Lógica de Programação Orientada a Objetos

Bem-vindos!

Objetivos

- ✓ Algoritmos de Ordenação

Insertion sort

```
def insertion_sort(arr):

    for i in range(1, len(arr)):

        key = arr[i]

        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key

    return arr

arr = [12, 11, 13, 5, 6]
arr = insertion_sort(arr)
print(arr)
```

Qual a complexidade desse algoritmo?

Selection sort

```
def selection_sort(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

A = [64, 25, 12, 22, 11]
A = selection_sort(A)
print(A)
```

Qual a complexidade desse algoritmo?

Bubble sort

```
def bubble_sort(arr):
    n = len(arr)

    for i in range(n):

        for j in range(0, n-i-1):

            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

arr = [64, 34, 25, 12, 22, 11, 90]
arr = bubbleSort(arr)
print(arr)
```

Qual a complexidade desse algoritmo?

Merge sort

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) / 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i+=1
            else:
                arr[k] = R[j]
                j+=1
            k+=1
```

```
while i < len(L):
            arr[k] = L[i]
            i+=1
            k+=1

        while j < len(R):
            arr[k] = R[j]
            j+=1
            k+=1
    return arr

arr = [12, 11, 13, 5, 6, 7]
arr = merge_sort(arr)
print(arr)
```

Qual a complexidade desse algoritmo?

Heap sort

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i] # swap

        heapify(arr, n, largest)
    return arr
```

```
def heap_sort(arr):
    n = len(arr)
    for i in range(n, -1, -1):
        heapify(arr, n, i)

    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

arr = [ 12, 11, 13, 5, 6, 7]
arr = heap_sort(arr)
print(arr)
```

Qual a complexidade desse algoritmo?

Bucket sort

```
def insertion_sort(b):
    for i in range(1, len(b)):
        up = b[i]
        j = i - 1
        while j >= 0 and b[j] > up:
            b[j + 1] = b[j]
            j -= 1
        b[j + 1] = up
    return b


x = [0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434]
x = bucket_sort(x)
print(x)
```

Qual a complexidade desse algoritmo?

```
def bucket_sort(x):
    arr = []
    slot_num = 10

    for i in range(slot_num):
        arr.append([])

    for j in x:
        index_b = int(slot_num * j)
        arr[index_b].append(j)

    for i in range(slot_num):
        arr[i] = insertion_sort(arr[i])

    k = 0
    for i in range(slot_num):
        for j in range(len(arr[i])):
            x[k] = arr[i][j]
            k += 1
    return x
```

Quick sort

```
def partition(arr, low, high):
    i = low - 1
    pivot = arr[high]

    for j in range(low , high):

        if arr[j] < pivot:
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

```
def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)

        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)
    return arr

arr = [10, 7, 8, 9, 1, 5]
n = len(arr)
arr = quick_sort(arr, 0, n - 1)
print(arr)
```

Qual a complexidade desse algoritmo?