CS 305
Design and Analysis of Algorithms

09 / 16 / 2021

Instructor: Michael Eckmann

Today's Topics

- Questions / Comments?
- Reading:
  - 29-38 (MergeSort, covered last time)
  - 56-57 (logarithms, to be covered today)
  - 65-67 (recurrence relations, covered last time)
  - 88-92 (recursion tree method of solving recurrence relations)
  - 68-82 (optional reading on 2 example problems and algorithms w/ recurrence relations)
  - 93-96 (Master Theorem, to be started today)
- Logarithms
- Master Method of solving recurrences
  - Prove case 1

Recurrence Relations

- For MergeSort
- Time(n) = 2 * Time(n/2) + n
    - Number of recursive calls is 2
    - The time for one recursive call is Time(n/2)
    - The time in a call is n
- In general for divide and conquer
- T(n) = a * T(n/b) + f(n)
    - Number of recursive calls is a
    - n/b is size of list in a recursive call
    - The time in a call is f(n)

Recurrence Relations

- In other words, a divide and conquer algorithm that creates *a* subproblems each a factor of *1/b* the size of the original problem and takes *f(n)* amount of time to do the divide and combine steps.
- $T(n) = a * T(n/b) + f(n)$
    - Number of recursive calls is a
    - n/b is size of list in a recursive call
    - The time in a call is f(n)

Recurrence Relations

- Let me draw the recursion tree for arbitrary a and b.
- How many leaves?
- Let's see an example with explicit values for a and b.
- How many leaves?
- $T(n) = a * T(n/b) + f(n)$
- Let's prove, by induction, that the number of leaves in a perfect binary tree is 2^d, where $d =$ depth of the tree.

Logarithms

$y = \log_b a$ iff $b^y = a$, $a > 0$, $b > 0$ and $b \,!= 1$

- In English: one way to read $\log_b a$ is "what exponent of b results in a?"
- When b=2, $\log_2 a$ is the number of times a can be cut in half (until you hit 1.)
- $\log_2 1024 = ?$ It is the power of 2 that gives you 1024 (or the number of times 1024 be cut in half until you hit 1.)

Logarithms

- Let’s show some properties of logarithms on the board.

Logarithms

- Let's go back and look at the a-ary tree and recall there are a^(log$_b$n) --- that is a to the power log$_b$n
- Any other way to specify a^(log$_b$n) ?

Logarithms

- Let's go back and look at the a-ary tree and recall there are a^($\log_b n$) --- that is a to the power $\log_b n$
- Any other way to specify a^($\log_b n$) ?
- n^($\log_b a$)
- better because it is n to some constant power

Master Method for Recurrences

- Soon we will learn the Master Method which can be used to solve recurrences of the form: $T(n) = a * T(n/b) + f(n)$
- There are 3 cases to consider each based on a relationship of time spent at leaves vs. at the root.
- As an example, and to help us shortly, let's now count the number of nodes in a perfect tree, say a 3-ary tree.

Master Method for Recurrences

- Recurrences of the form: $T(n) = a * T(n/b) + f(n)$

- Case 1: if time spent at root is “at most little bit smaller” than time spent at all the leaves, then the running time of algorithm is dominated by the time spent at the leaves
- Case 2: if time spent at root is big theta (same as) of time spent at all the leaves then it is lgn * time spent at leaves
- Case 3: if time spent at root is “at least little bit larger” than time spent at all the leaves, then the running time of algorithm is dominated by the time spent at the root

Master Method for Recurrences

- Let's write T(n) from that a-ary recursion tree as time spent at leaves plus time spent at internal nodes
- Let's prove case 1 of master theorem which will tell us the running time when time spent at root is "at most little bit smaller" than time spent at all the leaves