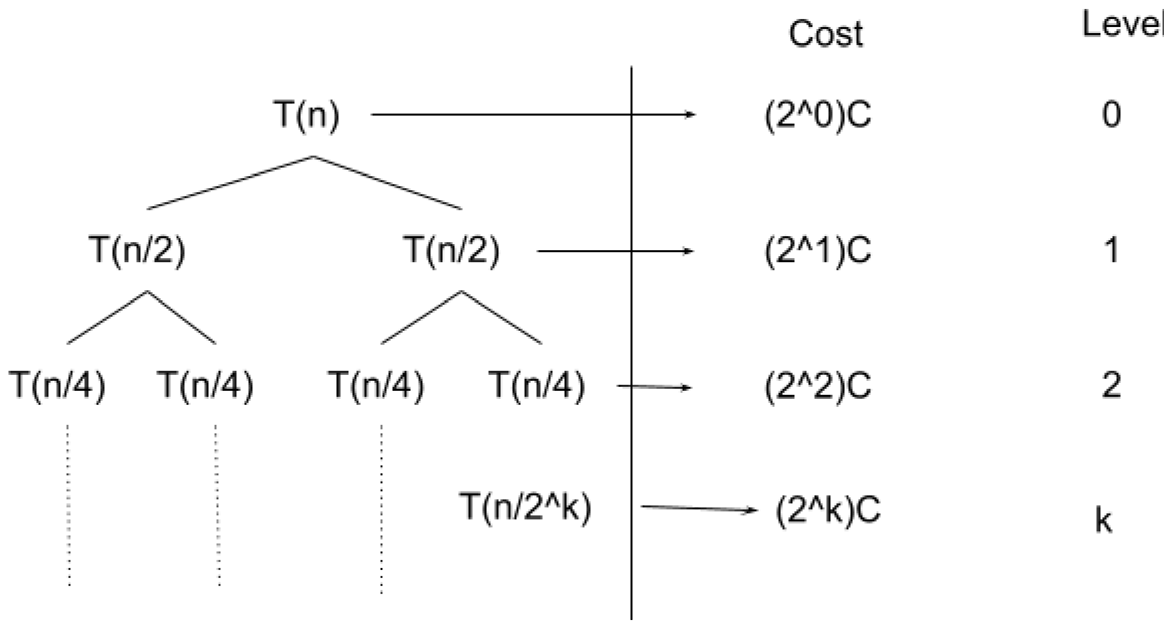
Sami Noor Syed
ONID: 934330738
CS 325 Analysis of Algorithms
Assignment #2
Due date: 7/5/22

1) **Solve the recurrence relation using three methods**
   a) $T(n) = 2(T(n/2)) + c_1;\ T(0) = c_2;\ T(1) = c_3$
   b) Substitution Method:
      i) exploration:
         (1) $T(n) = 2(T(n/2)) + c_1$
         (2) $T(n) = 2^2(T(n/2^2)) + 3c_1$ [keeping in mind that c is multiplied by the constant outside of T(n/2^k)]
         (3) $T(n) = 2^3(T(n/2^3)) + 7c_1$
         (4) $T(n) = 2^k(T(n/2^k)) + (2^k - 1)c_1$ [pattern]
      ii) $T(n/2^k)$ = T(1)
         (1) $n/2^k = 1$
         (2) $n = 2^k$
         (3) K*log(2) = log(n) [taking the log of both sides]
         (4) K = $\log_2(n)$
      iii) $T(n) = 2^{log(n)}(T(n/2^{log(n)})) + (2^{log(n)} - 1)c_1$
         (1) $T(n) = n(T(1)) + c_1(n) - c_1$ [using properties of logs]
         (2) $T(n) = c_3(n) + c_1(n) - c_1$ [simplify]
         (3) $T(n) = (c_3 + c_1)(n) - c_1$
      iv) $T(n) \in \Theta(n)$

c) Tree Method



i) $T(n/2^{k})$ = T(1)

(1) $n/2^{k}$ = 1

(2) $n = 2^{k}$

(3) K*log(2) = log(n) [taking the log of both sides]

(4) K = $\log_2(n)$ [this represents the number of levels in the tree]

ii) $2^{k} * c = 2^{log(n)} * c = n * c\ [using\ properties\ of\ logs]$

iii) If n is the number of leaves on the last level, the whole tree has (2n-1) nodes therefore the time complexity is C(2n-1) since there is a cost of c for each node in the tree.

iv) C(2n-1) = Θ(n)

d) Master method:

i) $2(T(n/2)) + c_1$

ii) a =2, b = 2, f(n) = c_1

iii) The master method can be applied because a >=1 and b>1 and f(n) is a polynomial function

iv) $n^{log(2)} = n ; c_1$

v) We see that $n >>> c_1$ for large values of n and therefor by the master method, $T(n) \in \Theta(n)$

2) Solve recurrence relations using any one method

a) A T(n) = 4T(n/2)+n

i) Master method:

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

iii) Q = $n^{log2(4)} = n^2$

iv) Q/f(n) = ∞, therefore $T(n) \in \Theta(n^2)$

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

i) Master Method:

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

iii) Q = $n^{log4(2)}$ = $n^{1/2}$

iv) Q/f(n) = 0, $T(n) \in \Theta(n^2)$

3) Implement an algorithm using the divide and conquer technique

Def kthElement(arr1,arr2,k):

“””

Function that finds the kth element of two sorted arrays when they are combined

This solution is based on the following observations:

1) **Loop invariant:** the kth element must be contained within array1[0: point1] and array2[0: point2] if the following are true
   a) point1+point2 = k
   b) array1[point1] !> array2[point2+1]
   c) array2[point2] !> array1[point2+1].
2) **Divide:** We can cut the arrays in half while adjusting k until array1[point1] < array2[point2+1] and array2[point2] < array1[point1+1].
3) **Conquer:** Once we know that our k selection of points contains only the values <= the kth element according to the paradigm above, we can decide which element is our kth element based on the circumstances the arrays in question

“”””

n = len(arr1)

m = len(arr2)

conquer: solve the subproblems with all of the base cases

Base cases:

If n ==1 or m == 1: # at least one of the arrays has a length of 1

Make arr 1 the array with only one element and n = 1 so that our later conditionals are consistent for all comparisons

```
    If k == 1: #if k == 1, the smaller of the two element will be the kth element. the larger will
    be the k+1 element
        return minimum(arr1[0], arr2[0])
    If k == 2:
        return maximum(arr[0], arr2[0])
    else: # given that k == 2, the larger value of the arrays will be equal to the kth element
        If arr2[k-1] < arr[0]:
            Return arr2[k-1] #if k is greater than 2, then arr[k-1] will be the kth
    element arr1[0] will be the k+1 element
        else:
            Return max(arr[0],arr2[k-2]) #the kth is the larger of the two
        #if arr1[0] is smaller, then the arr2[k-1] is the k+1 element and the kth element will
        be the larger of arr1[0] and arr2[k-2]
```

#divide/ break down the arrays into smaller more manageable sub parts

middle1 = (n-1)//2

middle2 = (m-1)//2

If k is larger than the number of elements contained in the first halves:

we don’t need to consider elements smaller than the largest of the other array, adjust k and arrays accordingly

```
        If arr1[middle2]> arr2[middle2]:
                # reduce the size of the second array by half and adjust the value
                # of k to represent that we removed values that may have been
                # included within the kth interval
                Return kthElement(arr1, arr2[middle2+1:]], k - middle2 -1)
        else:
                Return Def kthElement(arr1[middle1+1:], arr2, k - middle1 - 1)
else: # k is smaller or equal to the number elements contained in the first halves.
        If arr1[middle2]> arr2[middle2]:
                # reduce the size of the first array by half k does not need to be
                adjusted
                Return kthElement(arr1[:middle1+1], arr2, k)
        else:
                Return Def kthElement(arr1, arr2[:middle1+1], k)
```