Selection Sort

O selection sort é um algoritmno de ordenação, que funciona de maneira que o menor elemento do vetor v será sempre colocado em no índice i e o elemento em i será colocado no local do elemento trocado, sendo i = 0 e após a troca i++;

O algoritmo:

```
def SelectionSort(A):
    for i in range(len(A)):
        min_idx = i
        for j in range(i+1, len(A)):
            if A[min_idx] > A[j]:
                min_idx = j
        A[i], A[min_idx] = A[min_idx], A[i]
    return A
```

Fonte: Geeks for Geeks (https://www.geeksforgeeks.org/python-program-for-selection-sort/)

Descrição do algoritmo

O Algoritmo possui dois laços. Um laço interno e outro laço externo, o laço externo é responsável por manter a posição do primeiro elemento após os elementos já ordenado do vetor, e ele sempre será acrescido de um assim que o menor valor do vetor for encontrado no laço interno do algoritmo.

Por exemplo:

vetor = [6, 8, 2 ,5 ,3 ,1 ,9, 0, 7]

O indice (chamaremos de i) do laço externo, inicialmente, será 0 apontando para o valor 6 do vetor. O índice do laço interno (chamaremos de j) irá começar em (i + 1), ou seja no índice 1 apontando para o valor 8 do vetor. Como o valor 8 não é menor que 6, o índice j é incrementado em 1, apontando para o valor 2, como 2 é menor que 6 agora o índice de menor item do vetor apontará para o valor 2. O algoritmo vai continuar executando até o final do vetor, encontrando um valor menor que o valor à qual o índice de menor valor está apontado, esse índice é atualizado para o novo menor valor encontrado. No caso desse vetor, ele será atualizado novamente quando encontrar o número 1 no vetor, presente no índice 5, e depois será atualizado novamente quando encontrar o 0, no índice 7.

Como o algoritmo chegou ao final do vetor, e o índice de menor valor é 7, que aponta para o 0, então o 0 é trocado com o valor presente no índice i, e o algoritmo reinicia com o j começando novamente em i + 1.

Isso se repete até que i chegue ao final do vetor.

Complexidade do algoritmo

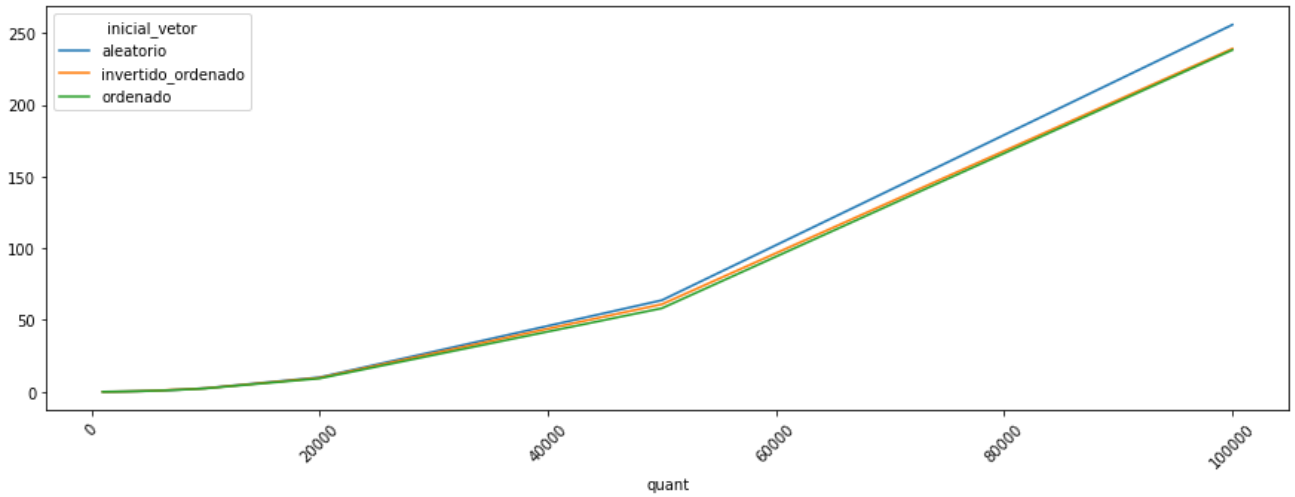
Esse algoritmo possui uma das piores complexidades. Pois a cada nova iteração do laço externo, o vetor é percorrido por inteiro no laço interno.

Para o teste do algoritmo foi gerado alguns vetores. A quantidade de itens nesses vetores vão de 2000, até 100000, sendo eles ordenados, ordenados descrescente e completamente aleatórios.

Abaixo podemos ver o gráfico de tempo que o selection sort demora com essa amostra de vetores criados.

In [10]:
```
# plotando os dados do selection sort
selection_data.pivot(index='quant', columns='inicial_vetor', values='tempo').
plot(rot=45, figsize=(15,5))
```

Out[10]: <AxesSubplot:xlabel='quant'>



Devido ao comportamento do algoritmo, ele sempre percorrer o vetor inteiro, a complexidade será sempre a mesma. Então não importa se o vetor está ordenado ou não, pois ele sempre executará a mesma quantidade de passos no laço interno.

Como o algoritmo executará no laço interno n vezes, n vezes, a complexidade será n x n = n^2.

Portanto a complexidade do algoritmo será O(n^2).

- pior caso: O(n^2)
- caso médio: O(n^2)
- melhor caso: O(n^2)

O Selection sort é um algoritmo in-place, portanto utiliza o vetor que está sendo ordenado para realizar a ordenação. Como o algoritmo utiliza uma variavel auxiliar para fazer a troca do menor item do vetor com o item presente no indice i, exite a complexidade O(1) a mais de espaço.

Vantagens

- É um algoritmo in-place, ou seja, sua complexidade de espaço é do tamaho do vetor a ser ordenado
- Ele é muito simples de ser implementado
- É um dos mais velozes na ordenação de vetores pequenos

Desvantagens

- Ele é um dos algoritmos mais lentos para vetores de grandes
- Ele sempre executa o laço interno inteiro (n^2 - n)/2 vezes, independendo se o vetor já estava ou não ordenado

É possível ver que a complexidade é a mesma para os tipos de vetores gerados e que o tempo de execução nesses três cenários são pouco distantes.

Abaixo é possível ver o tempo de execução do seleciont sort com um vetor de 100000 itens.

In [11]:
```
# Tabela de tempo com 100000 itens no vetor
selection_data.query('quant == 100000')[['inicial_vetor', 'tempo']].head()
```

Out[11]:

	inicial_vetor	tempo
36	ordenado	238.160951
37	invertido_ordenado	239.075675
38	aleatorio	255.737923

Em comparação ao selection sort, o mergesort que possui uma complexidade de O(n log n) tem um tempo de execução muito menor para o mesmo vetor de 10000 itens.

In [12]:
```
# Tabela de tempo do merge sort com 100000 itens no vetor
mergesort_data.query('quant == 100000')[['inicial_vetor', 'tempo']].head()
```

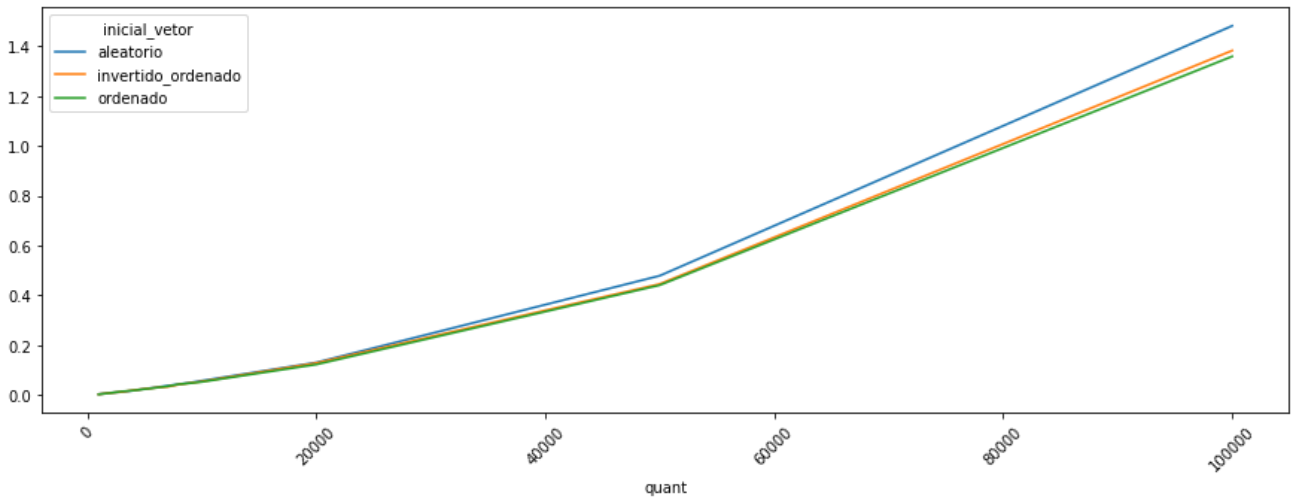
Out[12]:

	inicial_vetor	tempo
36	ordenado	1.358318
37	invertido_ordenado	1.381733
38	aleatorio	1.480874

Abaixo tem um gráfico da execução do merge sort para fins de comparação.

In [13]:
```
# plotando os dados do merge sort
mergesort_data.pivot(index='quant', columns='inicial_vetor', values='tempo').
plot(rot=45, figsize=(15,5))
```

Out[13]:
```
<AxesSubplot:xlabel='quant'>
```



Bubble Sort

O funcionamento

O bubble sort é um algoritmo simples, seu comportamento consiste em percorrer um vetor v inteiro, utilizando um índice i e comparando os valores v[i] e v[i+1] (no caso de um Bubble sort crescente), caso v[i] for maior que v[i+1] os dois são trocados de lugar. O algoritmo se repete até que o vetor v inteiro é percorrido e nenhuma troca for feita!

O algoritmo

```
def BubbleSort(A):
    trocado = True
    while trocado:
        trocado = False
        for i in range(len(A) - 2):
            if A[i] > A[i+1]:
                A[i], A[i+1] = A[i+1], A[i]
                trocado = True
    return A
```

Complexidade do algoritmo

Este também é um algoritmo de complexidade simples. Como o algoritmo irá percorrer o vetor e ir trocando os valores que estiverem desordenados, caso o vetor já esteja ordenado o algoritmo irá executar n vezes.

No pior caso, onde o vetor esteja ordenado inversamente o algoritmo terá que percorrer o vetor n vezes, n vezes: n x n = n^2

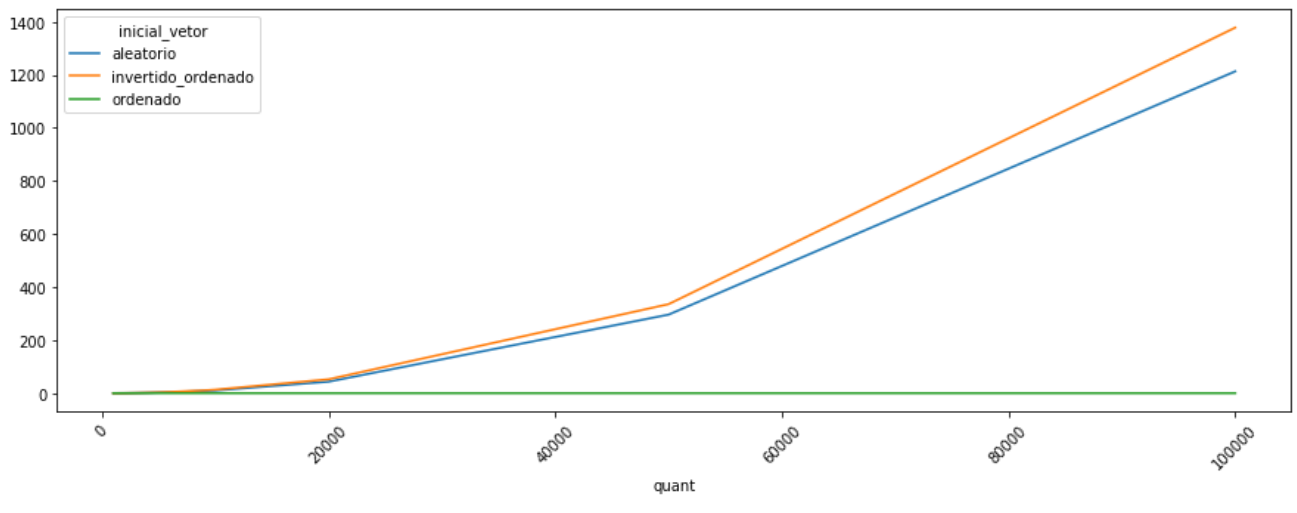
O algoritmo assim como o selection sort também é in-place, ou seja, utiliza o próprio vetor para a ordenação. É utilizado uma variavel auxiliar para a inversão dos valores que estiverem fora, logo exite uma complexidade de espaço de O(1).

- pior caso: $O(n^2)$
- caso médio: $O(n^2)$
- melhor caso: O(n)

Abaixo é possível ver o gráfico de performance do algoritmo Bubble sort.

In [17]:
```
bubble_data = pd.read_csv('csv/performance-bubble.csv')
bubble_data.pivot(index='quant', columns='inicial_vetor', values='tempo').plo
t(rot=45, figsize=(15,5))
```

Out[17]:
```
<AxesSubplot:xlabel='quant'>
```



No gráfico acima é possível a diferença de tempo no algoritmo entre o vetor ordenado e os outros vetores. O gráfico do tempo de execução demonstra a complexidade O(n) para os vetores já ordenados e a complexidade O(n^2) para o pior e médio caso.

In [15]:
```
# Tabela de tempo do bubble sort com 100000 itens no vetor
bubble_data.query('quant == 100000')[['inicial_vetor', 'tempo']].head()
```

Out[15]:

	inicial_vetor	tempo
36	ordenado	0.012742
37	invertido_ordenado	1378.076939
38	aleatorio	1213.258456

O vetor ordenado possui o tempo de execução de 0.012 segundos, enquanto os outros possuem um tempo de mais de 1200 segundos.