Assignment: Recursion, Recurrence Relations and Divide & Conquer

1. **Solve recurrence relation using three methods**:

   Write recurrence relation of below pseudocode that calculates x^n, and solve the recurrence relation using three methods that we have seen in the explorations.

```
power2(x, n):
   if n==0: #constant time
        return 1     #constant time
   if n==1:  #constant time
        return x  #constant time


   if (n%2)==0: #constant time
        return power2 (x, n//2) * power2 (x, n//2)  # T(n/2) * 2
   else:       #constant time
        return power2 (x, n//2) * power2 (x, n//2)* x # T(n/2) * 2
```

T(0) = time to solve problem of size 0 (base case)
T(1) = time to solve problem of size 1 (intermediate case)
T(n) = time to solve problem of size n (recursive case)

Recurrence relation:
$T(0) = T(1) = c_1$
$T(n) = 2T(n/2) + c$

A. Substitution Method:

Equation #		Substitution Works: (We will be substituting different Values into T(n) = 4T(n/2)
1st	1st: T(n) = 2T(n /2) + c <u>Building 2nd :</u> T(n) = 2T(n /2) + c T(n) = 2[T(n /2)] + c T(n) = 2[2T(n/4) + c] + c T(n) = [4T(n/4) + 2c] + c T(n) = 4T(n/4) + 3c	T(n) = 2T(n /2) + c T(n/2) = 2T((n/2) /2) + c T(n/2) = 2T(n/4) + c T(n/2) = [2T(n/4) + c]
2nd	2nd: T(n) = [4T(n/4) + 2c] + c T(n) = 4T(n/4) + 3c <u>Building 3rd:</u> T(n) = [4T(n/4) + 2c] + c T(n) = [4[T(n/4)] + 2c] + c T(n) = [4[2T(n/8) + c] + 2c] + c T(n) = [8T(n/8) + 4c] + 2c] + c T(n) = 8T(n/8) + 6c + c T(n) = 8T(n/8) + 7c	T(n) = 2T(n /2) + c T(n/4) = 2T((n/4) /2) + c T(n/4) = 2T(n/8) + c T(n/4) = [2T(n/8) + c]

3rd	$T(n) = 8T(n/8) + 6c + c$ $T(n) = 8T(n/8) + 7c$ $T(n) = 8[T(n/8)] + 6c + c$ $T(n) = 8[2T(n/16) + c] + 6c + c$ $T(n) = 16T(n/16) + 8] + 6c + c$ $T(n) = 16T(n/16) + 14c + c$	$T(n) = 2T(n/2) + c$ $T(n/8) = 2T(n/16) + c$
kth	1st: $T(n) = 2T(n/2) + c$ 2nd: $T(n) = 4T(n/4) + 2c + c$ $T(n) = 2^2T(n/2^2) + [2^2c + 2^1c] + c$ 3rd: $T(n) = 8T(n/8) + 6c + c$ $T(n) = 2^3T(n/2^3) + [2^3c + 2^2c + 2^1c] + c$ 4th : $T(n) = 16T(n/16) + 14c + c$ $T(n) = 2^4T(n/2^4) + [2^4c + 2^3c + 2^2c + 2^1c] + c$ Building kth: $T(n) = 2^kT(n/2^k) + [2^{(k)}c + 2^{(k-1)}c + 2^{(k-2)}c\ldots] + c$ $T(n) = 2^kT(n/2^k) + [c(2^k + 2^{k-1)} + 2\wedge k\text{-}2 + \ldots + 1)]$ $T(n) = 2^kT(n/2^k) + [c\ (2^k + 2^{(k-1)} + 2^{(k-2)} + \ldots + 1)]$ $(2^k + 2^{(k-1)} + 2^{(k-2)} + \ldots + 1) = 2(2^k - 1) / (2-1)$ $= 2(2^k - 1) / (1)$ $= 2(2^k - 1)$ Kth: $T(n) = 2^kT(n/2^k) + [c\ [\ 2(2^k - 1)]\ ]$	Base Case: $T(0) = T(1) = c_1$ Finding the value of k: $T(n/2^k) = T(1) = c_1$ $n/2^k = 1$ $n = 2^k$ $\log_2(n) = \log_2(2^k)$ $\log_2(n) = k\log_2 2$ $\log_2(n) = k$ $k = \log_2(n)$ Substituting k & $T(n/2^k)$: $k = \log_2(n)$ $T(n/2^k) = T(1) = c_1$ $T(n) = 2^kT(n/2^k) + [c\ [\ 2(2^k - 1)]\ ]$ $T(n) = 2^k\ T(1) + [c\ [\ 2(2^k - 1)]\ ]$ $T(n) = 2^k\ c_1 + [c\ [\ 2(2^k - 1)]\ ]$ $T(n) = 2^{(\log_2(n))}\ c_1 + [c\ [\ 2(2^{\log_2(n)} - 1)]\ ]$ $T(n) = n^{(\log_2(2))}\ c_1 + [c\ [\ 2(n^{\log_2(2)} - 1)]\ ]$ (properties of logarithms) $T(n) = nc_1 + [c\ [\ 2(n-1)]\ ]$ $T(n) = nc_1 + c[2n-2)]$ $T(n) = nc_1 + 2nc - 2c$, where c_1 & c are just some constants. **$\therefore T(n) \in \Theta(n)$**

B. Recursive Tree Method

T(0) = time to solve problem of size 0 (base case)
T(1) = time to solve problem of size 1 (intermediate case)
T(n) = time to solve problem of size n (recursive case)

Recurrence relation:
$T(0) = T(1) = c_1$
$T(n) = 2T(n/2) + c$

```
                  T(n) = c
                   /    \
            T(n/2) = c   T(n/2) = c
             /     \          /     \
    T(n/4) = c   T(n/4) = c T(n/4) = c T(n/4) = c
      /     \          /     \          /     \
T(n/8)  T(n/8)   T(n/8)   T(n/8)   T(n/8)  T(n/8)
 /  \    /  \     /  \     /  \     /  \     /  \
T(1)T(1) T(1)...........................................T(1)
```

Cost	Nodes
Level 0: c	2^0 Nodes
Level 1: 2c	2^1 Nodes
Level 2: 4c	2^2 Nodes
Level 3: 8c	2^3 Nodes
...	
Level i: 2^i c	2^i Nodes

At level 0: tree expanded from $T(n) \Rightarrow T(n/2^0)$;
At level 1: tree expanded from $T(n/2) \Rightarrow T(n/2^1)$;
At level 2: tree expanded from $T(n/4) \Rightarrow T(n/2^2)$;
At level 3: tree expanded from $T(n/8) \Rightarrow T(n/2^3)$;
At level i: tree expanded from $T(1) \Rightarrow T(n/2^i)$;

$n/2^i = 1$
$n = 2^i$
$\log_2(n) = \log_2(2^i)$
$\log_2(n) = i \log_2(2)$
$\log_2(n) = i$

Total cost:
$T(n) = c * 2^0 + c * 2^1 + c * 2^2 + ... + c * 2^{\log n}$
$T(n) = c(2^0 + 2^1 + 2^2 + ... + 2^{\log n})$
$= c * (2^{(\log n - 1)}) + c * 2^{\log n}$
$= 2c * 2^{\log n} - c$
$= 2c * n - c$
$\therefore \mathbf{T(n) \in \Theta(n)}$

C. Master Method

$T(n) = 2T(n/2) + c$

$T(n) = aT(n/b) + f(n)$

$a = 2, b = 2, f(n) = c$

Compare $n^{\log_b a}$ & $f(n)$

$f(n) = c \; ; n^{\log_2 2}$

$f(n) = c \; ; n$

$f(n) = c <<< n$

This falls under Case 1; f(n) grows asymptotically slower than $n^{\log_b a}$

$\therefore$ **T(n) = Θ(n)**

2. **Solve recurrence relation using any one method**:

   Find the time complexity of the recurrence relations given below using any one of the three methods discussed in the module. Assume base case T(0)=1 or/and T(1) = 1.

Case 1	Case 2	Case 3
If $f(n)$ grows asymptotically slower than $n^{\log_b a}$ i.e $n^d <<< n^{\log_b a}$ Then, the solution for our recurrence relation will be T(n) = Θ($n^{\log_b a}$) The intuition here is: Since f(n) grows slower than $n^{\log_b a}$, the dominating order of growth will be that of $n^{\log_b a}$.	If $f(n)$ and $n^{\log_b a}$ have a similar order of growth. i.e n^d = Θ($n^{\log_b a}$) Then, the solution for our recurrence relation will be T(n) = $\theta\left(n^d \log n\right)$	If $f(n)$ grows asymptotically faster than $n^{\log_b a}$ i.e $n^d >>> n^{\log_b a}$ Then, the solution for our recurrence relation will be T(n) = $\theta\left(n^d\right)$ The intuition here is: The time complexity of f(n) dominates in overall time complexity of recurrence relation.

a) $T(n) = 4T\,(\,n/2\,) + n$

$T(n) = aT(n/b) + f(n)$

$a = 4, b = 2, f(n) = n$

Compare $n^{\log_b a}$ & $f(n)$

$n^{\log_2 4}$; $f(n) = n$

$f(n) = n; n^{\log_2 4}$

$f(n) = n; n^2$

$f(n) = n <<< n^2$

$n^2 >>> f(n) = n$

This falls under Case 1; f(n) grows asymptotically slower than $n^{\log_b a}$

$\therefore$ **T(n) = Θ**(n^2)

b) $T(n) = 2T\ (\ n/4\ ) + n^2$

$T(n) = aT(n/b) + f(n)$

$a = 2,\ b = 4,\ f(n) = n^2$

Compare $n^{\log_b a}$ & $f(n)$

$n^{\log_4 2}$

$f(n) = n^2\ ;\ n^{1/2} = n^{\log_4 2}$

$f(n) = n^2 >>> n^{1/2}$

This falls under Case 3; f(n) grows asymptotically faster than $n^{\log_b a}$

$\therefore$ **$T(n) = \Theta(n^2)$**

3. **Implement an algorithm using divide and conquer technique**: Given two sorted arrays of size m and n respectively, find the element that would be at the k^{th} position in combined sorted array.

   a. Write a pseudocode/describe your strategy for a function kthelement(Arr1, Arr2, k) that uses the concepts mentioned in the divide and conquer technique. The function would take two sorted arrays Arr1, Arr2 and position k as input and returns the element at the k^{th} position in the combined sorted array.
   b. Implement the function kthElement(Arr1, Arr2, k) that was written in part a. Name your file **KthElement.py**

   Examples:

      Arr1 = [1,2,3,5,6] ; Arr2= [3,4,5,6,7] ; k= 5

      Returns: 4

      Explanation: 5^{th} element in the combined sorted array [1,2,3,3,4,5,5,6,6,7] is 4

Pseudocode:

#Within our KthElement function call the merge_sort function (indirect recursion) and pass our arr as the argument (merge_sort implements Divide & Conquer technique to sort arr)

```
merge_sort(arr):
        if len(arr) > 1 #base case – (if we can't enter this if statement then we successfully
        divided the array all the way down into subarrays of 1 element or the passed array is of size 1 and
        there is nothing to divide)
#split the passed array into 2 subarray (Divide part of the Divide &
Conquer technique)
        mid = len(arr) // 2
        right_half = arr[:mid] #start to mid
        left_half = arr[mid:]    #mid to end
        merge_sort(left_half) #direct recursion- split left subarray
        merge_sort(right_half)#direct recursion- split right subarray
        i = j = k = 0 #counters used to keep track of indices of left &
        right half of array
```

#loop is used to merge the two sorted halves of the original array into one sorted array.

#If the i-th element of left_half is greater than the j-th element of right_half, then the k-th element of arr is set to the j-th element of right_half, and j and k are incremented by 1.

```
        while i < len(left_half) and j < len(right_half):
```

#In each iteration of the loop, the function compares the i-th element of left_half w/ the j-th element of right_half.

#If the i-th element of left_half is less than or equal to the j-th element of right_half, then the k-th element of arr is set to the i-th element of left_half, and i and k are incremented by 1.

```
            if left_half[i] <= right_half[j]:
                arr[k] = left_half[i]
                i += 1
```

#If the i-th element of left_half is greater than the j-th element of right_half, then the k-th element of arr is set to the j-th element of right_half, and j and k are incremented by 1.

```
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1
```

After the while loop, if there are any remaining elements in left_half or right_half, the function enters two more while loops to add these remaining elements to the end of arr.

```
            while i < len(left_half):
                arr[k] = left_half[i]
                i += 1
                k += 1
            while j < len(right_half):
                arr[k] = right_half[j]
                j += 1
                k += 1
        return arr
```

#in our KthElement function we are given 2 sorted arrays; 1 of size n & the other 1 of size m, & index k as parameters

```
KthElement(Arr1, Arr2, k):
        Arr1 = [0…m-1] #size m
        Arr2 = [0…n-1]  #size n
```

#combine the array into one

```
arr = Arr1 + Arr2
```

```
#sort the array by calling function merge_sort
merge_sort (arr)
        return arr[k]
```

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]
        merge_sort(left_half)
        merge_sort(right_half)
        i = j = k = 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i] <= right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
```

```
                arr[k] = right_half[j]
                j += 1
            k += 1
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1
        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
    return arr

def kthElement(Arr1, Arr2, k):
    arr = Arr1 + Arr2 # combine the arrays
    arr = merge_sort(arr) # sort the combined array
    return arr[k-1]
```