Kevin Sekuj
1/18/2022
CS325: Analysis of Algorithms
Homework 2

1. Solve the recurrence relation using three methods

Recurrence Relation

$T(n) = c\ for\ n = 0$
$T(n) = c\ for\ n = 1$
$T(n) = 2T\left(\frac{n}{2}\right) + c\ for\ even\ n$
$T(n) = 2T\left(\frac{n}{2}\right) + c\ for\ odd\ n$

Substitution Method

$T(n) = 2T\left(\frac{n}{2}\right) + c \rightarrow \boldsymbol{Eq1}$

Substituting $\mathrm{n} \rightarrow \frac{\mathrm{n}}{2}$ in Eq1, we get,

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

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

Substituting $\mathrm{T}\left(\frac{\mathrm{n}}{2}\right)$ in Eq1,

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

$T(n) = 4T\left(\frac{n}{4}\right) + 3c \rightarrow \boldsymbol{Eq2}$

To solve this further, we substitute $\mathrm{T}\left(\frac{\mathrm{n}}{4}\right)$ into Eq2.

To find $\mathrm{T}\left(\frac{\mathrm{n}}{4}\right)$, substitute $\mathrm{n} \rightarrow \left(\frac{\mathrm{n}}{4}\right)$ in Eq1. We get,

$T\left(\frac{n}{4}\right) = 2T\left(\frac{\frac{n}{4}}{2}\right) + c \rightarrow 2T\left(\frac{n}{8}\right) + c$

Substituting $\mathrm{T}\left(\frac{\mathrm{n}}{4}\right)$ in Eq2, we get:

$$T(n) = 4\left(2T\left(\frac{n}{8}\right) + c\right) + 3c$$

$$= 8T\left(\frac{n}{8}\right) + 7c \rightarrow \boldsymbol{Eq3}$$

Thus,

$$T(n) = 2T\left(\frac{n}{2}\right) + c \rightarrow \boldsymbol{Eq1}$$

$$T(n) = 4T\left(\frac{n}{2^2}\right) + 3c \rightarrow \boldsymbol{Eq2}$$

$$T(n) = 8T\left(\frac{n}{2^3}\right) + 7c \rightarrow \boldsymbol{Eq3}$$

Let's say at the $\mathrm{k^{th}}$ equation we reach the base case. By observing the pattern of Eq 1, 2, 3, we can write our $\mathrm{k^{th}}$ equation as,

$$T(n) = 2^k \cdot T\left(\frac{n}{2^k}\right) + c \cdot (2^k - 1) \rightarrow \boldsymbol{k^{th}eq}$$

Since we arrived at the base case T(1) in the $\mathrm{k^{th}}$ equation, we can say

$$T\left(\frac{n}{2^k}\right) = T(1) \rightarrow \boldsymbol{Eq}\,*$$

Substituting this in our modified $\mathrm{k^{th}}$ equation:

$$T(n) = 2^k \cdot T(0) \cdot c = 2^k \cdot c$$

From our (Eq*) we can say $(n/2^k) = 1$

$$= \ n = 2^k$$

$$\log_2 n = \log_2 2^k$$

$$\log_2 n = k \log_2 2$$

$$\log_2 n = k$$

$$k = \log_2 n$$

Substituting this for the value of k in the $\mathrm{k^{th}}$ equation:

$$(2^{\log_2 n})(T\left(\frac{n}{2^{\log_2 n}}\right) + c(2^{\log_2 n} - 1)$$

$= (n)(T\left(\frac{n}{n}\right) + c(n-1)$ by log rule $a^{\log_a b} = b$

$= n \cdot T(1) + c(n-1)$

$= n \cdot c + c(n-1)$

$= nc + nc - c$

$= 2nc - c$

$= c(2n-1)$

Finally,

$\boxed{= T(n) = \Theta(n)}$

Recursion-Tree method

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

Level 0 – T(n): c

Level 1 – T(n/2): c + c

Level 2 – T(n/4): c + c + c + c
and so on until

Level i = T(1) + T(1) . . .

At this point, it becomes apparent that as the recursive tree deepens, nodes increase by a factor of two – 2^0, 2^1, 2^2, etc. Thus, at the i^{th} level, 2^i nodes will exist.

So, at the base case, level i, we have 2^i nodes – T(1) = $T(\frac{n}{2^i})$

$\frac{n}{2^i} = 1 \rightarrow n = 2^i$
$\rightarrow i = \log_2 n$

The total cost of the tree will be cost at each level * number of levels –
$= 2^i \rightarrow 2^{\log_2 n} \cdot c$
$= cn \left(\text{by log rule } a^{\log a^b} = b\right)$
$\boxed{= \Theta(n)}$

Master Method

$T(n) = 2T\left(\frac{n}{2}\right) + c$; here $\mathrm{a} = 2, \mathrm{b} = 2,$ and $\mathrm{f(n)} = c$
$= n^{\log_b a}$
$= n^{\log_2 2} = n^1$

Since f(n) = c, c is a constant, thus n^d <<< $n^{\log_b a}$

Thus $\boxed{T(n) = \Theta(\mathrm{n})}$

2. Solve the recurrence relation using any one method

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

Master Method

$T(n) = 4T\left(\frac{n}{2}\right) + n; here\ a = 4, b = 2, f(n) = n$
$= n^{\log_b a}$
$= n^{\log_2 4} = n^2$

Since n <<< n^2, **T(n) = Θ(n^2)**

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

Master Method

$T(n) = 4T\left(\frac{n}{4}\right) + n; here\ a = 2, b = 4, f(n) = n^2$

$= n^{\log_b a}$
$= n^{\log_4 2} = n^{1/2} = \sqrt{n}$

Since n^2 >>> √n, **T(n) = Θ(n^2)**

3. Implement an algorithm using divide and conquer

a. Pseudocode

```
def kthElement(arr1: list, arr2: list, k: int) -> element:
        if arr1 and arr2 are both empty return -1

        sum length of arr1 and arr2 in variable n

        if k is greater than sum of arr1 and arr2 or k is less than 0 return -1

        Initialize empty array of new array lengths
        create arr1_len, arr2_len to hold len of arr1 and arr2

        create variables to hold indices of arr1, arr2 and new array; arr1_idx,
    arr2_idx, new_idx


         while arr1 < length of arr1 and arr2 < length of arr2
            if element in arr1 index is less than arr2_idx in arr2:
                add the element in arr1 at arr1_idx to the new array
                increment the arr1_idx by 1

            else add element from arr2 at
                increment arr1_idx by 1

            if k equals to new_idx
                return the kth indexed item in the new array


            increment new_idx by 1


        while arr1_idx is less than arr1_len:
            add the element in arr1 at arr1_idx to the new array

            increment the arr1_idx by 1
            increment new_idx by 1

        while arr2_idx < arr2_len:
            add the element in arr2 at arr2_idx to the new array

            increment arr2_idx by 1
            increment new_idx by 1

        return kth element in new array
```

b. Implementation (KthElement.py)