Atividades de Complexidade de Algoritmos Aula 4

1- Mostre, intuitivamente, que o algoritmo Selection Sort não é estável para as chaves.

O algoritmo Selection Sort não é estável para as chaves por causa da sua natureza de troca direta de elementos. A estabilidade em algoritmos de ordenação significa que elementos com chaves iguais permanecerão na mesma ordem relativa após a ordenação, como estavam originalmente.

Suponha que você tenha uma lista de elementos com chaves iguais, mas em ordens diferentes. Por exemplo, [2a, 1, 2b, 3, 2c], onde os elementos 2a, 2b e 2c têm chaves iguais "2".

Quando o Selection Sort encontra o menor elemento (no caso, "1"), ele o trocará com o primeiro elemento da lista, independentemente de suas chaves. Neste ponto, a lista se tornará [1, 2, 2b, 3, 2c].

Agora, quando o Selection Sort encontra o próximo menor elemento (que é novamente "2"), ele o trocará com o segundo elemento da lista. No entanto, essa troca não mantém a ordem relativa original dos elementos com a mesma chave "2". A lista agora será [1, 2b, 2, 3, 2c].

A ordem relativa dos elementos com chave "2" foi alterada, o que significa que o Selection Sort não é estável.

Em contraste, algoritmos estáveis, como o Merge Sort ou o Insertion Sort, garantem que a ordem relativa dos elementos com chaves iguais seja mantida durante a ordenação, tornando-os preferidos quando a estabilidade é necessária.

2- O que é necessário para que a implementação do algoritmo Insertion Sort seja estável para as chaves?

Para tornar a implementação do algoritmo Insertion Sort estável para as chaves, você precisa garantir que a troca de elementos seja realizada apenas quando

estritamente necessário, ou seja, quando um elemento deve ser movido para a sua posição correta na ordenação.

Critério de comparação estável: Certifique-se de que a comparação de elementos seja feita de forma estável. Isso significa que, se dois elementos forem considerados iguais em termos de ordenação (ou seja, seu valor/chave é igual), a implementação deve garantir que o elemento que apareceu primeiro na lista original permaneça antes do outro elemento na lista ordenada. Isso pode ser feito comparando não apenas o valor/chave, mas também o índice original dos elementos.

Inserção ordenada: Ao inserir um elemento na parte já ordenada da lista, certifique-se de que ele seja inserido na posição correta, mantendo a ordem relativa original dos elementos com chaves iguais. Isso geralmente envolve mover elementos à direita do elemento inserido apenas se o elemento a ser inserido for menor em relação a eles.

Aqui está um pseudocódigo simplificado para tornar a implementação do Insertion Sort estável:

```python
def insertion_sort_stable(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        # Mova os elementos maiores que a chave para a direita
        # apenas se a chave for menor que eles e mantenha a ordem relativa
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1

        # Insira a chave na posição correta
        arr[j + 1] = key

# Exemplo de uso:
my_list = [(3, "Alice"), (2, "Bob"), (3, "Eve"), (1, "Charlie")]
insertion_sort_stable(my_list)
print(my_list)

```

Neste exemplo, estamos ordenando uma lista de tuplas, onde o primeiro elemento da tupla é a chave de ordenação. O algoritmo Insertion Sort modificado mantém a ordem relativa original dos elementos com chaves iguais, tornando-se estável.

3- Implemente o algoritmo Quicksort. Qual a complexidade de espaço utilizada na sua implementação?

```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quicksort(left) + middle + quicksort(right)

# Exemplo de uso:
my_list = [3, 6, 8, 10, 1, 2, 1]
sorted_list = quicksort(my_list)
print(sorted_list)
```

A complexidade de espaço (espaço adicional além do espaço para armazenar a lista original) para esta implementação do Quicksort é O(n), onde "n" é o tamanho da lista a ser ordenada. Isso ocorre devido ao uso de listas auxiliares (left, middle, right) para particionar os elementos. Cada chamada recursiva cria novas listas auxiliares, e seu tamanho total é limitado pelo tamanho da lista original, resultando em uma complexidade de espaço linear. Note que esta implementação não é in-place, o que significa que requer espaço adicional para armazenar as listas auxiliares.