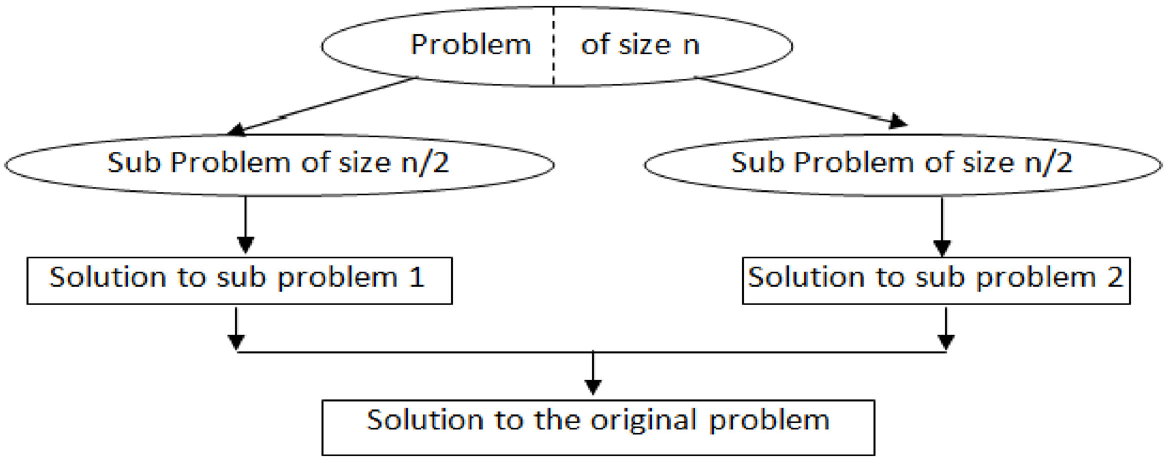
## Divide-and-conquer

Divide-and-conquer algorithms work 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 (typically recursively, though sometimes a different algorithm is employed when instances become small enough).
3. If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.


Problem of size n
Sub Problem of size n/2
Sub Problem of size n/2
Solution to sub problem 1
Solution to sub problem 2
Solution to the original problem

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances $P_1, P_2, \ldots, P_k$, $k \geq 1$;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC($P_1$),DAndC($P_2$),...,DAndC($P_k$));
    }
}
```

## Recurrence equation for Divide and Conquer

If the size of problem 'p' is n and the sizes of the 'k' sub problems are $n_1, n_2 \ldots n_k$, respectively, then

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases}$$

Where,

- T(n) is the time for divide and conquer method on any input of size n and
- g(n) is the time to compute answer directly for small inputs.
- The function f(n) is the time for dividing the problem 'p' and combining the solutions to sub problems.

## Recurrence equation for Divide and Conquer

- Generally, an instance of size **n** can be divided into **b** Instances of size **n/b**,
- Assuming $n = b^k$,

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

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

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

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

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

$$T(1) = 2$$

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

$$= 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n$$

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

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

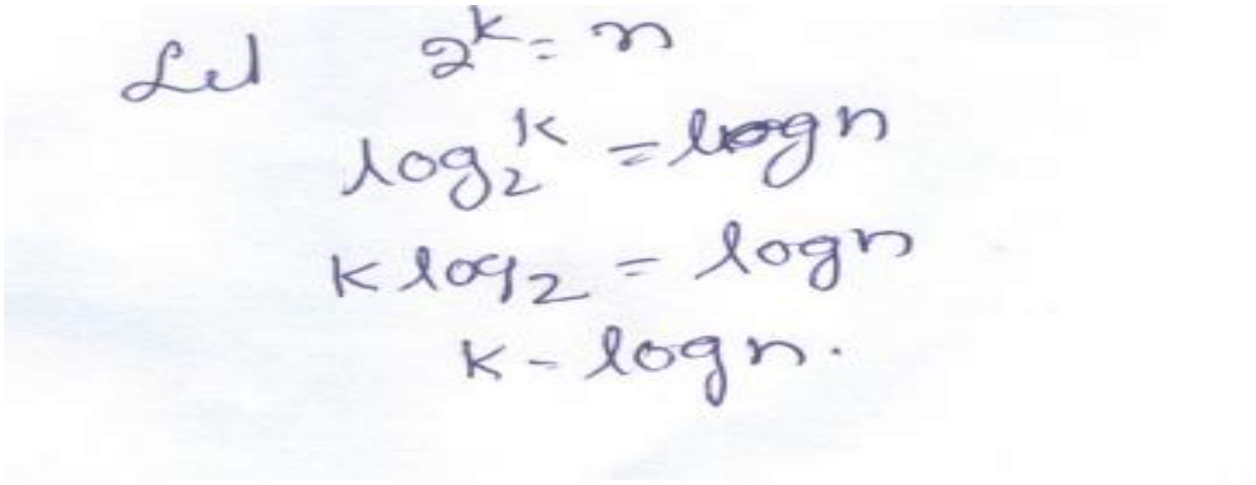
$$= 4\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n$$

$$= 8T\left(\frac{n}{8}\right) + \frac{4n}{4} + 2n$$

$$= 8T\left(\frac{n}{8}\right) + 3n$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

$$= 2^k T\left(\frac{n}{2^k}\right) + kn$$

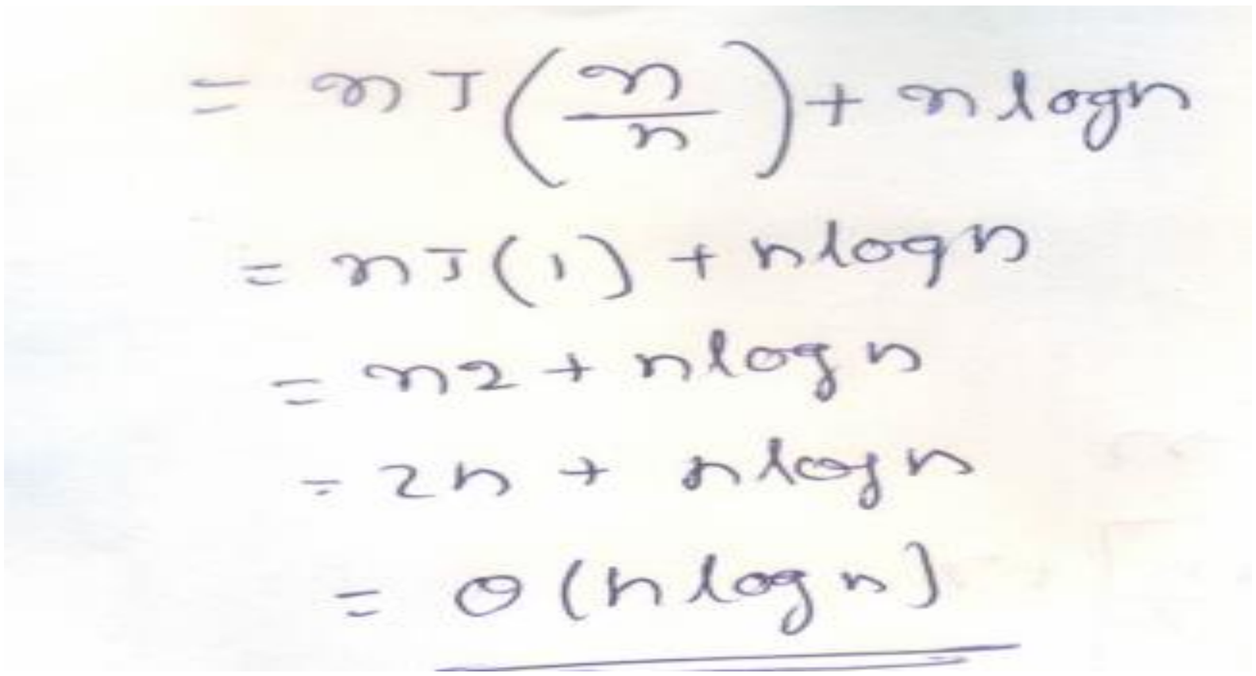


Let $2^k = n$

$\log_2^k = \log n$

$k \log_2 = \log n$

$k - \log n.$



$= nT\left(\frac{n}{n}\right) + n\log n$

$= nT(1) + n\log n$

$= n2 + n\log n$

$= 2n + n\log n$

$= \Theta(n\log n)$

**Solving recurrence relation using Master theorem**

It states that, in recurrence equation $T(n) = aT(n/b) + f(n)$, If $f(n) \in \Theta(n^d)$ where $d \geq 0$ then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the *O* and Ω notations, too.

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

Here a = 2, b = 2, and d = 0; hence, since a >$b^d$,

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

## Binary Search

Binary search is a remarkably efficient algorithm for searching in a sorted array. It works by comparing a search key $K$ with the array's middle element $A[m]$. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if $K < A[m]$, and for the second half if $K > A[m]$:

$$\begin{array}{c} K \\ \updownarrow \\ \underbrace{A[0]\dots A[m-1]}_{\substack{\text{search here if}\\ K<A[m]}} \;\; A[m] \;\; \underbrace{A[m+1]\dots A[n-1]}_{\substack{\text{search here if}\\ K>A[m]}}. \end{array}$$

```
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
    $l \leftarrow 0$; $r \leftarrow n - 1$
    while $l \le r$ 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
```

As an example, let us apply binary search to searching for $K = 70$ in the array

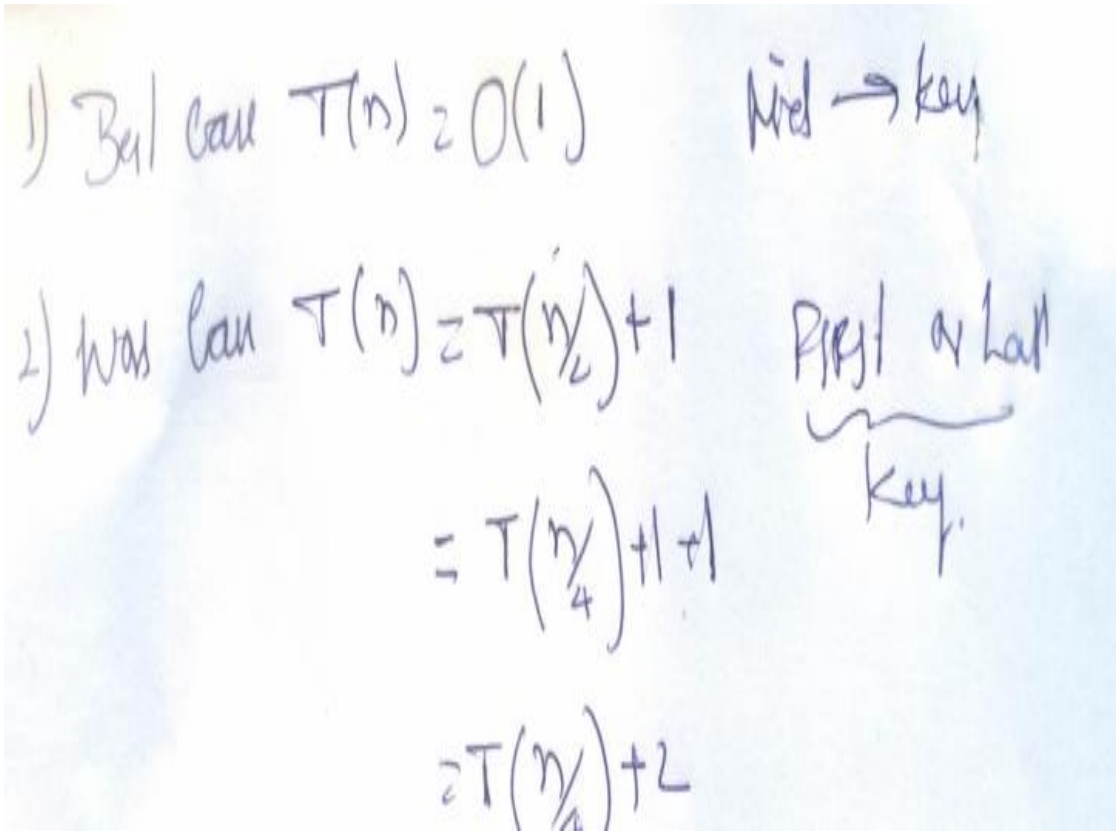
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 14 | 27 | 31 | 39 | 42 | 55 | 70 | 74 | 81 | 85 | 93 | 98 |

The iterations of the algorithm are given in the following table:

| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| value | 3 | 14 | 27 | 31 | 39 | 42 | 55 | 70 | 74 | 81 | 85 | 93 | 98 |
| iteration 1 | *l* | | | | | | *m* | | | | | | *r* |
| iteration 2 | | | | | | | | *l* | | *m* | | | *r* |
| iteration 3 | | | | | | | | *l,m* | *r* | | | | |

Recursive Binary search algorithm

```
int BinSrch(Type a[], int i, int l, Type x)
// Given an array a[i:l] of elements in nondecreasing
// order, 1<=i<=l, determine whether x is present, and
// if so, return j such that x == a[j]; else return 0.
{
   if (l==i) { // If Small(P)
      if (x==a[i]) return i;
      else return 0;
   }
   else { // Reduce P into a smaller subproblem.
      int mid = (i+l)/2;
      if (x == a[mid]) return mid;
      else if (x < a[mid]) return BinSrch(a,i,mid-1,x);
      else return BinSrch(a,mid+1,l,x);
   }
}
```

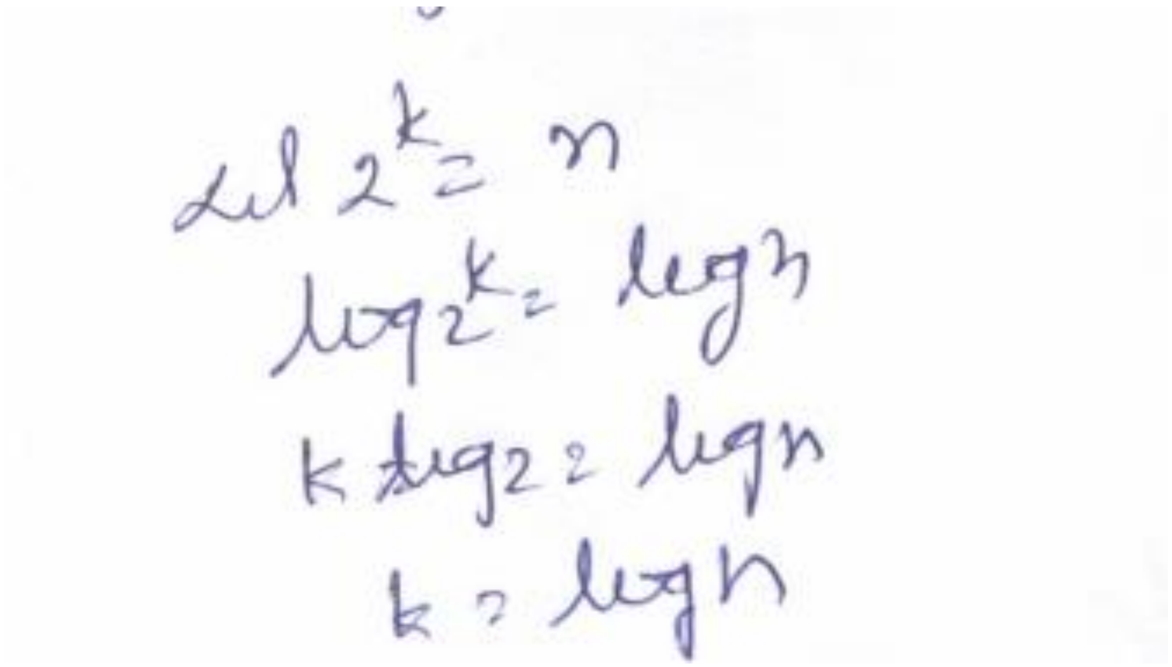


1) Best case $T(n) = O(1)$ Mid $\rightarrow$ key

2) Worst case $T(n) = T(n/2) + 1$ $\underbrace{\text{First or Last}}_{\text{key}}$

$= T(n/4) + 1 + 1$

$= T(n/4) + 2$

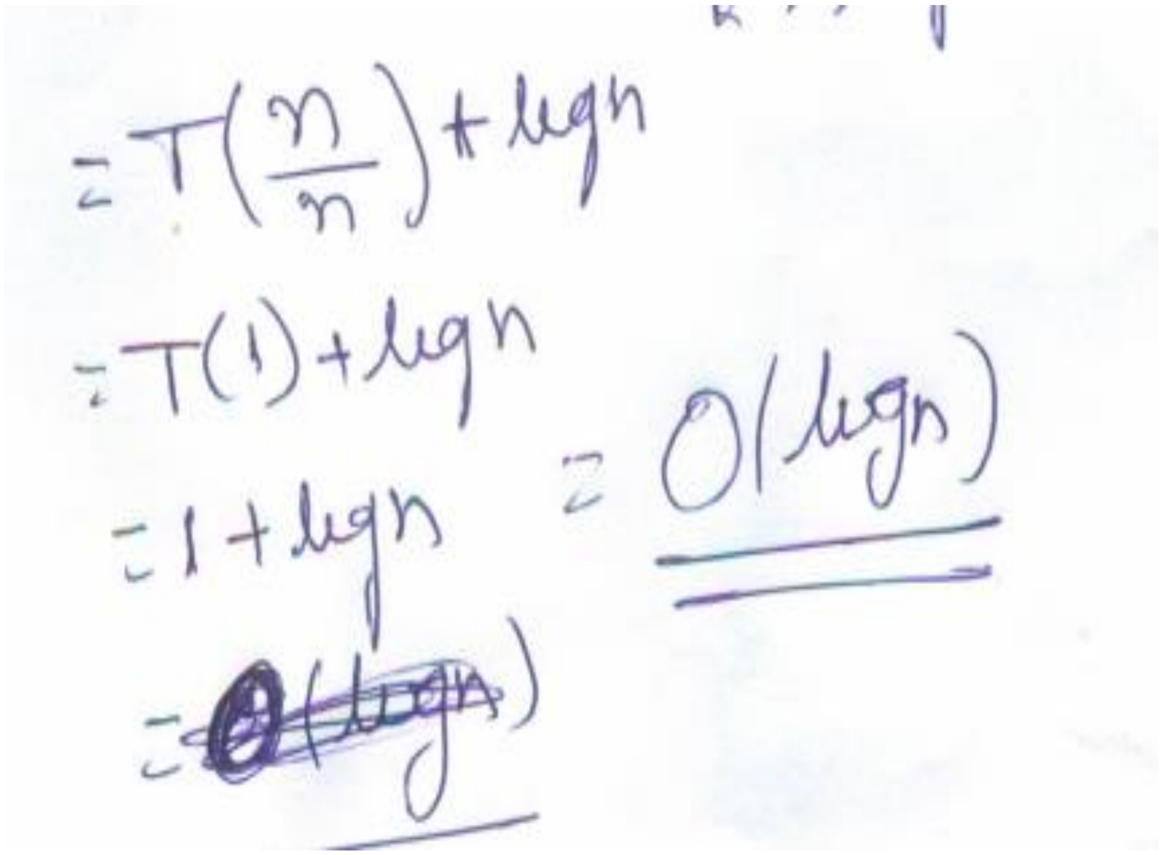


Let $2^k = n$

$\log 2^k = \log n$

$k \log 2 = \log n$

$k = \log n$



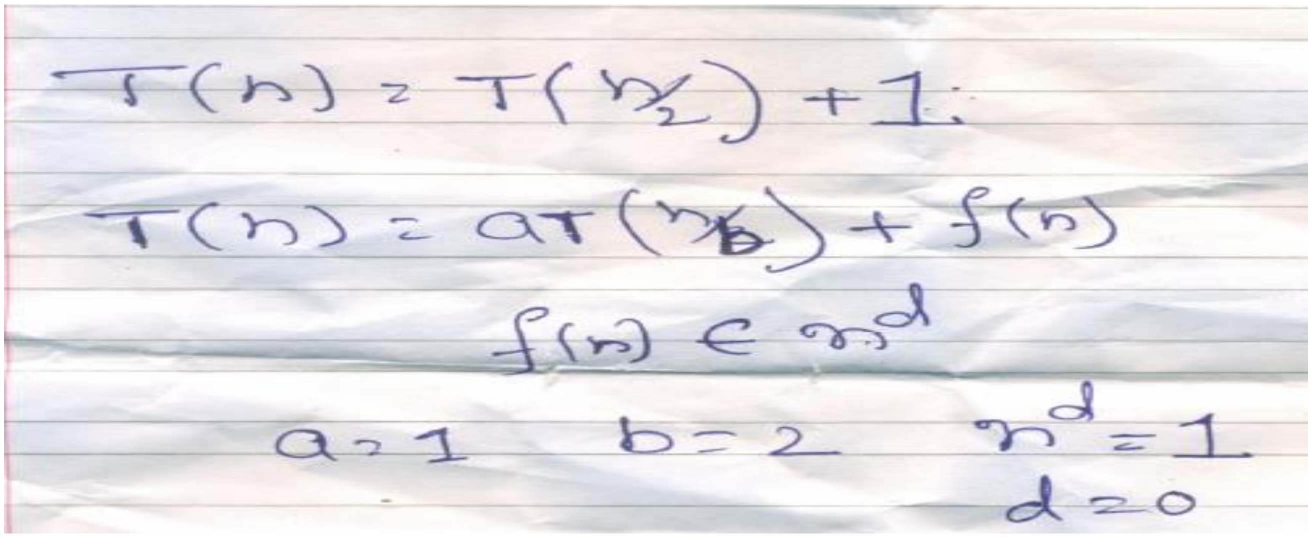
$= T\left(\frac{n}{n}\right) + \log n$

$= T(1) + \log n$

$= 1 + \log n = \underline{\underline{O(\log n)}}$

$= \cancel{O(\log n)}$

Using Master Theorems



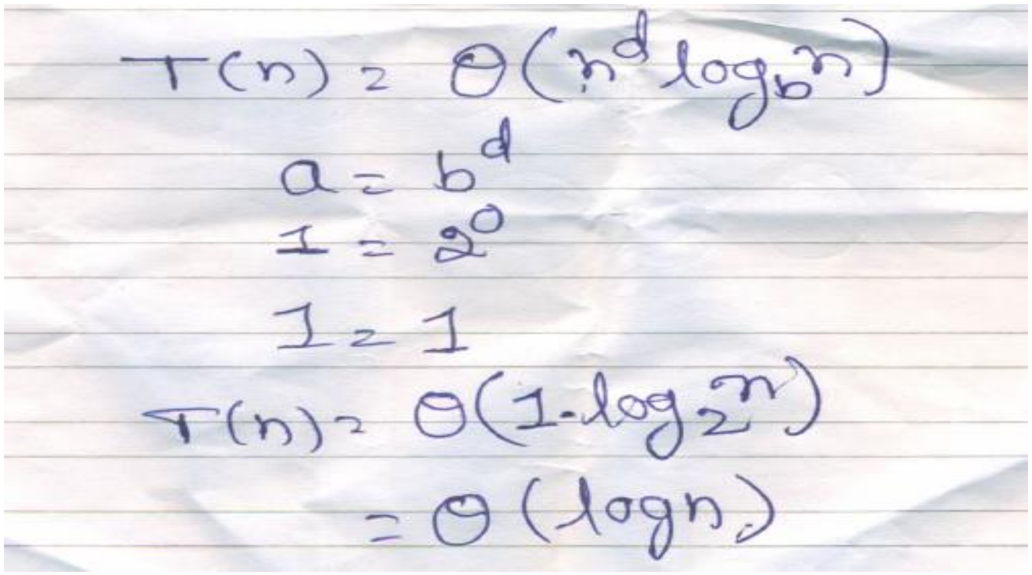
$T(n) = T(n/2) + 1$

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

$f(n) \in n^d$

$a = 1 \quad b = 2 \quad n^d = 1$

$d = 0$



$T(n) = \Theta(n^d \log_b n)$

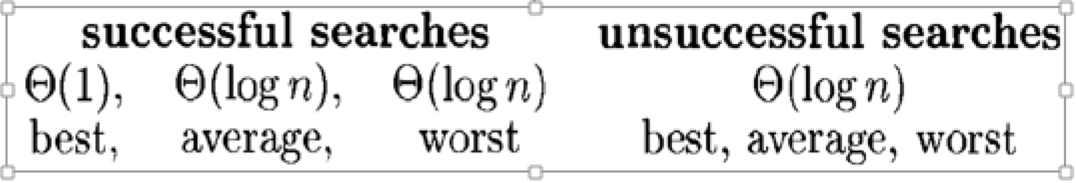
$a = b^d$

$1 = 2^0$

$1 = 1$

$T(n) = \Theta(1 \cdot \log_2 n)$

$= \Theta(\log n)$

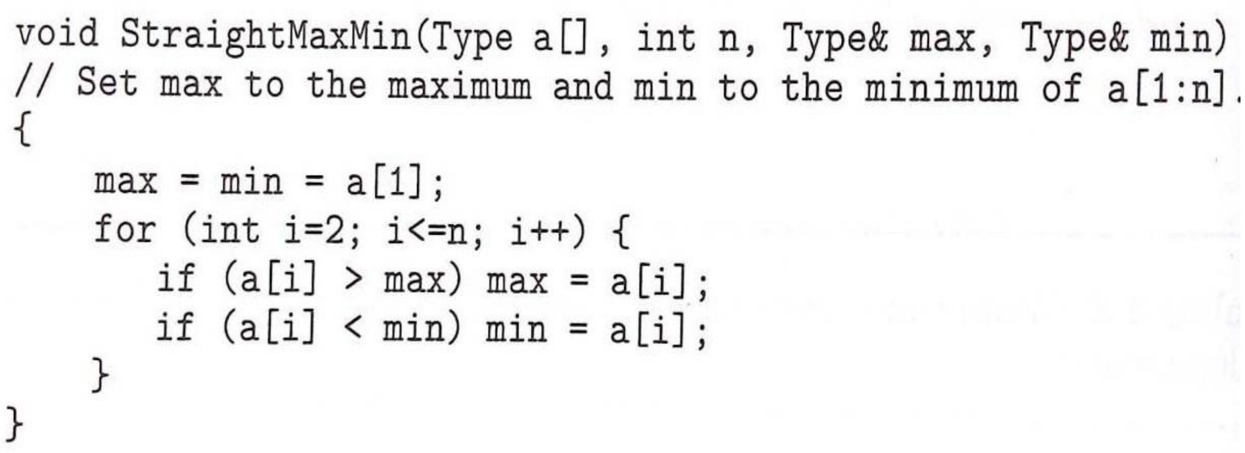


| successful searches | | | unsuccessful searches |
|---|---|---|---|
| $\Theta(1)$, | $\Theta(\log n)$, | $\Theta(\log n)$ | $\Theta(\log n)$ |
| best, | average, | worst | best, average, worst |

Max Min

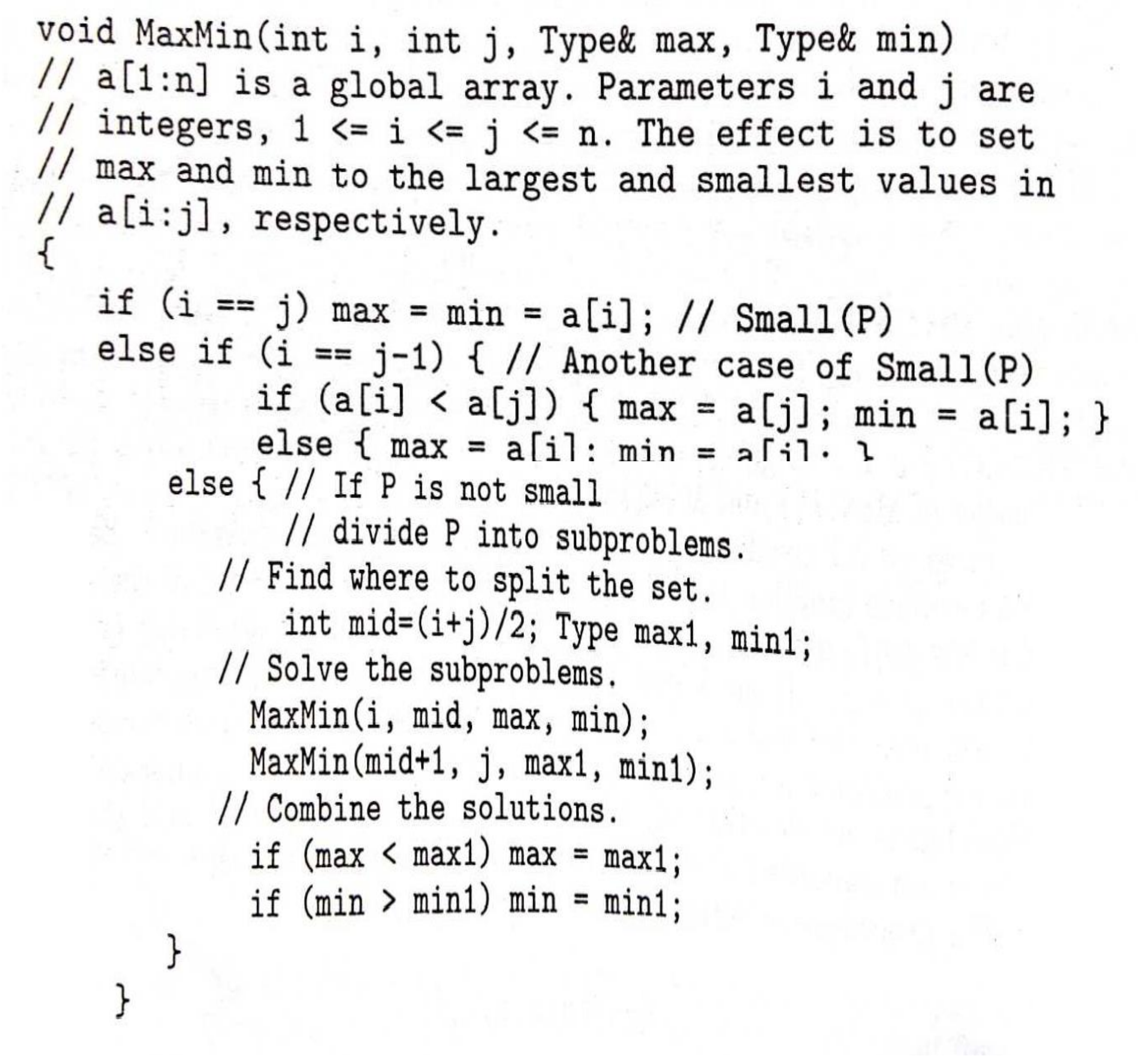
**Problem statement**

- Given a list of n elements, the problem is to find the maximum and minimum items. A simple and straight forward algorithm to achieve this is given below.



```
void StraightMaxMin(Type a[], int n, Type& max, Type& min)
// Set max to the maximum and min to the minimum of a[1:n].
{
    max = min = a[1];
    for (int i=2; i<=n; i++) {
        if (a[i] > max) max = a[i];
        if (a[i] < min) min = a[i];
    }
}
```

Algorithm based on D & C strategy

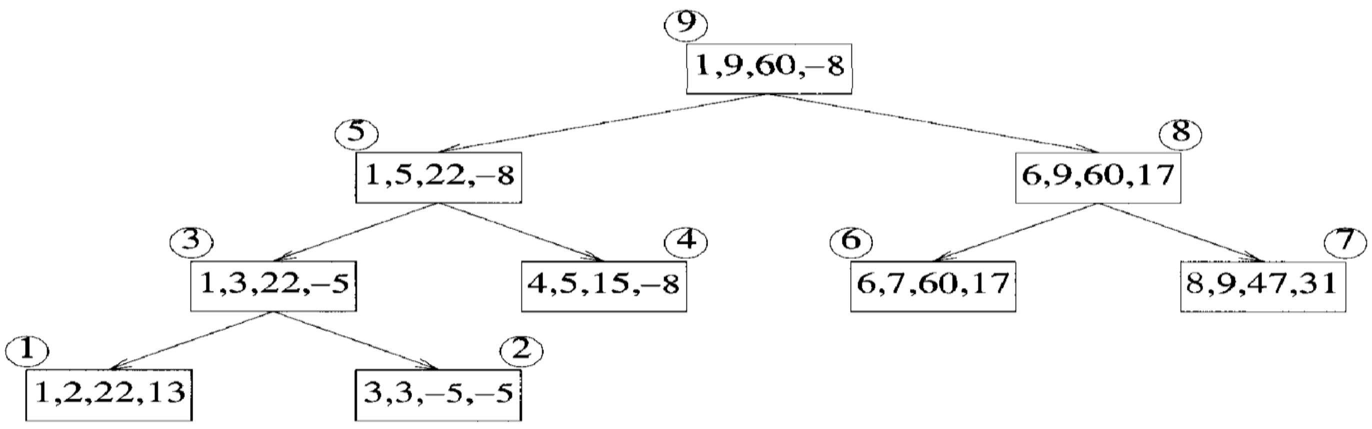


```
void MaxMin(int i, int j, Type& max, Type& min)
// a[1:n] is a global array. Parameters i and j are
// integers, 1 <= i <= j <= n. The effect is to set
// max and min to the largest and smallest values in
// a[i:j], respectively.
{
    if (i == j) max = min = a[i]; // Small(P)
    else if (i == j-1) { // Another case of Small(P)
            if (a[i] < a[j]) { max = a[j]; min = a[i]; }
            else { max = a[i]; min = a[j]; }
        else { // If P is not small
              // divide P into subproblems.
            // Find where to split the set.
              int mid=(i+j)/2; Type max1, min1;
            // Solve the subproblems.
             MaxMin(i, mid, max, min);
             MaxMin(mid+1, j, max1, min1);
            // Combine the solutions.
             if (max < max1) max = max1;
             if (min > min1) min = min1;
        }
   }
```

Example

Suppose we simulate MaxMin on the following nine elements:

| $a$: | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |
|---|---|---|---|---|---|---|---|---|---|
| | 22 | 13 | −5 | −8 | 15 | 60 | 17 | 31 | 47 |



$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ 2T(n/2)+2 & \text{if } n>2 \end{cases}$$

$1 \leftarrow$ Divi

$1 \leftarrow$ Compari

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

$$= 2\left[2T(n/4) + 2\right] + 2$$

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

$$= 4\left[2T(n/8) + 2\right] + 4 + 2$$

$$= 8T(n/8) + 8 + 4 + 2$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 2^3 + 2^2 + 2^1$$

$$= 2^i T\left(\frac{n}{2^i}\right) + 2^i + 2^{i-1} + 2^{i-2}$$

$$= 2^{k-1} . T\left(\frac{n}{2^{k-1}}\right) + 2^{k-1} + 2^{k-2} + 2^{k-3}$$

$$= 2^{k-1} . T\left(\frac{n}{2^{k-1}}\right) + \sum_{i=1}^{k-1} 2^i$$

$$= \frac{2^k}{2} T\left(\frac{2n}{2^k}\right) + \sum_{i=1}^{k-1} 2^i \quad \boxed{\frac{a(r^n - 1)}{r-1}}$$

$a = 2$
$r = 2$

Let $2^k = n$

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

$$\frac{2(2^{k-1}-1)}{1}$$

$$= 2\,2^{k-1} - 2$$

$$= \cancel{2}\frac{2^{k}}{\cancel{2}} - 2$$

$$= 2^{k} - 2$$

$$= \underline{\underline{n - 2}}$$

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

$$= \frac{n}{2}\cdot 1 + n - 2$$

$$= \frac{n}{2} + n - 2$$

$$= \frac{n + 2n - 4}{2}$$

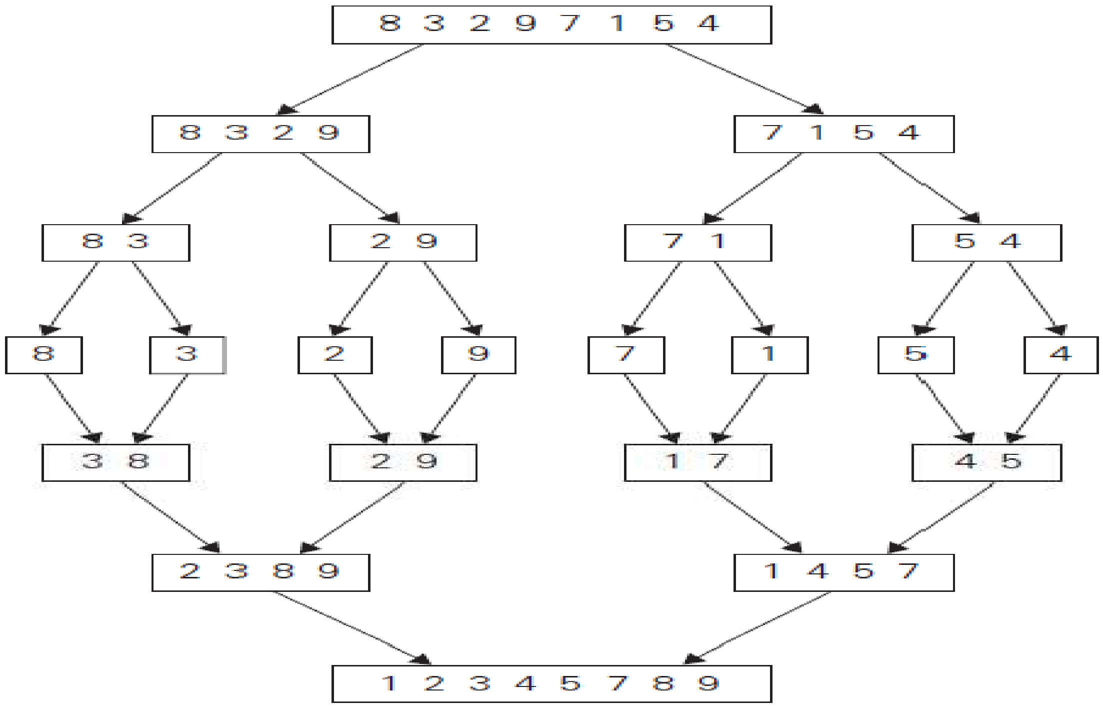
$$= \frac{3n}{2} - \frac{4}{2}$$

$$= \frac{3n}{2} - 2$$

$$= \Theta(n)$$

## Merge Sort

- Merge sort is a perfect example of divide-and conquer technique.
- It sorts a given array by
  - dividing it into two halves,
  - sorting each of them recursively, and
  - Then merging the two smaller sorted arrays into a single sorted one.



```
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)    //see below
```

```
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 $B[i] \le C[j]$
            $A[k] \leftarrow B[i]$;  $i \leftarrow i+1$
        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 $A[k..p+q-1]$
```

$T(1) = 1 \quad n = 1$

$$T(n) = 2T\left[\frac{n}{2}\right] + n$$

$$= 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n$$

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

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

$$= 4\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n$$

$$= 8T\left(\frac{n}{8}\right) + 3n$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

Let $2^k = n$

$\log 2^k = \log n$

$k \log_2 = \log n$

$k = \log n.$

$= nT\left(\frac{n}{n}\right) + n\lg n$

$= nT(1) + n\lg n$

$= n + n\lg n = \Theta(n\lg n)$

Best
Average.

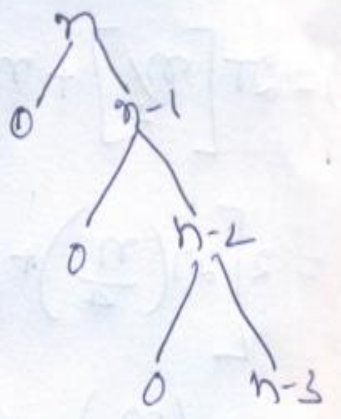
Worst case $= \frac{n(n+1)}{2}$

$= \frac{n^2+n}{2}$

$= \frac{n^2}{2} + \frac{n}{2}$

$= O(n^2)$

1 2 3 4



$= n + (n-1) + (n-2) + \cdots 1$

$= n(n+1)$

## Quick Sort

- Quicksort divides (or partitions) array according to the value of some pivot element A[s]
- Divide-and-Conquer:
    - If n=1 terminate (every one-element list is already sorted)
    - If n>1, partition elements into two; based on pivot element

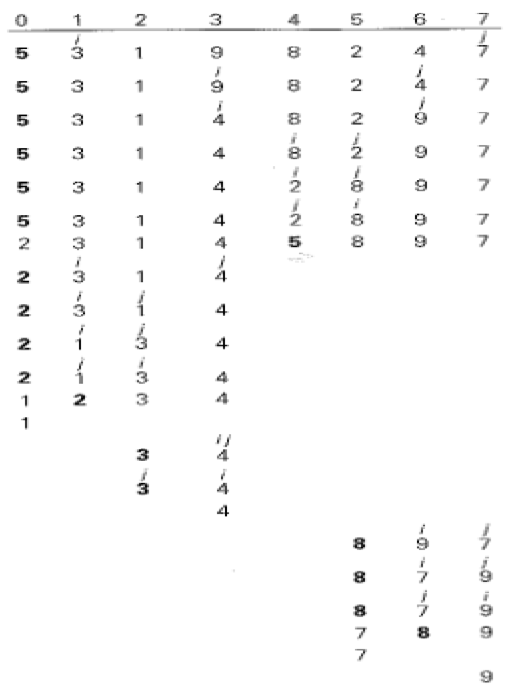
$$\underbrace{A[0]\dots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1]\dots A[n-1]}_{\text{all are } \ge A[s]}$$

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

```
//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, 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
    $Quicksort(A[l..s-1])$
    $Quicksort(A[s+1..r])$
```

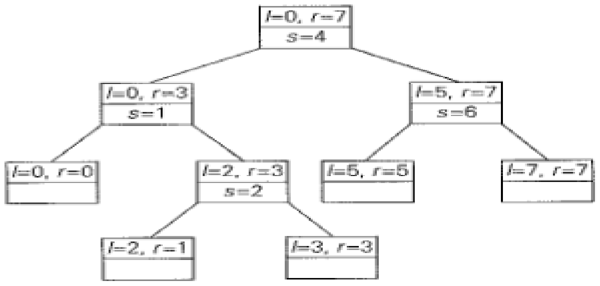
**ALGORITHM** *HoarePartition*$(A[l..r])$

```
    //Partitions a subarray by Hoare's algorithm, using the first element
    //        as a pivot
    //Input: Subarray of array $A[0..n-1]$, defined by its left and right
    //        indices $l$ and $r$ $(l < r)$
    //Output: 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 |
|---|---|---|---|---|---|---|---|
| **5** | $\overset{i}{3}$ | 1 | 9 | 8 | 2 | 4 | $\overset{j}{7}$ |
| **5** | 3 | 1 | $\overset{i}{9}$ | 8 | 2 | $\overset{j}{4}$ | 7 |
| **5** | 3 | 1 | $\overset{i}{4}$ | 8 | 2 | $\overset{j}{9}$ | 7 |
| **5** | 3 | 1 | 4 | $\overset{i}{8}$ | $\overset{j}{2}$ | 9 | 7 |
| **5** | 3 | 1 | 4 | $\overset{i}{2}$ | $\overset{j}{8}$ | 9 | 7 |
| **5** | 3 | 1 | 4 | $\overset{j}{2}$ | $\overset{i}{8}$ | 9 | 7 |
| 2 | 3 | 1 | 4 | **5** | 8 | 9 | 7 |
| **2** | $\overset{i}{3}$ | 1 | $\overset{j}{4}$ | | | | |
| **2** | $\overset{i}{3}$ | $\overset{j}{1}$ | 4 | | | | |
| **2** | $\overset{i}{1}$ | $\overset{j}{3}$ | 4 | | | | |
| **2** | $\overset{j}{1}$ | $\overset{i}{3}$ | 4 | | | | |
| 1 | **2** | 3 | 4 | | | | |
| 1 | | | | | | | |
| | | **3** | $\overset{ij}{4}$ | | | | |
| | | $\overset{j}{\mathbf{3}}$ | $\overset{i}{4}$ | | | | |
| | | | 4 | | | | |
| | | | | | **8** | $\overset{i}{9}$ | $\overset{j}{7}$ |
| | | | | | **8** | $\overset{i}{7}$ | $\overset{j}{9}$ |
| | | | | | **8** | $\overset{j}{7}$ | $\overset{i}{9}$ |
| | | | | | 7 | **8** | 9 |
| | | | | | 7 | | |
| | | | | | | | 9 |

(a)



(b)

$T(1) = 1$

n compar.
2 ← 2 part
Each – $n/2$

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

$$= 2\left[2T(n/4) + n/2\right] + n$$

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

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

$= 4\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n$

$= 8T\left(\frac{n}{8}\right) + n + 2n$

$= 8T\left(\frac{n}{8}\right) + 3n$

$= 2^3 T\left(\frac{n}{2^3}\right) + 3n$

$= 2^k T\left(\frac{n}{2^k}\right) + kn$

$= nT\left(\frac{n}{n}\right) + \log n \cdot (n)$ Let $2^k = n$

$=$ ~~$nT(1) + \log$~~ $n.$ $\log_2{}^k = \log n$

$= nT(1) + n\log n$ $k\log 2 = \log n.$

$= n + n\log n.$ $k = \log n$

$= \Theta(n\log n)$

Worst Case

- if A[0..n − 1] is a strictly increasing array and we use A[0] as the pivot,
  - the left-to-right scan will stop on A[1] while the right-to- left scan will go all the way to reach A[0], indicating the split at position 0
  - n + 1 comparisons required

Total comparisons

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Average Case

- Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n.
- A partition can happen in any position s (0 ≤ s ≤ n−1)
- n+1 comparisons are required for partition.
- After the partition, the left and right sub arrays will have s and n − 1− s elements, respectively.

Average Case

- Assuming that the partition split can happen in each position s with the same probability 1/n, we get

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$
$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

## Matrix Multiplication

**Direct Method:**

- Suppose we want to multiply two n x n matrices, A and B.
- Their product, C=AB, will be an n by n matrix and will therefore have $\mathbf{n^2}$ elements.
- The number of multiplications involved in producing the product in this way is $\Theta(n^3)$

$$C(i,j) = \sum_{1 \le k \le n} A(i,k)B(k,j)$$

## Matrix Multiplication

- Divide and Conquer method for Matrix multiplication

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

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

## Strassen's matrix multiplication

The principal insight of the algorithm
product C of two 2 × 2 matrices A and B
– with **just seven multiplications**

- This is accomplished by

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

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

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

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

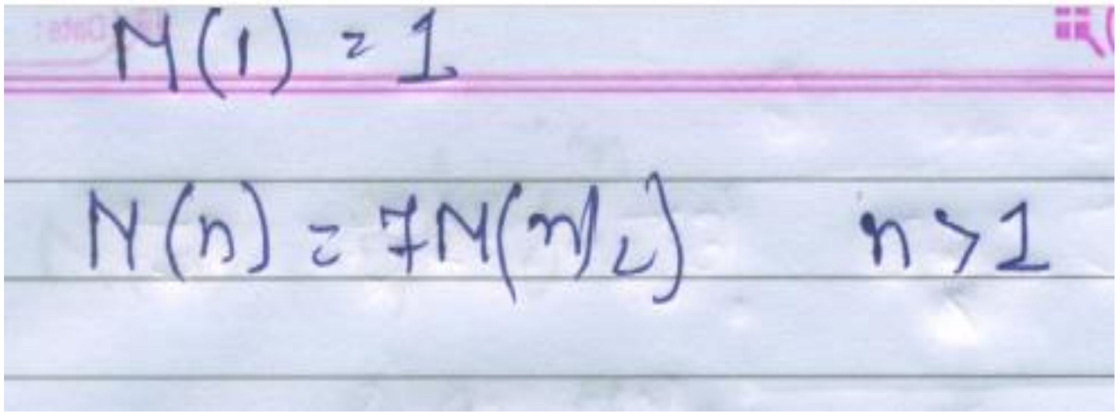
$$m_3 = a_{00} * (b_{01} - b_{11}),$$

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

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

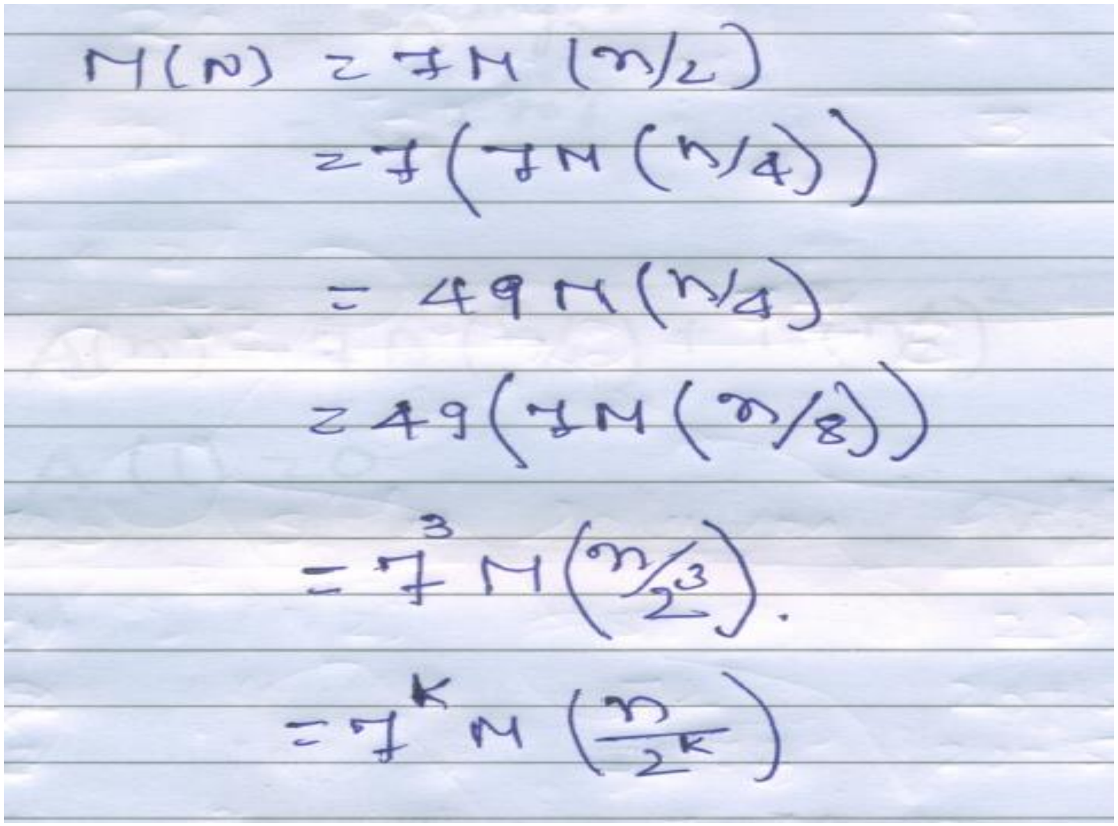
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$

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



$M(1) = 1$

$M(n) = 7M(n/2) \quad n > 1$



$M(n) = 7M(n/2)$

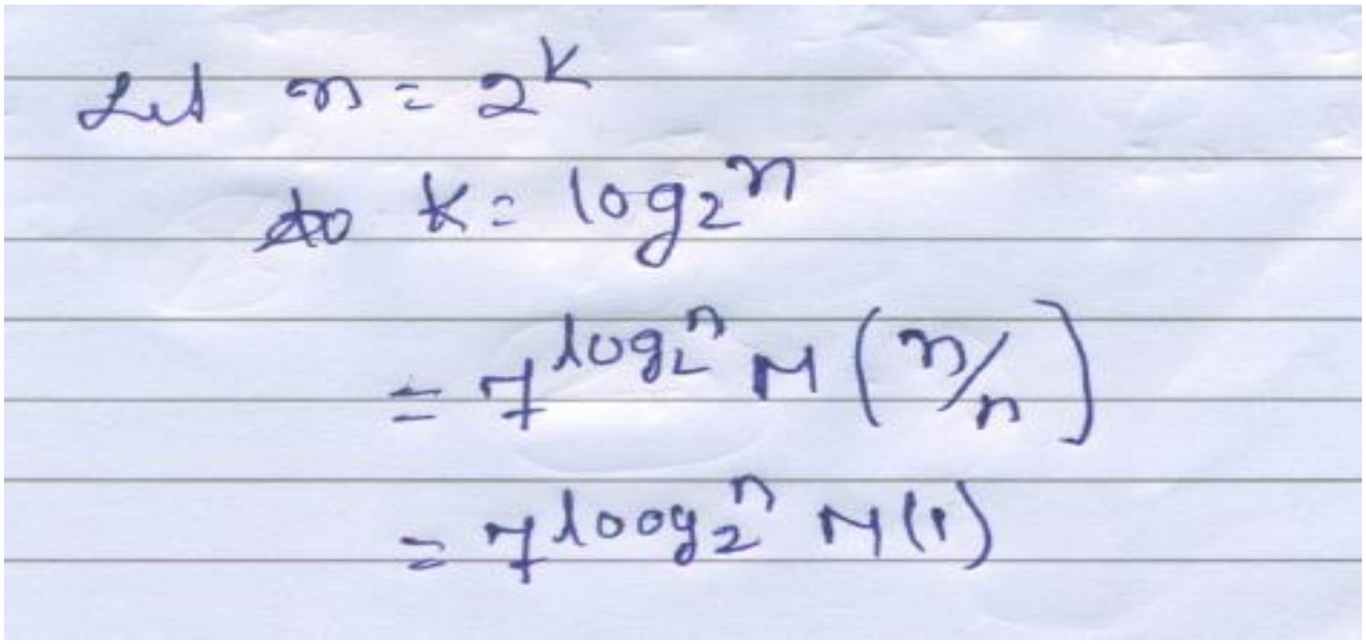
$= 7(7M(n/4))$

$= 49M(n/4)$

$= 49(7M(n/8))$

$= 7^3 M\left(n/2^3\right).$

$= 7^k M\left(\frac{n}{2^k}\right)$



Let $n = 2^k$

so $k = \log_2 n$

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

$= 7^{\mathrm{loog}_2 n} M(1)$

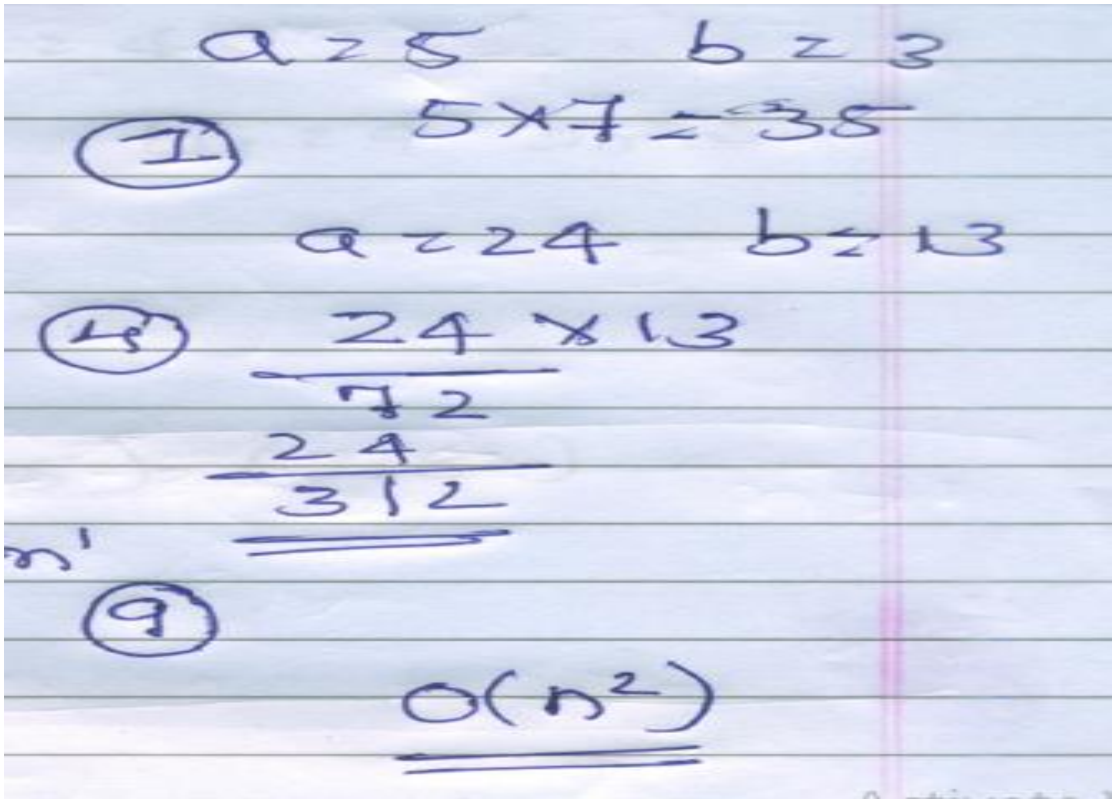
$$M(n) = 7^{\log_2 n}$$

$$= n^{\log_2 7}$$

$$= n^{2.807}$$

$$\boxed{\begin{array}{ll} A(n) = 7A(n/2) + 18(n/2)^2 & n > 1 \\ A(1) = 0 & \end{array}}$$

## Multiplication of large Integer



$a = 5 \quad b = 3$

① $5 \times 7 = 35$

$a = 24 \quad b = 13$

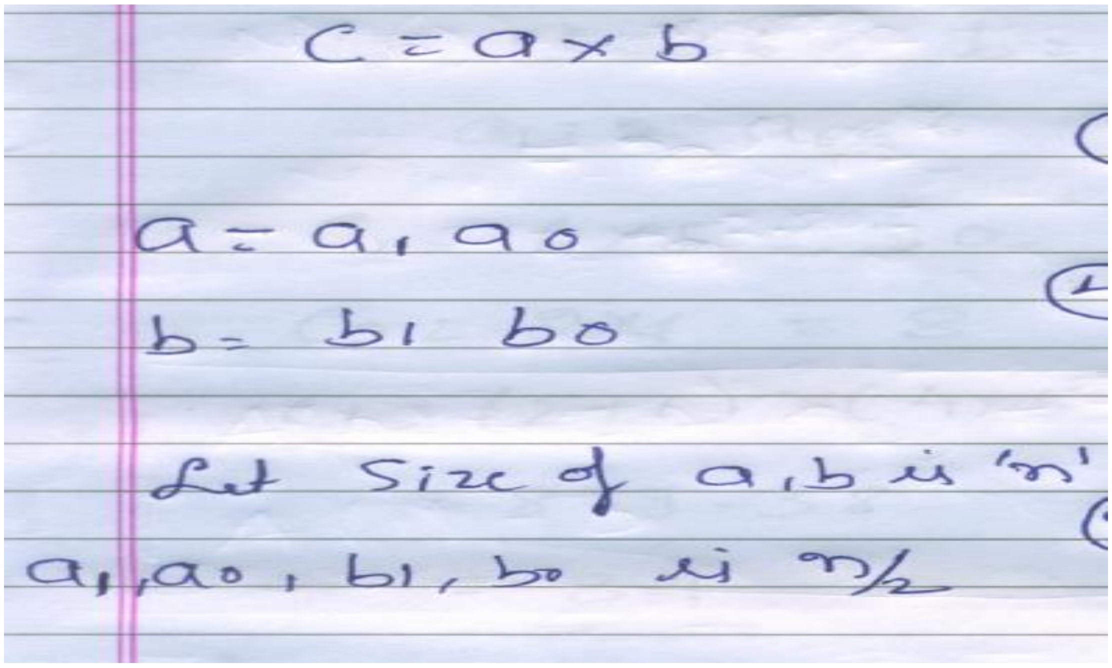
④ $24 \times 13$

```
 72
24
312
```

n'

⑨

$O(n^2)$



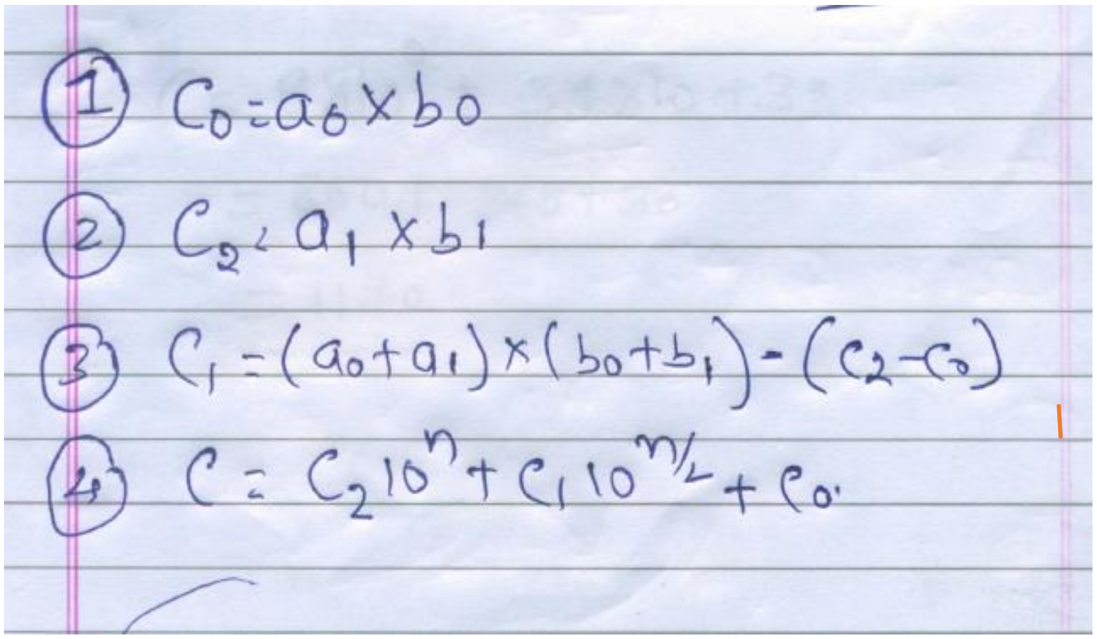
$C = a \times b$

$a = a_1 \, a_0$

$b = b_1 \, b_0$

Let size of a, b is 'n'

$a_1, a_0, b_1, b_0$ is $n/2$



1 $C_0 = a_0 \times b_0$

2 $C_2 = a_1 \times b_1$

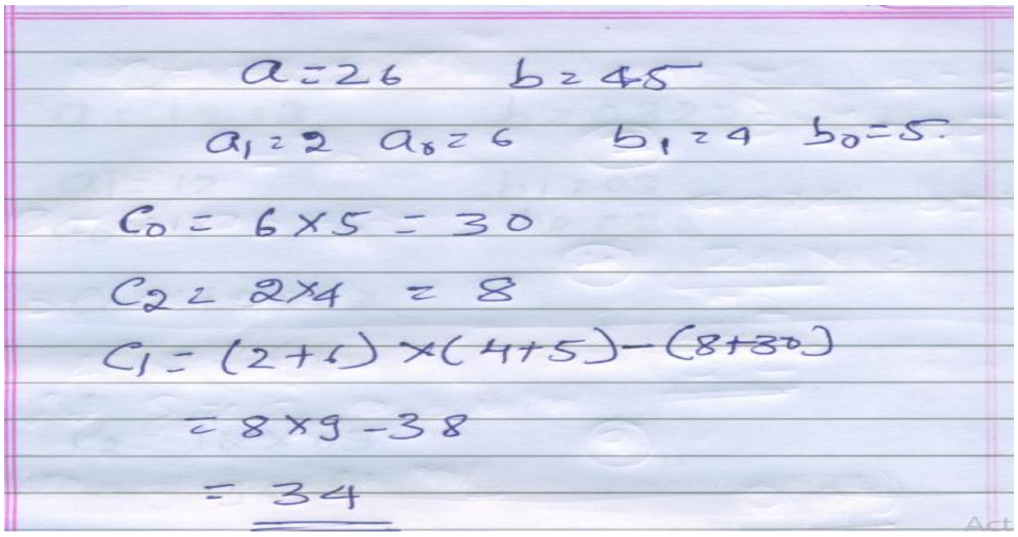
3 $C_1 = (a_0 + a_1) \times (b_0 + b_1) - (C_2 - C_0)$

4 $C = C_2 10^n + C_1 10^{n/2} + C_0$

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

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 halves and the sum of the $b$'s halves minus the sum of $c_2$ and $c_0$.



$a = 26 \quad b = 45$

$a_1 = 2 \quad a_0 = 6 \quad b_1 = 4 \quad b_0 = 5.$

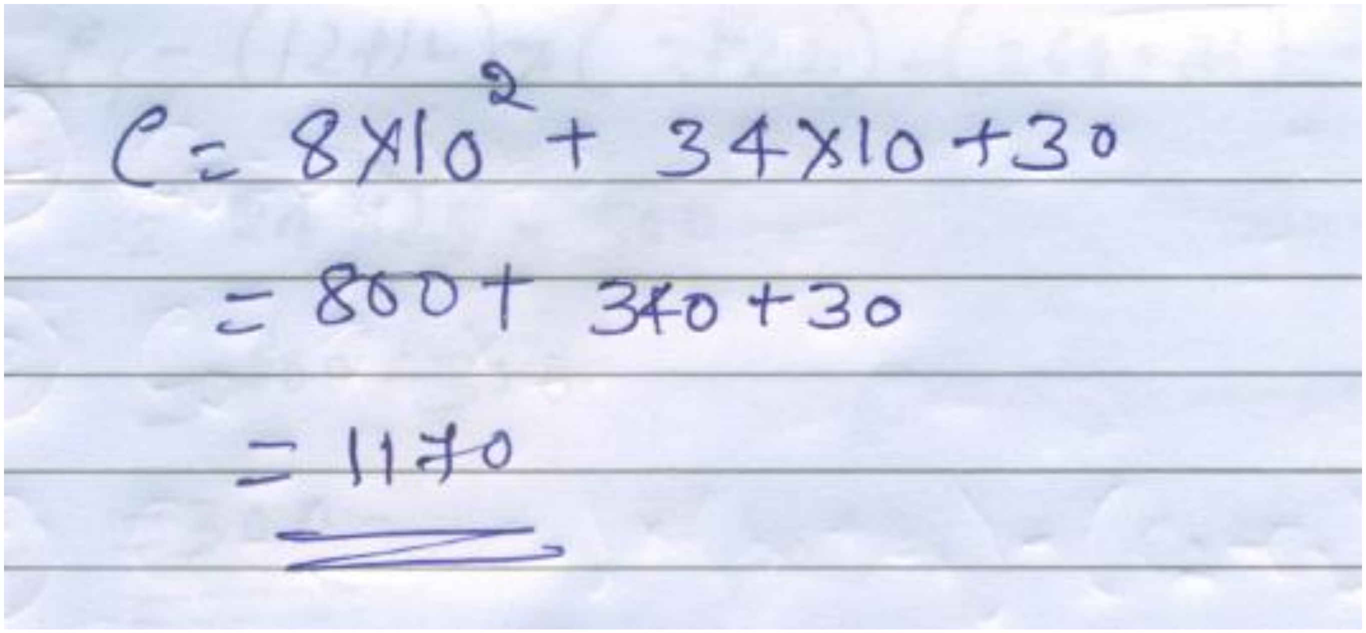
$C_0 = 6 \times 5 = 30$

$C_2 = 2 \times 4 = 8$

$C_1 = (2+6) \times (4+5) - (8+30)$

$= 8 \times 9 - 38$

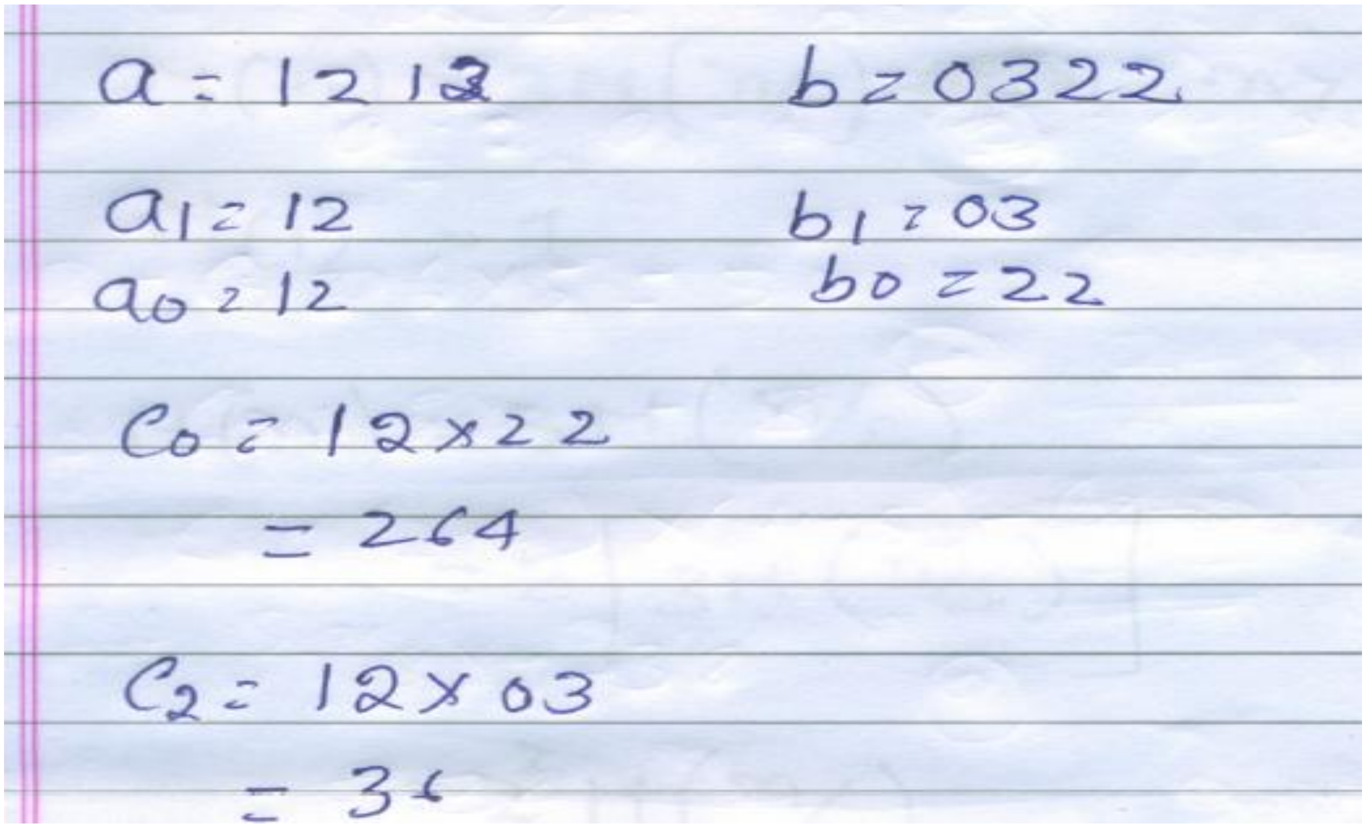
$= \underline{34}$



$C = 8 \times 10^2 + 34 \times 10 + 30$

$= 800 + 340 + 30$

$= 1170$



$a = 1212$ $\quad$ $b = 0322$

$a_1 = 12$ $\quad$ $b_1 = 03$

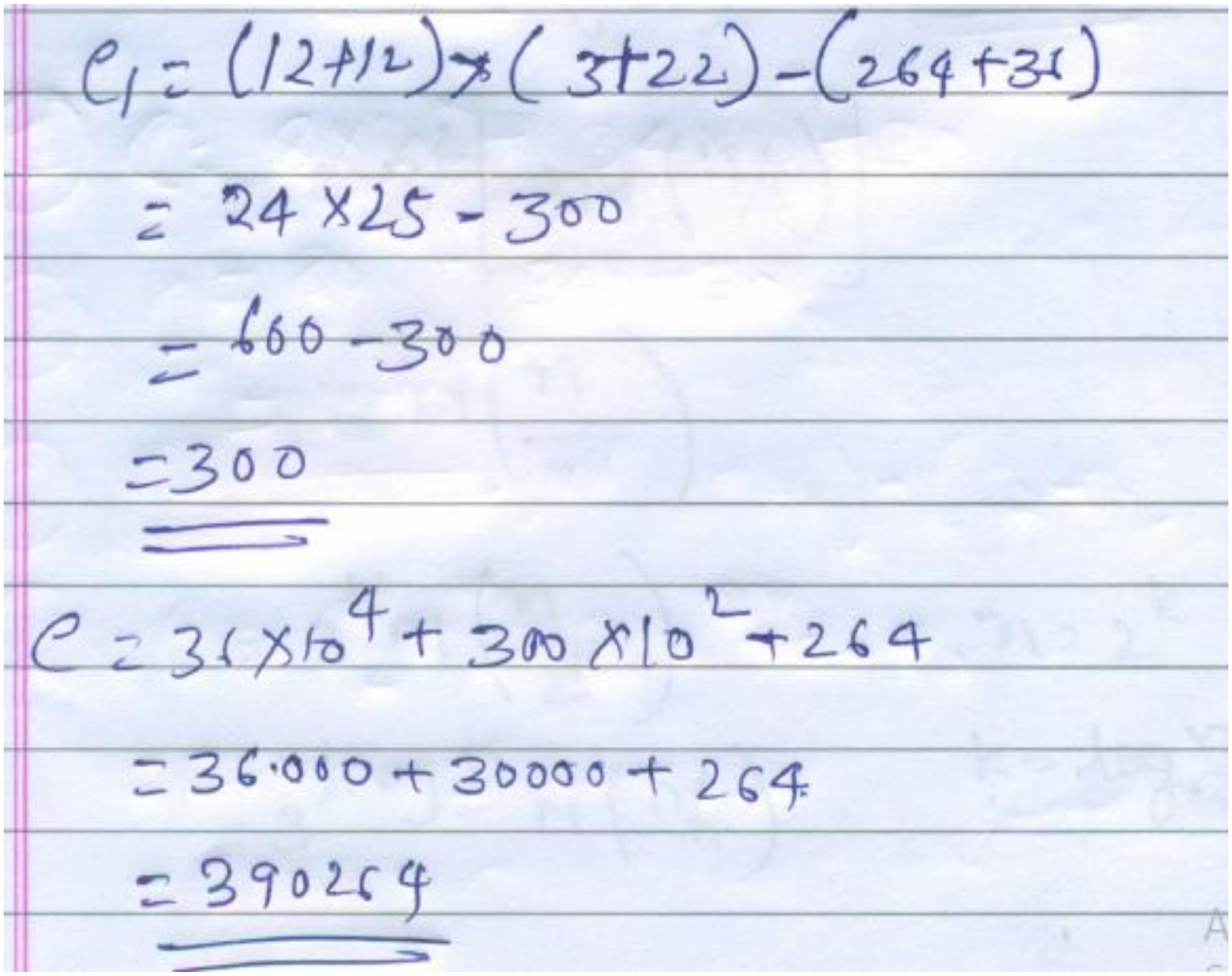
$a_0 = 12$ $\quad$ $b_0 = 22$

$C_0 = 12 \times 22$

$= 264$

$C_2 = 12 \times 03$

$= 36$



$c_1 = (12+12) * (3+22) - (264+31)$

$= 24 \times 25 - 300$

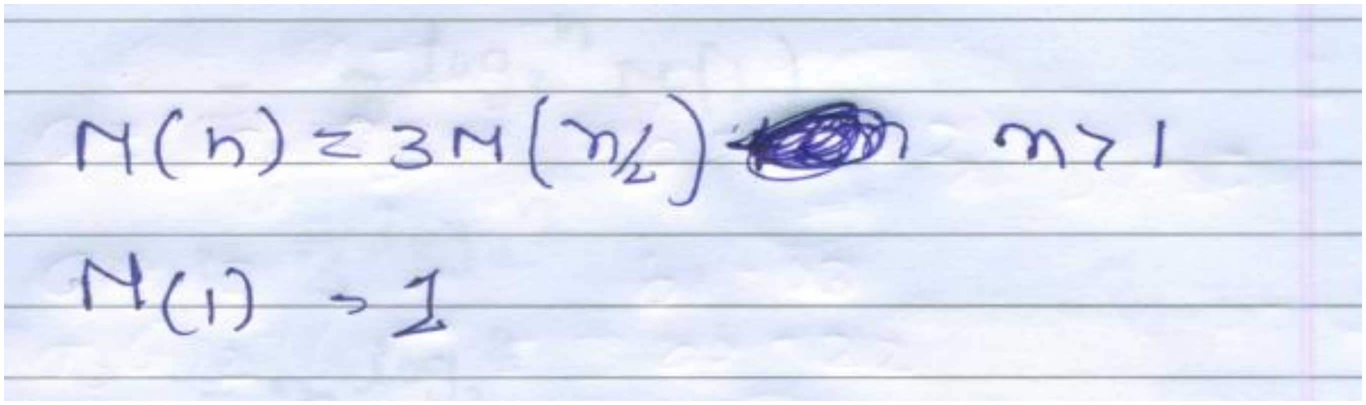
$= 600 - 300$

$= 300$

$c = 31 \times 10^4 + 300 \times 10^2 + 264$

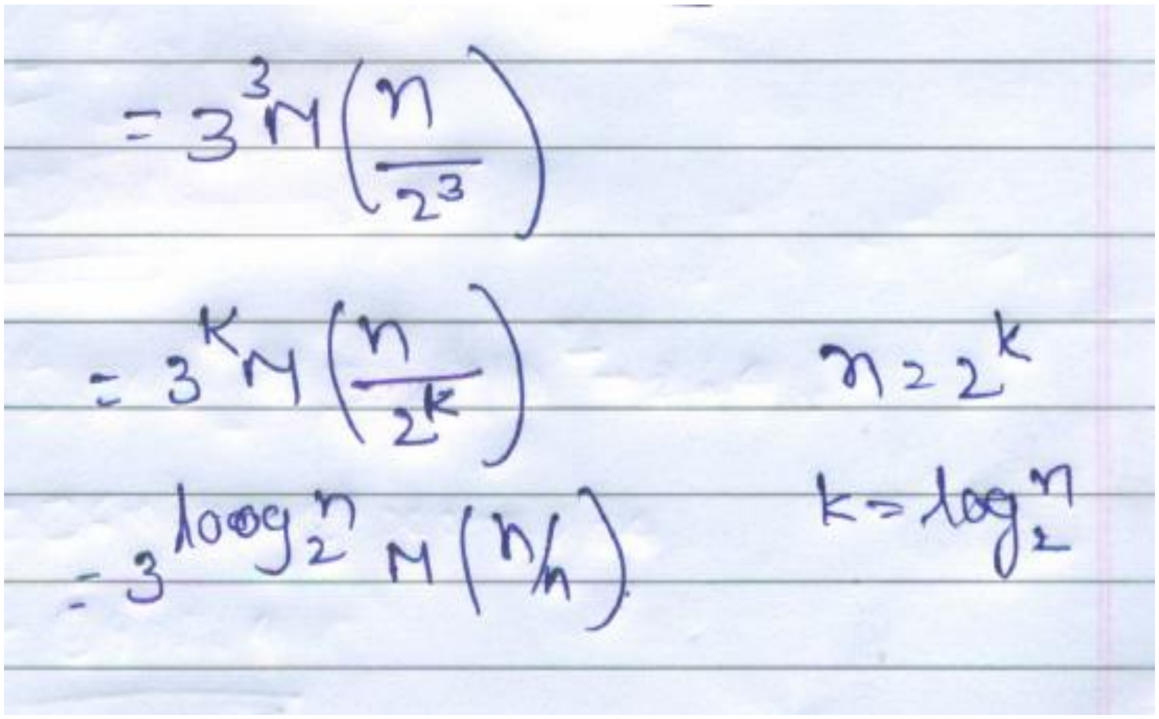
$= 36{\cdot}000 + 30000 + 264$

$= 390264$



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

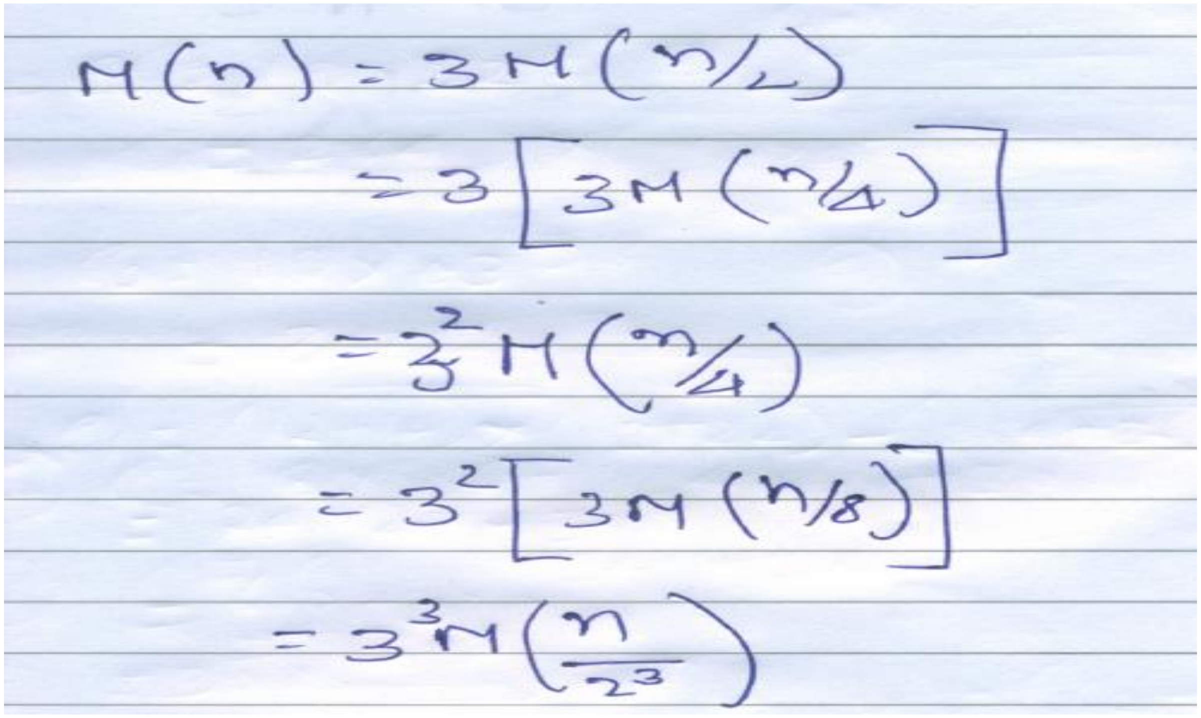
$M(1) = 1$



$= 3^3 M\left(\frac{n}{2^3}\right)$

$= 3^k M\left(\frac{n}{2^k}\right)$ $\quad n = 2^k$

$= 3^{\log_2 n} M(n/n)$ $\quad k = \log_2 n$



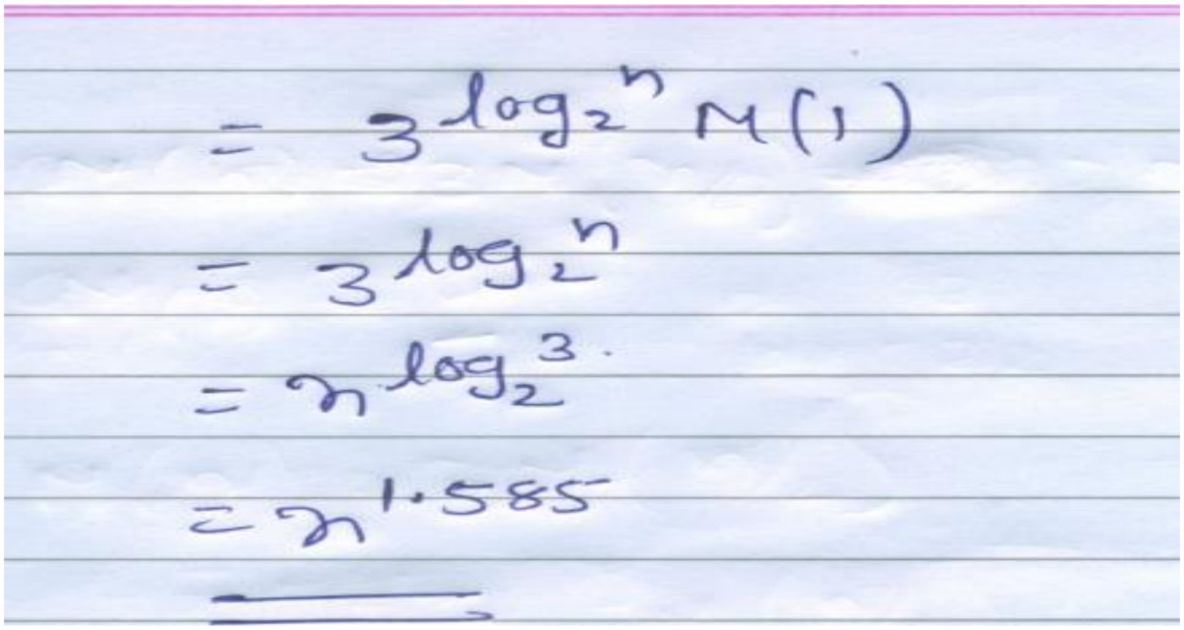
$M(n) = 3M(n/2)$

$= 3[3M(n/4)]$

$= 3^2 M(n/4)$

$= 3^2[3M(n/8)]$

$= 3^3 M\left(\frac{n}{2^3}\right)$



$= 3^{\log_2 n} M(1)$

$= 3^{\log_2 n}$

$= n^{\log_2 3}$

$= n^{1.585}$

## Binary Tree

**A binary tree T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees TL and TR called, respectively, the left and right subtree of the root.**

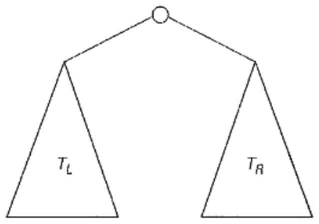
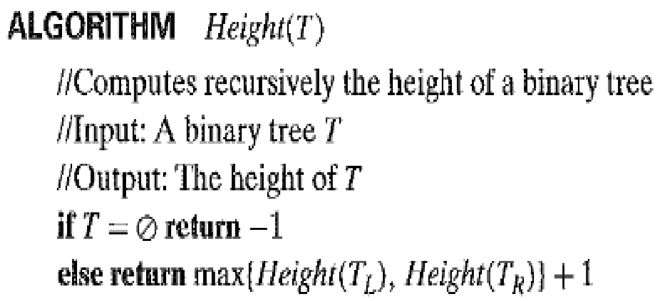


**FIGURE 4.4** Standard representation of a binary tree



```
ALGORITHM   Height(T)
    //Computes recursively the height of a binary tree
    //Input: A binary tree T
    //Output: The height of T
    if $T = \varnothing$ return $-1$
    else return $\max\{Height(T_L), Height(T_R)\} + 1$
```

We measure the problem's instance size by the number of nodes $n(T)$ in a given binary tree $T$. Obviously, the number of comparisons made to compute the maximum of two numbers and the number of additions $A(n(T))$ made by the algorithm are the same. We have the following recurrence relation for $A(n(T))$:

$$A(n(T)) = A(n(T_L)) + A(n(T_R)) + 1 \quad \text{for } n(T) > 0,$$
$$A(0) = 0.$$

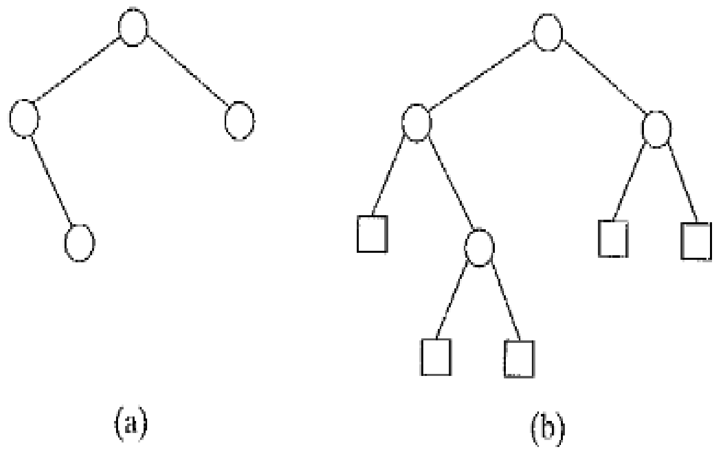


**FIGURE 4.5** (a) Binary tree. (b) Its extension. Internal nodes are shown as circles; external nodes are shown as squares.

The extra nodes (shown by little squares in Figure 4.5) are called external; the original nodes (shown by little circles) are called internal. By definition, the extension of the empty binary tree is a single external node. Checking Figure 4.5 and a few similar examples, it is easy to hypothesize that the number of external nodes x is always one more than the number of internal nodes n:

$$x = n + 1. \qquad \textbf{(4.5)}$$

To prove this formula, consider the total number of nodes, both internal and external. Since every node, except the root, is one of the two children of an internal node, we have the equation

$$2n + 1 = x + n,$$

which immediately implies equation (4.5).

Note that equation (4.5) also applies to any nonempty ***full binary tree***, in which, by definition, every node has either zero or two children: for a full binary tree, $n$ and $x$ denote the numbers of parental nodes and leaves, respectively.

Returning to algorithm *Height*, the number of comparisons to check whether the tree is empty is

$$C(n) = n + x = 2n + 1,$$

while the number of additions is

$$A(n) = n.$$

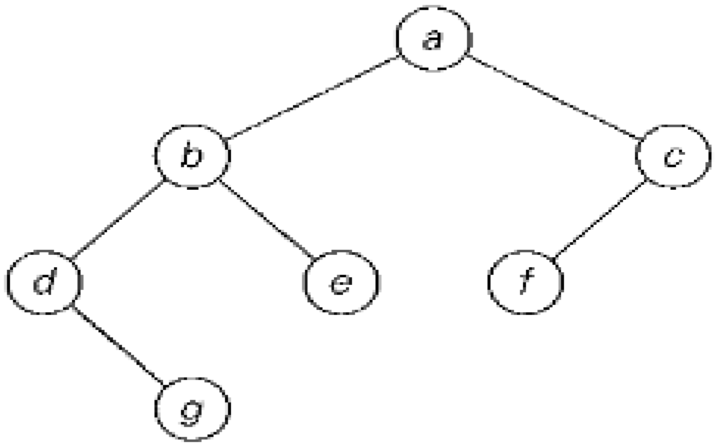
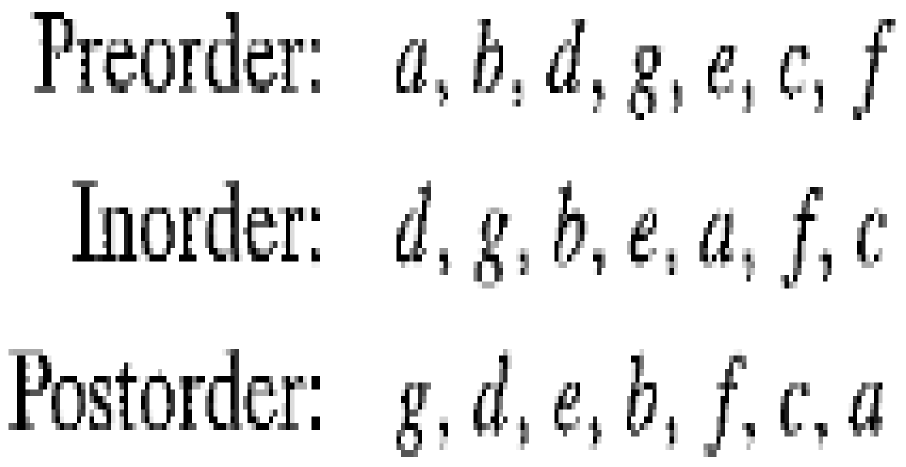


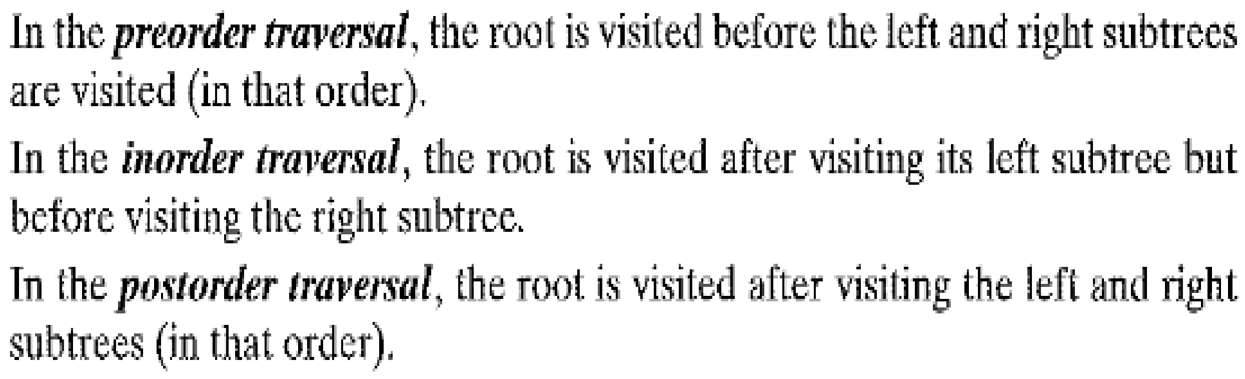
**FIGURE 4.6** Binary tree and its traversals



Preorder: $a, b, d, g, e, c, f$

Inorder: $d, g, b, e, a, f, c$

Postorder: $g, d, e, b, f, c, a$



In the ***preorder traversal***, the root is visited before the left and right subtrees are visited (in that order).

In the ***inorder traversal***, the root is visited after visiting its left subtree but before visiting the right subtree.

In the ***postorder traversal***, the root is visited after visiting the left and right subtrees (in that order).

## Decrease-and-Conquer

**The decrease-and-conquer technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem.**

### Decrease and Conquer Approach

- There are three major variations of decrease-and- conquer:

- decrease-by-a-constant, most often by one (e.g., insertion sort)
- decrease-by-a-constant-factor, most often by the factor of two (e.g., binary search)
- variable-size-decrease (e.g., Euclid's algorithm)

In the decrease-by-a-constant variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one

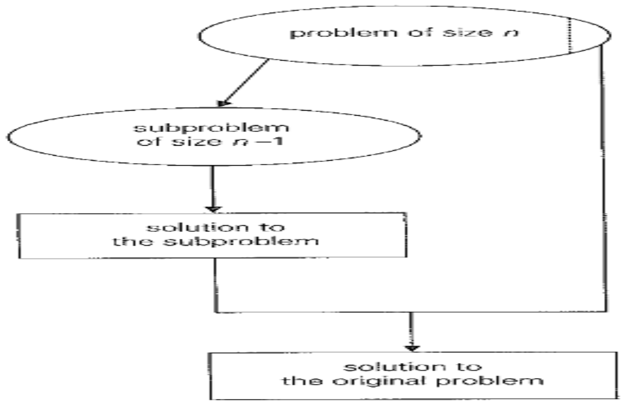


**FIGURE 5.1** Decrease (by one)-and-conquer technique

Consider, as an example, the exponentiation problem of computing $a^n$ for positive integer exponents. The relationship between a solution to an instance of size $n$ and an instance of size $n-1$ is obtained by the obvious formula: $a^n = a^{n-1} \cdot a$. So the function $f(n) = a^n$ can be computed either "top down" by using its recursive definition

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 1 \\ a & \text{if } n = 1 \end{cases} \qquad (5.1)$$

The decrease-by-a-constant-factor technique suggests reducing a problem's instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two

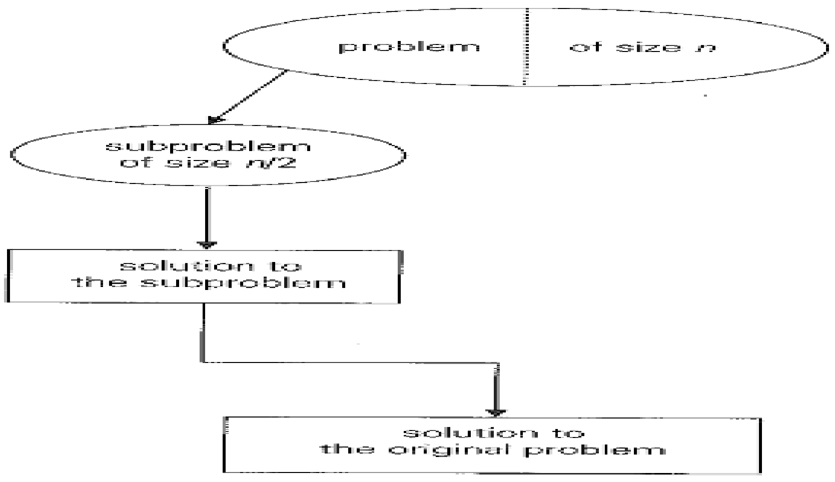


FIGURE 5.2 Decrease (by half)-and-conquer technique

For an example, let us revisit the exponentiation problem. If the instance of size $n$ is to compute $a^n$, the instance of half its size will be to compute $a^{n/2}$, with the obvious relationship between the two: $a^n = (a^{n/2})^2$. But since we consider here instances of the exponentiation problem with integer exponents only, the former does not work for odd $n$. If $n$ is odd, we have to compute $a^{n-1}$ by using the rule for even-valued exponents and then multiply the result by $a$. To summarize, we have the following formula:

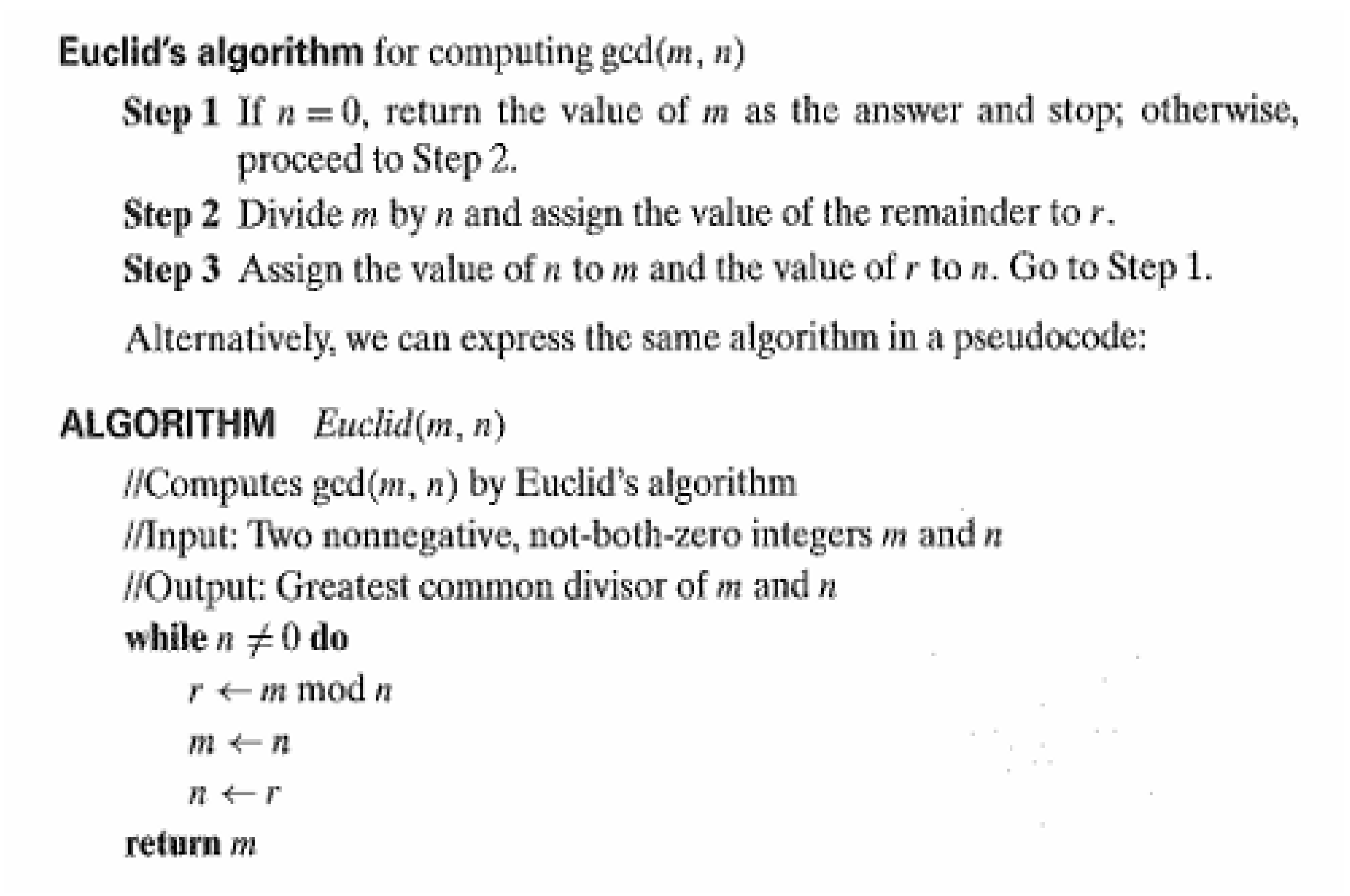
$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd and greater than 1} \\ a & \text{if } n = 1. \end{cases} \tag{5.2}$$

Finally, in the variable-size-decrease variety of decrease-and-conquer, a size reduction pattern varies from one iteration of an algorithm to another.
Euclid's algorithm for computing the greatest common divisor provides a good example of such a situation.

-

$$\gcd(m, n) = \gcd(n, m \bmod n).$$

Though the arguments on the right-hand side are always smaller than those on the left-hand side (at least starting with the second iteration of the algorithm), they are smaller neither by a constant nor by a constant factor. A few other examples of such algorithms appear in Section 5.6.



**Euclid's algorithm** for computing gcd($m$, $n$)

**Step 1** If $n = 0$, return the value of $m$ as the answer and stop; otherwise, proceed to Step 2.

**Step 2** Divide $m$ by $n$ and assign the value of the remainder to $r$.

**Step 3** Assign the value of $n$ to $m$ and the value of $r$ to $n$. Go to Step 1.

Alternatively, we can express the same algorithm in a pseudocode:

```
ALGORITHM  Euclid(m, n)
    //Computes gcd(m, n) by Euclid's algorithm
    //Input: Two nonnegative, not-both-zero integers m and n
    //Output: Greatest common divisor of m and n
    while n ≠ 0 do
        r ← m mod n
        m ← n
        n ← r
    return m
```

## Insertion sort

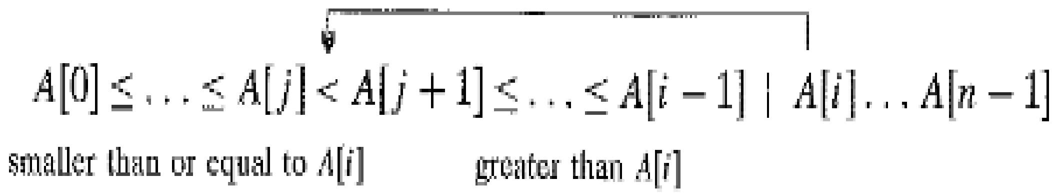
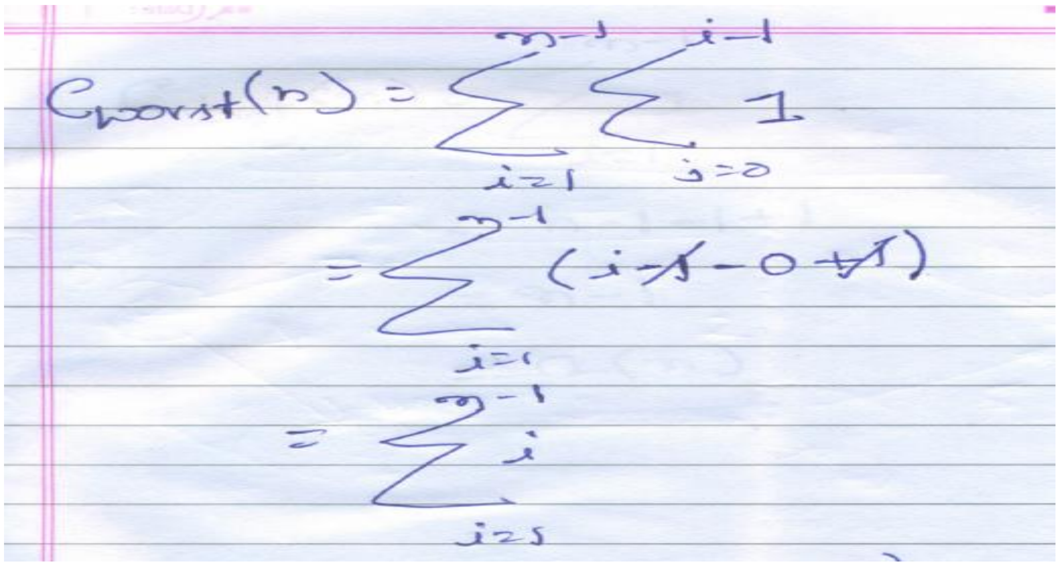

$A[0] \le \ldots \le A[j] < A[j+1] \le \ldots \le A[i-1] \mid A[i] \ldots A[n-1]$
smaller than or equal to $A[i]$
greater than $A[i]$

**FIGURE 5.3** Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

```
89 | 45   68   90   29   34   17
45   89 | 68   90   29   34   17
45   68   89 | 90   29   34   17
45   68   89   90 | 29   34   17
29   45   68   89   90 | 34   17
29   34   45   68   89   90 | 17
17   29   34   45   68   89   90
```

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

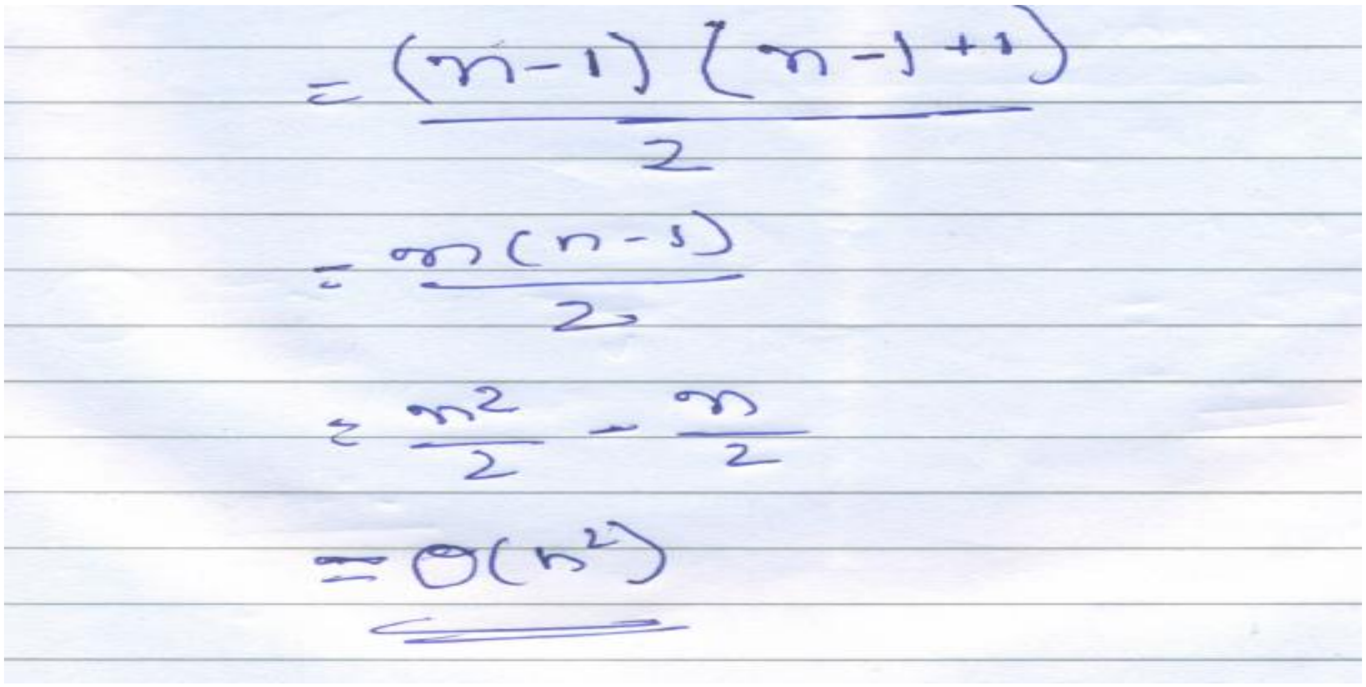
```
//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of $n$ orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ to $n-1$ do
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ and $A[j] > v$ do
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$
```



$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$= \sum_{i=1}^{n-1} (i - \cancel{1} - 0 + \cancel{1})$$

$$= \sum_{i=1}^{n-1} i$$

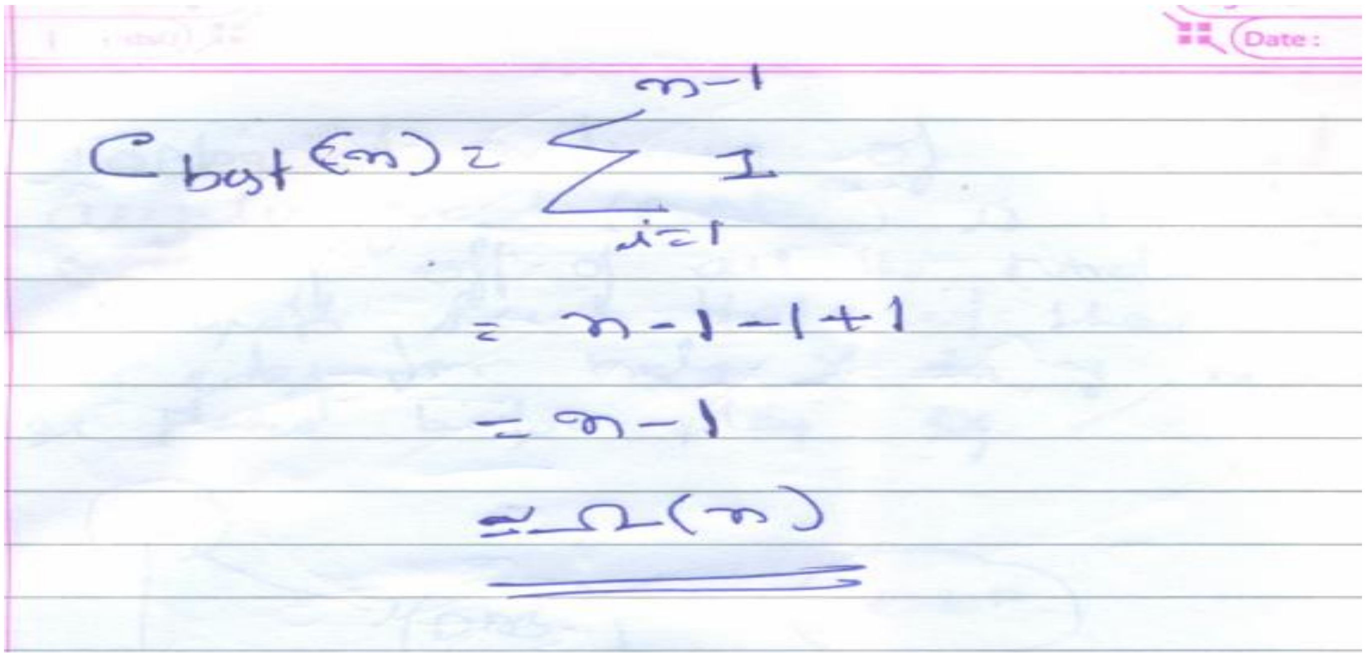


$$= \frac{(n-1)(n-1+1)}{2}$$

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

$$= \frac{n^2}{2} - \frac{n}{2}$$

$$= \Theta(n^2)$$



$$C_{best}(n) = \sum_{i=1}^{n-1} 1$$

$$= n-1-1+1$$

$$= n-1$$

$$\simeq \Omega(n)$$

## DFS AND BFS

### Depth-first search

It is also very useful to accompany a depth-first search traversal by constructng the so-called depth-first search forest.
The traversal's starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, it is attached as a child to the vertex from which it is being reached. Such an edge is called a tree edge because the set of all such edges forms a forest.

The algorithm may also encounter an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree).
Such an edge is called a back edge because it connects a vertex to its ancestor, other than the parent, in the depth-first search forest. Figure 5.5 provides an example of a depth-first search traversal, with the traversal's stack and corresponding depth- first search forest shown as well.

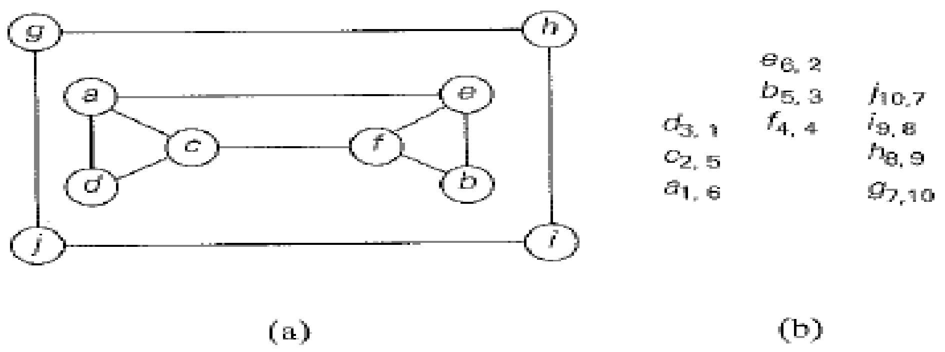
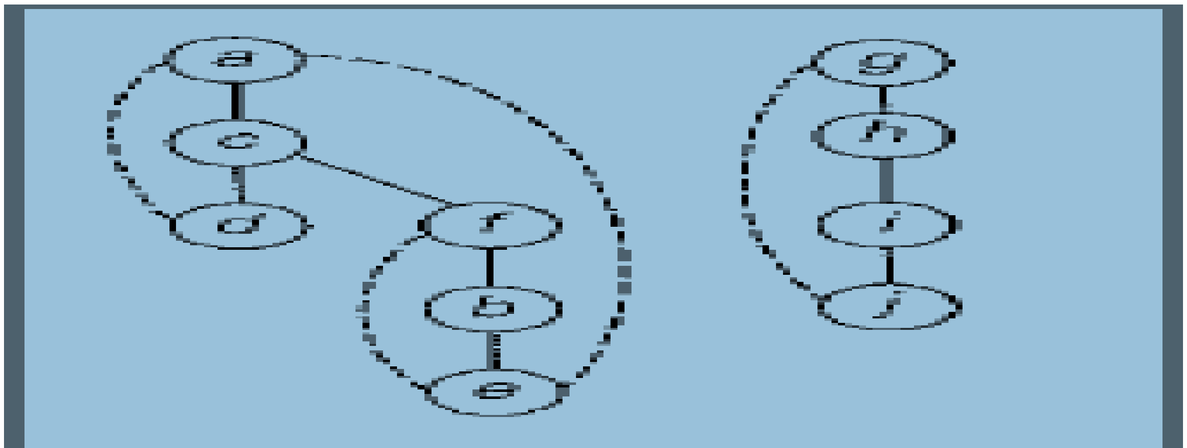


**FIGURE 5.5** Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex was visited, i.e., pushed onto the stack; the second one indicates the order in which it became a dead-end, i.e., popped off the stack). (c) DFS forest (with the tree edges shown with solid lines and the back edges shown with dashed lines).



(c)

**ALGORITHM** *DFS(G)*

```
//Implements a depth-first search traversal of a given graph
//Input: Graph $G = \langle V, E \rangle$
//Output: Graph $G$ with its vertices marked with consecutive integers
//in the order they've been first encountered by the DFS traversal

mark each vertex in $V$ with 0 as a mark of being "unvisited"
$count \leftarrow 0$
for each vertex $v$ in $V$ do
     if $v$ is marked with 0
        $dfs(v)$

$dfs(v)$
//visits recursively all the unvisited vertices connected to vertex $v$ by a path
//and numbers them in the order they are encountered
//via global variable $count$
$count \leftarrow count + 1$; mark $v$ with $count$
for each vertex $w$ in $V$ adjacent to $v$ do
     if $w$ is marked with 0
        $dfs(w)$
```

## Breadth-first search

Similarly to a DFS traversal, it is useful to accompany a BFS traversal by con-structing the so-called breadth-first search forest.

The traversal's starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called a tree edge.

If an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree) is encountered, the edge is noted as a cross edge.

Figure 5.6 provides an example of a breadth-first search traversal, with the traversal's queue and corresponding breadth-first search forest shown.

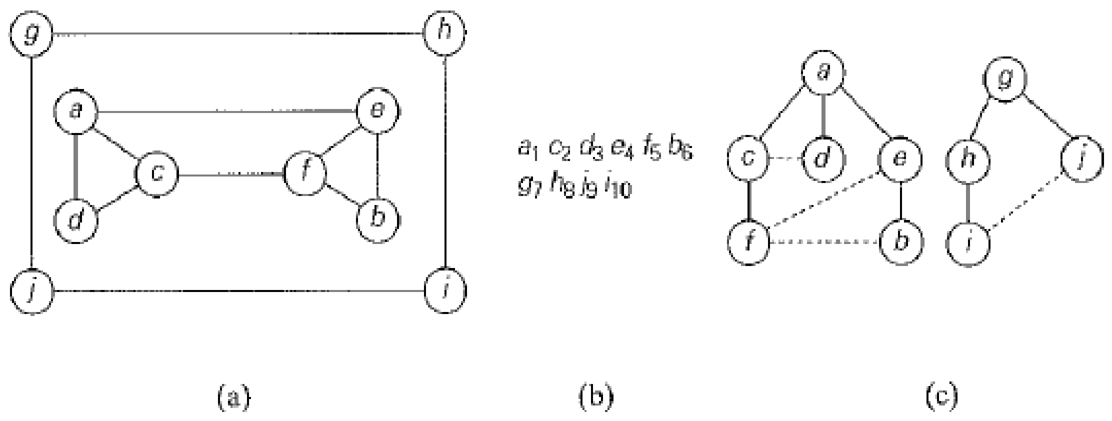

g
h
a
e
c
f
d
b
j
i
$a_1$ $c_2$ $d_3$ $e_4$ $f_5$ $b_6$
$g_7$ $h_8$ $j_9$ $i_{10}$
a
g
c
d
e
h
j
f
b
i
(a)
(b)
(c)

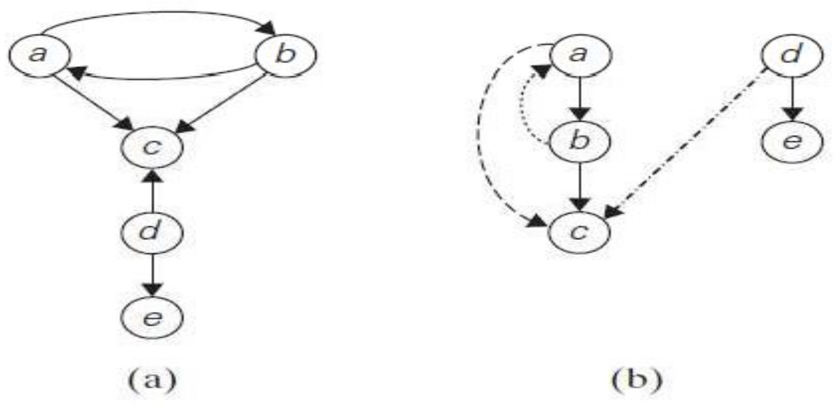
**FIGURE 5.6** Example of a BFS traversal. (a) Graph. (b) Traversal's queue, with the numbers indicating the order in which the vertices were visited, i.e., added to (or removed from) the queue. (c) BFS forest (with the tree edges shown with solid lines and the cross edges shown with dotted lines).

```
ALGORITHM  $BFS(G)$
    //Implements a breadth-first search traversal of a given graph
    //Input: Graph $G = \langle V, E \rangle$
    //Output: Graph $G$ with its vertices marked with consecutive integers
    //in the order they have been visited by the BFS traversal

mark each vertex in $V$ with 0 as a mark of being "unvisited"
$count \leftarrow 0$
for each vertex $v$ in $V$ do
    if $v$ is marked with 0
        $bfs(v)$

$bfs(v)$
//visits all the unvisited vertices connected to vertex $v$ by a path
//and assigns them the numbers in the order they are visited
//via global variable $count$
$count \leftarrow count + 1$;   mark $v$ with $count$ and initialize a queue with $v$
while the queue is not empty do
    for each vertex $w$ in $V$ adjacent to the front vertex do
        if $w$ is marked with 0
            $count \leftarrow count + 1$;   mark $w$ with $count$
            add $w$ to the queue
    remove the front vertex from the queue
```

**TABLE 5.1** Main facts about depth-first search (DFS) and breadth-first search (BFS)

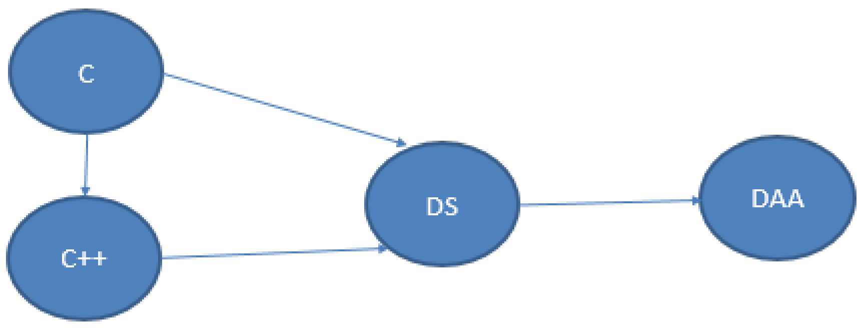
| | DFS | BFS |
|---|---|---|
| Data structure | stack | queue |
| No. of vertex orderings | 2 orderings | 1 ordering |
| Edge types (undirected graphs) | tree and back edges | tree and cross edges |
| Applications | connectivity, acyclicity, articulation points | connectivity, acyclicity, minimum-edge paths |
| Efficiency for adjacent matrix | $\Theta(\|V^2\|)$ | $\Theta(\|V^2\|)$ |
| Efficiency for adjacent lists | $\Theta(\|V\| + \|E\|)$ | $\Theta(\|V\| + \|E\|)$ |

## Topological Sorting



(a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at $a$.

**Topological ordering of direct acyclic graph (DAG) is the linear ordering of all the nodes in the graph such that if there is a edge for node X to Y then X should be placed before place Y**



## Digraph

- A ***directed cycle*** in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex
- For example, *a, b, a* is a directed cycle in the digraph in Figure given above.
- Conversely, if a DFS forest of a digraph has no back edges, the digraph is a ***dag***, an acronym for ***directed acyclic graph***.

## Topological Sort

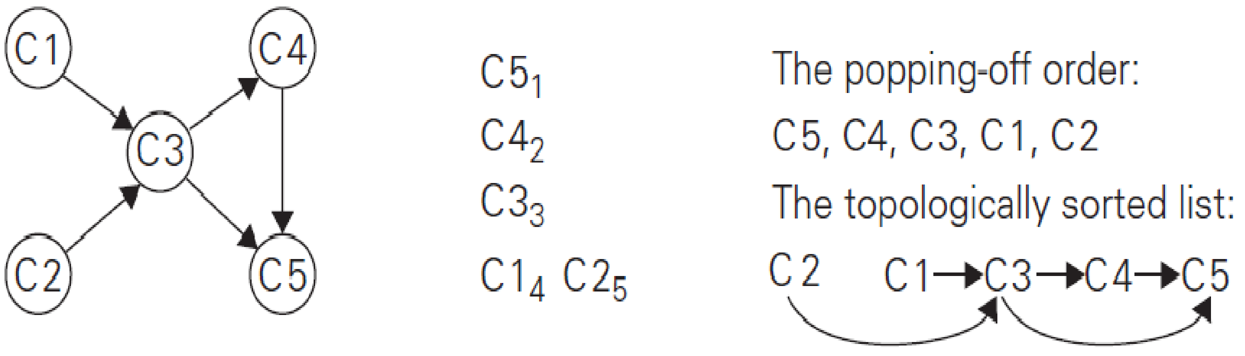
- For topological sorting to be possible, a digraph in question must be a DAG.
- There are two efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem.
  - The first one is based on depth-first search
  - The second is based on a direct application of the decrease-by-one technique.

## Topological Sorting based on DFS

**Method**

1. Perform a DFS traversal
2. Note the order in which vertices become dead-ends
3. Reversing this order yields a solution to the topological sorting problem, provided, no back edge has been encountered during the traversal.

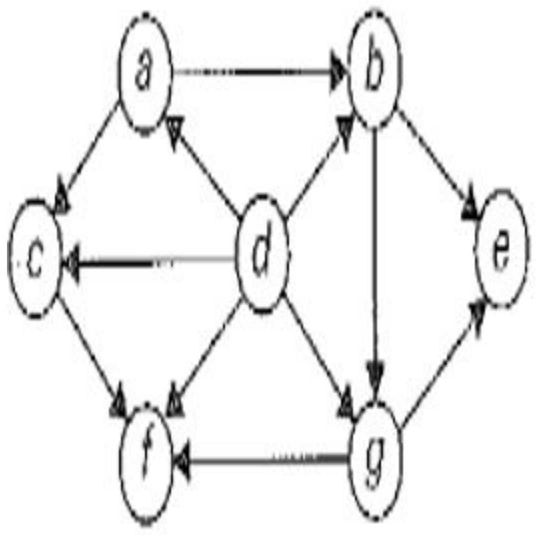
If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.



$C5_1$
$C4_2$
$C3_3$
$C1_4$ $C2_5$

The popping-off order:
C5, C4, C3, C1, C2
The topologically sorted list:
C2 C1→C3→C4→C5

(a) (b) (c)



(a)

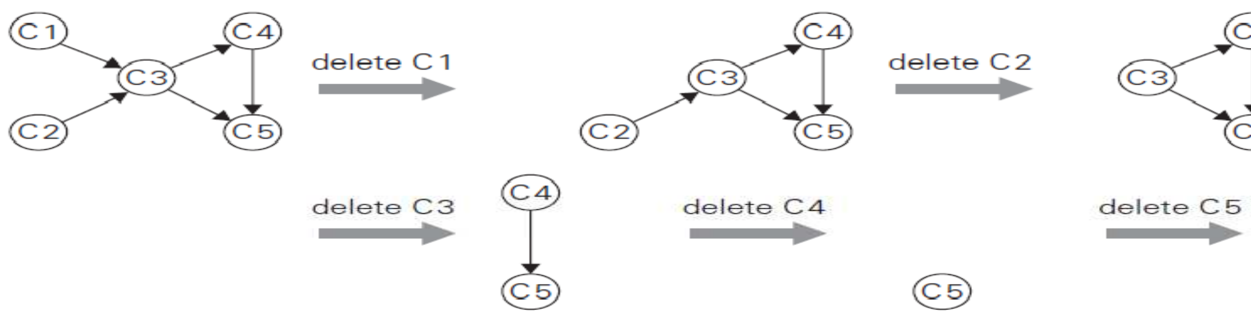
| VERTEX | DFS ORDER | Popping -OFF ORDER |
|---|---|---|
| a | 1 | 6 |
| b | 2 | 4 |
| e | 3 | 1 |
| g | 4 | 3 |
| f | 5 | 2 |
| c | 6 | 5 |
| d | 7 | 7 |

Pooing off order- e-> f-> g-> b-> c-> a-> d
Topological Order- d-> a-> c ->b-> g-> f ->e

## Topological Sorting using source-removal or decrease-and-conquer technique:

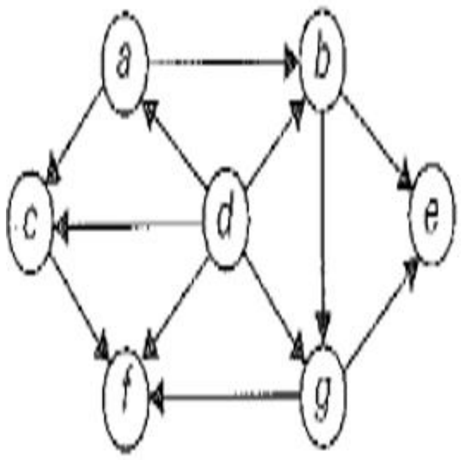
**Method:** The algorithm is based on a direct implementation of the decrease-(by one)-and-conquer technique:

1. Note down the in-degree of all the nodes.
2. Place the nodes which has in-degree zero
3. Decrement the in-degree of those nodes where there was an incoming edge from the nodes placed in step 2(Deleting the Vertex)
4. Repeat the Step2 and 3 until all nodes all placed
5. The order in which the vertices are deleted yields a solution to the topological sorting problem.



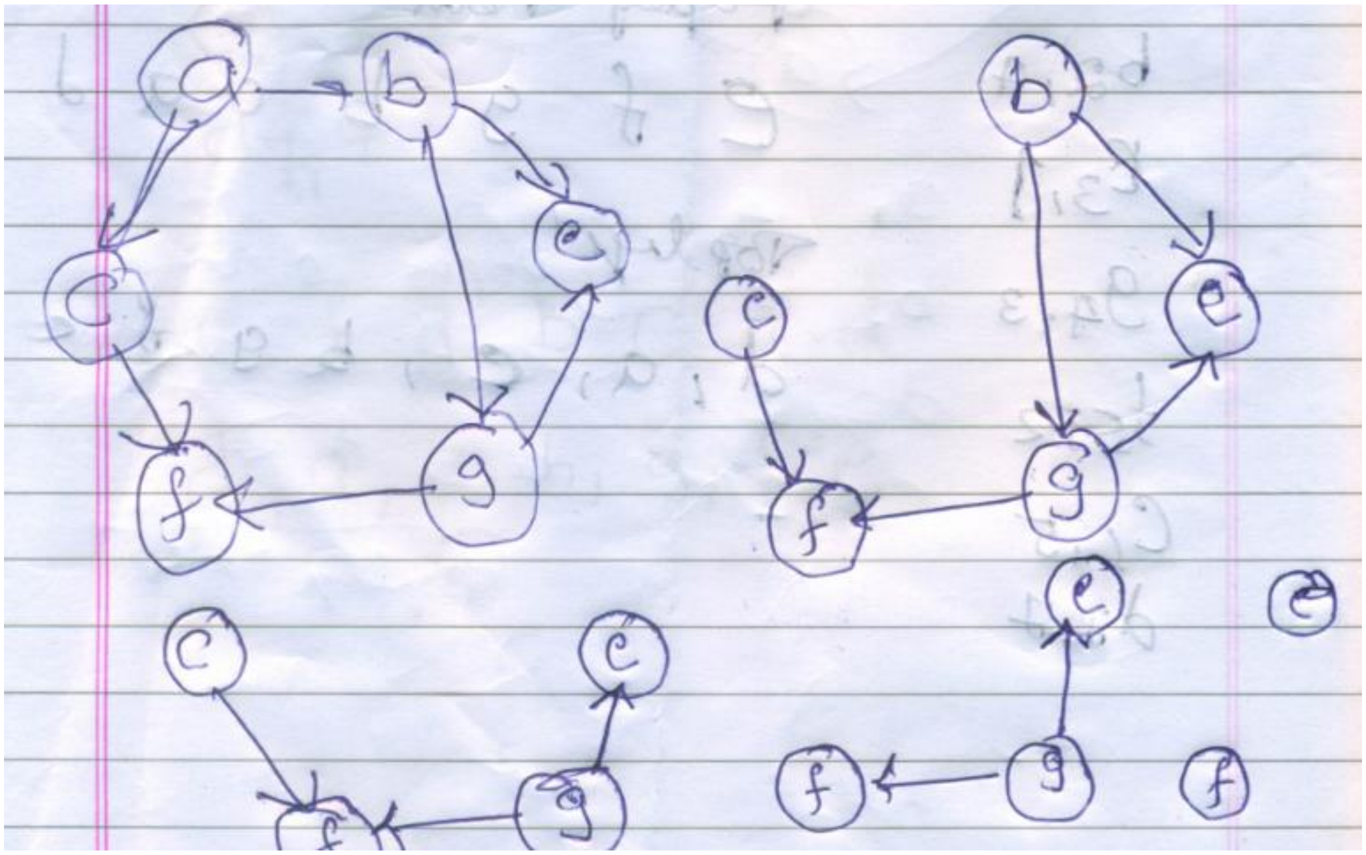
The solution obtained is C1, C2, C3, C4, C5

| In(C1) | 0 | | | | |
|---|---|---|---|---|---|
| In(C2) | 0 | 0 | | | |
| In(C3) | 2 | 1 | 0 | | |
| In(C4) | 1 | 1 | 1 | 0 | |
| In(C5) | 2 | 2 | 2 | 1 | 0 |



(a)

Topological Order- d ->a-> b-> c-> g-> e ->f



| In(a) | 1 | 0 | | | | | |
|---|---|---|---|---|---|---|---|
| In(b) | 2 | 1 | 0 | - | - | - | - |
| In(c) | 2 | 1 | 0 | 0 | - | - | - |
| In(d) | 0 | - | - | - | - | - | - |
| In(e) | 2 | 2 | 2 | 1 | 1 | 0 | - |
| In(f) | 3 | 2 | 2 | 2 | 1 | 0 | 0 |
| In(g) | 2 | 1 | 1 | 0 | 0 | - | - |

Topological Order- d ->a-> b-> c-> g-> e ->f

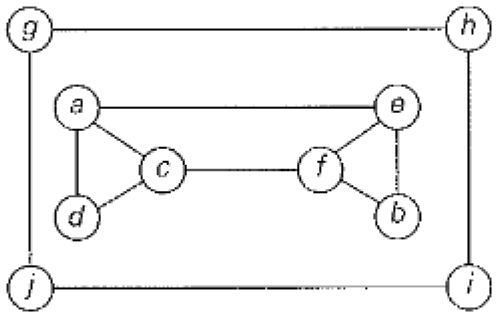
2 marks Questions

1. Define decrease and conquer technique and list any two of its variations.
2. What is a decrease by a constant technique and give an example.
3. What is a decrease by a constant factor technique and give an example.
4. What is a variable size decrease technique and give an example.

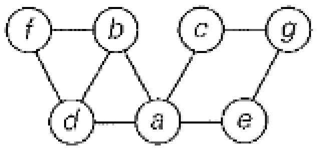
5. Write the time complexity for worst, best and average case of Insertion sort.
6. Give any four comparisons among Depth First Search and Breadth First Search.
7. Define digraph and dag.
8. Write an algorithm to find height of Binary tree.
9. Write the recurrence relation for Binary tree.
10. Define Topological sorting. Give an example.
11. Define following
    a. Tree Edge
    b. Cross Edge
    c. Back Edge
12. List the Two algorithms for solving topological sorting problem.
13. Define decrease-by-one technique in topological sorting.
14. Define divide and conquer technique and list its steps.
15. Give the structure of Recurrence equation for Divide and Conquer.
16. Write the time complexity for Merge and Quick sort
17. Write the time complexity for Strassen'sMatrix Multiplication and Multiplication Larger integer.
18. State Mater theorem of Divide and conquer.
19. Write the time complexity for worst, best and average case of Binary Search.
20. Write the steps followed in divide and Conquer approach.

Long Answer Questions **(THREE, FOUR OR FIVE Marks Questions)**

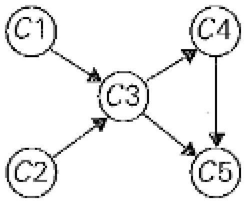
1. Write an algorithm to sort N numbers using Insertion sort. Derive the time complexity for worst case and best case.
2. Define decrease and conquer technique and explain its variations.
3. Write and explain Depth-First Search Algorithm with example
4. Write and explain Breadth-First Search Algorithm with example
5. Consider the following graph and perform following traversal and also draw the TREE.
    a. BFS
    b. DFS



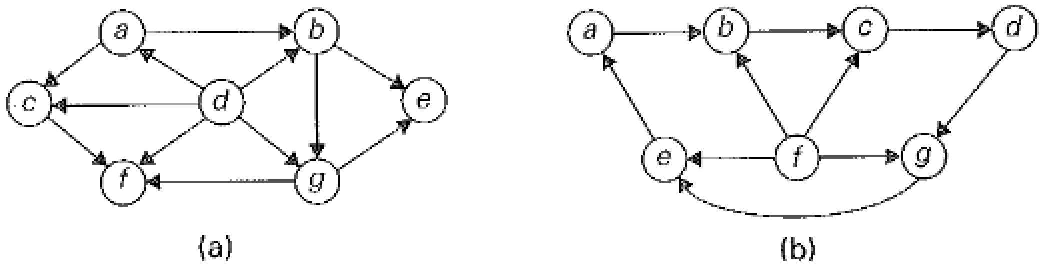
6. Consider the following graph and perform following traversal also draw the TREE.
    a. BFS
    b. DFS



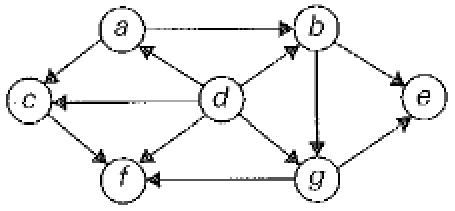
7. Give any five comparisons among Depth First Search and Breadth First Search.
8. Write a note on topological sorting with an example.
9. Explain DFS-based algorithm to solve the topological sorting problem and also write topological order for below graph.



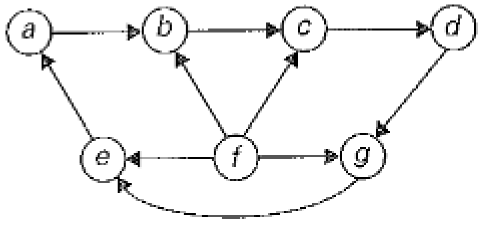
10. Apply the DFS-based algorithm to solve the topological sorting problem for the following digraphs



11. Apply the source-removal (Decrease by one) algorithm to solve the topological sorting problem for the following digraphs

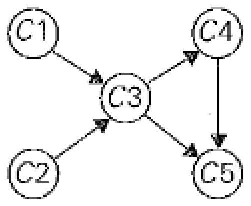


(a)



(b)

12. Explain source-removal (Decrease by one) algorithm to solve the topological sorting problem and also write topological order for below graph



13. Explain divide and conquer technique of solving problem with diagram.
14. Solve the recursion relation using Master theorems

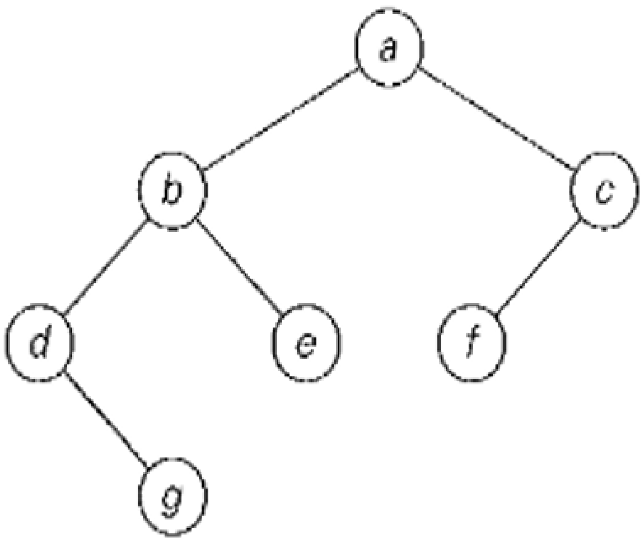
$$T(n) = 2T(n/2) + n$$

T(1)=2

15. Solve the recursion relation using Master theorems.

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

16. Write an algorithm to sort N numbers using Merge sort. Derive the time complexity.
17. Write an algorithm to sort N numbers using Quick sort. Derive the time complexity.
18. Write an algorithm to find maximum and minimum element in array Derive the time complexity.
19. Write an algorithm to search an element in an array of N numbers using Binary search. Derive the time complexity.
20. Traverse the Binary Tree a)Inorder b)Preorder c)Postorder



21. Explain the Strassen's algorithm of matrix multiplication and derive the time complexity.
22. ApplyStrassen's algorithm to multiply two Matrixes.

$$\begin{bmatrix} 3 & -1 \\ 2 & 4 \end{bmatrix} \times \begin{bmatrix} 2 & 3 \\ -1 & 4 \end{bmatrix}$$

23. Compute 1234 x 2526 usingdivide and conquer approach for the multiplication of two large numbers.
24. Explain the Multiplication two larger integer usingdivide and conquer approach and derive the time complexity.
25. Compute 34 x 26 using divide and conquer approach for the multiplication of two large numbers.
26. Apply Strassen's algorithm to compute exiting the recursion when n = 2, i.e., computing the products of 2-by-2 matrices by the brute-force algorithm.

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