TP 03

Ana Neri

28 Fevereiro 2023

Exercícios práticos sobre complexidade parte 2. (A maior parte das solução veio do chat GTP pode não estar completamente certa.)

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
   ```

Solução 1 (from chat still need to review)

1. Bubble sort
   - Best case scenario: when the input array is already sorted, with a total of $n-1$ comparisons and 0 swaps.
   - Worst case scenario: when the input array is in reverse order, with a total of $(n^2-n)/2$ comparisons and $n*(n-1)/2$ swaps.
2. Insertion sort:
   - Best case scenario: when the input array is already sorted, with a total of $n-1$ comparisons and 0 swaps.
   - Worst case scenario: when the input array is in reverse order, with a total of $(n^2-n)/2$ comparisons and $(n^2-n)/2$ swaps.

In the worst case scenario for both bubble sort and insertion sort, the number of comparisons is $(n^2-n)/2$. In the best case scenario for both algorithms, the number of comparisons is $n-1$, which is the minimum possible number of comparisons required to sort the array.

Devido a esta implementação penso que o melhor caso é capaz de ser um pouco diferente.

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

Solução 2 1. $T(n) = n + T(n-1)$ Utilizando uma árvore de recorrência, podemos visualizar que cada nível da árvore representa uma chamada recursiva de $T(n-1)$ e que o custo de cada nível é $n-k$, onde k é o nível atual. Então, temos:

$$T(n) = n + (n-1) + (n-2) + \cdots + 2 + 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Portanto, o limite superior para o tempo de execução é $O(n^2)$.

2. $T(n) = n + T\left(\frac{n}{2}\right)$ Utilizando novamente uma árvore de recorrência, temos que cada nível da árvore representa uma chamada recursiva de $T(n/2)$ e que o custo de cada nível é $n/2^k$. O número de níveis é dado por $\log_2 n$, pois a cada nível o tamanho do problema é dividido pela metade. Então, temos:

$$T(n) = n + \frac{n}{2} + \frac{n}{2^2} + \cdots + \frac{n}{2^{\log_2 n}} = n \sum_{i=0}^{\log_2 n} \frac{1}{2^i}$$

A série geométrica acima converge para 2 quando n tende ao infinito, então temos:

$$T(n) \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Portanto, o limite superior para o tempo de execução é $O(n)$.

3. $T(n) = k + 2 * T(n-1)$ com k constante

Utilizando uma árvore de recorrência, temos que cada nível da árvore representa uma chamada recursiva de $T(n-1)$ e que o custo de cada nível é 2^k. O número de níveis é dado por n, pois cada chamada recursiva reduz o tamanho do problema em 1. Então, temos: $T(n) = k + 2k + 2^2k + \cdots + 2^{n-1}k = k(2^n - 1)$

Portanto, o limite superior para o tempo de execução é $O(2^n)$.

4. $T(n) = n + 2 * T\left(\frac{n}{2}\right)$

Utilizando uma árvore de recorrência, temos que cada nível da árvore representa duas chamadas recursivas de $T(n/2)$ e que o custo de cada nível é n. O número de níveis é dado por $\log_2 n$, pois a cada nível o tamanho do problema é dividido pela metade. Então, temos:

$$T(n) = n \log_2 n$$

Portanto, o limite superior para o tempo de execução é $O(n \log n)$.

5. Observamos que, em cada nível i, há 2^i subproblemas, cada um com tamanho $\frac{n}{3^i}$. O custo em cada nível é $2^i \cdot k$, pois são realizadas 2^i operações de custo constante k. Portanto, o custo total é dado pela soma dos custos

em cada nível:

$$T(n) = \sum_{i=0}^{\log_3 n} 2^i \cdot k =$$
$$k \sum_{i=0}^{\log_3 n} 2^i =$$
$$k(2^{\log_3^{n+1}} - 1) =$$
$$k(2 \cdot (\log_3 n + 1) - 1) =$$
$$\mathcal{O}(\log_n)$$

Assim, temos que o tempo de execução é limitado superiormente por $O(\log n)$.

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.

Solução 3 Para analisar o tempo de execução do algoritmo de *merge sort*, podemos usar uma relação de recorrência. O tempo de execução do algoritmo depende do tempo de execução das chamadas recursivas e do tempo de execução da função `merge`.

Se $T(N)$ é o tempo de execução de `mergeSort` para um vetor de tamanho N, então temos que:

$$T(N) = 2T\left(\frac{N}{2}\right) + 2N$$

O primeiro termo representa o tempo de execução das duas chamadas recursivas (para as duas metades do vetor) e o segundo termo representa o tempo de execução da função `merge` que é linear no tamanho do vetor.

Podemos resolver essa relação de recorrência usando o método da árvore de recorrência ou o método de substituição. Aqui usaremos o método de substituição. Suponha que $T(N) = aN \log N + bN$ é uma solução para a relação de

recorrência acima, onde a e b são constantes a serem determinadas. Substituindo na equação, temos:

$$\begin{aligned} T(N) &= 2T\left(\frac{N}{2}\right) + 2N \\ &= 2a\left(\frac{N}{2}\right)\log\left(\frac{N}{2}\right) + 2b\frac{N}{2} + 2N \\ &= aN\log N - aN\log 2 + bN + 2N \\ &= aN\log N + (2a+b)N - aN \\ &= aN\log N + bN \end{aligned}$$

Para que essa solução seja válida, precisamos escolher a e b de modo que ela satisfaça as condições iniciais da relação de recorrência. Como $T(1) = 0$, temos:

$$a + b = 0$$

Para determinar a e b, podemos usar a condição de que $T(2) = 4$. Temos:

$$T(2) = 2T(1) + 4 \qquad\qquad = 4a + 2b \ = 4$$

Resolvendo o sistema de equações, obtemos $a = 2$ e $b = -2$. Portanto, a solução para a relação de recorrência é:

$$T(N) = 2N\log N - 2N$$

Assim, o tempo de execução do algoritmo de *merge sort* é $O(N\log N)$.

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.

Solução 4 No algoritmo de *quick sort*, o número de comparações e trocas pode variar dependendo do pivot escolhido em cada chamada da função de particionamento. No pior caso, o pivot escolhido sempre é o maior ou o menor elemento do array, o que leva a uma partição desbalanceada em cada chamada da função. Para calcular o número de comparações, consideramos o pior caso em que o pivot escolhido é sempre o menor ou o maior elemento do array, e a recorrência é dada por: $T(n) = T(0) + T(n-1) + (n-1)$

onde $T(0)$ é uma constante que representa o tempo de processamento da partição com apenas um elemento. O primeiro termo $T(0)$ é necessário porque a recorrência é iniciada a partir de um array de tamanho $n = 1$. O termo $T(n-1)$ é o tempo gasto na chamada recursiva em que todos os elementos exceto o pivô são maiores ou menores que ele, e o termo $(n-1)$ é o número de comparações necessárias para particionar o array. A solução dessa recorrência é $O(n^2)$.

No caso médio, o número de comparações é dado por $O(n \log n)$. O algoritmo de particionamento escolhe um pivô aleatório, o que leva a partições equilibradas em média. A probabilidade de escolher qualquer elemento como pivô é $1/n$, então, em média, cada elemento é escolhido uma vez como pivô. O tempo de processamento do algoritmo de particionamento é $O(n)$, e a recorrência é dada por: $T(n) = 2T(n/2) + cn$

onde c é uma constante que representa o tempo de processamento da função de particionamento e das trocas. A solução dessa recorrência é $O(n \log n)$.

O número de trocas realizadas pelo *quick sort* no pior caso é $O(n^2)$, e no caso médio é $O(n \log n)$.