18/10/2021

Course: Divide & Conquer, Sorting

Week1: Why Study Algo?

-> Imp for all branches for Compsc

-> Plays a key role in modern tech innovation

#Analogy:

- Classical Shortest path algo
- Effectiveness of public key Cryptography relies on this number-theoretic alg
- Computer graphics need computational Primitives supplied by geometric algo
- Database indices rely on balanced search tree data structures
- Computational biology uses dynamic prog alg to measure genome similarity
- .... so on

-> Search engine

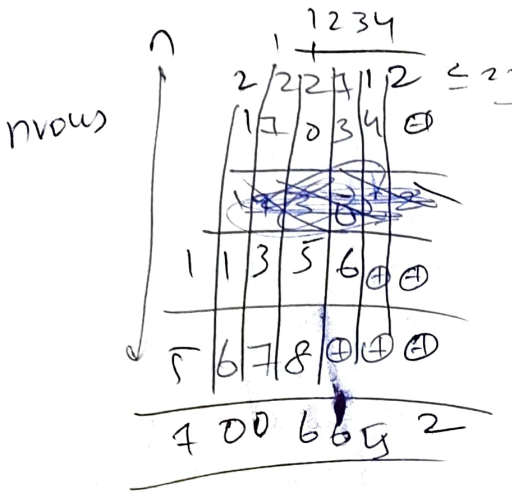
- novel "lens" on process outside of CS & tech
  - -> quantum mechanics, economic markets, evolution
- good for brain, challenging
- fun

Week 1.2 — Integer Multiplication

Input: two n-digit nums x, y

Output: product x, y

2 3 2
2 1 3 3 1
5 6 7 8



$\leq 2n$ basic operations, carryover etc

$2n^2$ operations

Upshot # operations overall $\leq$ Constant $\cdot n^2$
(Circa)

Algo Designer's Mantra

"Perhaps the most important principle for the good algorithm designer is to refuse to be content".

— Aho, Hopcroft, & Ullman, The Design & Analysis of Computer Alg, 1974

"CAN WE DO BETTER?!"

Week 1 Karatsuba Multiplication.

$x = (5\ 6)\ (7\ 8)$ — a, b

$y = (1\ 2)\ (3\ 4)$ — c, d

Step1: Compute $a \cdot c = 672$

Step2 Compute $b \cdot d = 2652$

Step3 Compute $(a+b)(c+d) = 134\ 46$
$= 6164$

Step4: Compute $(5) - (2)\ (1) : 2840$

Step5:
$$\begin{array}{r} 6720000 \\ 2652 \\ 284000 \\ \hline 7006652 \end{array}$$

A recursive Alg:

Basic Steps of recursive programs.

From IBM:,

1) Initialize the algorithm. They need a seed value to start with. Either pass a parameter to the function or by providing a gateway function that is not recursive but that sets up the seed values for the recursive calculation.

2.) Check whether the current value(s) being processed match the base case. If so, process & return the value.

3.) Refine Redefine the answer in terms of a smaller or simpler sub-problem or sub-problems

4.) Run the algo on the sub-problems

5.) Combine the results in the formulation of the answer.

6.) Return the results.

Recursive Algo (continued)

- Given a number x with n digits, it can be expressed, interms of two, n over two digit numbers

A = The first half of the digits shifted appropriately.

↳ multiplied by 10 raised to the power n over two. $10^{n/2}$

Example: $a = 56$ $b = 78$, $c = 12$, $d = 34$

Then: $x \cdot y = (10^{n/2} a + b) \cdot (10^{n/2} \cdot c + d)$

$= (5600 + 78) \cdot (1200 + 34)$

$= (5678) \times (1234)$

$=$

$x \cdot y = (10^{n/2} a + b) \cdot (10^{n/2} \cdot c + d)$

$= 10^n \cdot ac + 10^{n/2}(ad+bc) + bd$

n-seven → total straightforward approach

$10^{n/2} \cdot 10^{n/2} a \cdot c + 10^{n/2} a \cdot d + 10^{n/2} c \cdot b + bd$

$10^n ac + 10^{n/2}(ad+cb) + bd$

Idea: Recursively compute ac, ad, bc, bd, then compute (*) in the straightforward ways

- Pad ac with n zeros at the end.
- add (ad + bc) and pad with n^2 zeros
- sum up bd.

We need a base case

Karatsuba Multiplication:

Recall: $x \cdot y = 10^n ac + 10^{n/2}(ad+bc) + bd$

Step1: recursively Compute ac

Step2: recursively Compute bd

Step3: recursively Compute $(a+b)(c+d) = ac + ad + bc + bd$

trans'stick: ③ − ① − ② = ⇒ $= ad + bc$

Upshot: only need 3 recursive multiplications (and some additions)

Week 1-2: Course Topics:

1) Vocab for Design & analysis of Alg

2) Divide & Conq algo design paradigm

3) Randomization in alg design

4) Primitives about graphs

5) Use & implementation of Data Structures

1) Vocab for Design & Analysis of Algs

-> E.g.: "Big-Oh" notation
- "Sweet spot" for high level reasoning about algo

Big-O -> way to mathematicize the Sweet Spot.

2) Divide & Conquer Alg:

-> will apply to: Integer mult, sorting, matrix mult, closest pair

-> General analysis methods ("Master Method / Theorem")

3.) Randomization in algo design
-> will apply to Quick sort, Primality testing, graph partitioning, hashing

4.) Primitives for reasoning about graphs:

-> Connectivity information, Shortest path, Structure of information & Social networks.

5.) Use & Implementation of Data structures
-> Heaps, balanced binary search trees, hashing & some variants (e.g. Bloom & filters)

Topics in Second Course:

1) Greedy Algo design Paradigm
- Applications to minimum spanning trees
- Scheduling & information theoretic Coding

2) Dynamic Prog Algo design Paradigm
=> Design & Analysis with examples applications being in genome sequence alignment, the shortest path protocols in communication networks.

- NP - Complete prob & What to do about them
  "P. next equal to NP"
- Fast heuristics with provable guarantees
- Fast exact algo for special cases
- Exact algo that beat brute force search

Week 1-5: Merge Sort: Motivation & Example

Why Study Merge-Sort:

- Good intro to divide & Conquer:
  - Improves over Selection, Insertion, Bubble sorts
    - Then sorting algorithm Lack in Scale with N^2 -> quadratic scale Function of input Size (Constant number squared)
- Calibrate your preparation
- Motivates guiding Principles for algorithm analysis (worst-Case & asymptotic analysis)
- Analysis generalize to "Master Method".
  - Analysis generalizes to "Master Method"
  - Analysis of Merge Sort using What's Called "Recursion-Tree" method
    - Way of typing up the total number of operations that are executed by an algorithm.

and will allow to analyze ~~lots~~ Lots of Diff recursive algo, lots of Diff Divide & Conquer alg, including integer multi algo

The Sorting Problem:

Input: Array of n-numbers, unsorted

5	4	1	8	7	2	6	3

Output: same numbers, sorted in increasing order

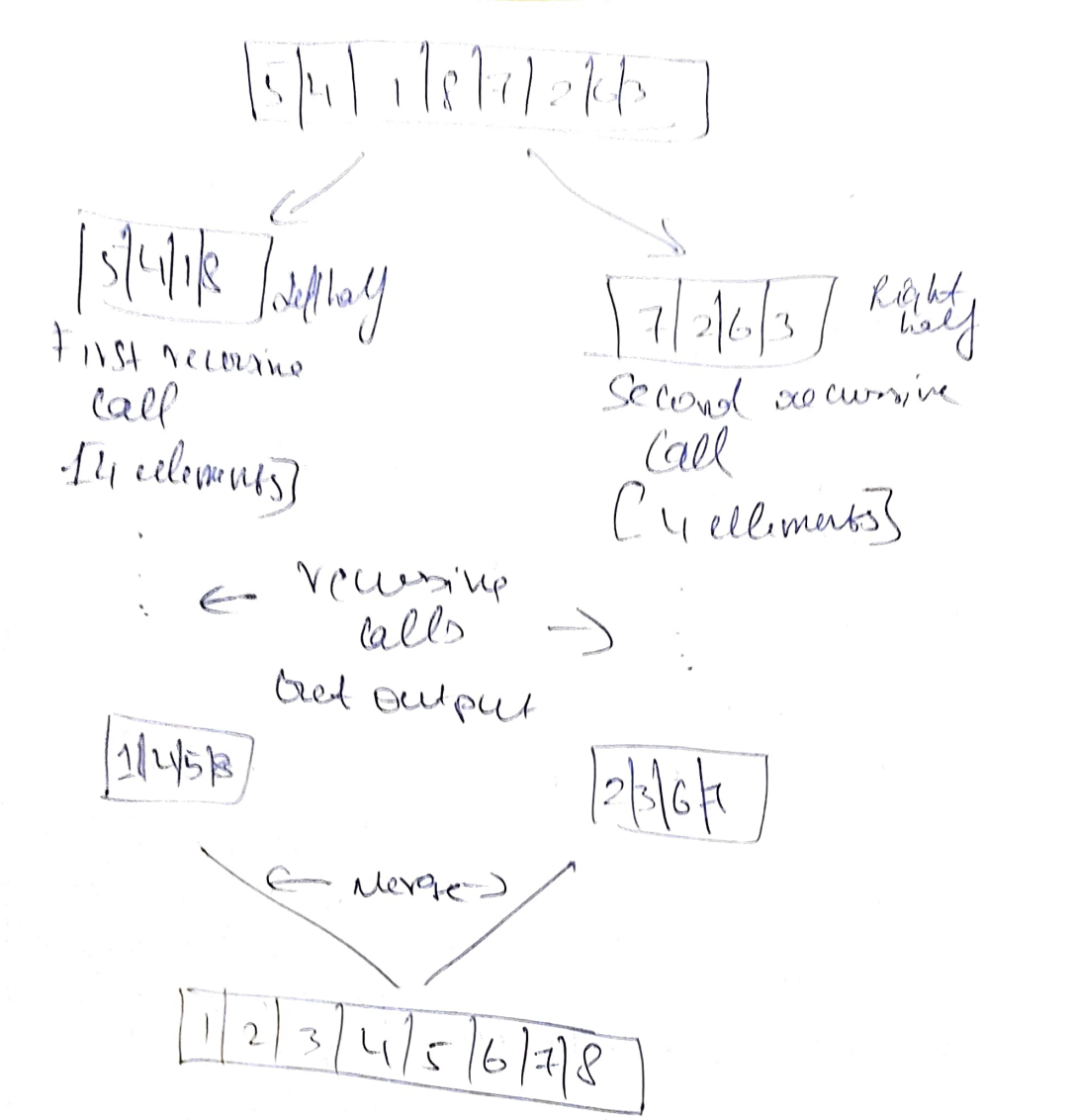
1	2	3	4	5	6	7	8

Exercise:
Assume Distinct
How the merge sort alg implementation and analysis would be diff if at all if there were ties? ☺.

Merge sort Algo
⇒ A recursive Alg, that means a program which calls itself and it calls itself on smaller sub Problems of the same form.

→ Ultimately → sort the given input array.

⇒ Call itself on smaller arrays
↳ A canonical divide & conq algo applica
where we take simply the input array, we split it in half, we solve the left half recursively, we solve the right half recursivel & then combine the results.



→ walk pointers down each of the two sorted sub-arrays, copying over, populating the output array in sorted order

~~Merge~~ Week 1.6 Merge-sort - pseudocode

- recursively sort 1st half of input array

→ " " 2nd " " "

→ Merge two sorted sublists into one

[ignore base cases]

Pseudo code

-> How to Combine Merge ?

Populate the output array in a sorted order, by traversing through the two sorted sub-arrays in parallel.

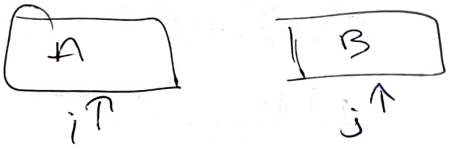
Pseudo code for Merge:

C = output array (length = n)

A - 1st sorted array [n/2]

B = 2nd " [n/2]

Counter



i, j -> initialized to one

i = 1

j = 1

FOR k = 1 to n

If A(i) < B(j)

C(k) = A(i)

i++

Else B(j) < A[i]

C(k) = B[j]

j++

end

Prac: 5, 7, 13, 8, 1, 6, 4, 2

5, 7, 13, 8

1, 6, 4, 2

~~1 5 6 4 2~~

1 8

Merge Sort running time ?

Key Ques: running time of merge sort of array of n numbers?

[running time ≈ # of lines of code executed]

Pseudocode for Merge:

C = output [length = n]

A = 1st sorted array [n/2]

B = 2nd sorted array [n/2]

i = 1
j = 1
} 2 operations

for k = 1 to n: ✓

if A(i) < B(j) ✓

C(k) = A(i) –

i++ –

else [B(j) < A(i)]

C(k) = B(j) –

j++ –

Four operations / Iteration

Upshot: - Running time of the merge sub-routine given an array of m numbers, is at most $(\le cn)$

$\le 4m + 2$

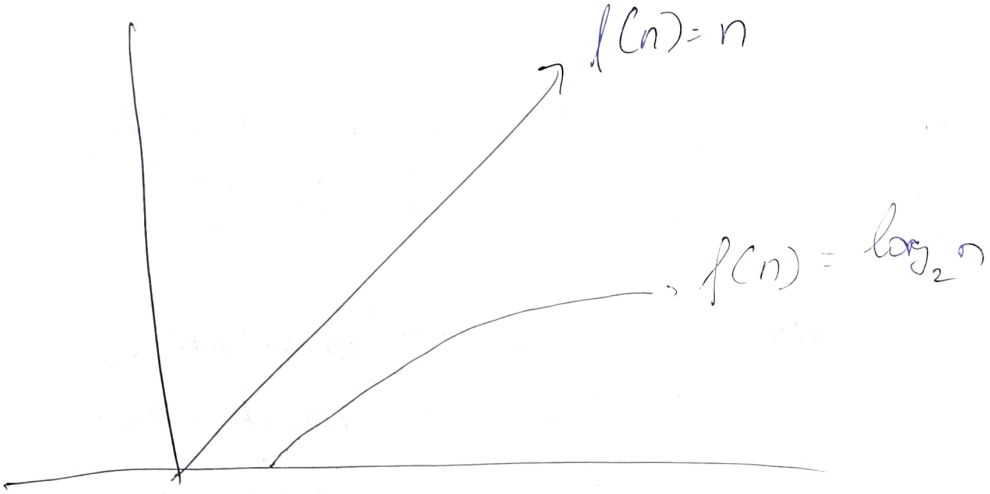
$\le 6m$ (since $m \ge 1$) Let's call $6m$ for time

Running time of Merge Sort:

Claim: Merge Sort executes

$\leq 6n \log_2 n + 6n$ operations

to Sort n numbers.



$\log_2 n$ → Curve becomes very flat

↙

times you divide by 2 until you get down to 1.

$\log_2 n$ slower than Identity function n

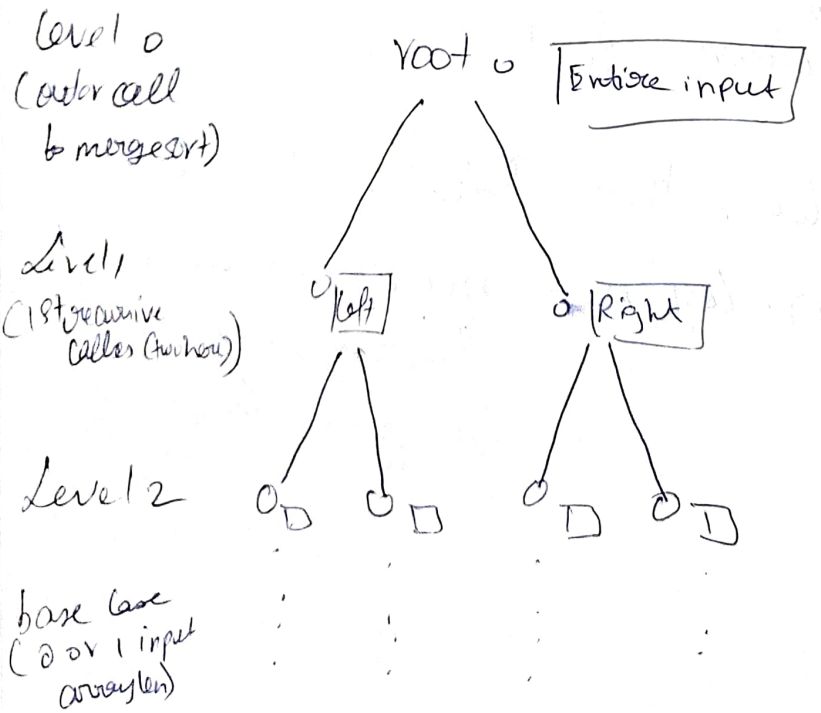
but $n \cdot \log_2 n$ → much, much faster, especially n grows large

Merge Sort analysis:- Week 1.7

Claim: For every input array of n numbers, Merge Sort produces a sorted output array and uses at most $6n \log_2 n + 6n$ operations

Proof of Claim (assuming n = power of 2):
("will use "recursive tree")

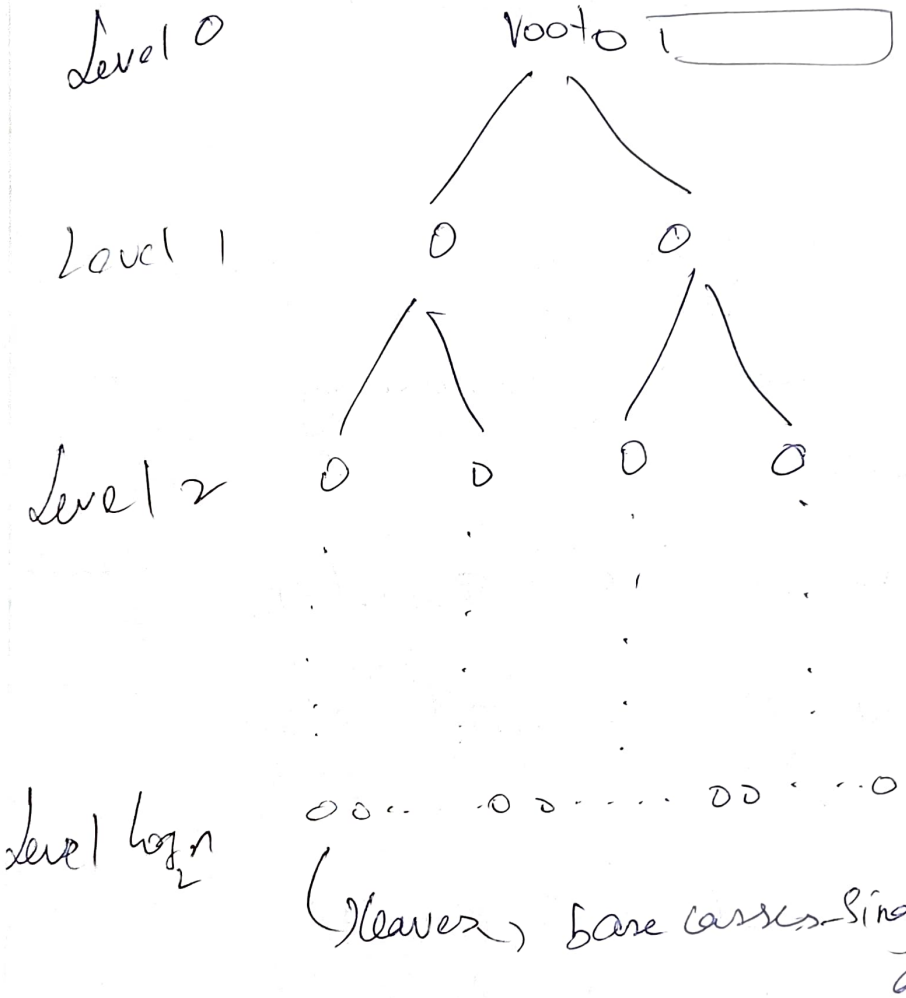
The idea of recursion tree method is to write out all of the work done by the recursive merge sort algo, in a tree structure with a children of a given node corresponding to the recursive calls made by that node.



Number of levels of the recursion tree is logarithmic of the size of the input array.

Basically the input size is being decreased by a factor of 2. with each level of recursion.

Total levels -> $(\log_2 n + 1)$ to be exact



(leaves, base cases - single-element arrays)

At each level, $j = 0, 1, 2, \ldots \log_2 n$, there are 2^j subproblems, each of size $\frac{n}{2^j}$ resp

Merge sort

↳ On an input of size n, it's going to need at most 6n lines of code.

Total # of operations at level j
(each of = 0, 1, 2, ... (log, n))

$\leq 2^j$ → # of level-j subproblems

$2^j \times \left(6 \times \left(\frac{n}{2^j}\right)\right)$

↳ subproblem size at level j

↳ 6 times that many number of lines of code.

↳ work per level j subproblem

$\therefore 2^j \times 6 \times \frac{n}{2^j} = 6n$ (independent of level j)

$\therefore$ 6n operations at level 0, root,
6n operations at level 1, level 2, ... soon

Number of subproblems is doubling with each level of the recursion tree, and secondly, the level of workdone per subproblem is halving with each level of the recursion tree.

∴ Total work:

$(6n) \times (\log_2 n + 1) = 6n \log_2 n + 6n$

$6n$ ↓ work/level

$\log_2 n + 1$ ↓ # of levels

Week: 1.8 — Guiding Principles for analysis of algorithms

Guiding Principle #1

"worst-case analysis": our running time bound holds for every input of length n.

- Particularly applicable for "general-purpose" routine

As opposed to:

- "average-case" analysis } requires domain knowledge
- benchmarks

Avg-case analysis

→ Avg running time of an algorithm under assumption about the relative frequencies of all inputs

Guiding Principles #2

"Won't pay much attention to constant factors, lower-order terms"

Justifications:

(1) Easy easier

(2) Constants depend on architecture/ compiler/ programming language

(3) Lose very little predictive power

The constant factors are important for crucial loops. Then, if that's the case, optimize the factors like crazy.

Guiding Principle #3:

"Asymptotic Analysis"

1) Focuses on cost of large running times input sizes

The performance of an algorithm as the size N of the input grows large, it tends to infinity.

Better than algorithm with quadratic dependencies.

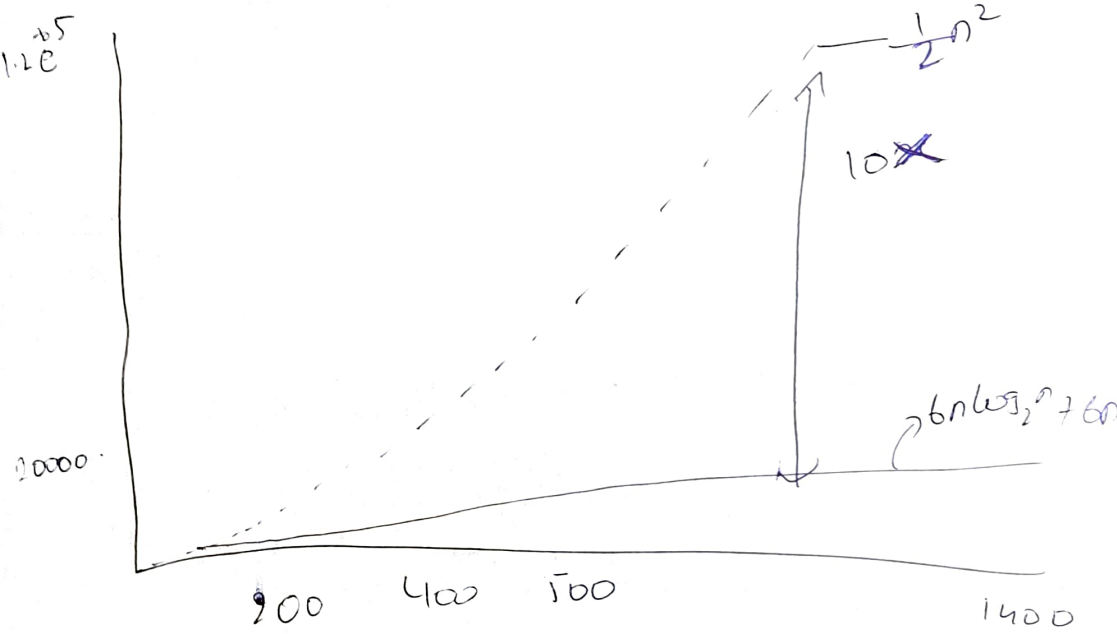
Mathematically, the running time of merge sort, $6N\log_2 N + 6N$ is BETTER than any function which has quadratic dependence on N $\rightarrow (N^2)$

Even like: $\frac{1}{2}n^2$ ~ insertion sort

This mathematical statement is only true if N is sufficiently large.

Justification:

Only big problems are interesting



For small problem sizes, use the algorithm with smaller const factors like insertion sort.

For larger problems, use algorithm with better rate of growth, mainly merge sort.

Fast algorithm ≃ worst-case running time grows slowly with input size

Sweet spot

We need fast algo with good asymptotic run time which grows slowly with the input size.

Holy grail: Linear running time

An algo whose number of instructions grows proportionally to the input size.

Week 1.9 - Asymptotic Analysis

The Gist:

Motivation.

Importance: Vocabulary for the design and Analysis of Algorithms (e.g. "Big-Oh" notation"
- "sweet spot" for high-level reasoning about algorithms

- Coarse enough to suppress architecture/language/compiler-dependent details.
- Sharp enough to make useful comparisons between different algorithms, especially on larger inputs

For ex: Asymptotic analysis will allow us to differentiate between better and worse approach to sorting, integer multiplication

High-Level - Idea: Suppress Constant factors
- Lower-order terms
  - ↳ irrelevant for large inputs
  - ↳ too system dependent

Example: Equate $6n\log_2 n$ with just $n\log n$

$6n \rightarrow$ lower-order terms, grows more slowly than $n\log n$

$6 \rightarrow$ leading constant factor
- ↳ we suppress these

Left with $n\log n$

Terminology: running time is $O(n\log n)$

("big-Oh" of $n\log n$)

where n - input size (eg. length of input array)

means, after we dropped the lower order terms, suppressed the leading constant factor, we are left with function $f(n)$

Example : One loop

Problem: does array A contain the integer t?

given A: array of length n and t (an integer)

$O(n)$

For two separate loops

↳ $O(n)$

For two nested loop

(e.g. $O(n^2)$ → quadratic time algorithm

$(n)(n-1)$

Week: 1.10 → Big-Oh: Notation

Big-Oh notation concerns functions defined on the positive integers

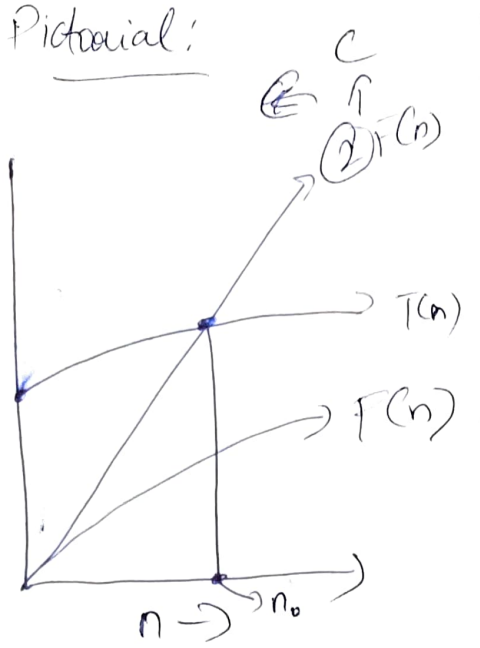
Let $T(n)$ - function on $n = 1, 2, 3, \ldots$

Usually, the worst-case running time of an algorithm

When is $T(n) = O(f(n))$?

If eventually for all sufficiently large n, $T(n)$ is bounded above by a constant multiple of $f(n)$

Pictorial:



Picture: $T(n) = O(f(n))$

When we ~~see~~ [illegible]
right, →

Formal Def:

$T(n) = O(f(n))$

If & only if there exist constants $c, n_0 > 0$ such that

$T(n) \leq c \cdot f(n)$ → $(c f(n))$

for all n that are equal or exceed n_0. → For all $n \geq n_0$

n_0 → crossing point.

Warning: c, n_0 independent of n

Basic Examples:

Claim: If $T(n) = a_k n^k + \dots a_1 n + a_0$. Then $T(n) = O(n^k)$

Proof: Choose $n_0 = 1$ & $c = |a_k| + |a_{k-1}| + \dots |a_1| + |a_0|$

Need to show that for all $n \geq 1$, $T(n) \leq c \cdot n^k$

$$T(n) \leq |a_k| n^k + \dots |a_1| n + |a_0|$$
$$\leq |a_k| n^k + \dots |a_1| n^k + |a_0| n^k$$
$$\leq c \cdot n^k$$

Example #2

Claim: For every $k \geq 1$, n^k is not $O(n^{k-1})$

Proof: by contradiction. Suppose $n^k = O(n^{k-1})$

Then, (two) 2 constants, a winning strategy,

$c, n_0 > 0$ such that

$$n^k \leq c \cdot n^{k-1} \quad \forall n \geq n_0$$

[cancelling n^{k-1} from both sides]:

$$n \leq c \quad \forall n \geq n_0$$

~~3n~~ 6n

$n^{1/2}$ $n^{5/3}$

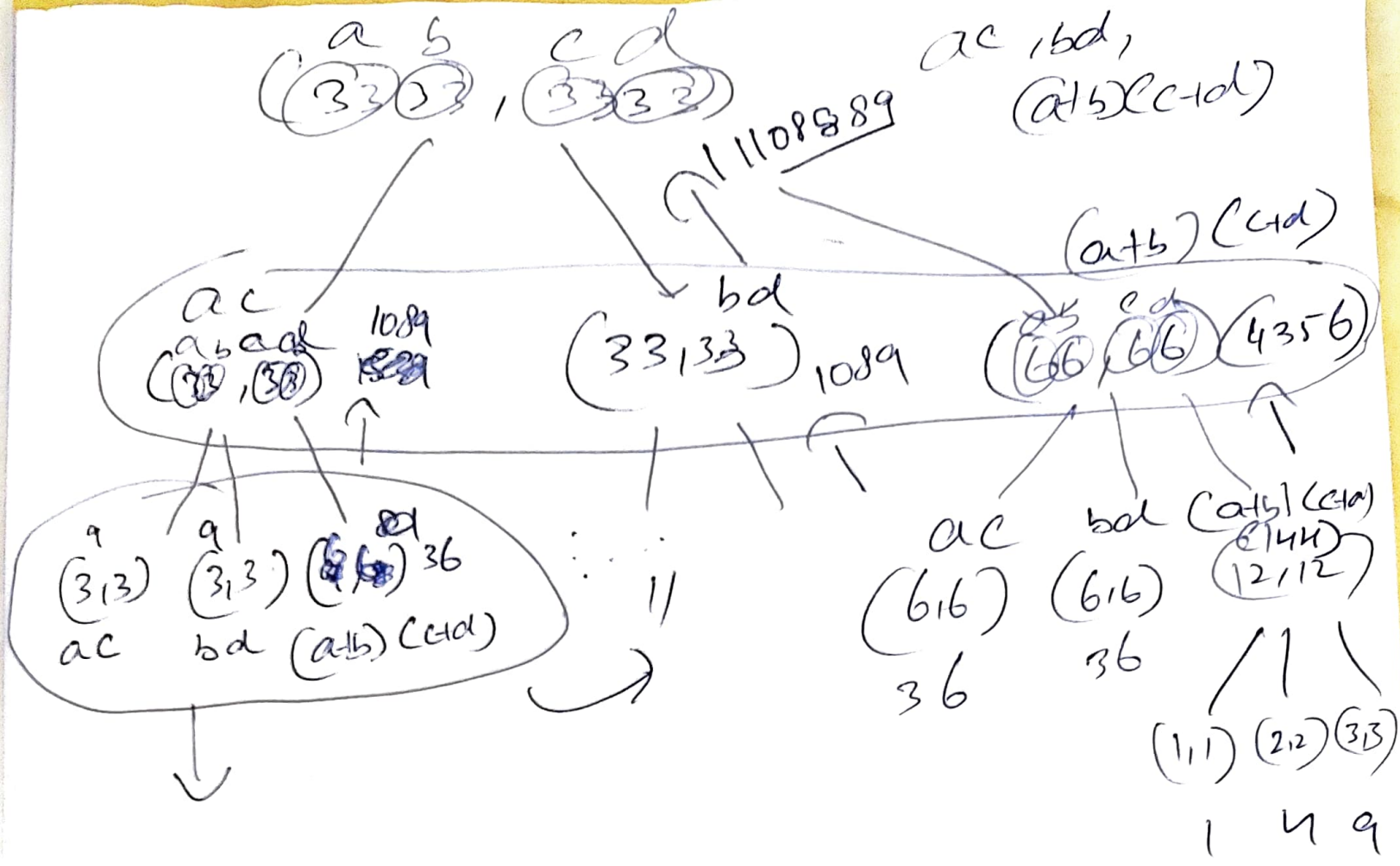
$$f(n) = O(g(n))$$

$$f(n) * \log_2(f(n)) = O(g(n) \log_2(g(n)))$$

$\leq \sqrt{n} \leq 2^{\sqrt{\log(n)}} \leq n^{1.5}$ $n^{5/3} \leq 10n$

a) 256 , b) 512, $1.34e^{154}$, d.) 1036 , e) 81

* e , a, b, d, c


a b
c d
(3 3)(3 3), (3 3)(3 3)
ac, bd, (a+b)(c+d)
1110889
(a+b)(c+d)
ac
bd
1089
(3 3, 3 3)
1089
(6 6, 6 6)
4356
ac
bd
(a+b)(c+d)
6144
12 12
(3,3)
(3,3)
ac
bd
(a+b)(c+d)
36
11
(6,6)
36
(6,6)
36
(1,1)
(2,2)
(3,3)
1
4
9