Assignment: Recursion, Recurrence Relations and Divide & Conquer

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

Write recurrence relation of below pseudocode that calculates 𝑥𝑛, 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) = c1

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  Building 2nd :  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  Building 3rd:  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) = 22T(n/22) + [22c + 21c ] + c  3rd: T(**n**) = 8T(n/8) + 6c + c  T(n) = 23T(n/23) + [23c +22c + 21c] + c  4th : T(**n**) = 16T(n/16) + 14c + c  T(n) = 24T(n/24) + [24c + 23c + 22c + 21c ] + c  Building kth:  T(n) = 2kT(n/2k) + [2(k) c + 2(k-1) c + 2(k-2) c…] + c  T(n) = 2kT(n/2k) + [c(2k + 2 k-1)  + 2^ k-2 + ... + 1)]  T(n) = 2kT(n/2k) + [c (2k + 2 (k-1)  + 2 (k-2) + ... + 1)]  (2k + 2 (k-1)  + 2 (k-2) + ... + 1) = 2(2k -1) / (2-1)  = 2(2k -1) / (1)  = 2(2k -1)  Kth: T(n) = 2kT(n/2k) + [c [ 2(2k -1)] ] | Base Case: T(0) = T(1) = c1  Finding the value of k:  T(n/2k) = T(1) = c1  n/2k = 1  n = 2k  log2(n)= log2(2k)  log2(n)= klog22  log2(n)= k  k = log2(n)  Substituting k & T(n/2k):  k = log2(n)  T(n/2k) = T(1) = c1  T(n) = 2kT(n/2k) + [c [ 2(2k -1)] ]  T(n) = 2k T(1) + [c [ 2(2k -1)] ]  T(n) = 2k c1 + [c [ 2(2k -1)] ]  T(n) = 2(log2(n)) c1 + [c [ 2(2log2(n) -1)] ]  T(n) = n(log2(2)) c1 + [c [ 2(nlog2(2) -1)] ] (properties of logarithms)  T(n) = nc1 + [c [ 2(n-1)] ]  T(n) = nc1 + c[2n-2)]  T(n) = nc1 + 2nc - 2c, where c1  & c are just some constants.  **∴T(n) ∈Θ(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) = c1

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

Nodes

20 Nodes

21 Nodes

22 Nodes

23 Nodes

2i Nodes

Cost

Level 0: c

Level 1: 2c

Level 2: 4c

Level 3: 8c

...

Level i: 2^i 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)

At level 0: tree expanded from T(n) => T(n/20);

At level 1: tree expanded from T(n/2) => T(n/21);

At level 2: tree expanded from T(n/4) => T(n/22);

At level 3: tree expanded from T(n/8) => T(n/23);

At level i: tree expanded from T(1) => T(n/2i);

n/2i= 1

n = 2i

log2(n) = log2(2i)

log2(n) = i log2(2)

log2(n) = i

Total cost:

T(n) = c \* 20+ c \* 21 + c \* 22 + ... + c \* 2log n

T(n) = c(20+ 21 + 22 + ... + 2log n)

= c \*(2(log n - 1)) + c\*2log n

= 2c \* 2log n - c

= 2c \* n - c

**∴T(n) ∈Θ(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 nlogba­­  & f(n)

f(n) = c ; nlog22­­

f(n) = c ; n

f(n) = c <<< n

This falls under Case 1; f(n) grows asymptotically slower than nlogba­­

**∴T(n) = Θ(n)**

1. **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.

Graphical user interface, text

Description automatically generated with medium confidence

a) 𝑇(𝑛) = 4𝑇 ( 𝑛/2 )+ 𝑛

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

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

Compare nlogba­­  & f(n)

nlog24 ; f(n) = n

f(n) = n; nlog24

f(n) = n; n2

f(n) = n <<< n2

n2 >>> f(n) = n

This falls under Case 1; f(n) grows asymptotically slower than nlogba­­

**∴T(n) = Θ(**n2**)**

b) 𝑇(𝑛) = 2𝑇 ( 𝑛/4 ) + 𝑛2

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

a = 2, b = 4, f(n) = n2

Compare nlogba­­  & f(n)

nlog42

f(n) = n2 ; n1/2 = nlog42

f(n) = n2 >>> n1/2

This falls under Case 3; f(n) grows asymptotically faster than nlogba­­

**∴T(n) = Θ(**n2**)**

1. **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 kth position in combined sorted array.
   1. 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 kth position in the combined sorted array.
   2. 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: 5th 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]