Analysis of Algorithms

Chapter 5.1, 5.2, 5.4

ROAD MAP

- **Divide And Conquer**
  - Mergesort
  - Quicksort
  - Multiplication of large integers
  - Strassen’s Matrix multiplication

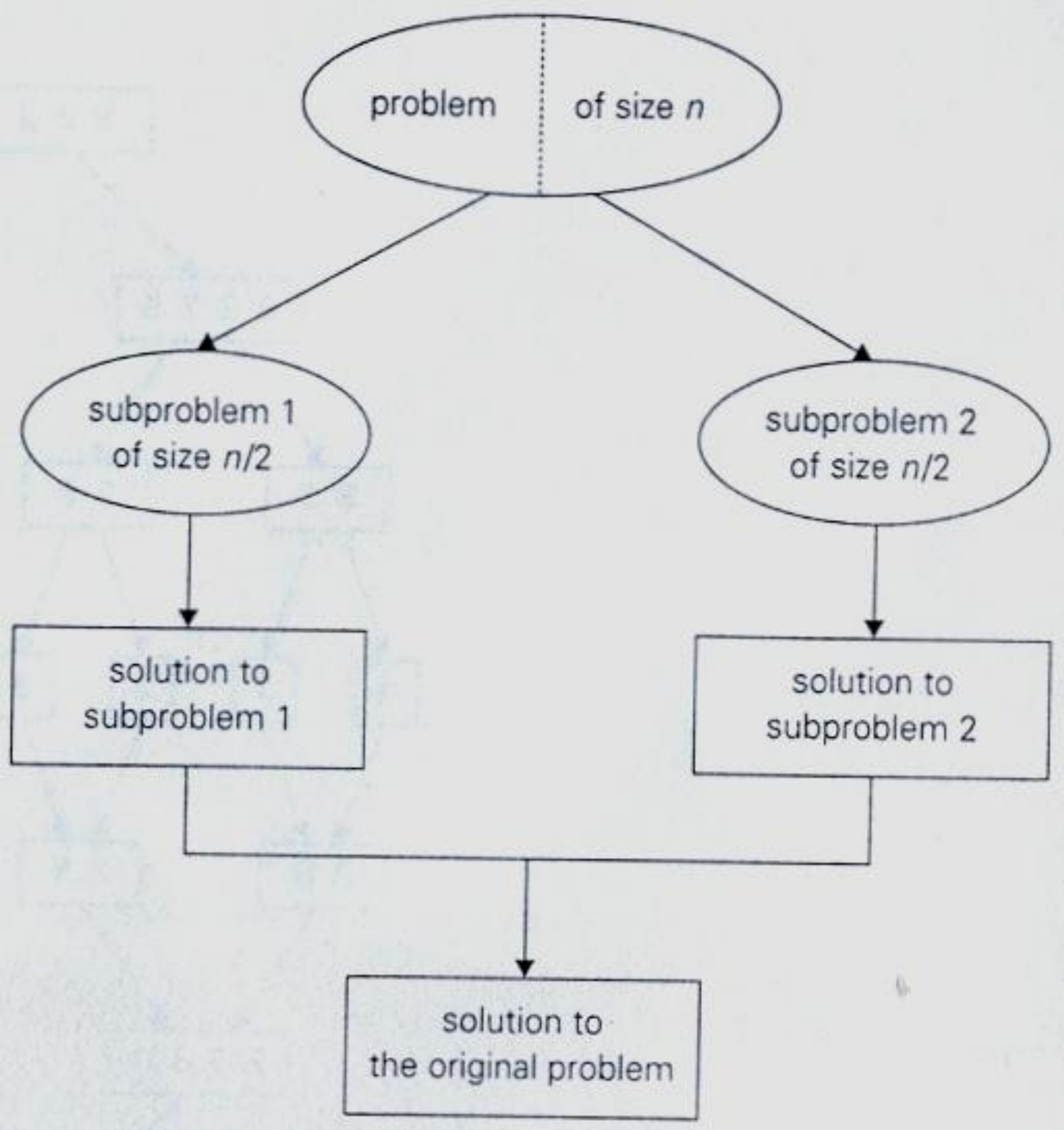
Divide And Conquer

A well known general algorithm design technique

Approach*:*

- A problem's instance is divided into several <u>smaller</u> instances of the <u>same</u> problem
  - ideally of about the same size
- The smaller instances are solved
  - typically recursively
- The solutions obtained for the smaller instances are combined to get a solution to the original problem

Divide And Conquer



Divide And Conquer

- **Algorithm :**

```
D&C (P)
if small (P) then return S(P)
else
{
    divide P into $P_1$, $P_2$, …, $P_k$    $k \geq 1$
    apply D&C to $P_i$
    return combine ( D&C ($P_1$), …, D&C ($P_k$) )
}
```

Divide And Conquer

- **Analysis :**

$$T(P) = T(P_1) + T(P_2) + \ldots + T(P_a) + \underbrace{f(n)}_{\text{to divide\& combine}}$$

$$T(n) = T(n_1) + T(n_2) + \ldots + T(n_a) + f(n)$$

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

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ 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 $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

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

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

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

A simple Example

Problem:

- Compute the sum of n numbers

Approach:

- Divide the problem into two subproblems

- What about the analysis?
  - Is it more efficient than brute force approach?

ROAD MAP

- **Divide And Conquer**
  - Mergesort
  - Quicksort
  - Multiplication of large integers
  - Strassen’s Matrix multiplication

Mergesort

- Mergesort is a perfect example of a successfull application of divide & conquer technique
- Solves the sorting problem

- Given array **`A[0..n-1]`**

Approach :

```
1. divide array into two halves
   A[0..n/2-1] and A[n/2..n-1]
2. sort each halve recursively
3. merge two smaller sorted arrays into a
single sorted one
```

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..(n/2)-1] to B[0..(n/2)-1]
    copy A[n/2..n-1]   to C[0..(n/2)-1]
    Mergesort (B[0..(n/2)-1])
    Mergesort (C[0..(n/2)-1])
    Merge (B, C, A)
```

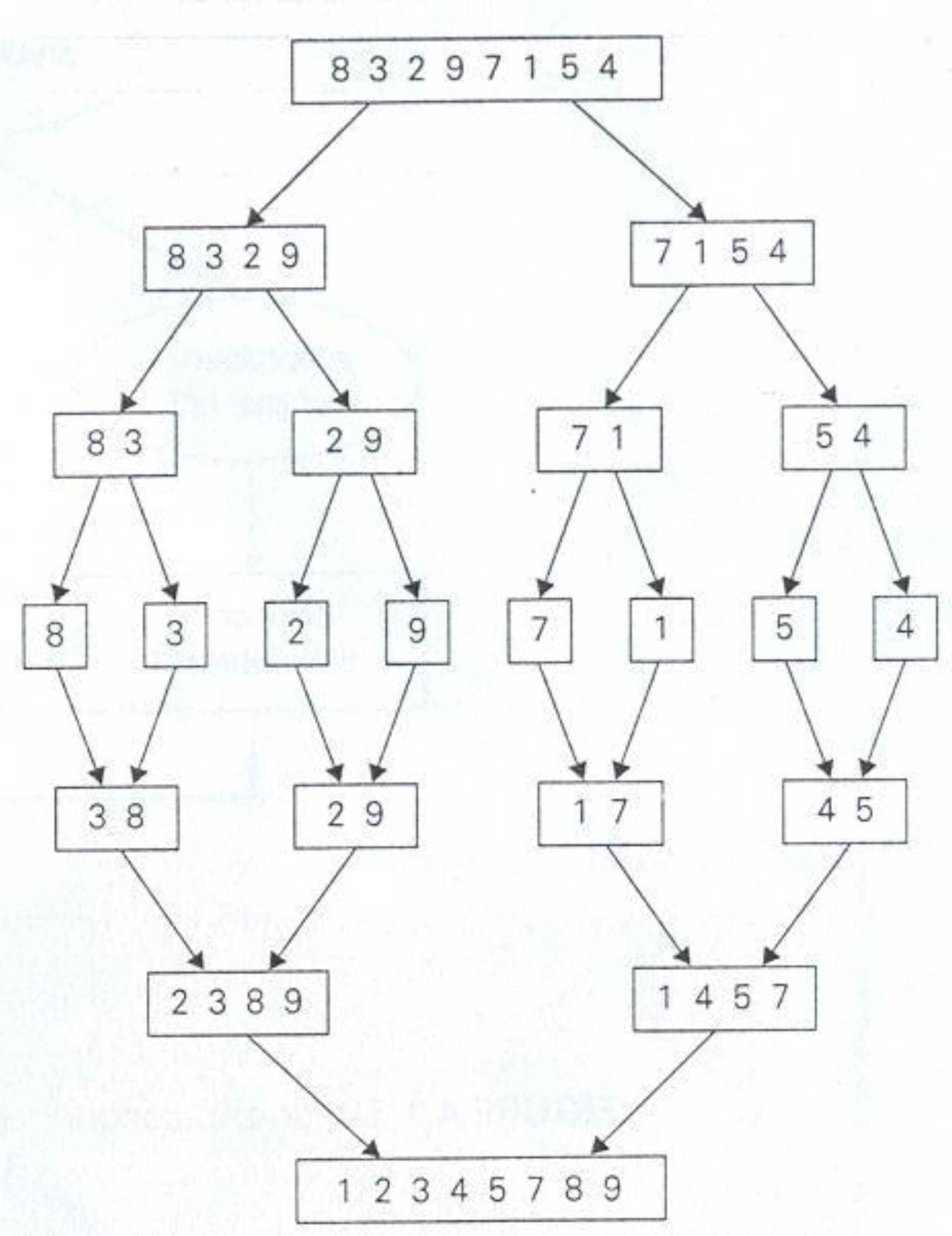
Mergesort

- **ALGORTHM** `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←0 ; j←0, k←0
while i<p and j<q do
    if B[i]≤C[j]
          A[k]←B[i]; i←i+1
    else
          A[k]←C[j]; j←j+1
    k←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]
```

Mergesort Example



Mergesort

Analysis :

Count the number of comparisons

$$C(n) = 2C(n/2) + C_{merge}(n) \qquad \text{for} \qquad n > 1,$$

$$C(1) = 0$$

What about the merge operation?

- Worst case: when the smaller comes from alternating array

$$C_{merge}(n) = n - 1$$

Mergesort

Analysis :

$$C_w(n) = 2C_w(n/2) + n - 1 \qquad \text{for} \qquad n > 1,$$

$$C_w(1) = 0$$

By backward substitution

$$C_w(n) = n\log_2 n - n + 1 = O(n\log n)$$

Or we can use Master Theorem if asymptotic solution is sufficient

Mergesort

- **Discussion :**
  - Perfect example of a successfull application of divide & conquer technique
  - Optimal with respect to number of comparisons
  - Disadvantages
    - Extra space used in Merge
      - How big it is?
      - How to reduce?
    - Recursive calls – stack space
      - use insertion sort for small # of elements
      - iterative

ROAD MAP

- **Divide And Conquer**
  - Mergesort
  - Quicksort
  - Multiplication of large integers
  - Strassen’s Matrix multiplication

Quicksort

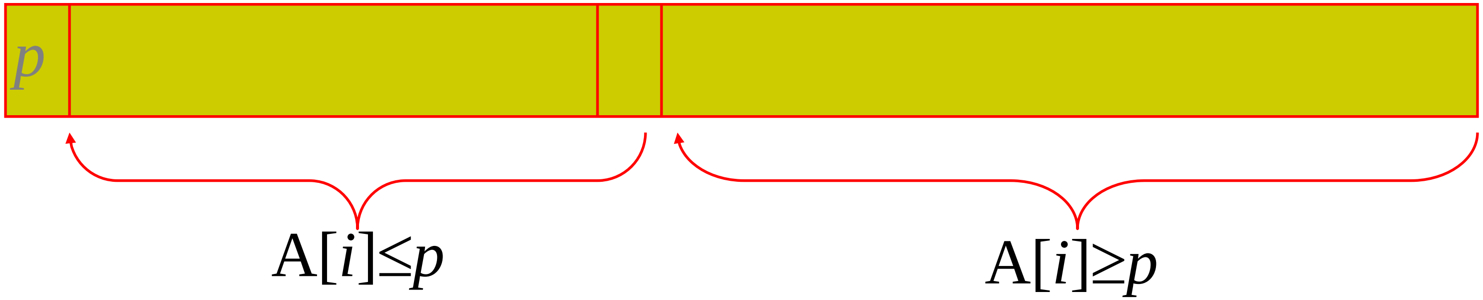
- Quicksort is an important sorting algorithm based on D & C strategy
- It sorts a given array `A[0..n-1]`

Quicksort

Given an array **`A[0..n-1]`**

Approach :

- 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

Quicksort

```
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 indces l and r
// Output : The subarray A[l..r] sorted in
nondecreasing order

If l<r
    s←Partition(A[l..r])
          //s is a split position
    Quicksort(A[l..s-1])
    Quicksort(A[s+1..r])
```

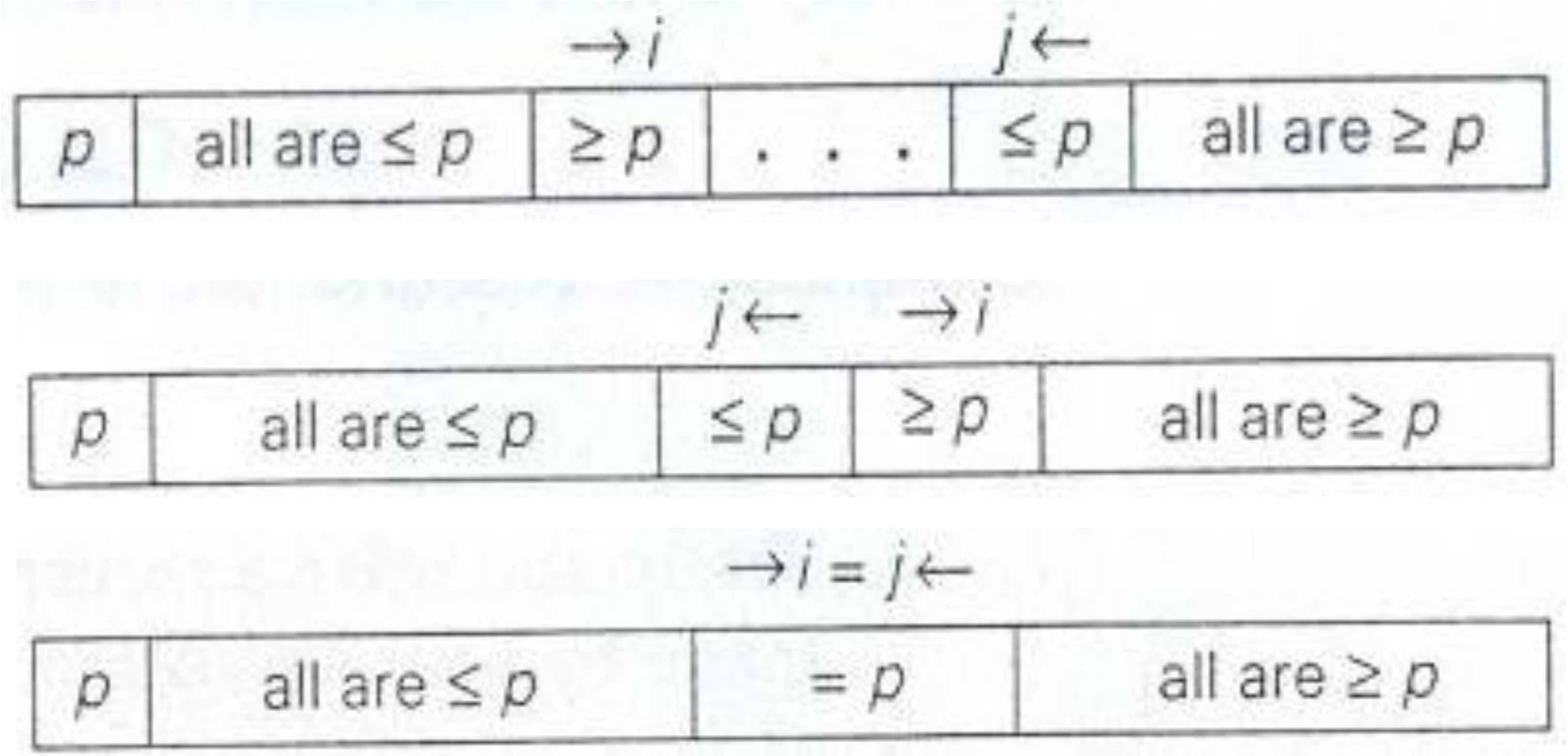
Quicksort

- How to achieve a partition of A[0..n-1] ?
  - Select an element with respect to whose value we are going to divide subarray
    - this element is called ***pivot***

- There are several strategies to select a pivot.
- For now we use the simplest strategy
  - Pivot is subarray’s first element; *p=A[0]*

Quicksort

- Partitioning :



$\rightarrow i$ $\quad$ $j \leftarrow$

p	all are $\le p$	$\ge p$	. . .	$\le p$	all are $\ge p$

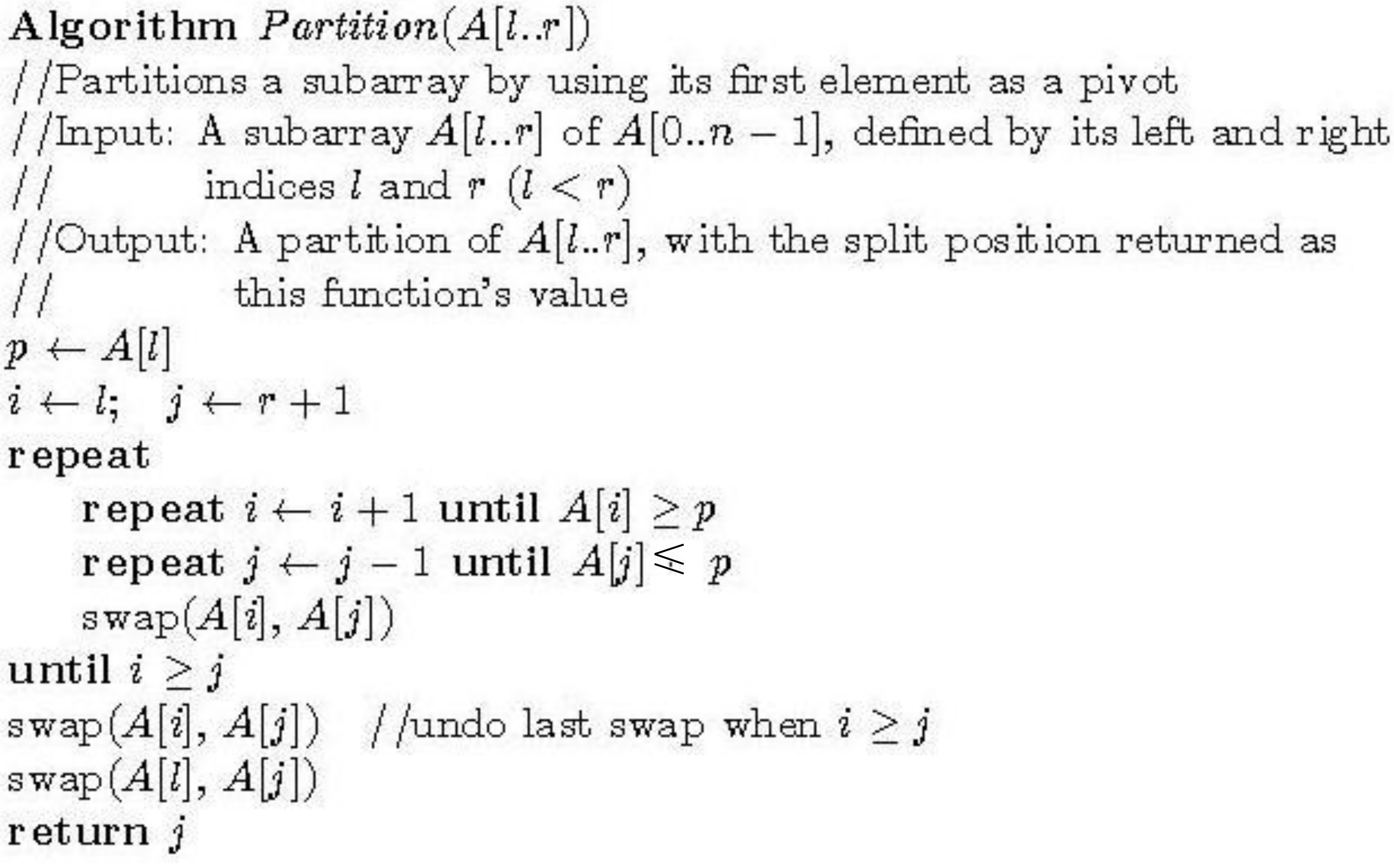
$j \leftarrow$ $\quad$ $\rightarrow i$

p	all are $\le p$	$\le p$	$\ge p$	all are $\ge p$

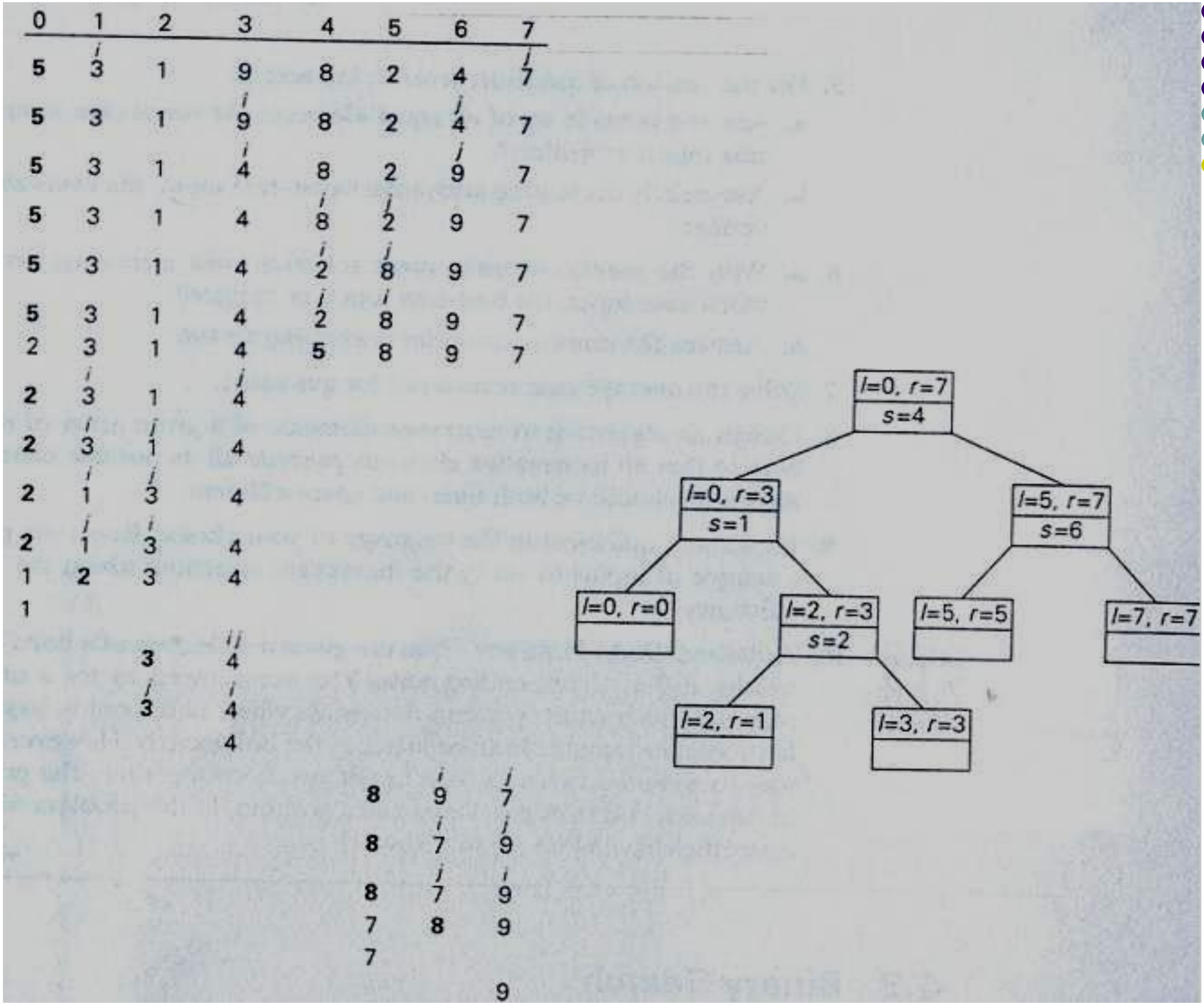
$\rightarrow i = j \leftarrow$

p	all are $\le p$	$= p$	all are $\ge p$

Partitioning Algorithm



```
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 $A[j] \leqslant p$
    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           j
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
                i   j
5   3   1   4   2   8   9   7
                j   i
5   3   1   4   2   8   9   7
2   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
```

Quick Sort

- **Analysis :**

  `n:# of elements`

  `T(partition) = O(n) → n+1`

  - **Best case**

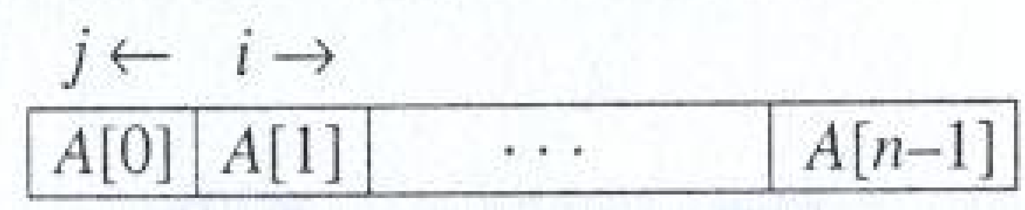
    If all the splits happen in the middle of the corresponding subarrays, it is the best case

$$T(n) = 2T(n/2) + n \qquad for \qquad n > 1$$

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

Quick Sort

- **Analysis :**
  - Worst-case
    - All splits will be skewed to the extreme
      - One of the two subarrays will be empty while the size of the other will be just one less than the size of a subarray being partitioned
      - If A[0 .. n-1] is a strictly increasing array and we use A[0] as the pivot
        - Left to right scan will stop on A[1]
        - Right to left scan will go all the way to reach A[0]



Quick Sort

- **Analysis :**
  - Worst-case
    - After comparisons and exchanging the elements the array must be sorted
    - So

$$T(n) = (n+1) + n + \ldots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \theta(n^2)$$

Quick Sort

Analysis :

- Average Case

each element has an equal probability of being the pivot

P = 1/n

$$T(n) = \frac{1}{n}\left(\sum_{k=1}^{n}\big(T(k-1)+T(n-k)\big)+n+1\right)$$

Quick Sort

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

$$nT(n)=\sum_{k=1}^{n}[T(k-1)+T(n-k)+(n+1)]$$

$$\begin{array}{rl} & nT(n)=2\left(T(0)+T(1)+...+T(n-1)\right)+n(n+1) \\ - & (n-1)T(n-1)=2\left(T(0)+...+T(n-2)\right)+n(n-1) \\ \hline & nT(n)-(n-1)T(n-1)=2T(n-1)+2n \end{array}$$

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

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

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

$$\vdots$$

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2\sum_{k=3}^{n+1} \frac{1}{k}$$

$$\frac{T(n)}{n+1} = 2\sum_{k=3}^{n+1} \frac{1}{k}$$

$$\frac{T(n)}{n+1} \le 2\log(n+1)$$

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

Quick Sort

Discussion :

- Quicksort is a very efficient algorithm
- However, its performance depends on the *pivot* point
- The farther we get from the median for the pivot value the more lopsided the partitions become and the greater the depth of the recursion needs to be

ROAD MAP

- **Divide And Conquer**
  - Mergesort
  - Quicksort
  - Multiplication of large integers
  - Strassen’s Matrix multiplication

Multiplication of Large Integers

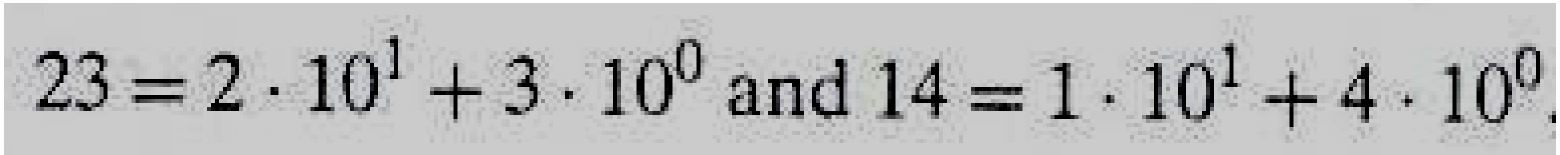
- Some applications require manupilation of large integers (over 100 decimal digits long)
  - Such as cryptology
- Such integers are too long to fit in a special word of a modern computer
  - They require special treatment
  - Does not take unit time

Multiplication of Large Integers

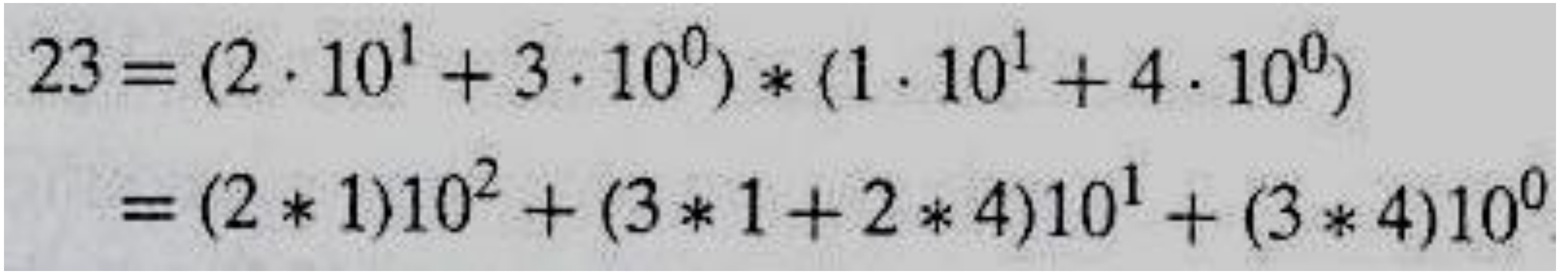
- Classical pen-pencil algorithm for multiplying two *n-digit* integer
  - Each of *n* digits of the first number is mulipied by each of *n* digits of second number
- The total is n^2 digit multiplications
- Is it possible to design an algorithm with fewer than n^2 digit multiplication?

Multiplication of Large Integers

Example: multiply 23 and 14

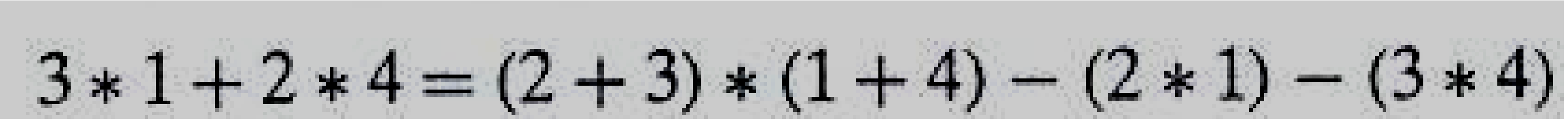


$$23 = 2 \cdot 10^1 + 3 \cdot 10^0 \text{ and } 14 = 1 \cdot 10^1 + 4 \cdot 10^0.$$



$$\begin{aligned} 23 &= (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0) \\ &= (2 * 1)10^2 + (3 * 1 + 2 * 4)10^1 + (3 * 4)10^0 \end{aligned}$$

- There are 4 multiplications in total
- The middle term can also be calculated as



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

- So the result can be obtained by three multiplications only

Multiplication of Large Integers

In general:

For any pair of two-digit integers $a = a_1 a_0$ and $b = b_1 b_0$, their product ***c*** can be computed by the formula

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

where

$c_2 = a_1 * b_1$ → product of their first digits

$c_0 = a_0 * b_0$ → product of their second digits

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ → 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

Multiplication of Large Integers

- **Approach :**

  If we want to multiply two *n-digit* integers a and b where a is positive even number

  - Divide both numbers in the middle
  - Denote first half of the a's digits by a_1 and second half by a_0
    - Same notations for b
  - $a = a_1a_0$ implies that $a=a_1 10^{n/2} + a_0$ and

    $b = b_1b_0$ implies that $b=b_1 10^{n/2} + b_0$

Multiplication of Large Integers

- We get

  $c = a*b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$

  $c = (a_1*b_1)10^n + (a_1*b_0 + a_0*b_1)10^{n/2} + (a_0*b_0)$

  $c = c_2 10^n + c_1 10^{n/2} + c_0$

  where

  $c_2 = a_1*b_1$ → product of their first halves

  $c_0 = a_0*b_0$ → product of their second halves

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

Multiplication of Large Integers

- If *n/2* is even, we can apply same method for computing products of c_2, c_1 and c_0.
- Thus we have a recursive algorithm to compute product of two *n-digit* integers
- Recursion is stopped
  - when n becomes 1
  - when we deem *n* small enough to multiply the numbers of that size directly

Multiplication of Large Integers

- **Analysis :**

  How many digit multiplications does this algorithm make?

Multiplication of Large Integers

- **Analysis :**

  Multiplication of n-digit numbers requires three multiplications of *n/2* digit number

  So

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

  solving it by backward substitution for n = 2^k yields

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

  since k = $\log_2 n$

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

Multiplication of Large Integers

Discussion :

- Used in many problems today
  - Cryptography
  - Security units of mobile devices
- Divide and conquer algorithm outperform the pen-and-pencil method on integers over 600 digits long

ROAD MAP

- **Divide And Conquer**
  - Mergesort
  - Quicksort
  - Multiplication of large integers
  - Strassen’s Matrix multiplication

Matrix Multiplication

- **Problem Definition :**

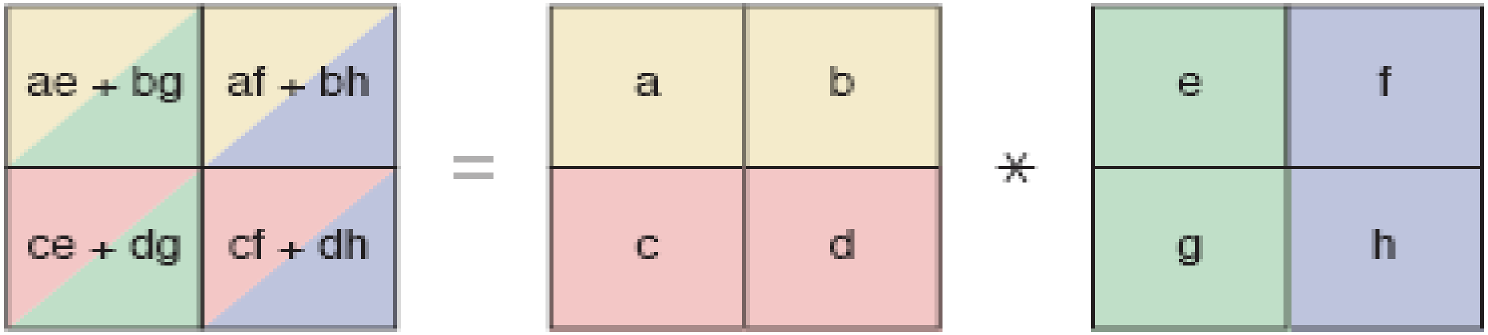
Find product *C* of two *n-by-n* matrices *A* and *B*

- We will see that matrix multiplication can be done using less than n^3 scalar multiplications

Matrix Multiplication

A simple divide and conquer strategy:

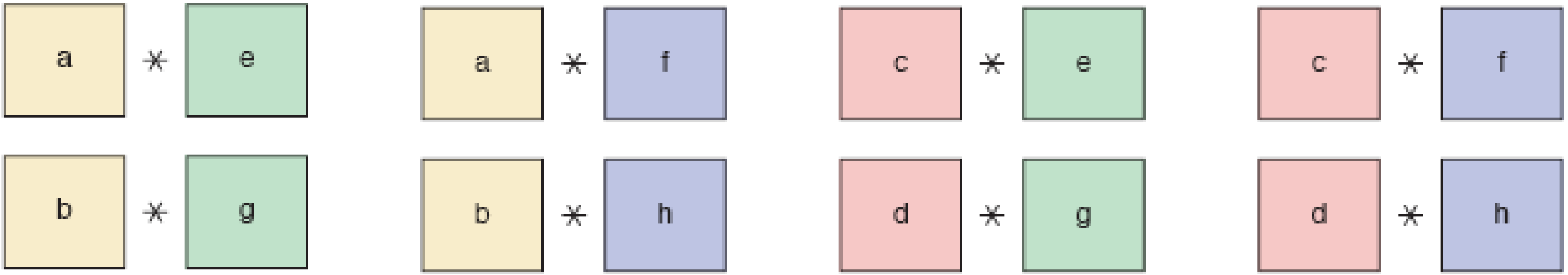
- Let *A* and *B* be two *n-by-n* matrices where *n* is a power of *2*
- We can divide *A, B* and their product *C* into four *n/2-by-n/2* submatrices each as follows



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

Matrix Multiplication

8 Sub-Problems:



a * e

b * g

a * f

b * h

c * e

d * g

c * f

d * h

This image cannot currently be displayed.

Analysis:

- 8 multiplication operation → (n/2)-by-(n/2) matrix
- 4 addition operation → (n/2)-by-(n/2) matrix

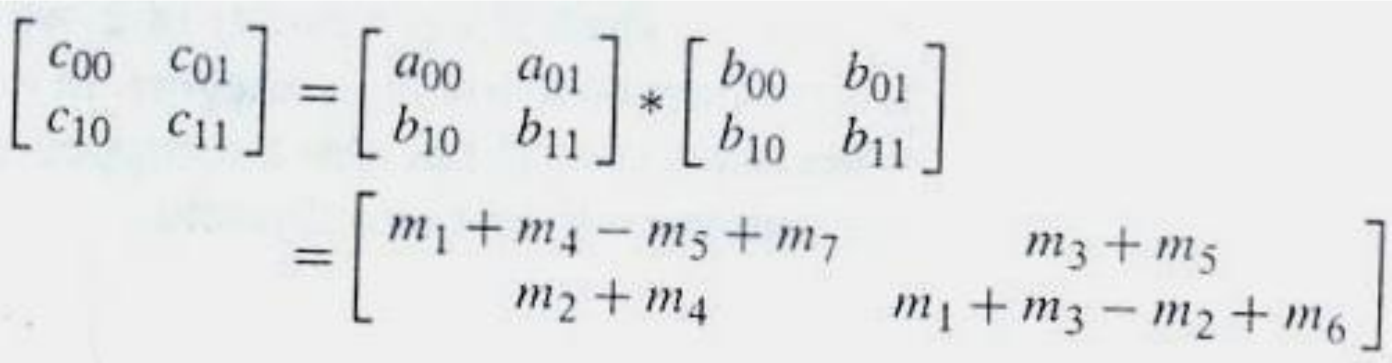
- $T(n)=8*T(n/2)+\Theta(n^2) = \Theta(n^3)$

Strassen’s Matrix Multiplication

- To perform matrix multiplication using less than n^3 scalar multiplications
- First lets consider the case of *2-by-2* matrix multiplication
  - We will show that this can be done using 7 multiplications instead of 8 multiplications required by brute-force algorithm.

Strassen’s Matrix Multiplication

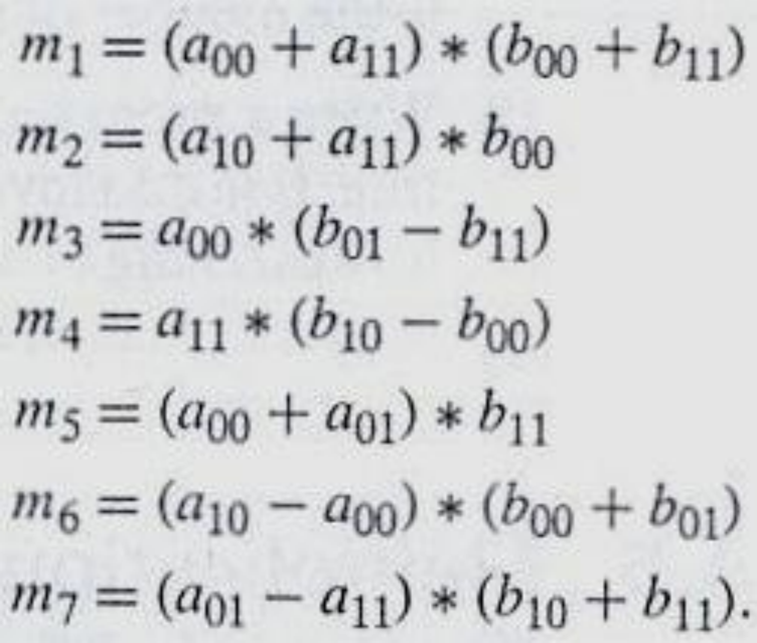
We can use the following formulas



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

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

where



$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$$

$$m_2 = (a_{10} + a_{11}) * b_{00}$$

$$m_3 = a_{00} * (b_{01} - b_{11})$$

$$m_4 = a_{11} * (b_{10} - b_{00})$$

$$m_5 = (a_{00} + a_{01}) * b_{11}$$

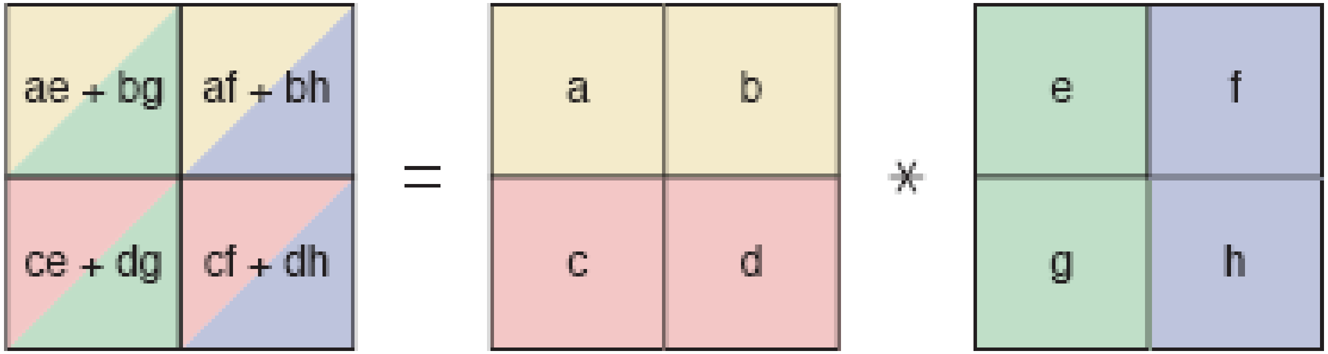
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$$

$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

Strassen’s Matrix Multiplication

- There are 7 multiplications.
- But how many additions are there?
- Is it good idea to use this method for *2-by-2* matrices?

Strassen’s Matrix Multiplication



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

7 multiplication operation

P1=a*(f-h) P2=(a+b)*h P3=(c+d)*e P4=d*(g-e)
P5=(a+d)*(e+h) P6=(b-d)*(g+h) P7=(a-c)*(e+f)

Solution:

a*e+b*g = P5+P4-P2+P6

a*f+b*h = P1+P2

c*e+b*h = P3+P4

c*f + d*h = P5+P1-P3-P7

Strassen’s Matrix Multiplication

Approach:

- Let *A* and *B* be two *n-by-n* matrices
  - where *n* is a power of *2*
- Divide *A* and *B* into four *n/2-by-n/2* submatrices
- Calculate 7 submatrix multiplications recursively
- Perform required additions to obtain the matrix C

Strassen’s Matrix Multiplication

- **Analysis :**

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

Since $n = 2^k$,

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

Since $k = \log_2 n$,

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

Strassen's Matrix Multiplication

Discussion :

- Saving in # of multiplications was achieved at the expense of making extra additions
  - We must check # of additions A(n)
  - $A(n) \in \Theta(n^{\log_2 7})$
  - Same order of growth as # of multiplication
- Efficiency is better than brute force
  - Brute force algorithm is n^3
- Is it good for memory efficiency?
- It is not the best algorithm for matrix multiplication
  - Coopersmith and Winogrand algorithm's efficiency is $O(n^{2.376})$

Divide & Conquer

- **Discussion :**

There are 3 criterias for efficiency of D&C algorithms

- # of subproblems
- Proportion of the main problem and subproblem
- Time to divide the problem and combine the sub-solutions