Além dos gráficos, descrever os seguintes aspectos: -

**- Como foram gerados os vetores aleatórios;**

R. Usamos a função **Math.random** do java para gerar os valores aleatórios. A saída do **random** gera um double de 0 até 0.99, cujo resultado é multiplicado por 101. No final gerando um número inteiro até 99;

A função **Math.random** foi colocada estrategicamente dentro de um for que tem o objetivo de inserir o valor gerado pelo **random** dentro de um vetor, que é executado até que seu tamanho seja totalmente preenchido:  
  
Tamanhos: 5, 10, 50, 100, 1000, 10000;

Tudo isto dentro de um método GenerateRandVet que recebe o tamanho setado na main.

**- Como foram adaptados os algoritmos para realizar a contagem do número de comparações;   
R**: Criamos uma classe abstrata que tem um atributo comparações, que guarda o número de comparações de cada algoritmo. Cada algoritmo deve então herdar essa classe e implementar a contagem de comparações

**- O que foi desenvolvido para que cada algoritmo ordenasse os mesmos vetores que os demais;**

R: Ainda usando a mesma classe Abstrata, criamos um método setVetor, que copia um vetor paraum atributo interno, vetor. O algoritmo ordena apenas o atributo criado internamente.

**- Explicar os resultados dos experimentos com base na literatura;**

Com base nos dados extraídos do algoritmo feito pelo grupo, conseguimos chegar a algumas conclusões:

Algoritmos:

* Bubble Sort: Este algoritmo é eficiente para pequeno volume de dados, quando maior o dado, maior o número de comparações a serem realizadas.
  + Complexidade quadrática.
  + No melhor caso se torna O(n)
* Selection Sort: Falando em comparações é melhor do que o Bubble, mas ainda sim atinge nossa penúltima colocação se fomos pensar em um ranking de algoritmos quadráticos.
  + Complexidade quadrática.
* Insertion Sort: Entre os algoritmos acima é o que possui melhor comportamento de acordo com nossa tabela de comparações, possui bom desempenho para vetores de tamanho até 100.
  + Complexidade quadrática.

Os três algoritmos acima, possuem um péssimo desempenho quando se é necessário ordenar vetores com grande volume de dados. Indo ao oposto destes algoritmos temos os algoritmos abaixo:

* HeapSort (complexidade n log n) – Ordenação por seleção
* MergeSort (complexidade n log n) – Dividir e Conquistar
* QuickSort (complexidade n log n) – Dividir e Conquistar
* CountSort {complexidade O(n+k)} – Ordenação por não comparação**\***
  + Adequado quando o número de chaves não é muito maior do que o número de elementos
* RadixSort {complexidade O(n.k)} – Ordenação por cadeia de caracteres**\***
  + É um algoritmo de ordenação estável.
* BucketSort {complexidade O(n)} – Ordenação em recipientes**\***
  + Bom quando o número de chaves é pequeno e há em média poucos elementos por recipiente.
  + Único desta lista que pode se tornar um quadrático no pior caso O(n²).

**\***Também conhecidos como algoritmos de ordenação por não comparação.

Todos estes algoritmos têm um excelente desempenho em termo de comparações quando se é necessário ordenar uma grande quantidade de dados, porém quando falamos de volume pequeno de dados suas comparações chegam a ser maior do que os três primeiros algoritmos listado neste documento.

Se colocarmos em ranking os algoritmos acima, chegaríamos no seguinte resultado:

* Counting Sort possui o menor número de comparação de dados, ele começa a apresentar seu bom desempenho a partir de vetores de tamanho 10, se mantendo eficiente para grande volume de dados.

**O algoritmo perfeito**

Com este estudo chegamos as seguintes conclusões:

‘Não existe algoritmo único que seja perfeito’. Para alcançarmos melhores resultados independentemente do tamanho da entrada de dados temos que usar o conceito de algoritmo **Híbrido**, ou seja, juntarmos algoritmos para se comportarem bem quando o volume de dados é menor e também ter um bom comportamento quando o volume de dados é maior.