UNIT II

DIVIDE AND CONQUER

Prepared By

J.Sahitha Banu

Assistant Professor

Department of Computer science& IT

Jamal Mohamed College,Trichy

General method

- Divide and Conquer is an algorithmic pattern.
- D**ivide and conquer algorithm** is a strategy of solving a large problem by

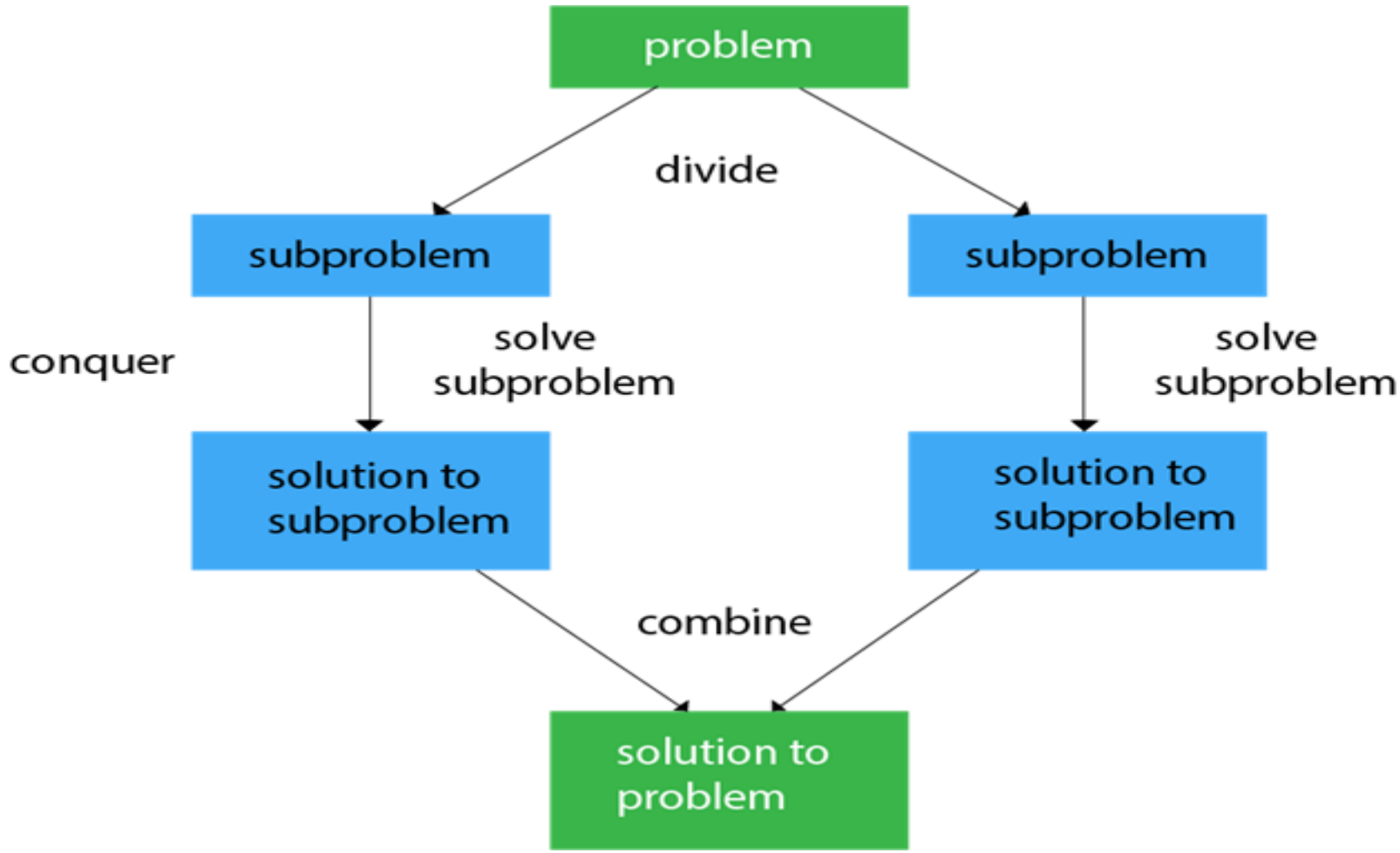
1.breaking the problem into smaller sub-problems

2.solving the sub-problems, and

3.combining them to get the desired output.

- Here are the steps involved:

1. **Divide**: Divide the given problem into sub-problems using recursion.
2. **Conquer**: Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
3. **Combine:** Combine the solutions of the sub-problems that are


problem
divide
subproblem
subproblem
conquer
solve
subproblem
solve
subproblem
solution to
subproblem
solution to
subproblem
combine
solution to
problem

Divide-and-Conquer

- A general methodology for using recursion to design efficient algorithms
- It solves a problem by:
  - Diving the data into parts
  - Finding sub solutions for each of the parts
  - Constructing the final answer from the sub solutions

THE DIVIDE AND CONQUER ALGORITHM

```
Divide_Conquer(problem P)
{
  if Small(P) return S(P);
  else {

    divide P into smaller instances P1, P2, ...,
  Pk, k≥1;

    Apply Divide_Conquer to each of these
  subproblems;

    return Combine(Divide_Conque(P1),
  Divide_Conque(P2),...,  Divide_Conque(Pk));
  }
}
```

- The Divide and Conquer algorithm is initially invoked as DAndC(P) where P is the problem to be solved.
- Small(P) is a Boolean valued function that determine whether the input size is small and answer can be computed with out splitting.
- If this is so then the function S is called.
- Otherwise P is divided in to smaller subproblems.
- The subproblems P1,P2…. Are solved by recursive applications of DAndC.
- Combine is a function that determines the solutions to P using the solutions to k subproblems.
- The computing time is described by recurrence relation

$$T(n)=\begin{cases} g(n) & n \text{ small} \\ T(n1)+T(n2)\ldots..+T(nk)+f(n) & \text{Otherwise} \end{cases}$$

- Where T(n)is the time for DAndC on any input of size n and g(n) is the time to compute for small inputs
- The function f(n) is the time to divide P and combining the solutions to sub problems
- The time complexity to solve such problems is given by a recurrence relation: –
- $T(n) = \begin{cases} T(1) & n=1 \\ a \cdot T(n/b) + f(n) & n>1 \end{cases}$
- Time to combine the solutions of the subproblems into a solution of the original problem.
- Where a and b are constants.
- The method to solve such recurrence relation is called the substitution method.
- This method repeatedly makes substitution for each occurrence until all occurrence disappear.

Example 3.1 Consider the case in which $a = 2$ and $b = 2$. Let $T(1) = 2$ and $f(n) = n$. We have

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + 3n \\ &\vdots \end{aligned}$$

In general, we see that $T(n) = 2^i T(n/2^i) + in$, for any $\log_2 n \geq i \geq 1$. In particular, then, $T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + n \log_2 n$, corresponding to the choice of $i = \log_2 n$. Thus, $T(n) = nT(1) + n \log_2 n = n \log_2 n + 2n$. □

Defective chess board problem

The Defective Chessboard problem, also known as the **Tiling Problem** is an interesting problem.

It is typically solved with a “divide and conquer” approach.

The algorithm has a time complexity of ***O(n²)***.

*A defective chess board is a 2^k*2^k board of squares with exactly one defective square.*

As the Size of these chessboards displayed here, 1x1 and 2x2 we don't have to place any L-shape tile in the first and rest can be filled by just using 1 L-shaped tile.

When k=0,there is one possible defective chess board.

For any k there are exactly 2^2k defective chess boards.

The problem

- Given a *n* by *n* board where *n* is of form 2^k where k >= 1 (Basically, n is a power of 2 with minimum value as 2).
- We are required to tile the board using triominos. A triomino is an L-shaped tile in a 2 × 2 block with one cell of size 1×1 missing.

Constraints

1.Tiling two triominoes may not overlap.

2.Triminoes should not cover defective square

3.Triominoes should cover all other squares.

So the number of triominoes used becomes **(2^2k-1)/3.**

When k=0 it has no defective square and number of triominoes is 0.

When k=1, there are exactly three non defective squares and these squares are covered using a triomino as shown already.

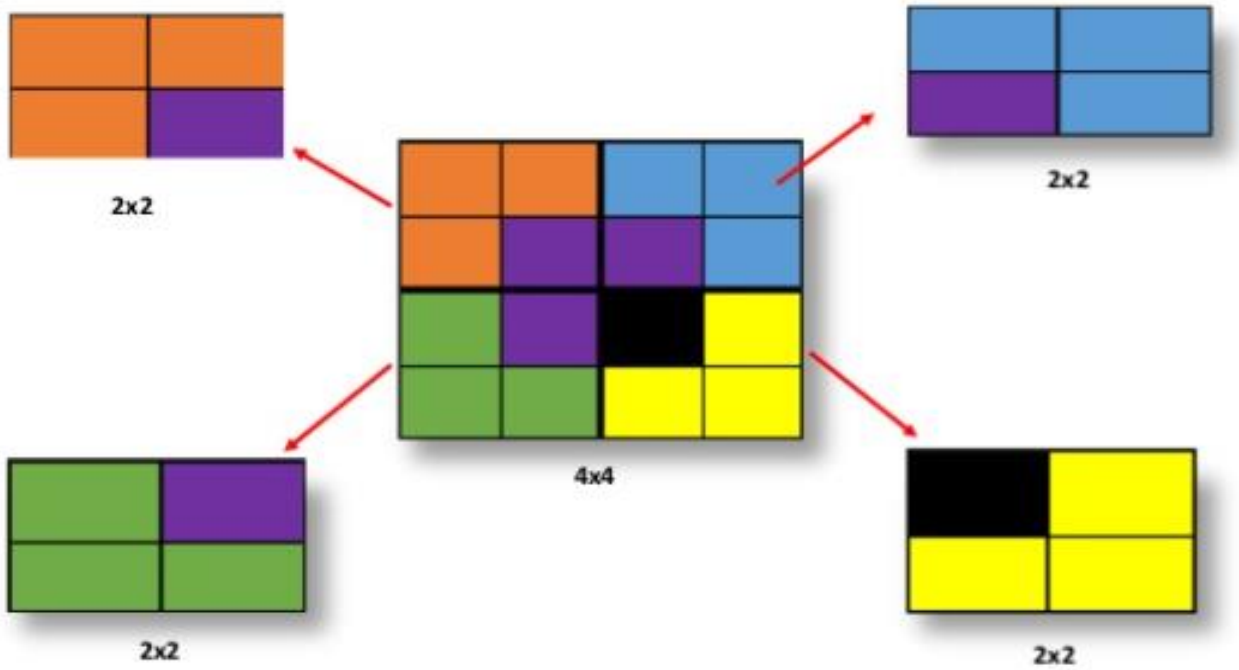
The Algorithm

- As mentioned earlier, a divide-and-conquer (DAC) technique is used to solve the problem.
- DAC entails splitting a larger problem into sub-problems, ensuring that each sub-problem is an exact copy of the larger one.

o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o

The method suggests reducing the problem of tiling a 2^k*2^k defective chess board to that of tiling smaller defective chess boards.

A natural partitioning of a 2^k*2^k chess board would be in to four 2^k-1*2^k-1 chessboards



The above partitioning can be done when k=2.

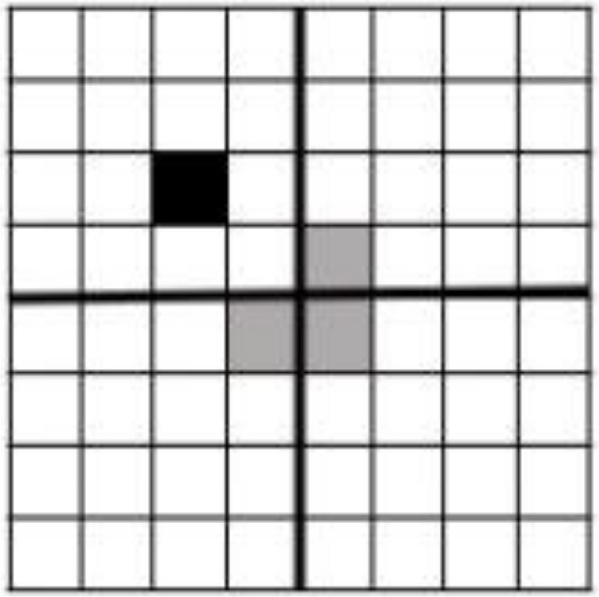
o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o
o	o	o	o	o	o	o	o
o	o	o	o	■	o	o	o

ALGORITHIM

1. For a 2x2 board with defective cell we just need to add the single tile to it.

2. We will place a L shape tile in the middle such that it does not cover the sub square in which there is already defective. All the cell have a defective cell.

3.Repeat this process recursively till we have a 2x2 board.



The recursion terminates when the chess board size has been reduced to 1*1.

The chess board’s only square is defective and no triminoes are to be placed.

Consider the divide and conquer alg.

Small(P) is true when k=0(we are dealing with 1*1 defective chess board);dividing P into smaller instances is done by dividing P in to 2^k-1*2^k-1.;combine the solutions obtained recursively for the four smaller requires no additional work.

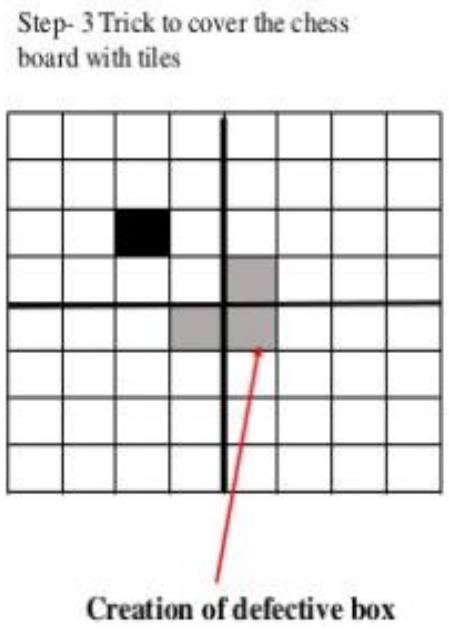
```
Algorithm TileBoard(topRow, topColumn, defectRow, defectColu
// topRow is row number of top-left corner of board.
// topColumn is column number of top-left corner of board.
// defectRow is row number of defective square.
// defectColumn is column number of defective square.
// size is length of one side of chess board.
{
    if (size = 1) return ;
    tileToUse := tile + +;
    quadrantSize := size/2;

    // tile top-left quadrant
    if (defectRow < topRow + quadrantSize&&
            defectColumn < topColumn + quadrantSize) ther
        // defect is in this quadrant
        TileBoard(topRow, topColumn, defectRow, defectColun
            quadrantSize);
    else
    {
        // no defect in this quadrant
        // place a tile in bottom-right corner
        board[topRow + quadrantSize - 1][topColumn + quad
            := tileToUse;
        // tile the rest
        TileBoard(topRow, topColumn, topRow + quadrantSiz
            topColumn + quadrantSize - 1, quadrantSize);
    }

    // code for remaining three quadrants is similar
}
```

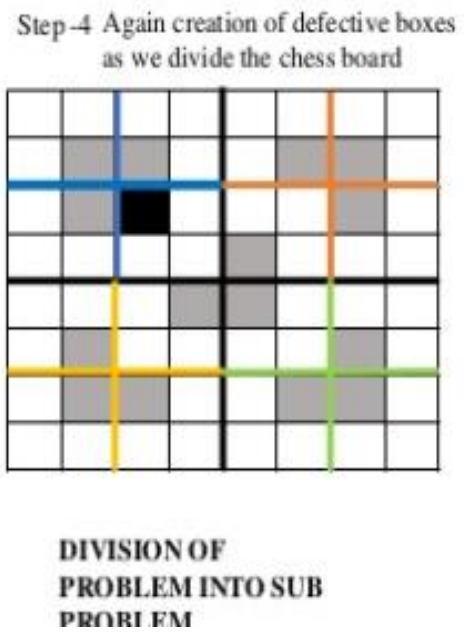
The alg. Uses two global variable board and tile.

board is a two dimensional array that represents a chess board. board[0,0] represents the top left corner of chess board.

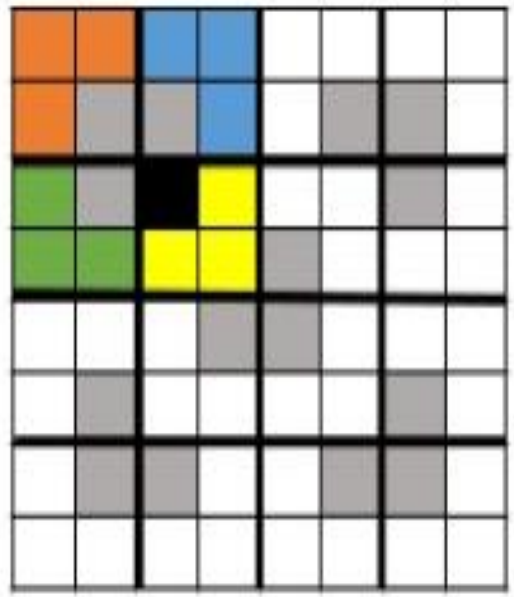
tile is a integer value with initial value 1

Initial invocation is TileBoard(0,0,dRow,dCol,Size)

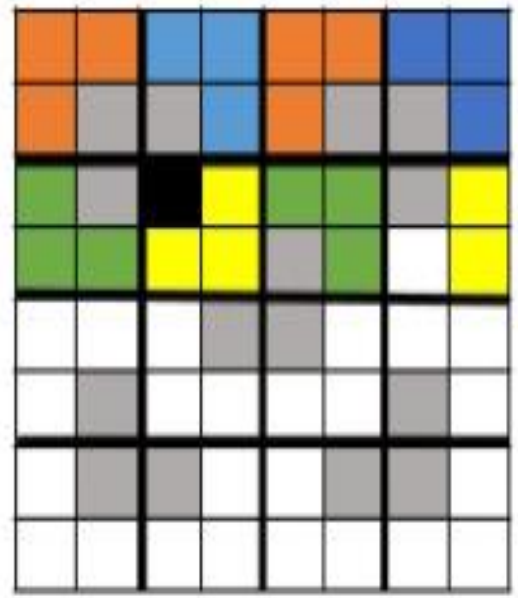




Step-5 As we have finally divided the problem into 2x2 board we will put the tiles.



Step-6 The procedure will continue until all the sub board are covered with the tiles.



Time complexity

$$t(k)=\begin{cases} d & k=0 \\ 4t(k-1)+c & k>0 \end{cases}$$

When k=0 size =1 and a constant d amount of time is spent.

When k>0 four recursive calls are made.The call take 4t(k-1) time.

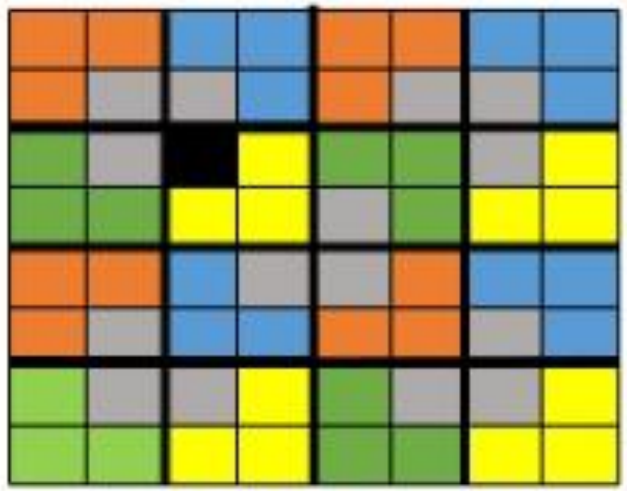
We solve this by using substitution method.

$t(k)=O(4^k)$

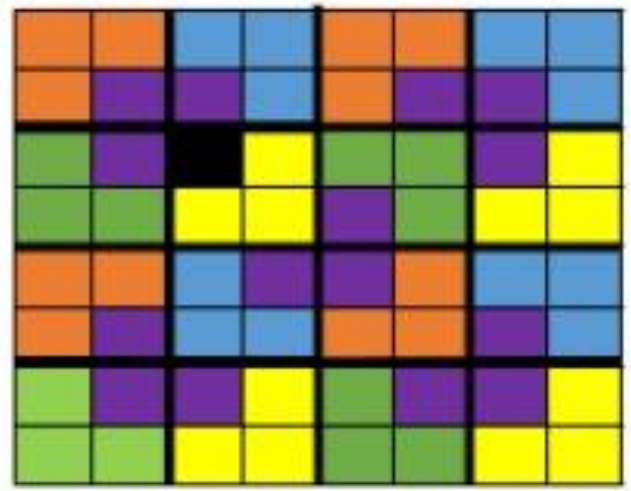
$=O(\text{number of tiles needed})$

O(1) time needed to place each tile and we cannot obtain an asymptotically faster alg than divide and conquer.

Step-7 The final chess board covered with all the titles and only left with the defectives which we created.



Step-7 Here we will cover the defectives which we have created as in the last, there should be only one defective left.



COMBINIG OF ALL SUB PROBLEMS

Binary Search

Let a_i 1<=i<=n be a list of elements that are sorted in non decreasing order.

The problem is determining whether a element is present in the list or not.

If x is present we are to determine a value j such that a_j=x.

If x is not present the j is set with 0.

Divide and conquer is used to solve the problem.

Let P=(n,ai,…….al,x) denote a arbitrary instance of this search problem.

Let Small(P) be true if n=1.

In this case S(P) will take the value I if x=a_i otherwise it is 0.

If P has more than one element then it is divided in to new sub problems as follows.

Pick an index q (in the range [i,l] and compare x with a_q

There are three possibilities.

1.x=a_q the problem P is immediately solved

2.x<a_q; then x to be searched in the sublist ai,ai+1…..a_q-1. Therefore p reduces to (q-1, ai,ai+1…..a_q-1,x)

3.x>a_q then x to be searched in the sublist a_q+1,……al. P reduces to (l-q,a_q+1….al,x)

The given problem reduced to two sub problems and it take O(1) time.

After a comparison with a_q, the instance remaining to be solved using divide and conquer again.

If q is always chosen such that a_q is the middle element (q=[n+1/2]) then the resulting search algorithm is known as binary search.

The answer to the new sub problem is also the problem to the original problem.

```
1    Algorithm BinSrch(a, i, l, x)
2    // Given an array a[i : l] of elements in nondecreasing
3    // order, 1 ≤ i ≤ l, determine whether x is present, and
4    // if so, return j such that x = a[j]; else return 0.
5    {
6        if (l = i) then  // If Small(P)
7        {
8            if (x = a[i]) then return i;
9            else return 0;
10       }
11       else
12       { // Reduce P into a smaller subproblem.
13           mid := ⌊(i + l)/2⌋;
14           if (x = a[mid]) then return mid;
15           else if (x < a[mid]) then
16                    return BinSrch(a, i, mid − 1, x);
17                else return BinSrch(a, mid + 1, l, x);
18       }
19   }
```

Algorithm 3.2 Recursive binary search

```
1   Algorithm BinSearch(a, n, x)
2   // Given an array a[1 : n] of elements in nondecreasing
3   // order, n ≥ 0, determine whether x is present, and
4   // if so, return j such that x = a[j]; else return 0.
5   {
6       low := 1; high := n;
7       while (low ≤ high) do
8       {
9           mid := ⌊(low + high)/2⌋;
10          if (x < a[mid]) then high := mid − 1;
11          else if (x > a[mid]) then low := mid + 1;
12              else return mid;
13      }
14      return 0;
15  }
```

Algorithm 3.3 Iterative binary search

Example 3.6 Let us select the 14 entries

$$-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$$

place them in $a[1:14]$, and simulate the steps that BinSearch goes through as it searches for different values of x. Only the variables low, $high$, and mid need to be traced as we simulate the algorithm. We try the following values for x: 151, -14, and 9 for two successful searches and one unsuccessful search. Table 3.2 shows the traces of BinSearch on these three inputs. □

$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

Table 3.2 Three examples of binary search on 14 elements

Comparisons between x and a[] are referred as element comparisons.

The number of comparisons needed to find each of the 14 element is

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Elements:	−15	−6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons:	3	4	2	4	3	4	1	4	3	4	2	4	3	4

The average is obtained by summing the comparisons needed to find all items and dividing by 14

45/14=3.21 comparisons per successful search.

The average number of element comparisons for unsuccessful search is (3+14 *4)/15=3.93

For a successful search the time complexity is O(log n)

For unsuccessful search the time complexity is Θ(log n)

Computing time of binary search algorithm for best, average and worst case is

For successful search

Θ(1)-best case

Θ(log n)-average case

Θ(log n)-worst case

For unsuccessful search

Θ(log n)- for best,average and worst case

Finding the maximum and minimum

1. Let us consider simple problem that can be solved by the divide-and conquer technique.
2. The problem is to find the maximum and minimum value in a set of 'n' elements.
3. By comparing numbers of elements, the time complexity of this algorithm can be analyzed.
4. Hence, the time is determined mainly by the total cost of the element comparison.

```
Algorithm straight MaxMin (a, n, max, min)
// Set max to the maximum & min to the minimum of a [1: n]
{
Max = Min = a [1];
For i = 2 to n do
{
If (a [i] > Max) then Max = a [i];
If (a [i] < Min) then Min = a [i];
}}
```

- **Explanation:**
- Straight MaxMin requires 2(n-1) element comparisons in the best, average & worst cases.
- By realizing the comparison of a [i]max is false, improvement in a algorithm can be done.
- Hence we can replace the contents of the for loop by, If (a [i]> Max) then Max = a [i]; Else if (a [i]< 2(n-1)
- On the average a[i] is > max half the time, and so, the avg. no. of comparison is 3n/2-1.
- The best case occurs when the elements are in increasing order and number of comparisons are n-1.
- The worst case occurs when the elements are in decreasing order and the number of comparisons are 2(n-1).

- **A Divide and Conquer Algorithm for this problem would proceed as follows:**
- a. Let P = (n, a [i],.....,a [j]) denote an arbitrary instance of the problem.
- b. Here 'n' is the no. of elements in the list (a [i],.....,a[j]) and we are interested in finding the maximum and minimum of the list.
- c. If the list has more than 2 elements, P has to be divided into smaller instances.
- d. For example, we might divide 'P' into the 2 instances, P1=([n/2],a[1],.....a[n/2]) & P2= (n-[n/2], a[[n/2]+1],....., a[n]) .
- After having divided 'P' into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm

```
1    Algorithm MaxMin(i, j, max, min)
2    // a[1 : n] is a global array. Parameters i and j are integers,
3    // 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
4    // largest and smallest values in a[i : j], respectively.
5    {
6         if (i = j) then max := min := a[i]; // Small(P)
7         else if (i = j − 1) then  // Another case of Small(P)
8              {
9                   if (a[i] < a[j]) then
10                  {
11                       max := a[j]; min := a[i];
12                  }
13                  else
14                  {
15                       max := a[i]; min := a[j];
16                  }
17             }
18             else
19             {    // If P is not small, divide P into subproblems.
20                  // Find where to split the set.
21                       mid := ⌊(i + j)/2⌋;
22                  // Solve the subproblems.
23                       MaxMin(i, mid, max, min);
24                       MaxMin(mid + 1, j, max1, min1);
25                  // Combine the solutions.
26                       if (max < max1) then max := max1;
27                       if (min > min1) then min := min1;
28             }
29   }
```

Complexity:

If T(n) represents this no., then the resulting recurrence relations is

$$T(n) = \begin{cases} T([n/2]+T[n/2]+2 & n>2 \\ 1 & n=2 \\ 1 & n=1 \end{cases}$$

When 'n' is a power of 2, n=2k for some positive integer 'k', then

$T(n) = 2T(n/2) + 2$

$= 2(2T(n/4)+2)+2$

$= 4T(n/4)+4+2$

*

*

$= 2k-1\ T(2) + \Sigma\ 1 \le I \le k-1 \le 2i$

$= 2k-1+ 2k - 2$

$T(n) = (3n/2) - 2$

Note that (3n/2) - 2 is the best-average and worst-case no. of comparisons when 'n' is a power of 2

1 2 3 4 5 6 7 8 9

22,13,-5,-8,15,60,17,31,47

i=j

1=9 false

i=j-1

1=8 false

mid=i+j/2

=1+9/2=5

mid=1+5/2=3

mid=6+9/2=7

mid=1+3/2=2

1 2 3 4 5 6 7 8 9

22,13,-5,-8,15,60,17,31,47

1,9,60,-8

1,5,22,-8 6,9,60,17

1,3,22,-5 4,5,-5,-8 6,7,60,17 8,9,47,31

1,2,22,13 3,3,-5,-5

Merge sort

Merge sort is a sorting technique based on divide and conquer technique.

With worst- case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort is one of the most efficient sorting.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Steps

1.Divide: Divide the unsorted list into two sub lists of about half the size.

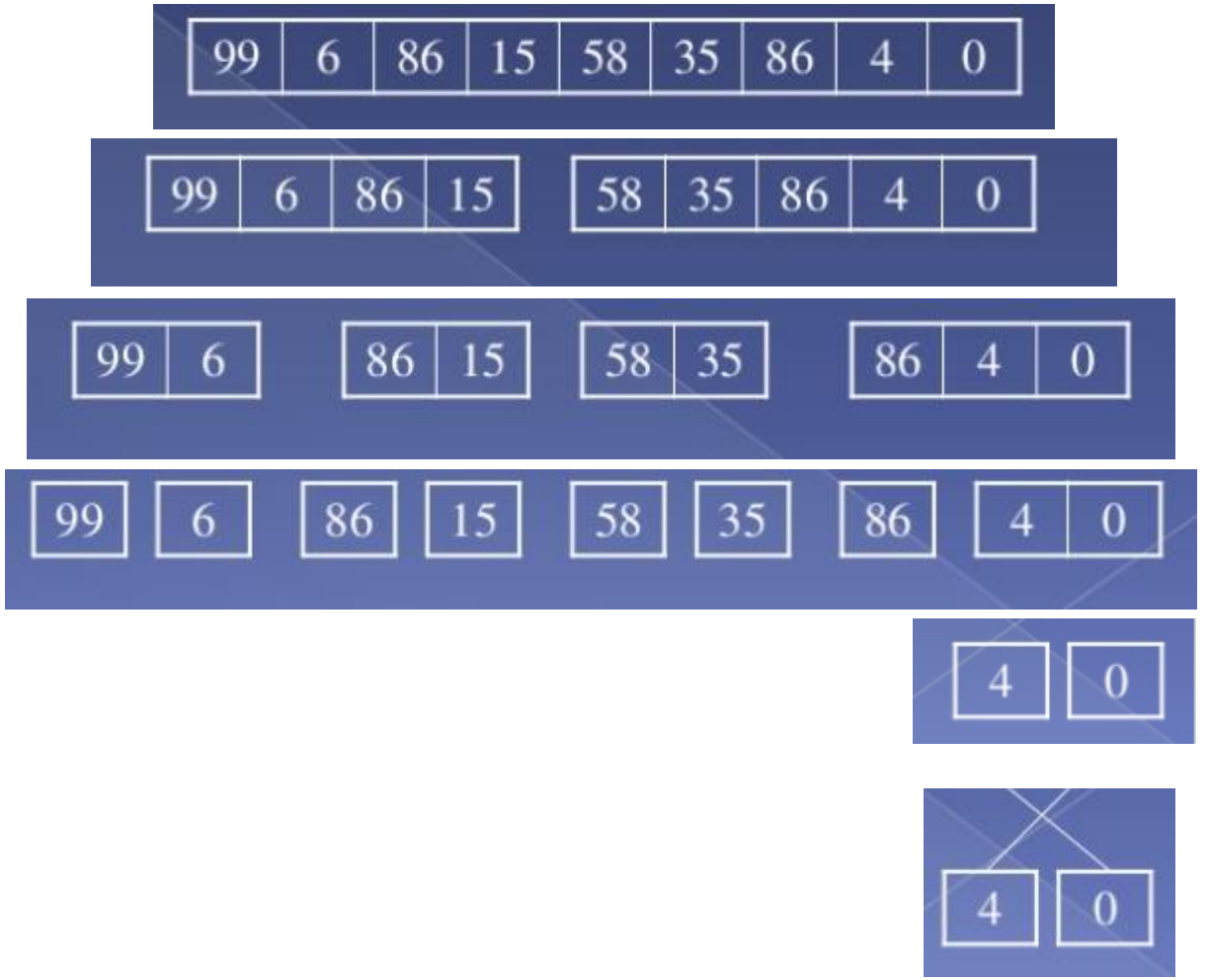
2.Conquer: Sort each of the two sub lists recursively until we have list sizes of length 1,in which case the list itself is returned.

3.Combine: Merge the two-sorted sub lists back into one sorted list.

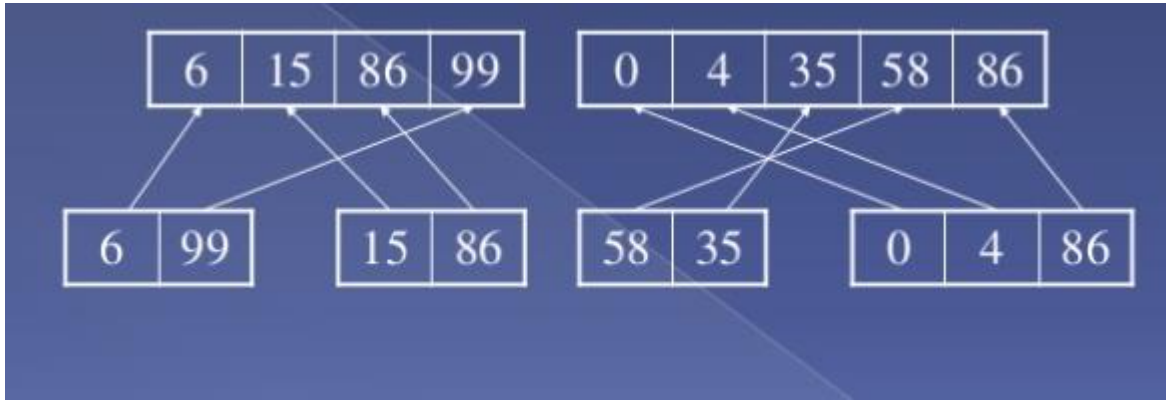
```
1   Algorithm MergeSort(low, high)
2   // a[low : high] is a global array to be sorted.
3   // Small(P) is true if there is only one element
4   // to sort. In this case the list is already sorted.
5   {
6       if (low < high) then  // If there are more than one element
7       {
8           // Divide P into subproblems.
9               // Find where to split the set.
10                  mid := ⌊(low + high)/2⌋;
11          // Solve the subproblems.
12              MergeSort(low, mid);
13              MergeSort(mid + 1, high);
14          // Combine the solutions.
15              Merge(low, mid, high);
16      }
17  }
```

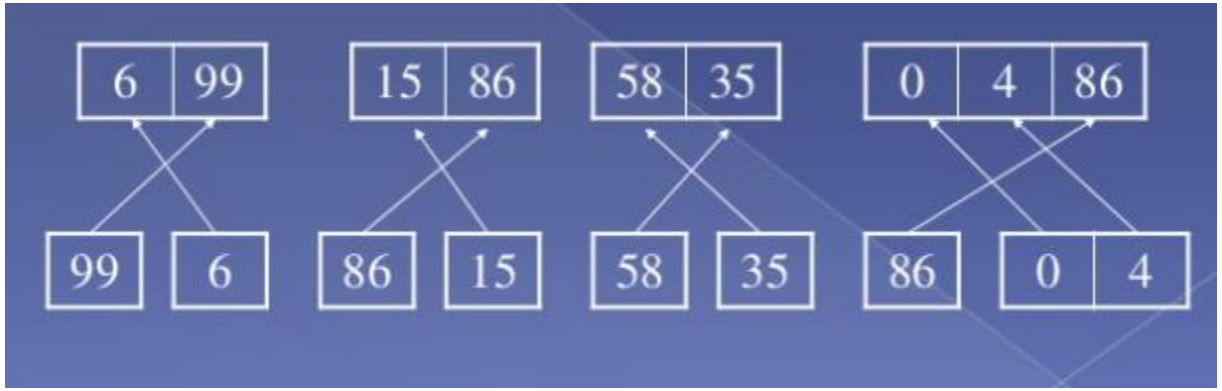
```
1    Algorithm Merge(low, mid, high)
2    // a[low : high] is a global array containing two sorted
3    // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4    // is to merge these two sets into a single set residing
5    // in a[low : high]. b[ ] is an auxiliary global array.
6    {
7        h := low; i := low; j := mid + 1;
8        while ((h ≤ mid) and (j ≤ high)) do
9        {
10           if (a[h] ≤ a[j]) then
11           {
12               b[i] := a[h];h := h + 1;
13           }
14           else
15           {
16               b[i] := a[j]; j := j + 1;
17           }
18           i := i + 1;
19       }
20       if (h > mid) then
21           for k := j to high do
22           {
23               b[i] := a[k]; i := i + 1;
24           }
25       else
26           for k := h to mid do
27           {
28               b[i] := a[k]; i := i + 1;
29           }
30       for k := low to high do a[k] := b[k];
31   }
```

- Example


99 6 86 15 58 35 86 4 0
99 6 86 15
58 35 86 4 0
99 6
86 15
58 35
86 4 0
99
6
86
15
58
35
86
4 0
4 0
4
0


0 4 6 15 35 58 86 86 99


6 15 86 99
0 4 35 58 86
6 99
15 86
58 35
0 4 86


6 99
15 86
58 35
0 4 86
99 6
86 15
58 35
86 0 4

Time complexity

If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation

$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

When n is a power of 2, $n = 2^k$, we can solve this equation by successive substitutions:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &\vdots \\ &= 2^k T(1) + kcn \\ &= an + cn \log n \end{aligned}$$

It is easy to see that if $2^k < n \leq 2^{k+1}$, then $T(n) \leq T(2^{k+1})$. Therefore

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

Quick Sort

Quicksort is a divide-and-conquer algorithm.

It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.

The sub-arrays are then sorted recursively.

This can be done in-place, requiring small additional amounts of memory to perform the sorting.

It can be about two or three times faster than its main competitors, merge sort and heapsort.

merged. In quicksort, the division into two subarrays is made so that the sorted subarrays do not need to be merged later. This is accomplished by rearranging the elements in $a[1:n]$ such that $a[i] \leq a[j]$ for all i between 1 and m and all j between $m+1$ and n for some m, $1 \leq m \leq n$. Thus, the elements in $a[1:m]$ and $a[m+1:n]$ can be independently sorted. No merge is needed. The rearrangement of the elements is accomplished by picking some element of $a[\ ]$, say $t = a[s]$, and then reordering the other elements so that all elements appearing before t in $a[1:n]$ are less than or equal to t and all elements appearing after t are greater than or equal to t. This rearranging is referred to as *partitioning*.

```
1   Algorithm QuickSort(p, q)
2   // Sorts the elements a[p], ..., a[q] which reside in the global
3   // array a[1 : n] into ascending order; a[n + 1] is considered to
4   // be defined and must be ≥ all the elements in a[1 : n].
5   {
6       if (p < q) then  // If there are more than one element
7       {
8           // divide P into two subproblems.
9               j := Partition(a, p, q + 1);
10                  // j is the position of the partitioning element.
11          // Solve the subproblems.
12              QuickSort(p, j − 1);
13              QuickSort(j + 1, q);
14          // There is no need for combining solutions.
15      }
16  }
```

- https://www.gatevidyalay.com/tag/quick-sort-ppt/

```
1    Algorithm Partition(a, m, p)
2    // Within a[m], a[m + 1], . . . , a[p − 1] the elements are
3    // rearranged in such a manner that if initially t = a[m],
4    // then after completion a[q] = t for some q between m
5    // and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
6    // for q < k < p. q is returned. Set a[p] = ∞.
7    {
8         v := a[m]; i := m; j := p;
9         repeat
10        {
11             repeat
12                  i := i + 1;
13             until (a[i] ≥ v);

14             repeat
15                  j := j − 1;
16             until (a[j] ≤ v);

17             if (i < j) then Interchange(a, i, j);

18        } until (i ≥ j);

19        a[m] := a[j]; a[j] := v; return j;
20   }

1    Algorithm Interchange(a, i, j)
2    // Exchange a[i] with a[j].
3    {
4         p := a[i];
5         a[i] := a[j]; a[j] := p;
6    }
```

Time Complexity

- To find the location of an element that splits the array into two parts, O(n) operations are required.
- This is because every element in the array is compared to the partitioning element.
- After the division, each section is examined separately.
- If the array is split approximately in half (which is not usually), then there will be $\log_2 n$ splits.
- Therefore, total comparisons required are $f(n) = n \times \log_2 n = O(n\log_2 n)$.

Worst case

- Quick Sort is sensitive to the order of input data.
- It gives the worst performance when elements are already in the ascending order.
- It then divides the array into sections of 1 and (n-1) elements in each call.
- Then, there are (n-1) divisions in all.
- Therefore, here total comparisons required are $f(n) = n \times (n-1) = O(n^2)$.

- Quick sort

- 1 2 3 4 5 6 7 8 9 10
- 65 70 75 80 85 60 55 50 45 *
- m=1 j=10 v=65
- i=i+1=1+1=2
- a[i]>=v
- 70>=65 true
- j=9
- a[j]<=v
- 45<=65 true
- 2<9 true
- 65 45 75 80 85 60 55 50 70 *
- i=3
- 75 >=65 true
- j=8
- 50<=65 true
- 3<8 true
- 65 45 50 80 85 60 55 75 70 *

- i=4
- 80>=65 true
- j=7
- 55<=65 true
- 4<=7

- 65 45 50 55 85 60 80 75 70 *
- i=5
- 85>=65 true
- j=6
- 60<=65 true
- 5<6 true
- 65 45 50 55 60 85 80 75 70 *
- i=6

- 85>=65 true
- j=5
- 60<=65 true
- 6<5 false no interchange
- a[m]=a[j] a[j]=m return j
- a[1]= 60 a[5]=65
- j=5
- 60 45 50 55 65 85 80 75 70 *

- 45 50 55 60 65 70 75 80 *

Selection Sort

The Partition algorithm of Section 3.5 can also be used to obtain an efficient solution for the selection problem. In this problem, we are given n elements $a[1 : n]$ and are required to determine the kth-smallest element. If the partitioning element v is positioned at $a[j]$, then $j-1$ elements are less than or equal to $a[j]$ and $n-j$ elements are greater than or equal to $a[j]$. Hence if $k < j$, then the kth-smallest element is in $a[1 : j-1]$; if $k = j$, then $a[j]$ is the kth-smallest element; and if $k > j$, then the kth-smallest element is the $(k-j)$th-smallest element in $a[j+1 : n]$. The resulting algorithm is function Select1 (Algorithm 3.17). This function places the kth-smallest element into position $a[k]$ and partitions the remaining elements so that $a[i] \leq a[k]$, $1 \leq i < k$, and $a[i] \geq a[k]$, $k < i \leq n$.

This sorting algorithm is an in-place comparison- based algorithm .

In which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end.

Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array.

This process continues moving unsorted array boundary by one element to the right

Each time selecting an item according to its ordering and placing it in the correct position in the sequence

1.The Selection Sort Algorithm

For each index position i

Find the smallest data value in the array from positions i through length - 1 , where length is the number of data values stored.

Exchange (swap) the smallest value with the value at position i.

Algorithm

Step 1 − Set MIN to location 0

Step 2 − Search the minimum element in the list

Step 3 − Swap with value at location MIN

Step 4 − Increment MIN to point to next element

Step 5 − Repeat until list is sorted

```
1    Algorithm Select1(a, n, k)
2    // Selects the kth-smallest element in a[1 : n] and places it
3    // in the kth position of a[ ]. The remaining elements are
4    // rearranged such that a[m] ≤ a[k] for 1 ≤ m < k, and
5    // a[m] ≥ a[k] for k < m ≤ n.
6    {
7         low := 1; up := n + 1;
8         a[n + 1] := ∞; // a[n + 1] is set to infinity.
9         repeat
10        {
11             // Each time the loop is entered,
12             // 1 ≤ low ≤ k ≤ up ≤ n + 1.
13             j := Partition(a, low, up);
14                  // j is such that a[j] is the jth-smallest value in a[ ].
15             if (k = j) then return;
16             else if (k < j) then up := j; // j is the new upper limit.
17                  else low := j + 1; // j + 1 is the new lower limit.
18        } until (false);
19   }
```

Algorithm 3.17 Finding the kth-smallest element

Time Complexity Analysis-

- Selection sort algorithm consists of two nested loops.
- Owing to the two nested loops, it has $O(n^2)$ time complexity.

	Time Complexity
Best Case	n^2
Average Case	n^2
Worst Case	n^2

- **Important Notes-**

- Selection sort is not a very efficient algorithm when data sets are large.
- This is indicated by the average and worst case complexities.
- Selection sort uses minimum number of swap operations O(n) among all the sorting algorithms.
-

- Strassen’s Matrix Multiplication
- Basic Matrix Multiplication

```
void matrix_mult ()
{
for (i = 1; i <= N; i++)
 {
   for (j = 1; j <= N; j++)
 {
  for(k=1;k<=N;k++)
   {
    compute Ci,j; }
   }}
```

$$C_{i,j} = \sum_{k=1}^{N} a_{i,k} b_{k,j}$$

$$\text{Thus } T(N) = \sum_{i-1}^{N} \sum_{j-1}^{N} \sum_{k-1}^{N} c = cN^3 = O(N^3)$$

Basic Matrix Multiplication

Suppose we want to multiply two matrices of size N x N: for example A x $B = C$.

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$$\mathbf{C_{11} = a_{11}b_{11} + a_{12}b_{21}}$$

$$\mathbf{C_{12} = a_{11}b_{12} + a_{12}b_{22}}$$

$$\mathbf{C_{21} = a_{21}b_{11} + a_{22}b_{21}}$$

$$\mathbf{C_{22} = a_{21}b_{12} + a_{22}b_{22}}$$

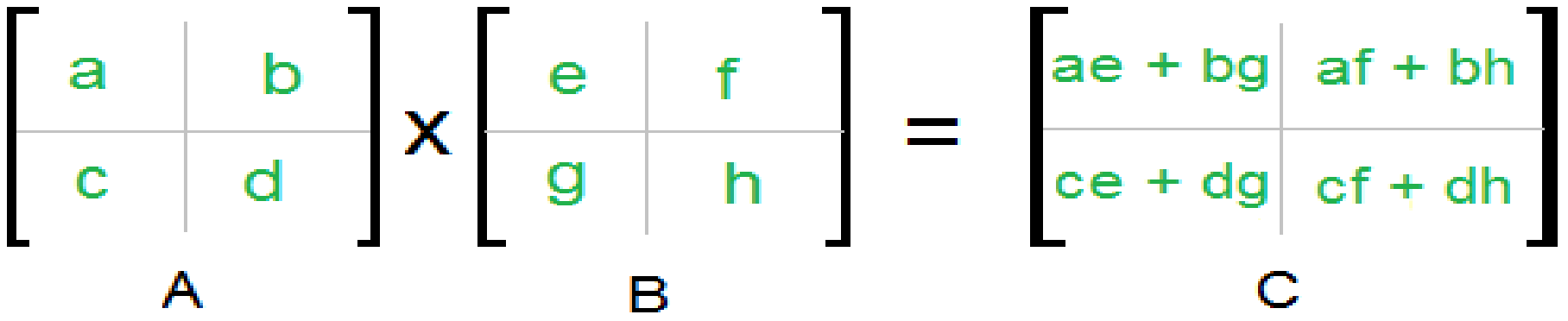
2x2 matrix multiplication can be accomplished in 8 multiplication.($\mathbf{2^{\log_2 8} = 2^3}$)

Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.

1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.

2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.



A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

- In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

$$T(N) = 8T(N/2) + O(N^2)$$

From Master's Theorem, time complexity of above method is $O(N^3)$ which is unfortunately same as the above naive method.

- ***Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?***
  In the above divide and conquer method, the main component for high time complexity is 8 recursive calls.
- The idea of **Strassen's method** is to reduce the number of recursive calls to 7.
- Strassen's method is similar to above simple divide and conquer method
- This method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

$$p1 = a(f - h) \qquad p2 = (a + b)h$$
$$p3 = (c + d)e \qquad p4 = d(g - e)$$
$$p5 = (a + d)(e + h) \qquad p6 = (b - d)(g + h)$$
$$p7 = (a - c)(e + f)$$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as

$$T(N) = 7T(N/2) + O(N^2)$$

From Master's Theorem, time complexity of above method is $O(N^{Log7})$ which is approximately $O(N^{2.8074})$

Generally Strassen's Method is not preferred for practical applications for following reasons.

1) The constants used in Strassen's method are high and for a typical application Naive method works better.

2) For Sparse matrices, there are better methods especially designed for them.

3) The submatrices in recursion take extra space.

4) Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen's algorithm than in Naive Method (Source: CLRS Book)