Data Structures and Algorithms .. Prof. D.W. Harder
Prof. Ladan Tahvildari

June 13th 2019.

ECE 250

all practice code/algorithms can be found on OSX "~/Documents/Revision material ECE/ECE250/ECE250.cpp" and subfolders to "/ECE 250/"!

- Objectives:

Study:

(i) Effective and Efficient data structures and algorithms (lang. independent settings).

(ii) Improve my conception of the levels of abstraction

(iii) Better program design experience

(iv) Emphasize on mathematical aspects of program efficiency.

est. tot: 35 hours

Topic (approx 35 lectures by Dr. Tahvildari.)	EST. TIME
started @ 10:10 PM 1) ~~Asymptotic and Algorithm Analysis~~ ✓	~~5 hours.~~ doc: 24/June/'19
2) ~~Elementary Data Structures~~ ✓	~~2 hours~~ 24/June/2019
3) ~~Hashing~~ ✓	~~3 hours~~ 24/June/2019
IMP % 4) ~~Search trees~~ ✓, ~~Balanced BSTs~~, B-Trees	2/7 24/June/2019 7 hours ~3.5/7 25/June
5) Heaps, Priority Queues	2 hours
IMP 6) Sorting Algorithms. (memorize the algorithms/practice) (memorize their optimisation levels)	3 hours.
7) Algorithmic Paradigms (Que?)	3 hours
IMP % 8) Graphs	7 hours
9) NP - completeness	3 hours.

June 13, 2019.

ECE 250:

Asymptotic Notation and Algorithm analysis:

※ topic start time: 10:10 pm on June 13, 2019.
est time to ~~of~~ completion for topic: 5 hours

def: Analysis:

- the theoretical study of computer-program performance and resource usage.

studying algorithms and performace allows us to make informed decisions about:

i • feasibility vs unviable solutions in terms of program designs

ii • Teaches the usefulness of scalability and how to code towards good scalability.

iii • Algorithmic math helps us communicate about program behaviour in a standard manner.

INSERTION SORT :

$O(n^2)$

pseudocode :

eg : Input : 8 2 4 9 3 6
Output : 2 3 4 6 8 9

```
def insertion-sort ( Array, n ) :            → probably 1
    for j iterates from (2) to (n) :          → (n-1) → technically n-1 & not n.
        key = Array [j]
        i = j-1
                                              → edited notes to make the array read start at 0 & not 1.
        while i ≥ 0 and Array [i] > key :
            Array [i+1] = Array [i]
            i = i - 1
        Array [i+1] = key
end
```

X═══════X

(n)

Array : | 8 | 2 | 4 | 9 | 3 | 6 | →

j = 1 → key = Array[j] → key = 2
→ i = j - 1 = 0

line 7 : → while i ≥ 0 ✓
and Array[i] = (8 > 2) ✓

line 8 : [illegible] Array[1] [illegible]
line 9 : i = 0 - 1 = -1 //ends while loop ×
line 11 : Array[0] = 2 ← (since key = 2)

| 8 | 2 | 4 | 9 | 3 | 6 | = 8

the above described lines show the process of $\underline{1}$ pass of (j) from (1) to (n-1:) in the initial state of Array and when j = 1

nd of Pass 1: | 2 | 8 | 4 | 9 | 3 | 6 |

nd of pass 2: | 2 | 4 | 8 | 9 | 3 | 6 | → i.e. 9 stays in its same [illegible]

pass 3: | 2 | 4 | 8 | 9 | 3 | 6 |

pass 4: | 2 | 3 | 4 | 8 | 9 | 6 |

pass 5: | 2 | 3 | 4 | 6 | 8 | 9 | → Array is now sorted !!

Running time:

→ depends on the input array (due sucks ????)

→ expect shorter size array to sort quicker based on the input array size

→ we need an upper bound on running time to ensure good/better performance.

---

worst - case : T(n) → maximum time of algorithm on an input of size n.

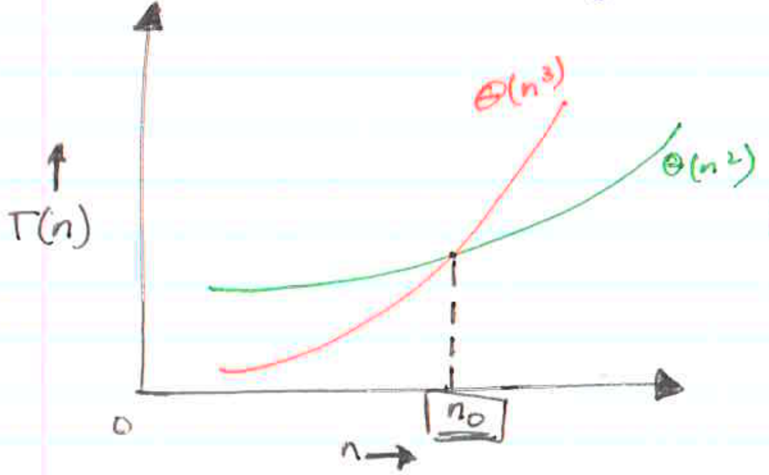
average - case: • can only be assumed sometimes. Need evidence from statistical evidence of inputs.

Best - case: never use this/never expect this.

IF ≥ [illegible] .

Asymptotic Analysis :

When n gets large enough, $\Theta(n^2)$ algorithm will always beat an $\Theta(n^3)$ algorithm!



if $n > n_0$ then $T(n)$ for an $\Theta(n^2)$ algorithm < $T(n)$ for an $\Theta(n^3)$ alg.

Review Floor / Ceiling / Arithmetic Series & Geometric Series

Time at end of insertion sort = 11:10pm on June 13, 2019.
1hour into topic

MERGE_SORT : $\Theta(n \cdot \log(n))$

i.e. better than insertion sort at values for n

- two functions at play for simplicity's sake

def of merge-sort fn 1 →

```
def merge-sort (Array, n):
    if n == 1:
        return;
    else:
        recursively sort Array[1...(n/2)] and
                         Array[(n/2+1)....n]
        merge(the two sorted lists from above)
```

$\Theta(1) \longrightarrow$ if n == 1: return;

$(T(n/2) + T(n/2)) \longrightarrow$ recursively sort

call to fn 2 → merge(the two sorted lists from above)

merge two sorted arrays

$\Theta(n)$

x x

$T(n)$ for the merge fn $\Rightarrow \Theta(n)$ when n = total num. of elements.

i.e. n = sum of num. of elements of both sorted arrays.

$\Rightarrow T(n) \Rightarrow$ linear for merging two sorted. arrays.

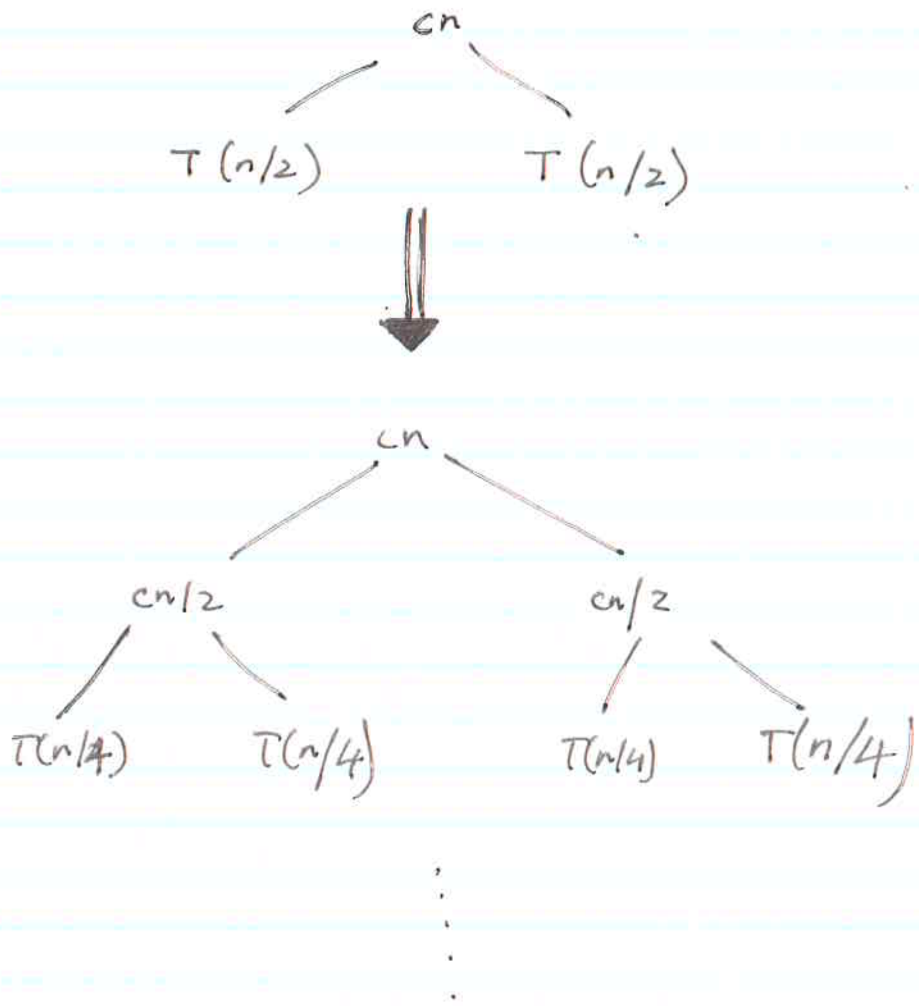
x x

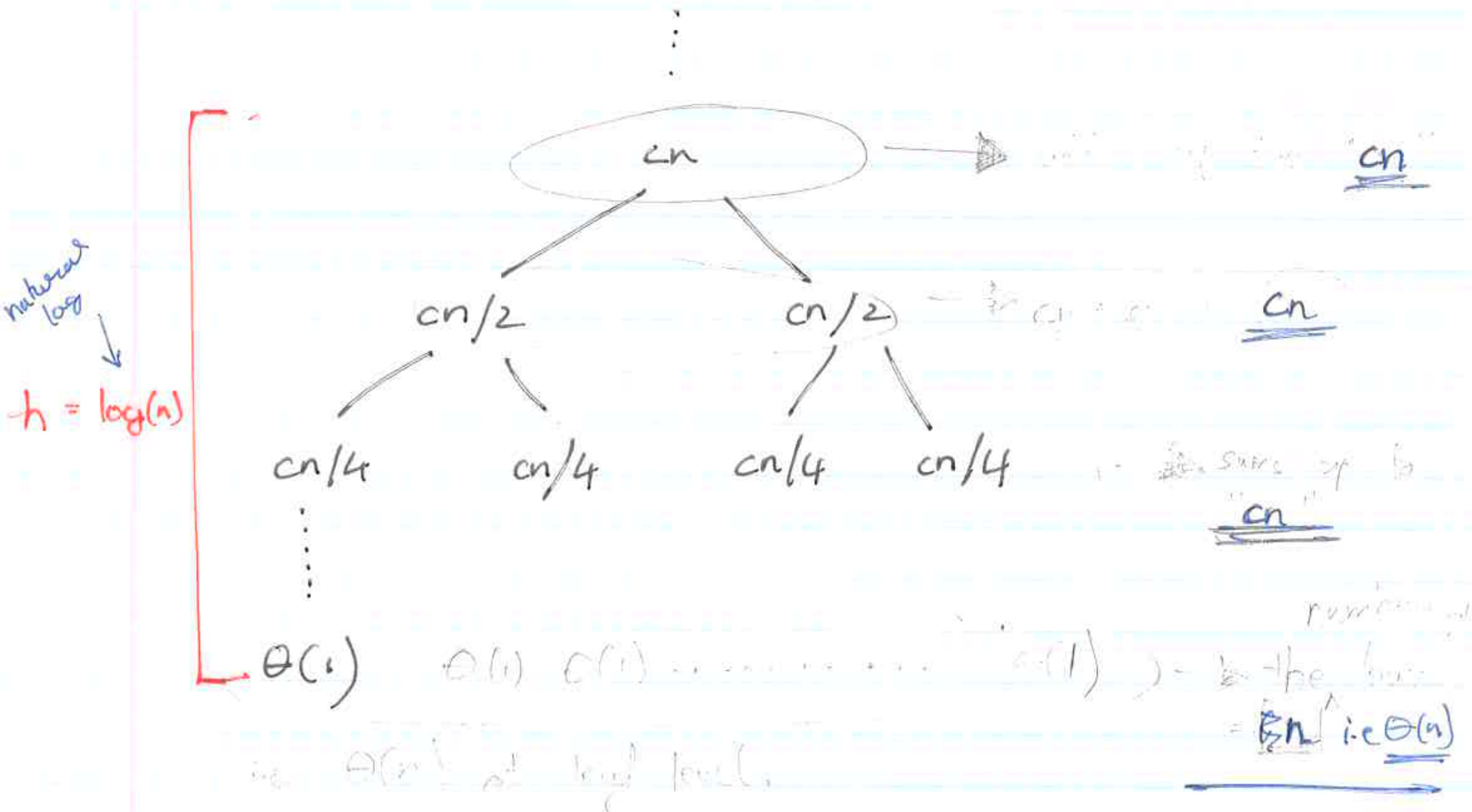
Jun 15

$\Rightarrow$ total expected time taken for merge-sort

$$\Rightarrow \quad T(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ 2T(n/2) + \Theta(n) & \text{if } n > 1; \end{cases}$$

Solve $T(n) = 2T(n/2) + cn$ where $c > 0$ and is constant:





total = $\Theta(n \log n)$

If we look at the run times of merge sort against insertion sort, we can see that merge sort's $\Theta(n \log n)$ runtime grows more slowly than Insertion sort's $\Theta(n^2)$

- As a result, asymptotically, merge sort beats insertion sort in the worst case

i.e. function behaviour in the limit.

Most important asymptotic notations

- O - big O notation → upper bound
  → $\approx \leq$
- Ω - big omega notation → lower bound
  → $\approx \geq$
- Θ - theta notation → tight bound

Recurrances

def: A recurrance is an equation / inequality that describes a function in terms of its value on smaller inputs.

3 methods to solve recurrances:

- Recursion Tree
- Repeated Substituition
- Master method

... lecture 4 cont'd.

Solving recurrances:

3 main methods:

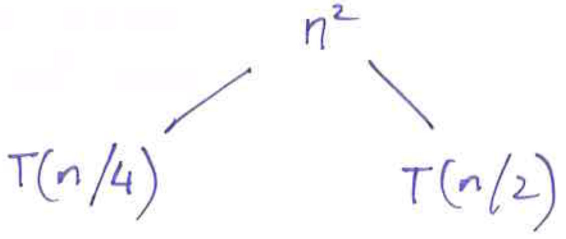
- Recursion-Tree method
- Repeated Substitution
- Master Method.

Only focusing on two methods for this course.

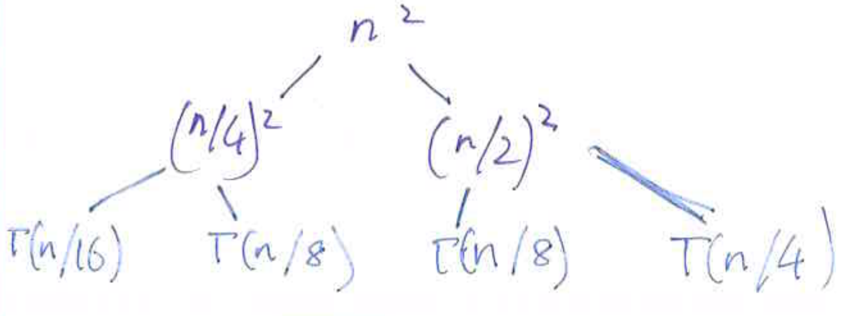
Recursion Tree Method

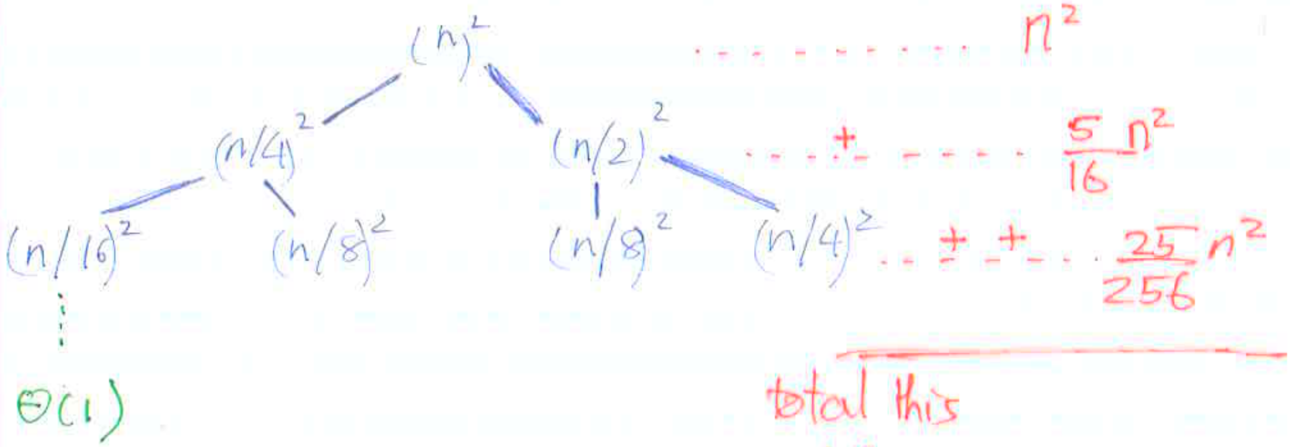
eg Solve $T(n) = T(n/4) + T(n/2) + n^2$

ans: $T(n)$ ↓



Sep 10) The root of a tree always represents the cost of combining the two subarrays you originally split the input array into.





total this up: $n^2\left(1+\frac{5}{16}+\left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3+\dots\right)$

this is a geometric series $= \Theta(n^2)$ → conclusion tight bound on n^2 as rest are constant for a given array size & may be disregarded

memorey refresh:

~~$1 + x + x^2 \dots$~~

~~$a = 1$~~
~~$r = x$~~
~~$r = \frac{a_{i+1}}{a_i}$~~

~~if $a_0 = 1$ then $\sum_{i=0}^{n} a_i = \frac{1-a^{n+1}}{1-a}$~~

~~if~~

No that makes NO sense rewrite this correctly.

DW Harder's Notes :

O(n) loops vs Θ(n) loops :

Algorithm 1 :

```
int find_max (int *array, int n) {

    int max = array[0]

    for (int i = 1; i < n; ++i) {
        if (array [i] > max) {
            max = array [i];
        }
    }, return max;
}
```

tries to find the max value in the array. i.e. Every element in the array has to be checked, i.e all n elements.

i.e. runtime = $\Theta(n)$ tight-bound!!

Algorithm 2 :

```
bool linear_search (int val, int *array, int n) {

    for ( int i = 0 ; i < n ; ++i) {
        if (array [i] == value) {
            return true;
        }
    }
    return false ;
}
```

tries to just find an element in the array, i.e. can be found at beginning or ending of given array or anywhere in-between.

i.e. runtime ⇒ $O(n)$ upper-bound!!

DW Harder's notes online

Asymptotic analysis practice questions

2.3) a) ∘ $n^2 + 2n + 5$

∘ n^2

i) if $n = 1000$

then relative error = $\frac{\text{absolute error}}{\text{actual value}}$

$$\Rightarrow \text{rel. error} = \frac{n^2 + 2n + 5 - n^2}{n^2 + 2n + 5}$$

$$= 2005 / 1000000 + 2005$$

$$= \underline{2005 / 1002005}$$

2.3) b) $n + \underline{\ln(n)}$ $\underline{n + 5}$

∘ $\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) = O(g(n))$

∘ $0 < \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) = \Theta(g(n))$

∘ $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$

∘ $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \omega(g(n))$

$\lim_{n \to \infty} \frac{f(n)}{g(n)} > 0 \Rightarrow f(n) = \Omega(g(n))$

2.3) b) ① $\boxed{n+5 = O(n+\ln(n))}$ — ?

② $\boxed{e^n = \omega(n^2+2)}$

③ $2n^2+3n+1 \qquad 3n^{\log(3)}+n$

$\hookrightarrow \lim_{n\to\infty} \frac{2n^2+3n+1}{3n^{\log(3)}+n} \quad * \quad \frac{n^2}{n^2}$

$\Rightarrow \lim_{n\to\infty} \frac{2+3/n+1/n^2}{3n^{\log(3)-2}+1/n} \Rightarrow \frac{2}{0} = \infty$

$\Rightarrow \boxed{(2n^2+3n+1) = \omega(3n^{\log(3)}+n)}$

④ $(2n+4) \lessgtr (5n\cdot\ln(n)+3n+2)$

$\hookrightarrow \lim_{n\to\infty} \frac{2n+4}{5n\ln(n)+3n+2} = \lim_{n\to\infty} \frac{2+4/n}{5\ln(n)+3+2/n}$

$\Rightarrow \lim_{n\to\infty} \frac{2}{5\ln(n)+3} \Rightarrow \underline{\underline{0}}$ as $n\to\infty$

$\Rightarrow \boxed{(2n+4) = o(5n\ln(n)+3n+2)}$

⑤ $n\ln(n) \lessgtr n\ln(n^5)$

$\lim_{n\to\infty} \frac{n\ln(n)}{\cancel{n}\cdot 5\ln(n)} = \underline{\underline{\frac{1}{5}}} \Rightarrow \boxed{n\ln(n) = \Omega(n\ln(n^5))}$

Jun 18' Diffs b/w the three $\lg \neq \log \neq \ln$

$\ln(n)$	$\lg(n)$	$\log(n)$
↓	↓	↓
base e	base 2	base 10
	⇒ $\log_2(n)$	$\log(n)$
		⇒ $\log_{10}(n)$

Repeated Substituition :

- substitute a few times till a pattern emerges
- write a formula in terms of n and the number of substitutions (i)

(as shown in assignment 1. 3)

28/Jun
notes
cont'd

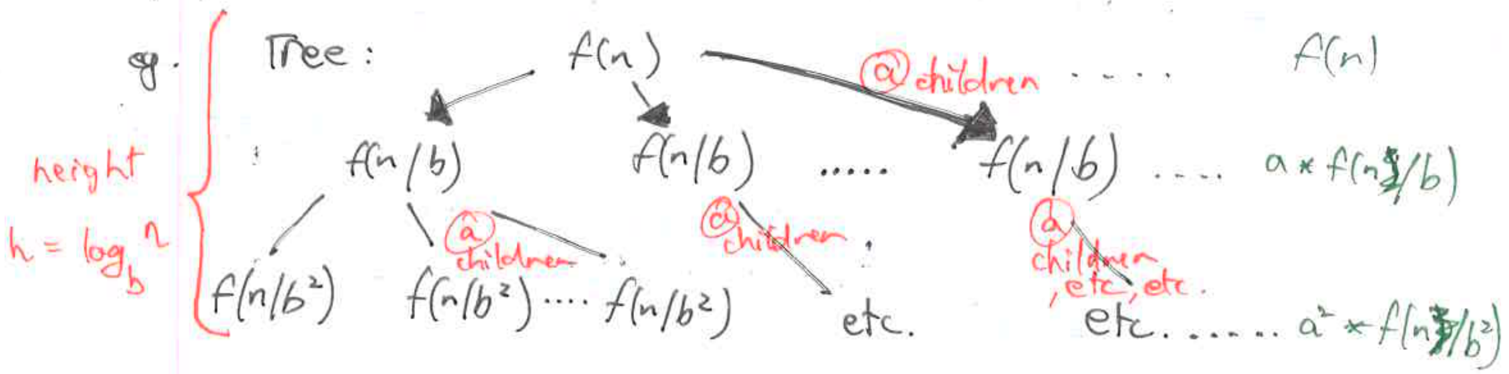
Master method for solving recurrances:

For recurrances of the form:

$$T(n) = a\,T(n/b) + f(n)$$

where f is asymptotically positive and
a & b are constant such that $\boxed{a \geq 1}$ & $\boxed{b > 1}$

If each recursion ~~[illegible]~~ in a function has the same number of children calls ~~each~~ ~~is~~



$\Rightarrow$ number of leaves $= \underline{a^h}$

$= ~~[illegible]~~ = a^{\log_b n}$

total: $\boxed{n \log_b a \; T(1)}$

i.e. tot. no. of leaves * run time at base case for leaves.

$\boxed{\text{no. of leaves} = n^{\log_b a}}$

3 cases of useage for the master method:

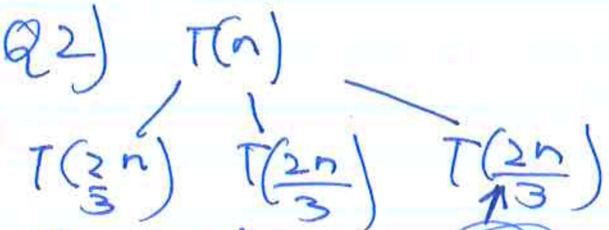
- Running time dominated by cost of running at the leaves
- Running time evenly distributed throughout the tree
- Running time dominated by the cost of running at the root.

→ Using the master method, we simply charecterise the dominant term to solve the recurrance.

→ In each of the above 3 cases, compare $f(n)$ with ~~$O(n\log$~~ $O(n^{\log_b a})$

a → no of children for each recursion
b → the term which n is divided by on each recursive call.

eg. in tutorial 1 Q2)



⇒ $b = 3/2$ since $n/b = n/(3/2) = \frac{2n}{3}$

Strat:

1) Determine a, b & $f(n)$ from a given recurrence

2) Determine $n^{\log_b a}$

3) Compare $f(n)$ and $n^{\log_b a}$ asymptotically

4) Determine the case of the mentioned 3 master theorem cases and apply as expected.

x x

Lecture 6:

Describing the 3 cases for using the master method:

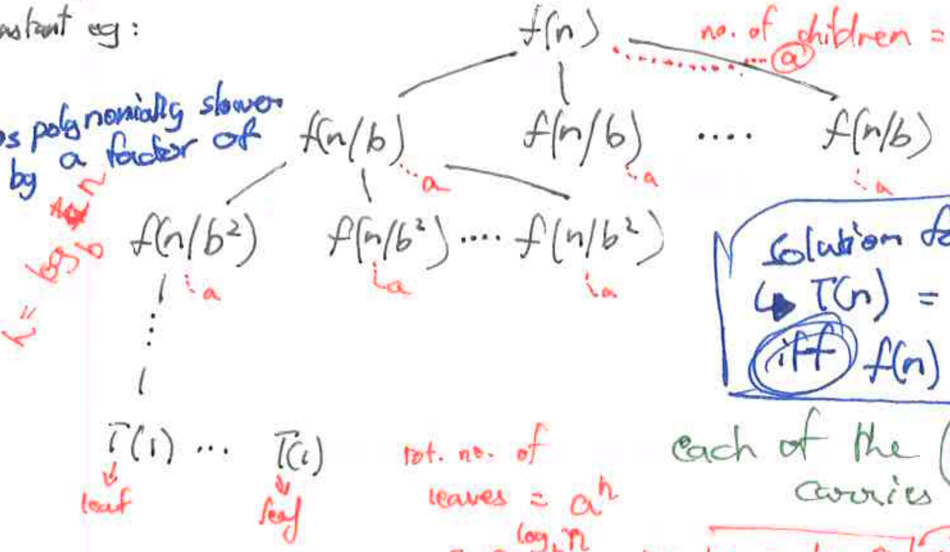
- Case 1: The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight. eg: weight of leaf = $\Theta(3)$ etc..

mp

se 1: $f(n) = O(n^{\log_b a - \varepsilon})$

some constant eg: $> \emptyset$.

e f(n) grows polynomially slower n $n^{\log_b a}$ by a factor of ε



tot. no. of leaves = a^n = $\underline{a^{\log_b n}}$

i.e. $\boxed{n = \log_b a}$ (irrelevant to example)

Solution for case 1 problems:

$\hookrightarrow T(n) = \underline{\underline{\Theta(n^{\log_b a})}}$

iff $f(n) = O(n^{\log_b a - \varepsilon})$: ε being a constant $> \emptyset$.

each of the $(a^{\log_b n})$ leaves carries $T(1)$ of the run time weight.

i.e Case 1

- <u>Case 2:</u> The weight is approximately the same on each of the $\log_b n$ levels.

Solution to case 2 ~~[illegible]~~ problems :

$$\boxed{T(n) = \Theta\left(n^{\log_b a} * \lg n\right)}$$

eg: to describe the situation more clearly to identify case 2 q's:

~~[illegible] = [illegible]~~

$$T(n) = a * T(n/b) + f(n)$$

if $f(n)$ = (for example) n^2

and $f(n) = \Theta(n^2)$

i.e. they ~~[illegible]~~ grow polynomially at the same rate.

=> case 2 since $f(n) = \Theta(n^2)$ (assuming $n^{\log_b a}$ calculates as n^2).

=> soln:

$$T(n) = \Theta\left(n^{\log_b a} * \lg n\right)$$

- Case 3: When the weight decreases geometrically from the root to the leaves.

The root holds a constant fraction of the total weight.

Solution : $\boxed{T(n) = \Theta(f(n))}$

When:

$f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$
then, $f(n)$ grows polynomially faster than $n^{\log_b a}$ by a factor of n^{ε};

AND if $f(n)$ satisfied the regularity condition

i.e. regularity condition

$\boxed{a f(n/b) \leq c f(n)}$ for some constant $\boxed{c < 1}$.

note: we're checking to ensure the weight of all children from root node → down decrease in weight compared to some negative/zero constant times $f(n)$.

i.e.

ensuring $f(n)$ grows polynomially faster than $n^{\log_b a}$.

then $\boxed{T(n) = \Theta(f(n))}$ as ~~shown~~ mentioned above

END OF

ALGORITHM/

ASYMPTOTIC

ANALYSIS

23/June/2019

Elementary Data Structures:

def • Continguous Storage: abstract data type designed such that memory locations right next to a memory address in question wil be of the same data type.

(Paraphrased a bit too much there lol)

def • Node-based Storage: uses "at least" two data points to keep track of information being stored.
(i) a reference to the object itself.
(ii) a reference to the next item.

more info Dr. LT's notes
Lecture 7 pg 9

IMP
Deadline: 26/June/2019

- Practice implementing dictionaries using arrays, linked lists, doubly linked lists while ensuring the time complexity constraints are met.?

Look through the course's definition of:

- Dictionaries
- Stacks
- Queues
- Hash maps.

23 / June / 2019

Lecture 8:

Direct Access Table : a rudimentary version of a hashmap and is manually implemented.

If the set of keys $K \subseteq \{0, 1, 2, 3, \ldots\ldots m-1\}$ such that the keys are distinct.

We may then, set up an array with "m" elements with all the keys from K. ~~Except the [illegible] with~~ k

i.e. $$T[k] = \begin{cases} x & \text{if } k \in K \text{ and } Key[x] = k \\ NIL & \text{otherwise.} \end{cases}$$

⇒ Operations thus would take $\boxed{\Theta(1)}$ time

- But a large range of keys to add to the direct access table causes an issue as the keys may be large.

24 / June

Hash functions : maps out the universe of all keys to records that are mapped to said keys (whatever it's a hash lol, ezy)