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:

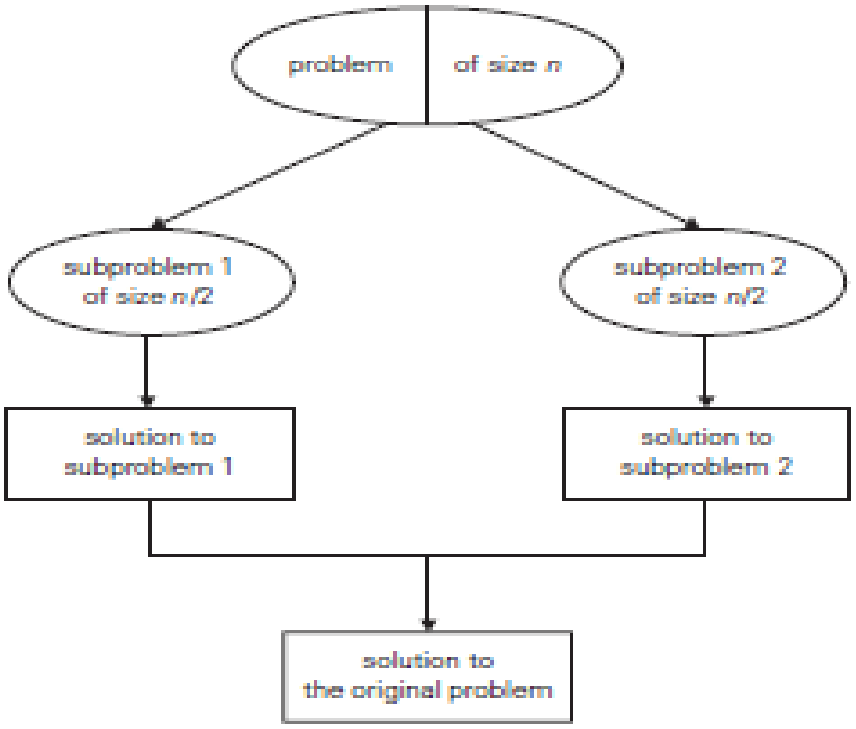
A problem's instance is divided into several smaller instances of the same problem, ideally of about the same size.

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*

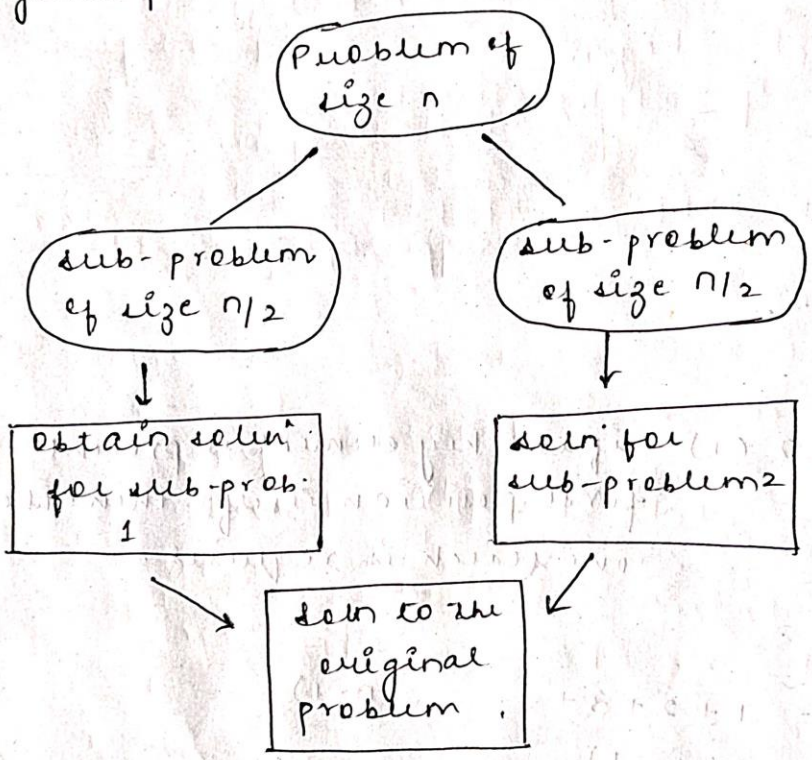
Demonstration Link:
https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/visualize/

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

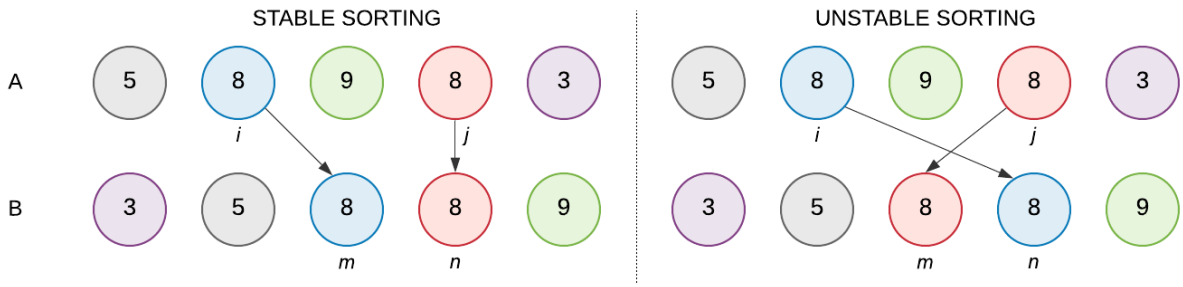
Definition:

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

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

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

$= \Theta (n \log n)$

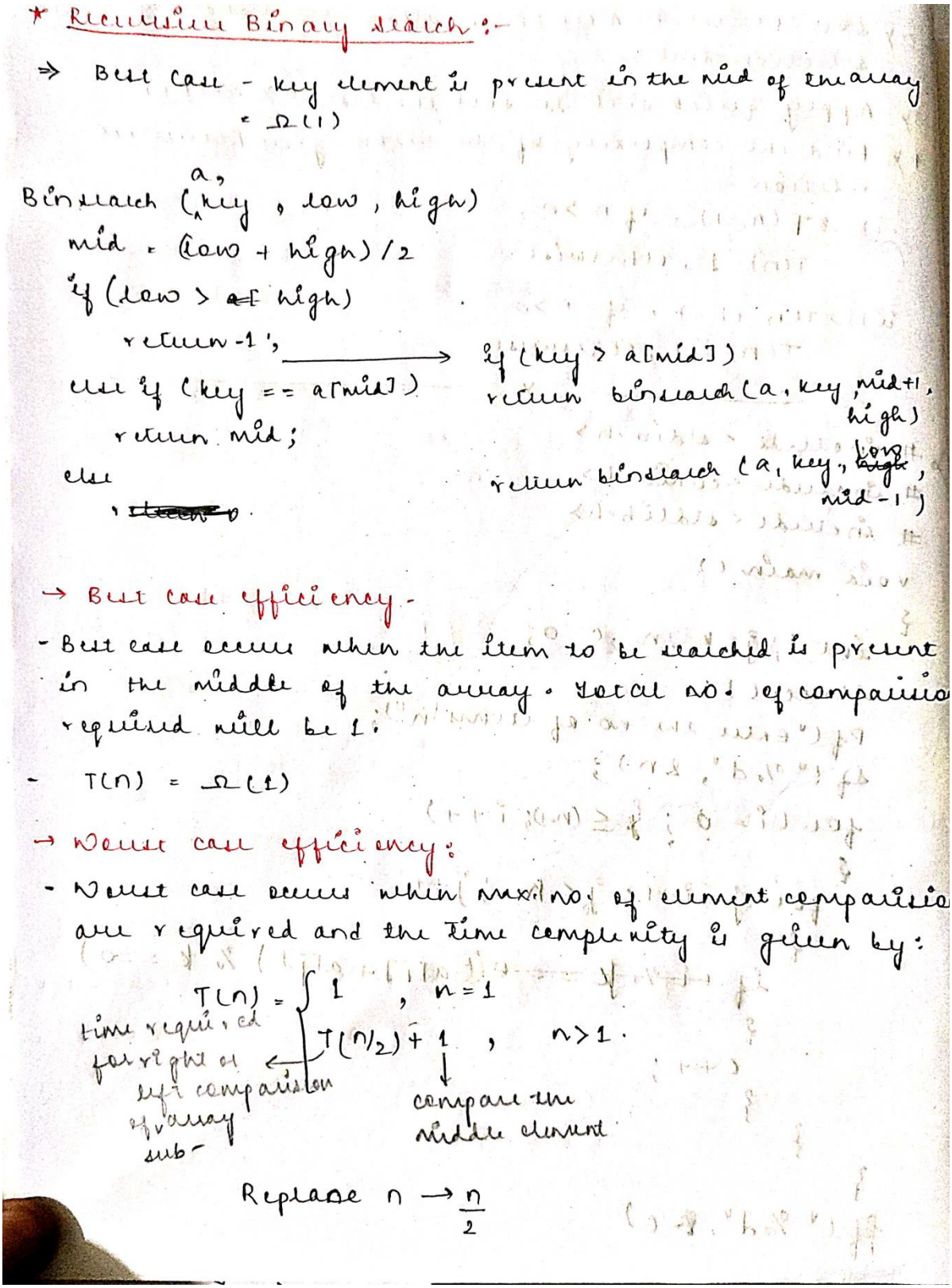
Advantages:

- 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 :

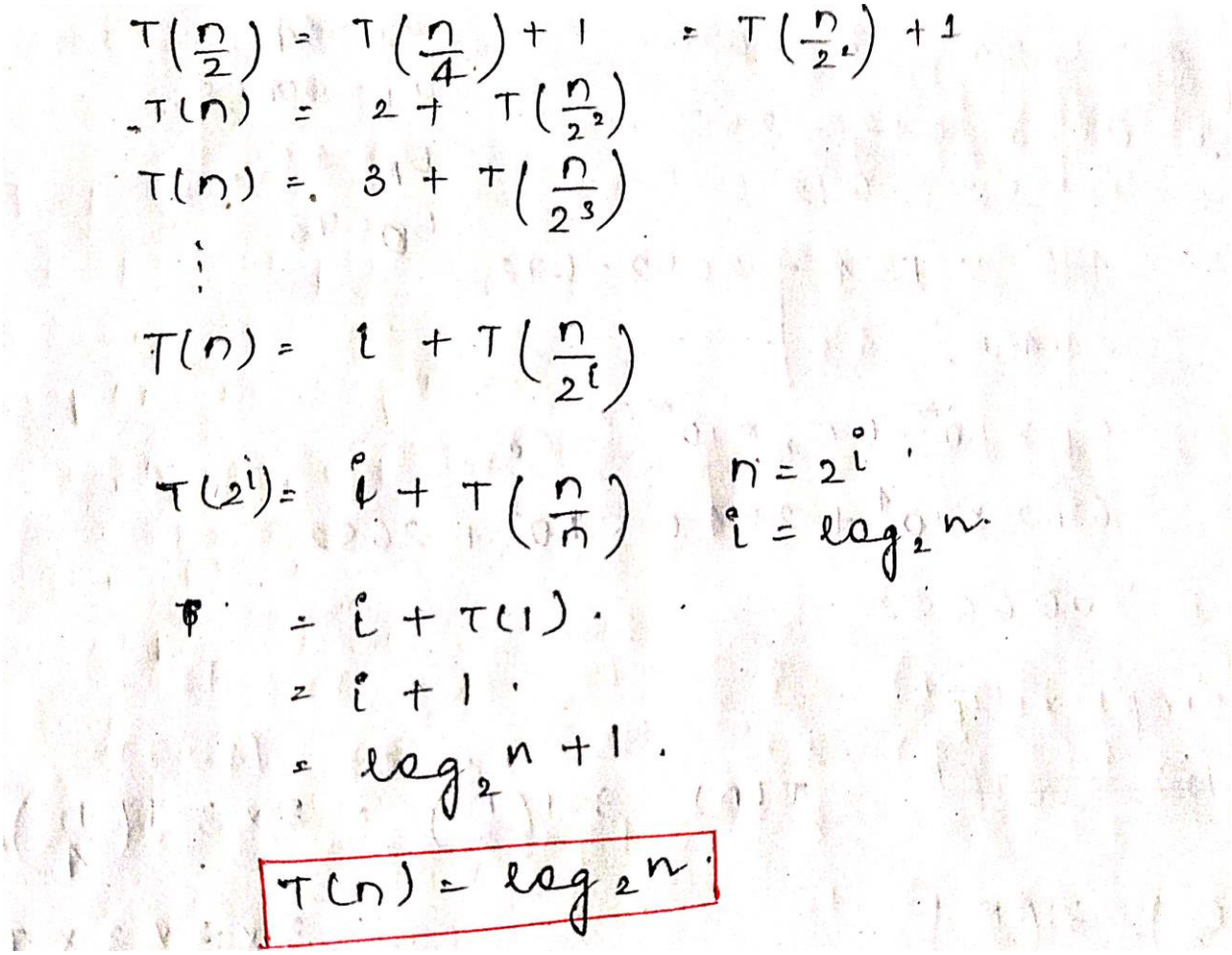
- Worst case occurs when max. no. of element comparisons are required and the time complexity is given by :

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

$T(n)$: 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

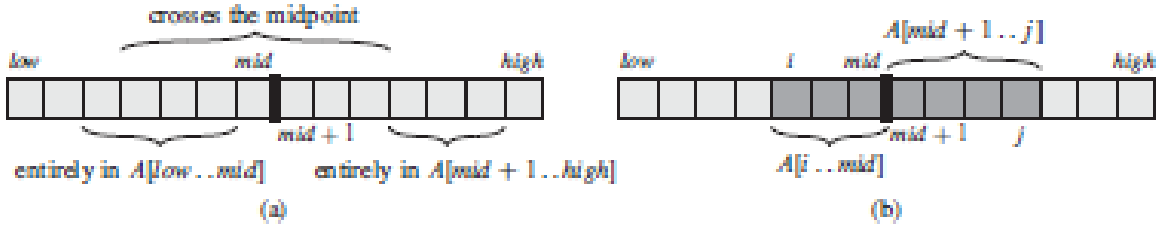
You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum.

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

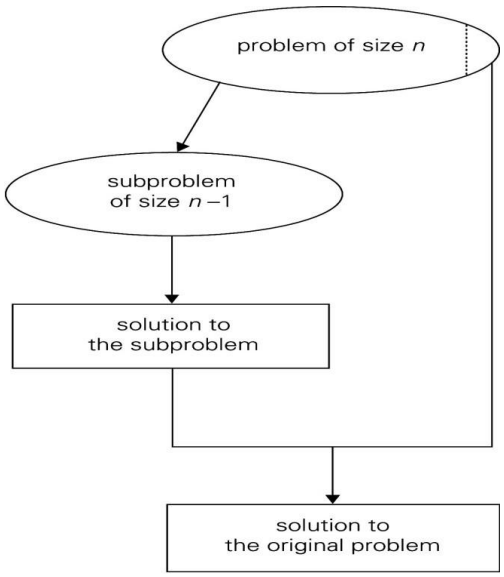
Decrease & conquer is a general algorithm design strategy based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem. The exploitation can be either top-down (recursive) or bottom-up (non-recursive).

The major variations of decrease and conquer are

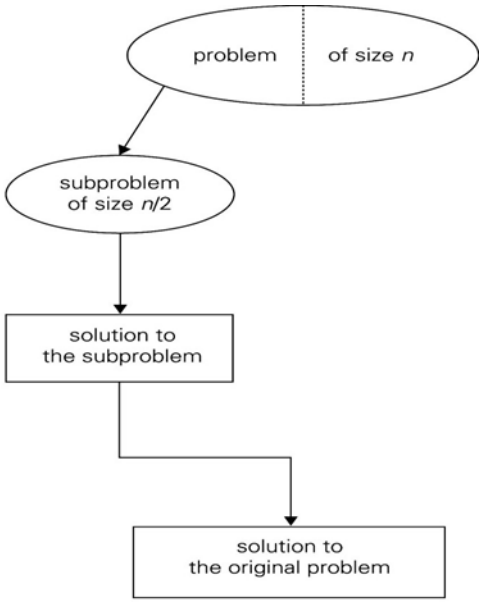
1. Decrease by a constant :(usually by 1):
    a. insertion sort
    b. graph traversal algorithms (DFS and BFS)
    c. topological sorting
    d. algorithms for generating permutations, subsets
2. Decrease by a constant factor (usually by half)
    a. binary search and bisection method
3. Variable size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

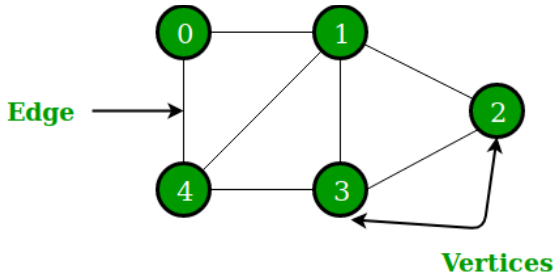
0>=0 & 20<10 (F)

A[1]=20

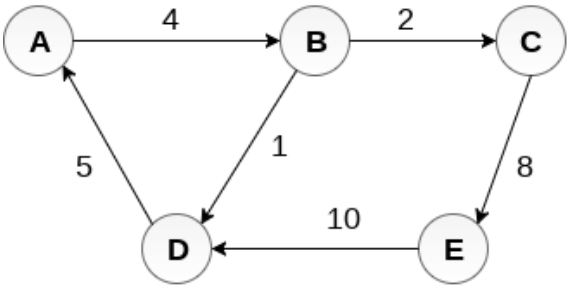
Advantages of insertion sort:

- 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



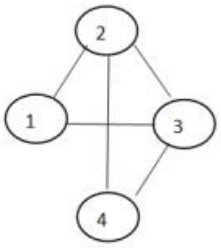
Graph



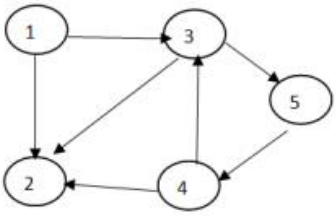
Weighted Directed Graph

	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

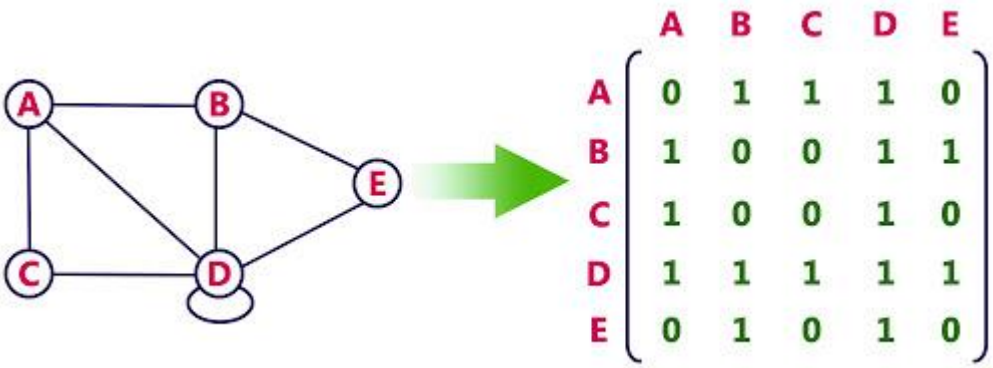
Adjacency Matrix



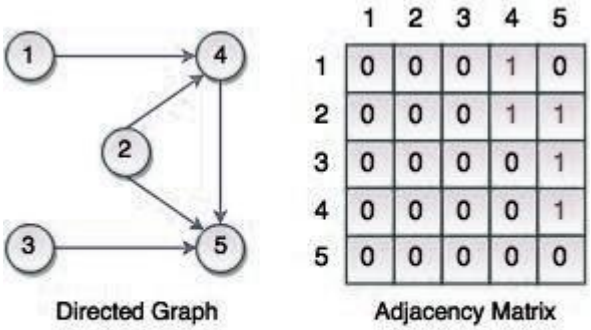
Undirected graph



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)

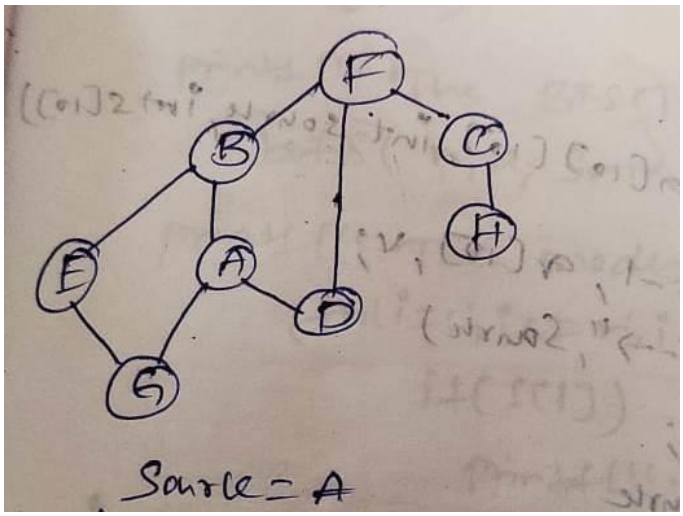
The traversal's starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called a tree edge. If an edge leading to a previously visited vertex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

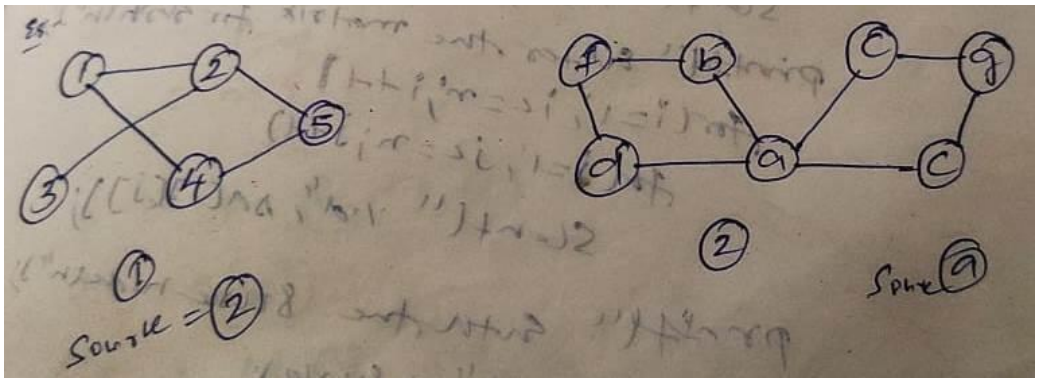
```
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = ⟨V, E⟩
//Output: Graph G with its vertices marked with consecutive integers
//        in the order they are visited by the BFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each 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);
}
```

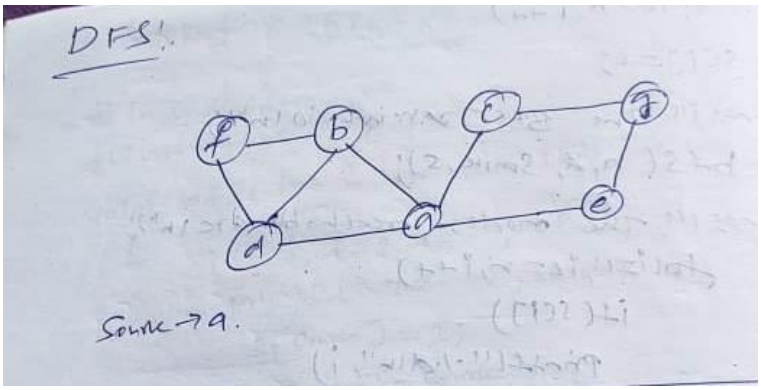
Depth First Search

- 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			



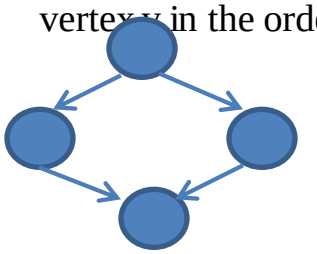
ii. #include<stdio.h>

```
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);
```

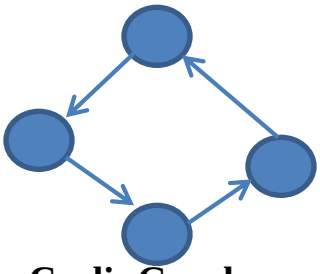
```
    printf("Enter  the adjacency matrix:\n");
    for(i=1;i<=n;i++)
            for(j=1;j<=n;j++)
                    scanf("%d",&a[i][j]);

    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

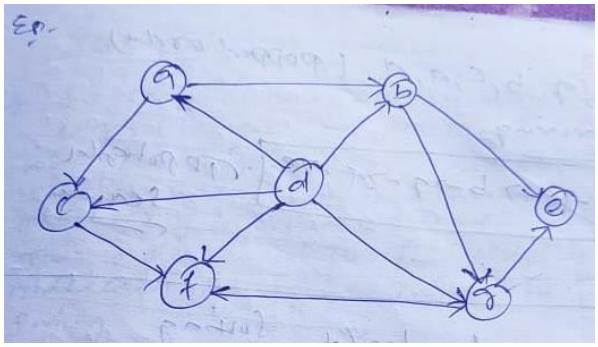
Step1: Select any arbitrary vertex

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

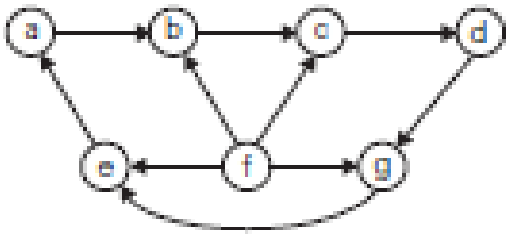
Step 3: when a vertex becomes dead it is removed from stack.

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.