Divide and Conquer, Sorting & Searching & Randomized Algorithms.

- Tim Roughgarden

Integer Multiplication.

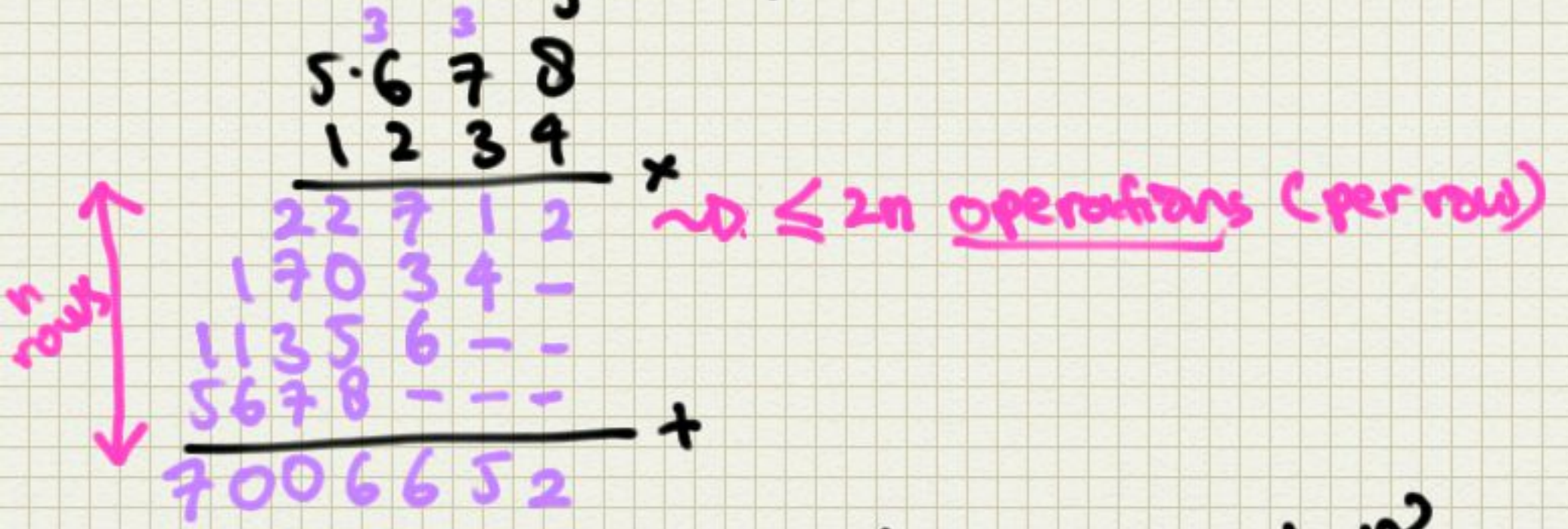
Input: two n-digit numbers x and y.

Output: the product $x \cdot y$

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

---

Grade-school algorithm,



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

Karatsuba Multiplication.

$x = 56\ 78$ (a, b)
$y = 12\ 34$ (c, d)

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 ③ − ② − ① $= 2840$

step 5 :

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

---

write $x = 10^{n/2} a + b$ and $y = 10^{n/2} c + d$

, write a, b, c, d are $n/2$-digit numbers

[example $a = 56$, $b = 78$, $c = 12$, $d = 34$]

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

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

Idea: recursively compute ac, ad, bc, bd

, then compute $(*)$ in the straightforward way

base case. (simple base case omitted).

for 1-digit number, each → 1 basic multiplication

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

Step 1: recursively compute $a \cdot c$

Step 2: recursively compute bd.

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

Gauss's trick ③ − ① − 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 (3rd edition)
- Mehlhorn / Sanders, Data Structures and Algorithms. The Basic Toolbox, 2008.

Merge Sort

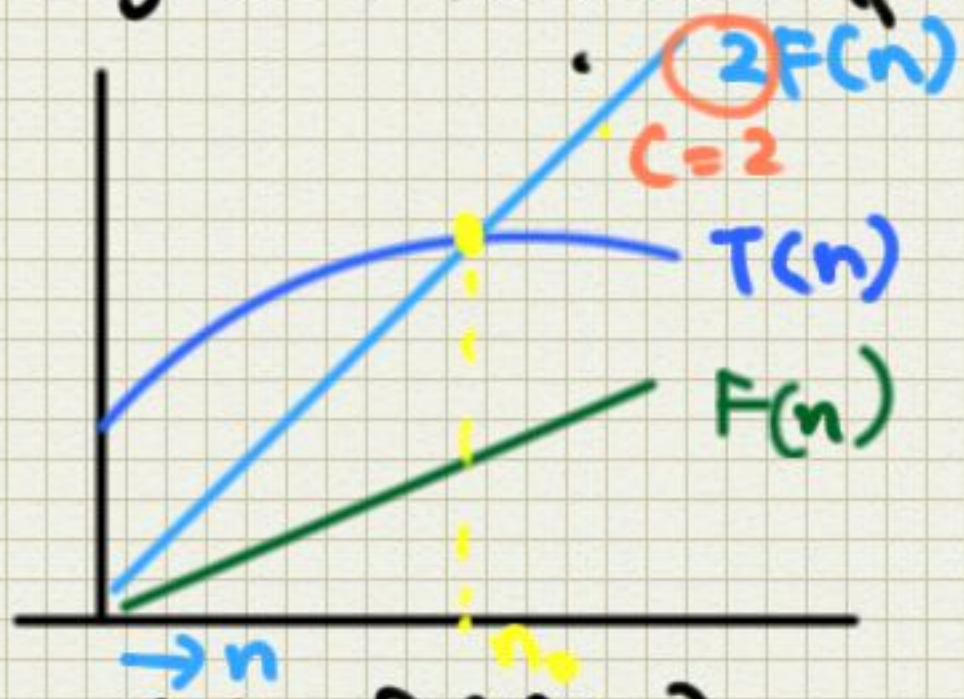
→ Intro to divide-and-conquer algorithm

→ Improvement over { selection, Insertion, Bubble } → sorting method.

algo analysis. ← { worst-case, best-case, average-case (distribution of inputs), benchmark }

domain knowledge ←

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

$$\begin{aligned} T(n) &\leq |a_k| n^k + \dots + |a_1| n + |a_0| \\ &\leq |a_k| n^k + \dots + |a_1| n^k + |a_0| n^k \\ &\leq c_1 \cdot n^k \end{aligned}$$

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

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

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

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

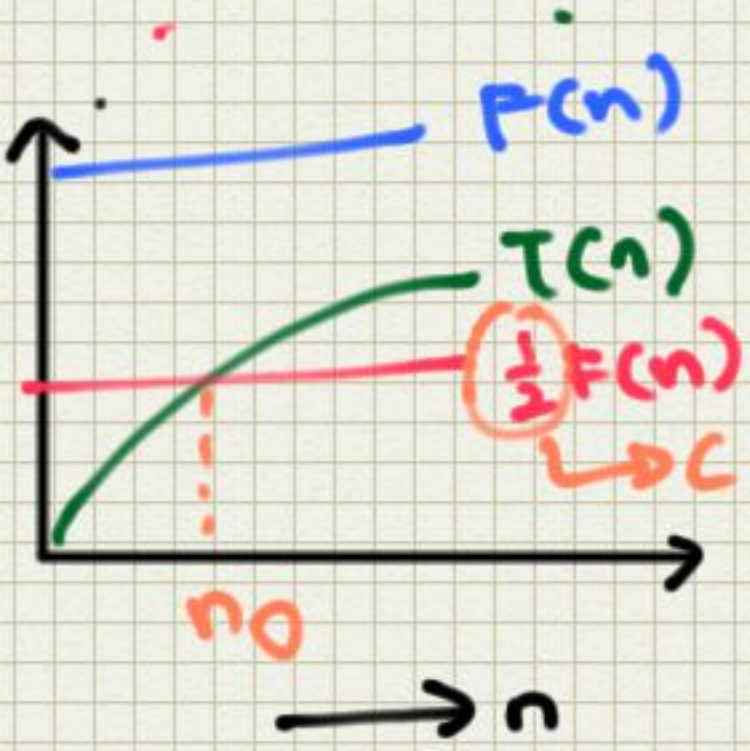
$\Downarrow$

false contradiction.

O (big-O) ≈ less than or equal to

Θ / omega ≈ greater than or equal to

Omega Notation.



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

$T(n) \geq C \cdot f(n) \quad \forall n \geq n_0$

Definition Theta (Θ) 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 <u>all</u> constants $c > 0$, $\exists$ a constant n_0 such that

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

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

proof: need to pick constants c, n_0 such that

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

note: $2^{n+10} \leq c \cdot 2^n$

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

$1024 \cdot 2^n \leq c \cdot 2^n$ (✓)

pick $c = 1024, n_0 = 1$. ✓

↳ $1024 \cdot 2 \leq 1024 \cdot 2$

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

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

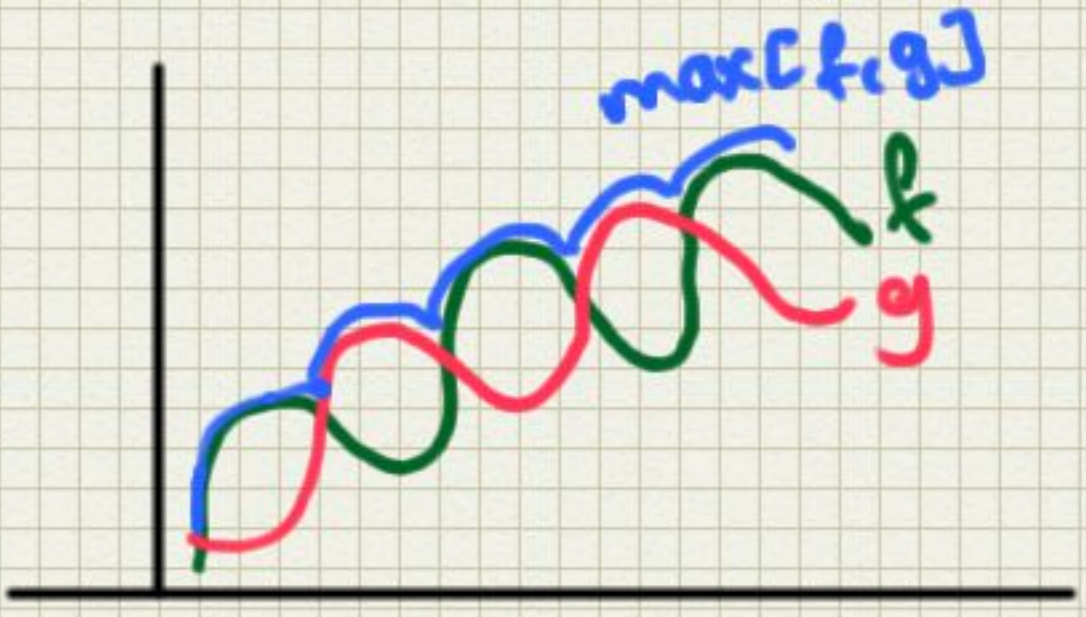
$\exists$ constants $c, n_0 > 0$, such that

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

, But then [[w/ cancelling 2^n]]

$$2^{9n} \leq c \Rightarrow \not\phi \text{ holds (false)}$$

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

$$2\max\{f(n), g(n)\} \geq \tfrac{1}{2}(f(n) + g(n))$$

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

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, ..., n in some arbitrary order.

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

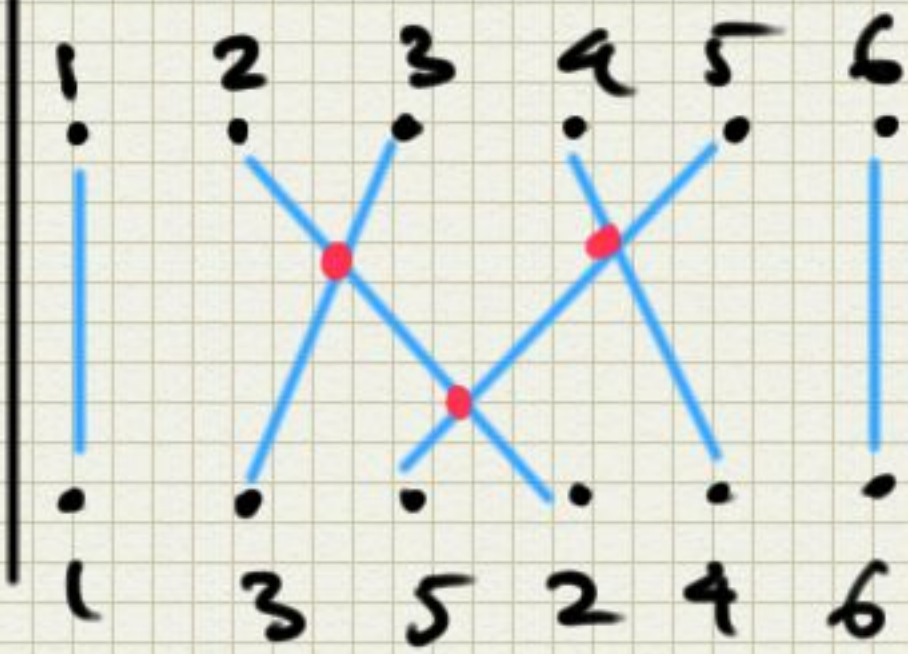
array in sorted order ⇒ # inversion is zero.

eg: (1, 3, 5, 2, 4, 6).

Inversions:

(left entri (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?

→ $\frac{6 \cdot 5}{2} = 15$.

$\binom{n}{2} = \left(\frac{n(n-1)}{2}\right)$

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 $i, j \leq n/2$ } recursively.
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 = count (1st half of A, n/2)
        y = count (2nd half of A, n/2)
        z = 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

↓

max split_inversion (n^2).

Lo the earlier index on the left half of an arr

&

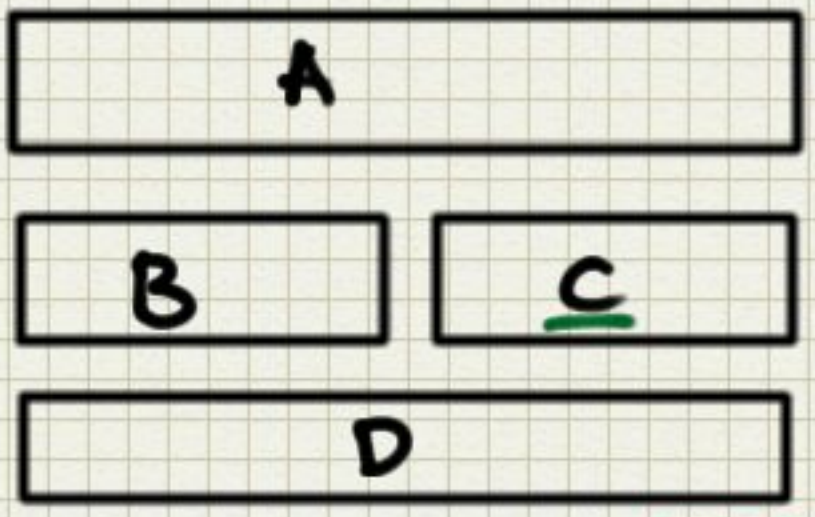
the second index on the right half

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 (1st half of A, n/2)
  (C, y)= sort_count (2nd half of A, n/2).
  (D, z = merge_countSplitInv (B, C)
  return x+y+z.
```

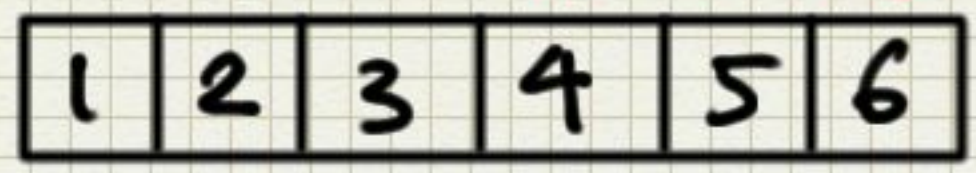


split inversion

When element from C get copied.

e.g. consider merging
eg Call inversions

Output



⇒ 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 2nd array C are precisely the numbers left in the 1st array B when y is copied to the output D.

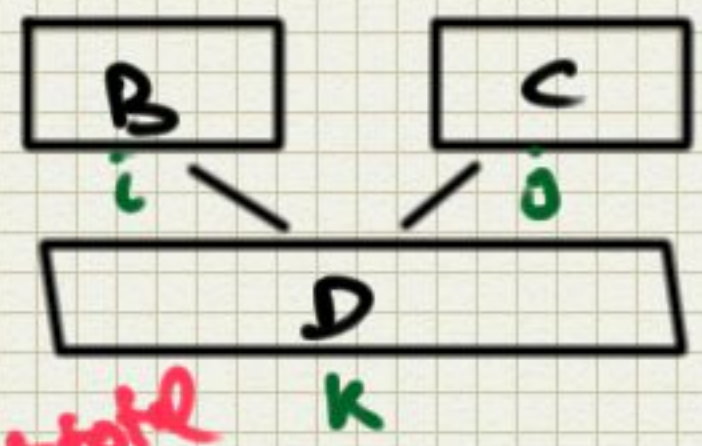
Proof: let x be an element of the 1st array B.

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 ⇒ 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 2nd array C gets copied to output D, increment total by number of elements remaining in 1st array B



Runtime:

→ merge _ count total

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

$c_1 \cdot n$ $c_2 \cdot n = (c_1 + c_2) \cdot n$ ✓

$\Theta(n)$.

Sort-and-Count → $\Theta(n \log n)$.

Matrix Multiplication.

(n x n matrices)

where $z_{ij} = (\text{ith row of } x) \cdot (\text{jth column of } y)$

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

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

---

e.g. (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

$\Theta(n^3)$

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

input size $(n^2) = \Theta(n) \cdot n^2 = \Theta(n^3)$

Applying Divide and conquer.

Idea

write $X = \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)$ and $Y = \left(\begin{array}{c|c} E & F \\ \hline G & H \end{array}\right)$

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

$$X \cdot Y = \left(\begin{array}{c|c} AE+BG & BF+BH \\ \hline CE+CG & CF+DH \end{array}\right)$$

---

step 1: recursively compute the 8 necessary products.
step 2: do the necessary additions

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

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

Strassen'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.

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

The seven products:

$P_1=A(F-H)$, $P_2=(A+B)H$,
$P_3=(C+D)E$, $P_4=D(G-E)$,
$P_5=(A+D)(E+H)$, $P_6=(B-D)(G+H)$,
$P_7=(A-C)(E+F)$

Claim $X\cdot Y=\begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}=\begin{pmatrix} P_5+P_4-P_2+P_6 & P_1+P_2 \\ P_3+P_4 & P_1+P_5-P_3-P_7 \end{pmatrix}$

Proof: $P_5+P_4-P_2+P_6 == AE+BG$? ✓

$AE+AH+DE+DH+DG-DE-AH-BH+BG+BH-DG-DH$

Master method

-A recursive algorithm., recall from the integer multiplication

write $x = 10^{n/2}a + b$, $y = 10^{n/2}c + d$

[where a, b, c, d are $n/2$-digit numbers].

, 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

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

(work done (⊛); work done by recursive calls.)

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

new recurrence:

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

$$\forall\, n > 1 : T(n) \leq 3T(n/2) + \Theta(n)$$

work done in recursive calls

note: for merge sort $T(n) \leq 2T(n/2) + \Theta(n)$.

The Master Method.

→ a "black box" for solving recurrences.

Assumption : all subproblems have equal size

Recurrence Format.

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

② . For all larger n:

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

a : number of recursive calls (# of subproblems) (≥ 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. (≥ 0).

[a, b, d are independent of n]

The master method

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

note $O(n^d \log n)$ → log base does't (?) matter as it will be surpressed by the leading constant, n^d

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

→ log base in the exponent matters because $n^2 < n^3$, ...

The Master Method.

a: # recursive calls
b: size of subproblem

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

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

---

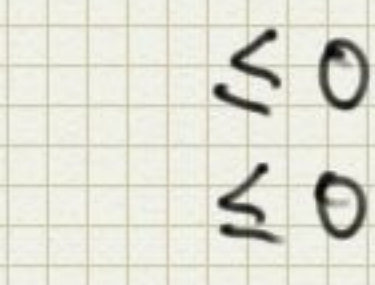
eg merge sort.

$a = 2$
$b = 2$ → half of the original problem $\left.\begin{matrix}\\\\\end{matrix}\right\} b^d = 2$
$d = 1$ → merge ($O(1) \Rightarrow$ linear)

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

---

eg: binary search of a sorted array.



$a = 1$
$b = 2$ } $a = b^d$
$d = 0$ } Case 1 $O(n^d \log n)$

$O(\log n)$.

$d = 0 \Rightarrow$ means constant time.

---

e.g: Integer multiplication ⊗ gauss trick

$a = 4$
$b = 2$ } 2
$d = 1$ }

$a > b^d$ → case 3

$T(n) \leq O(n^{\log_b a})$
$\leq O(n^{\log_2 4})$
$\leq O(n^2)$ ~ similar to high-school trick.

w/ gauss trick

$a = 3$
$b = 2$ } 2
$d = 1$ }

$a > b^d$

$T(n) \leq O(n^{\log_b a})$
$\leq O(n^{\log_2 3})$
$\leq O(n^{\log_2 3})$
$\leq O(n^{1.59})$ (better than $O(n^2)$)

example 5

strassen's matrix multiplication algorithm

$a = 7$
$b = 2$
$d = 2$

$a > b^d \rightarrow$ case 3.

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

↳ linear in matrix size

↓

beat the naive iterative algorithm $O(n^3)$.

Fictious recurrence

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

$a = 2$
$b = 2$
$d = 2$

$a < b^d \rightarrow$ case 2

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

Assume: recurrence is

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

(ii) $T(n) \leq aT(n/b) + cn^d$ (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)

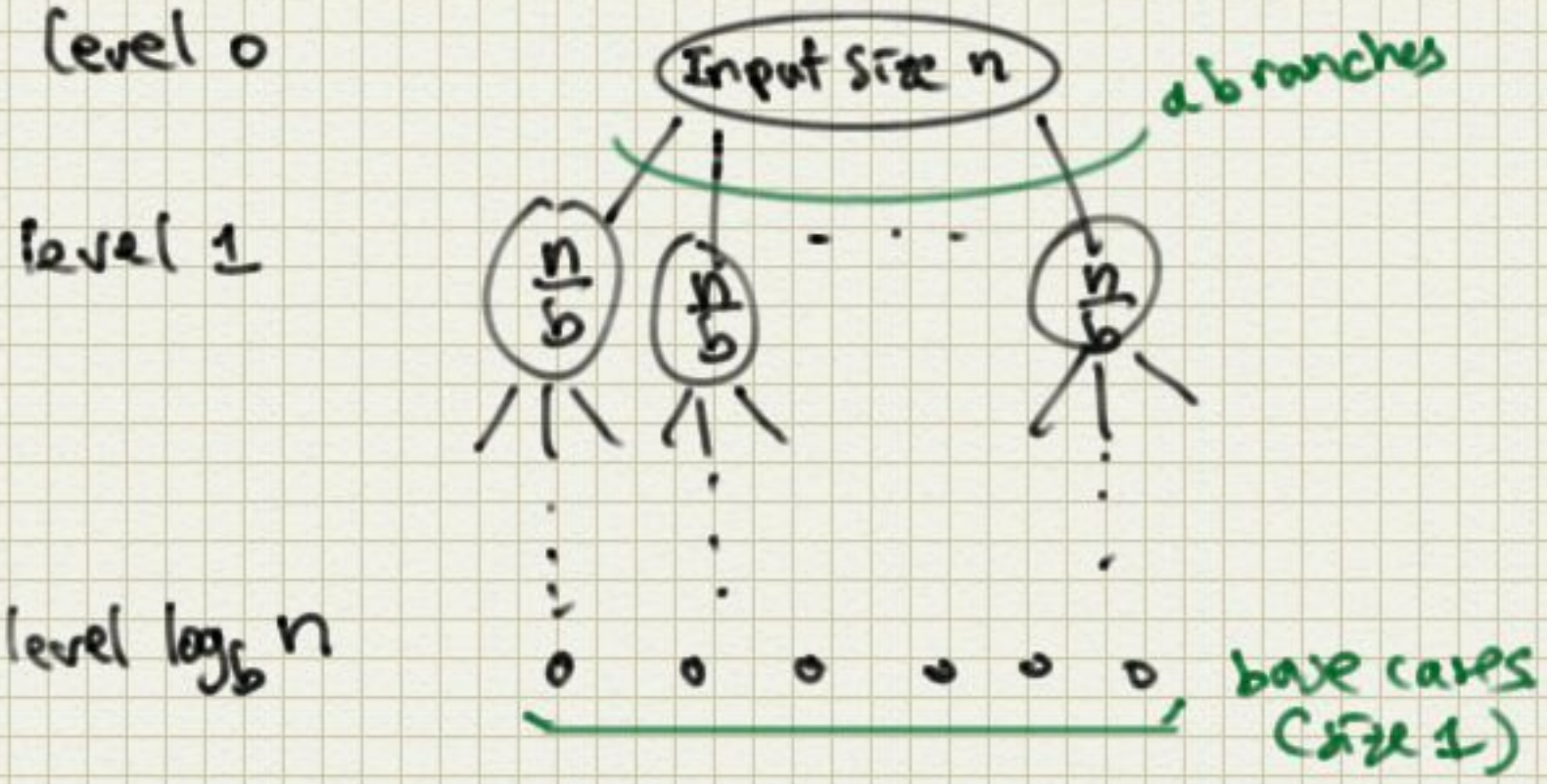
mergesort recursion tree.

at each level $j = 0, 1, 2, \ldots, \log_b n$ < a^j subproblems, each w/ of size n/b^j

↓

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

The recursion tree



---

1 Work at a single level (level j)

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

$\leq a^j \cdot c \cdot \left[\frac{n}{b^j}\right]^d$ — work per level-j subproblem

a^j: # of level-j subproblems

$\frac{n}{b^j}$: size input of each level j subproblem.

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

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

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

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

How to Think About (*)

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

Interpretation (tug of good k evil)

a = rate of subproblem proliferation. (RSP) → function of j

log(force of evil) as we go down to the tree

↓

more subproblems appear

b^d = rate of work shrinkage (RWS) → function of j

Why b^d? b is a factor by which on input size shrinked w/ the recursion level j

e.g. $b=2$ → so the next subproblem is half the previous level

d → the amount of work solving that subproblem.

$a=d=2$, $b=2$

$d=1$, $b=2$

↓

→ recursive half of the input w/ a linear work.

↓

- input half
- work also decrease by half

$\frac{1}{2} = 50\%$ ↓

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

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:

upperbound for level $j = cn^d \times \left(\frac{a}{b^d}\right)^j$

① . RSP = RWS ⇒ same amount of work each level.
(eg: merge sort) → $n^d \cdot \log n$. $\Theta(n^d \log n)$ → merge sort.

② . RSP < RWS → less amount each level ⇒
most work at the root (dominance) ~ $\Theta(n^d)$.

③ . RSP > RWS ⇒ more work each level ⇒
most work at the leaves [might expect Θ(# leaves)].

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

if $a = b^d$, then

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

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

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

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

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

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

Basic sum fact $\boxed{r = \frac{a}{b^d}}$

$r > 0, r \neq 1$

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

upshot

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

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

≤ 2.

(i.e) the 1st term of sum dominates. ↗ independent of k.

② If $r > 1$ is constant, RHS is $\leq r^k \cdot \left(1 + \frac{1}{r-1}\right)$.

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

(the last term of sum dominates).

Case 2

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

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

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

↳ $r < 1$.

↳ $\leq$ a constant (independent of n)

$\leq cn^d * c$

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

Case 3, if $a > b^d$

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

↳ $r > 1$.

$\leq cn^d * C$. largest term.

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

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

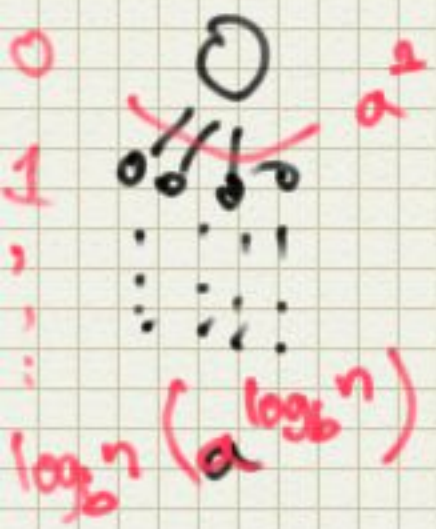
$\leq (b^{-d})^{\log_b n}$

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

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

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

$\leq O(a^{\log_b n})$ ⇒ # of leaves



Note: $\boxed{a^{\log_b n}} = \boxed{n^{\log_b a}}$ → simpler to apply

$[\log_b n \log_b a = \log_b a \log_b n.]$

→ more intuitive.