COL 351 : Analysis & Design of Algorithms

LECTURE 5

DIVIDE & CONQUER IV :

Counting Inversions (contd.), Matrix Multiplication and Master Theorem

July 31, 2024 | Rohit Vaish

COUNTING INVERSIONS

input: a length n array A of distinct integers

output: the number of inversions of A

COUNTING INVERSIONS

input: a length n array A of distinct integers

output: the number of inversions of A

inversions: pairs (i, j) of array indices with $i < j$ and $A[i] > A[j]$.

COUNTING INVERSIONS

Brute force algorithm: check every pair of indices (i, j)

$\Theta(n^2)$

COUNTING INVERSIONS

Brute force algorithm: check every pair of indices (i, j)

$\Theta(n^2)$

Can we do better?

Yes! $O(n \log n)$ algorithm via divide-and-conquer.

COUNTING INVERSIONS

Call an inversion (i, j) where $i < j$

left inversion if $i, j \leq n/2$

right inversion if $i, j > n/2$

split inversion if $i \leq \frac{n}{2} < j$

COUNTING INVERSIONS

Call an inversion (i, j) where $i < j$

left inversion if $i, j \leq n/2$

right inversion if $i, j > n/2$

Compute these recursively

split inversion if $i \leq \frac{n}{2} < j$ ← Compute these in "combine" step

HIGH-LEVEL ALGORITHM

input: an array A of n distinct integers

output: the number of inversions of A

HIGH-LEVEL ALGORITHM

input: an array A of n distinct integers

output: the number of inversions of A

if $n \leq 1$ return 0

HIGH-LEVEL ALGORITHM

input: an array A of n distinct integers

output: the number of inversions of A

if $n \leq 1$ return 0

else ℓ := recursively count inversions on left half of A

r := " " " right " "

s := count split inversions of A

return $\ell + r + s$

HIGH-LEVEL ALGORITHM

input: an array A of n distinct integers

output: the number of inversions of A

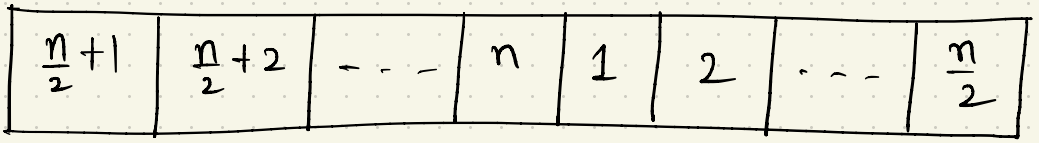
if $n \leq 1$ return 0

else ℓ := recursively count inversions on left half of A

r := " " " right " "

s := Count split inversions of A

return $\ell + r + s$

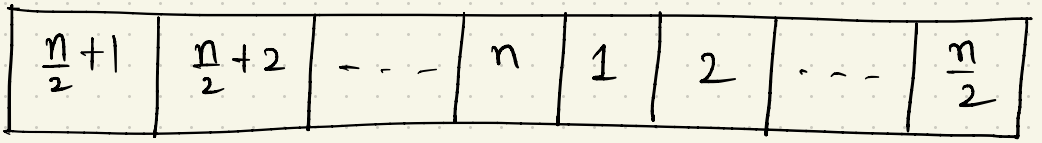
A = 

split inversions = ?

$A =$

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

split inversions $= n^2/4$

A = 

split inversions = $n^2/4$

Possible to compute split inversions in O(n) time ?

suffices for O(n log n) time overall

Piggyback on Merge Sort

Piggyback on Merge Sort

Suppose A has no split inversion.

Piggyback on Merge Sort

Suppose A has no split inversion.

Then, every element in left half of A $<$ every element in right half of A

Piggyback on Merge Sort

Suppose A has no split inversion.

Then, every element in left half of A $<$ every element in right half of A

What does merge subroutine do for such array?

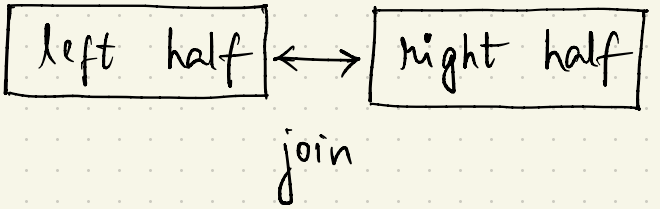
Piggyback on Merge Sort

Suppose A has no split inversion.

Then, every element in left half of A $<$ every element in right half of A

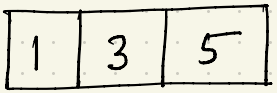
What does merge subroutine do for such array?

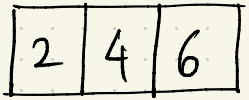
Concatenation!



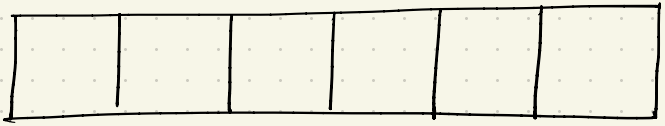
What does merge subroutine do when there are split inversions?

What does merge subroutine do when there are split inversions?

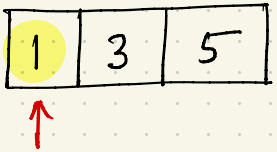
Consider merging 

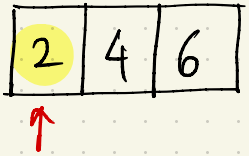
and 

.

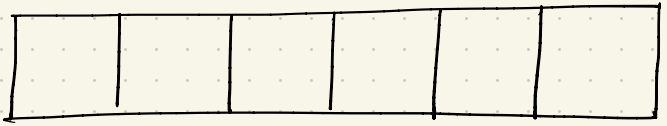
Output 

What does merge subroutine do when there are split inversions?

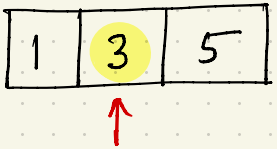
Consider merging 

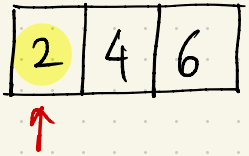
and 

.

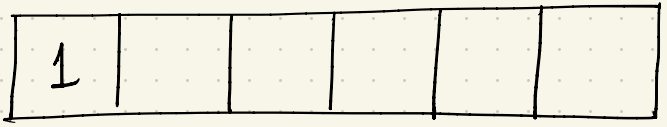
Output 

What does merge subroutine do when there are split inversions?

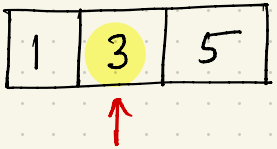
Consider merging 

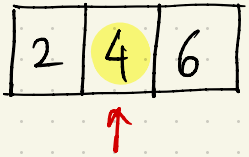
and 

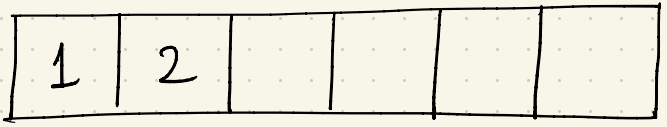
.

Output 

What does merge subroutine do when there are split inversions?

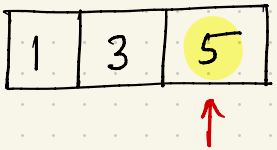
Consider merging 

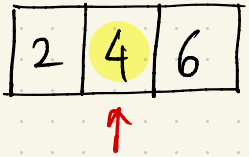
and 

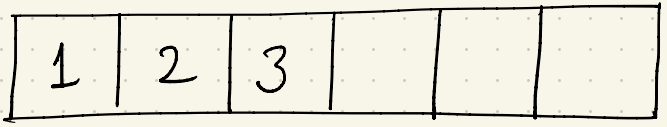
Output 

When "2" gets copied in output, the split inversions (3,2) and (5,2) are exposed.

What does merge subroutine do when there are split inversions?

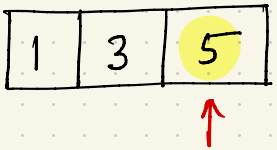
Consider merging 

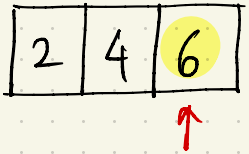
and 

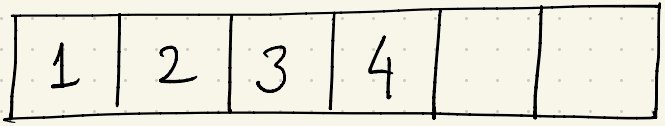
Output 

When "2" gets copied in output, the split inversions (3,2) and (5,2) are exposed.

What does merge subroutine do when there are split inversions?

Consider merging 

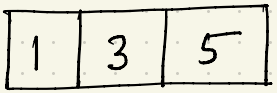
and 

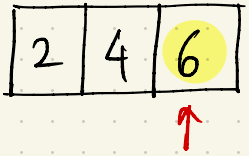
Output 

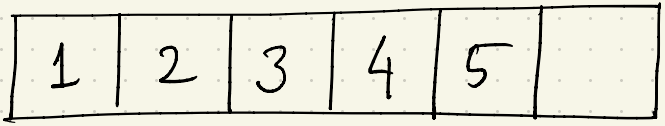
When "2" gets copied in output, the split inversions (3,2) and (5,2) are exposed.

When "4" gets copied in output, the split inversion (5,4) is exposed.

What does merge subroutine do when there are split inversions?

Consider merging 

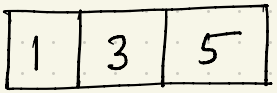
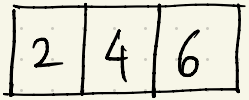
and 

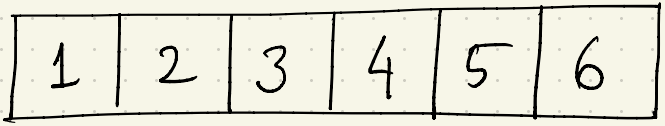
Output 

When "2" gets copied in output, the split inversions (3,2) and (5,2) are exposed.

When "4" gets copied in output, the split inversion (5,4) is exposed.

What does merge subroutine do when there are split inversions?

Consider merging  and  .

Output 

When "2" gets copied in output, the split inversions (3,2) and (5,2) are exposed.

When "4" gets copied in output, the split inversion (5,4) is exposed.

Claim: Let A be an array, and let L and R be sorted versions of first and second halves of A.

The elements $x \in L$ and $y \in R$ form a split inversion

if and only if

merge (L, R) copies y to the output before x.

Claim: Let A be an array, and let L and R be sorted versions of first and second halves of A.

The elements $x \in L$ and $y \in R$ form a split inversion

if and only if

merge (L, R) copies y to the output before x.

Proof: Since $x \in L$ and $y \in R$

(x, y) split inversion $\iff$ $y < x$ $\iff$ y copied before x.

ALGORITHM FOR COUNTING INVERSIONS

input: an array A of n distinct integers

output: the number of inversions of A

if $n \leq 1$ return 0

else l := recursively count inversions on left half of A

r := " " " right " "

s := count split inversions of A

return $l + r + s$

ALGORITHM FOR COUNTING INVERSIONS

input: an array A of n distinct integers

output: the number of inversions of A

if $n \leq 1$ return 0

else l := recursively count inversions on left half of A

r := " " " right " "

s := count split inversions of A

return $l + r + s$

ALGORITHM FOR COUNTING INVERSIONS

input: an array A of n distinct integers

output: the number of inversions of A

if $n \leq 1$ return 0

else $(L, l) :=$ recursively sort and count inversions on left half of A

$r :=$ " " " right " "

$s :=$ count split inversions of A

return $l + r + s$

ALGORITHM FOR COUNTING INVERSIONS

input: an array A of n distinct integers

output: the number of inversions of A

if $n \leq 1$ return 0

else $(L, l) :=$ recursively sort and count inversions on left half of A

$r :=$ " " " right " "

$s :=$ count split inversions of A

return $l + r + s$

ALGORITHM FOR COUNTING INVERSIONS

input: an array A of n distinct integers

output: the number of inversions of A

if $n \leq 1$ return 0

else $(L, \ell) :=$ recursively Sort and count inversions on left half of A

$(R, r) :=$ " Sort and " " right " "

$s :=$ Count split inversions of A

return $\ell + r + s$

ALGORITHM FOR COUNTING INVERSIONS

input: an array A of n distinct integers

output: the number of inversions of A

if $n \leq 1$ return 0

else $(L, \ell) :=$ recursively Sort and count inversions on left half of A

$(R, r) :=$ " Sort and " " right " "

$s :=$ Count split inversions of A

return $\ell + r + s$

ALGORITHM FOR COUNTING INVERSIONS

input: an array A of n distinct integers

output: the number of inversions of A

if $n \leq 1$ return 0

else (L, ℓ) := recursively (sort and) count inversions on left half of A

(R, r) := " (sort and) " " right " "

(B, s) := (merge and) count split inversions of A (using L and R)

return $\ell + r + s$

ALGORITHM FOR COUNTING INVERSIONS

input: an array A of n distinct integers

output: the number of inversions of A

if $n \leq 1$ return 0

else (L, ℓ) := recursively (sort and) count inversions on left half of A

(R, r) := " (sort and) " " right " "

(B, s) := (merge and) count split inversions of A (using L and R)

return $\ell + r + s$

How to implement this in linear time?

RECALL MERGE SUBROUTINE

```
initialize i := 1
initialize j := 1
for k := 1 to n
        if L[i] < R[j] then
                B[k] := L[i]
                increment i by 1
        else
                B[k] := R[j]
                increment j by 1

return B
```

MERGE-AND-COUNT-SPLIT-INVERSIONS

```
initialize i := 1
initialize j := 1
for k := 1 to n
        if L[i] < R[j] then
                B[k] := L[i]
                increment i by 1
        else
                B[k] := R[j]
                increment j by 1

return B
```

MERGE-AND-COUNT-SPLIT-INVERSIONS

initialize $s := 0$
initialize $i := 1$
initialize $j := 1$
for $k := 1$ to n
    if $L[i] < R[j]$ then
        $B[k] := L[i]$
        increment i by 1
    else
        $B[k] := R[j]$
        increment j by 1
        $s := s + \underbrace{\left(\frac{n}{2} - i + 1\right)}_{\#\text{ left in } L}$
return (B, s)

Theorem: For every input array A of length $n \geq 1$, our algorithm correctly computes the number of inversions of A and runs in $O(n \log n)$ time.

Theorem: For every input array A of length $n \geq 1$, our algorithm correctly computes the number of inversions of A and runs in $O(n \log n)$ time.

Proof: Exercise

MATRIX MULTIPLICATION

MATRIX MULTIPLICATION

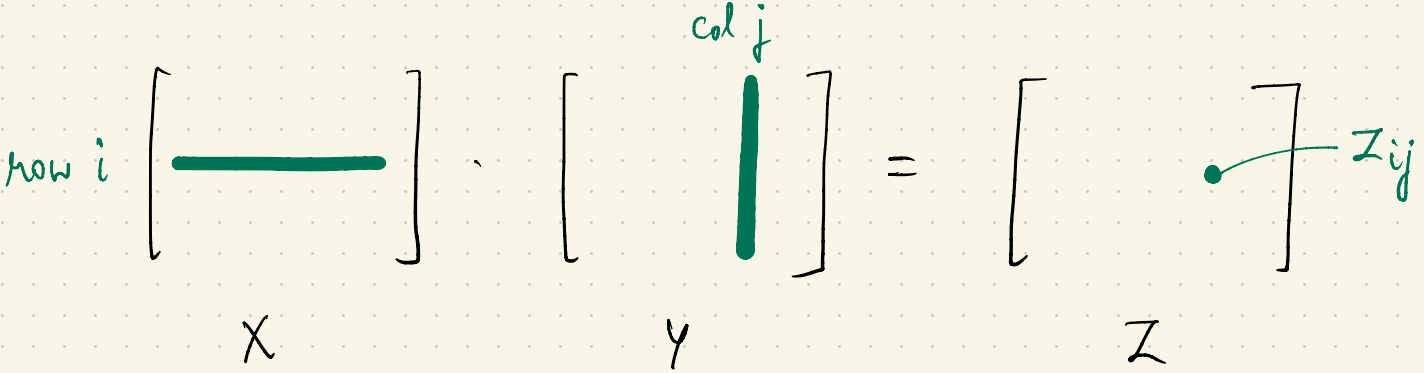
input: Two $n \times n$ matrices X and Y of integers

Output: the product $Z = X \cdot Y$ where $z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}$

MATRIX MULTIPLICATION

input: Two $n \times n$ matrices X and Y of integers

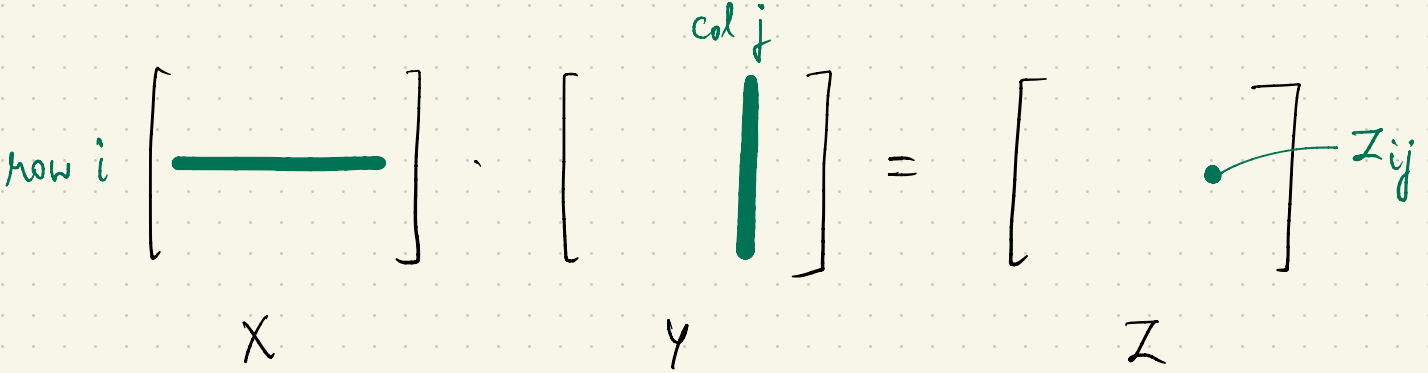
Output: the product $Z = X \cdot Y$ where $z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}$



MATRIX MULTIPLICATION

input: Two $n \times n$ matrices X and Y of integers

output: the product $Z = X \cdot Y$ where $z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj}$



input size: $O(n^2)$

MATRIX MULTIPLICATION

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} & \\ & \end{bmatrix}$$

X Y Z

MATRIX MULTIPLICATION

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & \\ & \end{bmatrix}$$

X Y Z

MATRIX MULTIPLICATION

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ & \end{bmatrix}$$

X Y Z

MATRIX MULTIPLICATION

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

X Y Z

MATRIX MULTIPLICATION

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

X Y Z

MATRIX MULTIPLICATION

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

X Y Z

Running time for $n \times n$ matrices : ?

MATRIX MULTIPLICATION

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

X Y Z

Running time for $n \times n$ matrices : $O(n^3)$

assuming multiplication/addition of two numbers takes $O(1)$ time

MATRIX MULTIPLICATION

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{X} \cdot \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{Y} = \underbrace{\begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}}_{Z}$$

Running time for $n \times n$ matrices : $O(n^3)$

Can we do better?

APPLYING DIVIDE & CONQUER

APPLYING DIVIDE & CONQUER

$$X = \left[\begin{array}{c|c} L & R \end{array}\right]_{n \times n}$$

non-square matrices

APPLYING DIVIDE & CONQUER

$$X = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]_{n \times n}$$

A, B, C, D are $\frac{n}{2} \times \frac{n}{2}$ matrices

(assuming n is even)

APPLYING DIVIDE & CONQUER

$$X = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]_{n \times n} \qquad Y = \left[\begin{array}{c|c} E & F \\ \hline G & H \end{array}\right]_{n \times n}$$

APPLYING DIVIDE & CONQUER

$$X = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]_{n \times n} \qquad Y = \left[\begin{array}{c|c} E & F \\ \hline G & H \end{array}\right]_{n \times n}$$

Then, $$X \cdot Y = \begin{bmatrix} A \cdot E + B \cdot G & A \cdot F + B \cdot H \\ C \cdot E + D \cdot G & C \cdot F + D \cdot H \end{bmatrix}$$

verify!

Recursive Algorithm

$$X \cdot Y = \begin{bmatrix} A \cdot E + B \cdot G & A \cdot F + B \cdot H \\ C \cdot E + D \cdot G & C \cdot F + D \cdot H \end{bmatrix}$$

RECURSIVE ALGORITHM

Step 1 :- Recursively compute the eight matrix products

$$X \cdot Y = \begin{bmatrix} A \cdot E + B \cdot G & A \cdot F + B \cdot H \\ C \cdot E + D \cdot G & C \cdot F + D \cdot H \end{bmatrix}$$

RECURSIVE ALGORITHM

Step 1 : Recursively compute the eight matrix products

$$X \cdot Y = \begin{bmatrix} A \cdot E + B \cdot G & A \cdot F + B \cdot H \\ C \cdot E + D \cdot G & C \cdot F + D \cdot H \end{bmatrix}$$

Step 2 : Do the necessary additions ($\theta(n^2)$ time)

RECURSIVE ALGORITHM

Step 1: Recursively compute the eight matrix products

$$X \cdot Y = \begin{bmatrix} A \cdot E + B \cdot G & A \cdot F + B \cdot H \\ C \cdot E + D \cdot G & C \cdot F + D \cdot H \end{bmatrix}$$

Step 2: Do the necessary additions ($\Theta(n^2)$ time)

Fact: Above algorithm has running time $\Theta(n^3)$ (by Master theorem)

Recursive Algorithm

Step 1: Recursively compute the eight matrix products

$$X \cdot Y = \begin{bmatrix} A \cdot E + B \cdot G & A \cdot F + B \cdot H \\ C \cdot E + D \cdot G & C \cdot F + D \cdot H \end{bmatrix}$$

Step 2: Do the necessary additions ($\Theta(n^2)$ time)

Fact: Above algorithm has running time $\Theta(n^3)$ (by Master theorem)

Is there a way to save a recursive call?

STRASSEN'S ALGORITHM

STRASSEN'S ALGORITHM

Step 1 : Recursively compute only seven matrix products

STRASSEN'S ALGORITHM

Step 1 : Recursively compute only seven matrix products

Step 2 : Do the necessary additions and subtractions (still $\Theta(n^2)$ time)

STRASSEN'S ALGORITHM

Step 1: Recursively compute only seven matrix products

Step 2: Do the necessary additions and subtractions (still $\Theta(n^2)$ time)

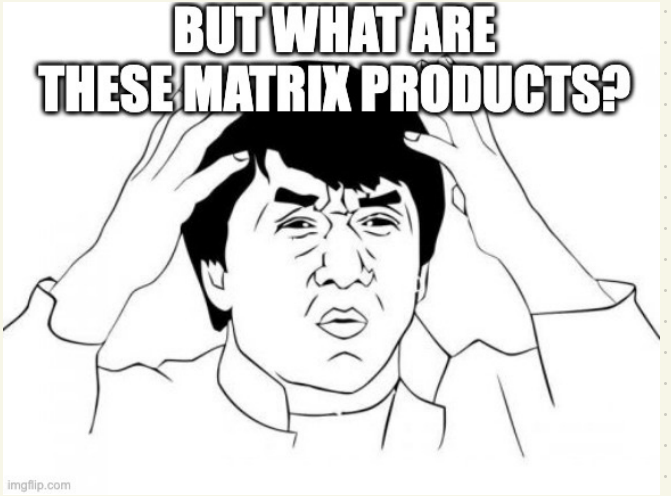
Spoiler alert: Better than $O(n^3)$ time (by Master theorem)

STRASSEN'S ALGORITHM

Step 1 : Recursively compute only seven matrix products

Step 2 : Do the necessary additions and subtractions (still $\Theta(n^2)$ time)

Spoiler alert : Better than $O(n^3)$ time (by Master theorem)



STRASSEN'S ALGORITHM

$$X = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]_{n \times n} \qquad Y = \left[\begin{array}{c|c} E & F \\ \hline G & H \end{array}\right]_{n \times n}$$

$$X \cdot Y = \begin{bmatrix} A \cdot E + B \cdot G & A \cdot F + B \cdot H \\ C \cdot E + D \cdot G & C \cdot F + D \cdot H \end{bmatrix}$$

STRASSEN'S ALGORITHM

$P_1 = A \cdot (F - H)$

$P_2 = (A + B) \cdot H$

$P_3 = (C + D) \cdot E$

$P_4 = D \cdot (G - E)$

$P_5 = (A + D) \cdot (E + H)$

$P_6 = (B - D) \cdot (G + H)$

$P_7 = (A - C) \cdot (E + F)$

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}_{n \times n} \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}_{n \times n}$$

$$X \cdot Y = \begin{bmatrix} A \cdot E + B \cdot G & A \cdot F + B \cdot H \\ C \cdot E + D \cdot G & C \cdot F + D \cdot H \end{bmatrix}$$

STRASSEN'S ALGORITHM

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

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}_{n \times n} \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}_{n \times n}$$

$$X \cdot Y = \begin{bmatrix} A \cdot E + B \cdot G & A \cdot F + B \cdot H \\ C \cdot E + D \cdot G & C \cdot F + D \cdot H \end{bmatrix}$$

Claim: $$X \cdot Y = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

STRASSEN'S ALGORITHM

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

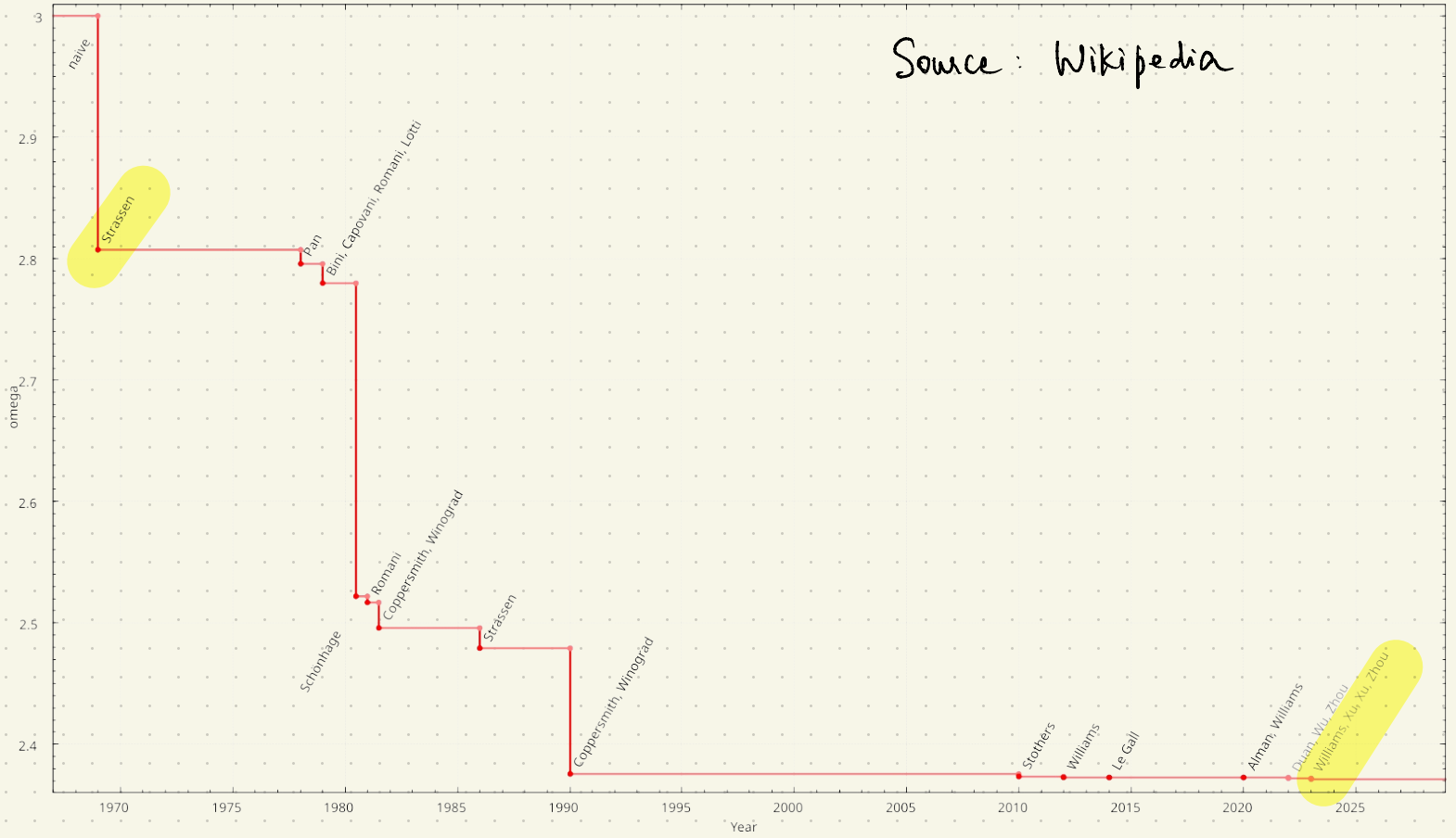
$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}_{n \times n} \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}_{n \times n}$$

$$X \cdot Y = \begin{bmatrix} A \cdot E + B \cdot G & A \cdot F + B \cdot H \\ C \cdot E + D \cdot G & C \cdot F + D \cdot H \end{bmatrix}$$

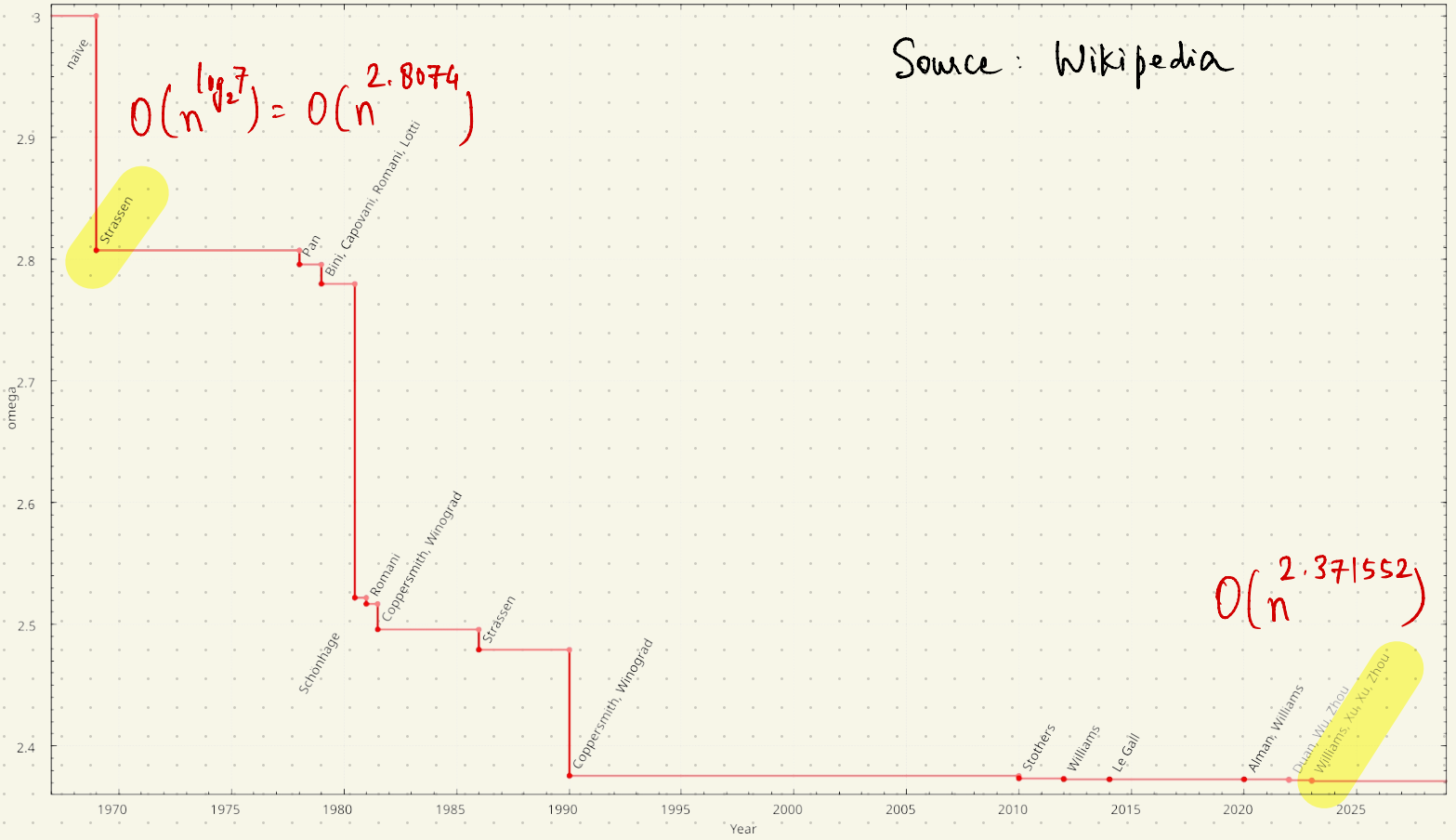
Claim: $X \cdot Y = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$

Strassen's algorithm saves a recursive call!

MATRIX MULTIPLICATION



MATRIX MULTIPLICATION



MASTER METHOD

Black-box for determining the running time of recursive algorithms

RECALL: INTEGER MULTIPLICATION

RECALL: INTEGER MULTIPLICATION

Grade-school multiplication $\Theta(n^2)$

RECALL : INTEGER MULTIPLICATION

Grade-school multiplication $\Theta(n^2)$

Recursive algorithm

$$\left[x \cdot y = 10^{n} ac + 10^{n/2}(ad + bc) + bd \right]$$

RECALL : INTEGER MULTIPLICATION

Grade-school multiplication $\Theta(n^2)$

Recursive algorithm

$$\left[x \cdot y = 10^{n} ac + 10^{n/2}(ad + bc) + bd \right]$$

$T(n)$

maximum number of operations the algorithm needs to multiply two n-digit numbers

RECALL : INTEGER MULTIPLICATION

Grade-school multiplication $\Theta(n^2)$

Recursive algorithm $T(n) \leq 4 \cdot T(n/2) + O(n)$

$$\left[x \cdot y = 10^{n} ac + 10^{n/2}(ad + bc) + bd \right]$$

RECALL : INTEGER MULTIPLICATION

Grade-school multiplication $\Theta(n^2)$

Recursive algorithm $T(n) \leq 4 \cdot T(n/2) + O(n)$

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

Karatsuba algorithm $T(n) \leq 3 \cdot T(n/2) + O(n)$

$$[(a+b)(c+d) - ac - bd = ad + bc]$$

MASTER METHOD

Black-box for solving recurrences

Key assumption: All subproblems have the same size.

for unbalanced subproblems and more: Akra-Bazzi method

STANDARD RECURRENCE FORMAT

STANDARD RECURRENCE FORMAT

Base case:

General case:

STANDARD RECURRENCE FORMAT

Base case: $T(n) \leq$ constant for all sufficiently small n

General case:

STANDARD RECURRENCE FORMAT

Base case: $T(n) \leq$ constant for all sufficiently small n

General case: for all larger n

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

STANDARD RECURRENCE FORMAT

Base case: $T(n) \leq$ constant for all sufficiently small n

General case: for all larger n

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

- a : number of recursive calls (≥ 1)
- b : factor by which input size shrinks (> 1)
- d : exponent in running time of "combine" step (≥ 0)

STANDARD RECURRENCE FORMAT

Base case: $T(n) \leq$ constant for all sufficiently small n

General case: for all larger n

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

independent of n

- a : number of recursive calls (≥ 1)
- b : factor by which input size shrinks (> 1)
- d : exponent in running time of "combine" step (≥ 0)

MASTER THEOREM

Theorem: If $T(n) \leq a\,T\left(\frac{n}{b}\right) + O(n^d)$, then

MASTER THEOREM

Theorem: If $T(n) \leq a\,T\left(\frac{n}{b}\right) + O(n^d)$, then

$$T(n) = \begin{cases} & \text{Case 1} \\ & \text{Case 2} \\ & \text{Case 3} \end{cases}$$

MASTER THEOREM

Theorem: If $T(n) \leq a\,T\left(\frac{n}{b}\right) + O(n^d)$, then

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

MASTER THEOREM

Theorem: If $T(n) \leq a\,T\left(\frac{n}{b}\right) + O(n^d)$, then

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

MASTER THEOREM

Theorem: If $T(n) \leq a\,T\left(\frac{n}{b}\right) + O(n^d)$, then

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

only upper bounds

Examples

EXAMPLES

Merge Sort

EXAMPLES

Merge Sort

$$T(n) \leq 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(n)$$

EXAMPLES

Merge Sort

$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$

$a = 2, b = 2, d = 1$

(Case 1)

EXAMPLES

Merge Sort

$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$

$a = 2, b = 2, d = 1$

(Case 1)

$O(n \log n)$

BINARY SEARCH

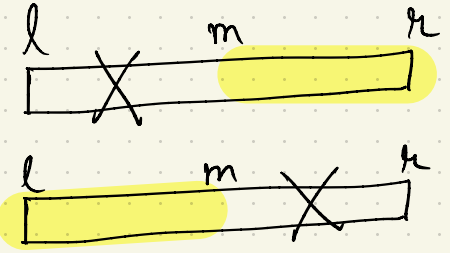
input: a sorted array A of length n and a target T

output: index of T in A (if exists)

BINARY SEARCH

input: a sorted array A of length n and a target T

output: index of T in A (if exists)



BINARY SEARCH

input: a sorted array A of length n and a target T

output: index of T in A (if exists)

initialize $\ell := 0$

initialize $r := n-1$

initialize $m := \left\lfloor \frac{\ell + r}{2} \right\rfloor$

if $A[m] < T$

recurse with $\ell := m+1$

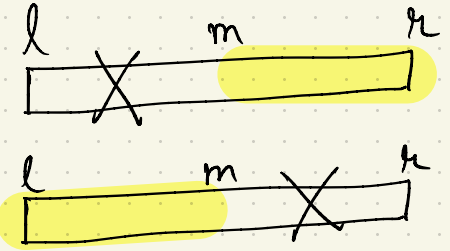
else if $A[m] > T$

recurse with $r := m-1$

else

return m

return No



EXAMPLES

Merge Sort

$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$

$a = 2, b = 2, d = 1$

(Case 1)

$O(n \log n)$

Binary Search

$T(n) \leq T\left(\frac{n}{2}\right) + O(1)$

$a = 1, b = 2, d = 0$

(Case 1)

$O(\log n)$

EXAMPLES

Recursive integer mult^n

$T(n) \leq 4T\left(\frac{n}{2}\right) + O(n)$

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

(Case 3)

$O\left(n^{\log_2 4}\right) = O(n^2)$

same as grade-school algorithm

Karatsuba mult^n

$T(n) \leq 3T\left(\frac{n}{2}\right) + O(n)$

$a = 3, b = 2, d = 0$

(Case 3)

$O\left(n^{\log_2 3}\right) = O(n^{1.59})$

faster!

Strassen's algorithm

$T(n) \leq 7T\left(\frac{n}{2}\right) + O(n)$

$a = 7, b = 2, d = 1$

(Case 3)

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