TP 03

Ana Neri

28 Fevereiro 2023

Exercícios práticos sobre complexidade parte 2.

Exercício 1 Para cada uma das funções de ordenação abaixo

- Identifique o melhor e pior casos em termos do número de comparações entre elementos do array e em termos do número de trocas efetuadas.
- Calcule o número de comparações entre elementos do array efetuadas nesses casos identificados.

1.
```
def bubbleSort(arr):
    n = len(arr)
    last_swap = n - 1
    while last_swap != 0:
        new_last_swap = 0
        for j in range(0, last_swap):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                new_last_swap = j
        last_swap = new_last_swap
```

2.
```
def insertionSort(arr):
    n = len(arr)
    if n <= 1:
      return
    for i in range(1, n):
        key = arr[i]
        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key
```

Exercício 2 Utilize uma árvore de recorrência para encontrar limites superiores para o tempo de execução dados pelas seguintes recorrências (assuma que para todas elas $T(0)$ é uma constante):

1. $T(n) = n + T(n-1)$
2. $T(n) = n + T\left(\frac{n}{2}\right)$
3. $T(n) = k + 2 * T(n-1)$ com k constante
4. $T(n) = n + 2 * T\left(\frac{n}{2}\right)$
5. $T(n) = k + 2 * T\left(\frac{n}{3}\right)$ com k constante

Exercício 3 Considere a seguinte definição da função que ordena um vetor usando o algoritmo de *merge sort*.

```
def mergeSort(arr, l, r):
    if l < r:
        # Same as (l+r)//2, but avoids overflow for
        # large l and h
        m = l+(r-l)//2
        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)
```

Considere que a função `merge (arr, l,m,r)` executa em tempo $T_{merge(N)} = 2 * N$. Apresente uma relação de recorrência que traduza o tempo de execução de `mergeSort` em função do tamanho do vetor argumento. Apresente ainda uma solução dessa recorrência.

Exercício 4 Considere o programa *quick sort*:

```
# Function to find the partition position
def partition(array, low, high):
    # choose the rightmost element as pivot
    pivot = array[high]
    # pointer for greater element
    i = low - 1
    # traverse through all elements
    # compare each element with pivot
    for j in range(low, high):
        if array[j] <= pivot:
            # If element smaller than pivot is found
            # swap it with the greater element pointed by i
            i = i + 1
            # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])
```

```
    # Swap the pivot element with the greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])
    # Return the position from where partition is done
    return i + 1
# function to perform quicksort
def quickSort(array, low, high):
    if low < high:
        # Find pivot element such that
        # element smaller than pivot are on the left
        # element greater than pivot are on the right
        pi = partition(array, low, high)
        # Recursive call on the left of pivot
        quickSort(array, low, pi - 1)
        # Recursive call on the right of pivot
        quickSort(array, pi + 1, high)
```

Este algoritmo segue uma estratégia *divide and conquer*, isto é começa por segmentar o array em duas partes que são ordendas separadamente.

Calcule o número de comparações efetuadas entre elementos do array, e o núemro de trocas no array. Apresente uma relação de recorrência que traduza o nº de comparações.

Apresente qual os valores da complexidade no pior caso e no caso médio.