

# Divide and Conquer, Sorting & searching & Randomized Algorithms.

- Tim Roughgarden

## Integer Multiplication.

**Input**: two  $n$ -digit numbers  $x$  and  $y$ .

**Output**: the product  $x \cdot y$

"Primitive operation": add or multiply 2 single-digit numbers

### Grade-school algorithm.

$$\begin{array}{r} 3 \quad 3 \\ 5 \cdot 6 \quad 7 \quad 8 \\ \hline 1 \quad 2 \quad 3 \quad 4 \end{array} \times \begin{array}{r} 22 \quad 7 \quad 1 \quad 2 \\ 190 \quad 3 \quad 4 \quad - \\ 1135 \quad 6 \quad - \quad - \\ 5678 \quad - \quad - \quad - \\ \hline 7006652 \end{array}$$

↑  
row

~ O.  $\leq 2n$  operations (per row)

**Upshot**: #operations overall  $\leq$  constant.  $n^2$  (like 4).

## Karatsuba Multiplication.

$$x = \begin{matrix} a \\ b \\ c \\ d \end{matrix}$$

$$y = \begin{matrix} 56 \\ 78 \\ 12 \\ 31 \end{matrix}$$

Step 1 : Compute  $a \cdot c = 672$

Step 2 : Compute  $b \cdot d = 2652$

Step 3 : Compute  $(a+b)(c+d) = 134 \cdot 46 = 6164$

Step 4 : Compute  $\textcircled{3} - \textcircled{2} - \textcircled{1} = 2840$

Step 5 :

$$\begin{array}{r} 672 \quad 0000 \\ 2652 \\ 284000 \\ \hline 7006652 \end{array} +$$

Write  $x = 10^{\frac{n}{2}} a + b$  and  $y = 10^{\frac{n}{2}} c + d$ , write  $a, b, c, d$  are  $\frac{n}{2}$ -digit numbers

[example  $a = 56, b = 78, c = 12, d = 31$ ]

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

$$= 10^n a \cdot c + 10^{\frac{n}{2}}(ad + bc) + b \cdot d$$

Idea: recursively compute  $ac, ad, bc, bd$ , then compute ( $\approx$ ) in the straightforward way.

base case. (simple base case omitted).

for 1-digit numbers, each  $\rightarrow$  1 basic multiplication

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

\*

Step 1 : recursively compute  $a \cdot c$

Step 2 : recursively compute  $b \cdot d$ .

Step 3 : recursively compute  $(a+b)(c+d) = \cancel{ac+ad} + \cancel{bc+bd}$

Gauss's trick

$$\textcircled{3} - \textcircled{1} - \textcircled{2}$$

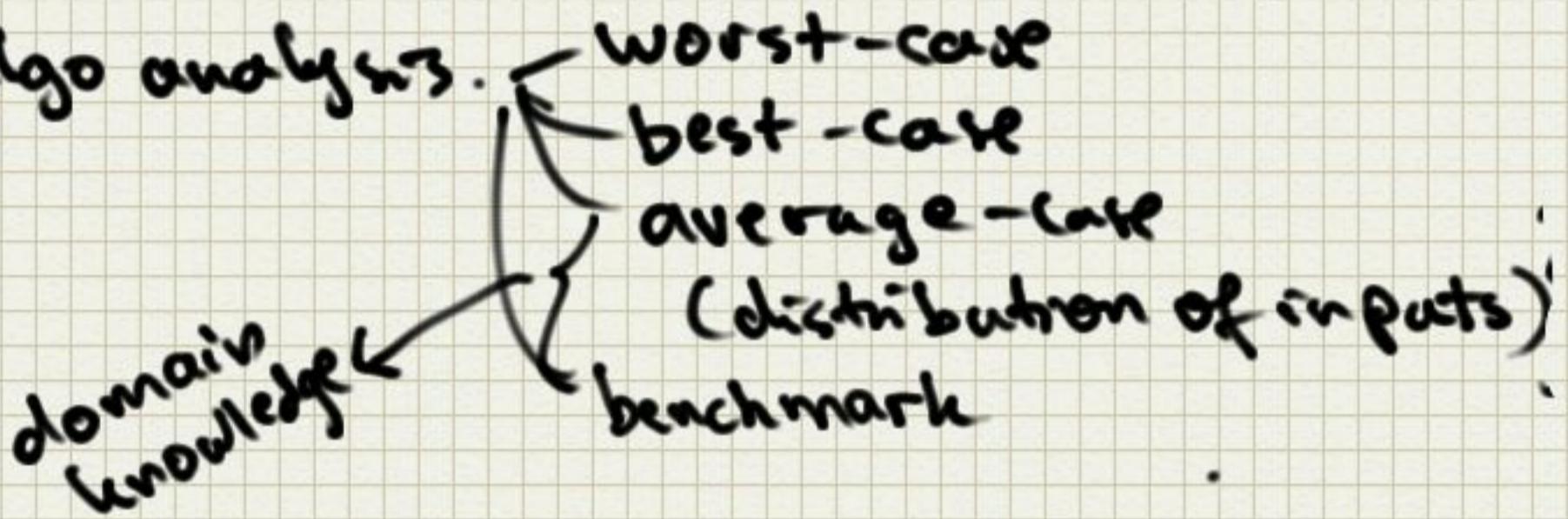
Upshot: only need 3 recursive multiplications!  
(and some additions)

- A few good algorithm books
- Kleinberg / Tardos, Algorithm Design, 2005
  - Dasgupta / Papadimitriou / Vazirani, Algorithms, 2006
  - Cormen / Leiserson / Rivest / Stein, Introduction to Algorithms, 2009 (3<sup>rd</sup> edition)
  - Mehlhorn / Sanders, Data Structures and Algorithms - The Basic Toolbox, 2008.

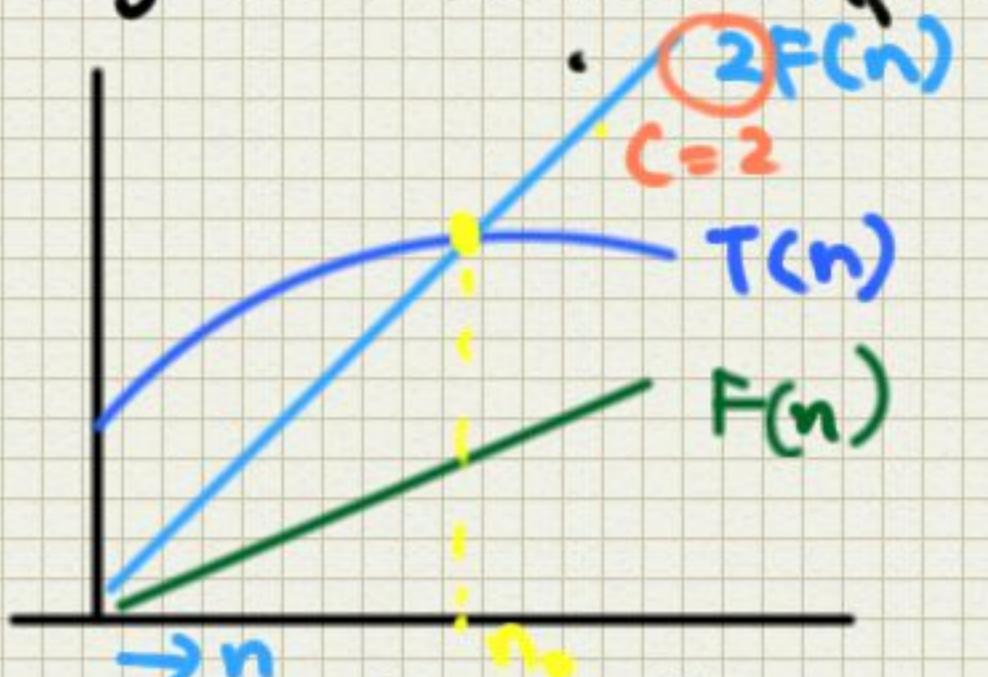
## Merge Sort

- Intro to divide-and-conquer algorithm
- Improvement over
  - selection
  - Insertion
  - Bubble } → sorting method.

algo analysis.



## Bog-Oh: Formal Definition.



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

Formal definition:  $T(n) = O(f(n))$

if and only if there exist constants  $c, n_0 > 0$   
such that  $T(n) \leq c \cdot f(n)$

for all  $n \geq n_0$ .

$n_0 \Rightarrow$  sufficiently large.

Example.

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  $\forall n \geq 1, T(n) \leq c \cdot n^k$ .

$$\begin{aligned} T(n) &\leq |a_k| n^k + \dots + |a_1| n + |a_0| \\ &\leq |a_k| n^k + \dots + |a_1| n^k + |a_0| n^k \\ &\leq c \cdot n^k \end{aligned}$$

## 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  $\exists$  constants  $c, n_0 > 0$ , such that  $n^k \leq c \cdot n^{k-1} \quad \forall n \geq n_0$

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

$$\frac{n^k}{n^{k-1}} \leq c \cdot \frac{n^{k-1}}{n^{k-1}}$$

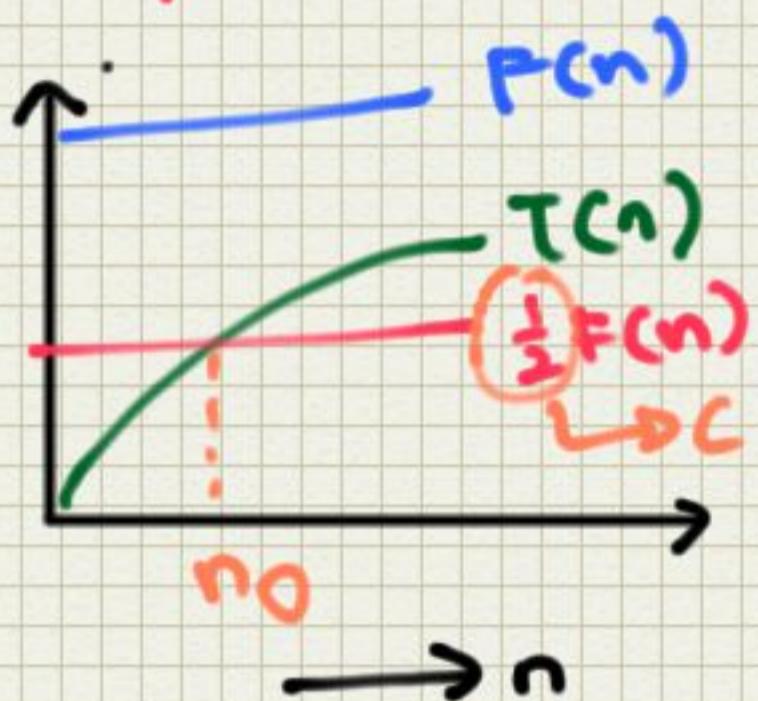
$$n \leq c \quad \forall n \geq n_0$$



false contradiction.

$O(bfg-O) \approx$  less than or equal to  
 $\Theta/\omega\approx$  greater than or equal to

## Omega Notation.



Definition:  $T(n) = \Omega(f(n))$   
if and only if  $\exists$  constants  
 $c, n_0$  such that  
 $T(n) \geq c \cdot f(n) \forall n \geq n_0$

## Definition Theta ( $\Theta$ ) notation

$T(n) = \Theta(f(n))$ , if and only if

$T(n) = O(f(n))$  and  $T(n) = \Omega(f(n))$

Equivalent:  $\exists$  constants  $c_1, c_2, n_0$  such that

$$c_1 f(n) \leq T(n) \leq c_2 f(n),$$

$\forall n \geq n_0$ .

e.g.:  $T(n) = \frac{1}{2}n^2 + 3n$

$$T(n) = \Omega(n)$$

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

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

## Little - Oh notation

Definition:  $T(n) = O(f(n))$ , if and only if  
for all constants  $c > 0$ ,  $\exists$  a constant  $n_0$   
such that

$$T(n) \leq c f(n) \quad \forall n \geq n_0$$

claim:  $2^{n+10} = O(2^n)$

proof: need to pick constants  $C, n_0$  such that

$$2^{n+10} \leq C \cdot 2^n \quad \forall n \geq n_0$$

Note:  $2^{n+10} \leq C \cdot 2^n$

$$2^n \cdot 2^{10} \leq C \cdot 2^n$$

$$1024 \cdot 2^n \leq C \cdot 2^n \quad (\checkmark)$$

, pick  $C = 1024, n_0 = 4.$  ✓

$$\hookrightarrow 1024 \cdot 2 \leq 1024 \cdot 2$$

Claim:  $2^{10n}$  is not  $O(2^n)$

Proof: by contradiction. If  $2^{10n} = O(2^n)$ , then

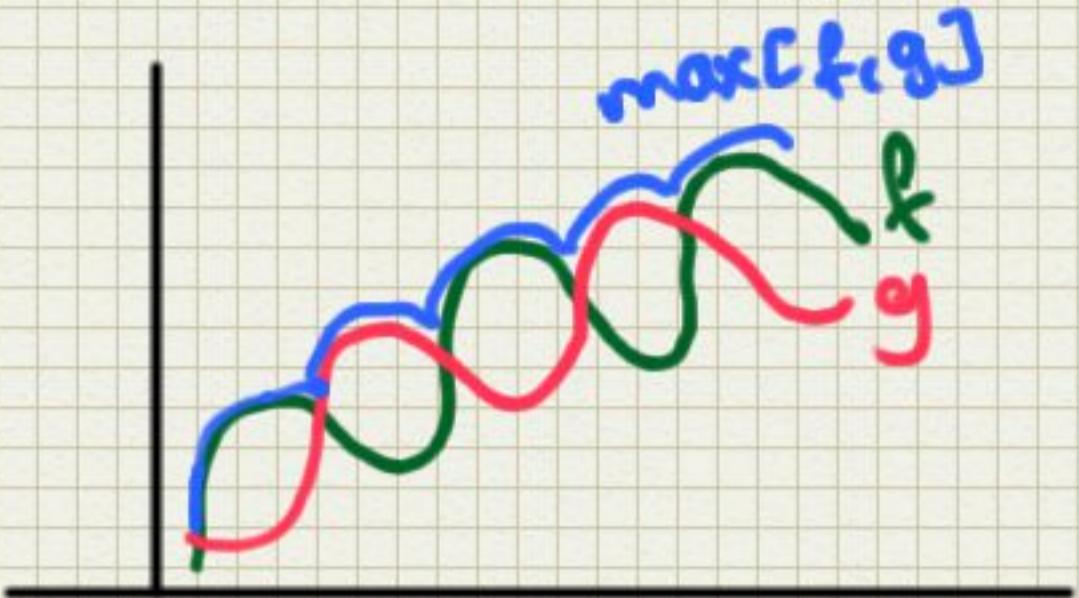
$\exists$  constants  $c, n_0 > 0$ , such that

$$2^{10n} \leq c \cdot 2^n \quad \forall n \geq n_0$$

But then [w/ canceling  $2^n$ ]

$$2^9n \leq c \Rightarrow \emptyset \text{ holds. (false)}$$

Claim: For every pair of (positive) functions  $f(n), g(n)$ ,  $\max[f, g] = \Theta(f(n) + g(n))$



Proof:  $[\max[f, g] = \Theta(f(n) + g(n))]$ .

for every  $n$ , we have

$$\max\{f(n), g(n)\} \leq f(n) + g(n)$$

and

$$2\max\{f(n), g(n)\} \geq (f(n) + g(n))$$

Thus:  $\frac{1}{2}(f(n) + g(n)) \leq \max\{f(n), g(n)\} \leq f(n) + g(n)$

I for all  $n \geq 1$

$$\Rightarrow \max\{f, g\} = \Theta(f(n) + g(n))$$

## Inversion (Week 2)

Input: array A containing the numbers  $1, 2, 3, \dots, n$  in some arbitrary order.

Output: number of inversions = number of pairs  $(i, j)$  of array indices w/  $i < j$  and  $A[i] > A[j]$ .

array in sorted  $\rightarrow$  # inversion is zero.

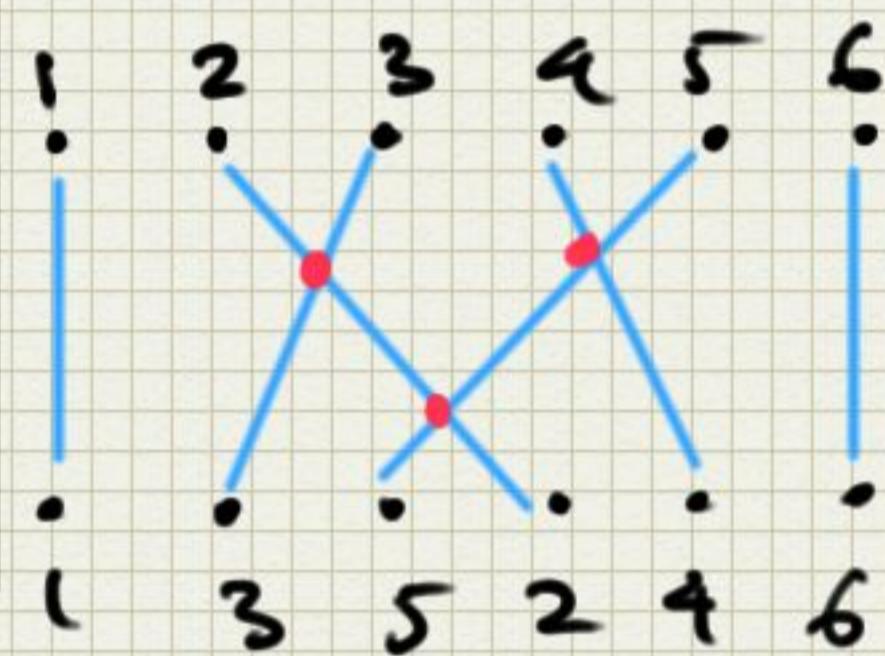
e.g.:  $(1, 3, 5, 2, 4, 6)$ .

Inversions:

(left entry (earlier) is bigger than the right (later entry))

$(5, 2)$   
 $(3, 2)$   
 $(5, 4)$

} 3 inversions.



Motivation: numerical similarity measure between two ranked lists.

(e.g. for "Collaborative filtering").

The largest possible number of inversions that a 6-element array can have?  $\binom{n}{2} = \frac{n(n-1)}{2}$   
 $\rightarrow \frac{6 \cdot 5}{2} = 15$ .

# High-Level Approach.

## Brute-force -

- double for loop ( $i$ ),  $j (> i)$  :  $\Theta(n^2)$  time
- 

Divide and conquer.

- Call an inversion ( $i, j$ ) [with  $i < j$ ]:
  - left inversion if  $i, j \leq n/2$
  - right inversion if  $i, j > n/2$
  - split inversion if  $i \leq n/2 < j$  → need a separate subroutine

Count (array A, length n)

if  $n = 1$  return 0

else

$x = \text{Count}(\text{1st half of } A, n/2)$

$y = \text{Count}(\text{2nd half of } A, n/2)$

$z = \text{CountSplitInv}(A, n)$

return  $x + y + z$ .

Goal: Implement CountSplitInv in linear  
 $(O(n))$ . time.



Count will run in  $O(n \log n)$

split-inversion

↳ the earlier index on the left half of an arr

$k$

the second index on the right half

max split-inversion ( $n^2$ ).

## SPLIT Recursion.

key idea: have recursive calls both  
count inversions      sort.

Motivation: merge subroutine naturally uncovers split inversions.

---

sort\_count (array A, length n)

if  $n=1$  return 0

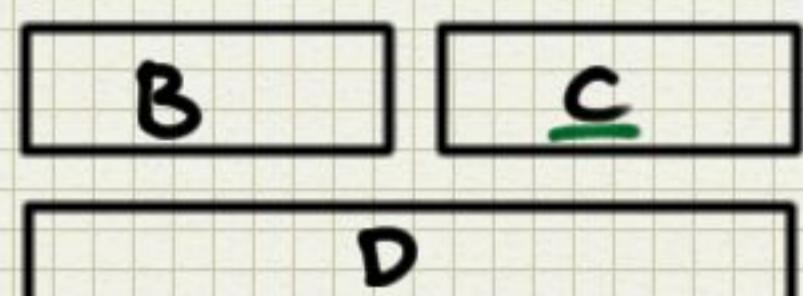
else

(B, x) = sort\_count ( $1^{\text{st}}$  half of A,  $n/2$ )

(C, y) = sort\_count ( $2^{\text{nd}}$  half of A,  $n/2$ ).

(D, z) = merge\_countSplitInv (B, C)

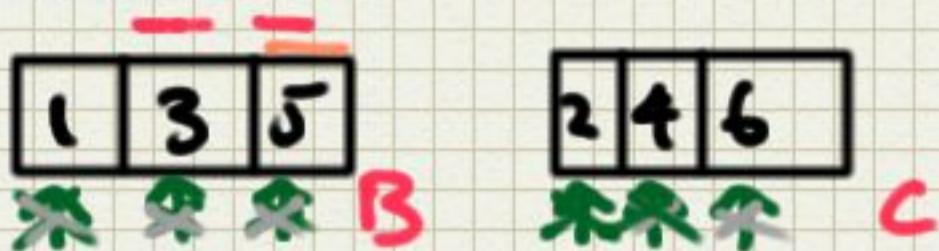
return  $x + y + z$ .



split inversion

when element from C get copied.

e.g. consider merging  
e.g. all inversions



Output

D

⇒ When 2 copied to output, discover split inversion  
(3, 2), (5, 2). # 2 inversion

⇒ When 4 copied to output, discover split  
inversions (5, 4). # 1 inversion

claim: the split inversions involving  
an element  $y$  of the 2<sup>nd</sup> array C are precisely  
the numbers left in the 1<sup>st</sup> array B when  
 $y$  is copied to the output D.

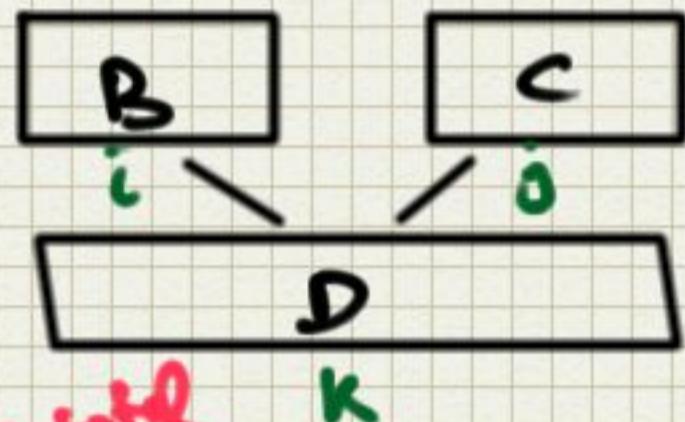
Proof: let  $x$  be an element of the 1<sup>st</sup> array B.

① If  $x$  copied to output D before  $y$ ,  
then  $x < y \Rightarrow$  no inversion as  $x < y$ .

② If  $y$  copied to output D before  $x$ ,  
then  $y < x \Rightarrow$  as  $x$  is in the left array  
( $y$  left index) and  $y$  is in the  
right array  $\Rightarrow$  split inversion.

## Merge-and-Count Split Inv.

- While merging the two sorted subarrays, keep running total of number of split inversions.
- increment: When element of 2<sup>nd</sup> array  $C$  gets copied to output  $D$ , increment total by number of elements remaining in 1<sup>st</sup> array  $B$ .



Runtime:

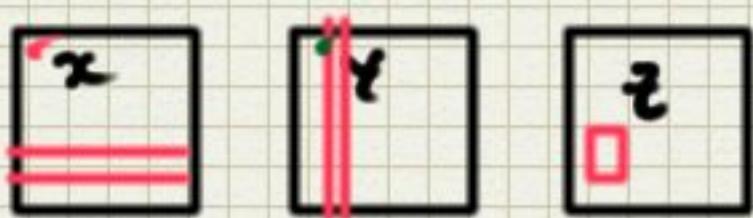
$$O(n) + O(n) = O(n)$$

$c_1 \cdot n$

$$c_2 \cdot n = (C_1 + C_2) \cdot n \quad O(n).$$

Sort-and-Count  $\rightarrow O(n \log n)$ .

## Matrix Multiplication.



( $n \times n$  matrices)

where  $z_{ij} = (\text{i}^{\text{th row}} \text{ of } X) . (\text{j}^{\text{th column}} \text{ of } Y)$

$$= \sum_{k=1}^n x_{ik} \cdot y_{kj}$$

Note: Input size  $\Theta(n^2)$ .

eg ( $n=2$ )

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{pmatrix}$$

straightforward iterative algo for matrix multiplication

$\Theta(n^3)$

$$z_{ij} = \sum_{k=1}^n x_{ik} y_{kj} = \Theta(n)$$

$$\text{Input size } (n^2) = \Theta(n) \cdot n^2 = \Theta(n^3)$$

## Applying Divide and conquer.

Idea

$$\text{write } X = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \text{ and } Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}$$

[where A through H are all  $n/2 \times n/2$  matrices].

$$X \cdot Y = \begin{pmatrix} AE + BG & BF + BH \\ CE + CG & CF + DH \end{pmatrix}$$

Step 1: recursively compute the 8 necessary products.

Step 2: do the necessary additions

running-time:  $O(n^2)$

Fact: total running - time  $O(n^3)$

Syrassen's algorithm (1969).

Step 1: recursively compute only 7 (cleverly chosen) products

Step 2: do the necessary (clever) additions & subtractions  
(still  $O(n^2)$  time).

Fact: better than cubic time! [See master method, later]

The details.

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \quad Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix}.$$

The seven products:

$$\begin{aligned} P_1 &= A(E-H), \quad P_2 = (A+B)H, \\ P_3 &= (C+D)E, \quad P_4 = D(G-E), \\ P_5 &= (A+D)(E+H), \quad P_6 = (B-D)(G+H), \\ P_7 &= (A-C)(E+F) \end{aligned}$$

claim  $X \cdot Y = \begin{pmatrix} AE + BG \\ CE + DG \end{pmatrix} = \begin{pmatrix} P_1 + P_4 - P_2 + P_6 \\ P_3 + P_4 \end{pmatrix} = \begin{pmatrix} P_1 + P_2 \\ P_1 + P_5 - P_3 - P_7 \end{pmatrix}$

Proof:  $P_1 + P_4 - P_2 + P_6 = AE + BG ? \checkmark$

$$\underline{AE} + \cancel{AH} + \cancel{DE} + \cancel{DH} + \cancel{DG} - \cancel{DE} - \cancel{AH} - \cancel{BH} + \underline{BG} + \cancel{BH} - \cancel{DB} - \cancel{PH}$$

## Master method

- A recursive algorithm., recall from the integer multiplication  
 where  $x = 10^{n/2}a + b$ ,  $y = 10^{n/2}c + d$   
 [where  $a, b, c, d$  are  $n/2$ -digit numbers].

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

Algorithm #1: recursively compute  $ac, ad, bc$ , and  $bd$   
 then compute \* in obvious way.

$T(n)$ : max number of operations, this algorithm needs to multiply two  $n$ -digits numbers. (worst-case) or (upper bound)

Recurrence: express  $T(n)$  in terms of running time of recursive calls.

Base case:  $T(1) \leq$  a constant

→ work done (\*)

$$\forall n > 1 : T(n) \leq 4T(n/2) + \Theta(n)$$



work done by recursive calls.

Algorithm #2 (Gauss): recursively compute  $ac, bd, (a+b)(c+d)$  [recall  $ad+bc = (3)-(1)-(2)$ ]

New Recurrence:

base case:  $T(1) \leq$  a constant

$$\forall n > 1 : T(n) \leq \underbrace{3T(n/2)}_{\text{work done in recursive calls}} + \Theta(n)$$

Note: for merge sort  $T(n) \leq \underline{2T(n/2)} + \Theta(n)$ .

## The Master Method

→ a "black box" for solving recurrences.

Assumption: all subproblems have equal size

### Recurrence format:

① Base case:  $T(n) \leq c$  constant,  $\forall$  sufficiently small  $n$ .

② For all larger  $n$ :

$$T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d), \text{ where}$$

a: number of recursive calls (# of subproblems) ( $\geq 1$ )

b: input size shrinkage factor ( $> 1$ ) e.g. in split into half  
 $b=2$

d: exponent in running time of "combine step"  
, outside of the recursive calls. ( $\geq 0$ ).

[a, b, d are independent of n]

### The master method

-  $O(n^d \log n)$  if  $a = b^d$  (case 1)

$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \text{ (case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \text{ (case 3)} \end{cases}$

note  $O(n^d \log n)$

*log base does not matter  
as it will be suppressed by the leading  
constant,  $n^d$*

$O(n^{\log_b a})$

*→ log base in the exponent matters*

*because  $n^2 < n^3, \dots$*

## The Master Method.

a: # recursive calls  
b: size of subproblem

if  $T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d)$

then

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \text{ (case 1)} \\ O(n^d) & \text{if } a < b^d \text{ (case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \text{ (case 3).} \end{cases}$$

e.g merge sort.

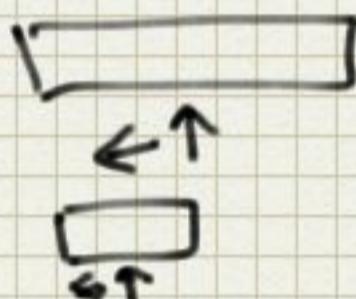
$$a = 2$$

$$b = 2 \rightarrow \text{half of the original problem} \quad \exists b^d = 2$$

$$d = 1 \rightarrow \text{merge } O(1) \rightsquigarrow \text{linear}$$

$$\text{Case 1} = T(n) \leq O(n^d \log n) = O(n \log n).$$

e.g. binary search of a sorted array.



$$a = 1$$

$$\begin{matrix} b = 2 \\ d = 0 \end{matrix}$$

$$a = b^d$$

$$\text{Case 1 } O(n^d \log n)$$

$$O(\log n).$$

$d = 0 \Rightarrow$  means constant time.

e.g. Integer multiplication w/ gauss trick

$$a = 4 \quad a > b^d \rightarrow \text{case 3}$$

$$\begin{matrix} b = 2 \\ d = 1 \end{matrix}$$

$$T(n) \leq O(n^{\log_b a})$$

$$\leq O(n^{\log_2 4})$$

$$\leq O(n^2)$$

~ similar to high-school trick.

w/ gauss trick

$$a = 3$$

$$a > b^d$$

$$\begin{matrix} b = 2 \\ d = 1 \end{matrix}$$

$$T(n) \leq O(n^{\log_b a})$$

$$\leq O(n^{\log_2 3})$$

$$\leq O(n^{\log_2 3})$$

$$\leq O(n^{1.59}) \quad (\text{better than } O(n^2))$$

## example 5

strassen's matrix multiplication algorithm

$$\left. \begin{array}{l} a=7 \\ b=2 \\ d=2 \end{array} \right\} a > b^d \rightarrow \text{case 3.}$$

$$T(n) = O(n^{\log_2 7}) = O(n^{2.81})$$

( $\hookrightarrow$  linear in matrix size)



bent the naive iterative algorithm  $O(n^2)$

## Fictitious recurrence

$$T(n) \leq 2T(n/2) + O(n^2)$$

$$\left. \begin{array}{l} a=2 \\ b=2 \\ d=2 \end{array} \right\} a < b^d \rightarrow \text{case 2}$$

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

Assume: recurrence is

(i)  $T(1) \leq C$

(ii)  $T(n) \leq aT(\frac{n}{b}) + cn^d$  (for some constant  $c$ ).

and.  $n$  is a power of  $b$ .

(note general case is similar, but more tedious).

Idea: generalize mergesort analysis (use of recursion tree)

Mergesort recursion tree.

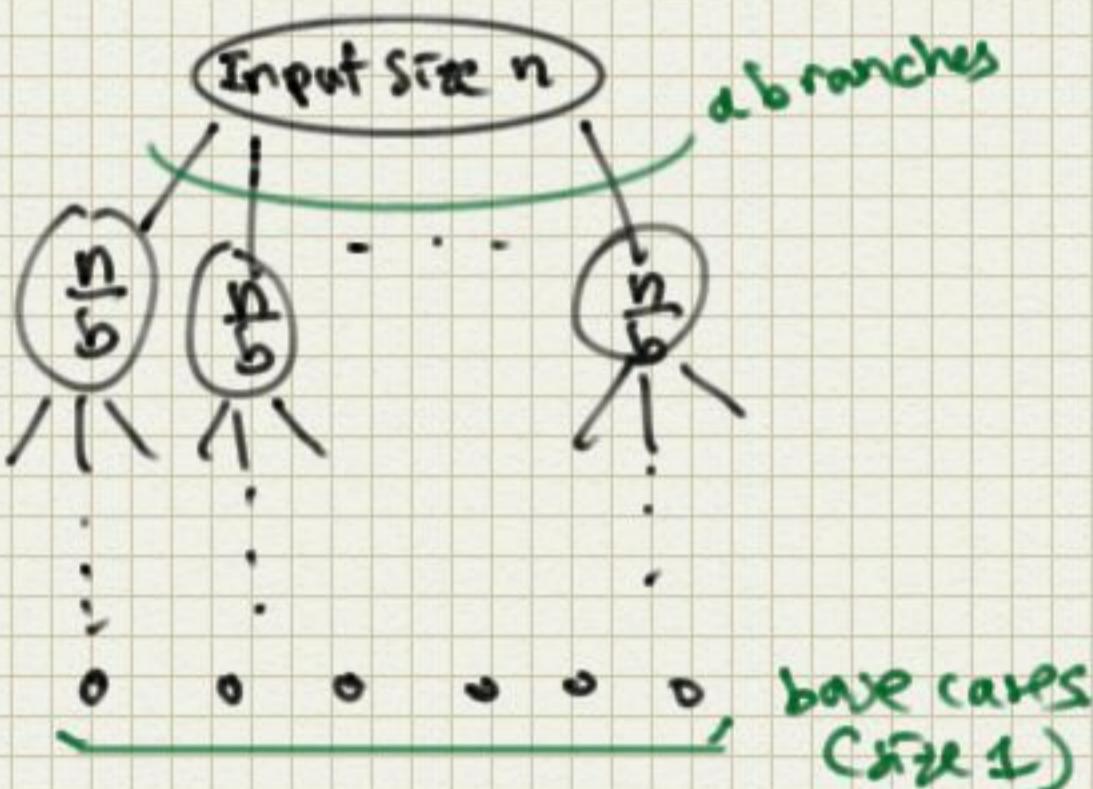
at each level  $j=0, 1, 2, \dots, \log_b n$   $\downarrow$    
  $\text{# of subproblems}$   $\leftarrow$  each  $\frac{n}{b^j}$  of size  $n/b^j$

# of times you can divide  $n$  by  $b$  before reaching 1

---

The recursion tree

level 0



level  $\log_b n$

---

1 Work at a single level (level  $j$ )

Total work at level  $j$  [ignoring work in recursive calls (later or subsequent work)]

$$\leq a^j \cdot C \cdot \left[ \frac{n}{b^j} \right]^d$$

↓  
size input  
of each level's  
subproblem.

work per level- $j$

$\rightarrow \frac{a}{b^d}$  : ratio!

$$\leq C n^d \left[ \frac{a}{b^d} \right]^j$$

Total work: summing over all levels,  $j = 0, 1, 2, \dots, \log_b n$

$$\text{total work} \leq C n^d \cdot \sum_{j=0}^{\log_b n} \left[ \frac{a}{b^d} \right]^j *$$

## How to Think About (\*)

Our upper bound on the work at level  $j$ :  $c n^d \times \left(\frac{a}{b}\right)^j$

Interpretation (tug of good & evil.)

$a$  = rate of subproblem proliferation. (RSP)  $\rightarrow$  function of  $j$

$\hookrightarrow$  (force of evil) as we go down to tree

$b$  more subproblems appear 

$b^d$  = rate of work shrinkage (RWS)  $\rightarrow$  function of  $j$

Why  $b^d$ ?  $b$  is a factor by which an input size shrunked w/the recursion level  $j$

e.g.:  $b = 2 \rightarrow$  so the next subproblem is half the previous level

$d \rightarrow$  the amount of work solving that subproblem.

$$\begin{array}{l} r_{\text{RSP}} = 2 \\ r_{\text{RWS}} = \frac{1}{2} \\ \downarrow \\ b = 2 \end{array}$$

$\rightarrow$  recursive half of the input w/ a linear work.

$\downarrow$   
• input half  
• work also decrease by half  $\frac{1}{2} = 50\% \downarrow$

$$\frac{1}{2^2} = \frac{1}{4} = 25\% \downarrow$$

---

RSP < RWS, the amount of work is decreasing w/the recursion level  $j$

RSP > RWS, the amount of work is increasing w/the recursion level  $j$

RSP & RWS are equal, then the amount of work is the same at every recursion level  $j$ .

Intuition for 3 cases:

Upper-bound for level  $j = Cn^d \times \left(\frac{a}{b^d}\right)^j$

- ①.  $RSP = RWS \Rightarrow$  same amount of work each level.  
(eg: merge sort)  $\rightarrow n^d \cdot \log n$ .  $\mathcal{O}(n^d \log n)$   $\rightarrow$  merge sort.
- ②.  $RSP < RWS \Rightarrow$  less amount each level  $\Rightarrow$   
most work at the root (dominance)  $\sim \mathcal{O}(n^k)$ .
- ③.  $RSP > RWS \Rightarrow$  more work each level  $\Rightarrow$   
most work at the leaves [ might expect  $\mathcal{O}(\# \text{leaves})$  ].

Total work :  $\leq cn^d * \sum_{j=0}^{\log_b n} \left(\frac{a}{b}\right)^j$  (k).

If  $a = b^d$ , then

$$\leq cn^d * \sum_{j=0}^{\log_b n} \left(\frac{b^d}{b}\right)^j$$

$$\leq cn^d * \sum_{j=0}^{\log_b n} (1)^j$$

$$\leq cn^d * (1 + \log_b n)$$

$$\leq cn^d (\log_b n + 1).$$

$$\leq O(n^d \log_b n)$$

$$\leq \Theta(n^d \log n).$$

Basic sum fact

$r > 0, r \neq 1$

$$r = \frac{a}{b}$$

$$1 + r^1 + r^2 + r^3 + \dots + r^k = \frac{r^{k+1} - 1}{r - 1}$$

upshot

→ independent of k.

① If  $r < 1$  is constant,  $\leq \frac{1}{1-r} = 2$  constant.

e.g.  $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots \leq \frac{1}{1-\frac{1}{2}}$   
( $r = \frac{1}{2}$ )  $\leq 2$ .

(re) the 1st term of sum dominates.

↑ independent of k.

② If  $r > 1$  is constant, R.H.S is  $\leq r^k \cdot (1 + \frac{1}{r-1})$ .

e.g.  $r = 2 \quad 1 + 2 + 4 + 8 + \dots \leq 2^k$   
↑ twice of the last term

(the last term of sum dominates).

$$\text{Total work: } \leq c n^d \times \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j \quad (*).$$

If  $a < b^d$  [RSP < RSW] : the amount of work decreasing at each level of recursion tree.

$$a < b^d \rightarrow \leq c n^d \times \sum_{j=0}^{\log_b n} \left(\frac{a}{b^d}\right)^j$$

$\hookrightarrow r < 1.$

$\hookrightarrow \leq 2 \text{ constant (independent of } n)$

$$\leq c n^d * C$$

$\leq O(n^d)$ .  $\rightarrow$  C total work dominated by root (top level)

Case 3, if  $a > b^d$

$$\text{Total work: } \leq c n^d + \sum_{j=0}^{\log_b n} \left[\frac{a}{b^d}\right]^j$$

$\hookrightarrow r > 1.$

$\leq c n^d + C$ . largest term.

$$\leq c n^d + C \left(\frac{a}{b^d}\right)^{\log_b n}.$$

$$\leq O\left(n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n}\right)$$

$$\leq (b^{-d})^{\log_b n}$$

$$\leq b^{-d \log_b n}$$

$$\leq b^{\log_b n - d} = n^{-d}$$

$$\leq O(n^d \cdot n^{-d} \cdot a^{\log_b n})$$

$\leq O(a^{\log_b n}) \Rightarrow \# \text{ of leaves}$



Note:

$$a^{\log_b n} = n^{\log_b a} \rightarrow \text{Simpler to apply}$$
$$[\log_b^n \log_b a] = \log_b a \log_b n.$$

↳ more intuitive.

Note. Worst-case running time of the quick sort.

$$\geq n + (n-1) + (n-2) + \dots + 1.$$

consider  $\gamma_2$  turning all

$$\text{at least } \gamma_2 \Rightarrow 0 \leq C_1(n^2) \leq n(n+1) + \dots + 1 \leq C_1(n^2)$$

$$f(n) = w(g(n))$$

$$f(n) \geq c g(n)$$

$$\frac{n}{2} \cdot \underbrace{\frac{n}{2}}_{\text{sum half off total}} \approx \frac{n}{4}$$

$$\text{Note: } \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$$

$$\begin{aligned} \sum_{k=1}^{n-1} k &= \frac{n(n-1)}{2} \\ &= 10 \end{aligned}$$

(sum half off total).

Best Case w/ the median element as a pivot.

Reason: let  $T(n)$  = running time of an array of size  $n$ .

$$\text{Then: } T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(n) \quad \begin{array}{l} \text{choose pivot} \\ \text{partition} \end{array}$$

$\overbrace{\text{pivot-median}}$   $\overbrace{\text{work outside recursive calls}}$   
 $\text{(work in the recursive calls)}$

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

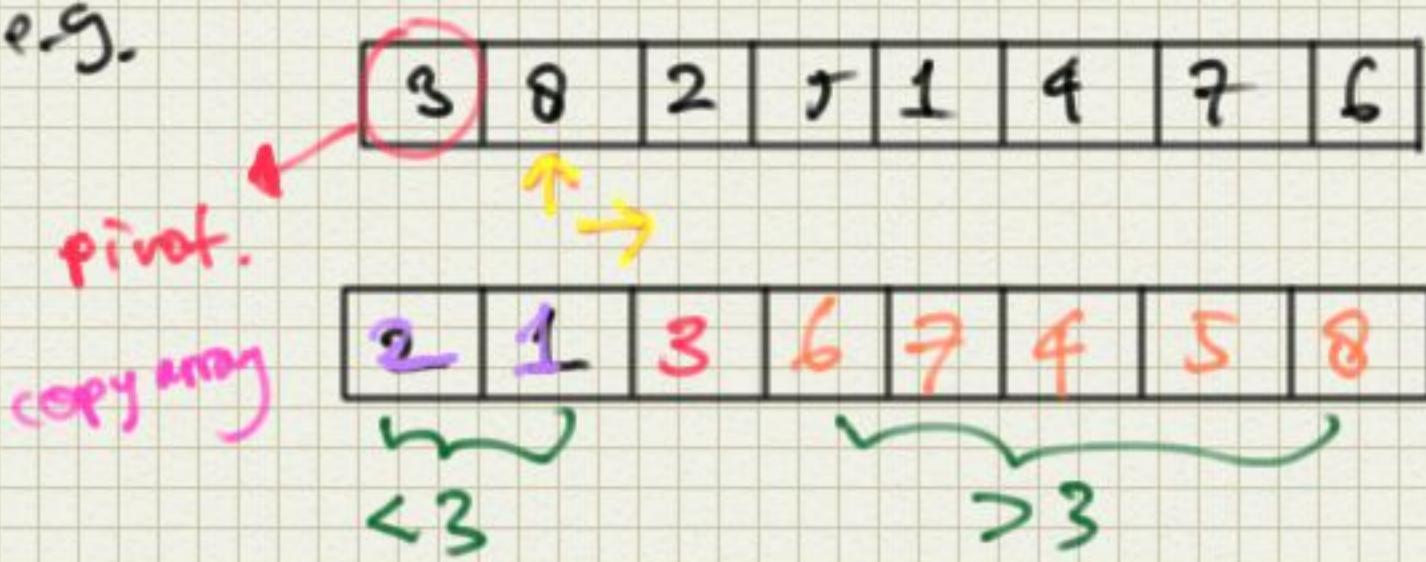
Note:  $a=2$      $b=2$      $d=1$

$$\left. \begin{array}{l} a \text{ vs } b^d \\ 2 \text{ vs } 2 \\ a=b^d \end{array} \right\} \quad \begin{array}{l} a \text{ vs } b^d \\ 2 \text{ vs } 2 \\ a=b^d \end{array} = \Theta(n^{d \log n}) = \underline{\Theta(n \log n)}$$

QuickSort : The partition subroutines.

Note - using  $O(n)$  extra memory, easy to partition around pivot in  $O(n)$  time.

e.g.



if bigger than  
3, fill the  
copy array  
from right.  
if < (less) than  
3 (pivot), fill  
the copy array  
from left.

( $\rightarrow$ ) : extra memory space.

In-Place Implementation.

assume : pivot = 1<sup>st</sup> element of array

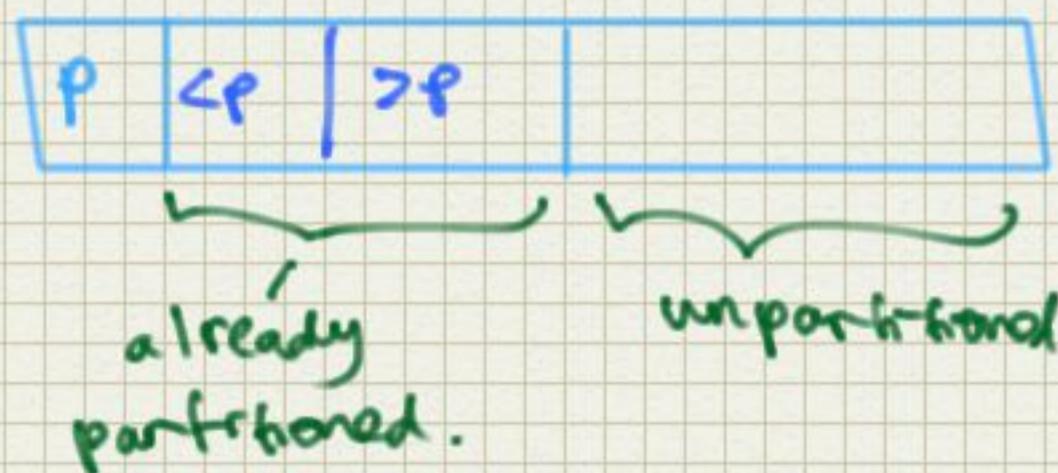
[If not, swap pivot  $\leftrightarrow$  1<sup>st</sup> element as preprocessing step]

High-level Idea:

- Single scan through array.

- Invariant:

everything we have looked at  
so far is partitioned.



## Partition example.

3	8	2	5	1	4	7	6
---	---	---	---	---	---	---	---

$i_0$  /  $\xrightarrow{\text{partitioned}}$

3	8	2	5	1	4	7	6
---	---	---	---	---	---	---	---

$i \rightarrow$   $\xrightarrow{\text{partitioned}}$   $j \rightarrow$   $\xrightarrow{\text{partitioned}}$

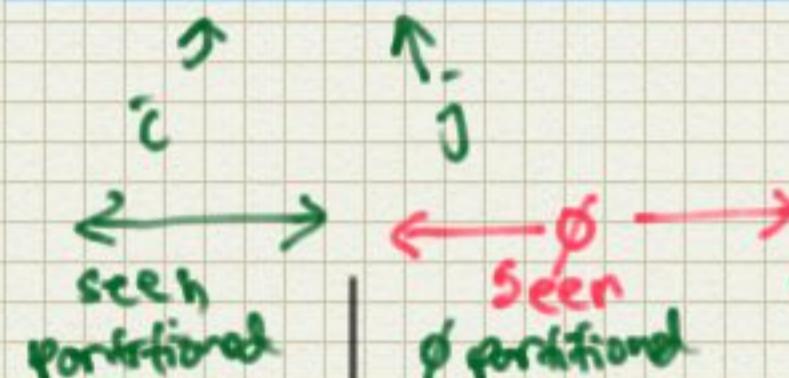
3	2	8	5	1	4	7	6
---	---	---	---	---	---	---	---

$\xrightarrow{\text{partitioned}}$

3	2	8	5	1	4	7	6
---	---	---	---	---	---	---	---

$\xrightarrow{\text{partitioned}}$   $\xrightarrow{\text{partitioned}}$

$P \leq p > p \quad ?$



$i$ : pointer between Left & Right bucket

$j$ : pointer between what we have seen & Not yet seen.

$i \rightarrow$   $\xrightarrow{\text{seen}}$   $j$

3	2	8	5	1	4	7	6
---	---	---	---	---	---	---	---

$\xrightarrow{\text{partitioned}}$   $\xrightarrow{\text{not partitioned}}$

[Fast-forwarding]

3	2	1	5	8	4	7	6
---	---	---	---	---	---	---	---

$i \rightarrow$   $\xrightarrow{\text{partitioned}}$   $j \rightarrow$   $\xrightarrow{\text{not partitioned}}$

$\xrightarrow{\text{partitioned}}$

3	2	1	5	8	4	7	6
---	---	---	---	---	---	---	---

$i' \rightarrow$   $j \rightarrow$

1	2	3	5	8	4	7	6
---	---	---	---	---	---	---	---

$i' \rightarrow$   $j \rightarrow$

$\xrightarrow{\text{partitioned}}$

## Pseudocode for partition.

partition(A, l, r)

$l$  = left boundary  
 $r$  = right boundary

[Input:  $A[l..r]$ ]

- $p = A[l]$

- $i = l + 1$

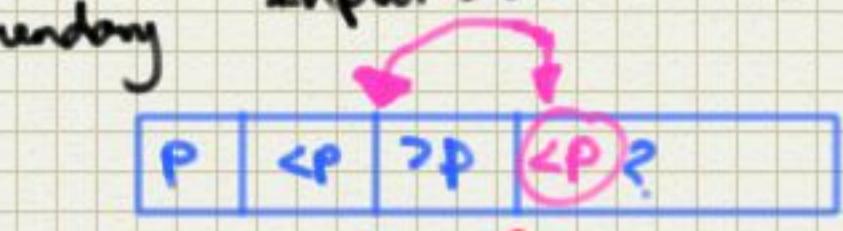
- for  $j = l + 1$  to  $r$

- if  $A[j] < p$ : [if  $A[j] > p$ , do nothing]

- swap  $A[j]$  and  $A[i]$ . # avoid redundant

- $i = i + 1$

- swap  $A[l]$  and  $A[i - 1]$ .



# avoid redundant  
swap by

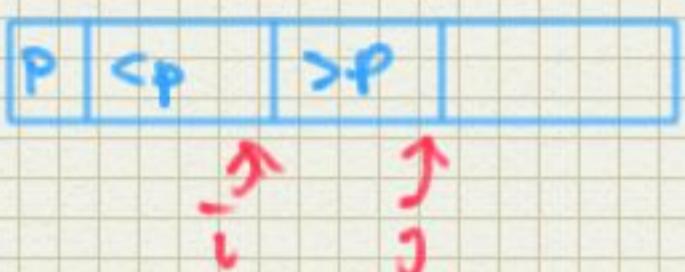
first check if there is  
an element greater than  
pivot, then allow swap

Running Time =  $O(n)$ , where  $n = r - l + 1$  (the length of the input (sub)array)

Reason:  $O(1)$  work per array entry

Correctness:

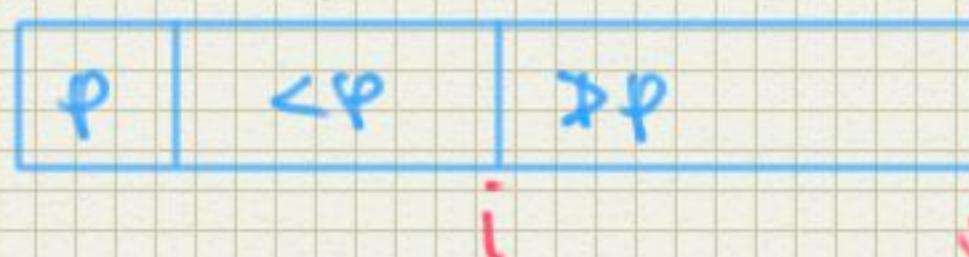
①  $A[l+1], \dots, A[i-1]$  are all less than the pivot



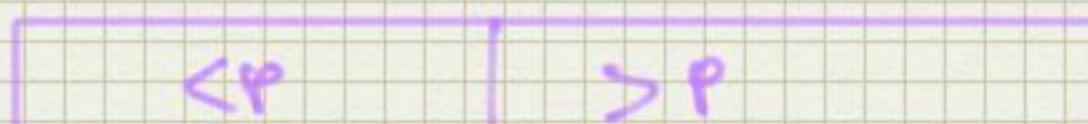
②  $A[i], \dots, A[j-1]$  are all greater than pivot



Consequence: at the end of the for loop



After final swap, array partitioned around pivot.



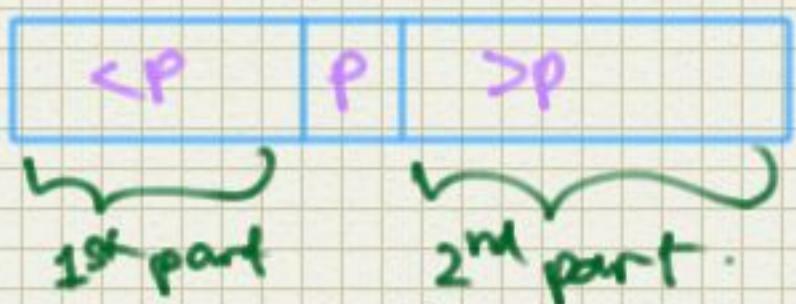
QuickSort: High-level description. [Hoare, §(21)]

Quicksort( array A, length n )

if  $n = 1$  return

$p = \text{choose pivot}(A, n)$

partition A around p  $\rightarrow$



recursively sort 1<sup>st</sup> part.

recursively sort 2<sup>nd</sup> part.

The importance of the pivot.

Q: running time of quick sort?

A: depends on the quality of the pivot?

good      bad

or good pivot  
Split the array

into two equal  
subarrays  $\rightarrow$  balanced subproblems.

unbalanced subproblems -

e.g.

suppose we implemented Quicksort so that choose pivot always selects the 1<sup>st</sup> element of the array.

what is the running time of this algorithm on an input array that is already sorted?

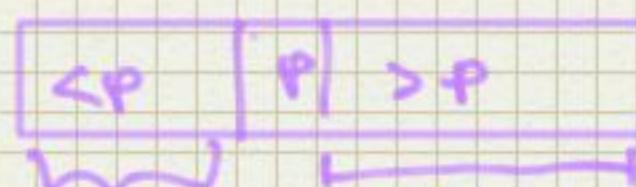
e.g.  $[1, 2, 3, 4, 5, 6, 7]$   $\Rightarrow$  Input

$\Theta(n^2)$  running time.

Reason:  $[1, \dots, n]$ .

1). choose pivot  $\Rightarrow 1$ .

2). Partition invoked on the sorted array.  
( $\phi$ swap, or do nothing in the already sorted array).



length n-1  
(still sorted).

$\Theta(n^2)$

$n + n-2 + n-3 + \dots + 1$

$= n(n+1)\Theta(n^2)$

$\frac{1}{2}n^2 > \sum_{i=1}^n i < n^2$

3). Only 1 recursive call w/  $n-1$ .

since 1<sup>st</sup> part is empty

& continue recursive  $n-2, n-3, \dots, 1$ .

Note. Worst-case running time of the quick sort.

$$\geq n + (n-1) + (n-2) + \dots + 1.$$

consider  $\gamma_2$  turning all

$$\text{at least } \gamma_2 \Rightarrow 0 \leq C_1(n^2) \leq n(n+1) + \dots + 1 \leq C_1(n^2)$$

$$f(n) = \omega(g(n))$$

$$f(n) \geq c g(n)$$

$$\frac{n}{2} \cdot \underbrace{\frac{n}{2}}_{\text{sum half off total}} \approx \frac{n}{4}$$

$$\text{Note: } \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$$

$$\begin{aligned} \sum_{k=1}^{n-1} k &= \frac{n(n-1)}{2} \\ &= 10 \end{aligned}$$

(sum half off total).

Best Case w/ the median element as a pivot.

Reason: let  $T(n)$  = running time of an array of size  $n$ .

$$\text{Then: } T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(n) \quad \begin{array}{l} \text{choose pivot} \\ \text{partition} \end{array}$$

$\overbrace{\text{pivot-median}}$   $\overbrace{\text{work outside recursive calls}}$   
(work in the recursive calls)

$$T(n) = O(n \log n)$$

Note:  $a=2$      $b=2$      $d=1$      $\left\{ \begin{array}{l} a \text{ vs } b^d \\ 2 \text{ vs } 2 \\ a=b^d \end{array} \right. = \Theta(n^{d \log n}) = \underline{\Theta(n \log n)}$

## Random Pivots

key question: How to choose pivots? Big idea: Random!  
[equally likely prob.]

Note: in every recursive call, choose the pivot randomly.

Hope: A random pivot is "pretty good"

Intuition: If always get a 25-75 split, good enough for  
 $O(n \log n)$  running time

② Half of elements [26 - 75 (inclusive) in [1, 2, 3, ... 100]]  
would give a 25-75 split or better

Quicksort Theorem:

For every input array of length  $n$ , the average running time  
of quick sort (w/ random pivots) is  $O(n \log(n))$

Note:

- 1). hold for every input (no assumptions on the data).
- 2). "average" is over random choices made by the algorithm (r-e, pivot choices).

Fix input array  $A$  of length  $n$   
sample space  $\Sigma$  = all possible outcomes of all random choices  
in quicksort (r.e.: pivot sequences)

key random variable: for  $\sigma \in \Sigma$ ,

$C(\sigma)$  = # of comparisons between two input elements  
made by quicksort (given random choices,  $\sigma$ )

lemma: running time of quicksort dominated by comparisons

$\exists$  constant  $c$ , so that  $\forall \sigma \in \Sigma$   $RT = \# \text{ of primitive operations, executed by quicksort.}$

Goal:  $E[C] = O(n \log n)$

---

Note: Can't apply master method due to [randomness, creating unbalanced subproblems]  
decomposition principle.

[ $A$  = fixed input array].

Notation:  $z_i = i^{\text{th}}$  smallest element of  $A$ . ( $i^{\text{th}}$  order statistic)

e.g.

6	8	10	2
$z_2$	$z_3$	$z_4$	$z_1$

$\Sigma$ : possible pivots

$X$ : # of comparisons made on a particular pivot by quicksort

for  $\sigma \in \Sigma$ , indices  $i < j$ , let

$X_{ij}(\sigma)$  = # of times  $z_i, z_j$  get compared  
in quicksort w/ pivot sequence  $\sigma$

fix two elements of the input array.  
 How many times can these two elements get compared  
 with each other during the execution of Quicksort?  
 [0 or 1].

Note: in the partition subroutine  $\Rightarrow$  comparison happens  
 & it always involves the pivot element  
 (all comparison involves)  
 a pivot element

Note: the comparison φ happen twice, why?

as the two elements being compared only when one is the pivot  
 which is then excluded from future recursive calls.

therefore: each  $X_{ij}$  is an "indicator", i.e (0-1) random variable.

A decomposition approach.

$\text{so: } C(\sigma) = \# \text{ of comparisons between input elements.}$

$X_{ij}(\sigma) = \# \text{ of comparisons between } z_i \text{ and } z_j$

Thus:  $\forall \sigma, C(\sigma) = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{ij}(\sigma)$

Linearity of expectation.

The expectation of a sum = the sum of the expectations  
 whether or not the random variables independent or not

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}]$$

comlicated

simpler

$$\text{since } E[X_{ij}] = 0 \cdot \Pr[X_{ij}=0] + 1 \cdot \Pr[X_{ij}=1]$$

$$E[X_{ij}] = \Pr[X_{ij}>0]$$

$$\text{thus } E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \Pr[z_i, z_j \text{ get compared}] \quad *$$

## General decomposition principle

- 1). Identify random variables,  $Y$ , that you care about.  
e.g # of comparisons in quicksort
- 2). Express  $Y$  as sum of indicator random variables

$$Y = \sum_{l=1}^m X_l$$

- 3). Apply Linearity of expectation.

$$E[Y] = \sum_{l=1}^m \Pr[X_l = 1]$$


---

QuickSort

$C(\sigma) = \# \text{ of comparisons quicksort makes w/pivot } \sigma$   
 $X_{ij}(\sigma) = \# \text{ of times } z_i, z_j \text{ get compared}$

Recall:  $E[C] = \sum_{\bar{i}=1}^{n-1} \sum_{\bar{j}=\bar{i}+1}^n \Pr[X_{\bar{i}\bar{j}} = 1] = \Pr[z_i, z_j \text{ get compared}]$

$\bar{i}, \bar{j}$  th smallest entries in array

key claim:  $\forall \bar{i} < \bar{j}, \Pr[z_i, z_j \text{ get compared}] = \frac{2}{(\bar{j}-1+\bar{i})}$

Proof:: Fix  $z_i, z_j$  with  $\bar{i} < \bar{j}$

(consider the set  $z_i, z_{i+1}, \dots, z_{j-1}, z_j \Rightarrow$  contiguous order of statistics  
 Inductively:  $\hookrightarrow (\bar{j}-\bar{i}+1)$  elements)

Note:  $j-i+1 \Rightarrow$  parallelizing

as long as none of these are chosen as a pivot,  
 all are passed to the same recursive call.

e.g.:  $X_{\bar{j}-\bar{i}+1} = \{z_3, z_4, z_5, z_1, z_7\}$

pivot  
 $z_1$  or  $z_2$

pivot  $z^*$  (greater than all elements).

\*  $X_{\bar{j}-\bar{i}+1}$  all show up in the left

\*  $X_{\bar{j}-\bar{i}+1}$  all in the right of the pivot  $\hookrightarrow$  passed to 1<sup>st</sup> recursive call

- Consider the first among  $z_i, z_{i+1}, \dots, z_{j-1}, z_j$  that gets chosen as a pivot.
- ① if  $z_i$  or  $z_j$  gets chosen first, then  $z_i$  &  $z_j$  get compared.
  - ② if one of  $z_{i+1}, \dots, z_{j-1}$  gets chosen first, then  $z_i$  &  $z_j$  are never compared. [split into two different recursive calls].

Note: since pivots always chosen uniformly at random,

$$\Pr[z_i, z_j \text{ gets compared}] \rightarrow \text{Case one.}$$

$\frac{\# \text{ choices that lead to case 1}}{\# \text{ total choices}}$

each of  $z_i, z_{i+1}, \dots, z_{j-1}, z_j$  is equally likely to be the first.

$(j-i+1) \sim \# \text{ total choices.}$

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{(j-i+1)}$$

$$E[C] = 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{1}{(j-i+1)}$$

$\underbrace{\quad}_{\cdot O(n^2) \text{ terms}}$

Note: for each fixed  $i$ , the inner sum

$$\sum_{j=i+1}^n \frac{1}{(j-i+1)} \approx \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots$$

$\frac{1}{n^2}$  How many terms are there?

Note: as  $j$  increases to  $n$ , the  $\frac{1}{j}$  becomes smaller so the most significant is in the first term, i.e.:  $\frac{1}{n+1}$

$$\frac{1}{n+1} \approx \frac{1}{2}.$$

$$E[C] = 2 \sum_{i=1}^n \sum_{j=i+1}^{n-i} \frac{1}{(j-i+1)}$$

For each fixed  $i$ , the sum is

$$\sum_{j=i+1}^n \frac{1}{j-i+1} = \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$$

$\leq n$  choice for  $j$

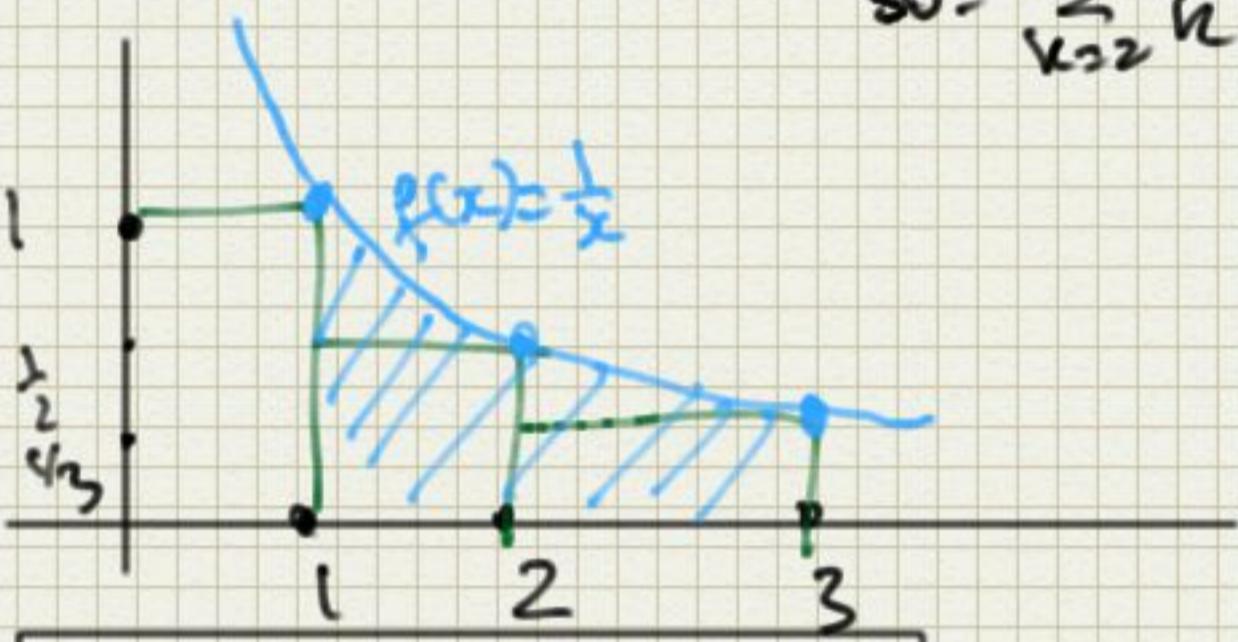
$$E[C] \leq 2 \cdot n \cdot \sum_{k=2}^n \frac{1}{k}$$

$\leq n$  choice for  $i$   
(actually  $(n-1)$  choice for  $i$ )

$$E[C] \leq 2n \sum_{k=2}^n \frac{1}{k}$$

| claim:  $\sum_{k=2}^n \frac{1}{k} \leq \ln n$

Proof of claim:



$$\text{So: } \sum_{k=2}^n \frac{1}{k} \leq \int_1^n \frac{1}{x} dx$$

$$= \ln x \Big|_1^n$$

$$= \ln n - \ln 1$$

$$\leq \underline{\ln n}$$

$$E[C] \leq 2n \ln n.$$