ITCS-6114/8114

HOMEWORK 2

NAME : ANKIT VAITY
STUDENT ID: 801203693

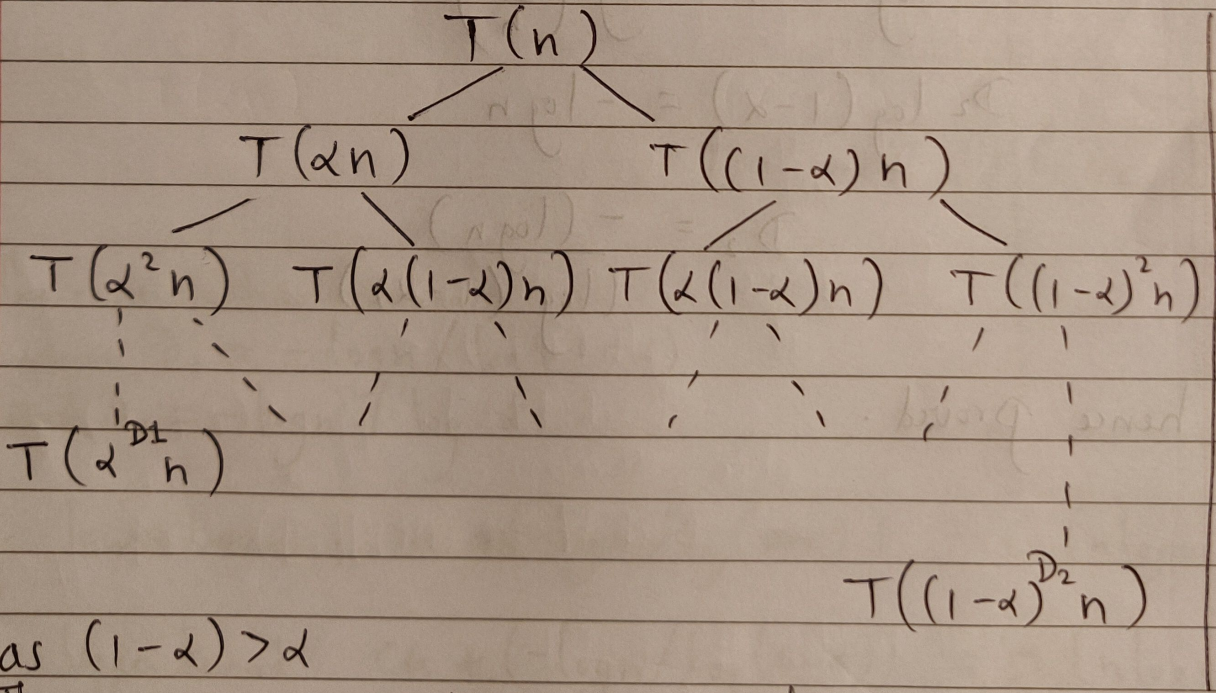
1

(a) Suppose the splits at every level of quicksort are in proportion $1-\alpha$ to α, where $0 < \alpha < 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately $-(\log n)/(\log \alpha)$ and the maximum depth of a leaf in the recursion tree is approximately $-(\log n)/(\log(1-\alpha))$

The recurrence for quick sort is $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$ where $c > 0$ and $0 < \alpha < 1/2$

$\therefore (1-\alpha) > \alpha$

Recursion Tree:



as $(1-\alpha) > \alpha$

The minimum depth of the tree will be the depth of the left subtree (D_1)

$\therefore \alpha^{D_1} n = 1$

$\alpha^{D_1} = \frac{1}{n}$

$$D_1 \log \alpha = \log\left(\frac{1}{n}\right)$$

$$D_1 \log \alpha = -\log n$$

$$D_1 = -\frac{(\log n)}{(\log \alpha)}$$

The maximum depth of the tree will be the depth of the right subtree (D_2)

$$\therefore (1-\alpha)^{D_2} n = 1$$

$$(1-\alpha)^{D_2} = \frac{1}{n}$$

$$D_2 \log(1-\alpha) = \log\left(\frac{1}{n}\right)$$

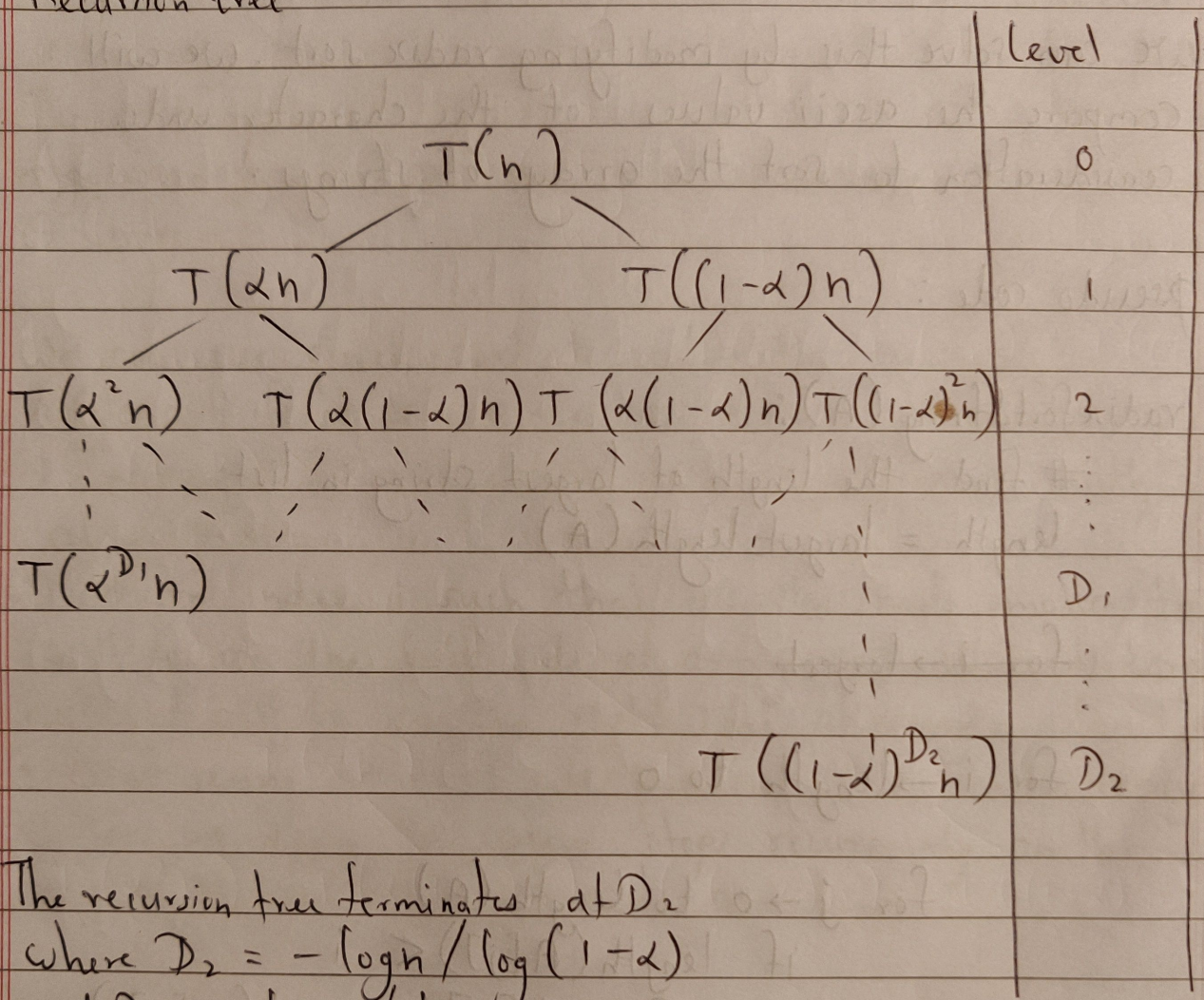
$$D_2 \log(1-\alpha) = -\log n$$

$$D_2 = -\frac{(\log n)}{(\log(1-\alpha))}$$

hence proved.

(b) Use the recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$, where $c > 0$ is a constant.

Recursion tree:



The recursion tree terminates at D_2
where $D_2 = -\log n / \log(1-\alpha)$
and $D_1 = -\log n / \log \alpha$

∴ lower bound : $cn * (-\log n / \log(\alpha)) = O(n \log n)$

Upper bound : $cn * (-\log n / \log(1-\alpha)) = \Omega(n \log n)$

∴ Total cost will be $\Theta(n \log n)$

ANSWER : $\Theta(n \log n)$

2. You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings in the array is n. Show how to sort the strings in O(n) time

we can solve this by modifying radix sort. we will compare the ascii values of the character under consideration to sort the array of strings.

pseudo code:

```
radixSortString(A):
    # find the length of largest string in list
    length = largestLength(A)

    ~~for i -> largest~~

    for i -> length to 0

        for j -> 0 to length(A)
            if length(A[j]) < i
                Bucket[0].append(A[j])
            else  # ascii value of 'a' is 97
                Bucket[ascii(A[j][i]) - 97].append(A[j])

        # flatten the bucket
        A -> flatten(Bucket)

    return A
```

Time complexity: The best case time complexity will be O(n) when the strings will be of smaller length in which case $d \approx 1$, $O(d \times n) \approx O(n)$

3. Nuts and bolts: You are given a collection of n bolts of different widths and n corresponding nuts. You are allowed to try a nut and boalt together, from which you can determine whether the nut is larger than the bolt, smaller than the bolt or matches the bolt exactly. However there is no way to compare two nuts together or two bolts together. The problem is to match each bolt to its nut. Design an algorithm for this problem with an average-case efficiency of $\Theta(n \log n)$

We can use quick sort to solve this problem:
(1) We will first select the last element of bolts array as pivot
(2) Rearrange the array nuts using quick sort with bolts[last] as a pivot.
(3) Return index i such that all the nuts smaller than nuts[i] will be on the left side of array and all nuts larger than nuts[i] will be on the right side of array.
(4) Next using nuts[i] we partition the bolts array.
(5) Now we apply the above steps recursively on the left and right sub array of nuts and bolts.

pseudo code:

```
def partition (arr, low, high, pivot)
    i → low
    for j → 0 to high
        if arr[j] < pivot
            swap (arr[i], arr[j])
            i → i+1
        else if arr[j] == pivot:
            swap (arr[i], arr[high])
            j → j-1
    swap (arr[i], arr[high])
    return i
```

P.T.O.

```
matchNutBolts(nuts, bolts, low, high)
    if low < high

        #partition nuts with last element of bolts
        pivot = partition (nuts, low, high, bolts[high])

        #partition bolts using ~~to~~ nuts[pivot]
        partition (bolts, low, high, nuts[pivot])

        #recursive calls on left and right subarray of nuts and bolts
        matchNutBolts(nuts, bolts, low, pivot - 1)
        matchNutBolts(nuts, bolts, pivot + 1, high)
```

Time complexity : partitioning is done in O(n). We are recursively partitioning left and right subarray of nuts and bolts so the average time complexity will be O(n log n). similar to quick sort.

4. You are given two sorted lists of size m and n. Give an $O(\log m + \log n)$ time algorithm for computing the k^{th} smallest element in the union of the two list.

→ we can solve this with binary search

If we consider two sorted arrays A and B we will compare A[i] with B[j] where $i \to k/2$ and $j \to k/2$.

(1) if ~~A[k/2]~~ $A[i] < B[j]$ then there might be some elements on the right side of A[i] that are smaller than ~~B[k/2]~~ B[j]. So our k^{th} element might be present in the right of A[i] or in the left of B[j]

(2) if $A[i] > B[j]$ then there might be some elements on the right side of B[j] that are smaller than A[i]. So we will look in right side of B[j] and left side of A[i]

(3) We will perform the above steps recursively and if at some point $k \to 1$ then we will return minimum of A[k] and B[k] which will be our base condition.

Pseudo code:

```
findKth ( A , B , size_a , size_b , k )
        # we assume size a is always less than size b
        if (size_a > size_b )
                #swap arrays and call the function
                return findKth (B, A, size_b, size_a, k)

        #if smaller array size becomes zero return k^th element of larger array
        if (size_a == 0 and size_b > 0)
                return B[k-1]

        if k == 1
                return min(A[k-1], B[k-1])
```

P.T.O.

Recursion cases

i → min (size_a, k/2)

j → min (size_b, k/2)

```
if A[i-1] < B[j-1]
    return findKth(A[i-1:], B[:j-1], size_a-i, j, k-i)
else
    return findKth(A[:i-1], B[j-1:], i, size_b-j, k-j)
```

Recursion Cases

i → min (size_a, k/2)

j → min (size_b, k/2)

```
if A[i] < B[j]
    return findKth(A[i+1:], B, size_a-i, j, k-i)
else
    return findKth(A, B[j+1:], i, size_b-j, k-j)
```

Time complexity: In every iteration we are decreasing the size of either one of our arrays. So the time complexity of our algorithm will be $O(\log m + \log n)$