DESIGN AND ANALYSIS OF ALGORITHMS

Course Code: 19CS4DCDAA

Module 2

DIVIDE-AND-CONQUER: Introduction, Master theorem, Quick sort, Mergesort, Multiplication of Large Integers and Strassen's Matrix Multiplication. The maximum subarray problem**.**

DECREASE-AND-CONQUER: Representation of Graphs, Insertion Sort, Depth-First Search and Breadth-First Search, Topological Sorting

Divide and Conquer:

Definition: Divide & conquer is a general algorithm design strategy with a general plan as follows:

1. DIVIDE:

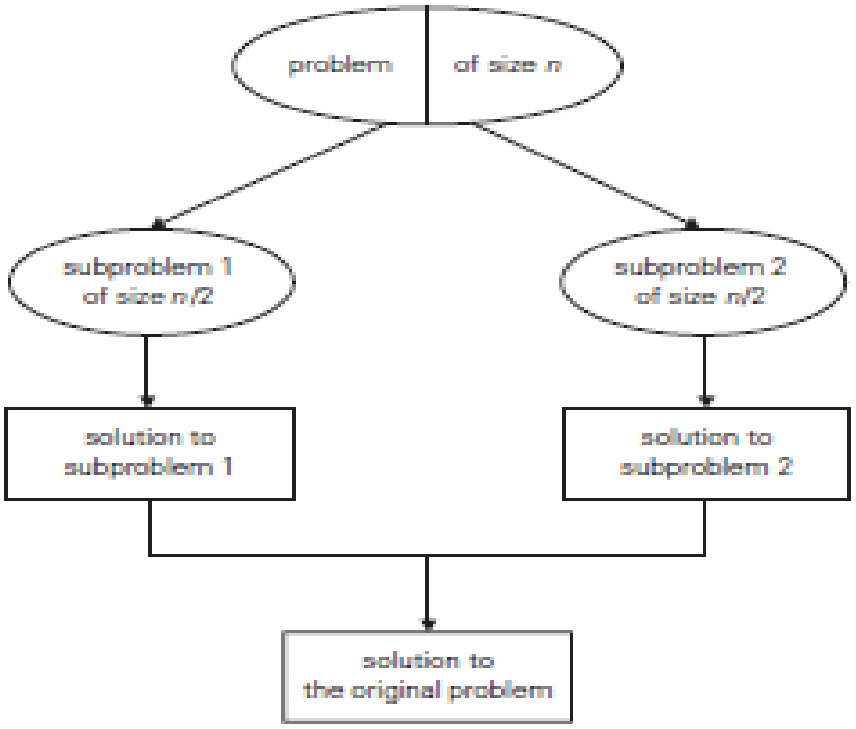
A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.

2. RECUR:

Solve the sub-problem recursively.

3. CONQUER:

If necessary, the solutions obtained for the smaller instances are combined to get a solution to the original instance.



IGURE 5.1 Divide-and-conquer technique (typical case).

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.
The recurrence for the running time T(n) is as follows:

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

where:

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

Therefore, the order of growth of T(n) depends on the values of the constants a & b and the order of growth of the function f(n).

Master theorem

Theorem: If $f(n) \in \Theta(n^d)$ with $d \geq 0$ in recurrence equation
$T(n) = a*T(n/b) + f(n)$, then

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

Example: Let T(n) = 2T(n/2) + 1, solve using master theorem.
Solution:
Here: a = 2
b = 2
f(n) = Θ(1)
d = 0
Therefore:
$a > b^d$ i.e., $2 > 2^0$
Case 3 of master theorem holds good.
Therefore: $T(n) \in \Theta(n^{\log a})$
$\in \Theta(n^{\log 2})$
$\in \Theta(n)$

Quick Sort

Quicksort is the other important sorting algorithm that is based on the divide-and conquer approach. A partition is an arrangement of the array's elements so that all the elements to the left of some element $A[s]$ are less than or equal to $A[s]$, and all the elements to the right of $A[s]$ are greater than or equal to it:

$$A[0] \ldots A[s-1] \quad A[s] \quad A[s+1] \ldots A[n-1]$$
$$\text{all are} \le A[s] \qquad \text{all are} \ge A[s]$$

Obviously, after a partition is achieved, $A[s]$ will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of $A[s]$ independently.

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 *Partition(A[l..r])*
//Partitions a subarray by using the first element

// as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices *low* and *high*
//Output: Partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[low]$
$i \leftarrow low+1$; $j \leftarrow high$
while (1)
while(i<high && *p≥a[i])
i++;
while(j>=low && *p< a[j])
j -- ;
if(i<j)
swap*(A*[*i*], *A*[*j*]*)*
else
swap*(A*[*low*], *A*[*j*]*)*
return *j*

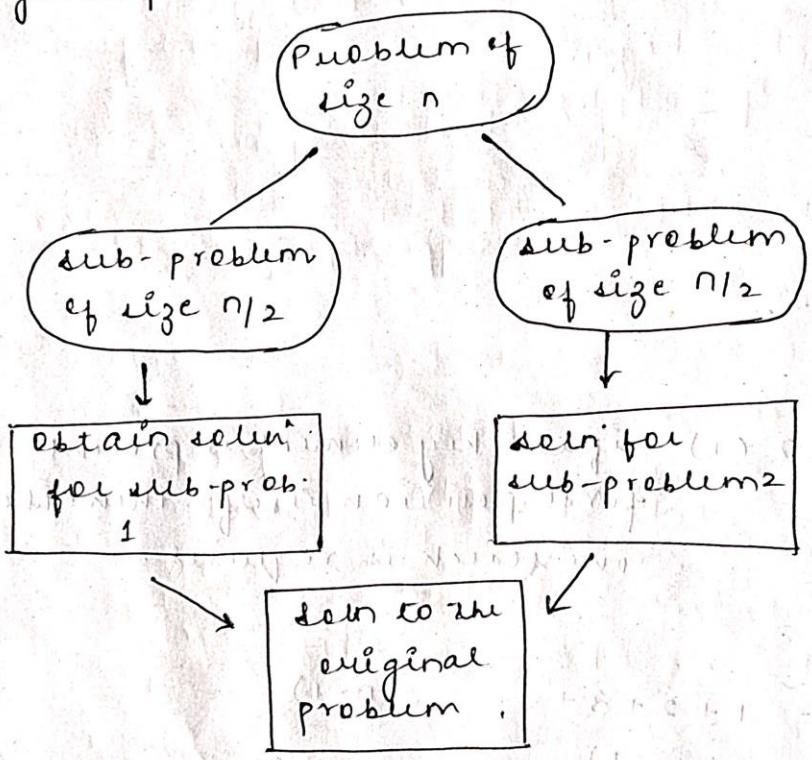
Demonstration Link:
https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/visualize/

8 Feb, '18

MODULE 2 :

★ Divide and conquer :

- Its a top-down technique for designing algorithms which consist of dividing the problem into smaller sub problems of same size. and solⁿ. are obtained for sub problems and combined to get a solⁿ. for original problem.



NOTE :

- In typical case a problem of size 'n' is divided into two instances of size $n/2$. Generally, an instance of size 'n' can be divided into 'b' instances of size 'n/b' with 'a' of them needing to be solved.
- To get the running time $t(n) = a * T(n/b) + f(n)$ — (1)

where, $a \geq 1$; $b > 1$

- $f(n)$ is a fn. that accounts for time spend on dividing the problem and combining solns.

Master's theorem:

- solving recurrence relation of above type (1)
- If $f(n)$ belongs to $\theta(n^d)$

$$f(n) \in \theta(n^d).$$

$$d \geq 0.$$

- In case of eq. (1) (recurrence relation):

$$T(n) = \begin{cases} \theta(n^d) & ; \text{if } a < b^d \\ \theta(n^d \log_b n) & ; \text{if } a = b^d \\ \theta(n^{\log_b a}) & ; \text{if } a > b^d \end{cases}$$

Quick sort: LAB

```
#include <stdio.h>
#include <conio.h>
#include <time.h>
#include <stdlib.h>
void quicksort(int a[], int low, int high)
{
  int j;
  if (low < high)
  {
    j = partition(a, low, high);
    Quicksort(a, low, j-1);
    Quicksort(a, j+1, high);
  }
}
int partition(int a[], int low, int high)
{
  int i, j, key, temp;
  key = a[low];
```

```
    i = low+1;
    j = high;
while(1)
{
    while((i<high) && (key >= a[i]))
      i++;
    while((j >= low) && (key < a[j]))
      j--;
    if (i<j)
    {
      temp=a[i];
      a[i] = a[j];
      a[j] = temp;
    }
    else
    {
      temp = a[low];
      a[low] = a[j];
      a[j] = temp;
      return j;
    }
}
}
void main()
{
  int a[100], i, n;
  float start, end;
  clrscr();
  printf("Enter the no. of ele\n");
  sf("%d", &n);
```

```
pf("Enter array elements \n");
for(i=0; i<n; i++)
    a[i] = random(999);
pf("The elements are:");
start = clock();
delay(1000);
quicksort(a, 0, n-1);
end = clock();
pf("The sorted elements are:");
for(i=0; i<n; i++)
    pf("%d", &a[i]);
pf("The time taken by prog. is %f \n",
        diff time(end, start)/CLK_TCK);
getch();
}
```

09/2/18.

ANALYSIS OF QUICK SORT:

1) Best case time efficiency:

If key element is present exactly at the centre dividing the array into two equal parts then

$$T(n) = \begin{cases} 0 & ; \quad n = 1 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n & ; \quad n > 1 \end{cases}$$

- $T\left(\frac{n}{2}\right)$ → to sort left sub array
- $T\left(\frac{n}{2}\right)$ → to sort right sub array
- n → to find pivot element & partition 'n' elements.

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

Generally; $a = 2 \quad b = 2 \quad d = 1$

$\therefore a = b^d$ (on comparing with Master's theorem)

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

$$= n \log_2 n$$

$$\boxed{T(n) = n \log_2 n} \rightarrow \text{Best case.}$$

2. Worst case time efficiency :-

Worst case occurs when at each invocation of fn°. the current is partitioned into 2 sub arrays with one being empty. This occurs if all the elements are arranged in ascending or descending order.

$$T(n) = \begin{cases} 0 & ; n = 1 \\ T(0) + T(n-1) + n & ; n > 1 \end{cases}$$

$T(n) = T(n-1) + n \Rightarrow$ Recurrence relation.

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

$T(n-2) = T(n-3) + n-2.$

$$T(n) = T(n-i) + (n) + (n-1) + (n-2) + (n-3)$$

$$\boxed{O(n) = n^2}$$

3. Average case:

Merge Sort

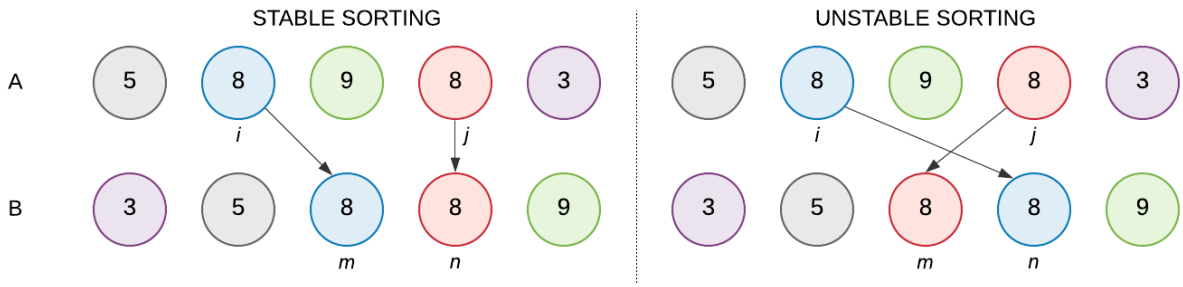
Definition:

Merge sort is a sort algorithm that splits the items to be sorted into two groups; recursively sorts each group, and merges them into a final sorted sequence.

Features:

- Is a comparison based algorithm
- Is a stable algorithm
- Is a perfect example of divide & conquer algorithm design strategy
- It was invented by John Von Neumann

Stable sorting maintains the position of two equals elements relative to one another.



ALGORITHM Merge sort (A[0… n-1])

//sorts array A by recursive merge sort

//i/p: array A

//o/p: sorted array A in ascending order

```
MergeSort(A, l, h):
    if l < h
   mid = (l+h)/2
    mergeSort(A, l, mid)
    mergeSort(A, mid+1, h)
    merge(A, l, mid, h)
```

ALGORITHM Merge (a[ ], l,mid,h)

```
 i=low
j=mid+1
k=low
c[20];
while(i<=mid && j<=high) do

if(a[i]<a[j])
c[k++]<- a[i++]
else
c[k++]<- a[j++]
end while

while(i<=mid)
c[k++]<- a[i++]
while(j<=high)
c[k++]<- a[j++]

for i=low to  high
a[i]<- c[i];
end for
```

https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/visualize/

Example:

Apply merge sort for the following list of elements: 6, 3, 7, 8, 2, 4, 5, 1

i=0	1	2	3	4	5	6	7
6	3	7	8	2	4	5	1

Mid= 3

I=0	1	2	3
6	3	7	8

4	5	6	7
2	4	5	1

Mid= 1

Mid=

I=0	1
6	3

I=2	3
7	8

4	5
2	4

6	7
5	1

I=0
6

I=1
3

I=2
7

I=3
8

4
2

5
4

6
5

7
1

I=0	1
3	6

i

I=2	3
7	8

j

4	5
2	4

6	7
1	5

I=0	1	2	3
3	6	7	8

4	5	6	7
1	2	4	5

i=0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

Analysis:

- Input size: Array size, n
- Basic operation: key comparison
- Best, worst, average case exists:

Worst case: During key comparison, neither of the two arrays becomes empty before the other one contains just one element.

- Let C(n) denotes the number of times basic operation is executed. Then

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

$C(1) = 0$

where, $C_{merge}(n)$ is the number of key comparison made during the merging stage. In the worst case:

$C_{merge}(n) = 2\ C_{merge}(n/2) + n-1$ for $n > 1$

$C_{merge}(1) = 0$

- Solving the recurrence equation using master theorem:

$C(n) = 2C(n/2) + n-1$ for $n > 1$

$C(1) = 0$

Here $a = 2$

b = 2

f(n) = n; d = 1

Therefore $2 = 2^1$, case 2 holds

$C(n) = \Theta (n^d \log n)$

$= \Theta (n^1 \log n)$

$= \Theta (n \log n)$

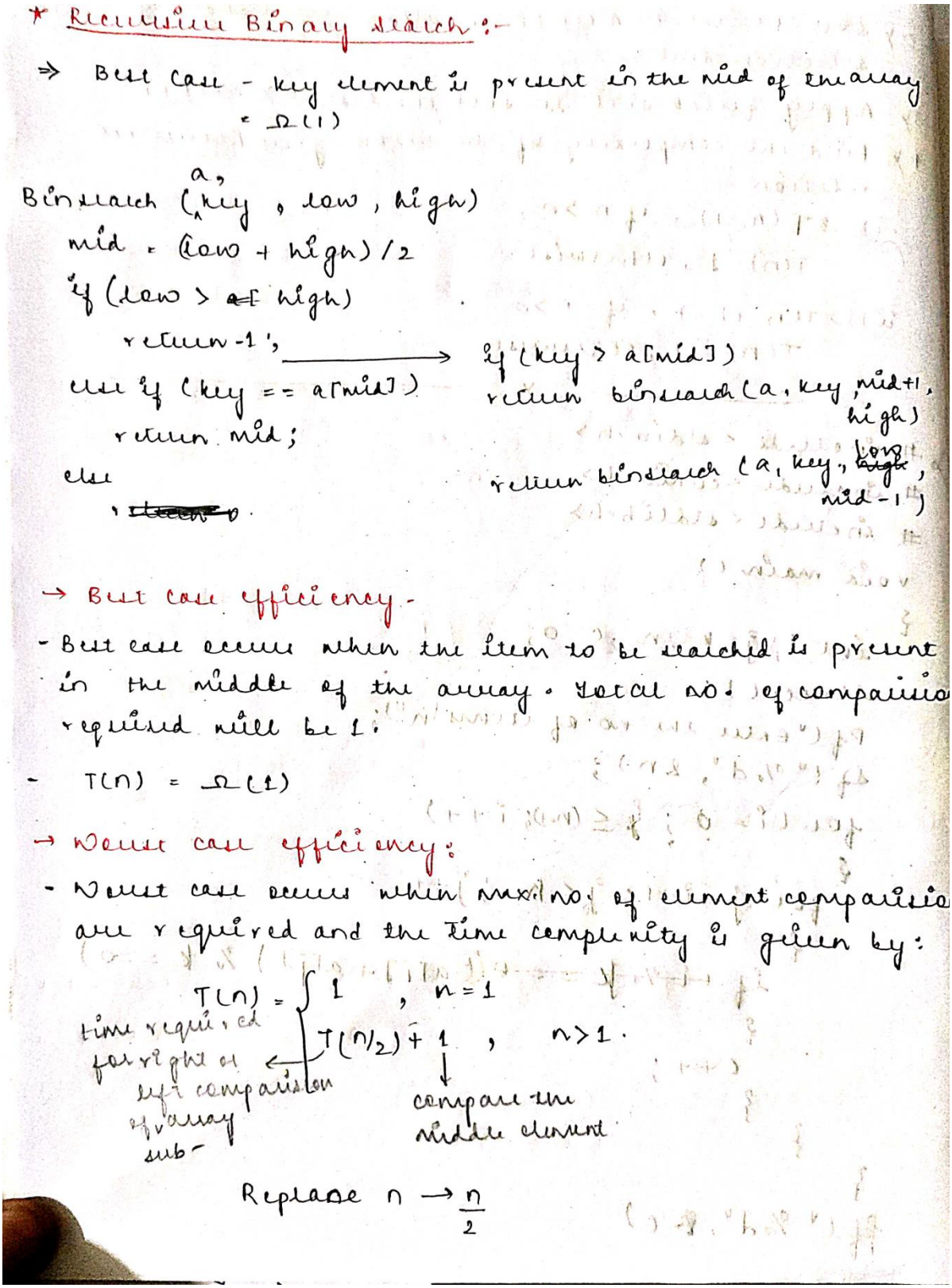
Advantages:

- Number of comparisons performed is nearly optimal.
- Merge sort will never degrade to $O(n^2)$
- It can be applied to files of any size

Limitations:

- Uses O(n) additional memory

Recursive Binary Search



* Recursive Binary search :-

⇒ Best Case - key element is present in the mid of the array
= Ω(1)

```
BinSearch (a, key, low, high)
    mid = (low + high) / 2
    if (low > high)
        return -1;
    else if (key == a[mid])
        return mid;
    else
        if (key > a[mid])
            return binsearch(a, key, mid+1, high)
        return binsearch(a, key, low, mid-1)
```

→ Best case efficiency -

- Best case occurs when the item to be searched is present in the middle of the array. Total no. of comparisons required will be 1.
- $T(n) = \Omega(1)$

→ Worst case efficiency :

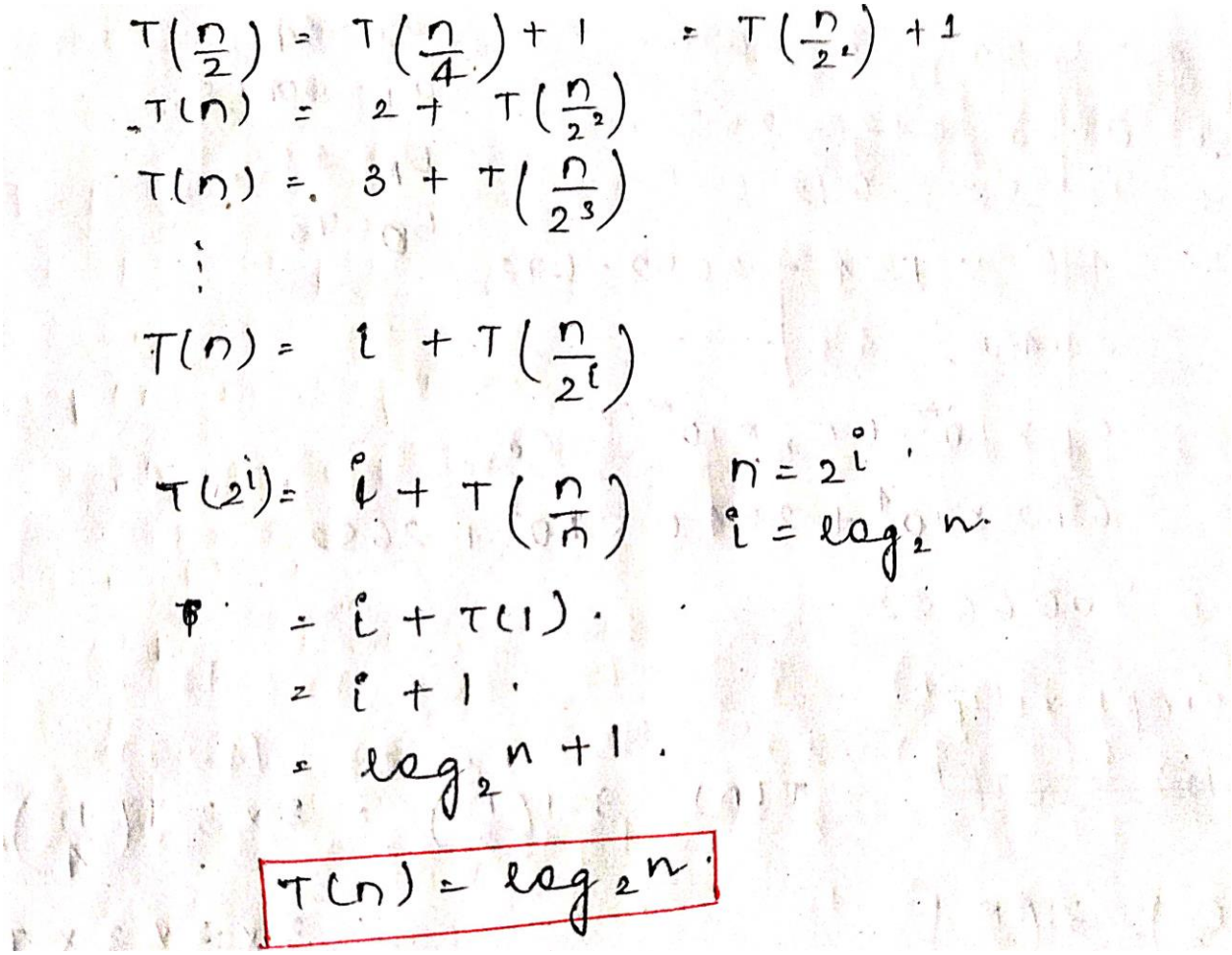
- Worst case occurs when max. no. of element comparisons are required and the time complexity is given by :

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

$T(n)$: time required for right of left comparison of array sub-

$T(n/2)$: Replace $n \rightarrow \frac{n}{2}$

$+1$: compare the middle element



Multiplication of Large Integers & Strassen's Matrix Multiplication

Multiplication of Large Integers:

Some applications, notably modern cryptography, require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern computer, they require special treatment.

To demonstrate the basic idea of the algorithm, let us start with a case of two-digit integers, say, 23 and 14. These numbers can be represented as follows:

$23 = 2 . 10^1 + 3 . 10^0$ and $14 = 1 . 10^1 + 4 . 10^0$.

Now let us multiply them:

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

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

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

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

Of course, there is nothing special about the numbers we just multiplied. For any pair of two-digit numbers a = a1a0 and b = b1b0, their product c can be computed by the formula

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

where

$c2 = a_1 * b_1$ is the product of their first digits,

$c0 = a_0 * b_0$ is the product of their second digits,

$c1 = (a_1 + a_0) * (b_1 + b_0) - c_2 - c_0$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 and c_0.

If n/2 is even, we can apply the same method for computing the products c2, c0, and c1. Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers. In its pure form, the recursion is stopped when n becomes 1. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

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

$M(n) = 3M(n/2)$ for $n > 1$, $M(1) = 1$.

A=3,b=2 d=0

$= 3>2^0$=true

$=n^{\log_b a}$

$=n^{\log_2 3}$

$=n^{1.585}$

Eg: A=1234 B=5678

A0=34

A1=12

B0=78

B1=56

C0=a0 b0=2652

C1=a1 b1=12*56=672

C2= (34+12)(78+56)-672-2652

=2840

$C1*10^m+c2*10^{m/2}+c0$ m= no. of digits

$=672*10^4+2840*10^2+2652$

=7006652

Compute $2101 * 1130$ by applying the divide-and-conquer algorithm outlined in the text.

Strassen's Matrix Multiplication

We have seen that the divide-and-conquer approach can reduce the number of one-digit multiplications in multiplying two integers; we should not be surprised that a similar feat can be accomplished for multiplying matrices. Such an algorithm was published by V. Strassen in 1969 [Str69]. The principal insight of the algorithm lies in the discovery that we can find the product C of two 2 × 2 matrices X and Y with just seven multiplications as opposed to the eight required by the brute-force algorithm.

$$X=\begin{matrix} A & B \\ C & D \end{matrix} \quad * \quad Y=\begin{matrix} E & F \\ G & H \end{matrix}$$

C1=A*E +B*G
C2=A*F+A*H
C3=C*E+D*G
C4=C*F+D*H
Total No: of multiplications=8

P1=A*(F-H)
P2=H*(A+B)
P3=E*(C+D)
P4=D*(G-E)
P5= (A+D)*(E+H)
P6= (B-D) *(G+H)
P7= (A-C)*(E+F)

$$C=\begin{matrix} P6+P5+P4-P2 & P1+P2 \\ P3+P4 & P1+P5-P3-P7 \end{matrix}$$

$$X=\begin{matrix} A & B \\ C & D \end{matrix} \quad * \quad Y=\begin{matrix} E & F \\ G & H \end{matrix}$$

1. **AHED**
2. **DIAGONALS**
3. **LAST CR**
4. **FIRST CR**

- **For Columns put "-"**
- **For Row put "+"**
- **If Checking in X go for opposite Column**
- **If Checking in Y go for opposite row**

$$X=\begin{matrix} 1 & 2 \\ 5 & 6 \end{matrix} \quad * \quad Y=\begin{matrix} 8 & 7 \\ 1 & 2 \end{matrix}$$

P1=1*(7-2)=5
P2=2*(1+2)=6
P3=8*(5+6)=88
P4=6*(1-8)=-42
P5= (1+6)*(8+2)=70
P6= (2-6) *(1+2)=-12
P7= (1-5)*(8+7)=-60

$$C=\begin{matrix} -12+70-42-6 & 5+6 \\ 88-42 & 5+70-88+60 \end{matrix}$$

$$C=\begin{matrix} 10 & 11 \\ 46 & 47 \end{matrix}$$

Time Complexity:

$$T(n)=\begin{cases}1 & n=1\\ 7*T\left(\frac{n}{2}\right) & n>1\end{cases}$$

Applying Master's Theorem
a=7, b=2,d=0
$=7>2^0$
$=n^{\log_2 7}$

$= n^{2.807}$

By brute force Approach it is n^3

$$X=\begin{matrix}1 & 4\\ 3 & 5\end{matrix} \quad * \quad Y=\begin{matrix}1 & 3\\ 4 & 7\end{matrix}$$

Maximum Sub Array Problem

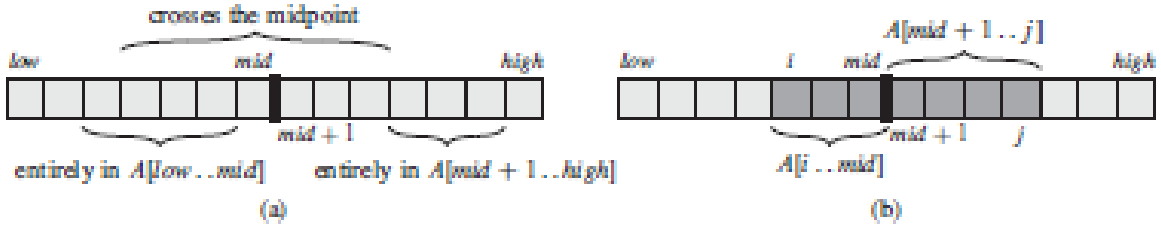
You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum.

For example, if the given array is {-2, -5, **6, -2, -3, 1, 5**, -6}, then the maximum subarray sum is 7 (see highlighted elements).

Suppose we want to find a maximum subarray of the subarray A[*low* …. *High*]. Divide-and-conquer suggests that we divide the subarray into two subarrays of as equal size as possible. That is, we find the midpoint, say *mid,* of the subarray, and consider the subarrays A[*low…mid*] and A[*mid* +1 …*high*]. Any contiguous subarray A[i … j]of A[*low* …*high*] must lie in exactly one of the following places:

- entirely in the subarray A[*low* … *mid*], so that *low* <=i <= j <= *mid*,
- entirely in the subarray A[*mid* +1… *high*] so that *mid* <i <=j <=*high*, or
- crossing the midpoint, so that *low* <= i <=*mid* < j <= *high*.

Therefore, a maximum subarray of A[*low…. High]* must lie in exactly one of these places.




```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2      return (low, high, A[low])          // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 4      (left-low, left-high, left-sum) =
            FIND-MAXIMUM-SUBARRAY (A, low, mid)
 5      (right-low, right-high, right-sum) =
            FIND-MAXIMUM-SUBARRAY (A, mid + 1, high)
 6      (cross-low, cross-high, cross-sum) =
            FIND-MAX-CROSSING-SUBARRAY (A, low, mid, high)
 7      if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8          return (left-low, left-high, left-sum)
 9      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10          return (right-low, right-high, right-sum)
11      else return (cross-low, cross-high, cross-sum)
```

```
FIND-MAX-CROSSING-SUBARRAY (A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4      sum = sum + A[i]
 5      if sum > left-sum
 6          left-sum = sum
 7          max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11      sum = sum + A[j]
12      if sum > right-sum
13          right-sum = sum
14          max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

We denote by T (n)the running time of FIND-MAXIMUM-SUBARRAY on a subarray of n elements. For starters, line 1 takes constant time. The base case, when n =1, is easy: line 2 takes constant time, and so

T(1)=O(1).

Each of the sub problems solved in lines 4 and 5 is on a subarray of n/2 elements (our assumption that the original problem size is a power of 2 ensures that n/2 is an integer), and so we spend T (n/2) time solving each of them. The call to FIND-MAX-CROSSING-SUBARRAY in line 6 takes O(n) time.

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

Combining equations (4.5) and (4.6) gives us a recurrence for the running time $T(n)$ of FIND-MAXIMUM-SUBARRAY:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases} \quad (4.7)$$

a=2,b=2,d=1

$a=b^d$

$2=2^1$

= nlogn

DECREASE-AND-CONQUER

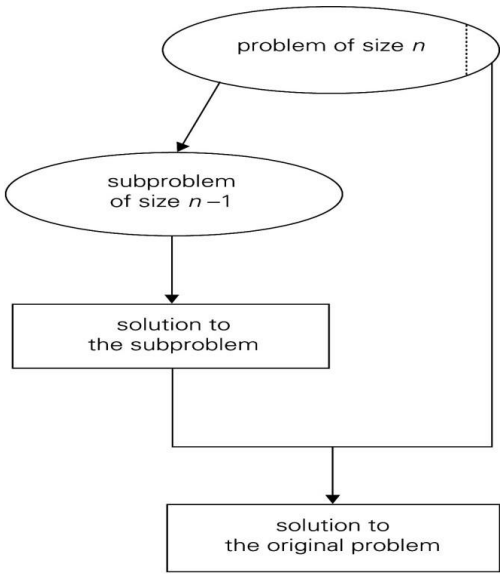
Decrease & conquer is a general algorithm design strategy based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. The exploitation can be either top-down (recursive) or bottom-up (non-recursive).

The major variations of decrease and conquer are

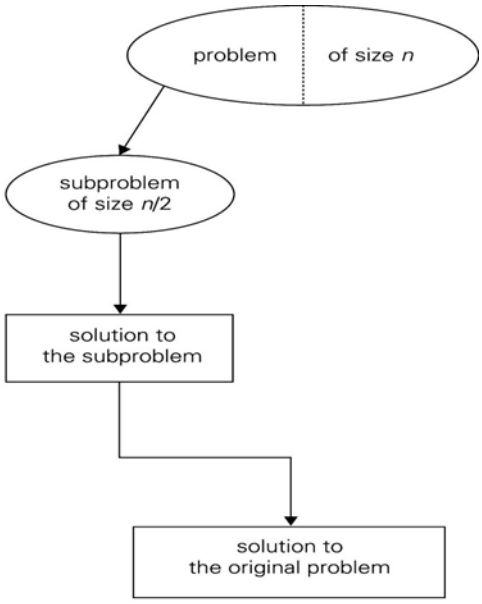
1. Decrease by a constant :(usually by 1):
    a. insertion sort
    b. graph traversal algorithms (DFS and BFS)
    c. topological sorting
    d. algorithms for generating permutations, subsets
2. Decrease by a constant factor (usually by half)
    a. binary search and bisection method
3. Variable size decrease
    a. Euclid‘s algorithm

Following diagram shows the major variations of decrease & conquer approach.

Decrease by a constant :(usually by 1):



Decrease by a constant factor (usually by half)



Insertion sort:

Description:

Insertion sort is an application of decrease & conquer technique. It is a comparison based sort in which the sorted array is built on one entry at a time

$$A[0] \le \ldots \le A[j] < A[j+1] \le \ldots \le A[i-1] \mid A[i] \ldots A[n-1]$$

smaller than or equal to $A[i]$ greater than $A[i]$

ALGORITHM Insertion sort(A [0 … n-1])

//sorts a given array by insertion sort

//i/p: Array A[0…n-1]

//o/p: sorted array A[0…n-1] in ascending order

for i→1 to n-1

item←A[i]

j←i-1

while j ≥ 0 AND item<a[j] do

A[j+1]←A[j]

j←j – 1

end while

A[j + 1]←item.

Analysis:

- **Input size:** Array size, n
- **Basic operation:** key comparison
- Best, worst, average case exists

  Best case: when input is a sorted array in ascending order:

  Worst case: when input is a sorted array in descending order:
- Let $C_{worst}(n)$ be the number of key comparison in the worst case. Then

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

- Let Cbest(n) be the number of key comparison in the best case.

Then

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

Example:
Sort the following list of elements using insertion sort:
25,75,40,10, 20

0	1	2	3	4
10	20	25	40	75

j i

i=1,item= 75, j=0

0>=0 & 75<25 (f)

i=2,item=40,j=1,0

1>=0 & 40<75 (T)

0>=0 & 40<25(F),

I=3, item=10,j=2,1,0

2>=0 & 10<75(T)

1>=0 & 10<40(T)

0>=0 & 10<25(T)

a[0]=10

i=4, item=20,j=3,2,1,0

3>=0 & 20 <75(T)

2>=0 & 20 <40 (T)

1>=0 & 20<25 (T)

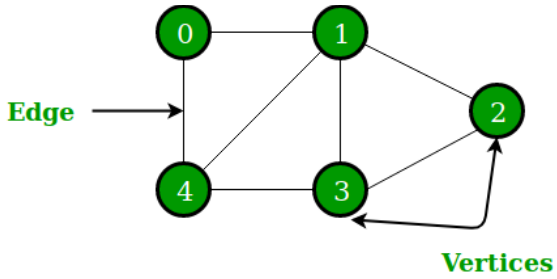
0>=0 & 20<10 (F)

A[1]=20

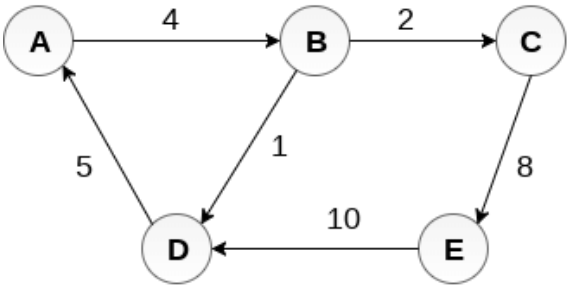
Advantages of insertion sort:

- Simple implementation. There are three variations
  - Left to right scan
  - Right to left scan
  - Binary insertion sort
- Efficient on small list of elements, on almost sorted list
- Running time is linear in best case
- Is a stable algorithm
- Is an in-place algorithm

Representation of Graphs



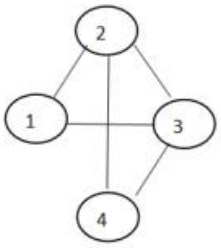
Graph



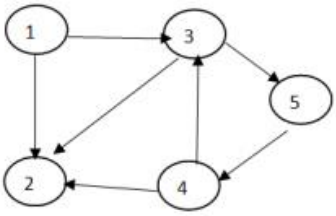
Weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

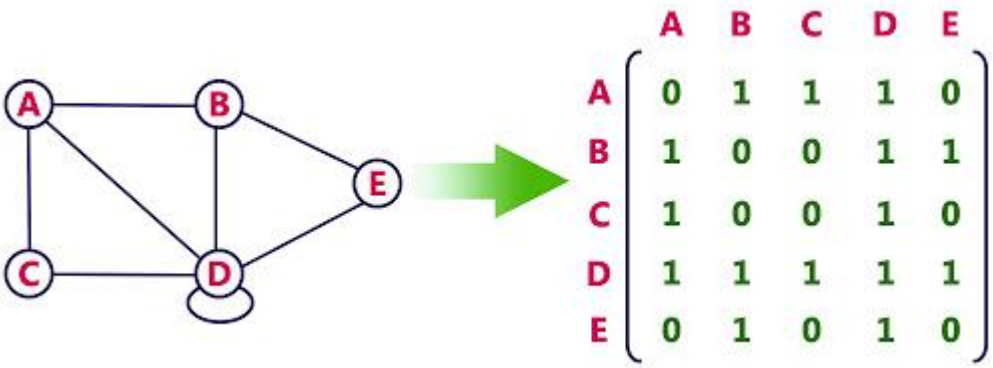
Adjacency Matrix



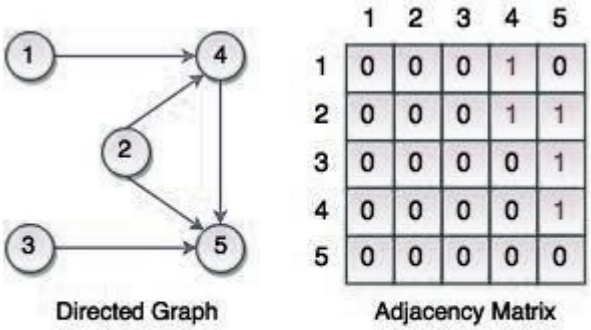
Undirected graph



Directed graph.



Matrix Representation of Graph



Directed Graph

Adjacency Matrix

Fig. Adjacency Matrix Representation of Directed Graph

Graph Traversals

- Traversals means visiting the nodes of graph one after the other in systematic manner.
- Traversals can start at any arbitrary vertex.
- There are two types of graph traversal techniques

a. Breadth First Search (BFS)
b. Depth First Search (DFS)

Breadth First Search (BFS)

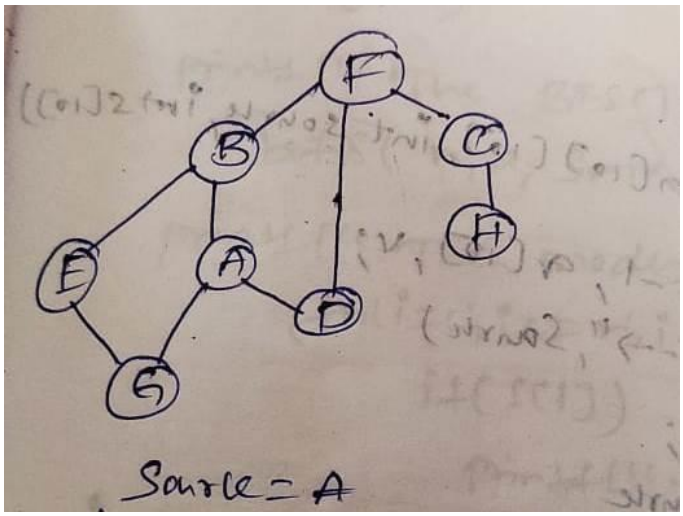
The traversal's starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called a tree edge. If an edge leading to a previously visited vertex other than its immediate predecessor (i.e., its parent in the tree) is encountered, the edge is noted as a cross edge.

Here is pseudocode of the breadth-first search.

ALGORITHM *BFS*(G)

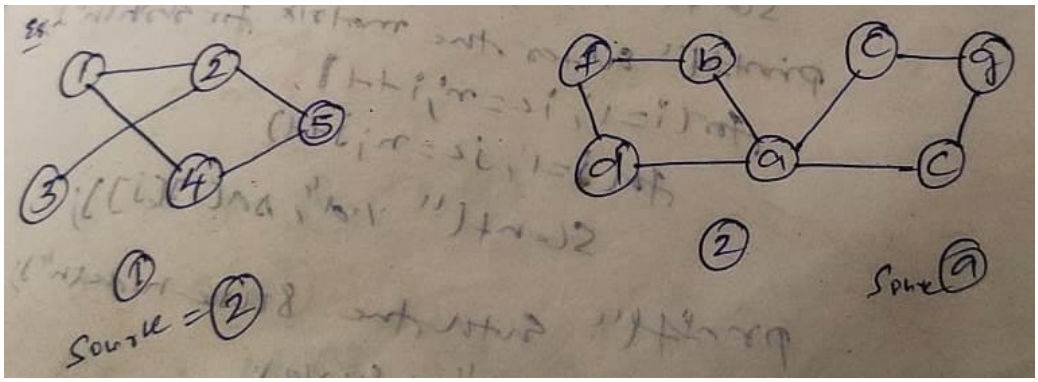
```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//        in the order they are visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each 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 numbers them in the order they are visited
//via global variable count
count ← 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 ← count + 1;  mark w with count
            add w to the queue
    remove the front vertex from the queue
```

Traverse the following graph using BFS



Deque	Adjacent Nodes	Nodes Visited	Queue
-	-	A	A
A	B D G	A B D G	B D G
B	A E F	A B D G E F	D G E F
D	A F	A B D G E F	G E F
G	A E	A B D G E F	E F
E	B G	A B D G E F	F
F	B C D	A B D G E F C	C
C	F H	A B D G E F C H	H
H	C	A B D G E F C H	EMPTY

A -> B->D->



Lab Program 2

i) Using Decrease and Conquer strategy design and execute a program in C, to print all the nodes reachable from a given starting node in a graph using BFS method.

i.
```
#include<stdio.h>

void bfs(int n,int a[10][10],int source,int s[10])
{
  int i,f=0,r=-1,q[10],v;

  printf("%d--->",source);
  s[source]=1;
  q[++r]=source;
  while(f<=r)
  {
        v=q[f++];

        for(i=1;i<=n;i++)
              if(s[i]==0 && a[v][i])
              {
                    q[++r]=i;
                    printf("%d--->",i);
                    s[i]=1;
              }
  }
}


void main()
{
  int n,a[10][10],i,j,source,s[10];

  printf("enter the number nodes in the graph\n");
  scanf("%d",&n);

  printf("enter the matrix for the graph\n");
  for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
              scanf("%d",&a[i][j]);

  printf("enter the source node\n");
  scanf("%d",&source);

  for(i=1;i<=n;i++)
        s[i]=0;
```

```
    printf("the  BFS traversal is\n");

    bfs(n,a,source,s);

    printf("\nthe  nodes reachable are\n");

    for(i=1;i<=n;i++)
            if(s[i])
                    printf("%d\n",i);
}
```

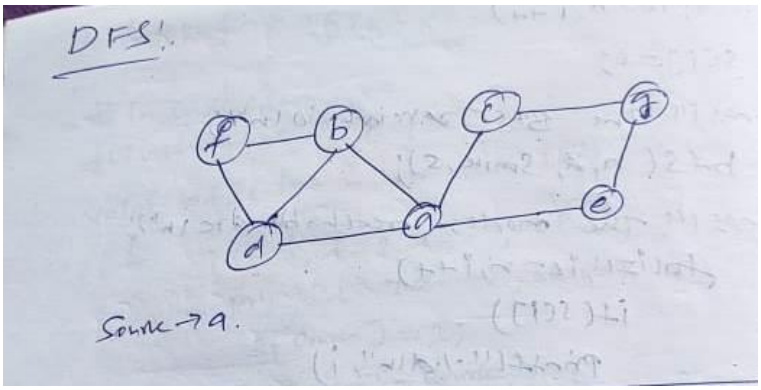
Depth First Search

- DFS starts visiting vertices of a graph at an arbitrary vertex by marking it as visited.
- It visits graph's vertices by always moving away from last visited vertex to an unvisited one, backtracks if no adjacent unvisited vertex is available.
- Is a recursive algorithm, it uses a stack
- A vertex is pushed onto the stack when it's reached for the first time
- A vertex is popped off the stack when it becomes a dead end, i.e., when there is no adjacent unvisited vertex

ALGORITHM DFS (G)
//implements DFS traversal of a given graph
//i/p: Graph G = { V, E}
//o/p: DFS tree
Mark each vertex in V with 0 as a mark of being —unvisited‖ count→ 0
for each vertex v in V do
if v is marked with 0
dfs(v)

dfs(v)
count→count + 1 //mark v with count
for each vertex w in V adjacent to v do
if w is marked with 0
dfs(w)

Stack	Adjacent Nodes	Nodes Visited	Pop stack
a	-	a	-
a	b	a b	-
A b	d	a b d	-
A b d	f	a b d f	-
a b d f	-	a b d f	f
a b d	-	a b d f	d
a b	-	a b d f	b
a	c	a b d f c	-
A c	g	a b d f c g	-
a c g	e	a b d f c g e	-
a c g e	-	a b d f c g e	e
a c g	-	a b d f c g e	g
A c	-	a b d f c g e	C
a	-	a b d f c g e	a
empty			



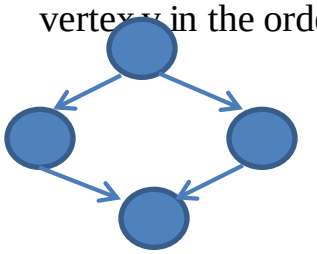
ii. #include<stdio.h>

```
void dfs(int n,int a[10][10],int s[10],int source);

void main()
{
    int j, s[10], a[10][10], source, i, n, flag=0;

    printf("Enter  the number of nodes:\n");
    scanf("%d",&n);
```

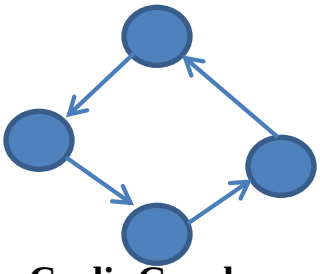
```
    printf("Enter  the adjacency matrix:\n");
    for(i=1;i<=n;i++)
            for(j=1;j<=n;j++)
                    scanf("%d",&a[i][j]);

    printf("Enter  the sourcs node:\n");
    scanf("%d",&source);
    for(i=1;i<=n;i++)
            s[i]=0;

    dfs(n,a,s,source);

    for(i=1;i<=n;i++)
            if(s[i]==0)
                flag=1;

    if(flag==1)
             printf("graph  not connected");
        else
            printf("graph  is connected");
}


void dfs(int n,int  a[10][10],int  s[10],int  source)
{
  int i;
  s[source]=1;
  printf("-->%d",source);
  for(i=1;i<=n;i++)
    if(s[i]==0  && a[source][i]==1)
      dfs(n,a,s,i);
}
```

Topological Sorting

- Applicable for directed acyclic graph.
- The topological sorting of DAG is a linear ordering of all the vertices such that every edge (u,v) in graph G the vertex u appears before the vertex v in the ordering.



Acyclic Graph



Cyclic Graph

- Two types of Sorting Methods
  1. DFS Method
  2. Source Removal Method

Topological Sorting using DFS

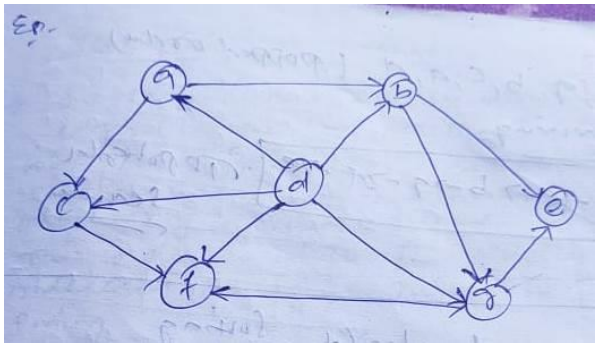
Step1: Select any arbitrary vertex

Step2: when a vertex is visited for first time it is pushed to stack

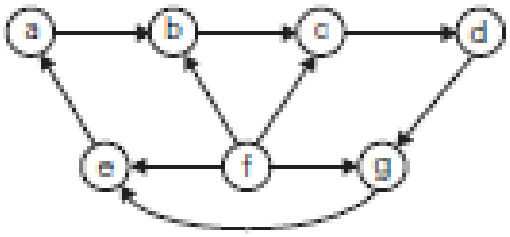
Step 3: when a vertex becomes dead it is removed from stack.

Step 4: repeat step 2 to 3 for all vertices

Step 5: reverse the order of deleted items to get topological sequence.



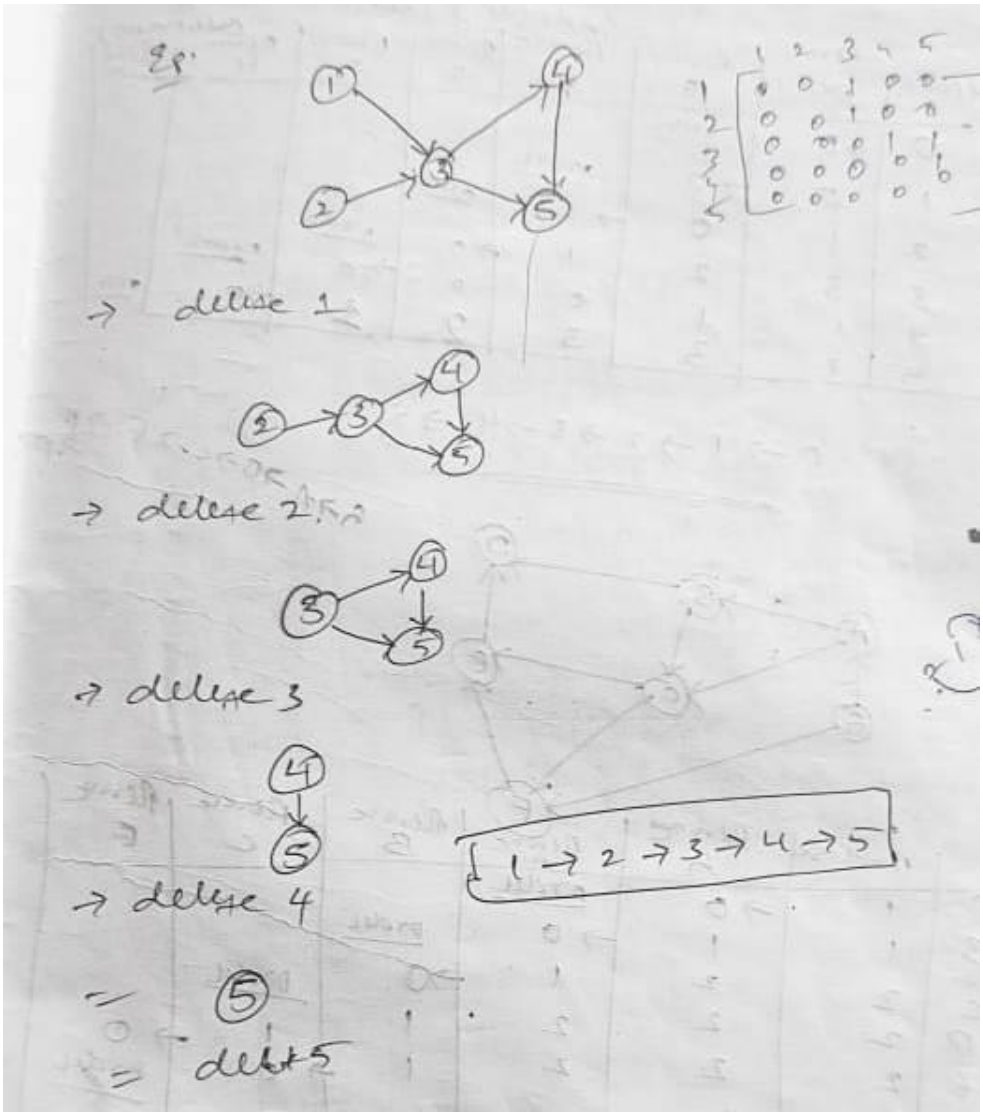
Stack	**Adjacent Nodes**	**Nodes Visited**	**Pop stack**
a	-	a	-
A	b	A b	-
A b	e	A b e	-
A b e	-	A b e	e
A b	g	A b e g	-
A b g	f	A b e g f	-
A b g f	-	A b e g f	f
A b g	-	A b e g f	g
A b	-	A b e g f	b
A	c	A b e g f c	-
A c	-	A b e g f c	c
A	-	A b e g f c	a
d	-	A b e g f c d	-
d	-	A b e g f c d	d
empty			
POP Sequence	e->f->g->b->c->a->d		
Topological Sequence	**d->a->c->b->g->f->e** (reverse of pop sequence)		



(b)

Source Removal Method

- In this method a vertex with no incoming edges (in degree 0) is selected and deleted along with the outgoing edges. If there are several edges with no incoming edges arbitrarily a vertex is selected.
- The order in which the vertices are visited and deleted one by one results in topological sequence.



CHAPTER 5

IV

DECREASE-and-CONQUER

The Decrease and conquer is an approach solving a problem by

* changing an instance into smaller instance of the problem
* solve the smaller instance
* convert the solution of smaller instance into a solution for larger instance.

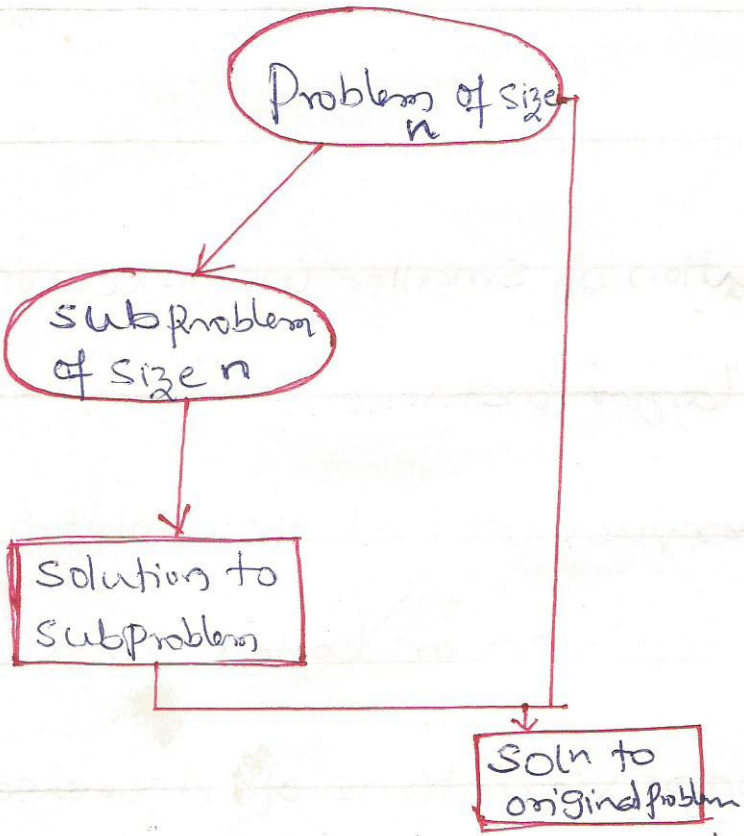
In decrease and conquer method the problem can be solved using topdown or bottom up solution.

The three major variations of decrease and conquer are

1. decrease by constant
2. decrease by constant factor
3. Variable size decrease

I. DECREASE BY CONSTANT

In this method the size of the instance is reduced by same constant on each iteration of the algorithm. Generally this constant is equal to one. The decrease by constant is illustrated in following fig.



Eg: Algorithm to Compute a^n.

In this algorithm the size of the problem, that is reduces by 1 in each recursive call

The algorithm is

```
algorithm power(a,n)
    // a and n are i/ps.
begin
    if (n=1) return a
    return power(n-1,a) * a
End
```

↑ n reduces by 1

Applications of decrease by constant and conquer method is used to solve following problems

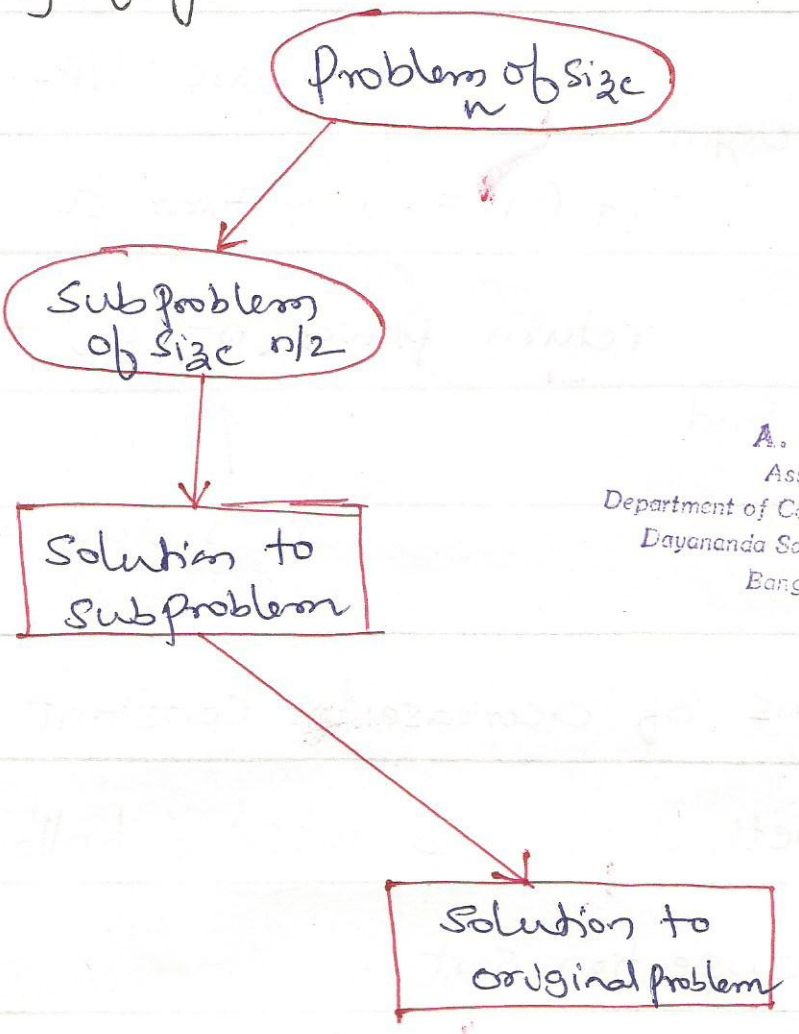
1. Insertion sort
2. Graph searching algorithm
   * Depth first Search
   * Breadth first Search
   * Topological Sorting.

II DECREASE BY A CONSTANT FACTOR

Decrease by a constant factor decreases the instant size by half or by some other fraction.

(4)

The decrease by constant factor is illustrated in following fig.



A. M. PRASAD
Assistant Professor
Department of Computer Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

Eg: Algorithm to compute a^n

1. $a^n = a$ if $n = 1$ Eg: $4^1 = 4$

2. $a^n = a^{n/2} * a^{n/2}$ if n is even

   Eg. if $n=8$, $4^8 = 4^4 * 4^4$

3. $a^n = a^{(n-1)/2} * a^{(n-1)/2} * a$ if n is odd

   if $n=9$, $4^9 = 4^4 * 4^4 * 4$

```
Algorithm  power(a, n)
    // a and n are inputs
    begin
        if (n=1) return a
        if (n>1 and n%2 =0) power(a, n/2)
    return power(a,(n-1)/2) * power(a,(n-1)/2) * a
    End.
```

n reduces by n/2

The applications of decrease by constant factor ~~one~~ binary search. The Problem instance reduces by half inorder to obtain solution.

A. M. PRASAD
Assistant Professor
Department of Computer Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

III Variable size decrease.

The Variable size decrease is one of the variations of decrease and conquer technique. In Variable size decrease method the size reduction pattren varies from one iteration of an algorithm to another.

Eg: The typical ~~variation~~ of this variation is

finding GCD of two numbers using Euclid's algorithm. The formula for finding GCD is

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

Two numbers m and n goon varying until GCD value is found.

Insertion Sort:

Generally the insertion sort maintains a zone of sorted elements. If any unsorted element is obtained then it is compared with the elements in sorted zone and then it is inserted at the proper position in sorted zone.

Illustration of insertion sort: consider the elements

9
7
5
4
2

The number of elements are five, insertion sort needs (n-1) cycles to arrange them in order.

I cycle. bigger → V=7

9 7 7
7 ⇒ 9 9
5 5 5
4 4 4
2 2 2

Store second element in variable V, if first element is bigger than V, copy the first element in second position. Then copy V to first position

II cycle

7 bigger → V=5
9
5
4
2

7 bigger → V=5 7 5 ↙V
9 7 7
9 9 9
4 4 4
2 2 2

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

III cycle

5
7
9
4→V
2

5 bigger V=4
7
9
9
2

5 → bigger V=4 5 4 ↙V
7 7 5 5
9 7 7 7
9 9 9 9
2 2 2 2

IV cycle

4 9>2^V 7>2^V 5>2^V 4>2^V 2 ↙V
5 4 4 4 4 4
7 5 5 5 5 5
9 7 7 7 7 7
V→2 9 9 9 9 9

Algorithm for Insertion Sort given below

```
Algorithm InsertionSort(A[1....n])
// Input array A[1...n] of ordable elements
// output A[1...n] sorted in ascending order
begin
    for i ← 2 to n do
begin   v ← A[i]
        j ← i-1
    while (j ≥ 1 and A[j] > v) do
begin   A[j+1] ← A[j]
        j ← j-1
Endwhile
        A[j+1] ← v
End for
End.
```

Time Efficiency: Let T(n) be the time taken by insertion sort algorithm to arrange numbers in ascending order. The worst case input is all the 'n' numbers in decending order. The time complexity depends upon basic operation.

The basic operation is A[j]>V. and how many times it is executed depends upon loops.

$$T(n) = \sum_{i=2}^{n} \sum_{j=1}^{i-1} 1$$

```
for i<- 2 to n
V= a[i]
j= i-1
while(j>=1 and a[j]>V)
```

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

$$= \sum_{i=2}^{n} i-1-1+1$$

$$= \sum_{i=2}^{n} i-1$$

$$= 1 + 2 + 3 + \cdots\cdots\cdots + (n-1)$$

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

$$= \frac{n^2-n}{2}$$ note: ignore constant

$$= n^2-n$$ note: Consider higher order term

$= n^2$ Since it is worst case time efficiency use O notation.

So $T(n) = O(n^2)$

Advantages of Insertion sort

1. simple to implement
2. This method is efficient, when n is small
3. This is stable algorithm
4. It is in-place sorting algorithm.

Disadvantages are

1. not efficient compare to quick, mergesort
2. Not suitable when 'n' is large.

GRAPH TRAVERSALS

Graph Traversal means visiting nodes of a graph one after the other in a systematic way. Traversal can start from any arbitrary vertex. The two important graph traversal methods are 1) Deapth first Search (DFS)

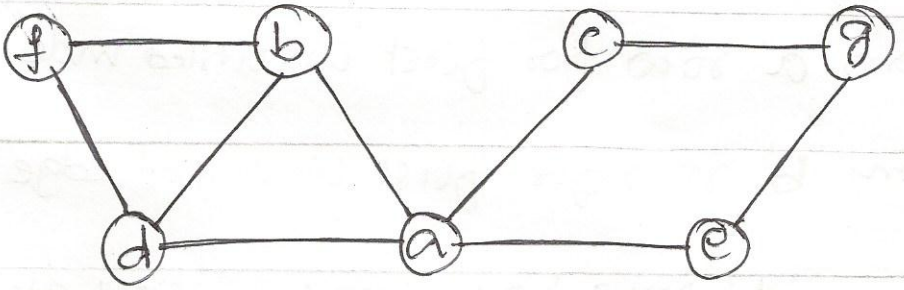
2) Breadth first Search (BFS)

Deapth first Search (DFS)

In DFS a arbitrary Vertex called source is selected and it is visited. Then from source find out the first immediate or adjacent edge and visit that node. Repeat the same process from the new node visited. The search will terminate when all the vertices have been examined or visited.

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engine ..g
Bangalore - 560 078

Consider the following graph



for the graph given, write adjacency matrix.

consider the source as a

(12)

	a	b	c	d	e	f	g
a	0	1	1	1	1	0	0
b	0	0	0	1	0	1	0
c	1	0	0	0	0	0	1
d	1	1	0	0	0	1	0
e	1	0	0	0	0	0	1
f	0	1	0	1	0	0	0
g	0	0	1	0	1	0	0

first node to be visited is source a

now scan 'a' row for first unvisited node b, visit

now scan 'b' row for first unvisited edge d. visit

now scan 'd' row for first unvisited edge f, visit

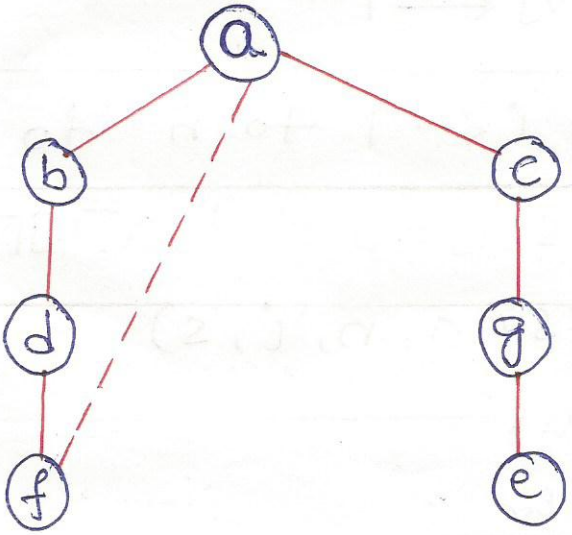
now scan 'f' row for first unvisited edge, two edges are present but they are visited. Now consider the vertices in reverse order one by one and search for unvisited edge. and visit that node

from f, d, b no vertex can be visited but from a, c can be visited, visit c. Now from vertex c, search first unvisited node, g can be visited. from g, vertex e can be visited. Now all vertices are visited, search stops.

The DFS order is

a, b, d, f, c, g, e

In this example, after visiting vertex f, no node can be visited from f, so unvisited node is searched from visited nodes. The from a vertex c is visited. They are called as Back Edges.



The data structure "Stack" is used to conduct DFS. All visited nodes are stored in stack.

When no unvisited edges are present from vertex pop the vertex on top of the stack and search for unvisited edge. This process is repeated till all nodes are visited.

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

Algorithm:

```
Algorithm Dfs(a, n, v, s)
    // a is adjacency matrix
    // n is number of nodes
    // v is source node
    // s is array indicates vertices visited or not
begin
    s[v] ← 1
    for i ← 1 to n do
    if (s[i]=0 and a[v][i]=1) then
     Dfs(a, n, i, s)
    Endif
    Endfor
End.
```

Analysis.

Every node is visited once, if adjacency matrix is used, it is $n \times n$ matrix is used

$$T(n) = O(|V^2|) \quad V \text{ is Vertices.}$$

if adjacency list is used then

$$T(n) = O(|V| + |E|)$$

Applications of DFS

1. To find graph is connected or not
2. To check graph is acyclic or not
3. To find Spanning Tree
4. To solve Topological Sorting.

Breadth first Search (BFS)

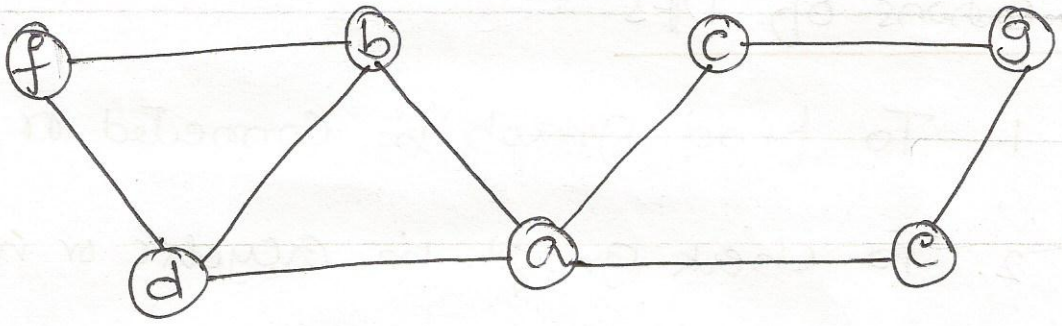
BFS is method of traversing the Graph by visiting each node of the Graph. In BFS arbitrary node called source node is selected and that is the first node to be visited. From source

node visit all nodes adjacent to it. All nodes visited are stored in queue. The pop the vertex from queue and visit all the nodes adjacent to it and store them in queue. Then pop the next vertex and repeat the same. till queue becomes empty.

Vertex

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engine...g
Bangalore - 560 078

Consider the following Graph



The adjacency matrix is

	a	b	c	d	e	f	g
a	0	1	1	1	1	0	0
b	0	0	0	0	0	1	0
c	1	0	0	0	0	0	1
d	0	1	0	0	0	1	0
e	1	0	0	0	0	0	1
f	0	1	0	1	0	0	0
g	0	0	1	0	1	0	0

If source is a, visit a first, then from a which all the nodes can be visited, visited all of them ie

a, b, c, d, e

now pop the element from queue, b, visit all the nodes from b and add them to queue

a, b, c, d, e, f

Pop the next element c, visit all unvisited nodes

a, b, c, d, e, f, g

This process is repeated till queue becomes empty.

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

Algorithm: BFS(a, n, s, source)

// a is adjacency matrix, n is no of nodes
// source is starting node, s indicates node visited/not

```
begin   f ← r ← 01
        q[r] ← source
        s[source] ← 1,  Print source
        while (f <= r) do
begin   u ← q[f]
        f ← f+1
```

(18)

```
for i ← 1 to n do
begin if (s[i]=0 and a[u][i]=1) then
    begin
        r ← r+1
        q[r] ← i
        s[i] ← 1
        Print i
    Endif
Endfor
Endwhile
End.
```

Time Efficiency: The basic operation is if (s[i]=0 and a[u][i]=1). The basic operation execution depends upon loops for and while. Since a graph can have 'n' nodes, the while loop runs 'n' times. For loop runs 1 to n each time.

Let T(n) be the time taken to visit 'n' vertices using BFS method

$$T(n) = \sum_{j=1}^{n} \sum_{i=1}^{n} 1$$

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

$$T(n) = \sum_{j=1}^{n} n - 1 + 1$$

$$T(n) = \sum_{j=1}^{n} n$$

$$T(n) = n(n-1+1)$$

$$T(n) = n^2$$

Since this is time taken in average or worst case θ or O notations are used. So

$$T(n) = \theta(n^2) \text{ or } \theta(|V^2|)$$

The differences between BFS and DFS are

(20)

1. Data structures is Stack	Data structure is Queue
2. Tree edges and back edges are present	Tree edges and cross edges are present
3. Exploration of node is postponed as soon as a unvisited node is reached.	A node is fully explored before exploration of new node
4. Two vertex ordering is used	one Vertex ordering is used.

TOPOLOGICAL SORTING

The method of arranging vertices in some specific manner is called topological sort. The method is directed acyclic graph, each time select the vertex without indegrees, that is the node visited. The remove the node from graph, check the other nodes with indegree zero, select that node and remove the node from the graph and repeat the same process.

A graph which is cyclic doesnot have topological sequence. Topological sorting can be done using two methods

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

1. DFS method
2. Source removal method.

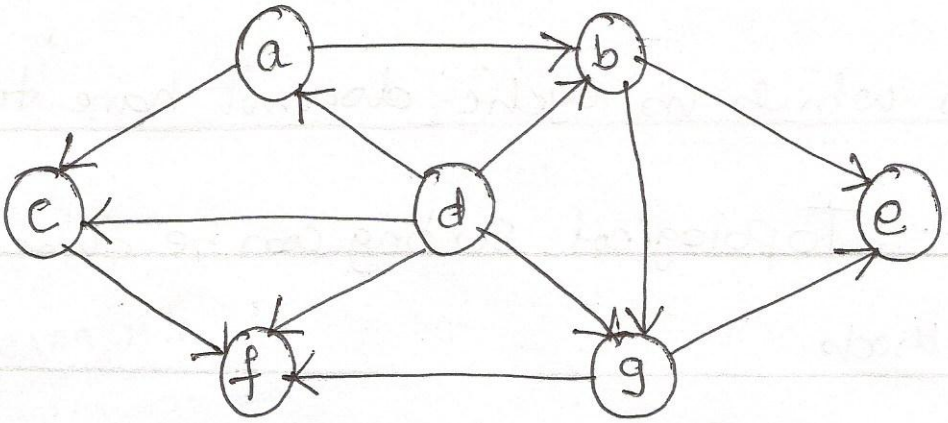
DFS method

The following steps are used to find topological order

1. Select any arbitrary Vertex
2. When Vertex visited first time it is pushed on to stack
3. When Vertex becomes a dead end, it is removed
4. repeat above two steps till all nodes are over.
5. Reverse the order of deleted nodes.

Consider the graph shown

(22)



Let Source be a

Stack	nodes to be visited	Node deleted
a	b	–
a, b	e	–
a, b, e	from e no nodes so delete e	e
a, b	g	–
a, b, g	f	–
a, b, g, f	from f no nodes so delete f	f
a, b, g	from g no nodes so delete g	g
a, b	from b no nodes so delete b	b
a	c	–
a, c	from c no nodes so delete c	c

a — from a no nodes so delete a — a

Now Stack is empty, any node left in graph push it into stack. The node left is d

d — no node can be visited, so pop — d

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

The nodes poped sequence is

e, f, g, b, c, a, d

The topological order is reverse of popped sequence

d, a, c, b, g, f, e

Algorithm:

```
algorithm topology(s,n, a)
// s is array indicates node visited/not
// a is adjacency matrix
// n is number of vertices.
Step 1: for i ← 01 to n do
        if (s[u]=0) call DFS(u, n, a)
        endfor
```

Step 2: for i ← n downto 1

Print result[i]

Step 3: End

Algorithm DFS(u, n, a)

begin

S[u] ← 1

for v ← 1 to n do

if (a[u][v]=1 and s[v]=0) then

DFS(v, n, a)

Endif

Endfor

result[j++] = u

End

Time Efficiency: Let T(n) be the time taken by algorithm to sort vertices. The basic or key operation is

if (a[u][v]=1 and S[v]=0) then
DFS(v, n, a)

The number of times it is executed on for loop

```
for V ← 1 to n do
if (a[u][v] = 1 and s[v] = 0) then
DFS (v, n, a)
```

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

$$T(n) = \sum_{v=1}^{n} \sum_{v=1}^{n} 1$$

$$T(n) = \sum_{v=1}^{n} n-1+1$$

$$T(n) = \sum_{v=1}^{n} n$$

$$T(n) = n(n-1+1)$$

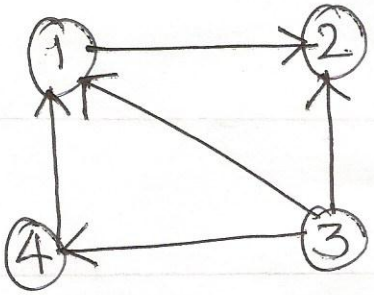
$$T(n) = n^2$$

So time efficiency is $T(n) = \Theta(n^2)$

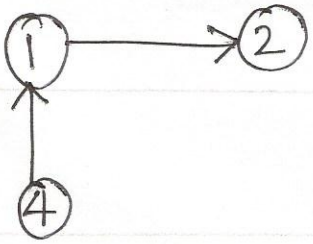
TOPOLOGICAL SORTING USING SOURCE REMOVAL

In this method, a vertex with no incoming edges is selected and deleted along without going edges. Then select the vertex with no indegrees and delete that edge along with outgoing edges. Repeat this method till all nodes in graph are deleted. The nodes deleted order is topological sequence

Eg: Consider the following graph



step 1: delete 3, because it has no indegrees

step 2:  delete node 4, no indegrees

step 3: 

delete node ①, no indegrees

step 4: ②

delete ②, no indegrees

The deleted sequence is 3, 4, 1, 2 and is topological order.

The logic is for given graph write adjacency matrix

	1	2	3	4
1	0	1	0	0
2	0	0	0	0
3	1	1	0	1
4	1	0	0	0

Now, take sum of each column, the column with sum equal to zero is the first node to be

visited. Then remove that column and row and take the sum of elements in column. The column with 0 is the next node to be visited. This is repeated all nodes are over.

```
Algorithm Topology (a, n)
// a is adjacency matrix, n is number of nodes
begin
    for i ← 1 to n do
    for j ← 1 to n do
    if (a[i][j] = 1) sc[j] = sc[j] + 1
    End
    End
    for i ← 1 to n do
begin
    if (sc[i] = 0) then
begin
        Print i
        sc[i] = -1
        for j ← 1 to n do
begin
        if (a[i][j] = 1) then
        sc[j] = sc[j] - 1
        Endif
        i = 0
        End for
    Endif
    Endfor
End
```

A. M. PRASAD
Assistant Professor
Department of Compuier Science & Engineering
Dayananda Sagar College of Engineering
Bangalore - 560 078

SPACE AND TIME TRADE OFFS

In space and time trade off the idea is to preprocess the problem's input in whole or in a part and store the additional information obtained to accelerate solving the problem later. This is called input enhancement technique

The algorithms under this are

<1> Sorting by counting
<2> Horspool's algorithm
<3> Hashing.

SORTING BY COUNTING:

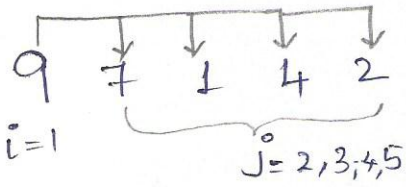
Sorting by counting is also called as comparison counting. It is one of the sorting approaches which uses the input enhancement technique to sort the data. It is used to sort the elements in ascending order. In this method, for each element of the list to be sorted total number of elements smaller than this element are recorded in a table. These recorded numbers would indicate the positions of the elements in the sorted list.

In this algorithm the i^{th} element is compared with the remaining 'j' elements. If incase the i^{th} element is greater than

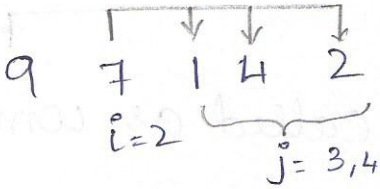
j^{th} element then the count[i] is incremented. Else if the j^{th} element happens to be greater than the i^{th} element the count[j] is incremented. Here i varies from 1 to (n-1) and j from i+1 to n. At the end of all the n-1 passes the count table elements indicates the position of each element of the array in the sorted list.

ILLUSTRATION OF ALGORITHM

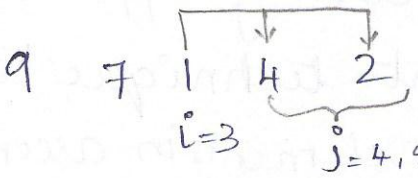
Consider the elements 9, 7, 1, 4, 2



count[1] = 1+1+1+1+1 = 5



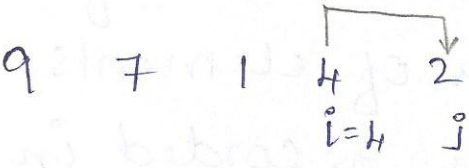
count[2] = 1+1+1+1 = 4



count [3] = 1

count [4] = 1+1 = 2

count [5] = 1+1 = 2



count[4] = 2+1 = 3

9 7 1 4 2
i

count[5] = 2+0 = 2

∴ Count table elements are

1	2	3	4	5
5	4	1	3	2

The sorted elements are

	1	2	3	4	5
b[]	1	2	4	7	9

In the above example, during the 1st pass, i=1, j = 2, 3, 4, 5. a[1] = 9. This element is compared with a[2 ... 5]. If a[1] > a[2 ... 5] then, count[1] is incremented. If a[1] < a[j] where j varies from 2 to 5 then, count[j] is incremented. Thus, we get count[1] = 5 which implies that the element 9 resides in the 5th position in the sorted array.

ALGORITHM FOR SORTING BY COUNTING:

```
Algorithm sorting_by_counting (a, n)
BEGIN
    for i ← 1 to n do
        count[i] = 1

    for i ← 1 to n-1 do
    BEGIN
        for j ← i+1 to n do
        BEGIN
            if (a[i] > a[j])
                count[i] ← count[i] + 1
            else
                count[j] ← count[j] + 1
        END FOR
    END FOR
    for i ← 1 to n do
    b[count[i]] = a[i]
END
```

TIME EFFICIENCY:

Let T(n) be the time required to sort an array using sorting by counting. Let the number of elements be n. Then, the time complexity is given depends on the two for loop $i \leftarrow 1$ to $n-1$ and $j \leftarrow i+1$ to n. The basic operation is if (a[i] > a[j]). The number of times it gets executed is given by.

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1$$

$$= \sum_{i=1}^{n-1} [n-(i+1)+1]$$

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

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

$$= (n-1) + (n-2) + \cdots + 3+2+1$$

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

$$= n^2 - n$$

NOTE: Constants are not considered in time efficiency analysis

$$= n^2$$

NOTE: Higher order term is considered

Since it is average case analysis

$$T(n) = \theta(n^2)$$

HORSPOOL'S ALGORITHM

Horspool's algorithm is one of the string matching algorithm that uses the technique called input enhancement to search for a pattern in a given text.

In this algorithm initially we create a Shift table using the formula $m-1-i$ where m is the length of the pattern string and i varies from 0 to $m-2$. This indicates the shift to be performed during searching process. The following 2 possibilities may occur during pattern searching.

(1) If the required character is not present in the pattern, then the pattern is shifted by its length

(2) If the character occurs in the text then the pattern is shifted based on the entry present in the shift table.

We first compare the last character of the pattern with the corresponding i^{th} character of the text. If the 2 characters are equal then the other characters of the pattern string are matched. Else the shift is performed as mention above.

The above process is performed til either the pattern is found or the text is exhausted.

ILLUSTRATION OF HORSPOOL'S ALGORITHM:

Consider the ~~text~~ string

TEXT: INDIA _ IS _ A _ DEMOCRATIC

PATTERN: DEMO

SHIFT TABLE

D ---> 3

E --> 2

M --> 1

OTHER --> 4

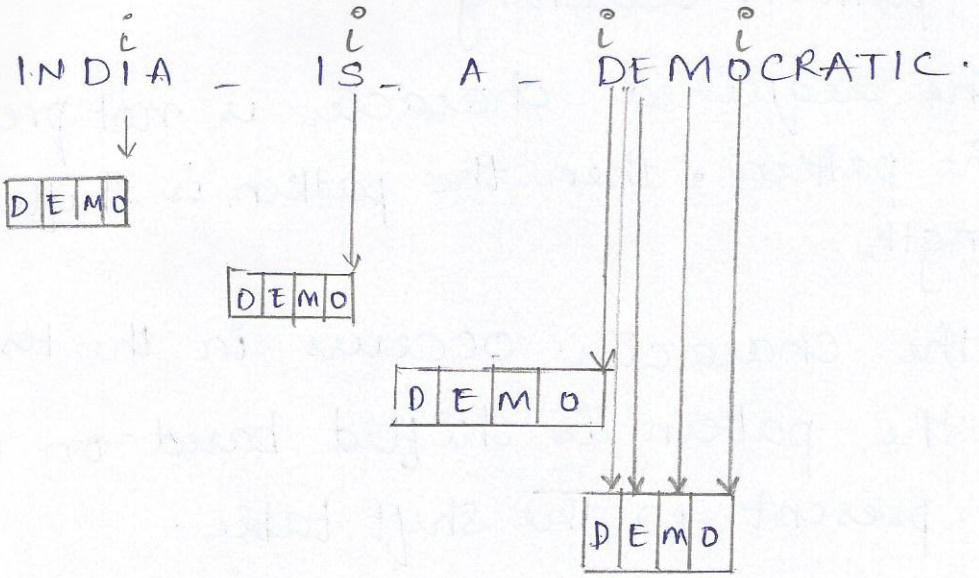


TABLE [I] --> 4
TABLE [N] --> 4
[D] --> ~~4~~ 3
[I] --> 4
[A] --> 4
[-] --> 4
[I] --> 4
[S] --> 4
[A] --> 4
[-] --> 4
[D] --> ~~4~~ 3
[E] --> ~~4~~ 2
[M] --> ~~4~~ 1
[O] --> 4
[C] --> 4
[R] --> 4
[A] --> 4
[T] --> 4
[I] --> 4
[C] --> 4

```
Algorithm  Horspool (T, P, m, n)

begin
    for i ← 0 to n-1 do
     table [T[i]] ← m
     for i ← 0 to m-2 do
     table[P[i]] ← m-1-i
       i = m-1
     while (i <= n-1) do
     begin
         j ← 0
       while (j < m and T[i-j] = P[m-1-j])
           j ← j+1
       if (j = m)
           return i-m+1
       else
           i ← i + table[T[i]]
    end
    return -1
end.
```

HASHING

Hashing is a searching technique using the principles of space and time trade offs.

Hashing is based on the idea of distributing keys among a one-dimensional array called the Hash table. The distribution is done by computing for each of the keys the value of some predefined function h called the hash function. This function assigns an integer between 0 and m-1 called the hash address to a key.

Eg: Let an array consists of elements 0, 2, 5, 9, 7
Let the choosen hash function be

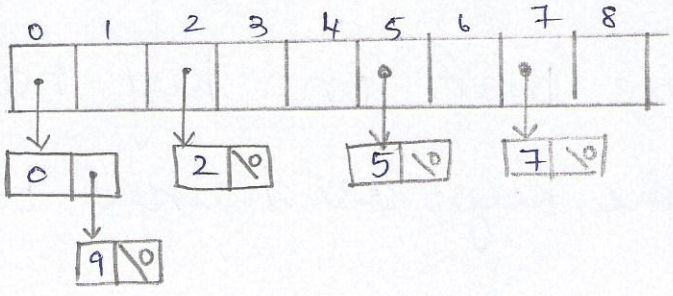
$$h(k) = k \bmod m$$

where k are the keys and ~~m~~ let $m = 9$.

Then,

$$h(0) = 0 \bmod 9 = 0$$
$$h(2) = 2 \bmod 9 = 2$$
$$h(5) = 5 \bmod 9 = 5$$
$$h(9) = 9 \bmod 9 = 0$$
$$h(7) = 7 \bmod 9 = 7$$

We construct the hash table as below



In order to search an element, the hash function is applied on the element and the hash address thus obtained is sequentially searched to find if the element is present.

In order to choose a good hash function the following conditions are to be satisfied.

(a) A hash function must be able to uniformly distribute all the keys in a hash table

(b) A hash function has to be easy to compute

Hashing can be performed in 2 ways:

(1) Open hashing

(2) Closed hashing.

OPEN HASHING:

It is a hashing technique in which keys are stored in linked lists attached to an array cell of hash table using an appropriate hash function. The hash value obtained is used to find the index of list to which the item is to be added. If more than one key has same hash value, then the keys are hashed to the same location.

When an item has to be searched

it is necessary to find the hash value of the item using the hash function. This hash value corresponds to an index which indicates the address of the key's location. If the address list is NULL then the item is not present. Else the list is searched sequentially.

ILLUSTRATION OF OPEN HASHING

Consider the following list of words

A FOOL AND HIS MONEY ARE SOON PARTED

In this example let us assign each character their position in the alphabetical order and compute the sum. Further the sum is divided by 13 and the remainder is taken as hash address. Hence, we get,

A = 1

FOOL = 6 + 15 + 15 + 12 = 48

AND = 1 + 14 + 4 = 19

HIS = 8 + 9 + 19 = 36

MONEY = 13 + 15 + 14 + 5 + 25 = 72

ARE = 1 + 18 + 5 = 24

SOON = 19 + 15 + 15 + 14 = 63

PARTED = 16 + 1 + 18 + 20 + 5 + 4 = 64

Let h(key) = key mod 13

$\therefore$ h(A) = 1 mod 13 = 1

h(FOOL) = 48 mod 13 = 9

h(AND) = 19 mod 13 = 6

$h(\text{HIS}) = 36 \bmod 13 = 10$

$h(\text{SOON})$

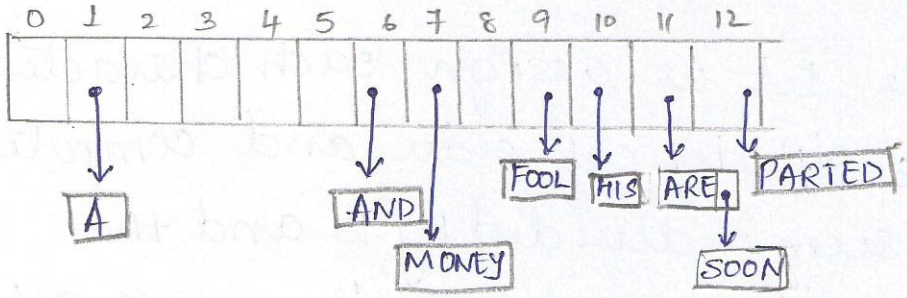
$h(\text{MONEY}) = 72 \bmod 13 = 7$

$h(\text{ARE}) = 24 \bmod 13 = 11$

$h(\text{SOON}) = 63 \bmod 13 = 11$

$h(\text{PARTED}) = 64 \bmod 13 = 12$

The hash table thus constructed is



In the above example the words ARE and SOON have the same hash address. This condition is called collision. Collision can be overcome by using singly linked list attached to the array cell. Collision must be avoided in order to obtain efficient search table. Therefore, a suitable hash function has to be obtained.

TIME EFFICIENCY

The time efficiency of hashing depends on the linked list length which in turn depends on the size of table and hash function.

If the hash function distributes n keys among m cells of the hash table almost evenly then, each linked list will be n/m keys long. The ratio $\alpha = n/m$ is called the load factor of the hash table. Then, if S is the number of successful search and U is the number of unsuccesful search we have,

$$S \approx 1 + \frac{\alpha}{2}$$

$$U = \alpha .$$

In the best case possibility if the hash function is suitably choosen then,

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

CLOSED HASHING (OPEN ADDRESSING)

In closed hashing all the keys are stored in a hash table without the use of linked lists. In order to employ collision resolution the simplest method is linear probing.

In this method using the hash function a hash address to a key is obtained. If the cell is vacant then the key is placed in that cell. Else it checks for the vaccancy of the cell's immediate successor. If that is vacant the key is filled in. Else the vaccancy is checked. Note that

if we encounter the end of hash table then the search is proceeded from the begining of the table, that is, it is treated as a circular array.

To search for a given key k, we start by computing h(k) where h is the hash function used in the table's construction. If the cell h(k) is empty, the search is unsuccessful. If the cell is not empty, we must compare k with cell's occupant: if they are equal then key is found else we compare k with the adjacent cells and continue in this manner til we encounter a the key or an empty cell.

ILLUSTRATION OF CLOSED HASHING

Let h(key) = key mod 13

KEYS	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
HASH ADDRESS	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

In the above example a collision occurs between "ARE" and "SOON". Since "ARE" is placed in cell 11 and cell 12 is vacant SOON occupies cell 12. Further, "PARTED" occupies cell 0 as it is the next cell vaccant after 12.

COLLISION

DEFINITION: If the size of the hash table is smaller than the total number of keys generated then 2 or more keys will have same hash address. This phenomenon is called collision.

This can be overcome using linked list and is called as collision resolution.