Universidade Estadual de Montes Claros
Programa de Pos-Graduacão em Modelagem Computacional e Sistemas (PPGMCS)
Acadêmico Jorge Gustavo dos Santos Pinho

Algoritmos de Ordenação

Este trabalho apresenta uma comparação entre os algoritmos ***Bubble Sort***, ***Insertion Sort***, ***MergeSort*** *e* ***QuickSort***. Para cada um dos algoritmos foram testados o melhor e pior caso. O desenvolvimento foi feito usando linguagem de programação **Python**, e os gráficos dos experimentos foram feitos usando a biblioteca **matplotlib**, já o tempo que cada algoritmo levou para ordenar uma determinada lista foi medido usando a biblioteca **timeit**.

Todos os algoritmos foram testados com entradas que crescem de 0 a 10000 incrementadas em 10 para melhor e pior caso, os gráficos foram gerados com os tempos de pior e melhor caso para cada uma das entradas.

Em primeiro lugar serão apresentados os algoritmos usados pra análise e posteriormente uma comparação dos quatro algoritmos com o seu tempo de execução e o gráfico de crescimento de cada um.

Bubble Sort

O primeiro algoritmo implementado foi o *Bubble Sort*, ele é bem simples, composto por dois laços **for** aninhados, um laço percorre a matriz do inicio para o fim e o outro percorre do fim para o início. No segundo **for** é feita a comparação dos elementos para realização das trocas. Ele é um algoritmo relativamente lento e não mostra diferenças significativas para melhor e pior caso, tendo um custo $\Theta(n^2)$.

Para os testes realizados foi feita uma implementação do algoritmo adaptada do exemplo presente no livro *Data Structures and Algorithm Analysis* de *Clifford A. Shaffer*, e pode ser vista no Quadro 1.

Quadro 1- Implementação do algoritmo Bubble Sort.

```
def bubble_sort(A, n):
    for i in range(n):
        for j in range(n - 1, i, -1):
            if A[j] < A[j - 1]:
                A[j], A[j - 1] = A[j - 1],
A[j]
    return A;
```

Independente de como o vetor está o *Bubble Sort* sempre executa n^2 comparações. Na Figura 1 podemos ver o gráfico de tempo do algoritmo *Bubble Sorte* para o melhor e pior caso, e na Figura 2 o gráfico de crescimento para um custo $\Theta(n^2)$.

Figura 1 - Gráfico do algoritmo *Bubble Sort* para o pior caso em verde e melhor caso em azul.

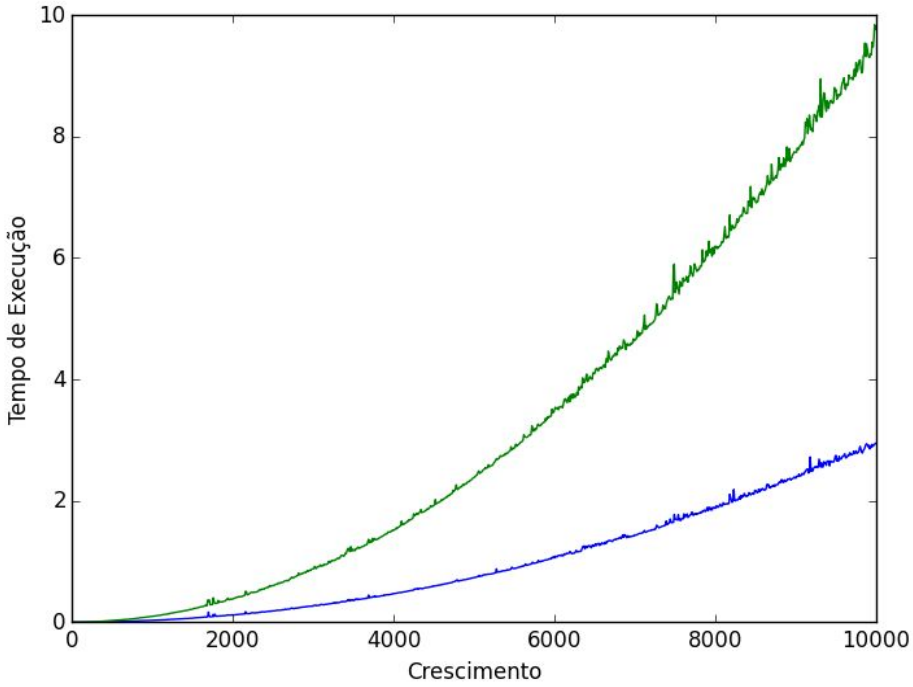
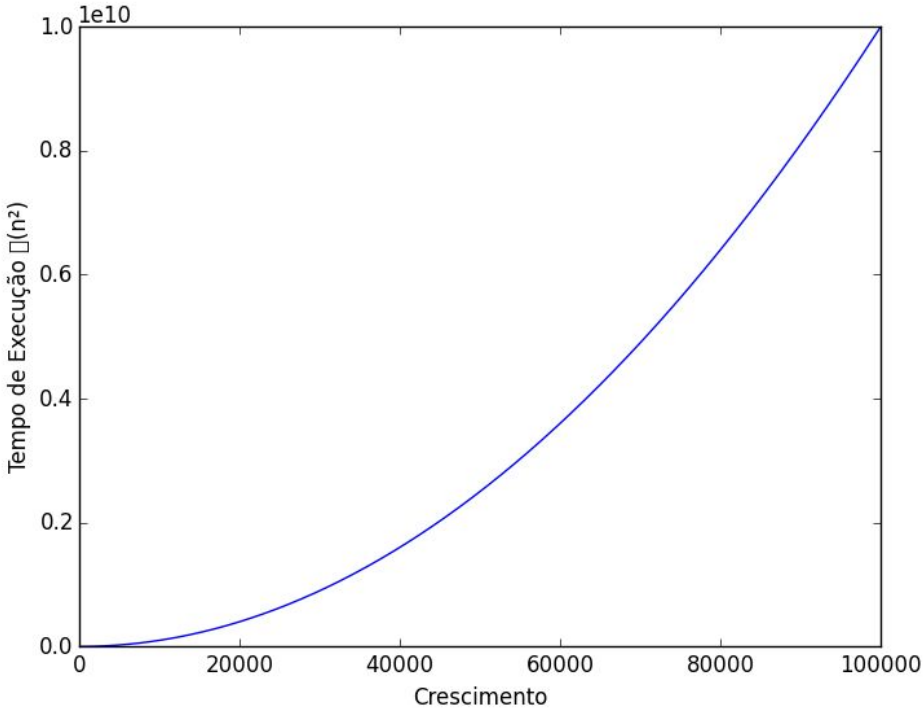


Figura 2 - Gráfico de Θ (n²).



Ao olharmos para os gráficos das figuras 1 e 2 podemos ver que no teste empírico a curva de crescimento do algoritmo para o pior caso (Figura 1 em verde) é bem semelhante ao

crescimento n^2, no entanto para o melhor caso o algoritmo se mostrou um pouco mais rápido do que n^2 mostrando uma curva que está entre n e n^2.

Insertion Sort

O *Insertion Sort* percorre uma lista desordenada é insere os elementos em suas posições em uma lista já ordenada. Assim como o *Bubble Sort* ele também pode ser feito com dois laços **for**, um para percorre a lista desordenada e outro para procurar a posição de inserção do elemento na lista já ordenada.

Para uma lista desordenada o tempo de execução do *Insertion Sort* é $\Theta(n^2)$ em contrapartida o tempo de execução para uma lista ordenada é de $\Theta(n)$, isso se deve ao fato de que quando a lista já está ordenada ele não entra no segundo laço **for**. Esse comportamento nos dá uma boa escolha quando trabalhamos com listas levemente desordenadas. O Quadro 2 mostra uma implementação do algoritmo, também adaptada do livro *Data Structures and Algorithm Analysis* de *Clifford A. Shaffer*.

Quadro 2 - Implementação do algoritmo *Insertion Sort*.

```
def insertion_sort(A, n):
    for i in range(n):
        for j in range(i, 0, -1):
            if A[j] > A[j - 1]:
                break
            A[j], A[j - 1] = A[j - 1], A[j]
    return A
```

Figura 3 - Gráfico do algoritmo *Insertion Sort* para o pior caso em verde e melhor caso em azul.

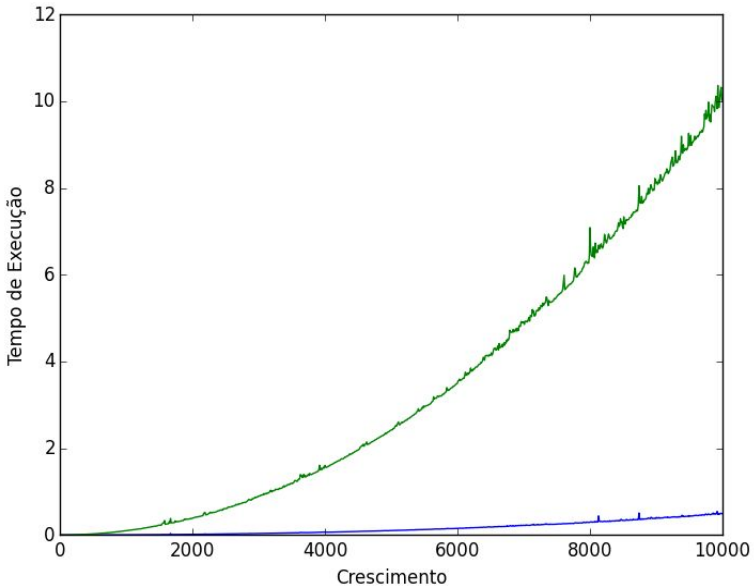
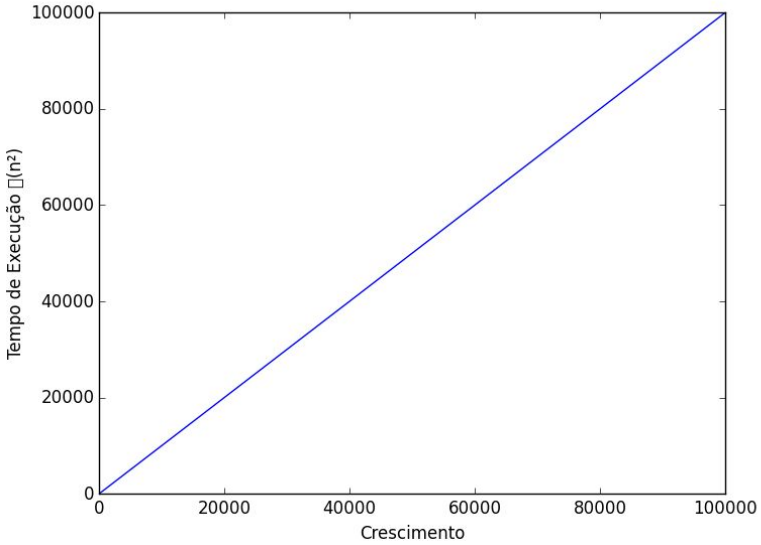


Figura 4 - Gráfico de Θ (n).



Assim como o *Bubble Sort* o *Insertion Sort* apresentou um gráfico muito parecido com a curva n² para o pior caso nos testes empíricos, e comprovando o que a diz teoria para os testes de melhor caso o algoritmo mostrou um crescimento linear. Se compararmos o gráfico da Figura 4 que mostra um crescimento linear podemos ver claramente que para o melhor caso o algoritmo *Insertion Sort* apresenta crescimento linear.

Mergesort

O *Mergesort* usa uma abordagem de problema dividir para conquistar, ele quebra a lista em duas partes, ordena cada parte depois mescla as duas partes. O custo desse algoritmo é Θ (nlogn), seu conceito é relativamente simples porém é um pouco complexo para implementar, neste trabalho foi feita um implementação recursiva, também adaptada do livro *Data Structures and Algorithm Analysis* de *Clifford A. Shaffer,* como pode ser visto no Quadro 3.

Quadro 3 - Implementação do algoritmo *Mergesort*.

```
def merge_sort(A, temp, left, rigth):

    if (rigth == left):
        return
    mid = int((left+rigth)/2)

    merge_sort(A, temp, left, mid)
    merge_sort(A, temp, mid+1, rigth)

    for i in range(left, rigth+1):
        temp[i] = A[i]

    i1 = left
    i2 = mid+1

    for curr in range(left, rigth+1):
        if i1==mid+1:
            A[curr] = temp[i2]
            i2 = i2 +1
        elif i2>rigth:
            A[curr] = temp[i1]
            i1 = i1 +1
        elif temp[i1] < temp[i2]:
            A[curr] = temp[i1]
            i1 = i1 +1
        else:
            A[curr] = temp[i2]
            i2 = i2 +1
```

Figura 5 - Gráfico de Θ (nlogn).

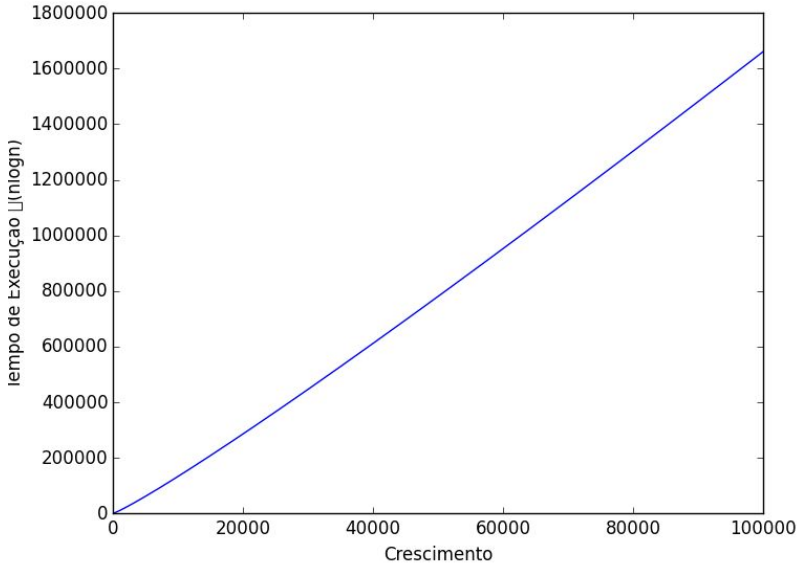
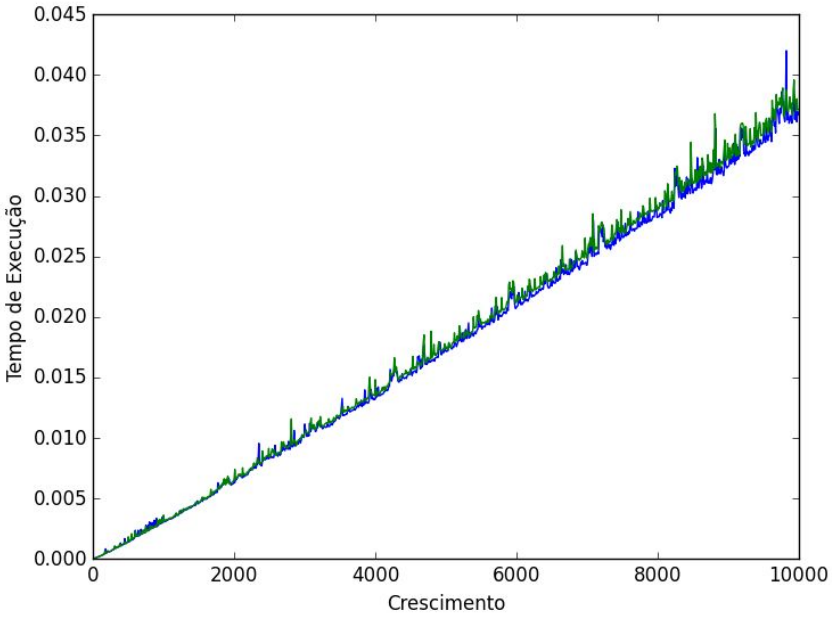


Figura 6 - Gráfico do algoritmo *Mergesort* para o pior caso em verde e melhor caso em azul.



Ao analisarmos o gráfico gerado pelos testes realizados com o algoritmo *MergeSort* Figura 6 e o gráfico nlogn Figura 5 percebemos uma grande semelhança. Outro analise a ser feita são as curvas para melhor e pior caso que se assemelham em muito, a diferença mais notável é que para o melhor caso a curva está levemente abaixo da apresentada no pior caso, mostrando que no melhor caso o algoritmo é um pouco mais rápido, mas ainda assim seu custo é Θ (nlogn). A conclusão que se chega com os testes empíricos é que o algoritmo se comporta com um fator de crescimento nlogn para o melhor e pior caso.

QuickSort

O último algoritmo analisado foi o *QuickSort,* ele é considerado um algoritmo de ordenação rápida no entanto seu tempo de execução para pior caso é $\Theta(n^2)$, em contrapartida pra média ele apresenta um tempo Θ(nlogn). O *QuickSort* assim como o *Mergesort* também usa a abordagem dividir para conquistar, que no seu caso pode ser dividido em três passos: dividir; conquistar; e combinar.

Uma particularidade do *QuickSort* e que o pior caso e quando a entrada ordenada já está ordenada pois o número de comparações será n^2.

O *QuickSort* usado para teste foi implementa com base no livro Algoritmos Teoria e Prática de Thomas H. Cormen e pode ser visto no Quadro 4. Já o gráfico do crescimento do algoritmo nos testes empíricos pode ser visto na Figura 7.

Quadro 4 - Implementação do algoritmo *QuickSort*.

```
def partition(A, p, r):
    x = A[r]
    i = p-1
    for j in range(p, r):
        if A[j] <= x:
            i += 1
            A[i], A[j] = A[j], A[i]
    A[i+1], A[r] = A[r], A[i+1]
    return i+1

def quick_sort(A, p, r):
    if p<r:
        q = partition(A, p, r)
        quick_sort(A, p, q-1)
        quick_sort(A, q+1, r)
```

Figura 7 - Gráfico do algoritmo *QuickSort* para entradas randômica simulando casos médios e melhores.

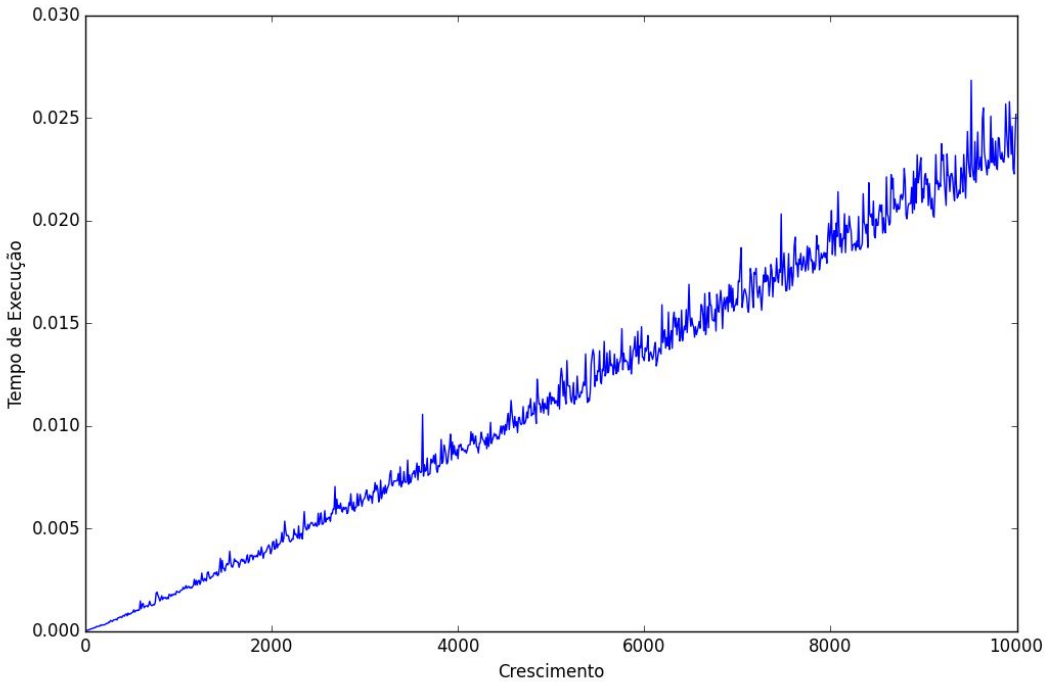
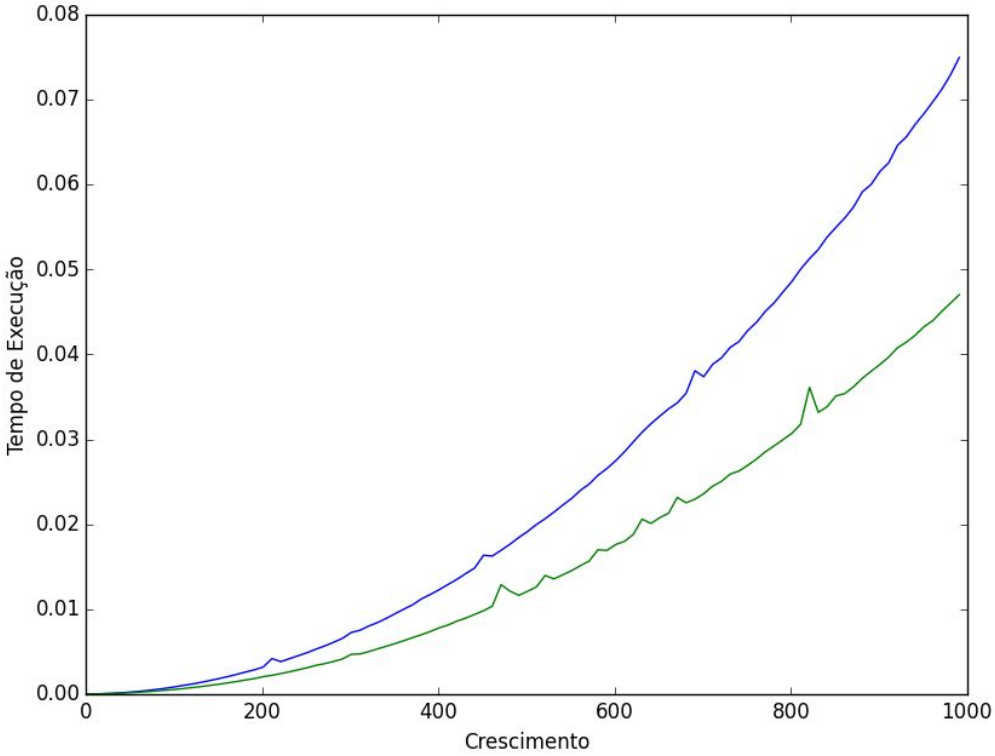


Figura 8 - Gráfico do algoritmo *QuickSort* para o pior caso vetor ordenado em azul e pior caso vetor em ordem inversa em verde.

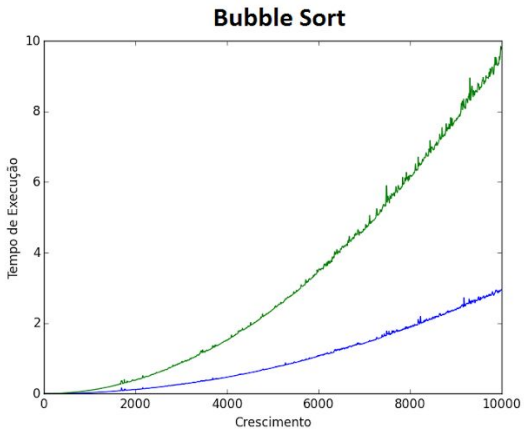


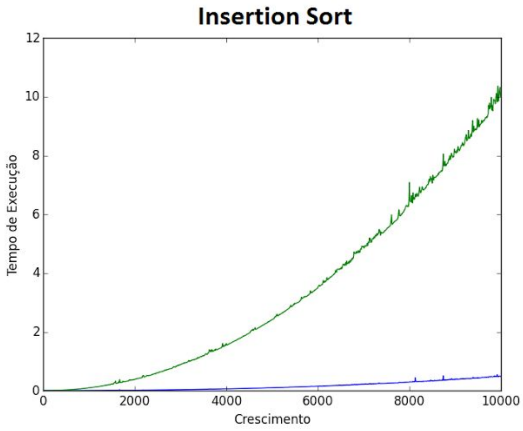
Ao analisarmos o gráfico de crescimento gerado pelo *Quick Sort* podemos ver que ele se comporta com uma curva n^2 para uma entrada ordenada e um pouco menor que n^2 para uma entrada em ordem inversa, já pra um vetor de valores randômicos o *Quick Sorte* mostrou uma taxa de crescimento satisfatória que se assemelha em muito com uma curva nlogn do *Merge Sort*.

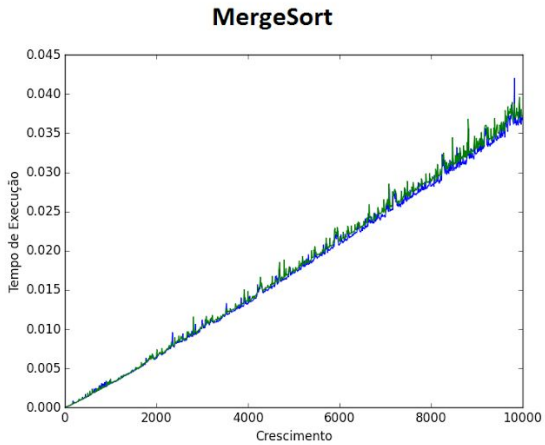
Conclusão

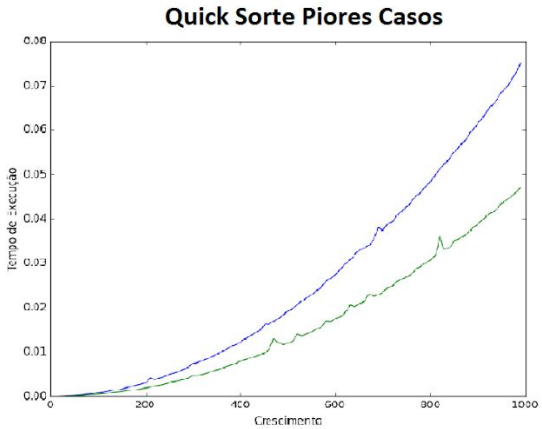
A conclusão que se chega é que cada algoritmo tem suas particularidades e que sua escolha deve ser feita tomando como critério o problema a ser resolvido. Assim a Figura 8 mostra os gráficos de crescimentos dos quatro algoritmos lado a lado, com ele podemos tem uma visão de qual algoritmo se sai melhor para uma entrada de tamanho n.

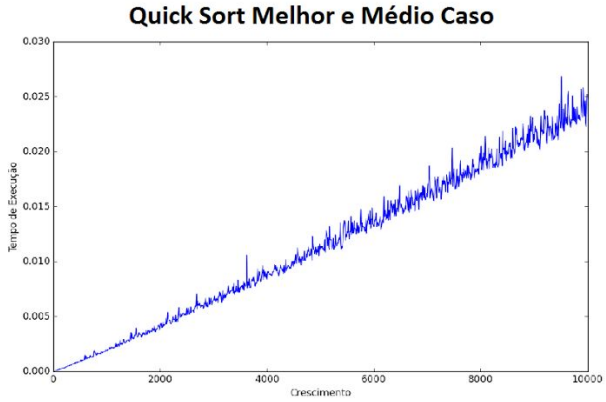
Outra observação é que apesar da simples implementação dos algoritmos *Bubble Sort* e *Insertion* seus tempos de execução e seu crescimento são bem maiores se comparado aos outros dois algoritmos. E apesar do Quick Sort mostrar um comportamento ruim para o pior caso ele se sai bem pra médio e melhor caso.











Com os testes empíricos foi possível visualizar com clareza o que nos diz a teoria de analise de complexidade de algoritmos, também foi possível demonstrar na prática que o comportamento definido em teoria é valido quando aplicado em problemas reais.

Bibliografia

SHAFFER, Clifford A. **Data Structures and Algorithm Analysis.** diponivel em: **<**http://people.cs.vt.edu/shaffer/Book/C++3e20130328.pdf**>;** Acesso 28/10/2015.

CORMEN, Thomas H. **Algoritmos: teoria e prática.** 2.ed. Rio de Janeiro: Elsevier, 2002.