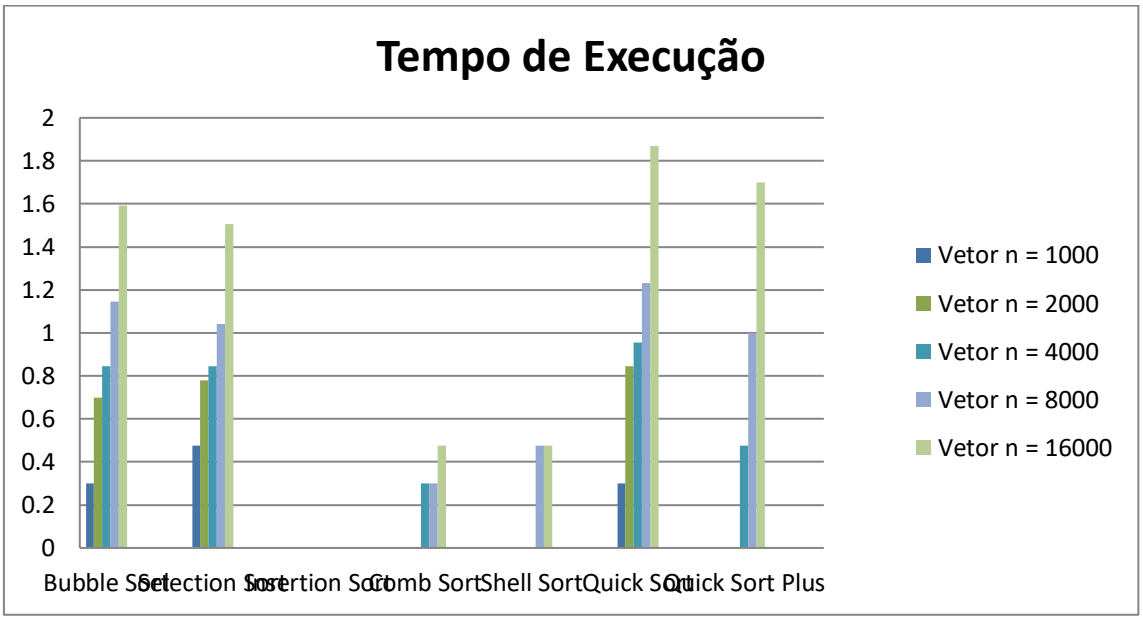
Quick, Selection e Bubble Sort podem tem bons números de trocas para caso do vetor ordenado, porém para a deliberação do vetor estar ordenado, exige muitos passos computacionais de comparação, tornando um dos piores

casos quando se trata de passos computacionais para quantificar comparações.

Tabela 4 - Vetor Ordenado (Tempo de Execução)

Vetor Ordenado (Tempo de Execução)	Vetor n = 1.000	Vetor n = 2.000	Vetor n = 4.000	Vetor n = 8.000	Vetor n = 16.000
Bubble Sort	2	5	7	14	39
Selection Sort	3	6	7	11	32
Insertion Sort	0	0	0	0	1
Comb Sort	1	1	2	2	3
Shell Sort	0	1	1	3	3
Quick Sort	2	7	9	17	74
Quick Sort Plus	1	1	3	10	50

Gráfico 3 - Tempo de Execução (Ordenado)



A conclusão que se tem é semelhante à acima: Insertion Sort, Comb Sort e Shell Sort são métodos bem eficientes vetores já ordenados, assim como o Quick Sort Plus apresentam mudanças significativas quanto ao pior

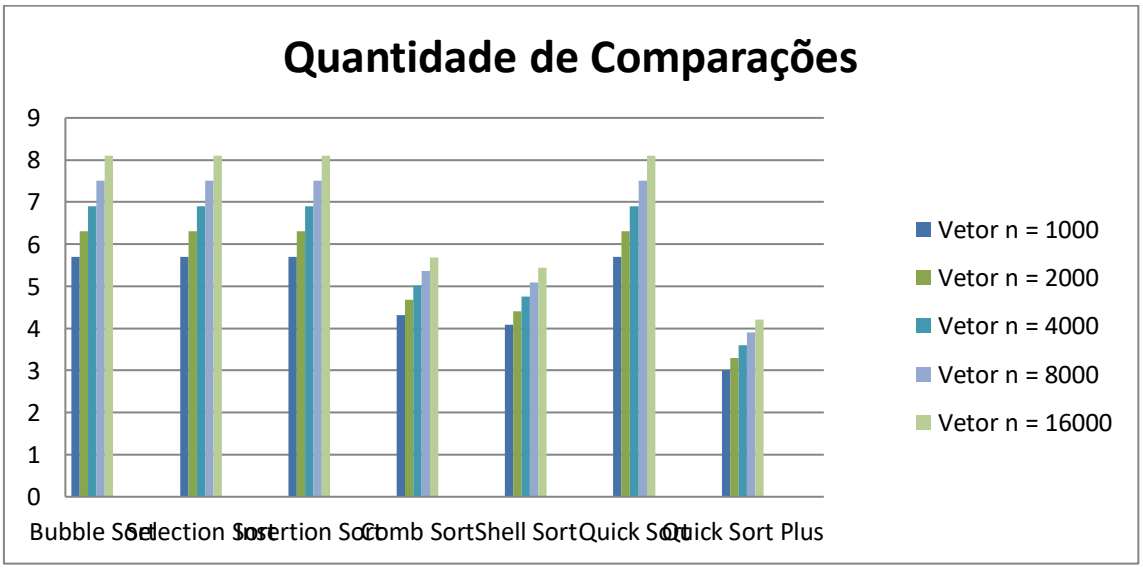
caso do Quick Sort. Uma observação a ser feita quanto ao valor nulo de alguns casos no gráfico, que se justificam por valores fora de escala de milissegundos do gráfico, nem em termos decimais (A ferramenta de medição não pega quantidade de tempos mais diminuta que essa).

A próxima sequência de testes corresponde ao vetor inversamente ordenado, chamado invertido nesse estudo:

Tabela 5 - Vetor Invertido (Comparações)

Vetor Invertido (Comparações)	Vetor n = 1.000	Vetor n = 2.000	Vetor n = 4.000	Vetor n = 8.000	Vetor n = 16.000
Bubble Sort	499.500	1.999.000	7.998.000	31.996.000	127.992.000
Selection Sort	499.500	1.999.000	7.998.000	31.996.000	127.992.000
Insertion Sort	500.499	2.000.999	8.001.999	32.003.999	128.007.999
Comb Sort	20.705	47.372	102.718	229.382	490.741
Shell Sort	12.067	25.264	57.427	122.135	274.823
Quick Sort	499.500	1.999.000	7.998.000	31.996.000	127.992.000
Quick Sort Plus	1.002	2.002	4.002	8.002	16.002

Gráfico 4 - Quantidade de Comparações (Invertido)



A quantidade de comparações para o caso dos inversamente ordenados continua tendo a mesma quantidade de comparações para o caso de Quick, Bubble e Selection Sort, todos sendo $O(n^2)$. Com o adicional do Insertion Sort ser afetado gravemente para este caso (nesse aspecto que ressalta-se uma vantagem do Shell Sort).

Outro algoritmo que modifica e aprimora é o Quick Sort Plus em relação ao clássico Quick Sort, com a estratégia da mediana de três, transforma seu pior caso $O(n^2)$ em um dos casos mais fáceis de complexidade $O(n)$.

O caso probabilístico de Shell Sort e Comb Sort dificultam uma simples expressão de suas funções de complexidades, no entanto é notável que esses não são seus piores casos.

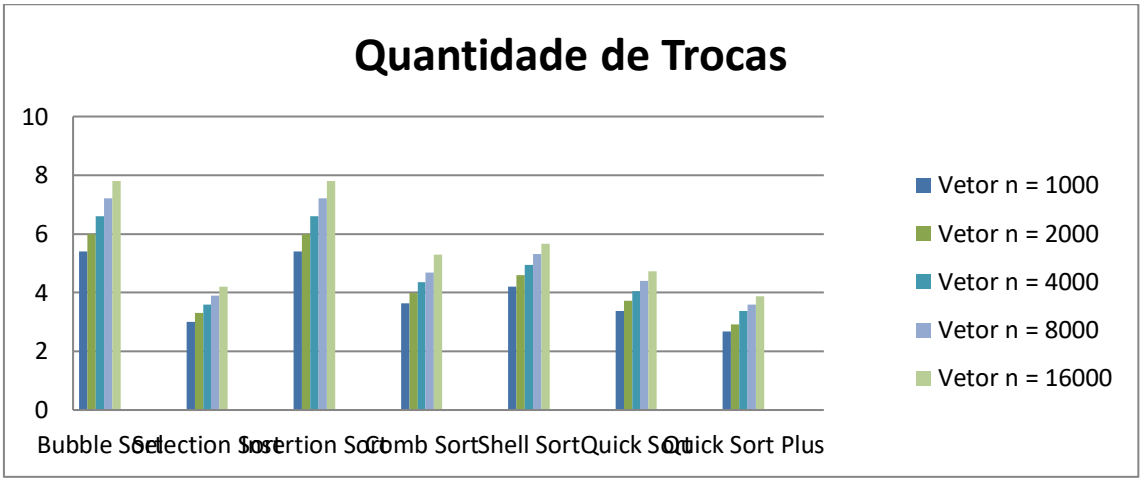
Tabela 6 - Vetor Invertido (Trocas)

Vetor Invertido (Trocas)	Vetor n = 1.000	Vetor n = 2.000	Vetor n = 4.000	Vetor n = 8.000	Vetor n = 16.000
Bubble Sort	499.500	1.999.000	7.998.000	31.996.000	127.992.000
Selection Sort	999	1.999	3.999	7.999	15.999
Insertion Sort	500.499	2.000.999	8.001.999	32.003.999	128.007.999
Comb Sort	1.536	3.452	7.394	15.718	33.202
Shell Sort	12.067	25.264	57.427	122.135	274.823
Quick Sort	999	1.999	3.999	7.999	15.999
Quick Sort Plus	505	1.005	2.005	4.005	8.005

Antes da análise gráfica, é interessante a comparação entre essa tabela de trocas (tabela 6) e a tabela de trocas do vetor ordenado (tabela 3). Bubble Sort que não precisava trocar nada quando ordenado, no pior caso de ordenação inversa, toda comparação resulta em troca. O mesmo com Insertion Sort e Shell Sort apesar de que Shell Sort não exigir muitas comparações. Selection

Sort, Quick Sort, Quick Sort plus mantêm a mesma quantidade de trocas de maneira independente da quantidade de comparações.

Gráfico 5 - Quantidade de Trocas (Invertido)



Uma observação interessante a se abstrair desse gráfico é a proximidade entre os valores de troca entre Selection Sort, Quick Sort e Quick Sort Plus. Apesar da enorme vantagem em quantidade de trocas do Quick Sort Plus, o gráfico reúne os valores dos três com proximidade decorrente da escala logarítmica, que os agrupa como valores maiores que 2, porém menores que 3 (isso correspondente a ordem de grandeza dos valores).

É notável relações $O(n^2)$ de número de comparações como Bubble Sort e Insertion Sort, tal como $O(n)$ no caso do Quick Sort e Selection Sort. Ainda sendo mais ótimo para o caso $O(n/2)$ do Quick Sort Plus. Consideradas aprimoramentos do Bubble Sort e Insertion Sort, os métodos Comb Sort e Shell Sort apresentam um diminuto valor de trocas em constraste.

A comparação dos métodos em relação a trocas e comparações conclui-se que:

- Bubble Sort e Insertion Sort apresentam seu pior caso, resultando em muita troca e muita comparação;
- Selection Sort e Quick Sort não trocam muitos valores no vetor, porém exige muitas comparações para deliberar tais ações;

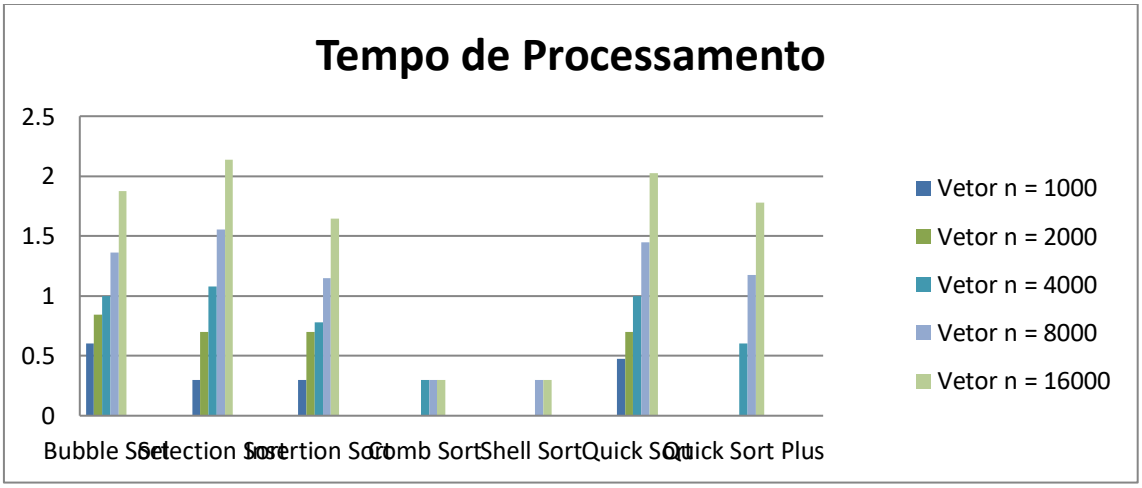
- Quick Sort Plus é extremamente eficiente na manipulação do vetor em função de comparações e trocas, porém sua implementação recursiva demanda passos computacionais que encarecem sua execução em questão temporal;
- Comb Sort e Shell Sort, sendo aprimoramentos do Bubble e Insertion respectivamente, apresentam as melhores performance dado o estado de completa inversão do vetor;

Todas as afirmações dadas acima são comprovadas pela tabela 7 e gráfico 6

Tabela 7 - Vetor Invertido (Tempo de Execução)

Vetor Invertido (Tempo de Execução)	Vetor n = 1.000	Vetor n = 2.000	Vetor n = 4.000	Vetor n = 8.000	Vetor n = 16.000
Bubble Sort	4	7	10	23	75
Selection Sort	2	5	12	36	137
Insertion Sort	2	5	6	14	44
Comb Sort	1	1	2	2	2
Shell Sort	1	0	1	2	2
Quick Sort	3	5	10	28	106
Quick Sort Plus	1	1	4	15	60

Gráfico 6 - Tempo de Processamento (Invertido)



O último contexto de teste nos quais os métodos foram submetidos são com vetores com geração randômica de valores. Para cada tamanho de problema, foi realizado testes com 5 amostras, sendo resultados de tabela a média dessa geração randômica, como por exemplo a tabela 8:

Tabela 8 - Vetor Randômico (Comparações)

Vetor Randômico (Comparações)	Vetor n = 1.000	Vetor n = 2.000	Vetor n = 4.000	Vetor n = 8.000	Vetor n = 16.000
Bubble Sort	499.500	1.999.000	7.998.000	31.996.000	127.992.000
Selection Sort	499.500	1.999.000	7.998.000	31.996.000	127.992.000
Insertion Sort	252.978,4	997.168,6	3.987.165,4	15.967.608	64.002.352,2
Comb Sort	21.704	49.371	111.515,8	245.380	526.538,8
Shell Sort	15.743,6	39.198,4	89.647,8	207.739,6	461.032,2
Quick Sort	7.597,4	16.572,6	38.335,4	80.981,8	176.036,6
Quick Sort Plus	1.002	2.002	4.002	8.002	16.002

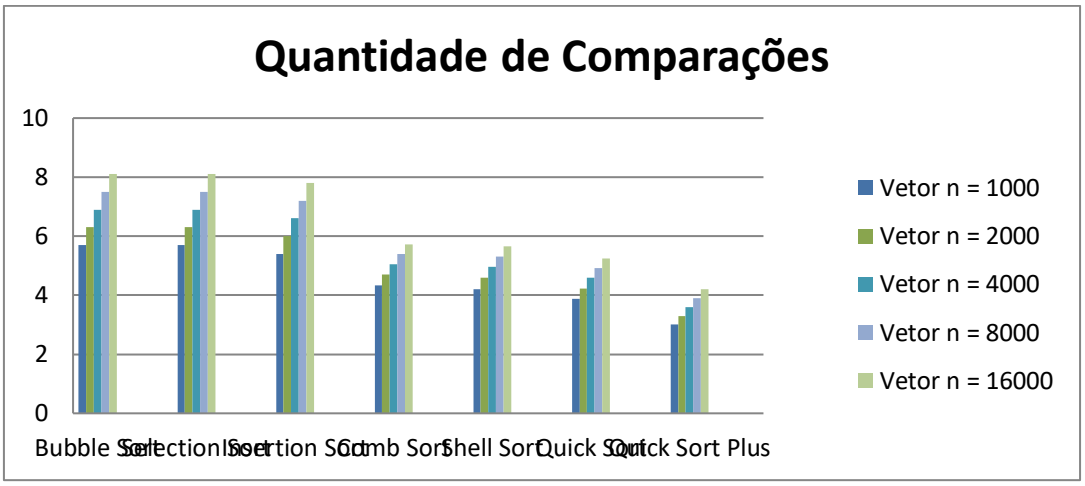
Uma análise interessante a se entender desses testes é que algumas metodologias de ordenação seguem um modelo rígido de comparações, como é o caso de: Bubble Sort, Selection Sort e Quick Sort Plus, que independente dos valores do vetor, sua estrutura segue um padrão fixo para identificar e ordenar.

Se tratando da aleatoriedade de amostras, percebe-se um vantajoso uso dos algoritmos recursivos (Quick Sort e Quick Sort Plus). E o exacerbado uso computacional dos métodos Bubble Sort, Selection Sort e Insertion Sort.

Outro ponto relevante a se considerar é o contraste entre a quantidade de comparações quando o vetor está ordenado crescente ou decrescente em relação ao aleatório, evidenciando que métodos como Comb Sort e Shell Sort lidam de maneira menos eficiente em caso de aleatoriedade. Os passos computacionais de Bubble Sort e Selection Sort se mantêm constantes $O(n^2)$,

enquanto Insertion Sort tem sua complexidade reduzida pela metade do pior caso (ordenação decrescente) $O(n^2/2)$.

Gráfico 7 - Quantidade de Comparações (Randômico)



Algo a ser ressaltado é que apesar de não lidarem tão bem com o caso de valores desordenados, Comb Sort e Shell Sort ainda apresentam resultados bons em comparação à suas versões básicas Bubble e Insertion Sort.

Tabela 9 - Vetor Randômico (Trocas)

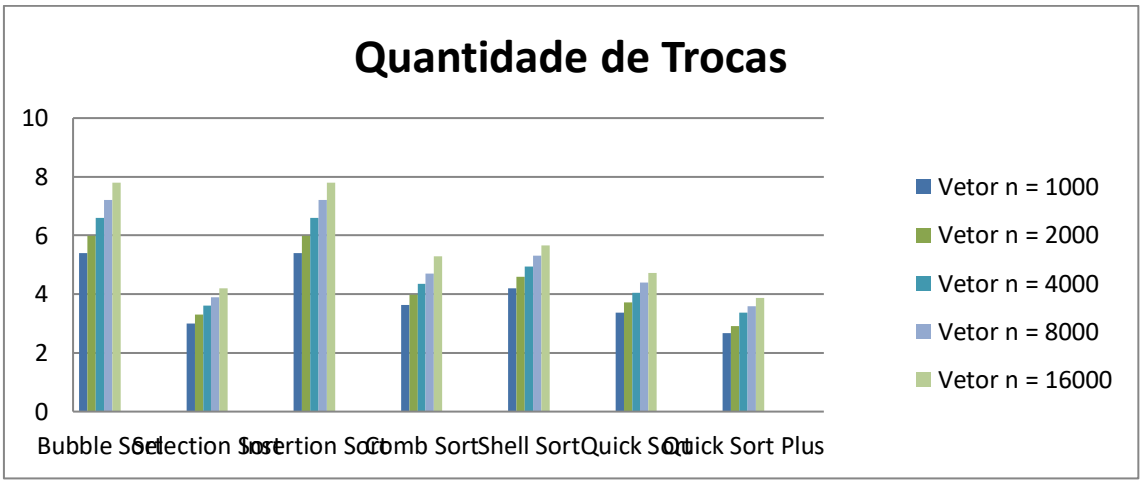
Vetor Randômico (Trocas)	Vetor n = 1.000	Vetor n = 2.000	Vetor n = 4.000	Vetor n = 8.000	Vetor n = 16.000
Bubble Sort	251.979,4	995.169,6	3.983.166,4	15.959.609	63.986.353,2
Selection Sort	999	1.999	3.999	7.999	15.999
Insertion Sort	252.978,4	997.168,6	398.7165,4	15.967.608	64.002.352,2
Comb Sort	4.303,8	9.787,2	22.608	49.352,6	192.817,4
Shell Sort	15.743,6	39.198,4	89.647,8	207.739,6	461.032,2
Quick Sort	2.360	5.219,8	11.351,6	24.610,2	53.094,2
Quick Sort Plus	473,2	839,6	2315	3.951,2	7.564,6

A quantidade de trocas possui algumas peculiaridades de métodos para métodos, como Insertion Sort e Shell Sort provando que a quantidade de trocas

corresponde à quantidade de comparações, quanto ao caso de Selection Sort, o limitante para quantidade de trocas é o tamanho do vetor.

Algoritmos como Comb Sort, Quick Sort, Selection Sort e Quick Sort Plus apresentam uma quantidade bem razoável de trocas referente ao tamanho do vetor, pois tentam ordenar um vetor de modo a colocar os elementos próximos de suas posições finais. Enquanto os demais métodos trabalham muito em encaixar um valor em seu lugar final, mesmo que traga o menor valor para o fim do vetor.

Gráfico 8 - Quantidade de Trocas (Randômico)



Relacionando os dois valores, para casos randômicos conclui-se:

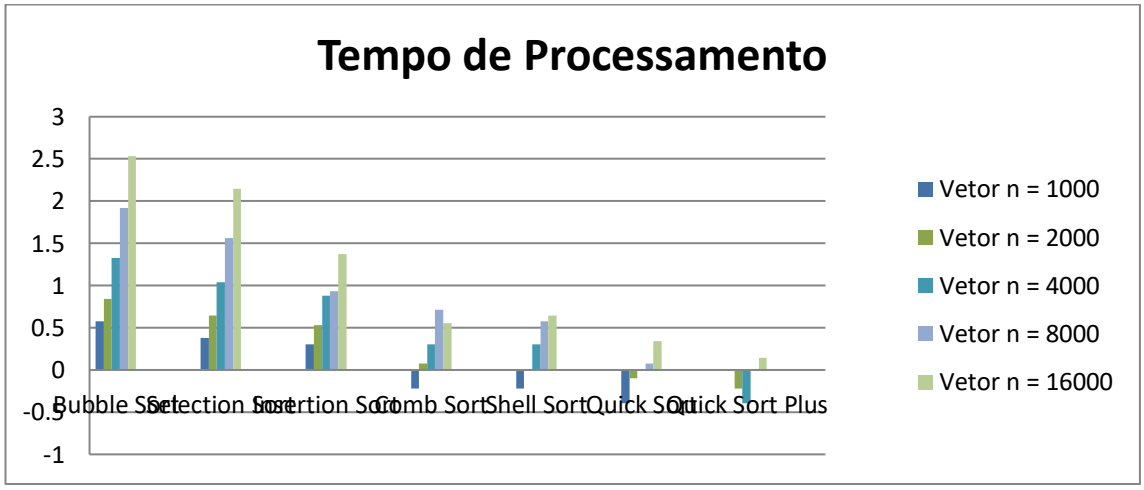
- Insertion Sort e Bubble Sort demandam muito tempo para comparar e trocar valores;
- Selection Sort, apesar de comparar bastante, faz poucas trocas, sendo o segundo melhor nesse aspecto quando se trata em vetores aleatórios;
- Shell Sort e Comb Sort tem seu desempenho razoável para amostras randômicas
- Quick Sort tem sua melhor execução para esses tipos de vetores, bastando apenas poucos ajustes para sua super otimização com o Quick Sort Plus, que tem valores absurdamente baixos em contraste ao tamanho do problema.

As afirmações acima são provadas pela tabela 10 e o gráfico 9:

Tabela 10 - Vetor Randômico (Tempo de Execução)

Vetor Randômico (Tempo de Execução)	Vetor n = 1.000	Vetor n = 2.000	Vetor n = 4.000	Vetor n = 8.000	Vetor n = 16.000
Bubble Sort	3,8	7	21,2	83,6	339,2
Selection Sort	2,4	4,4	11	36,6	139,6
Insertion Sort	2	3,4	7,6	8,6	23,4
Comb Sort	0,6	1,2	2	5,2	3,6
Shell Sort	0,6	1	2	3,8	4,4
Quick Sort	0,4	0,8	1	1,2	2,2
Quick Sort Plus	0	0,6	0,4	1	1,4

Gráfico 9 - Tempo de Processamento (Randômico)



Um elemento notável desse gráfico é que pela tabela registrar valores de média, existiram valores mensurados com milissegundos nulos, eventos que ocorreram em uma escala menor, a média com esses valores acarretaram valores decimais de média (menores que 1), que resultam em valores negativos no gráfico em escala logarítmica, podendo ser interpretadas como eventos que podem acontecer em frações de milissegundos.

CONCLUSÃO

Dado os resultados dos testes, resolve-se sintetizar todas as análises feitas em descrições diretas de casos bons para implementação de cada um dos algoritmos estudados nesse documento:

- Bubble Sort: Bem aplicado para conferir vetores ordenados, a troca por elementos da vizinhança dificulta muita casos de vetores decrescentes ou gerados aleatoriamente, principalmente quando o problema é de grande escala;

- Selection Sort: A quantidade de trocas sendo sempre fixa, sua aplicação exige muitas comparações, sendo aplicável caso isto algo não problemático. Tem as mesmas questões do Bubble Sort em relação a problemas de escalas grandes.
- Insertion Sort: Tem seu melhor desempenho para vetores praticamente já ordenados (desempenho linear de $O(n)$), apresenta as mesmas questões do Bubble Sort e Selection Sort.

- Comb Sort: Essa versão do Bubble Sort tem os melhores desempenhos para dados praticamente ordenados crescentemente ou decrescentemente, sua eficiência é notável conforme o crescimento do problema pelo deslocamento rápido de seus valores e aproximação de suas posições finais.

- Shell Sort: Sendo baseada na Insertion Sort, esse algoritmo só perde de seu antecessor no caso ordenado diretamente, sua ordenação em blocos de sub-listas tem o melhor desempenho em listas inversamente ordenadas, seu desempenho para dados randômicos é análogo à relação de Comb Sort para Bubble Sort, é razoável.

- Quick Sort: Sua implementação recursiva exige preocupação em gastos computacionais e de memória para seu uso, dados previamente ou praticamente ordenados são processadas de maneira bem lenta nesse algoritmo, extraindo seu potencial apenas em vetores de geração aleatória.

- Quick Sort Plus: Aprimoramento de extrema relevância dado o uso da mediana de 3, podendo tratar de vetores aleatórios ou praticamente ordenados com facilidade, o único problema seria o uso de memória causado pela recursão, que escala grandemente ao passo do problema.

Sendo um relatório de Análise de Algoritmos, outras conclusão além dos algoritmos é que essa documentação explora conceitos de Análise de Algoritmos, exercita práticas de instrumentação e análise de desempenho; e explica, de modo aprofundado, o conceito dos algoritmos de ordenação listados, tal como sua implementação na linguagem de programação Java.