## Design and Analysis of Algorithms

## CS375 Fall 2017

## Theory Assignment 2

Release Date: 02/20/2019

Due: 03/01/2019 at start of class

Remember to include the following statement at the start of your answers with a signature by the side.

“I have done this assignment completely on my own. I have not copied it, nor have I given my solution to anyone else. I understand that if I am involved in plagiarism or cheating I will have to sign an official form that I have cheated and that this form will be stored in my official university record. I also understand that I will receive a grade of **0** for the involved assignment for my first offense and that I will receive a grade of **“F” for the course** for any additional offense.”

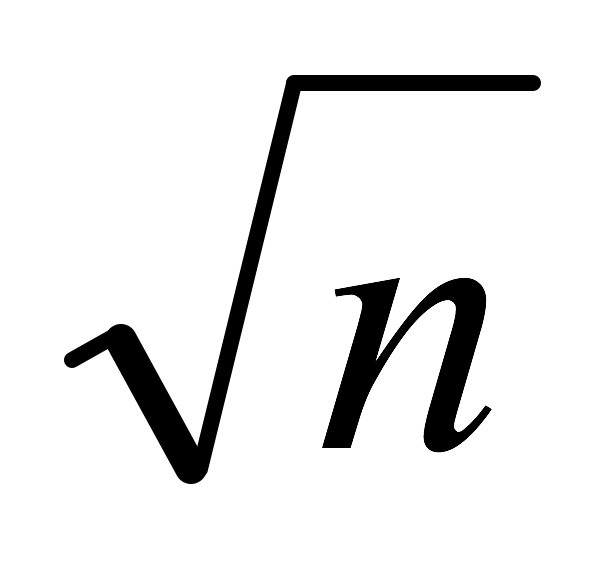
All solutions of theory assignments must be typed (no handwritten solutions) and submitted in hard copy. Advance electronic submission to the TA is acceptable if the student is expected to miss the class on the due date.

1. (24 points) Use the Master theorem to solve the following recurrences (show necessary steps to justify your answer).

a) T(n)=3T(n/4)+n.

a =3, b = 4, f(n) = n, c = 1.

We now compare and c to see which case it will be. We get that so this means its case 3 which means we get Θ(n)

b) T(n)=2T(n/4)+ lgn.

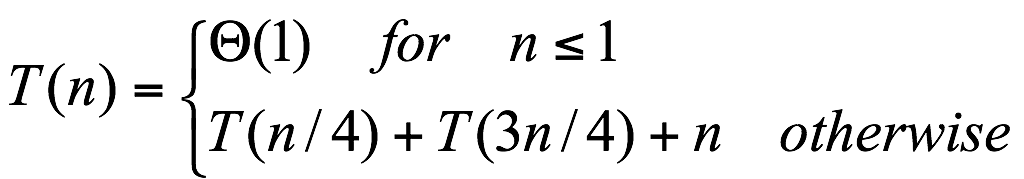
a = 2, b = 4, f(n) = , c = 1/2

we compare to our c value to see which case it will be. When we comapre values we get that they are both equal ! which means case 2  of the master theorem. Our result is Θ(n)

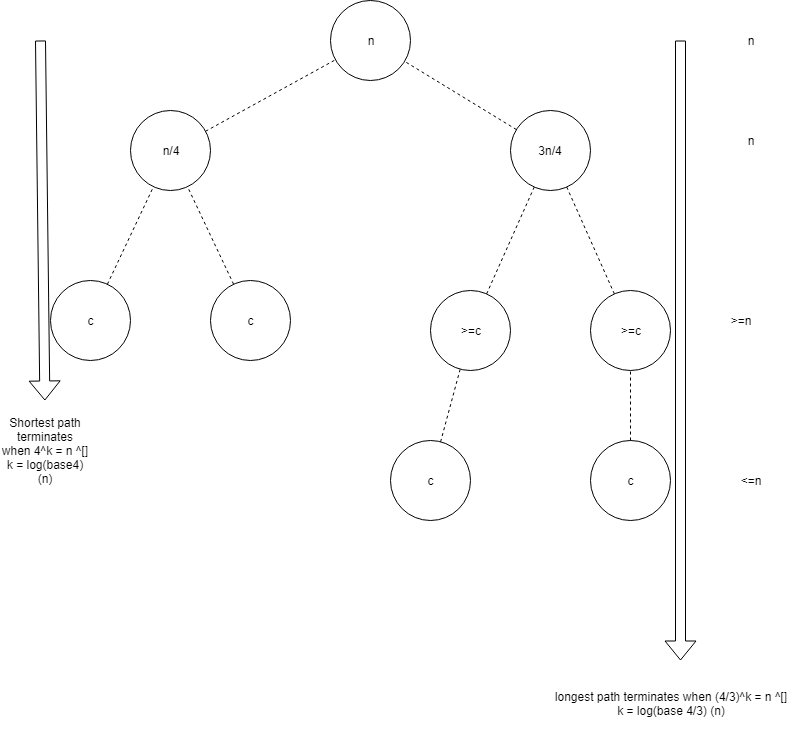
c) T(n)=5T(n/2)+n2.

a = 5, b = 2, f(n) = , c = 2

we compare to our c value to see which case it will be. When comapred we get that > 2 (our c value) meaning case 1 of Θ()

2. (20 points) Solve the recurrence **

using the recursion tree method. Draw the recursion tree and show the aggregate instruction counts for the following levels (0th, 1st, and last levels), and derive the Θ growth class for *T(n)* with justifications*.*



Θ(nlogn)

3. (10 points) Use the substitution method to prove that *T*(*n*) = *T*(*n* – 1) + *n*  *O*(*n*2). You can assume *T*(1) = 1.

Base case n = 1. Assume n>1. Suppose T(n-1) is O((n-1)^2) = O(n^2)

Show that T(n) is also in O(n^2)

T(n) = T(n-1) + n

< c ( n-1) ^2 + n, assyme c>1 wlog

< c n^2 – 2cn + c + n

< c n^2 – (2c-1)n + c

<c n^2

For n > 1, c > 1

Notice that when c>1, 2c-1 > c…so you get:

<c n^2 – (2c – 1)n + c

<c n^2 – (c)n +c

When n > 1, -(c)n+c = (1-n) c < 0…so you get

< c n^2 – (c)n +c

< c n^2

Since there is a constant c s.t T(n) < c n^2, T(n) is in O(n^2)

4. (21 points) Assume that you are given an array of *n* (*n* >=1) elements sorted in non-descending order. Design a *ternary* search function that searches the array for a given element *x* by applying the divide and conquer strategy. Hint: extend the binary search example introduced in the class - divide the array into three subarrarys where each subarray has *n*/3 (or almost *n*/3) elements).

Your answer should contain four parts:

1. Briefly describe the divide, conquer, and combine steps;
   1. Divide: Divide the array into three as of equal as it can be parts, by taking mid1, and mid2 and consider the subarrays to the left and up to midpoint 1, to the right and up to midpoint 2 , between midpoint 1 and midpoint 2. The answer will lie somewhere within those arrays if present.
   2. Conquer: By finding a max sub array of original problem and going to the “correct” direction depending on the midpoints that it finds.
   3. Combine: By finding the the element out of the given subarray and returning that element
2. Clearly define the recursive function ternarySearch(*x*, *A*, *left*, *right*), where *x* is the element to search for in the array *A* with starting index *left* and ending index *right*;
   1. The way I tackle this problem is by making a ternarySearch problem, where ternarySearch( x (the input im looking for), A(the array), left(the starting index), right(the ending index))

TernarySearch(x, A, left, right)

{

If(right >= 1){  
 //finding midpoints

Mid1 = left + (right -1eft) / 3 //int math

Mid2 = right – (right – 1eft)/3 //int math

If (A[mid1] == x) return whats in array[mid1] //we found it!

If(A[mid2] == x) return whats in array[mid2] //we found it!

//recursive (narrowing the subarrays)

If (x < A[mid1] ) return TernarySearch(x, A, left, mid1 – left) //lies between left and mid1

Else If (x > A[mid2]) return TernarySearch(x, A, mid2+left, right) //lies between mid2 and right

Else return TernarySearch(x, A, mid1 + left, mid2 – left)

}

Return -1 //or some sort of error if it was not found

}

1. Clearly define the recursive time complexity function T(*n*) for ternarySearch(*x*, *A*, *left*, *right*);

T(n) = T(n/3) + 4 since it divides the problems by 3 and makes at most 4 comparisonsT(1) = 1 when its< 1.

1. Solve the recursive T(*n*) by the master theorem

A = 1, b = 1, f(n) = 4…meaning c = 0

Now we take log(base b) (a) and compare it to c. we get log(base 3) (1) =0. Since 0 =0 (our c value) then this means its case 2 where Θ(n^c log^(k+1)n) = Θ(n^0log(base3)n) = Θ(log(base3)(n)) which essentially boils down to Θ(log(n))

5. (25 points) The *median* of a list of numbers is its 50th percentile: half the numbers are bigger than it, and half are smaller. For instance, the median of [45, 1, 10, 30, 25] is 25, since this is the middle element when the numbers are arranged in order. If the list has even length, there are two choices for what the middle element could be, in which case we pick the smaller of the two. For example, the median of [45, 1, 10, 30] is 10.

Computing the median of *n* numbers is easy: just sort them. The drawback of this approach is that this takes O(*n* log *n*) time, whereas we would ideally like something linear. We have reason to be hopeful, because sorting is doing far more work than what we really need - we just want the middle element and don't care about the relative ordering of the rest of them. Can we develop a recursive solution for deciding the median of a list of numbers?

When looking for a recursive solution, it is paradoxically often easier to work with a *more general* version of the problem. In our case, the generalization we will consider is *selection*.

SELECTION

*Input:* A list of numbers *S*; an integer *k*

*Output:* The *k*th smallest element of *S*

For instance, if *k* = 1, the minimum of *S* is sought, whereas if *k* = ceiling(|*S*|/2), it is the median.

Develop a divide-and-conquer approach to selection (and hence a solution for the finding median problem). Hint: for any number *v*, imagine splitting list *S* into three categories: elements smaller than *v*, those equal to *v* (there might be duplicates), and those greater than *v*.

In your answer, show the following:

1. Briefly describe the divide, conquer, and combine steps;
   1. Divide:will divide array into values that are less than some pivot point (left) and values that are greater than pivot point(right)
   2. Conquer: by sorting and actively finding the median in a recursive but efficient manner.
   3. Combine: a sorted array at the end and median will be middle value of array
2. Clearly define the recursive function for selection(*S*, *k*); (note: this is not the function for the time complexity of the selection function.)

Selection(S, K)

{

If |S| = 1, output x1 , where s = {x1, x2……,x} of array/ subarray

Else {

SLeft = {x, element in S, xi < x1}

Sright = {x, element in S, xi > x1

}

If SL| = k-1 Return x1; //then pivot is median

If |SL| > k-1 find(SL, k)//median will be in SL..recurse

If |SL| < k-1 find(SR, k-|S| - 1) //median in SR…recurse

}

1. Analyze the best case and worst case time complexity of this approach given input size *n*.

Since we compared n-1 comparisons. We have T(n) <= (n-1)

T(n) <= (n -1) + T(max|SL|, |Sr|)

T(n) <= (n-1) + ( n-2) + T(n-2)…..

If we keep going we see that it a pettern keeps repeating with the recursive calls…at some point towards all of the inputs we can conclude that worst case is slow…. Θ(n^2)

Now this all falls under picking a good or bad pivot. Previously we assumes we picked a “bad pivot”. Now if we pick a “good pivot” so something that lies at least ¾ n comparisons vs n-1 comparisons. If we end up randomly picking a good pivot then our best case scenario is o(n) since less comparisons will have to be made.

6. **Bonus Question** (20 points):

We know that the master theorem does not apply to the recursive function *T*(*n*) = 2*T*(*n*/2) + *n* / lg *n*.

Use the recursion tree method to solve this recursion. Draw the recursion tree and show the aggregate instruction counts for the following levels (0th, 1st, and last levels), and derive the Θ growth class for *T(n)* with justifications*.*

When you start unrolling the recursion we get T(n) = 2T ( n/2) + n/(logn) = ….= 2^(k)T(n/(2^k)) + n 1/(log(n/2)) = 2^kT(n/2^k) + n 1/(logn-i)

Assuming base case is T(1) = 1 this means that n = 2^k.

2^kT(n/2^k) + n 1/(logn-i) = n +n 1/(k-i)

Second sum can be approximated as log (k)….k = log(n).

O(nloglogn)