COE428 Lecture Notes Week 3 (January 23, 2017)

Table of Contents

Topics (from course outline)

The following table shows the topics for this course week by week.

The topics in **bold** is for **this** week.

The topics in grey have been covered.

Other topics are for the future....

Week	Date	Topics
1	Jan 9	Introduction. Course overview. Intro to algorithms.
2	Jan 16	**Analyzing and designing algorithms. Recursion.**
3	Jan 23	**Complexity analysis.**
4	Jan 30	Recurrence equations. Data Structures.
5	Feb 6	Stacks and Queues.
6	Feb 13	Heapsort. Hashing.
	Feb 20	*Study week.*
7	Feb 27	Trees and Priority Queues.
8	March 6	Binary Search Trees (BST).
9	March13	Balanced BSTs (including Red-Black Trees)
10	March 20	Graphs.
11	March 27	Elementary graph algorithms.
12	April 3	Elementary graph algorithms. (continued)
13	April 20	Review

Review

- The time to perform recursive algorithms is often expressed as a *recurrence*.
- Example: Merge Sort: $T(n) = 2T(n/2) + n$ (time to merge sort *n* items = time to sort each half + time to merge two sorted lists where merging is a linear algorithm.)
- Closed-form exact solution to $T(n) = 2T(n/2) + n$ is $T(n) = n \lg n$ which can be proven by *mathematical induction.*
- The algorithms so far:

Name	Description	Complexity
Selection Sort	Sort by selecting minimum (over and over)	quadratic
Merge Sort	Sort by splitting in 2, sorting each half, then merging	Linear logarithmic
Binary search	Search an ordered list	logarithmic
Euclid's algorithm	Greatest common divisor between "big" and "small"	logarithmic
Towers of Hanoi	Move disks from one tower to another respecting rules	Exponential (2^n)

Answers to last week's questions

1. An algorithm with complexity $\Theta(\sqrt{n})$ takes 6 ms to solve a problem of size 1600. Estimate the time to solve a problem of size 10,000.

   Answer: $\frac{\sqrt{10000}}{\sqrt{1600}} = 100/40 = 2.5$**.** So it takes $2.5 \times 6 = 15$ ms to solve a problem of size 10,000

2. Draw a recursion tree for $T(n) = 2T(n/2) + k_1 n + k_2$. Guess the exact solution and prove it by mathematical induction.

   Answer: Discussed in class (see below)

3. Draw a recursion tree for $T(n) = 2T(n/4) + T(n/2) + n$. Guess the solution. Try to prove it.

Answer: We will look at the recursion tree below. A reasonable guess would be $T(n) = n \lg n$. Unfortunately, it is wrong! The table below calculates a few values bottom up assuming that $T(1) = 0$.

n	$T(n) = 2T(n/4) + T(n/2) + n$	$n \lg n$
1	0	0
2	0 + 0 + 2 = 2	2
4	0 + 2 + 4 = 6	8
8	2*2 + 8 + 8 = 20	24

Preamble: Week 3 and 4 lectures

- The topics for this week and the next are mainly mathematical.
- The techniques used will be used to analyze algorithms studied in the rest of the course.

Some math

- You are expected to know certain mathematical facts. (Usually, no formula sheet or calculators are allowed in tests/exams.)
- Some of these basic formulas:

Arithmetic series

- $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = n(n-1)/2$

Geometric series

- $\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n = (a^{n+1} - 1)/(a-1)$

logarithms

- $\log ab = \log a + \log b$
- $\log a^b = b \log a$
- $\log_a x = \frac{\log_b x}{\log_b a}$
- $x = b^{\log_b x}$

Harmonic series

- *Harmonic numbers* are defined as $H_n = \sum_{i=1}^{n} \frac{1}{n} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$
- The Harmonic series is divergent, but it diverges slowly.
- For large n, $H_n \approx \ln n < \ln n + 1$
- Hence any multiple of H_n is a *logarithmic.*

Notes on exponential functions

- Any function $n^{1+\epsilon}$ (where $\epsilon > 0$) ultimately grows faster than ***any*** polynomial.

- For example, $n^{1.00001}$ grows faster than $n^{1000000}$. (This is easy to prove using L'Hopital's rule.)
- Consider a^n compared to b^n where $a > b$. Then a^n grows "infinitely" faster than b^n.

Asymptotic (Big-O, Big-Θ, Big-Ω) notation

Big-O (upper bound)

- We say that $f(n) = O(g(n))$ if there exist constants c and n_0 such that:

  $f(n) \le cg(n)$ for all $n > n_0$

Big-Omega (lower bound)

- We say that $f(n) = \Omega(g(n))$ if there exist constants c and n_0 such that:

  $f(n) \ge cg(n)$ for all $n > n_0$

Big-Theta (tight bound)

- We say that $f(n) = \Theta(g(n))$ if there exist constants c_1, c_2 and n_0 such that:

  $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n > n_0$
- Equivalently, $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

How to "guess" a recurrence solution

Finding a guess by "unfolding" (aka "substitution")

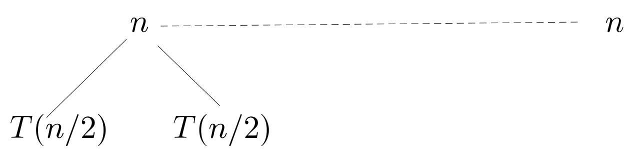
- Previously we calculated *T(n)* from the bottom up.
- We can "unfold" it from the "top down" as follows:

$$\begin{aligned} T(n) &= 2T(n/2) + n = 2(2T(n/4) + n/2) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n \\ &= 16T(n/16) + 4n \end{aligned}$$

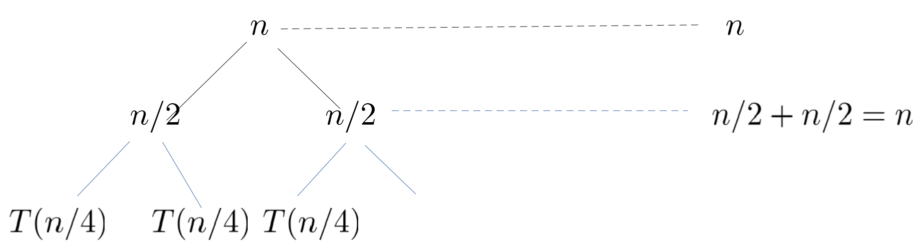
  etc...
- If we assume that *n* is a power of 2, we would eventually obtain: $T(n) = nT(n/n) + n \lg n$
- Since we have assumed T(1) = 0, this implies
  $T(n) = nT(n/n) + n \lg n = nT(1) + n \lg n = n \times 0 + n \lg n = n \lg n$

Finding a guess by drawing a recursion-tree

- We start by representing $T(n) = 2T(n/2) + n = T(n/2) + T(n/2) + n$ as a graph where we put the non-recursive part (*n* in this case) on the top row and put each recursive part on a row below.



- We now expand the tree diagram downwards:



Different ways to draw recursion-trees

- The textbook (CLRS) starts with a diagram with a single node: T(n).
- It then expands each node isolating the non-recursive part and adding child nodes for each recursive part.
- For example, to draw the recursion-tree for $T(n) = 2T(n) + n$, start with:

*T(n) = 2T(n/2) + k1*n + k2 recurrence tree and analysis*

Guess: $T(n) = k_1 n \lg n + k_2(1 + 2 + 4 + \ldots + n) = k_1 n \lg n + k_2(2n - 1)$

Assuming *n* is a power of 2 and that $T(1) = k_2$

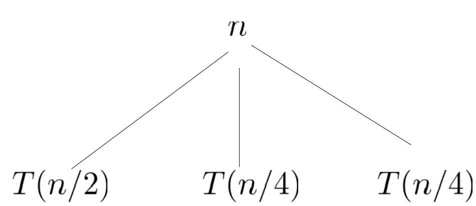
We need to prove that $T(2n) = 2k_1 n \lg 2n + k_2(4n - 1)$

- By definition: $T(2n) = 2T(n) + 2k_1 n + k_2$

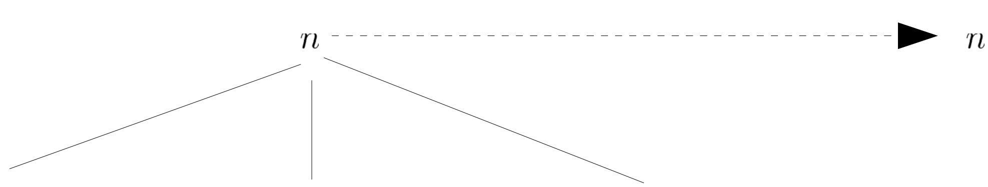
- Using the inductive hypothesis, we have: $T(2n) = 2(k_1 n \lg n + k_2(2n-1)) + 2k_1 n + k_2$
- Simplifying, we obtain: $T(2n) = 2k_1 n(\lg n + 1) + k_2(4n-1)$
- Noting that $1 = \lg 2$, we get: $T(2n) = 2k_1 n(\lg n + \lg 2) + k_2(4n-1)$
- Using the identity $\log a + \log b = \log ab$, we obtain: $T(2n) = 2k_1 n \lg 2n + k_2(4n-1)$
- **QED**

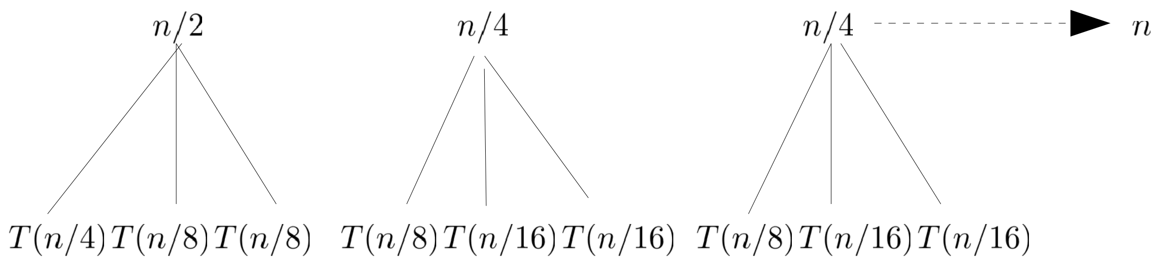
Big-O analysis of T(n) = T(n/2) + 2 T(n/4) + n

- We start by drawing a recursion tree:



- Expanding the second row, we get:





- If we continue the expansion, we will see that the sum of the non-recursive elements for each level is *n*.
- Alas, the sum of the non-recursive elements is **not** *n* for **all** levels.
- First, note that the leaves of the tree are for $T(1) = 0$
- Hence the left-most branch of the tree has depth $\log_2 n$ whereas the right-most branch has depth $\log_4 n = \frac{\log_2 n}{2}$.
- Hence for all levels below $\log_4 n$ contribute less than *n*.

Analyzing T(n) = 2T(n/4) + T(n/2) + n using Big-O

- Recall that $f(n) = O(g(n))$ if there exist constants *c* and n_0 such that:

  $f(n) \leq cg(n)$ for all $n > n_0$
- Now, suppose we "pretend" that every level of the $T(n) = 2T(n/4) + T(n/2) + n$ recursion tree contributed *n*. (Of course, this is not true for levels greater than $\log_4 n$(i.e. for the bottom half of the tree.)
- But we can now say that all levels contribute $\leq n$to the total.
- Furthermore, the leaves (the bottom nodes on the tree) are $T(1) = 0$, so they contribute nothing.
- The maximum depth of the tree is $\lg n$ and since each level contributes at most *n*, we can say that $T(n) \leq n \lg n$.
- Consequently, $T(n) = O(n \lg n)$

Analyzing T(n) = 2T(n/4) + T(n/2) + n using Big-Omega

- When we say that an algorithm has O(g(n)) complexity, this means the algorithm is *no worse* than g(n). In other words, Big-Oh gives an *upper bound* on the complexity.
- It is also possible to define an asymptotic *lower bound* called Big-Omega. When we say an algorithm has $\Omega(g(n))$ complexity, we guarantee that the algorithm is *no better* than g(n) asymptoticly (i.e. for large enough *n*).

- We define Big-Omega as follows: $f(n) = \Omega(g(n))$ if there exist constants c and n_0 such that:

  $f(n) \geq cg(n)$ for all $n > n_0$
- Consider the recursion-tree for $T(n) = 2T(n/4) + T(n/2) + n$.
- Suppose we only consider the top half: i.e. the first $\log_4 n$ levels.
- Since more than half of the tree is neglected, the total contributions from these levels will be less than the real total.
- But we have already seen that each of these levels contributes *n*.
- Consequently, we can say $T(n) \geq n \log_4 n = (1/2)n \lg n$
- Therefore, $T(n) = \Omega(n \log n)$

Big Theta

- We say that $f(n) = \Theta(g(n))$ if there exist constants c_1, c_2 and n_0 such that:

  $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n > n_0$
- Equivalently, $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
- Since $T(n) = 2T(n/4) + T(n/2) + n$ is both $T(n) = \Omega(n \log n)$ and $T(n) = O(n \lg n)$, we can also say that $T(n) = \Theta(n \lg n)$

More about Asymptotic Notation.

- Another way to determine whether functions are Big-O or Big-Omega is to use limits.
- We assume that all functions are monotonically increasing and non-negative for sufficiently large n.
- In particular, if $\lim_{n\to\infty} f(n)/g(n) < \infty$, then $f(n) = O(g(n))$
- However, if $\lim_{n\to\infty} f(n)/g(n) = \infty$, then $f(n) = \Omega(g(n))$
- Finally, $0 < \lim_{n\to\infty} f(n)/g(n) < \infty$, then $f(n) = \Theta(g(n))$

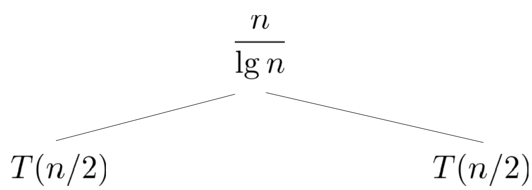
Big-Theta by inspection: some "rules of thumb"

- If $f(n)$ is the sum of terms, find the term that grows fastest. That gives you $\Theta(fastest\ growing\ term)$. And ignore anything that multiplies this term (or any other term).
- Examples:
  - $f(n) = \lg n + 5n^2 + \sqrt{n} = \Theta(n^2)$
  - $f(n) = 6n + 5 \times 10^{20} n^3 + 12134 \times (\sqrt{n})^7 = \Theta(n^{3.5})$

- $n \log n! + n^2 = \Theta(n^2 \log n)$
- $n^{123} + 1.01^n = \Theta(1.01^n)$
- $n\sqrt{n}\left(\sum_{i=1}^{n} 1/i\right) + 15n \lg n = \Theta(n^2 \log n)$
- $4n! + 5^n = \Theta(5^n)$

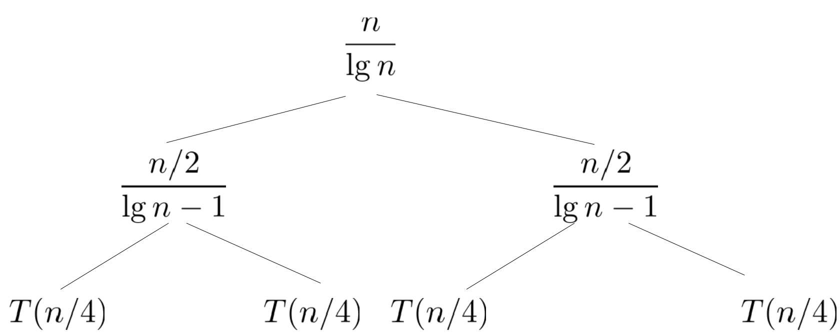
Solving $T(n) = 2T(n/2) + \frac{n}{\lg n}$ where $n \geq 2$

- Start with the basic recursion tree:



- Now expand the $T(n/2)$ layer noting that

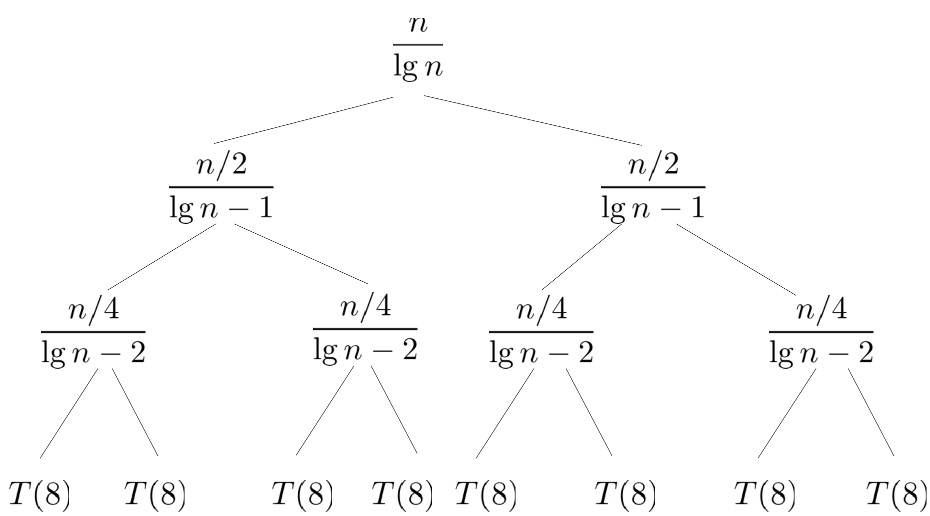
$$T(n/2) = 2T(n/4) + (n/2)/\lg(n/2) = \frac{n/2}{\lg n - \lg 2} = \frac{n/2}{\lg n - 1}$$



- Let's expand one more layer noting that:

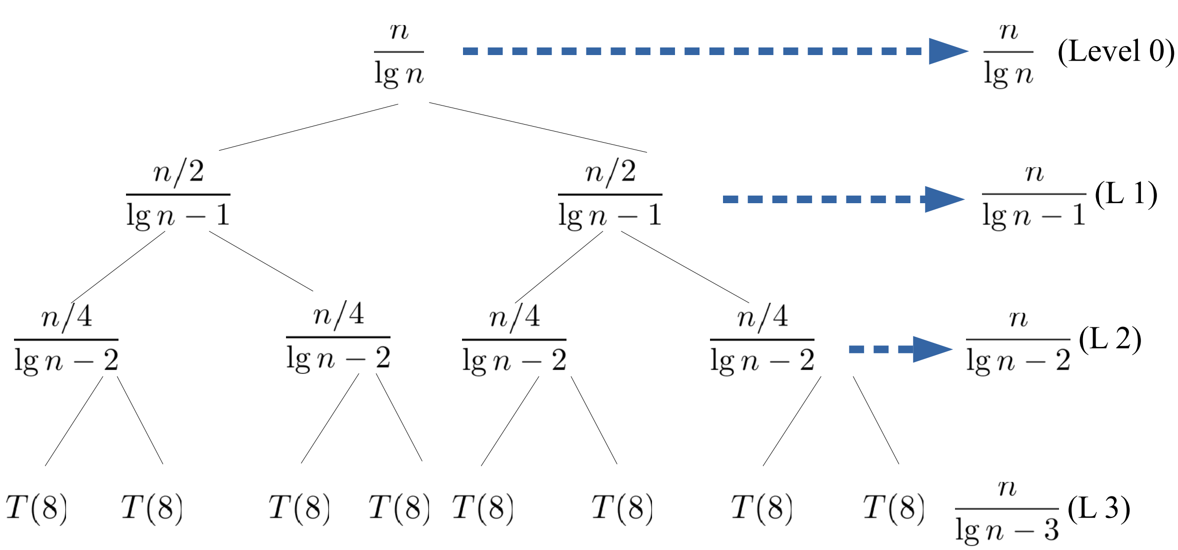
$$T(n/4) = 2T(n/8) + (n/4)/\lg(n/4) = \frac{n/4}{\lg n - \lg 4} = \frac{n/4}{\lg n - 2}$$

- We get:



- Let's add up the (non-recursive) contributions of each layer:

- Let's add up the (non-recursive) contributions of each layer:



- How many levels are there?
- Assuming $n = 2^m$, there will be $m - 1$ levels.
- Adding up the contributions from each level, we get:

- $T(n) = \sum_{i=0}^{m-2} \frac{n}{\lg n - i} = \sum_{i=0}^{m-2} \frac{n}{m-i} = n \sum_{i=0}^{m-2} \frac{1}{i+1} = nH_{m-1} \approx n \ln m.$
- But $m = \lg n$.
- So, $T(n) \approx n \ln \lg n = \Theta(n \log \log n)$

Questions

1. Fill in the columns labelled $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$ and $f(n) = \Theta(g(n))$ as true or false.

$f(n)$	$g(n)$	$f(n) = O(g(n))$?	$f(n) = \Omega(g(n))$?	$f(n) = \Theta(g(n))$?
$2^{\lg n} + 5$	n			
$1.001^n + 6n^3$	1.2^n			
$1.001^n + 6n^3$	n^{1000}			
$3 \times 4^{\lg n} + 5n\sqrt{n}$	$n^{1.5}$			
$3 \times 4^{\lg n} + 5n\sqrt{n}$	n^2			
$\log n!$	$n \log n + n$			

2. Determine the Big-Oh complexity of $T(n) = 2T(n/2) + n/\lg n$. (You may find this challenging!) (Note: valid only for $n \geq 2$. You may assume any convenient base case. You may also assume that *n* is a power of 2.)
3. Determine the simplest Big-Theta complexity of each the functions below by inspection.
   a) $32 + n \log_5 n + n^2 \sqrt{n} = \Theta(\qquad)$
   b) $1000000n + 5 \times 2^{n^2} + 123 \times 3^n = \Theta(\qquad)$
   c) $5n^3 + \left(\sum_{i=1}^{n} i^2\right)^2 + 20n^5 = \Theta(\qquad)$
   d) $\log_{10} n! + n^2 \lg n = \Theta(\qquad)$
   e) $\left(\sum_{i=1}^{n} 1/i\right)^{1.2} + 15 \lg n = \Theta(\qquad)$

References (text book and online)

- *CLRS:* Chapter 3.1, 3.2
- kclowes book: Chapter 4