ALESSANDRO SOARES DA SILVA

MATRICULA: 20231023705

3° LISTA DE ALGORITMO

1. Implemente as funções da seção 6.5 do livro do Cormen 4th Ed. em sua linguagem favorita e proponha um exemplo de uso com uma demonstração.

```python
import numpy
import random
def heapify(arr, n, i): #função responsável pela troca dos elemento na árvore
    maior = i
    esquerda = 2 * i + 1
    direita = 2 * i + 2

    if esquerda < n and arr[esquerda] > arr[maior]:
        maior = esquerda

    if direita < n and arr[direita] > arr[maior]:
        maior = direita

    if maior != i: #verifica se há alteração nos índices
        arr[i], arr[maior] = arr[maior], arr[i]  # Troca os elementos
        heapify(arr, n, maior)  # Chama recursivamente para o filho modificado
################################################################################
def build_heapSort(arr): # função que constroui a heapmax
    n = len(arr)
    # Constrói um heap máximo
    for i in range((n // 2) - 1, -1, -1): #construção de baixo para cima
        heapify(arr, n, i)
################################################################################
def heapSort(arr): #faz a ordenação, começando pelos ultimos valores
    n = len(arr)
    build_heapSort(arr)
    # Extrai elementos um por um do heap começando pelo maiores (ultimos)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]  # Troca os elementos
        heapify(arr, i, 0)
################################################################################

def heapmax(arr): # função para retornar o maior elemento
    build_heapSort(arr)
    return arr[0]
################################################################################

def heap_extract_max(arr): # função que estrai o maior elemento da árvore
    n = len(arr)
    if (n < 1):
        raise("HEAP UNDERFLOW")
    build_heapSort(arr)
    max_elemento = arr[0]
    arr[0] = arr.pop()  # retira o elemento da arvore
    heapify(arr,len(arr),0) # refaz ávore heapmax
    print ("maximo elemento extraido", max_elemento)
    return max_elemento
################################################################################
```

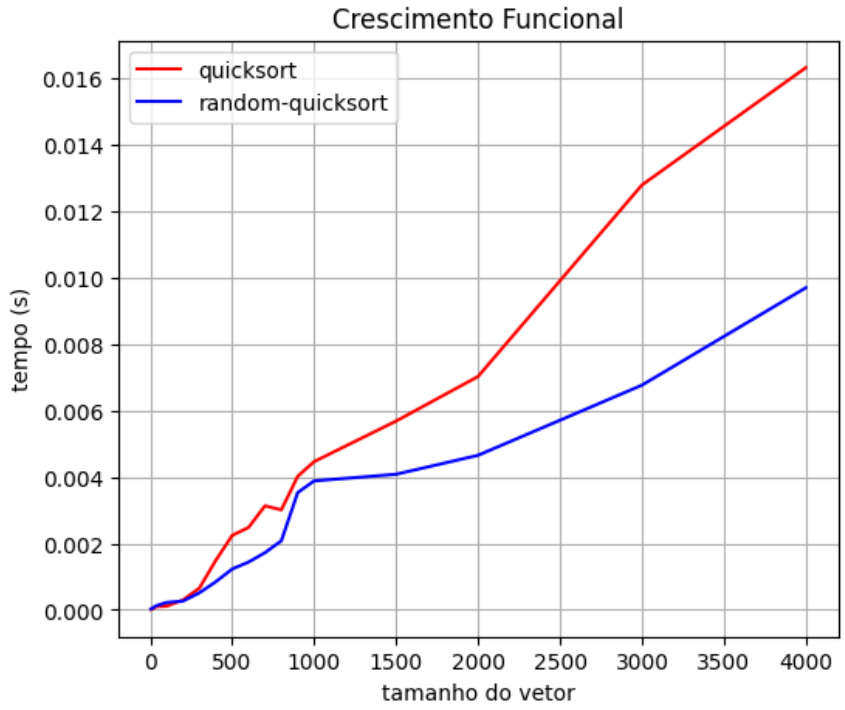
```python

def heap_increase_key(arr, i, k):
    if(k < arr[i]):
      raise ValueError("Nova chave é menor que a chave atual")
    arr[i] = k # incrementa um elemento no conjunto na possição i fornecida
    build_heapSort(arr) #Controi um nova árvore incluindo o novo elemento
################################################################################

def max_heap_insert(arr, k):
    n = len(arr)
    # Adiaciona o elemento no final do array
    arr.append(k)
    #É necessário chamar a função heap_increse_key para ajustar a posição
    heap_increase_key(arr, n, k)

if __name__ == "__main__":

    # Exemplo de uso:
    #arr = numpy.random.randint(-20, 20, 13)
    arr = [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]
    print("Array fornecido:", arr)
    heapSort(arr)
    print("Array ordenado:", arr)
    max_value = heapmax(arr)
    print("Maior valor:", max_value)
    heap_extract_max(arr)
    print("Novo array ordenado com o maior valor retirado:", arr)
    heap_increase_key(arr, 0, 25)
    print("Novo array ordenado com o maior valor incrementado:", arr)
    max_heap_insert(arr, 5)
    print("novo array com o valor inserido na árvore:", arr)
```

```
Array fornecido: [16, 4, 10, 14, 7, 9, 3, 2, 8, 1]
Array ordenado: [1, 2, 3, 4, 7, 8, 9, 10, 14, 16]
Maior valor: 16
maximo elemento extraido 16
Novo array ordenado com o maior valor retirado: [14, 10, 9, 4, 7, 8, 3, 1, 2]
Novo array ordenado com o maior valor incrementado: [25, 10, 9, 4, 7, 8, 3, 1, 2]
novo array: [25, 10, 9, 4, 7, 8, 3, 1, 2, 5]
```

2. Mostre com experimentos numéricos quando suas próprias implementações de Quicksort e do Quicksort aleatório são mais vantajosas quando comparadas uma com a outra.

```python
import numpy
import time
import  random
import matplotlib.pyplot as plt
def partition(arr, menor, maior):
    # tomar o ultimo elemento como pivo
    pivo = arr[maior]
    # i recebe o menor elemento do vetor
    i = menor - 1

    for j in range(menor, maior):
        # Se o elemento atual é menor que o pivo
        if (arr[j] <= pivo):
            i = i + 1
            # troca o elemento
            arr[i], arr[j] = arr[j], arr[i]
    # Trocar o arr[i+1] e o pivo
    arr[i + 1], arr[maior] = arr[maior], arr[i + 1]

    return i + 1

def quicksort(arr, menor, maior):
    if (menor >= maior):
        return
    else:
        #Obeter o indice da partição
        p = partition(arr, menor, maior)
        # Ordenar recursivamento os elementos e depois particionar
        quicksort(arr, menor, p-1)
        quicksort(arr, p+1, maior)
    return arr

def randomized_quicksort(vetor):
    if len(vetor) <= 1:
        return vetor

    pivo_index = random.randint(0, len(vetor) - 1)
    pivo = vetor[pivo_index]
    esquerda = []
    direita = []

    for indice, elemento in enumerate(vetor): #itera o vetor retornndo o indice e o valor nessa posição
        if indice == pivo_index:
            continue  # Ignora o proprio pivo
        if elemento < pivo:
            esquerda.append(elemento)
        else:
            direita.append(elemento)

    return randomized_quicksort(esquerda) + [pivo] + randomized_quicksort(direita)

def gerar_numeros_aleatorios_nao_repetidos(n):
  numeros = []
  while len(numeros) < n:
    numero = numpy.random.randint(1, 10000)
    if numero not in numeros:
      numeros.append(numero)
  return numeros

```

```
60
61 if __name__ == "__main__":
62
63     #vetor = gerar_numeros_aleatorios_nao_repetidos(900)
64     vetor = numpy.random.randint(1, 100000, 5)
65     print('Vetor de entrada:', vetor)
66     n = len(vetor)
67     start_time = time.time()
68     #A = quicksort(vetor, 0, n - 1)
69     A = randomized_quicksort(vetor)
70     end_time = time.time()
71     time = end_time - start_time
72     print('Vetor de saída:', A)
73     print("Execution time:", end_time - start_time, "seconds")
```



X = ([5, 10, 15, 20, 30, 40, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1500, 2000, 3000, 4000, 5000, 10000, 20000, 30000, 40000, 50000, 100000, 200000, 300000, 400000, 500000, 1000000, 2000000, 3000000, 4000000, 5000000, 10000000])

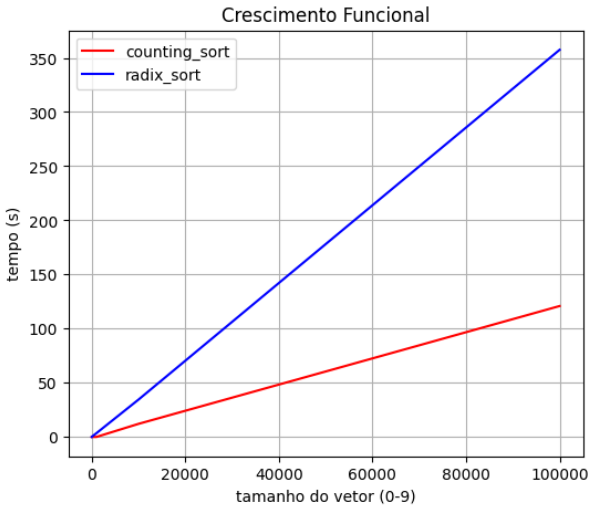
Y = ([0.0000121, 0.0000226, 0.0000305, 0.0000529, 0.0000767, 0.000103, 0.000113, 0.000298, 0.000656, 0.001490, 0.00224, 0.00248, 0.00313, 0.00301, 0.00402, 0.00446, 0.00568, 0.00702, 0.01278, 0.01631, 0.02344, 0.02970, 0.05912, 0.1316, 0.2076, 0.2773, 0.3852, 0.7741, 1.81, 2.89, 3.95, 5.09, 11.56, 32.77, 50.780, 74.000, 112.000, 406.000])

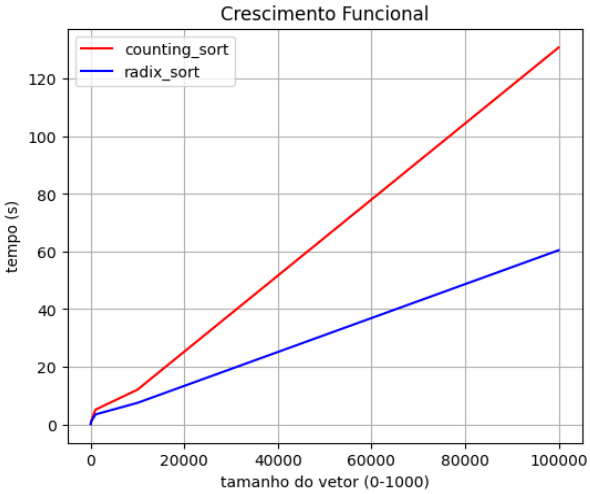
Z = ([0.0000317, 0.0000403, 0.0000655, 0.0000670, 0.0001090, 0.000131, 0.000223, 0.000266, 0.000514, 0.000847, 0.00123, 0.00144, 0.00172, 0.00208, 0.00353, 0.00388, 0.00408, 0.00465, 0.00676, 0.00969, 0.01392, 0.01698, 0.03646, 0.0811, 0.1167, 0.1769, 0.2127, 0.4680, 1.09, 1.65, 2.35, 3.05, 06.89, 16.24, 26.385, 42.070, 56.250, 165.000])

3. Mostre com experimentos numéricos quando o Radix-sort com o Count-sort é mais rápido que o Count-sort sozinho. Utilize suas próprias implementações ou alguma implementação existente explicando os resultados.

```python
import numpy as np

def radix_sort(A):
    n = len(A)
    max_val = A[0]
    for i in range(1,n):
        if(A[i] > max_val):
            max_val = A[i]  # verificação de maior valor no array
    exp = 1
    while (max_val/exp) > 0:
        counting_sort(A, exp) # chamada recurssiva do counting_sort
        exp *= 10   #muda para a casa das dezenas, centanas, ....
    return A
def counting_sort(A, exp): # acrescentado o argumento exp, para considerar (uni,dez,cen,...)
    n= len(A)
    saida = [0] * n # cria um vetor com as dimensões do vetor original
    count = [0] * 10 # cria um vetor com 10 digito (0-9)
    for i in range(n):
        index = (A[i]/exp) # feito operação com o exp para tratar das dez, cen,..
        count[int(index%10)] += 1 # acumula repetições nas posições do vetor count
    for i in range(1,10):
        count[i] += count[i-1] # calcula quantas posiçoes do count serão repetidas [1,3,4,6]
    i = n-1 # atribui em i a ultima posição do vetor A
    while (i>=0):
        index = (A[i]/exp) # insere os valores do vetor em index
        saida[count[int(index % 10)]-1] = A[i] # insere o valor de A nas respectivas, respeitando as repetições e sequencia
        count[int(index%10)] -= 1 #decrementa a variavel de iteração
        i -= 1
    i = 0
    for i in range(0, len(A)):
        A[i] = saida[i] # insere os valores sequenciais no vetor A

if __name__ == "__main__":
    #arr = [1, 4, 3, 2, 2, 5, 4, 2, 9, 6]
    arr = np.random.randint(0, 1000, 20)
    print("vetor fornecido",arr)
    vetor = radix_sort(arr)
    print("vetor ordenado:", vetor)
```

```python
import numpy as np
def counting_sort(A):
    n= len(A)
    saida = [0] * n # cria um vetor com as dimensões do vetor original
    count = [0] * 10 # cria um vetor com 10 digito (0-9)
    for i in range(n):
        index = A[i] # insere os valores do vetor em index
        count[index] += 1 # acumula repetições nas posições do vetor count
    for i in range(1,10):
        count[i] += count[i-1] # calcula quantas posiçoes do count serão repetidas [1,3,4,6]
    i = n-1 # atribui em i a ultima posição do vetor A
    while (i>=0):
        index = A[i] # insere os valores do vetor em index
        saida[count[index] - 1] = A[i] # insere o valor de A nas respectivas, respeitando as repetições e sequencia
        count[A[i]] -= 1 #reduz a quantidade de repetições para a proxima iteração (trabalha com uni/dez/cen)
        i -= 1 #decrementa a variavel de iteração
    i = 0
    for i in range(0, len(A)):
        A[i] = saida[i] # insere os valores sequenciais no vetor A
    return A

if __name__ == "__main__":
    #arr = [1, 4, 3, 2, 2, 5, 4, 2, 9, 6]
    arr = np.random.randint(0, 9, 20)
    print("vetor fornecido",arr)
    vetor = counting_sort(arr)
    print("vetor ordenado:", vetor)
```





Couting_sort: Na minha imlplementação o couting-sort conseguiu ordenar de forma bem rápida os valores de 0 a 9, mesmo para valores de vetores com mais de 10000 posições, no entanto ele precisa de muito espaço de memoria já que tem que criar 3 vetores do mesmo tamanho.

Radix_sort: ficou muito lento para ordenar valores de 0 a 9, mesmo para valores de vetores pequenos abaixo de 500 posições, no entanto quando os numero foram colocados acima de 1000, o gráfico se inverteu

Conclusão: A grande diferença está no tamanho de cada valor, valores com mais digitos são ideais para o radix-sort. E ocupam menos espaço de memoria mesmo para numeros grandes.