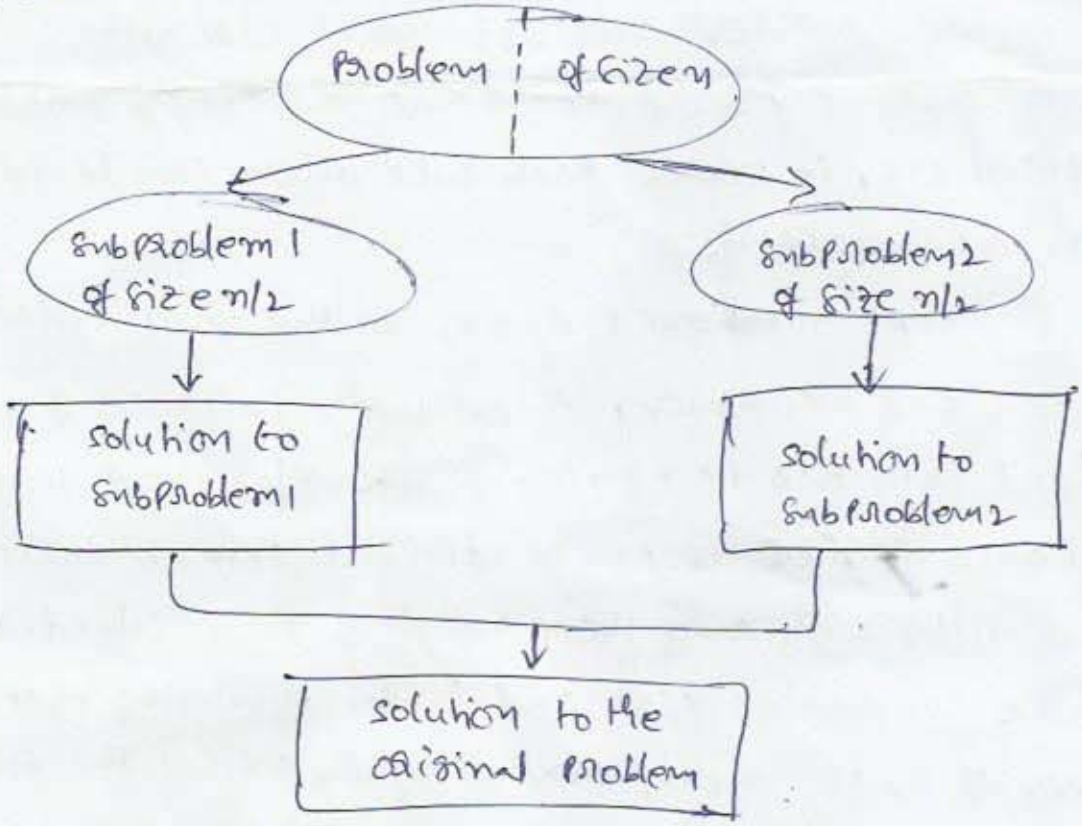
UNIT- II

DIVIDE - AND - CONQUER

Divide - and - conquer algorithms works according to the following general plan:

1. A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.
2. The smaller instances are solved.
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.



Divide - and - conquer technique.

As an example, let us consider the problem of computing the sum of n numbers $a_0, \cdots, a_{n-1}$. If $n>1$, we can divide the problem into two instances of the same problem:

to compute the sum of the first $\lfloor n/2 \rfloor$ numbers and to compute the sum of the remaining $\lceil n/2 \rceil$ numbers. (Of course, if $n = 1$, we simply return a_0 as the answer.) Once each of these two sums is computed, we can add their values to get the sum in question:

$$a_0 + \ldots + a_{n-1} = (a_0 + \ldots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \ldots + a_{n-1})$$

Is this an efficient way to compute the sum of n numbers? A moment of reflection, a small example of summing, say, four numbers by this algorithm, a formal analysis, and common sense all lead to a negative answer to this question.

Thus, not every divide-and-conquer algorithm is necessarily more efficient than even a brute-force solution. Though we consider only sequential algorithms here, the divide-and-conquer technique is ideally suited for parallel computations, in which each subproblem can be solved simultaneously by its own processor.

As mentioned above, in the most typical case of divide-and-conquer, a problem's instance of size n is divided into two instances of size $n/2$. More generally, an instance of size n can be divided into b instances of size n/b, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.) Assuming that size n is a power of b, to simplify our analysis, we get the following recurrence for the running time $T(n)$:

$$T(n) = a\,T(n/b) + f(n), \longrightarrow (1)$$

where $f(n)$ is a function that accounts for the time spent on the dividing the problem into smaller ones and on

combining their solutions. (For the summation example, $a = b = 2$ and $f(n) = 1$.) Recurrence (1) is called the general divide-and-conquer recurrence. Obviously, the order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$. The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem (master theorem)

Theorem (master theorem) If $f(n) \in \Theta(n^d)$ with $d \geq 0$ in recurrence equation (1), then.

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

(Analogous results hold for the O & Ω notations, too).

For ex, the recurrence equation for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, $a = 2, b = 2$, and $d = 0$; hence, since $a > b^d$,

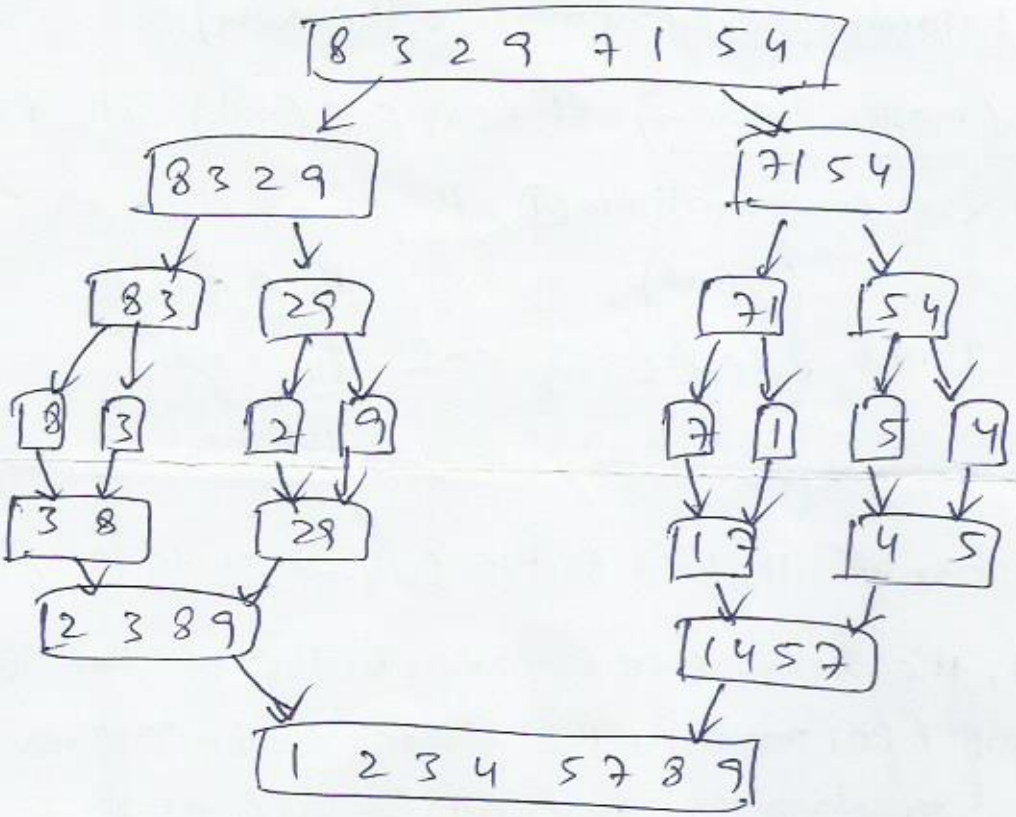
$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Ex1: Find the order of growth for solutions of the following recurrences.

1. $T(n) = 4T(n/2) + n$, $T(1) = 1$. $d = 1 \xrightarrow{\text{case 3}} \Theta(n^{\log_2 4}) = \Theta(n^2)$
2. $T(n) = 4T(n/2) + n^2$, $T(1) = 1$. $d = 2 \xrightarrow{\text{case 2}} \Theta(n^2 \log n) = \Theta(n^2 \log n)$
3. $T(n) = 4T(n/2) + n^3$, $T(1) = 1$. $d = 3 \xrightarrow{\text{case 1}} \Theta(n^d) = \Theta(n^3)$.

Mergesort

Mergesort is a perfect example of a successful application of the divide-and-conquer technique. It sorts a given array $A[0 \cdots n-1]$ by dividing it into two halves $A[0 \cdots \lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor \cdots n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.



Example of mergesort operation.

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

The merging of two sorted arrays can be done as follows: Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the array are copied to the end of the new array.

Algorithm merge($B[0 \cdots p-1]$, $C[0 \cdots q-1]$, $A[0 \cdots p+q-1]$)
// Merges two sorted arrays into one sorted array.
// Input: Arrays $B[0 \cdots p-1]$ and $C[0 \cdots q-1]$ both sorted.
// Output: Sorted array $A[0 \cdots p+q-1]$ 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].
```

How efficient is mergesort? Assuming for simplicity that n is a power of 2, the recurrence relation for the

number of key comparisons $C(n)$ is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \; C(1) = 0.$$

let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merge stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needed to be processed is reduced by one. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (ex., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{merge}(n) = n-1$, and we have the recurrence.

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \; C_{worst}(1) = 0.$$

Hence, according to the master theorem, $C_{worst}(n) \in O(n \log n)$.

Ex: Apply merge sort to sort the list E, X, A, M, P, L, E in alphabetical order.

Quicksort It is based on the divide-and-conquer approach. unlike mergesort, which divides its input's elements according to their position in the array, quicksort divides them according to their value. quicksort rearranges elements of a given array $A[0 \dots n-1]$ to achieve its partition. partition is a situation where all the elements ~~of a given array~~ $A[0], \dots, A[n-1]$ before some position s are smaller than or equal to $A[s]$ and all the elements after position s are greater than or equal to $A[s]$. i.e.,

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are} \ge A[s]}$$

Obviously, after a partition has been achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two subarrays of the elements preceding & following $A[s]$ independently. (ex, by the same method)

ALGORITHM quicksort($A[l \dots r]$)
// sorts a subarray by quicksort.
// Input: A subarray $A[l \dots r]$ of $A[0 \dots n-1]$, defined by its left and right indices l and r.
// Output: subarray $A[l \dots r]$ 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])
```

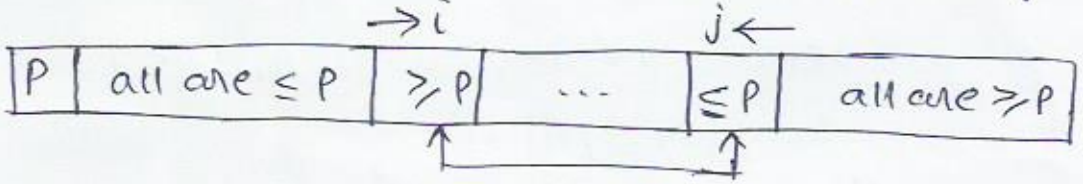
A partition of $A[0 \dots n-1]$ and its subarray $A[l \dots r]$ $(0 \le l < r \le n-1)$ can be achieved by the following algorithm.

First, we select an element w.r.to whose value we are going to divide the subarray. We call this element as the pivot. There are several different strategies for selecting a pivot; we will return to this issue when we analyze the algorithm's efficiency. For now, we use the simplest strategy of selecting the subarray's first element: $p = A[l]$.

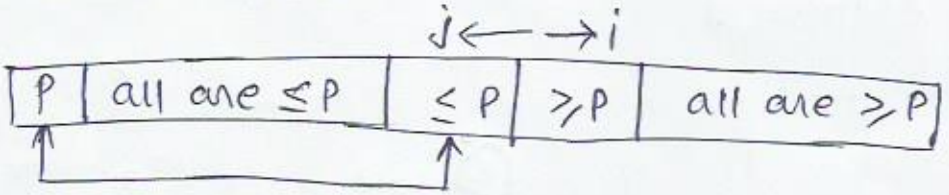
There are also several alternative procedures for rearranging elements to achieve a partition. Here we use an efficient method based on two scans of the subarray: one is left-to-right and the other right-to-left, each comparing the subarray's elements with the pivot. The left-to-right scan, denoted below by index i, starts with the second element. Since we want elements smaller than the pivot to be in the first part of the subarray, this scan skips over elements that are smaller than the pivot and stops on encountering the first element greater than or equal to the pivot. The right-to-left scan, denoted below by index j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the second part of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

After both scans stop, three situations may arise, depending on whether or not the scanning indices

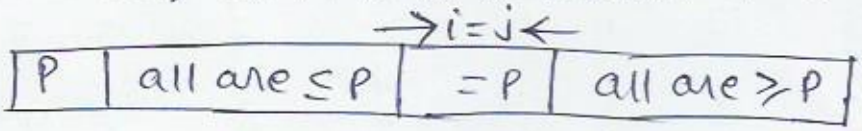
have crossed. If scanning indices i and j have not crossed, i.e., $i < j$, exchange $A[i]$ & $A[j]$ and resume the scans by incrementing i and decrementing j, respectively:



If the scanning indices have crossed over, i.e., $i > j$, you will have partitioned the array after exchanging the pivot with $A[j]$:



Finally, if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p (why?). Thus, you have the array partitioned, with the split position $s = i = j$:



You can combine the last case with the case of crossed-over indices ($i > j$) by exchanging the pivot with $A[j]$ whenever $i \geq j$.

Pseudocode implementing this partitioning procedure.

```
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 ← 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
```

Ex:

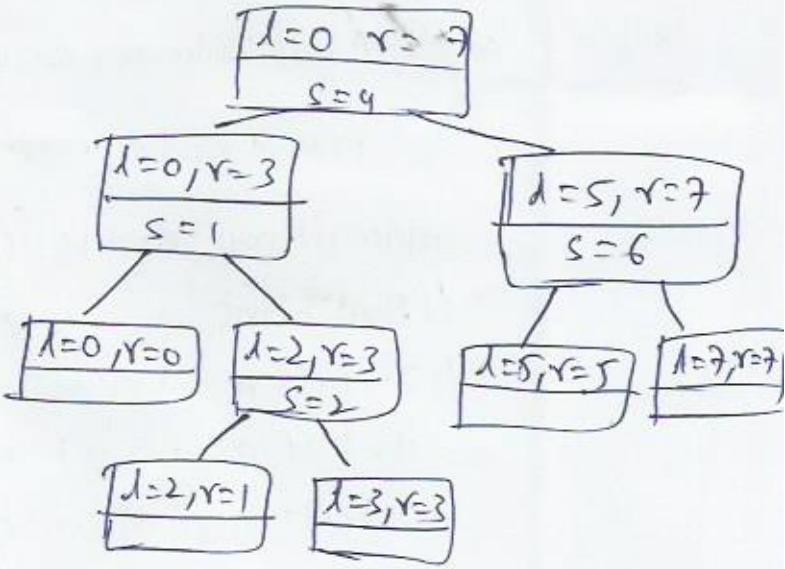
	0	1	2	3	4	5	6	7	
									l=0 r=7
Pivot	5	3	1	9	8	2	4	7	
	5	3	1	4	8	2	9	7	after swapping A[i] & A[j]
	5	3	1	4	8	2	9	7	
	5	3	1	4	2	8	9	7	after swapping A[i] & A[j]
	5	3	1	4	2	8	9	7	
	2	3	1	4	5	8	9	7	after swapping A[l] & A[j]

Subarray

Pivot 2	3	1	4	
	3	1	4	
	1	3	4	swap A[i] & A[j]
	1	3	4	
	1	2	3	swap A[l] & A[j]

Subarray

3	4	
3	4	
	4	



```
        i   j
8   9   7
        i   j
8   7   9
        j   i
8   7   9
7   8   9      swap A[j] & A[l]
7

        9
```

Example of quicksort

Quicksort efficiency analysis: If all the splits happen in the middle of corresponding subarrays, we will have the best case. The no. of key comparisons in the best case will satisfy the recurrence

$$C_{best}(n) = 2 \cdot C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

According to the master theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

```
 0   1   2   3   4   5   6   7
15   12  13  11  20  18  22  14
          (7)  ↓split
```

apply recursion to left subarray

```
14   12  13  11      15      18  22  20
     (3) ↓split              5   ↓split (2)
     1   2   3              6   7
     11  12  13  14     18   22  20
         ↓split
(2)  0   1   2
     11  12  13
         ↓split
         1   2
(1)      12  13
```

apply recursion to right subarray

```
↓split
(1)  6   7
     20  22
```

Total no. of comparisons = 7+3+2+2+1+1 = 16.

no. of comparisons in each split is $n-1$ i.e., except pivot, no. of comparisons for remaining elements.

Strassen's matrix multiplications (multiplying two square matrices)

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}_{2\times2} \qquad B = \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}_{2\times2}$$

$$C = A * B$$

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

$$= \begin{bmatrix} a_{00}\cdot b_{00} + a_{01}\cdot b_{10} & a_{00}\cdot b_{01} + a_{01}\cdot b_{11} \\ a_{10}\cdot b_{00} + a_{10}\cdot b_{10} & a_{10}\cdot b_{01} + a_{11}\cdot b_{11} \end{bmatrix}$$

Brute-force algorithm need 8 multiplications & 4 additions. Reduce no. of multiplications to 7 by increasing additions/subtractions (proposed by V. Strassen).

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

where,

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

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

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

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

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

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

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

$c_{00} = m_1 + m_4 - m_5 + m_7$ & $p+s-t+v$

$c_{01} = m_3 + m_5$ & $r+t$

$c_{10} = m_2 + m_4$ & $q+s$

$c_{11} = m_1 + m_3 - m_2 + m_6$ & $p+r-q+u$

Apply Strassen's algorithm to compute:

$$\begin{bmatrix} 1 & 2 & 1 & 1 \\ 0 & 3 & 2 & 4 \\ 0 & 1 & 1 & 1 \\ 5 & 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} 2 & 1 & 0 & 2 \\ 1 & 2 & 1 & 1 \\ 0 & 3 & 2 & 1 \\ 4 & 0 & 0 & 4 \end{bmatrix}$$

Exiting the recursion when $n=2$, compute the product of 2-by-2 matrices by brute-force algorithm.

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

$$C = A * B$$

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

$$= \begin{bmatrix} A_{00}\cdot B_{00} + A_{01}\cdot B_{10} & A_{00}\cdot B_{01} + A_{01}\cdot B_{11} \\ A_{10}\cdot B_{00} + A_{11}\cdot B_{10} & A_{10}\cdot B_{01} + A_{11}\cdot B_{11} \end{bmatrix}$$

C_{00} can be computed as either $A_{00}\cdot B_{00} + A_{01}\cdot B_{10}$ or as $m_1 + m_4 - m_5 + m_7$ where m_1, m_4, m_5, & m_7 are found by strassen's formulas. If the seven products of n/2-by-n/2 matrices are computed recursively by the same method, we have strassen's algorithm for matrix multiplication.

asymptotic efficiency — If $M(n)$ is the no. of multiplications made by strassen's algorithm in multiplying two n-by-n matrices (where n is a power of 2), we get the following recurrence relation for it:

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

$M(n/2) = 7\cdot M(n/4)$ $M(n/4) = 7\cdot M(n/8)$ $M(n/8) = 7\cdot M(n/16)$.

. . . . . .

Apply Backward substitutions, we get.

$$M(n) = 7\,M(n/2)$$
$$= 7\cdot 7\,M(n/4)$$
$$= 7^2\cdot 7\,M(n/8)$$
$$\vdots$$
$$= 7^k\,M(n/2^k)$$

let $n = 2^k$ $\therefore k = \log_2 n$.

$$M(n) = 7^{\log_2 n}\,M(1)$$

$a=7\ b=2\ n=c$ $\left[\because a^{\log_b c} = c^{\log_b a}\right]$

$$\therefore M(n) = n^{\log_2 7} \simeq n^{2.807}.$$

Brute-force algorithm takes n^3