Assignment 1, CSN-212

Paras Chetal, Enrollment No. 15114049

January 23, 2017

1 Merge Sort

1.1 Algorithm (in python)

```
def merge(alist, l, m, r):
    lleft = m - l + 1
    lright = r- m

    left = [0] * (lleft)
    right = [0] * (lright)

    for i in range(0 , lleft):
        left[i] = alist[l + i]

    for j in range(0 , lright):
        right[j] = alist[m + 1 + j]

    i = 0
    j = 0
    k = l

    while i < lleft and j < lright :
        if left[i] <= right[j]:
            alist[k] = left[i]
            i += 1
        else:
            alist[k] = right[j]
            j += 1
        k += 1

    while i < lleft:
        alist[k] = left[i]
        i += 1
        k += 1

    while j < lright:
        alist[k] = right[j]
        j += 1

def merge_sort(alist, p, r):
    if p < r:
        mid = (r + p - 1)//2
        merge_sort(alist, p, mid)
        merge_sort(alist, mid+1, r)
        merge(alist, p, mid, r)
```

1.2 Correctness

The following loop invariant must hold:

At the start of each iteration of the while loop in the merge funciton, the list alist[l:k-1] contains the k-l smallest elements of left[0:lleft-1] and right[0:lright-1] sorted. Also, left[i] and right[j] will be the smallest elements of the corresponding array which have not yet been put in the alist.

1.2.1 Initialization

At the begining, l = k =¿ alist[l:k-1] is empty. Also, i = j = 1, and left[i] and right[j] are smallest elements which haven't been copied back to alist.

1.2.2 Maintenence

Let's assume that left[i] ¡ right[j] =¿ left[i] is the smallest element which has not yet been copied to the alist. alist[l:k-1] contains the k-l smallest elements, copying left[i] will result in it containing k-l+1 smallest elements. This establishes the loop invariant.

1.2.3 Termination

After the last iteration of our loop, one among the left or right are now completely copied over. Also, alist[l:k-1] contains the sorted elements comprising of k - l smallest elements of left[0:lleft] and right[0:lright].

1.3 Analysis

1.3.1 Divide

We only compute the middle element of subarray, thus D(n) = $\theta(1)$

1.3.2 Conquer

Recursively solving for two sub problems, thus 2T(n/2) contribution to running time

1.3.3 Combine

The merge function takes $\theta(n) time, sincethereareniterationsoftheinnerwhileloopeachtakingconstanttime.$

1.3.4 Best, average and worst case

if n¿1 then
    T(n) = 2T(n/2) + $\theta(n)$
    and if n = 1
    T(n) = $\theta(1)$
    also, T(n) is of the order $\theta(nlogn)$
    which can be proved through the recursion tree.

2 Quick Sort

2.1 Algorithm(python)

```
def partition(alist, p, q):
    m = alist[p]
    left = p+1
    right = q
    flag = False

    while not flag:
        while left <= right and alist[left] <= m:
            left = left + 1
        while alist[right] >= m and right >=left:
            right = right -1
        if right < left:
            flag= True
        else:

            tmp=alist[left]
            alist[left]=alist[right]
            alist[right]=tmp

    tmp=alist[p]
    alist[p]=alist[right]
    alist[right]=tmp
    return right

def quick_sort(alist, p, q):
    if p < q:
        m = partition(alist, p, q)
        quick_sort(alist, p, m-1)
        quick_sort(alist, m+1, q)
    return alist
```

2.2 Correctness

2.2.1 Initialization

At the begining, there is only one element which is therefore sorted.

2.2.2 Maintenance

There are two cases depending on the conditional statement. The values in the one of the subarray are all less than or equal to m. The values in the other are greater than m.

2.2.3 Termination

At termination, we have 3 kinds of values: those less than or equal to m, those greater than m and a single element containing m. The running time of the partition function is $\theta(n)$

2.3 Analysis

2.3.1 Worst-case

When the partitioning results in n-1 elements in one sub array and the other with 0.
T(n) = T(n-1) + $\theta(n)$

2.3.2 Best-case

If the split is even, quicksort runs much faster.
T(n) ¡ = 2T(n/2) + $\theta(n)$
For this, T(n) = O(nlogn)

2.3.3 Balanced

If there is always a fixed ratio in which the split occurs, let's say 3/4 for instance, then
T(n) ¡ = T(3n/4) + T(n/4) + cn
In this case too, complexity - O(nlogn)

2.3.4 Average case

The alternate between good and bad splits is like the running time of only good splits, still O(nlogn), but the hidden constants are larger.