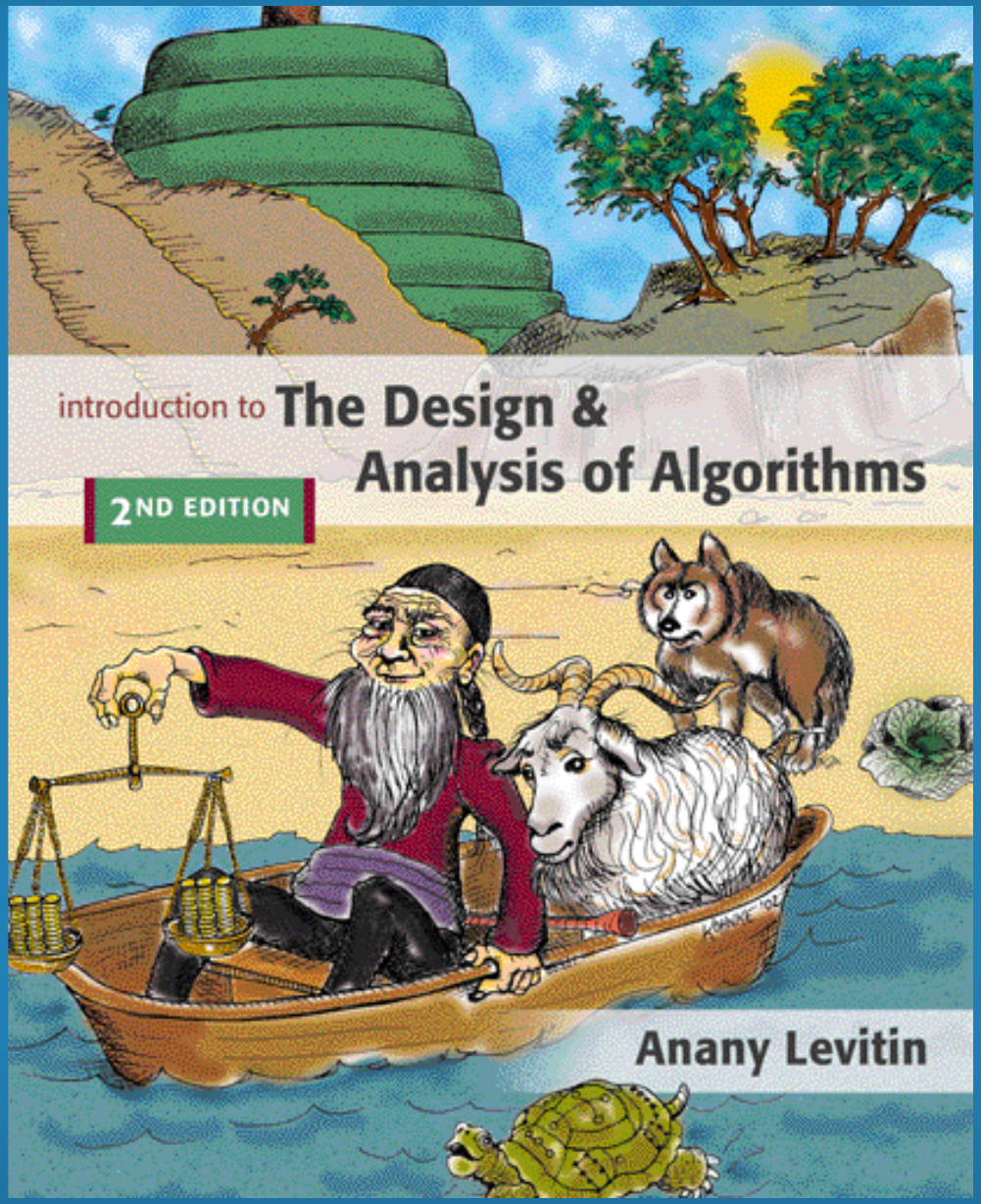
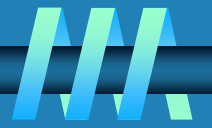
Chapter 4

Divide-and-Conquer

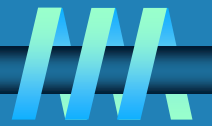


Difference



- The **decrease and conquer** technique **is** similar to **divide and conquer**, except instead **of** partitioning a problem into multiple subproblems **of** smaller size, we use some technique to **reduce** our problem into a single problem that **is** smaller than the original.

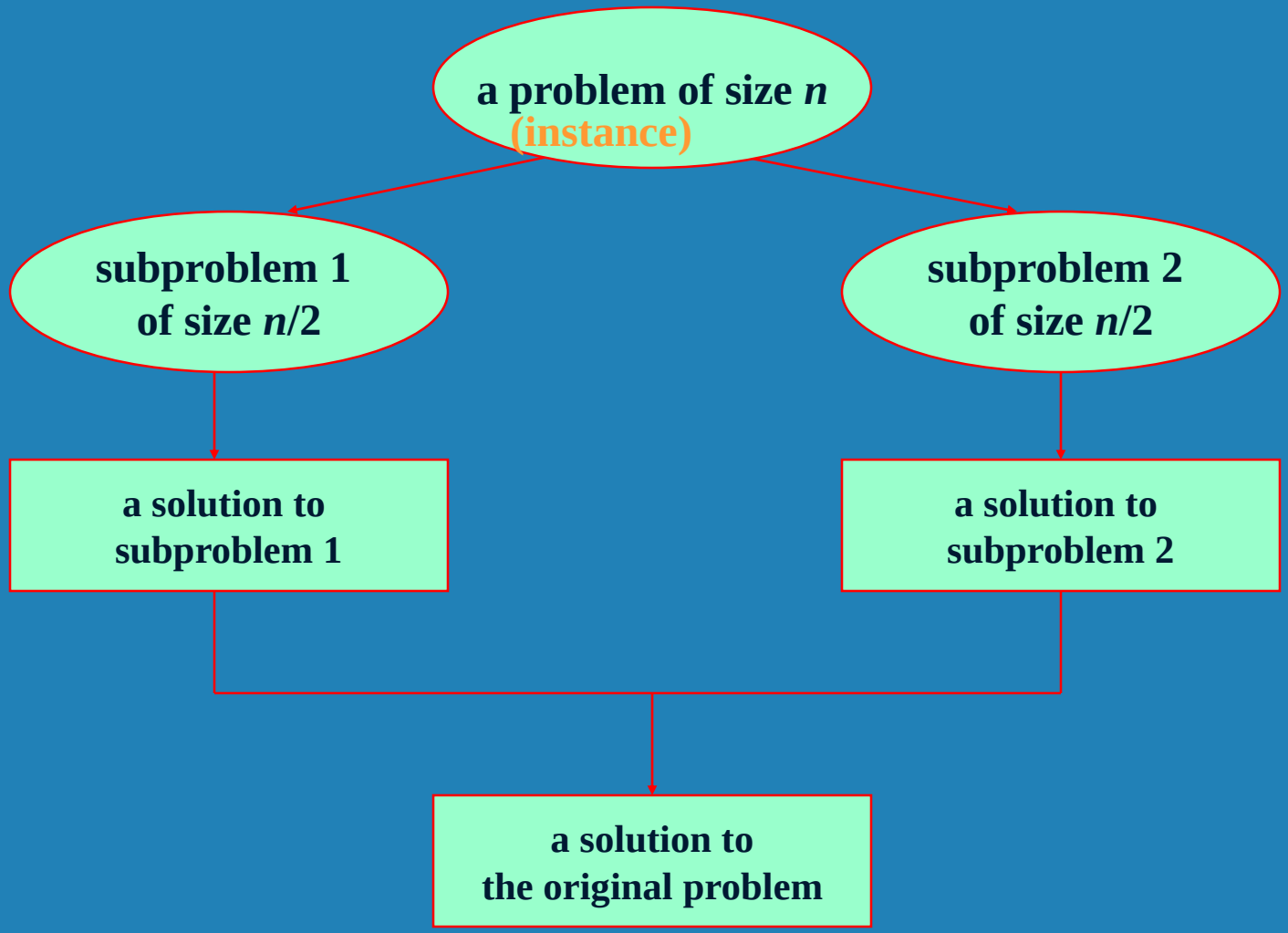
Divide-and-Conquer



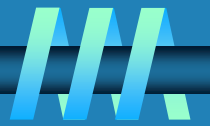
The most-well known algorithm design strategy:

1. **Divide instance of problem into two or more smaller instances**
2. **Solve smaller instances recursively**
3. **Obtain solution to original (larger) instance by combining these solutions**

Divide-and-Conquer Technique (cont.)

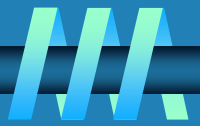


Divide-and-Conquer Examples



- Sorting: Mergesort and Quicksort
- Binary tree traversals
- Binary search (?)
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm

General Divide-and-Conquer Recurrence



$T(n) = a\,T(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

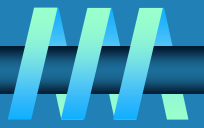
If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in ?$ $\Theta(n^2)$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in ?$ $\Theta(n^2 \log n)$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in ?$ $\Theta(n^3)$

Mergesort



- Split array A[0..*n*-1] into about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Pseudocode of Mergesort

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

Pseudocode of Merge

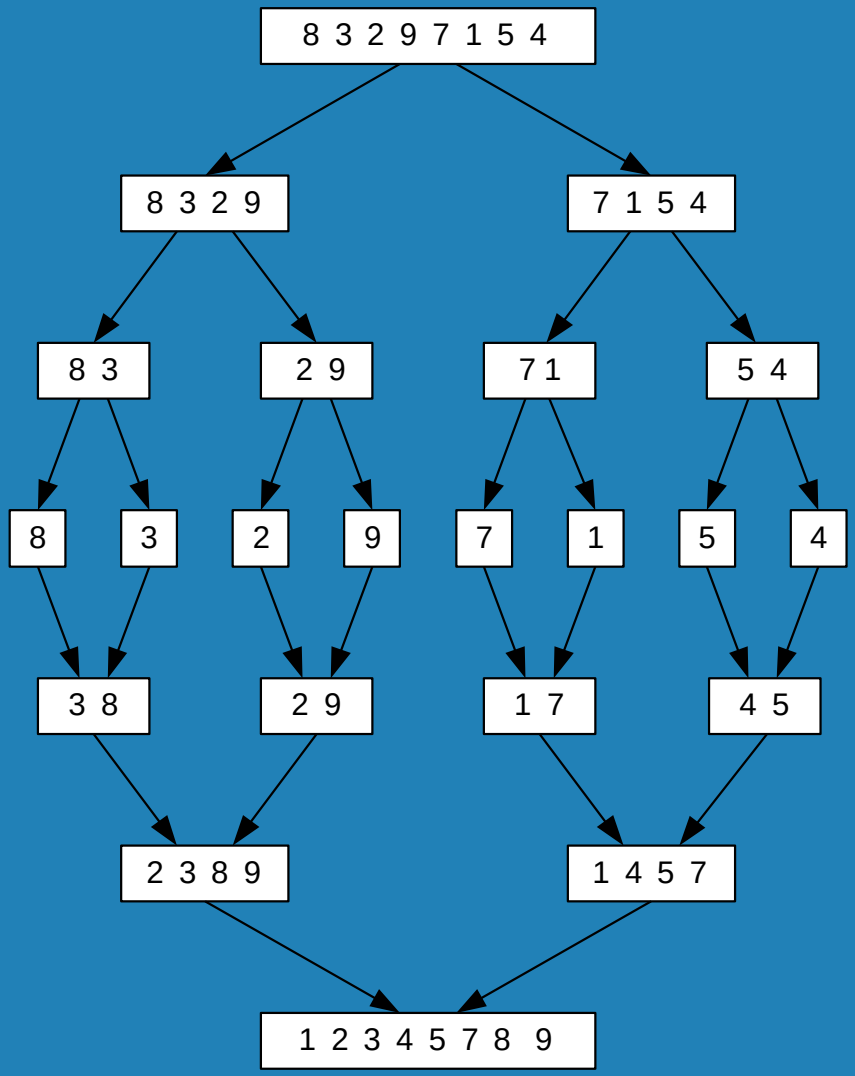
ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C

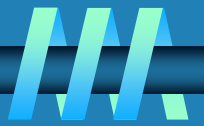
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i + 1$
    else $A[k] \leftarrow C[j];\ j \leftarrow j + 1$
    $k \leftarrow k + 1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

Time complexity: $\Theta(p+q) = \Theta(n)$ comparisons

Mergesort Example



Analysis of Mergesort



- All cases have same efficiency: $\Theta(n \log n)$

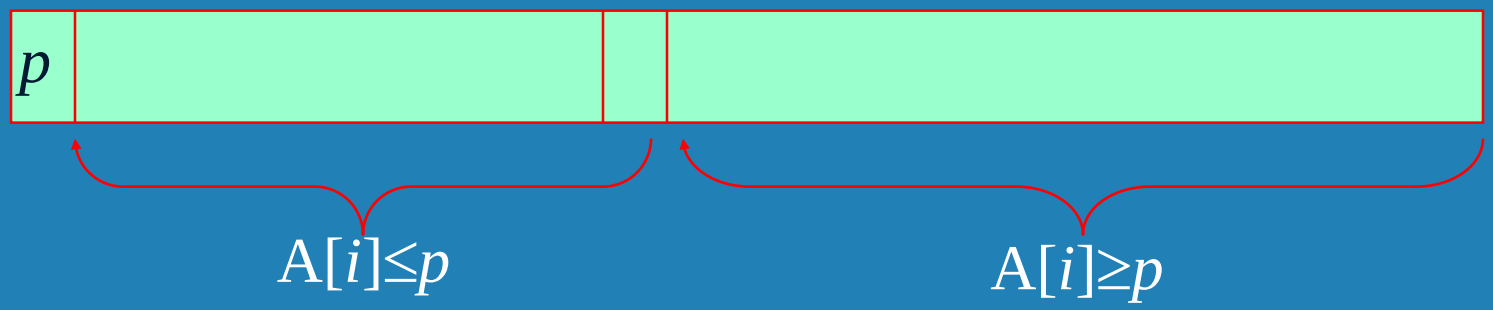
$$T(n) = 2T(n/2) + \Theta(n),\ T(1) = 0$$

- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:

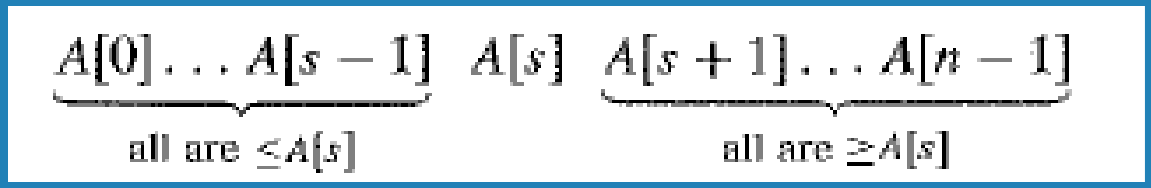
$$\lceil \log_2 n! \rceil \approx n \log_2 n - 1.44n$$

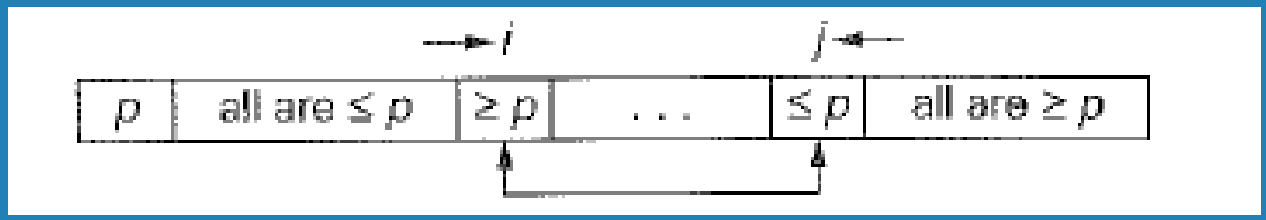
- Space requirement: $\Theta(n)$ (not in-place)
- Stable Algorithm
- Can be implemented without recursion (bottom-up)

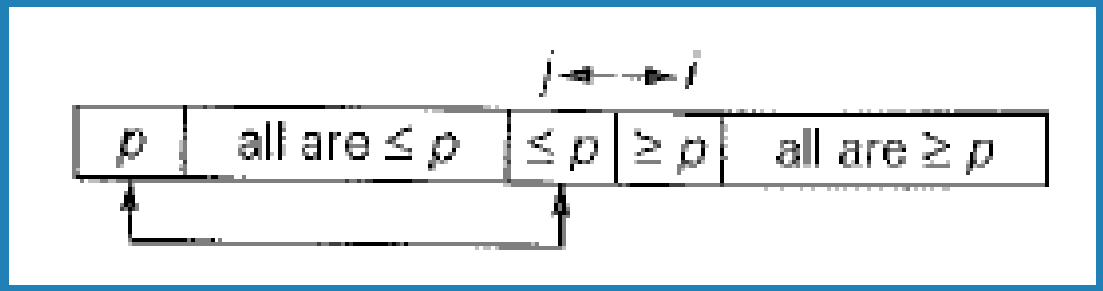
Quicksort

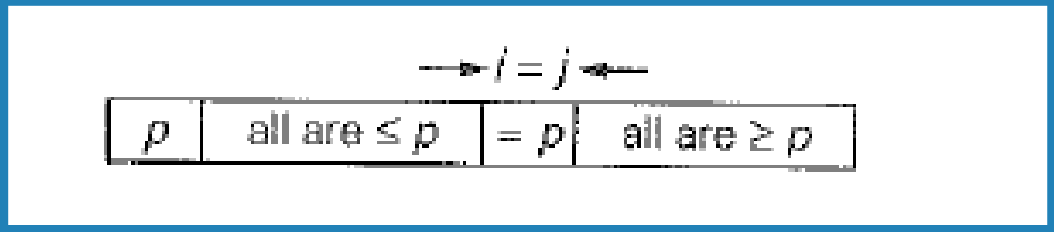
- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot
- 

- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively









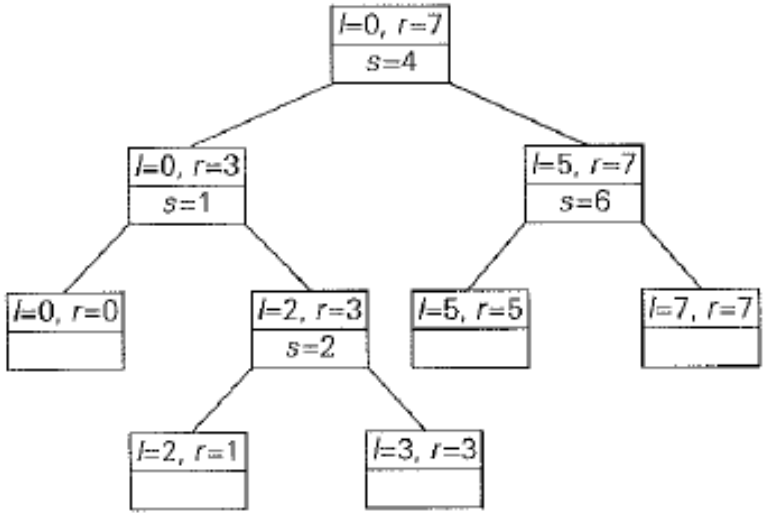
ALGORITHM *Quicksort*$(A[l..r])$

//Sorts a subarray by quicksort
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right indices
// l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
  $s \leftarrow$ *Partition*$(A[l..r])$ //s is a split position
  Quicksort$(A[l..s-1])$
  Quicksort$(A[s+1..r])$

ALGORITHM *Partition*$(A[l..r])$

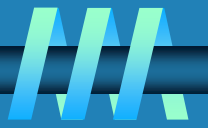
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: A partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
  repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
  swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

0	1	2	3	4	5	6	7
	i						*j*
5	3	1	9	8	2	4	7
			i			*i*	
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	4	8	2	9	7
				i	*j*		
5	3	1	4	8	2	9	7
				j	*j*		
5	3	1	4	2	8	9	7
				j	*i*		
5	3	1	4	2	8	9	7
5	3	1	4	**5**	8	9	7
	i		*j*				
2	3	1	4				
	i	*j*					
2	3	1	4				
	i	*j*					
2	1	3	4				
	j	*i*					
2	1	3	4				
1	**2**	3	4				
1							
			ij				
		3	4				
		j	*i*				
		3	4				
			4				
						i	*j*
					8	9	7
						i	*j*
					8	7	9
						j	*i*
					8	7	9
					7	**8**	9
					7		
							9



(b)

Algorithm Quicksort



Algorithm Quicksort(A[*l*…r])

Input: A subarray A[*l*…r] of A[0…n-1]

Output: Subarray A[*l*…r] sorted in nondecreasing order

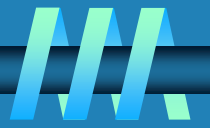
if *l* < r

S← Partition(A[*l*…r])

Quicksort (A[*l*… s-1])

Quicksort (A[s+1 …r])

Partitioning Algorithm



ALGORITHM *Partition(A[l .. r])*

//Input: A subarray *A[l..r]* of *A[O .. n* - 1], defined by its left and right indices *l* and *r (l < r)*

//Output: A partition of *A[l..r*], with the split position returned as

// this function's value

p ←A[*l*]

i ←*l*; j ← r + 1;

repeat

  repeat i ←i + 1 until A[i] ≥ p

  repeat j ← j - 1 until A[j] ≤ p

  swap(A[i], *A[j])*

until *i* ≥*j*

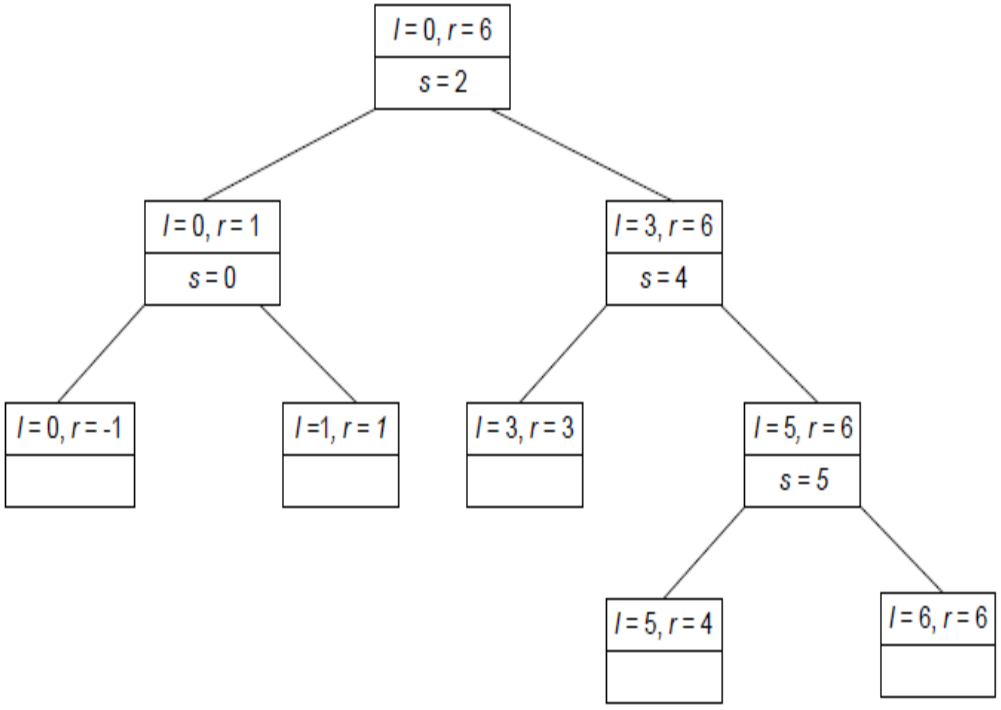
swap(A[i], *A[j])* //undo last swap when *i* >=*j*

swap(A[l], *A[j])*

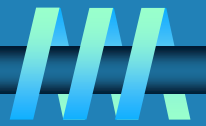
Return *j*

$<$ or $i > r$ or $j = l$

0	1	2	3	4	5	6
	i					j
E	X	A	M	P	L	E
		j	i			
E	E	A	M	P	L	X
A	E	**E**	M	P	L	X
	i j					
A	E					
j	i					
A	E					
A	E					
	E					
				i		j
			M	P	L	X
				i	j	
			M	P	L	X
				i	j	
			M	L	P	X
				j	i	
			M	L	P	X
			L	M	P	X
			L			
						i j
					P	X
					j	i
					P	X
					P	X
						X



Quicksort Example



5 3 1 9 8 2 4 7

2 3 1 4 5 8 9 7

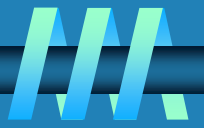
1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

Analysis of Quicksort



- Best case: split in the middle —
- $C_{best}(n) = 2C_{best}(n/2) + n$ for $n>1$
- Worst case: sorted array! —

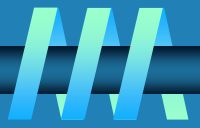
$$C_{worst}(n) = (n+1) + n + (n-1) + (n-2) + \ldots\ldots + 3$$

$$= \frac{(n+1)(n+2)}{2} - 3$$

- Average case: random arrays — $\Theta(n \log n)$
- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion

  These combine to 20-25% improvement

Binary Search



Very efficient algorithm for searching in sorted array:

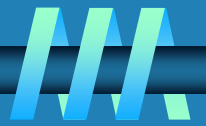
K

vs

A[0] . . . A[m] . . . A[n-1]

If K = A[m], stop (successful search); otherwise, continue searching by the same method in A[0..m-1] if K < A[m] and in A[m+1..n-1] if K > A[m]

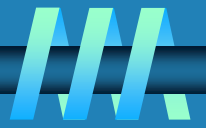
$l \leftarrow 0$; $r \leftarrow n-1$
while $l \leq r$ **do**
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if $K = A[m]$ **return** m
    else if $K < A[m]$ $r \leftarrow m-1$
    else $l \leftarrow m+1$
return -1



ALGORITHM *BinarySearch*$(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//       a search key K
//Output: An index of the array's element that is equal to K
//        or −1 if there is no such element
l ← 0; r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

Analysis of Binary Search



- Time efficiency
  - worst-case recurrence: $C_w(n) = 1 + C_w(\lfloor n/2 \rfloor)$, $C_w(1) = 1$
    solution: $C_w(n) = \lceil \log_2(n+1) \rceil$

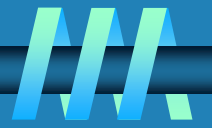
    This is VERY fast: e.g., $C_w(10^6) = 20$

- Optimal for searching a sorted array

- Limitations: must be a sorted array

- Bad (degenerate) example of divide-and-conquer because only one of the sub-instances is solved

Binary Tree Algorithms



Binary tree is a divide-and-conquer ready structure!

Ex. 1: Classic traversals (preorder, inorder, postorder)

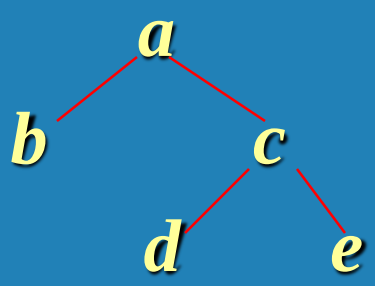
Algorithm ***Inorder*****(*****T*****)**

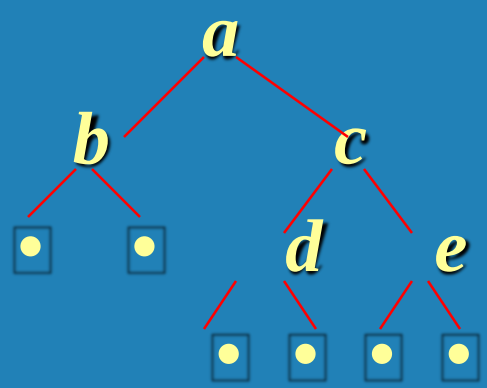
if $T \neq \varnothing$

  $Inorder(T_{left})$

  print(root of ***T*****)**

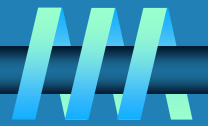
  $Inorder(T_{right})$



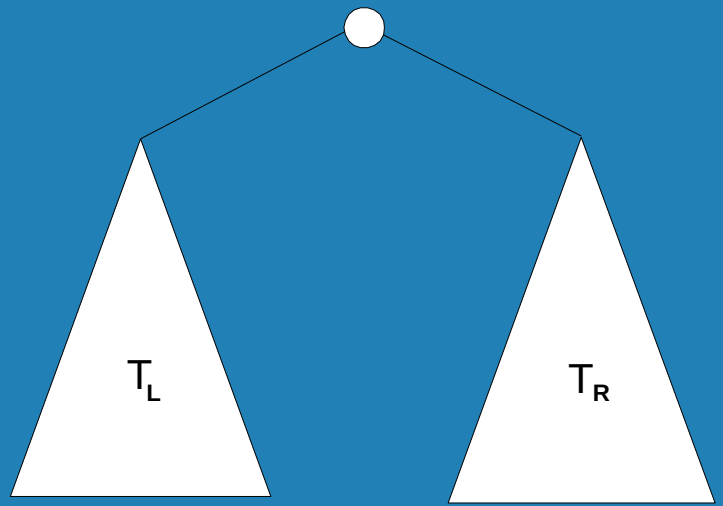


Efficiency: Θ(*n*). Why? Each node is visited/printed once.

Binary Tree Algorithms (cont.)

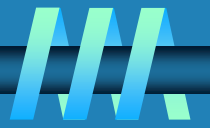


Ex. 2: Computing the height of a binary tree



$h(T) = \max\{h(T_L), h(T_R)\} + 1$ if $T \neq \varnothing$ and $h(\varnothing) = -1$

Efficiency: $\Theta(n)$. Why?



ALGORITHM *Height(T)*

//Computes recursively the height of a binary tree

//Input: A binary tree *T*
//Output: The height *of T*

If T = Ø return -1
else return *max(Height(TL), Height(TR)}* + 1

Multiplication of Large Integers

Consider the problem of multiplying two (large) *n*-digit integers represented by arrays of their digits such as:

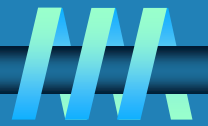
A = 12345678901357986429 B = 87654321284820912836

The grade-school algorithm:

$$
\begin{array}{r}
a_1\ a_2 \ldots\ a_n \\
b_1\ b_2 \ldots\ b_n \\
\hline
(d_{10})\ d_{11} d_{12} \ldots\ d_{1n} \\
(d_{20})\ d_{21} d_{22} \ldots\ d_{2n} \\
\ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots \\
(d_{n0})\ d_{n1} d_{n2} \ldots\ d_{nn} \\
\hline
\end{array}
$$

Efficiency: $\Theta(n^2)$ single-digit multiplications

Multiplication of Large Integers



Traditional Way to Multiply 25 × 63

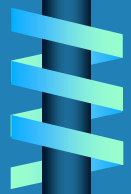
Requires four single-digit multiplications and some additions.

STEP A		STEP B		STEP C		STEP D		STEP E
25		25		25		25		
× 63		× 63		× 63		× 63		
1200	+	15	+	60	+	300	=	1575

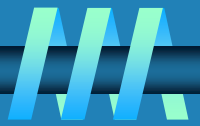
Karatsuba Method for 25 × 63

Requires three single-digit multiplications plus some additions and subtractions.

STEP A	B	C	D	E	F	G
Break numbers up.	Multiply the tens.	Multiply the ones.	Add the digits.	Multiply the sums.	Subtract B and C from E.	Assemble the numbers.
25 → 2 5	2	5	2 + 5 = 7	7	63	12
63 → 6 3	× 6	× 3	6 + 3 = 9	× 9	− 15	36
	12	15		63	− 12	+ 15
					36	1575



Multiplication of Large Integers



MULTIPLIED SAVINGS: As numbers increase in size, the Karatsuba method can be used repeatedly, breaking large numbers into small pieces to save an increasing number of single-digit multiplications.

Traditional way to multiply 2,531 × 1,467 requires 16 single-digit multiplications.

2531 × 1467 + 2531 × 1467 + 2537 × 1467 + 2531 × 1467 = 3712977

Karatsuba method to multiply 2,531 × 1,467 requires 9 single-digit multiplications.

STEP	A	B	C	D	E	F	G
	25 31 14 67	25 × 14 = 350	31 × 67 = 2077	25 + 31 = 56 14 + 67 = 81	56 × 81 = 4536	4536 − 2077 − 350 = 2109	350 + 2109 + 2077 = 3712977
Run Karatsuba method on:		25 14	31 67		56 81		

Lucy Reading-Ikkanda/Quanta Magazine

$$23 * 14 = (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0)$$
$$= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0.$$

The last formula yields the correct answer of 322, of course, but it uses the same four digit multiplications as the pen-and-pencil algorithm. Fortunately, we can compute the middle term with just one digit multiplication by taking advantage of the products $2 * 1$ and $3 * 4$ that need to be computed anyway:

$$2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - 2 * 1 - 3 * 4.$$

For any pair of two-digit integers $a = a_1a_0$ and $b = b_1b_0$, their product c can be computed by the formula

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0,$$

where

$c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 and c_0.

Now we apply this trick to multiplying two n-digit integers a and b where n i a positive even number. Let us divide both numbers in the middle—after all, w promised to take advantage of the divide-and-conquer technique. We denote th first half of the a's digits by a_1 and the second half by a_0; for b, the notations are b and b_0, respectively. In these notations, $a = a_1a_0$ implies that $a = a_1 10^{n/2} + a_0$, an $b = b_1b_0$ implies that $b = b_1 10^{n/2} + b_0$. Therefore, taking advantage of the sam trick we used for two-digit numbers, we get

$$\begin{aligned} c = a * b &= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\ &= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\ &= c_2 10^n + c_1 10^{n/2} + c_0, \end{aligned}$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,

$c_0 = a_0 * b_0$ is the product of their second halves,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halve and the sum of the b's halves minus the sum of c_2 and c_0.

For $2101 * 1130$:

$$\begin{aligned} c_2 &= 21 * 11 \\ c_0 &= 01 * 30 \\ c_1 &= (21+01) * (11+30) - (c_2 + c_0) = 22 * 41 - 21 * 11 - 01 * 30. \end{aligned}$$

For $21 * 11$:

$$\begin{aligned} c_2 &= 2 * 1 = 2 \\ c_0 &= 1 * 1 = 1 \\ c_1 &= (2+1) * (1+1) - (2+1) = 3 * 2 - 3 = 3. \\ \text{So, } 21 * 11 &= 2 \cdot 10^2 + 3 \cdot 10^1 + 1 = 231. \end{aligned}$$

For $01 * 30$:

$$\begin{aligned} c_2 &= 0 * 3 = 0 \\ c_0 &= 1 * 0 = 0 \\ c_1 &= (0+1) * (3+0) - (0+0) = 1 * 3 - 0 = 3. \\ \text{So, } 01 * 30 &= 0 \cdot 10^2 + 3 \cdot 10^1 + 0 = 30. \end{aligned}$$

For $22 * 41$:

$$\begin{aligned} c_2 &= 2 * 4 = 8 \\ c_0 &= 2 * 1 = 2 \\ c_1 &= (2+2) * (4+1) - (8+2) = 4 * 5 - 10 = 10. \\ \text{So, } 22 * 41 &= 8 \cdot 10^2 + 10 \cdot 10^1 + 2 = 902. \end{aligned}$$

Hence

$$2101 * 1130 = 231 \cdot 10^4 + (902 - 231 - 30) \cdot 10^2 + 30 = 2,374,130.$$

How many digit multiplications does this algorithm make? Since multiplication of n-digit numbers requires three multiplications of $n/2$-digit numbers, the recurrence for the number of multiplications $M(n)$ will be

$$M(n) = 3M(n/2) \quad \text{for } n > 1, \qquad M(1) = 1.$$

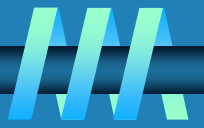
Solving it by backward substitutions for $n = 2^k$ yields

$$\begin{aligned} M(2^k) &= 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2 M(2^{k-2}) \\ &= \cdots = 3^i M(2^{k-i}) = \cdots = 3^k M(2^{k-k}) = 3^k. \end{aligned}$$

Since $k = \log_2 n$,

$$M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}.$$

First Divide-and-Conquer Algorithm



A small example: A * B where

$23 = 2.\ 10^1 + 3.\ 10^0$ and $14 = 1.10^1 + 4.10^0$.

$23 * 14 = (2.\ 10^1 + 3.\ 10^0)* (\ 1.10^1 + 4.10^0\ )$

$= (2*1)10^2 + (2*4 + 3*1)\ 10^1 + (3*4\ )10^0$

In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are *n*-digit, A_1, A_2, B_1, B_2 are *n*/2-digit numbers),

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

Recurrence for the number of one-digit multiplications M(*n*):

$$M(n) = 4M(n/2), \quad M(1) = 1$$

Solution: $M(n) = n^2$

Second Divide-and-Conquer Algorithm

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

The idea is to decrease the number of multiplications from 4 to 3:

$$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$$

I.e., $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2,$ **which requires only 3 multiplications at the expense of (4-1) extra add/sub.**

Recurrence for the number of multiplications M(*n*):

$$M(n) = 3M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

What if we count both multiplications and additions?

Example of Large-Integer Multiplication

2135 * 4014

= (21*10^2 + 35) * (40*10^2 + 14)

= (21*40)*10^4 + c1*10^2 + 35*14

where c1 = (21+35)*(40+14) - 21*40 - 35*14, and

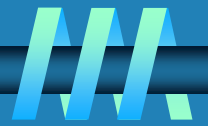
21*40 = (2*10 + 1) * (4*10 + 0)

= (2*4)*10^2 + c2*10 + 1*0

where c2 = (2+1)*(4+0) - 2*4 - 1*0, etc.

This process requires 9 digit multiplications as opposed to 16.

Conventional Matrix Multiplication



- **Brute-force algorithm**

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

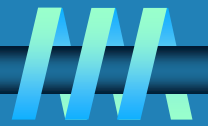
$$= \begin{pmatrix} a_{00} * b_{00} + a_{01} * b_{10} & a_{00} * b_{01} + a_{01} * b_{11} \\ a_{10} * b_{00} + a_{11} * b_{10} & a_{10} * b_{01} + a_{11} * b_{11} \end{pmatrix}$$

8 multiplications

4 additions

Efficiency class in general: $\Theta(n^3)$

Strassen's Matrix Multiplication



- **Strassen's algorithm for two 2x2 matrices (1969):**

$$\begin{pmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} * \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix}$$

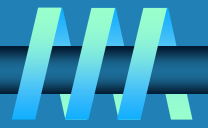
$$= \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{pmatrix}$$

- $m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$
- $m_2 = (a_{10} + a_{11}) * b_{00}$
- $m_3 = a_{00} * (b_{01} - b_{11})$
- $m_4 = a_{11} * (b_{10} - b_{00})$
- $m_5 = (a_{00} + a_{01}) * b_{11}$
- $m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$
- $m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

7 multiplications

18 additions

Strassen's Matrix Multiplication

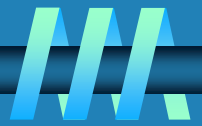


Strassen observed [1969] that the product of two matrices can be computed in general as follows:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$= \begin{pmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{pmatrix}$$

Formulas for Strassen's Algorithm



$$M_1 = (A_{00} + A_{11}) * (B_{00} + B_{11})$$

$$M_2 = (A_{10} + A_{11}) * B_{00}$$

$$M_3 = A_{00} * (B_{01} - B_{11})$$

$$M_4 = A_{11} * (B_{10} - B_{00})$$

$$M_5 = (A_{00} + A_{01}) * B_{11}$$

$$M_6 = (A_{10} - A_{00}) * (B_{00} + B_{01})$$

$$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$$

For the matrices given, Strassen's algorithm yields the following:

$$C = \left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

where

$$A_{00} = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix}, \quad A_{01} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}, \quad A_{10} = \begin{bmatrix} 0 & 1 \\ 5 & 0 \end{bmatrix}, \quad A_{11} = \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix},$$

$$B_{00} = \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix}, \quad B_{01} = \begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix}, \quad B_{10} = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}, \quad B_{11} = \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}.$$

Therefore,

$$M_1 = (A_{00} + A_{11})(B_{00} + B_{11}) = \begin{bmatrix} 4 & 0 \\ 6 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 7 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix},$$

$$M_2 = (A_{10} + A_{11})B_{00} = \begin{bmatrix} 3 & 1 \\ 7 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix},$$

$$M_3 = A_{00}(B_{01} - B_{11}) = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 \\ -5 & 4 \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix},$$

$$M_4 = A_{11}(B_{10} - B_{00}) = \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} = \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix},$$

$$M_5 = (A_{00} + A_{01})B_{11} = \begin{bmatrix} 3 & 1 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix},$$

$$M_6 = (A_{10} - A_{00})(B_{00} + B_{01}) = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 & 2 \\ 2 & 5 \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ -2 & -3 \end{bmatrix},$$

$$M_7 = (A_{01} - A_{11})(B_{10} + B_{11}) = \begin{bmatrix} -1 & 1 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 6 & 3 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix}.$$

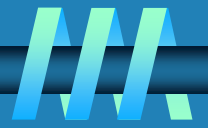
Accordingly,

$$
\begin{aligned}
C_{00} &= M_1 + M_4 - M_5 + M_7 \\
&= \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix} + \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix} - \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix} + \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix} = \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix}, \\
C_{01} &= M_3 + M_5 \\
&= \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix} + \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix} = \begin{bmatrix} 7 & 3 \\ 1 & 9 \end{bmatrix}, \\
C_{10} &= M_2 + M_4 \\
&= \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix} + \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 1 \\ 5 & 8 \end{bmatrix}, \\
C_{11} &= M_1 + M_3 - M_2 + M_6 \\
&= \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix} + \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix} - \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ -2 & -3 \end{bmatrix} = \begin{bmatrix} 3 & 7 \\ 7 & 7 \end{bmatrix}.
\end{aligned}
$$

That is,

$$
C = \begin{bmatrix} 5 & 4 & 7 & 3 \\ 4 & 5 & 1 & 9 \\ 8 & 1 & 3 & 7 \\ 5 & 8 & 7 & 7 \end{bmatrix}.
$$

Analysis of Strassen's Algorithm



If *n* is not a power of 2, matrices can be padded with zeros.

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \qquad M(1) = 1.$$

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2 M(2^{k-2}) = \dots$$
$$= 7^i M(2^{k-i}) \cdots = 7^k M(2^{k-k}) = 7^k.$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$