~~Chapter 01~~

Lecture-01 (MIT Open)

Introduction- Analysis of Algorithms, Insertion Sort, Mergesort

Instructors - Erik Demaine, Charles Leiserson

Prerequisite

⇒ Discrete Mathematics, Probability, Programming Experience

Analysis of Algorithms:

Theoritical study of computer program performance and resourse usage

What's more important than perf?

- ✓ ~~programmer time~~
- ✓ correctness
- ✓ simplicity
- ✓ maintainability
- ✓ ~~Stabi~~ Robustness
- ✓ functionality
- ✓ Modularity
- Security
- Scalabity
- ✓ user-friendliness

extensibility
reliability

Why study algos and performance then?

→ real-time constraints

→ user-friendliness

→ measures the line between feasible and non-feasible (takes too much time or too much memory)

→ gives a language to talk about ~~gro~~ program behavior

→ performance is the currency of computing ~~all these~~

Sorting: Input: A sequence $\langle a_1, a_2, a_3, \ldots a_n \rangle$ of

Output: permutation of that number

$\langle a_1', a_2', a_3' \ldots\ldots, a_n' \rangle$

$ (such that)

$a_1' \leq a_2' \leq \ldots \leq a_n'$ (monotonically increasing)

Insertion sort:

Insertion sort (A, n) // Sorts $A[1 \ldots n]$

```
for j ← 2 to n
    do key ← A[j]
       i ← j-1
       while i>0 and A[i]>key
           do A[i+1] ← A[i]
              i ← i-1
       A[i+1] ← key
```

Running time:

→ Depends on input order

→ Depends on input size
- parameterize in input size

→ Want upper bounds
- guarantee an user

Kinds on analysis

Worst case ~~(analysis)~~ (usually)

$T(n)$ = max time on any input of size n

Average case (sometimes)

$T(n)$ = expected time over all inputs of size n

= time of every input * probability that will be that input

(Need assumption of statistical distribution)

Best case (bogus)

works good for some inputs (change to cheat)

~~Insertion sort~~ Worst Case Scenario

→ Depends on computer [relative speed on same machine]

→ absolute speed (on different machine)

There comes the idea of asymptotic analysis

i) Ignore machine dependent constants

ii) Look at the growth of $T(n)$ as $n \to \infty$

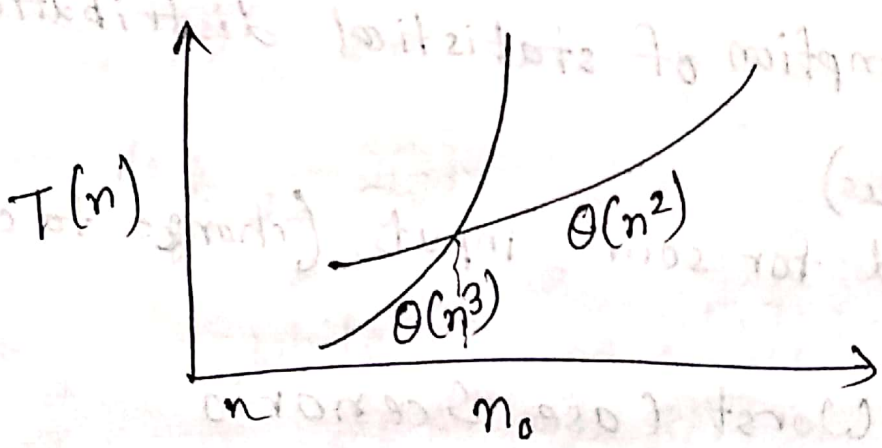
[Now we can compare both relative and absolute speed]

Aysmptotic Notation:

Θ-notation → Drop low-order terms
Ignore leading constants

Ex: $3n^3 + 9n^2 + 5n + 6046 = \Theta(n^3)$

As $n \to \infty$ $\Theta(n^2)$ always beats $\Theta(n^3)$



Insertion sort analysis

Worst case: input reverse sorted

$$T(n) = \sum_{j=2}^{n} \Theta(j) \quad \text{[according to pseudocode]}$$

$$= \Theta(n^2) \quad \text{[arithmatic series]}$$

Is insertion sort fast?
Moderately fast for small n
Not at all for big n

Merge Sort:

Key subroutine "MERGE"

MERGE does $A[1 \ldots n]$.

1) If $n=1$, done
2) Recursively sort $A[1 \ldots \lceil n/2 \rceil]$ and $A[\lceil n/2 \rceil + 1 \ldots n]$
3) "Merge" 2 sorted" list $\Theta(n)$

$T(n)$
$\Theta(1)$
$2T(n/2)$
$\Theta(n)$

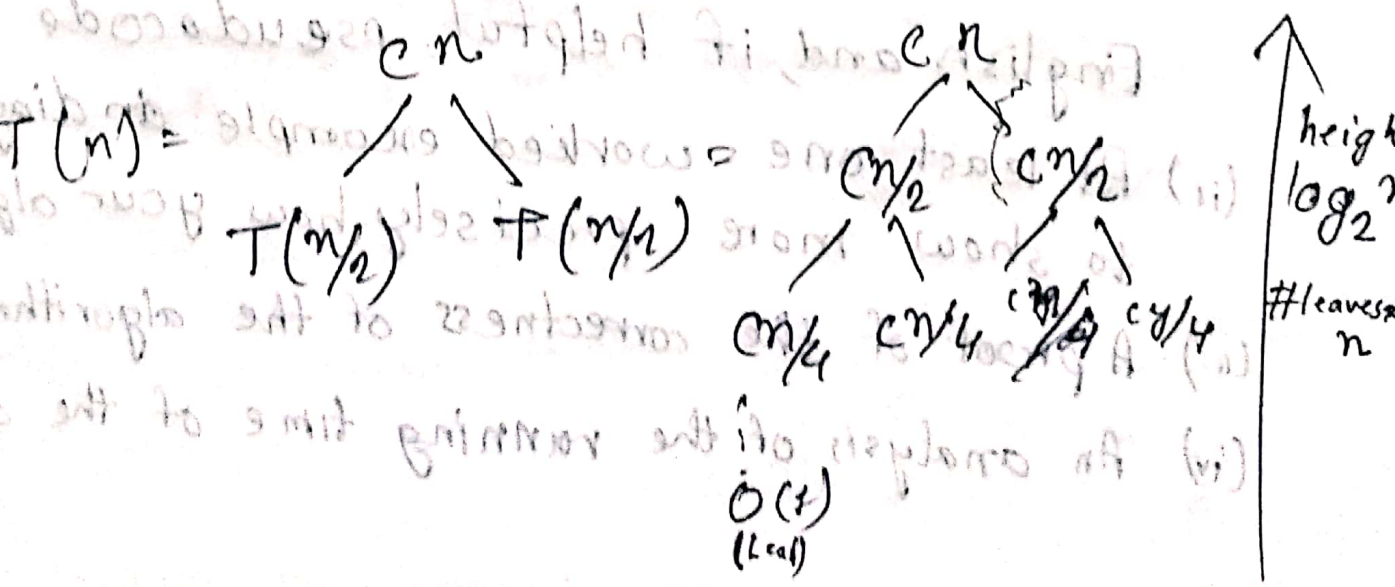
~~$T(n) = \Theta(1) + 2T(n/2)$~~

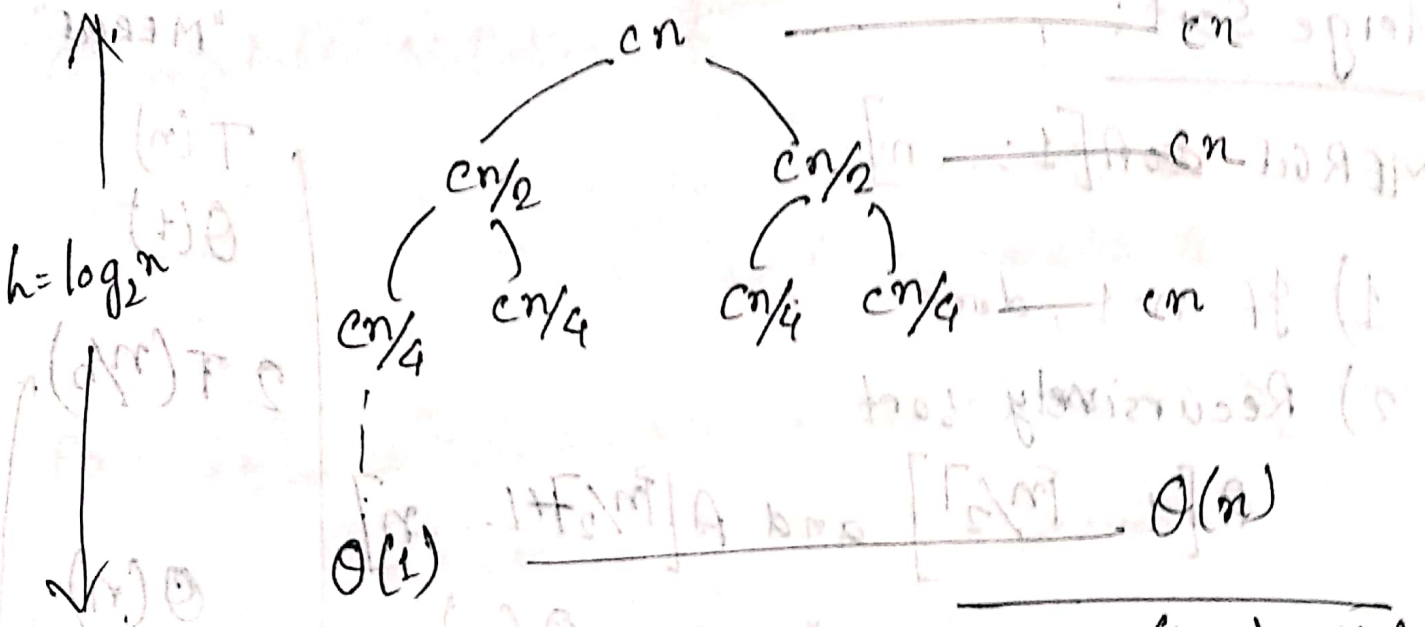
Sloppiness $\Rightarrow$ $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$ asymptotically doesn't matter

Recurrence:

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

Recursion tree: $T(n) = 2T(n/2) + cn$ cost c





Total = $cn(\log_2 n) + \theta(n)$

$= \theta(n \log_2 n)$

Faster than $\theta(n^2)$ insertion sort

→ for a large enough dataset ~~ins~~ merge sort beats insertion sort

How to provide an algorithm?

(i) A description of the algorithm in English and, if helpful, pseudocode

(ii) At least one worked example or diagram to show more precisely how your algo works

(iii) A proof of the correctness of the algorithm

(iv) An analysis of the running time of the algo

Lecture 2: Asymtotic Notation; Recarrences, Substitution, Master Method

Asymptotic Notation:

O-notation

$f(n) = \Theta(g(n))$ means there are consts $c > 0$, $n_0 > 0$ such that

$$0 \leq f(n) \leq cg(n)$$

for all $n \geq n_0$

Ex: $2n^2 = O(n^3) \Rightarrow 2n^2 \in O(n^3)$

↑ one way →

Set definition of O-notation:

$O(g(n)) = \{f(n)$: there exist consts $c > 0, n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$

~~$2n^2 \in O(n^3)$~~

Macro substitution (convention)

A set in a formula represents an anonymous function in the set

Example: $f(n) = n^3 + O(n^2)$

There is a function $h(n) = O(n^2)$ such that

$$f(n) = n^3 + h(n)$$

Ex: $n^2 + O(n) = O(n^2)$

means for any $f(n) \in O(n)$ there is an $h(n) \in O(n^2)$ such that $n^2 + f(n) = h(n)$

Ω notation (lower bound)

$\Omega(g(n)) = \{f(n)$: there exist constants $c > 0$, $n_0 > 0$ such that $0 < c\,g(n) < f(n)$ for all $n > n_0\}$

$\sqrt{n} = \Omega(\log_2 n)$

Analogy

$O \quad \Omega \quad \Theta \quad o \quad \omega$

$\leq \quad \geq \quad = \quad < \quad >$

Θ-notation

$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

~~$n^2 + \Theta(n)$~~ $n^2 + O(n) = \Theta(n^2)$

Strict notations: o-notation and ω-notation

$o(g(n)) = \{f(n)$: for any constant $c > 0$, there is a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$

$\omega(g(n)) = \{f(n)$: for any constant $c > 0$, there is a constant $n_0 > 0$ such that $0 < cg(n) < f(n)$ for all $n \geq n_0\}$

$2n^2 = O(n^3)$ $[n_0 = 2/c]$

$\sqrt{n} = \omega(\log_2 n)$ $[n_0 = 1 + 1/c]$

$\frac{1}{2}n^2 = \Theta(n^2) \neq o(n^2)$

Solving recurrences:

Substitution method:

1) Guess the form of the solution
2) Verify by induction
3) Solve for constants

Example:

$T(n) = 4T(n/2) + n$ $\quad O(n^2)$

$[T(1) = \theta(1)]$

Guess $T(n) = O(n^3)$

Assume $T(k) \leq ck^3$ for $k < n$ [we need to write out the constant]

$$T(n) = 4T(n/2) + n$$
$$\leq 4c(n/2)^3 + n$$
$$= \frac{1}{2}cn^3 + n$$
$$= cn^3 - \left(\frac{1}{2}cn^3 - n\right)$$
$$\leq cn^3 \text{ if } \frac{1}{2}cn^3 - n \geq 0 \text{ eg } c \geq 1, n \geq 1$$

Base $T(1) = \theta(1) \leq c$ (if c is sufficiently large)

For $1 \le n < n_0$ we have "$O(1)$" $\le cn^3$, if we pick c big enough.

[The bound is not tight]

8. 4/10/20 (29)

$T(n) = O(n^2)$?

Assume $T(k) \le ck^2$ for $k < n$:

$$T(n) = 4T(n/2) + n$$
$$\le 4c(n/2)^2 + n$$
$$= cn^2 + n$$
$$\not\le cn^2 \text{ for no choise of } c > 0 \text{ lose}$$

$\Rightarrow T(n) = O(n^2) \longrightarrow$ wrong

$= cn^2 - (-n)$

To strengthen the hypothesis -> substract a low-order term

Assume $T(k) \le c_1 k^2 - c_2 k$

$$T(n) = 4T(n/2) + n$$
$$= 4\left(c_1(n/2)^2 - c_2(n/2)\right) + n$$
$$= c_1 n^2 + (1 - 2c_2)n$$
$$= c_1 n^2 - c_2 n - \underbrace{(-1 + c_2)n}_{\text{residiul} \ge 0 \text{ if } c_2 \ge 1}$$

$\leq c_1 n^2 - c_2 n$

c_1 needs to be larger than c_2

$T(1) \leq c_1 - c_2$, if c_1 sufficiently large

$T(1) = \theta(1)$

Recursion Treee Method:

⇒ A recursion tree models the cost (time) of a recursive execution of an algorithm

⇒ The recursion-tree method can be unreliable, just like any method that uses elipses

⇒ The recursion-tree promotes intuition however

⇒ The recursion-tree method is good for generating guesses for substitution method

Lec-03

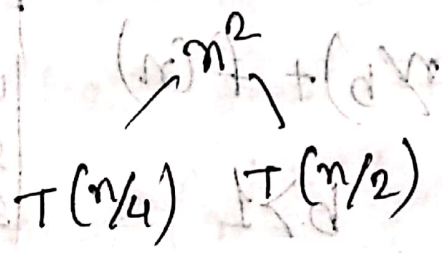
05/10/2020

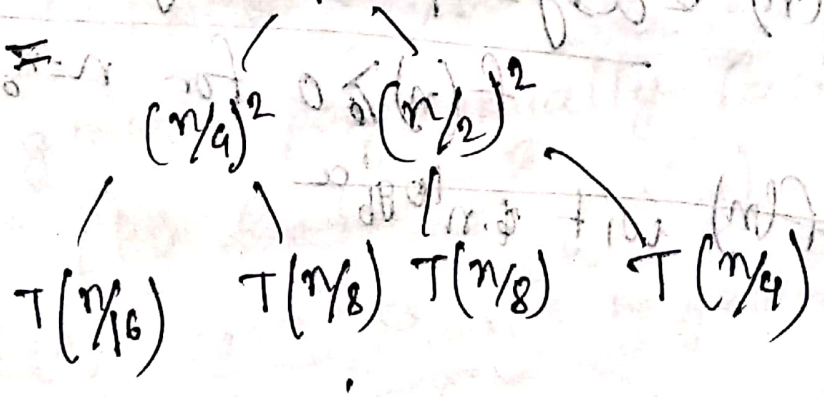
Quicksort, Median

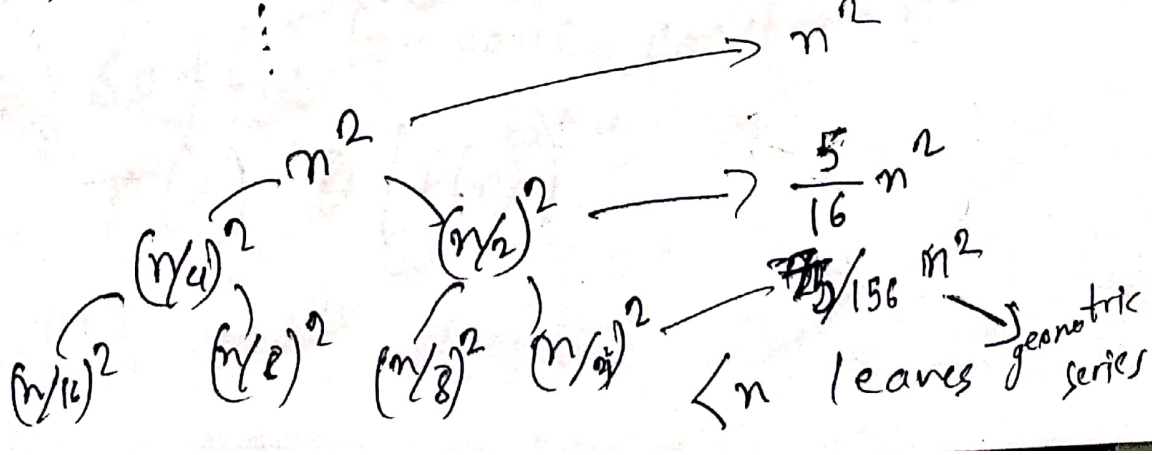
05/10/2020 (38)

- Find out what the answer is with recursion-tree method. Then prove that it actually right with the substitution method.

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







Total = (level-by-level)

$$\leq \left(1 + \frac{5}{16} + \frac{25}{256} + \frac{5^k}{16^k}\right) n^2$$

$$\leq 2n^2$$

$$= O(n^2)$$

Master Method:

applies to the recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$, $b > 1$

every problem we recurse on should be of same size

$f(n)$ = asymptotically positive

$f(n) \geq 0$ for $n = n_0$

~~Compare $f(n)$ with $n^{\log_b a}$~~

AE Lab01

07/10/2020

Travelling problem (20 pages)

Three common cases of master method (49)

1) $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$
→ $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor)
Solution: $T(n) = \Theta(n^{\log_b a})$

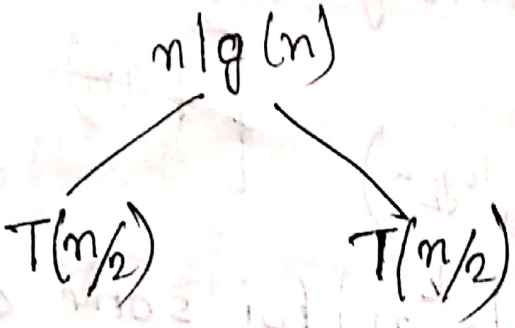
2) $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$
→ $f(n)$ and $n^{\log_b a}$ grow at similar rates
Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$

3) $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.
→ $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor)
and $f(n)$ satisfies the regularity condition that $af(n/b) \leq cf(n)$ for some constant $c < 1$
Solution: $T(n) = \Theta(f(n))$

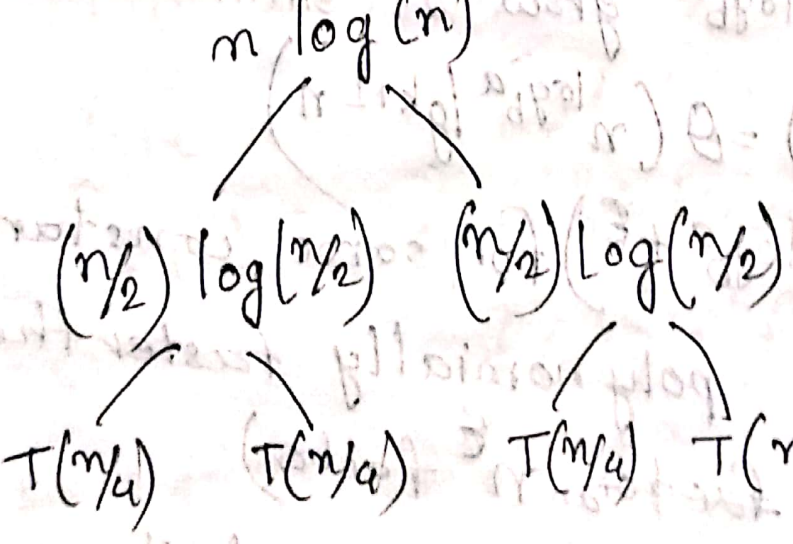
Assignment 1

1) $T(n) = 2T(n/2) + n\lg(n)$

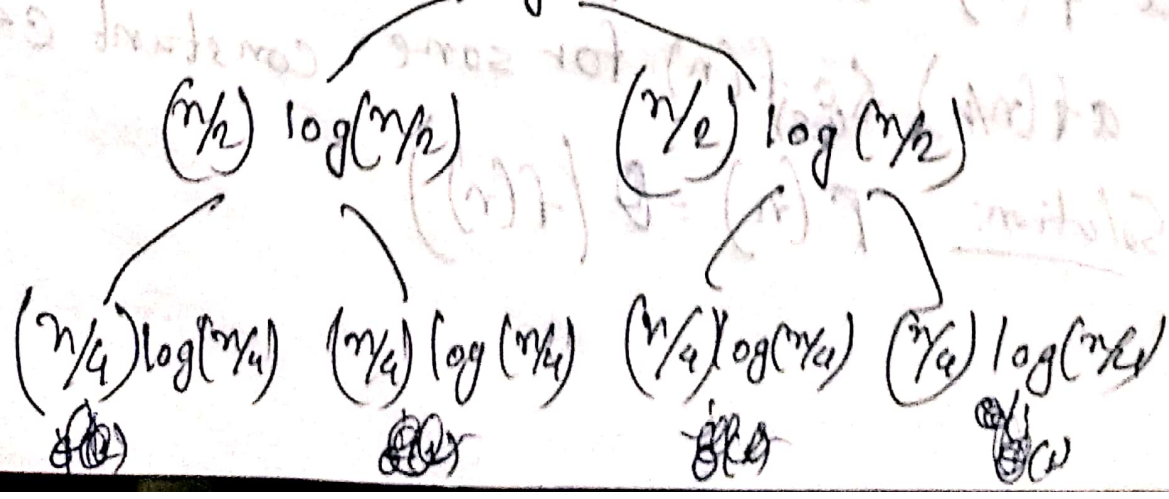
Recursion Method Expansion



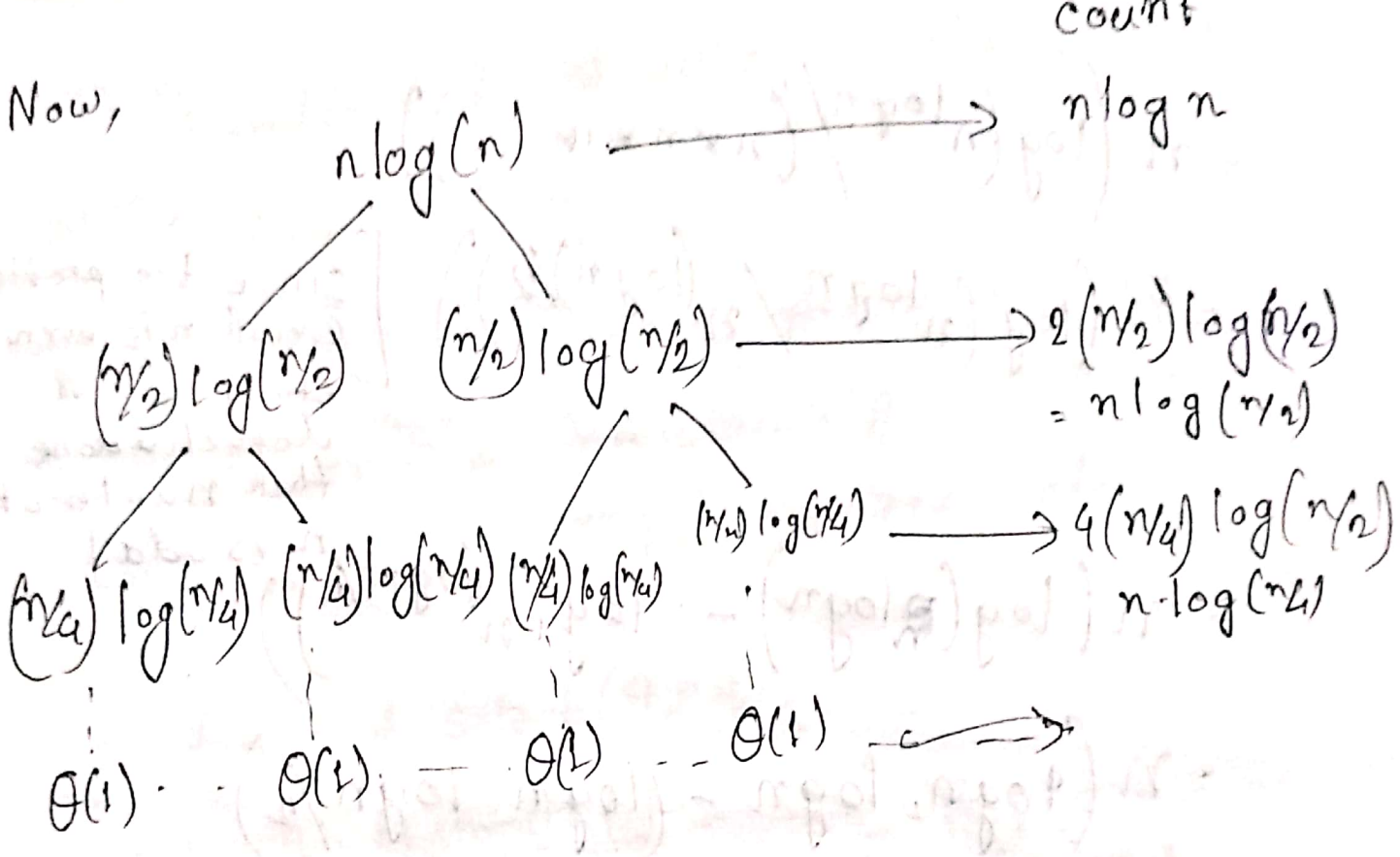
=



=



Now,



Total level-by level iterations $\leq$

$$n\log n + n\log(n/2) + n\log(n/4) + \ldots\ldots$$

$$= n\left(\log n + \log(n/2) + \log(n/4) + \ldots + \log 2\right) \quad [+\log(1) = 0]$$

$$= n\left(\log\left((n * n * n * \ldots * n)/2/4/8 \ldots /n\right)\right)$$

$$= n\left(\log\left(n^{\log n}/2/4/8/\ldots/n\right)\right) \quad [\log n \text{ denotes maximum level in binary tree}]$$

$$= n\left(\log\left(n^{\log n}/(1*2*4*8*\ldots*n/4*n/2*n)\right)\right)$$

$$= n\left(\log\left(n^{\log n}\left((1*n)*(2*n/2)*(4*n/4)\ldots\right)\right)\right)$$

$$= n\left(\log\left(n^{\log n}/(n*n*n*\cdots)\right)\right)$$

$$= n\left(\log\left(n^{\log n}/n^{(\log n)/2}\right)\right)$$

[since the ~~previous~~ overall n is even if we detach one then number of n is odd]

$$= n\left(\log\left(n^{\log n}\right) - \log\left(n^{(\log n)/2}\right)\right)$$

$$= n\left(\log n \cdot \log n - (\log n \cdot \log n)/2\right)$$

$$= n(\log n)^2/2$$

$$= O(n(\log n)^2)$$ [big-O doesn't care about constant]

[Lecture 2 continuation]

Ex 1 $T(n) = 4T(n/2) + n$

$a \quad b \quad f(n)$

$n^{\log_b a} = n^2$ [bigger than n]

Case 1: $T(n) = \theta(n^2)$ [$f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$

Ex 2 $T(n) = 4T(n/2) + n^2$

Case 2: $f(n) = \theta(n^2 \lg^0 n)$ that is $k = 0$

$T(n) = \theta(n^2 \lg n)$

Ex 3 $T(n) = 4T(n/2) + n^3$

Case 3: $f(n) = \Omega(n^{2+\varepsilon})$ where $\varepsilon = 1$

and $4(n/2)^3 \leq cn^3$ for $c = 1/2$

$T(n) = \theta(n^3)$

Ex 4: $T(n) = 4T(n/2) + n^2/\log n$

$T(n) = n^2 \log(\log n)$ [cannot be solved with masters method]

(Expand)

12/10/2020

Lec-4

Dynamic Programming

Divide and Conquer

Recursive

Bellman equation

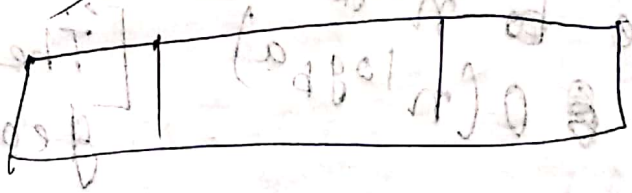
1) understand the structure of the problem
2) Recursive soln formulation optimally
3) Calculate the stores remembers the best point (memoization
4) Reconstruct the soln

Optimal Substructure.

Pg. 360

$$\max_{k=1}^{n-1} \left(P(n) + P(n-k) \right)$$

rod cutting



Matrix-chain multiplication

Viterbi, BDTW

[Lecture 3] 13/10/2020 (6)

Divide and Conquer

(Binary Search, Powering a number, Fibonacci numbers, Matrix Multiplication, Strassen's algo, VLSI tree layout)

Recall: Master Theorem $T(n) = aT(n/b) + f(n)$

Case 1: $f(n) = O\left(n^{\log_b a - \varepsilon}\right) \Rightarrow T(n) = \Theta\left(n^{\log_b a}\right)$

Case 2: $f(n) = \Theta\left(n^{\log_b a} \lg^k n\right) \Rightarrow T(n) = \Theta\left(n^{\log_b a} \lg^{k+1} n\right)$

Case 3: $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ & $af(n/b) \leq (1-\varepsilon) f(n) \Rightarrow T(n) = \Theta\left(f(n)\right)$

Divide and conquer / Divide and rule / divide et impera

Tried and tested way of conquering a land by dividing it into sections, ~~of some kind of~~ Then making them no longer like each other. Then conquer little by little and conquer ultimately conquer them all. This was practice by the British

This is an algorithm design technique which is very basic and powerful

18/10/2020

Lec-5

Dynamic Programming

MCM

Combinatorial prob

Catalan Number

Parenthesis

DTW (Dynamic Time Warping)

Fast DTW

Viterbi: MM, HMM

19/10/2020

Lec-6

34) NP Completeness (Decision Probs)

Halting Problem

Reduction

SAT (satisfiability prob)

0-1 knapsack

[Lecture3 Cont]

19/10/2020
(6)

Running time of merge sort

$$T(n) = 2T(n/2) + \Theta(n)$$

Case 2 $\Rightarrow \Theta(n \lg n)$

~~Beas~~

Binary search: find (x) in sorted array

1. Divide [Compary x with middle element]
2. Conquer [Recurse in one subarray]
3. Combine : trivial

$$T(n) = T(n/2) + \Theta(1)$$

Case 2 $\Rightarrow \Theta(\lg n)$

Powering a number: given number x, integer $n \geq 0$, compute x^n

Naive algo $\Theta(n)$

~~Dan~~ Divide and conquer

$$x^n = \begin{cases} x^{n/2} \cdot x^{n/2} & \text{if } n \text{ even} \\ x^{\frac{n-1}{2}} \, x^{\frac{n-1}{2}} \cdot x & \text{if } n \text{ odd} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1)$$
$$= \Theta(\lg n)$$

Fibonacci number:

$$F_n = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ f_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

Naive Recursive algo

Time $\Omega(\phi^n)$

Exponential time

$\phi = \frac{1+\sqrt{5}}{2}$ [Golden Ratio]

Bottom up algorithm

Compute $F_0, F_1, F_2 \ldots F_n$ (storing in an array

Time $\Theta(n)$

Naive Recursive squaring:

$F_n = \phi^n / \sqrt{5}$ rounded to the nearest integer

[not allowed]

Doesn't work well in machines for roundoff error

Recursive Squaring

Thm:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

Time $\Theta(\lg n)$

Proof: by induction

Base: $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}$

Step:

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

Matrix Multiplication:

Inp: $A = [a_{ij}]$ $B = [b_{ij}]$ $i,j = 1, 2, \ldots n$

Out: $C = [c_{ij}] = AB$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Standard alg $\Theta(n^3)$

```
for i ← 1 to n
do for j ← 1 to n
     do cij ← 0
        for k ← 1 to n
           do cij ← cij + aik · bkj
```

Divide and Conquer

Idea: $n \times n$ matrix

= 2×2 block matrix

of $n/2 \times n/2$ sub matrices

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

C = A · B

$r = ae + bg$
$s = af + bh$
$t = ce + dg$
$u = cf + dh$

8 mults of $(n/2) \times (n/2)$ submatrices

4 adds of $(n/2) \times (n/2)$ submatrices $\Theta(n^2)$

$$T(n) = 8\,T(n/2) + \Theta(n^2)$$

Case 1: $\Theta(n^{\log_b a}) = \Theta(n^3)$

No better than previous

This Divide and Conquer didn't work

16/10/2020

Strassen's algorithm:

Idea: Multiply 2×2 matrices with only 7 recursive mults

$P_1 = a \cdot (f - h)$

$P_2 = (a+b) \cdot h$

$P_3 = (c+d) \cdot e$

$P_4 = d \cdot (g - e)$

$P_5 = (a+d) \cdot (e+h)$

$P_6 = (b-d) \cdot (g+h)$

$P_7 = (a-c) \cdot (e+f)$

$r = (a+d)(e+h) + d(g-e) - (a+b)h$
$+ (b-a)(g+h)$

$= ae + ah + de + dh + dg$
$- de - ah - bh + bg + bh - dg - dh$

$= ae + bg$

$a = (a+d)(e+h) + a(f-h) -$
$(c+d)e - (a-c)(e+f)$

$= ae + de + ah + dh + af - ah$
$ce + de - ae + ce - af + cf$

$= af + dh$

$r = P_5 + P_4 - P_2 + P_6$

$s = P_1 + P_2$

$t = P_3 + P_4$

$u = P_5 + P_1 - P_3 - P_7$

$\therefore s = a(f-h) + (a+b)h$

$= af - ah + ah + bh$

$= af + bh$

7 mults, 18 adds/subs

No dependence on commutativity of mult

$\Longleftrightarrow$

$r = ae + bg$

$s = af + bh$

$t = ce + dg$

$u = cf + dh$

$t = (c+d)e + d(g-e)$

$= ce + de - de + dg$

$= ce + dg$

D&C

1) Divide: Partition A and B into $(n/2) \times (n/2)$ submatrices. Form terms to be multiplied using + and - $\Theta(n^2)$

2) Conquer: Perform 7 multiplications of $(n/2) \times (n/2)$ submatrices recursively

3) Combine: Form C using + and - on $(n/2) \times (n/2)$ submatrices $\Theta(n^2)$

$\Rightarrow T(n) = 7T(n/2) + \Theta(n^2)$

$T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.81})$ [Case 1]

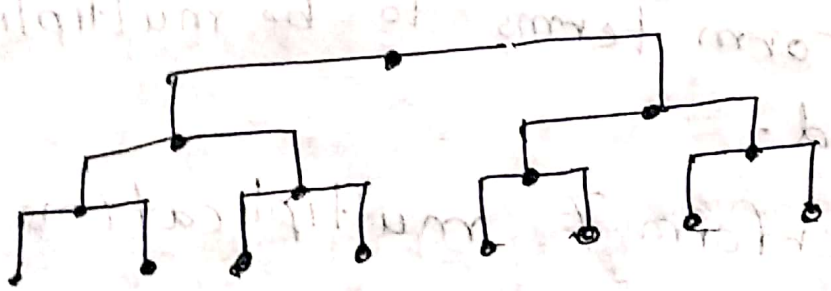
The number 2.81 may not seem much smaller. But in runtime Strassen's algo beats ordinary by $n \geq 32$

Best to date: $\theta(n^{2.376\ldots})$ [theoritically]

VLSI layout:

Problem: Embed a complete binary tree on n leaves in a grid with minimum area

Naive embedding



$H(n) = H(n/2) + \theta(1)$ (height)

$= \theta(\lg n)$ (case 2)

$W(n) = 2W(n/2) + \theta(1)$ (width)

$= \theta(n)$ (case 1)

$\text{Area} = \theta(n \log n)$

H-Tree Embedding

$\log_4 2 = \frac{1}{2}$

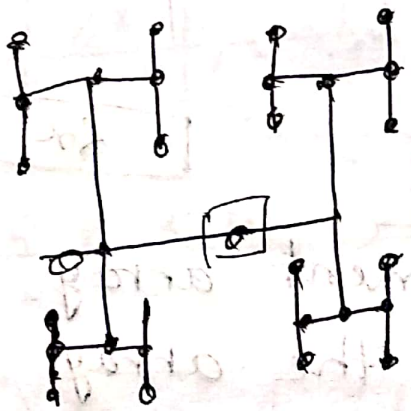
$W(n) = \theta(\sqrt{n})$

$H(n) = \theta(\sqrt{n})$

$A(n) = \theta(n)$

$T(n) = 2T(n/4) + \theta(n^{1/2-\varepsilon})$ goal

$= 2T(n/4) + \theta(1) = \theta(\sqrt{n})$ case 1



Lecture 4
Quicksort

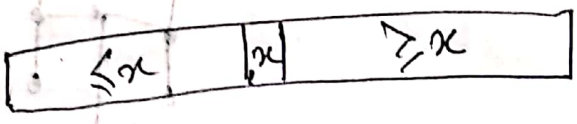
20/10/2020

- Divide and Conquer
- Partitioning
- Worst case analysis
- Intuition
- Randomized Quicksort
- Analysis

- Proposed by Hoare 1962
- D&C
- Sorts "in place" (like insertion sort, not like merge)
- Very practical (with tuning)

R

Divide and Conquer:



Quicksort an n-element array

1) Divide: Partition the array into two subarrays around a pivot x such that elements in lower subarray ≤ x ≤ elements in upper subarray

2) Conquer: Recursively sort the two sub-arrays

3) Combine: Trivial

⇒ Key: Linear Time Θ(n) Partitioning Subroutine

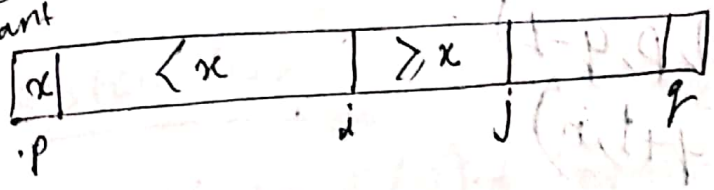
Quicksort ⇒ Recursive Partitioning

Mergesort ⇒ Recursive merging

```
Partition(A,p,q)   // A[p..q]
                   // pivot A[p]
x ← A[p]
i ← p
for j ← p+1 to q
   do if A[j] ≤ x
         then i ← i+1
              exchang  A[i] ↔ A[j]
exchange A[p] ↔ A[i]
return i
```

Running time
= $O(n)$ for n elements

Invariant



~~x: 6, 10, 13, 5, 8, 3, 2, 11,~~

(x ← 6)

6, (10), 13, (5), 8, 3, 2, 11
i j ⟷ j

6, 5, (13), 10, 8, 3, (2), 11
   i   j

~~6, 3, 13, 10, 8, 5, 2, 11~~
   i   j

~~6, 3,~~

6, 5, 3, (10), 8, 13, (2), 11
      i   j

(6) 5, 3, (2) 8, 13, 10, 11 (j gets out of the array)
i j

2, 5, 3, 6, 8, 13, 10, 11
≤ pivot | pivot | ≥ pivot

```
Quicksort (A, p, r)
    if p<r
    then q ← Partition (A, p, r)
         Quicksort (A, p, q-1)
         Quicksort (A, q+1, r)
```

Initial call Quicksort (A, 1, n)

Analysis

- Assume all input elements are distinct
- In practice, there are better partitioning algorithms for when duplicate input elements may exist
- T(n) = worst case running time on an array of n elements

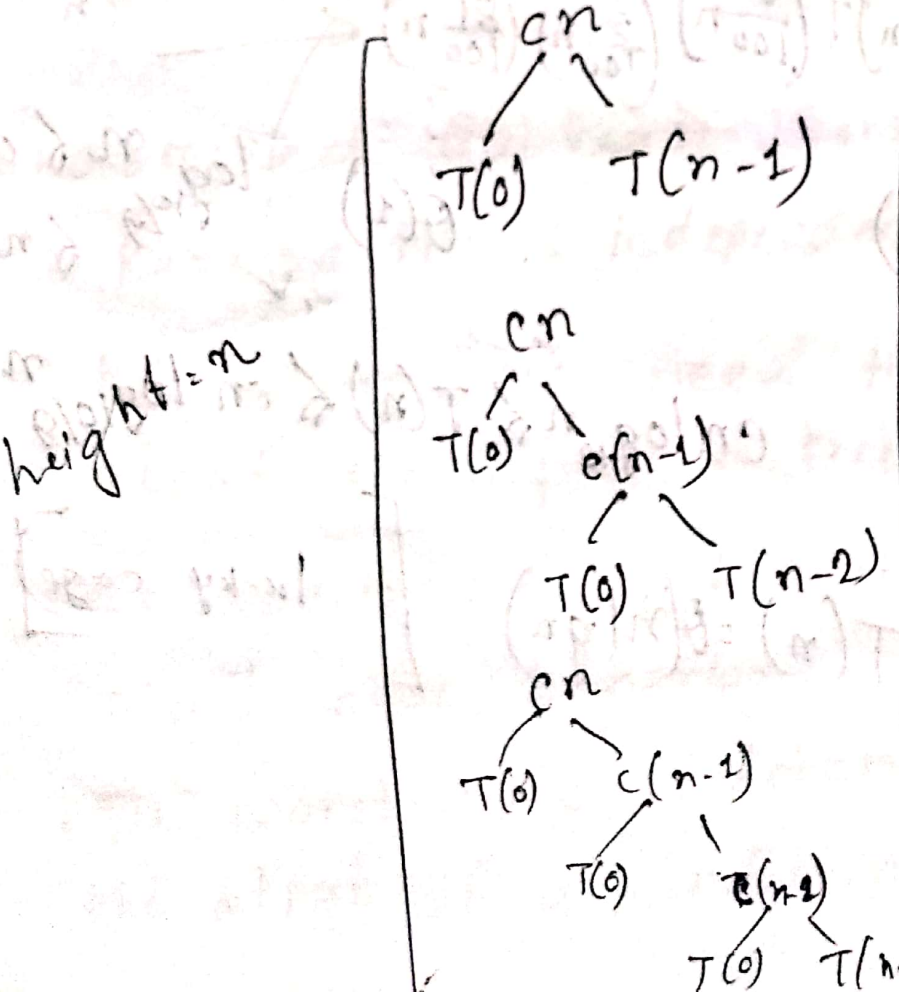
- Input sorted or reverse sorted
- Partition around min or max element
- One side of partition always has no elements

$$T(n) = T(0) + T(n-1) + \Theta(n)$$
$$= \Theta(1) + T(n-1) + \Theta(n)$$
$$= \Theta(n^2) \text{ [arithmetic series]}$$

|| like insertion sort

Recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



height = n

$$T(n) = \Theta(n) + \Theta(n^2)$$
$$= \Theta(n^2)$$

$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

Best Case Analysis: (intuition only)

If we're lucky, Partition splits the array evenly $(n/2 : n/2)$

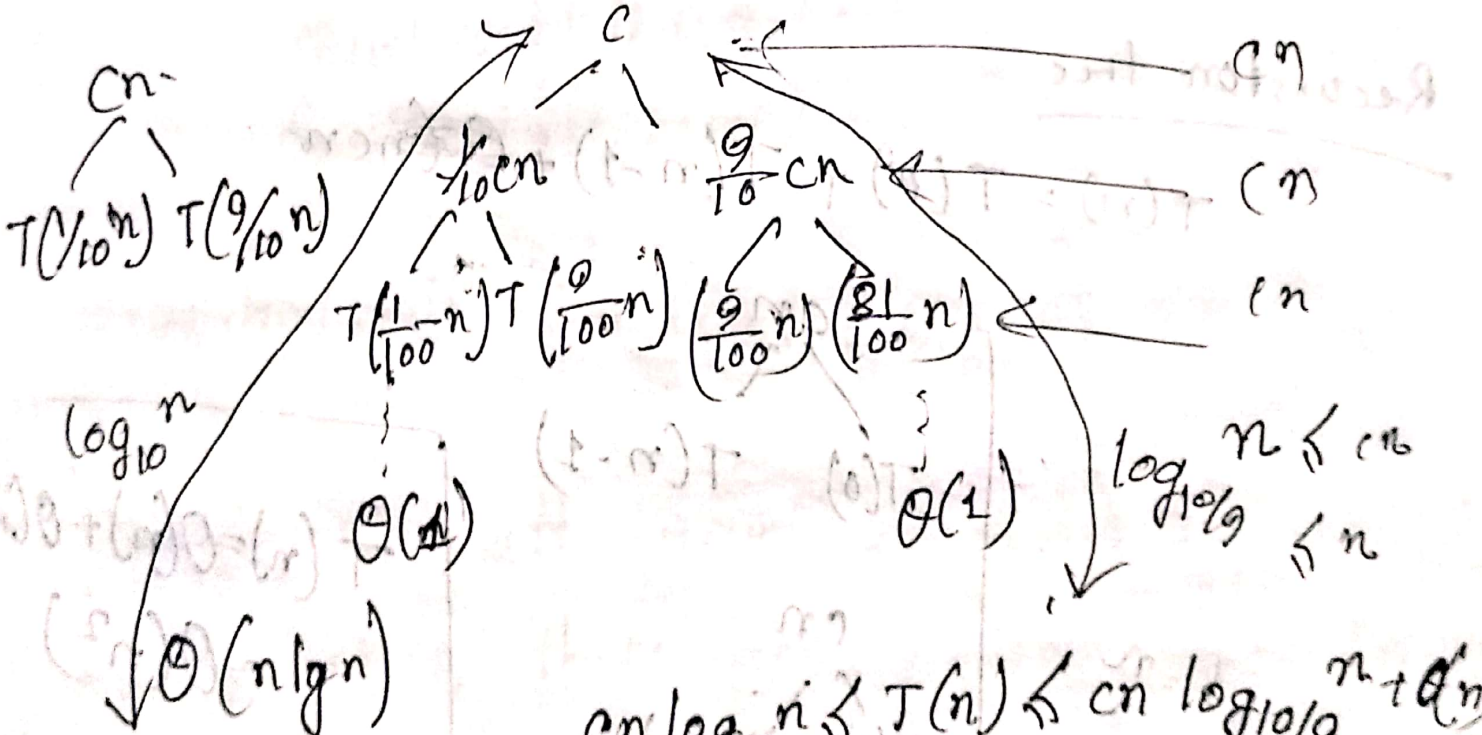
$$T(n) = 2T(n/2) + \Theta(n)$$

$$= \Theta(n \lg n)$$ [same as mergesort]

Case 2: Master

What if the split is always $\frac{1}{10} : \frac{9}{10}$?

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n)$$



$$cn \log_{10} n \leq T(n) \leq cn \log_{10/9} n + \Theta(n)$$

$$T(n) = \Theta(n \lg n)$$ [lucky case]

• Suppose we alternate lucky, unlucky, lucky, unlucky, ...

$L(n) = 2U(n/2) + \Theta(n)$ ← lucky

$U(n) = L(n-1) + \Theta(n)$ ← unlucky

$L(n) = 2(L(n/2 - 1) + \Theta(n/2)) + \Theta(n)$

$= 2L(n/2 - 1) + \Theta(n)$

$= \Theta(n \lg n)$ Lucky

⇒ Make sure we are usually lucky

Randomized Quicksort (Partition around a random element)

- Running time is independent of input ordering
- No assumptions need to be made about the input distribution
- No specific input elicits the worst-case behavior
- The worst-case is determined only by the output of a random-number generator.

Analysis: (Pivot on random element)

Let, $T(n)$ = the random variable for the running time of randomized quicksort on an input size n, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the indicator random variable

$$X_k = \begin{cases} 1 & \text{if Partition generates a } k : n-k-1 \text{ split} \\ 0 & \text{otherwise} \end{cases}$$

$E[X_k] = \Pr\{X_k = 1\} = 1/n$, since all the splits are equally likely, assuming elements are distinct

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split} \\ \vdots \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split} \end{cases}$$

Having recurrence ↓ n recursive calls

To obtain an upper bound let's assume that ith element always falls in the larger side of the partition

$$T(n) = \sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n) \right)$$ [multiplying X_k to handle all possible cases]

Calculating Expectation:

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n) \right) \right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k \left(T(k) + T(n-k-1) + \Theta(n) \right) \right]$$

[Linearity of expectation]

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E\left[T(k) + T(n-k-1) + \Theta(n) \right]$$

[X_k is independent as declared earlier]

$$= \frac{1}{n} \left(\sum_{k=0}^{n-1} E[T(k)] + \sum_{k=0}^{n-1} E[T(n-k-1)] + \sum_{k=0}^{n-1} \Theta(n) \right)$$

[Linearity of expectation, $E[X_n] = \frac{1}{n}$]

$$= \frac{2}{n} \sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$ [Summations have identical terms]

Hairy recurrence

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

· [$k=0,1$ can be observed in $\Theta(n)$] ⇒ for technical convenience

Prove: $E[T(n)] \leq an\lg n$ for const $a > 0$

Choose a big enough so that

$an\lg n \geq E[T(n)]$ for small $n \geq 2$

Use fact: $\sum_{k=2}^{n-1} k\lg k \leq \frac{1}{2}n^2\lg n - \frac{1}{8}n^2$

Substitution:

$$E[T(n)] \leq \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n)$$

$$\leq \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \Theta(n)$$

$$= an\lg n - \left(\frac{an}{4} - \Theta(n)\right)$$

[desired − residual]

$$\leq an\lg n$$ [if a is chosen close enought so that $an/4$ dominates the $\Theta(n)$]

In practice

i) Quicksort is a great general-purpose sorting algorithm

ii) Typically over twice as fast as merge sort

iii) Can benifit substantially from code tuning

iv) Behaves well ~~even~~ even ~~if the mem~~ with caching and virtual memory

Lecture 5: Linear-time sorting: Lower bounds, Counting sort, Radix Sort

How fast can be sort? (depends on model of what we can do with the elements)

All the sorting algorithms we have seen so far are comparison sorts: only use comparisons to determine the relative order of elements.

Eg: insertion sort, merge sort, quicksort, heapsort

The best worst-case running time that we've seen so far for comparison sorting is $\Theta(n \lg n)$

⇒ Decision trees can help us answer that question

~~Quick sort~~

quicksort → $\Theta(n \lg n)$ randomized or $\Theta(n^2)$

heapsort → $\Theta(n \lg n)$

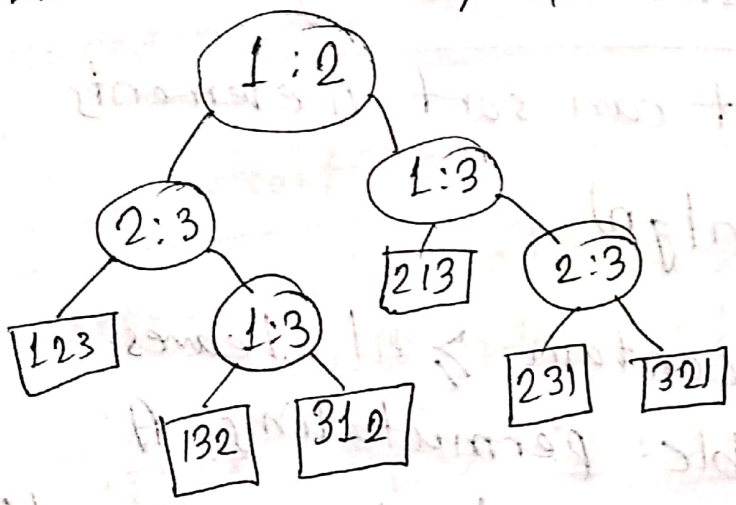
merge sort → $\Theta(n \lg n)$

Insertion sort → $\Theta(n^2)$

Prove: No comparison sorting algorithm runs better than $\Theta(n \lg n)$

Decision tree model [powerful to find out but not as an analysis or representation of algo]

Example sort $\langle a_1, a_2, a_3 \rangle$ = $\langle 9, 4, 6 \rangle$



Each internal node is labeled $i:j$ for $i, j \in \{1, 2, \ldots n\}$

- The left subtree shows subsequent comparisons if $a_i \leq a_j$
- The right subtree shows subsequent comparisons if $a_i \geq a_j$

Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$ has been established.

A decision tree can model the execution of any comparison sort:

- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements
- The tree contains the comparisons along all possible instruction ~~trees~~ traces.

- The running time of the algorithm = the length of the path taken
- Worst-case running time = height of tree

Lower bound for decision tree sorting:

Any decision tree that can sort n elements must have height $\Omega(n \lg n)$

Proof: The tree must contain $\geq n!$ leaves. since there are $n!$ possible permutations. A height-h binary tree has $\leq 2^h$ leaves thus $n! \leq 2^h$

$h \geq \lg(n!)$ [lg is monotonically increasing

$\geq \lg((n/e)^n)$ [Stirling's formula]

$= n \lg n - n \lg e$

$= \Omega(n \lg n)$

Corollary: Heapsort and merge sort are asymtotically optimal comparison sorting algorithms.

(We need to get out of comparison model)

31/10/2020 (38)

Sorting in linear time:

Counting sort:

Input $A[1....n]$,
each $A[i] \in \{1, 2, ...k\}$

Output $B[1, ...n]$ = sorting of A

Auxilary storage $C[1 ..., k]$

Algorithm:

$\theta(k)$ { for $i \leftarrow 1$ to k
do $C[i] \leftarrow 0$

$\theta(n)$ { for $j \leftarrow 1$ to n
do $C[A[j]] \leftarrow C[A[j]] + 1$ [$C[i] = |\{key = i\}|$

$\theta(k)$ { for $i \leftarrow 2$ to k
do $C[i] \leftarrow C[i] + C[i-1]$ [$C[i] = |\{key \le i\}|$

$\theta(n)$ { for $j \leftarrow n$ downto 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] - 1$

$\theta(n+k)$

Lec-7

26/10/2020

NP-Completeness

Chapter 34

Topsort
Critical Path

Completeness and Reducibility

Quiz

CCNA

Running time: $\Theta(k+n)$ [a conditioning can be done if k ... the we can choose ~~the~~ counting]

If $k = \Theta(n)$ then counting sort takes $\Theta(n)$ time

- But sorting sorting takes $\Omega(n \lg n)$ time
- Where's the fallacy [a failure in reasoning]

Answer:

- Comparison sorting takes $\Omega(n \lg n)$ time
- Counting sort is not a comparison sort
- In fact, not a single comparison between elements occur

Stable Sorting:

preserves the relative order of the equal elements

→ Counting sort is stable

Radix Sort: (going to work for much larger range of numbers in linear time)

Origin: Herman Hollerith's card sorting machine (MIT Lecturer) for the 1890's US Census [was reported in 6 weeks]

- Digit-by-digit sort
- Hollerith's orginal (bad) idea: sort on most significant digit first
- Good idea: Sort on least-significant digit first with auxiliary stable sort

⇒ Tabulating machine company in 1911 → later turned to IBM 1924

Correctness of radix sort:

Induction on digit position

→ Assume that the number are sorted by their lower order $t-1$ digits

→ Sort on digit t

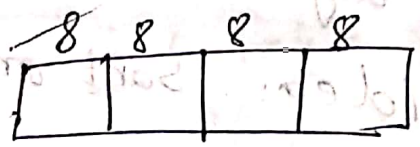
- Two numbers that differ in digit t are correctly sorted: ⇒ sorted order

- ~~Two numbers that differ in digit t are~~
- Two numbers equal in digit t are put in the same order as the input ⇒ correct order [stability]

Analysis of radix sort:

- Assume counting sort is the auxiliary stable sort $O(n+k)$
- Sort n computer words of b bits each ($0 \cdot 2^b$ digits)
- Each word can be viewed as having b/r base-2^r digits

Example 32-bit word



r=8 ⇒ b/r = 4 passes ~~of counting~~ sort on base 28 digits or r=16 ⇒ b/r = 2 passed or counting sort on base 2^{16} digits

How many passes ~~should~~ should be made?

If each b-bit word is ~~chosen~~ broken into r-bit pieces, each pass of counting sort takes $\Theta(n+2^r)$ time. Since there are b/r passes, we have

$$T(n,b) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

Choose r to minimize $T(n,b)$

⇒ Increasing r means fewer passes, but as $r \gg \lg n$ the time grow exponentially

Choosing r:

Minimize $T(n,b)$ differentiating wrt. r and setting to 0

We don't want $2^r \gg n$, and there's no harm asymptotically in choosing r as large as possible subject to this constraint. Choosing $r = \lg n$ implies $T(n,b) = \Theta(bn/\lg n)$

- For numbers in range from 0 to $n^d - 1$ (0 to $2^b - 1$), we have $b = d \lg n$ ⇒ radix sort runs in $\Theta(dn)$ time

$d = b/\lg n$

In practice radix sort is fast for large inputs as well as ∮ simple to code and maintain

· for 32-bit numbers

→ we need 3 passes $d = \frac{b}{\lg n} \approx 3$

→ Merge and quicksort do at least $\lceil \lg 2000 \rceil - 11$ passes

Downside: Unlike quicksort radix sort displays little locality of referent and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.

Best algo for sort $\Theta(n\sqrt{\lg \log n})$ expected

$O(n \lg \log n)$ ← advanced one

Lec-8

2/11/2020

0-1 Knapsack DP (Weak NP)

Approximate Algorithm (ch35)

Vertex-cover problem

TSP

Set cover problem

Greedy-set-cover: (Set)

Lecture-6: Order Statistics, Median

- Select the kth smallest of n elements [the element with rank k]
  - → $k=1$: minimum
  - → $k=n$: maximum
  - → $k = \lfloor (n+1)/2 \rfloor$ or $\lceil (n+1)/2 \rceil$: median

Naive Algorithm: Sort and index kth element

Worst case running time = $\Theta(n \lg n) + \Theta(1)$

$= \Theta(n \lg n)$

using mergesort or heapsort.

Randomized divide and conquer

Rand_Select (A, p, q, i) ▷ ith smallest of A[p..q]

```
if p = q then return A[p]
r ← Rand-partition(A, p, q)
k ← r - p + 1
if i = k then return A[r]
if i < k
    then return Rand_Select(A, p, r-1, i)
    else return Rand_Select(A, r+1, q, i-k)
```

randomized quicksort

$k = rank A[r]$

Intuition for analysis: [Assuming all elements are distinct]

Lucky $= T(9n/10) + \Theta(n)$ $\quad n^{\log_{10/9} 1} = n^0 = 1$ [Case 3]

$= \Theta(n)$

Unlucky $= T(n-1) + \Theta(n)$

$= \Theta(n^2)$ [worse than sort]

Analysis of expected time:

Let $T(n)$ = the random variable for the running time of Rand-select on an input size of n. Assuming random numbers are independent

For $k = 0 \ldots n-1$ define random indicator ~~variable~~

$$X_k = \begin{cases} 1 & \text{if Partition generates a } k : n-k-1 \text{ split} \\ 0 & \text{otherwise} \end{cases}$$

[left hand side]

To X_k = ~~is~~ indicator random varible

To obtain an upper bound, assume that the ith element always falls in the larger side of the partition:

$$T(n) = \begin{cases} T(\max\{0, n-1\}) + \theta(n) & \text{if } 0 : n-1 \text{ split} \\ T(\max\{1, n-2\}) + \theta(n) & \text{if } 1 : n-1 \text{ split} \\ \vdots \\ T(\max\{n-1, 0\}) + \theta(n) & \text{if } n-1 : 0 \text{ split} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \left(T(\max\{k, n-k-1\}) + \theta(n) \right)$$

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \left(T_k(\max\{k, n-k-1\}) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right]$$

[Linearity of expectation]

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E\left[T(\max\{k, n-k-1\}) + \Theta(n)\right]$$

X_k is independent from other random choices ~~a~~ $T(\max\{k, n-k-1\})$ is a recursive call which makes it ~~of~~ own choices and doesn't influence or depend on X_k

$$= \frac{1}{n} \sum_{k=0}^{n-1} E[T(\max\{k, n-k-1\})] + \frac{1}{n} \sum_{k=0}^{n-1} \Theta(n)$$

all random numbers are chosen with uniform probability so $E[X_k] = \frac{1}{n}$ using linearity of expectation

$$\leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + \Theta(n)$$

This can be done because the terms are getting repeated. when $k=0$ or $k=n-1$ we get the same $T(n-1)$ which results in a

$$\sum_{k=0}^{n-1} E\left[T(\max\{k, n-k-1\})\right] \leq 2 \sum_{k=\lfloor n/2 \rfloor}^{n-1} E\left[T(k)\right]$$

Claim: $E[T(n)] \leq cn$ for suffient large $c > 0$

Substitution method:

$$E[T(k)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + \theta(n) \quad [\text{Induction}]$$

$$= \frac{2c}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} k + \theta(n)$$

$$\leq \frac{2c}{n} \left(\frac{3}{8}n^2\right) + \theta(n) \quad \left[\sum_{k=\lfloor n/2 \rfloor}^{n-1} k < \frac{3}{8}n^2\right]$$

$$= cn - \left(\frac{cn}{4} - \theta(n)\right)$$

(cn revised - residual)

[If c is chosen suffiently large so that $cn/4$ dominates $\theta(n)$]

Running time $\theta(cn) = \theta(n)$

worst case $\theta(n^2)$ [if really unlucky]

Worst case linear time order-statistics.

→ Is there an algorithm that runs in linear time in the worst case?

Yes due to Blum, Floyd, Pratt, Rivest and Tarjan [1973]

Idea: generate a good pivot recursively

select (i, n)

1) Divide the n elements into groups of 5. Find the median of each 5 elements group by rote

2) Recursively select the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.

3) Partition around the pivot x Let k = rank(x)

4) if i = k then return x

elseif i < k

then recursively select the ith smallest element in the lower part

else recursively select the (i-k)th smallest element in the upper part

Running time

$T(n) = O(?n) = \theta(n) + \underbrace{T(n/5)}_{\text{Step-2}} + \underbrace{T(3n/4)}_{\text{Step-4}}$

$3\lfloor \lfloor n/5 \rfloor / 2 \rfloor$ elements $\leq x$

$3\lceil n/10 \rceil$ elements are $\leq x$

$3\lfloor n/10 \rfloor$ elements are $\geq x$

so each side as almost $7/10$ elements

Simplification: for $n \geq 50$ we have $3\lfloor n/10 \rfloor \geq n/4$

Therefore for $n \geq 50$ the recursive call to select in step 4 is executed recursively on $< 3n/4$ elements

Thus the recurrence for running time can assume that step 4 takes $T(3n/4)$ in the worst case

for $n \leq 50$, we know that the worst case time is $T(n) = \theta(1)$

Solving the recurrence

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) + O(n)$$

Claim $T(n) \leq cn$

Proof: substitution

$$T(n) \leq \frac{c}{5}n + \frac{3c}{4}n + O(n)$$

$$= \frac{19}{20}cn + O(n)$$

$$= cn - \left(\frac{1}{20}cn - O(n)\right) \quad \text{(for } c \text{ sufficiently large)}$$

~~$T(n) = O(n)$~~

$\leq cn$

Since the work at each level of recursion is a constant fraction (19/20) smaller, the work per level is a geometric series dominated by the linear work at the root

In practice it runs slowly, because c in front of n is large

→ Hence the randomized algorithm is far more practical

Lecture 15: Dynamic Programmic, Longest Common Subsequences

Dynamic Programming

Here programming means any tabular method to accomplish something.

Design technique, like D & C

Example: LCS (Longest Common Subsequence)

→ Given two sequences $x[1...m]$ and $y[1...n]$, find a longest subsequence common to them both

Bruteforce LCS algo:

Check every subsequence of $x[1...m]$ to see if it's also a subsequence of $y[1...n]$.

Analysis:

- Check every subsequence of $x[1...m]$ to see if it's also a subsequence of $y[1...n]$

→ checking = $O(n)$ time per sequence

→ 2^m sequences of x (each bit vector of length m determines a distinct subsequence of x)

Worst case running time = $O(n 2^m)$ = exponential time

Towards a better algorithm:

1. Look at the length of longest common subsequence
2. Extend the algo to find LCS itself

$|s|$ = length of sequence

Strategy: consider prefixes of x and y

Define $c[i,j] = |LCS(x[1..i], y[1..j])|$

then $c[m,n] = LCS[x,y]$

Recursive formulation:

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j] \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases}$$

Proof:

Case $x[i] == y[j]$

Let $z[1 \ldots k] = LCS(x[1 \ldots i], y[1 \ldots j])$,

where $c[i,j] = k$

Then, $z[k] = x[i]$ $(= y[j])$, or else z could be extended by tracking on $x[i]$

Thus, $z[1 \ldots k-1]$ is CS of $x[1 \ldots i-1]$ and $y[1 \ldots j-1]$

Claim: $z[1 \ldots k-1] = LCS(x[1, \ldots i-1], y[1 \ldots j-1])$

Suppose

w is a Longer common sequence that $|w| \geq k-1$

Cut and paste $w \| z[k]$ is certainly a CS of $x[1 \ldots i]$ and $y[1 \ldots j]$ with length $> k$ [Contradition]

($\|$: string concatenation)

So proof by contraction to the claim

Thus $c[i-1, j-1] = k-1$

$c[i,j] = c[i-1, j-1] + 1$

Other cases similar ☒

Hallmarks of dynamic programming: (whenever we see these two dp is supposed to work)

i) Optimal substructure
ii) Overlapping subsequence

Optimal Substructure: An optimal solution to a problem (instance) contains optimal solutions to subproblems

Here $z = LCS(x, y)$. then any prefix of z is an LCS of a prefix x and a prefix y.

Recursive algo. for LCS

```
LCS(x, y, i, j)  # ignoring base cases
if x[i] = y[j]
    then c[i,j] ← LCS(x, y, i-1, j-1) + 1
    else c[i,j] ← max{ LCS(x, y, i-1, j),
                       LCS(x, y, i, j-1) }
return c[i,j]
```

Worst case: if $x[i] \neq y[j]$, the algo evaluates two subproblems, each with one param decremented

Expanding the recursion tree we can say that we are solving subproblems that are already solved. [Height-m+n → work exponential]
which takes us to the second hallmark

Overlapping subproblems: A recursive solution contains a "small" number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of length m and n is only mn

Memoization: After computing solution to a subproblem store in the table. Subsequent calls check the table to avoid redoing work.

```
LCS (x,y, i,j)
  if c[i,j] = NIL
    then if x[i] = y[j]
      then c[i,j] ← LCS (x,y, i-1, j-1) + 1
           c[i,j] ← max{LCS (x,y, i-1, j),
                        LCS (x, y, i-1, j-1)}
    return c[i,j]
```

same as before