UNIT-II

Divide and Conquer

General Method

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from O (n^2) to O (n log n) to sort the elements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Divide and conquer algorithm consists of two parts:

Divide : Divide the problem into a number of sub problems. The sub problems are solved recursively.

Conquer : The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide–and–conquer is a very powerful use of recursion.

Control Abstraction of Divide and Conquer

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

```
   DANDC (P)
{ if SMALL (P) then return S (p);
     else
     { divide p into smaller instances $p_1, p_2, …. P_k, k \square 1$;
          apply DANDC to each of these sub problems;
          return (COMBINE (DANDC ($p_1$) , DANDC ($p_2$),…., DANDC ($p_k$));
     }
}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. These sub problems $p_1, p_2, . . . , p_k$ are solved by recursive application of DANDC.

If the sizes of the two sub problems are approximately equal then the computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2\,T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, T (n) is the time for DANDC on 'n' inputs g (n) is the time to complete the answer directly for small inputs and
f (n) is the time for Divide and Combine

BINARY SEARCH:

The binary search method is more efficient than the linear search method. The array must be sorted for binary search, which is not necessary for linear search. Binary search Binary search is a search strategy based on divide and conquer. The method divides the list into two halves at each step and checks weather the element to be searched is on the top or lower side of the array. If the element is located in the center, the algorithm returns.

Let us assign minimum and maximum index of the array to variables *low* and *high*, respectively. The middle index is computed as (low + high)/2.
In every iteration, the algorithm compares the middle element of the array with a key to be searched. Initial range of search is A[0] to A[n – 1]. When the key is compared with A [*mid*], there are three possibilities :

1. The array is already sorted, so if key < A[mid] then key cannot be present in the bottom half of the array. Search space of problem will be reduced by setting the *high* index to (mid – 1). New search range would be A[*low*] to A[*mid* – 1].
2. If key > A[*mid*] then the key cannot be present in the upper half of the array. Search space of problem will be reduced by moving the *low* index to (mid + 1). New search range would be A[*mid* + 1] to A[*high*]
3. If key = A[*mid*], the search is successful and algorithm halts.

This process is repeated till index *low* is less than *high* or element is found.

Algorithm for Binary Search

```
Algorithm BINARY_SEARCH(A, Key)
// Description : Perform a binary search on array A
// Input : Sorted array A of size n and Key to be searched
// Output : Success / Failure

low ← 1
high ← n
while low < high do
    mid ← (low + high) / 2
    if A[mid] == key then
        return mid
    else if A[mid] < key then
        low ← mid + 1
    else
    high ← mid – 1
    end
end
return 0
```

Binary search reduces search space by half in every iteration. In a linear search, the search space was reduced by one only.

If there are n elements in an array, binary search, and linear search have to search among (n / 2) and (n – 1) elements respectively in the second iteration. In the third iteration, the binary search has to scan only (n / 4) elements, whereas linear search has to scan (n – 2) elements. This shows that a binary search would hit the bottom very quickly.

Complexity Analysis of Binary Search

Best Case:

In binary search, the key is initially compared to the array's middle element. If the key is in the center of the array, the algorithm only does one comparison, regardless of the size of the array. As a result, the algorithm's best-case running time is $T(n) = 1$.

Worst Case:

Every iteration, the binary search search space is decreased by half, allowing for maximum log2n array divisions. If the key is at the leaf of the tree or it is not present at all, then the algorithm does log2n comparisons, which is maximum. The number of comparisons increases in logarithmic proportion to the amount of the input. As a result, the algorithm's worst-case running time would be $T(n) = O(\log 2\ n)$.

The problem size is reduced by a factor of two after each iteration, and the method does one comparison. Recurrence of binary search can be written as $T(n) = T(n/2) + 1$. Solution to this recurrence leads to same running time, i.e. O(log2n). Detail derivation is discussed here:

In every iteration, the binary search does one comparison and creates a new problem of size n/2. So recurrence equation of binary search is given as,

$T(n) = T(n/2) + 1$, if $n > 1$

$T(n) = 1$, if $n = 1$

Only one comparison is needed when there is only one element in the array. That's the trivial case. This is the boundary condition for recurrence. Let us solve this by iterative approach,

$T(n) = T(n/2) + 1$ **…(1)**
Substitute n by n/2 in Equation (1) to find T(n/2)

$T(n/2) = T(n/4) + 1$ **…(2)**
Substitute value of T(n/2) in Equation (1),

$T(n) = T(n/2^2) + 1$ **…(3)**
Substitute n by n/2 in Equation (2) to find T(n/4),

$T(n/4) = T(n/8) + 1$

Substitute value of T(n/4) in Equation (3),

$T(n) = T(n/2^3) + 3$
.

.

After k iterations,

$T(n) = T(n^k) + k$
Binary tree created by binary search can have maximum height $\log_2 n$
So, $k = \log_2 n \Rightarrow n = 2^k$
$T(n) = T(2^k/2^k) + k$

= T(1) + k

From base case of recurrence,

$T(n) = 1 + k = 1 + \log_2 n$
$T(n) = O(\log_2 n)$

Average Case:

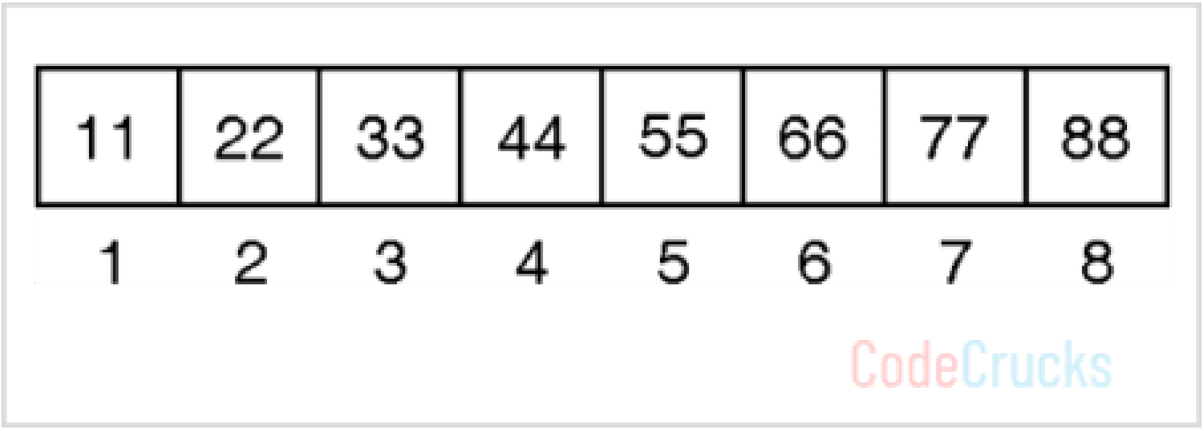
The average case for binary search occurs when the key element is neither in the middle nor at the leaf level of the search tree. On average, it does half of the $\log_2 n$ comparisons, which will turn out as $T(n) = O(\log_2 n)$.
The complexity of linear search and binary search for all three cases is compared in the following table.

	Best case	**Average case**	**Worst case**
Binary Search	O(1)	$O(\log_2 n)$	$O(\log_2 n)$
Linear Search	O(1)	O(n)	O(n)

Example of Binary Search

Example: Find element 33 from array A {11, 22, 33, 44, 55, 66, 77, 88} using binary search.
Solution:
The array is already sorted, so we can directly apply binary search on A. Here Key = 33.

Iteration 1: Initial search space



Binary Search – Step 1

low = 1, high = 8,

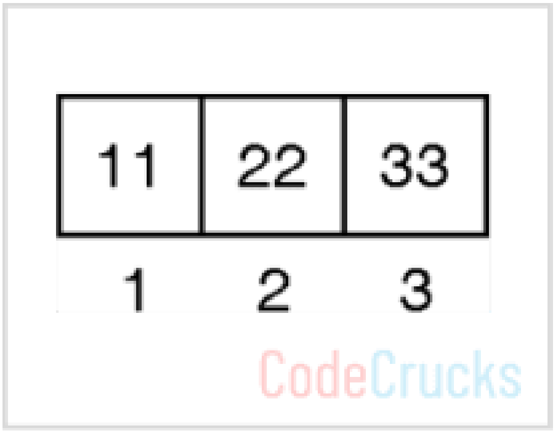
mid = low + high) / 2 = (1 + 8)/ 2 = 9/2 = 4

A[mid] = A[4] = 44

As Key < A[4], update high

high = mid – 1 = 4 – 1 = 3

Iteration 2 : New search space



low = 1, high = 3,

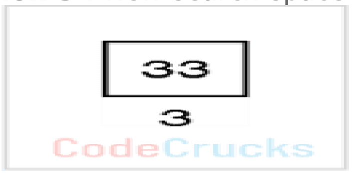
mid = (low + high) / 2 = (1 + 3)/ 2= 4/2 = 2

A[mid] = A[2] = 22

As Key > A[2], update low

low = mid + 1= 2 + 1 = 3

Iteration 3 : New search space



low = 3, high = 3,

mid = (low + high) / 2 = (3 + 3)/ 2= 6/2 = 3

A[mid] = A[3] = 33

Key = A[3], An element found, so return mid.

Max-Min Problem

Max-Min problem is to find a maximum and minimum element from the given array. We can effectively solve it using divide and conquer approach.
In the traditional approach, the maximum and minimum element can be found by comparing each element and updating Max and Min values as and when required. This approach is simple but it does (n – 1) comparisons for finding max and the same number of comparisons for finding the min. It results in a total of 2(n – 1) comparisons. Using a divide and conquer approach, we can reduce the number of comparisons.

Divide and conquer approach for Max. Min problem works in three stages.

- If a_1 is the only element in the array, a_1 is the maximum and minimum.
- If the array contains only two elements a_1 and a_2, then the single comparison between two elements can decide the minimum and maximum of them.
- If there are more than two elements, the algorithm divides the array from the middle and creates two subproblems. Both subproblems are treated as

an independent problem and the same recursive process is applied to them. This division continues until subproblem size becomes one or two.

After solving two subproblems, their minimum and maximum numbers are compared to build the solution of the large problem. This process continues in a bottom-up fashion to build the solution of a parent problem.

Algorithm for Max-Min Problem

```
Algorithm DC_MAXMIN (A, low, high)
// Description : Find minimum and maximum element from array using divide and
conquer approach
// Input : Array A of length n, and indices low = 0 and high = n - 1
// Output : (min, max) variables holding minimum and maximum element of array

if n == 1 then
    return (A[1], A[1])
else if n == 2 then
    if A[1] < A[2] then
        return (A[1], A[2])
    else
        return (A[2], A[1])
    else
    mid ← (low + high) / 2
    [LMin, LMax] = DC_MAXMIN (A, low, mid)
    [RMin, RMax] = DC_MAXMIN (A, mid + 1, high)
    if LMax > RMax  then
        // Combine solution
        max ← LMax
    else
        max ← RMax
    end
    if LMin < RMin then
        // Combine solution
        min ← LMin
    else
        min ← RMin
    end
    return (min, max)
end
```

Complexity analysis

The conventional algorithm takes $2(n – 1)$ comparisons in worst, best and average case.

DC_MAXMIN does two comparisons to determine the minimum and maximum element and creates two problems of size $n/2$, so the recurrence can be formulated as

$T(n) = 0$, if $n = 1$

$T(n) = 1$, if $n = 2$

$T(n) = 2T(n/2) + 2$, if $n > 2$

Let us solve this equation using interactive approach.

$T(n) = 2T(n/2) + 2$ **… (1)**

By substituting n by $(n / 2)$ in Equation (1)

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

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

$= 4T(n/4) + 4 + 2$ **… (2)**
By substituting n by n/4 in Equation (1),

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

Substitute it in Equation (1),

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

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

$= 2^3 T(n/2^3) + 2^3 + 2^2 + 2^1$
.

.

After k – 1 iterations

t can be observed that divide and conquer approach does only [(3n/2) – 2] comparisons compared to 2(n – 1) comparisons of the conventional approach.

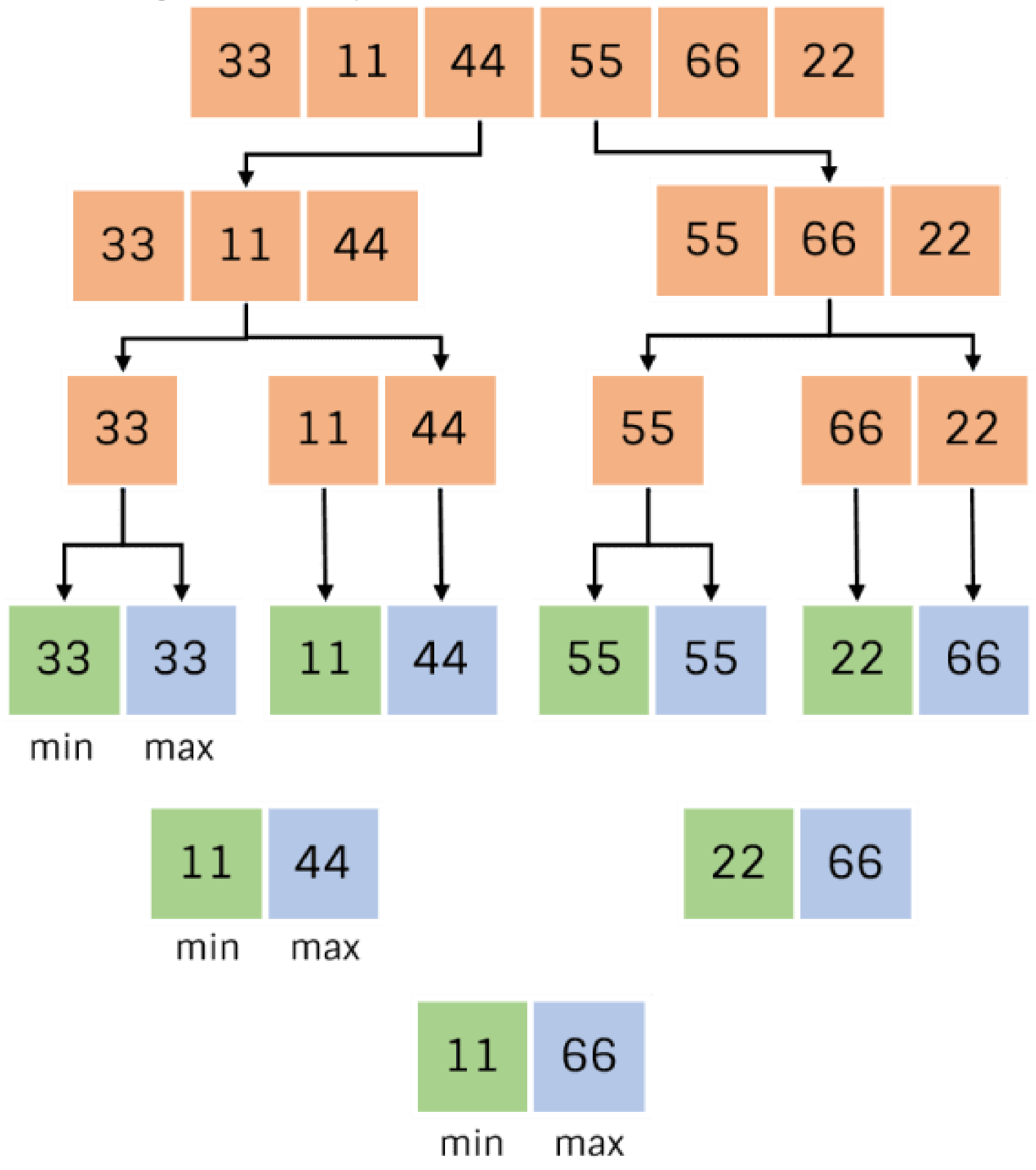
For any random pattern, this algorithm takes the same number of comparisons.

It can be observed that divide and conquer approach does only comparisons compared to 2(n – 1) comparisons of the conventional approach. For any random pattern, this algorithm takes the same number of comparisons.

Example

Problem: Find max and min from the sequence <33, 11, 44, 55, 66, 22> using divide and conquer approach
Solution:
During the divide step, the algorithm divides the array until it reaches a size of one or two. Once the array size reaches the base case, we may get the maximum and minimum number from each array recursively. This method is continued until all of the subarrays have been processed. The figure below depicts the complete procedure.

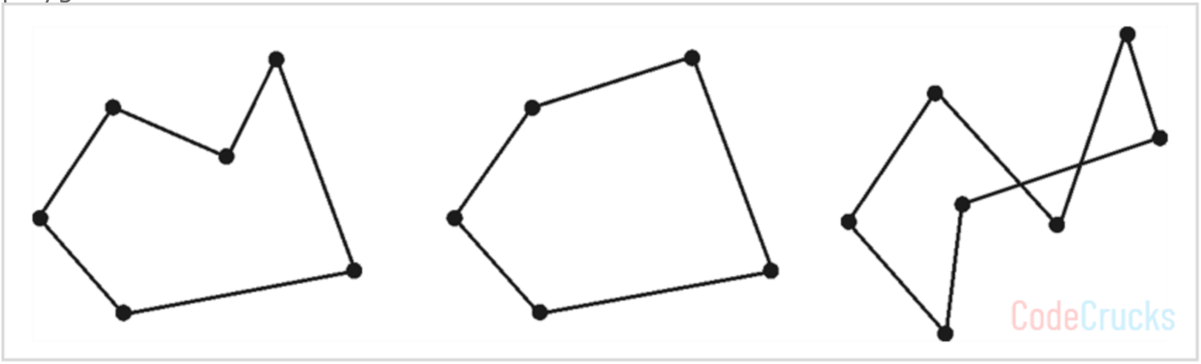
Max-min using divide and conquer method



The orange cell represents the stage of stepwise division. In the conquer stage, the smallest element from the parent arrays is placed in the green cell, while the largest element is stored in the blue cell.

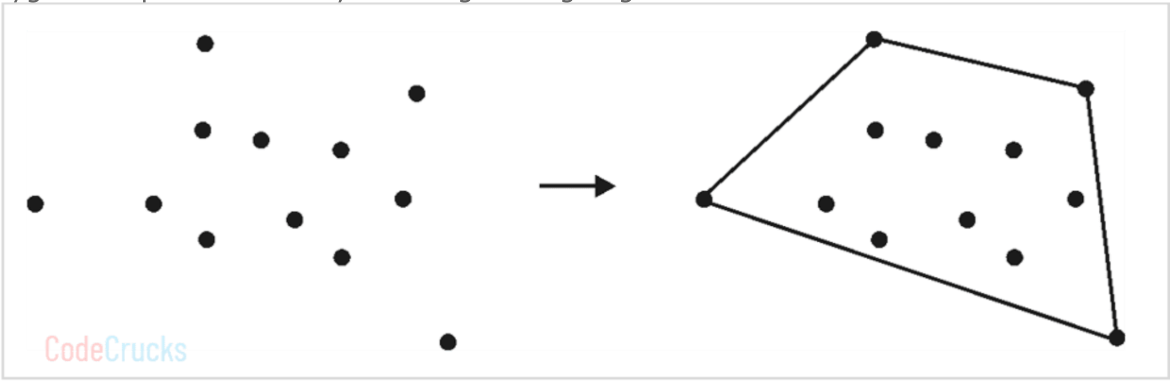
Convex Hull using Divide and Conquer

Convex hull is the smallest region covering given set of points. Polygon is called **convex polygon** if the angle between any of its two adjacent edges is always less than 180^0. Otherwise, it is called a concave polygon. Complex polygons are self-intersecting polygons.



(a)Concave polygon (b) Convex polygon (c) Complex polygon

The **convex hull** of the set of points Q is the convex polygon P that encompasses all of the points given. The problem of finding the smallest polygon P such that all the points of set Q are either on the boundary of P or inside P is known as the convex hull problem. The convex hull of the points in question is seen in following figure. The vertex of a polygon is a point shared by two neighboring edges.



Brute Force Approach

The brute force method for determining convex hull is to construct a line connecting two points and then verify whether all points are on the same side or not. There are such n(n – 1) /2 lines with n points, and each line is compared with the remaining n – 2 points to see if they fall on the same side. As a result, the brute force technique takes $O(n^3)$ time. Can we use divide and conquer more effectively?

Divide and Conquer Approach

There exist multiple approaches to solve convex hull problem. In this article, we will discuss how to solve it using divide and conquer approach.

- Sort all of the points by their X coordinates. The tie is broken by ranking points according to their Y coordinate.
- Determine two extreme points A and B, where A represents the leftmost point and B represents the rightmost point. A and B would be the convex hull's vertices. Lines AB and BA should be added to the solution set.
- Find the point C that is the farthest away from line AB.
- Calculate the convex hull of the points on the line AC's right and left sides. Remove line AB from the original solution set and replace it with AC and CB.
- Process the points on the right side of line BA in the same way.
- Find the convex hull of the points on the left and right of the line connecting the two farthest points of that specific convex hull recursively.

Algorithm of Convex Hull

Algorithm for finding convex hull using divide and conquer strategy is provided below:

```
Algorithm ConvexHull(P)
// P is a set of input points

Sort all the points in P and find two extreme points A and B
S1 ← Set of points right to the line AB
S2 ← Set of points right to the line BA
Solution ← AB followed by BA

Call FindHull(S1, A, B)
Call FindHull(S2, B, A)
```

Algorithm for subroutine FindHull is described below :

```
Algorithm FindHull(P, A, B)

if isEmpty(P) then
   return
else
   C ← Orthogonally farthest point from AB
   Solution ← Replace AB by AC followed by CB
   Partition P – { C } in X0, X1 and X2
   Discard X0 in side triangle

   Call FindHull(X1, A, C)
   Call FindHull(X2, C, B)
end
```

Complexity analysis

Pre-processing step is to sort the points according to increasing order of their X coordinate. Sorting can be done in O(nlog$_2$n) time. Finding two farthest points from the sorted list takes O(1) time. Dividing points into two halves S$_1$ and S$_2$ take O(1) time by joining A and B. In the average case, S$_1$ and S$_2$ contain half of the points. So, recursively computing the convex hull of A and B takes T(n/2) each. Merging of two convex hulls is done in linear time O(n), by finding the orthogonally farthest point. Total running time after preprocessing the points is given by,

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

$= 2T(n/2) + n$ … (1)

Solving original recurrence for n/2,

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

Substituting this in equation (1),

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

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

.

.

.

After k substitutions,

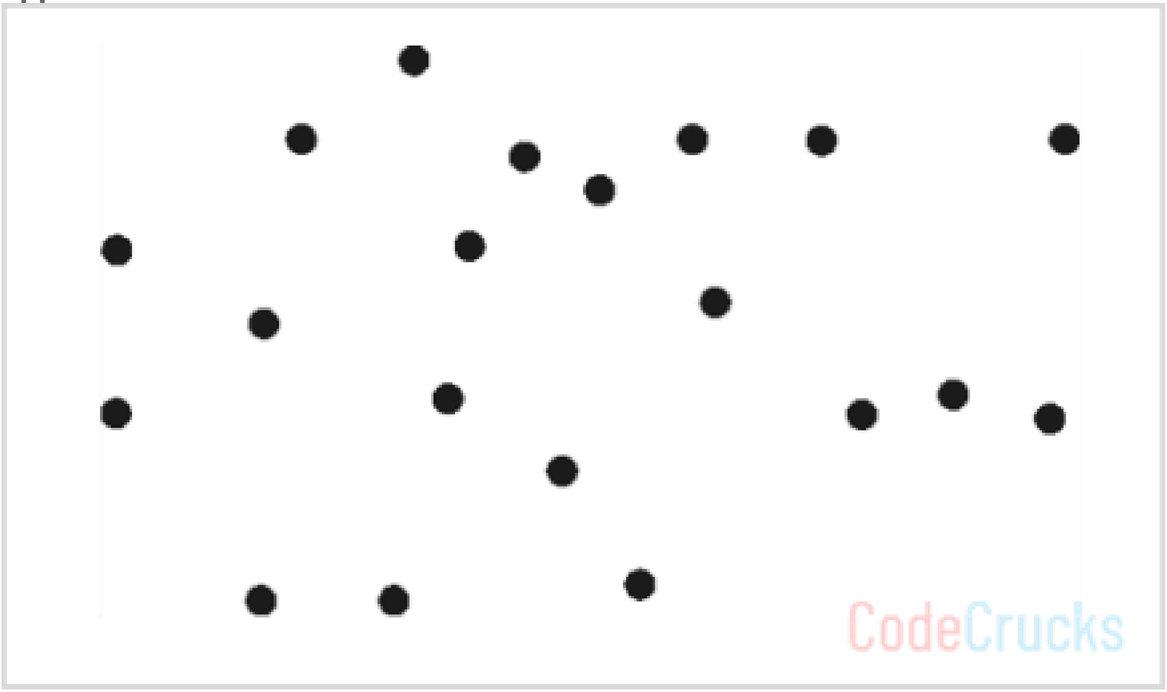
$T(n) = 2^k\ T(n/2^k) + k.n$ … (2)

Division of array creates binary tree, which has height $\log_2 n$, so let us consider that k grows up to $\log_2 n$,
$k = \log_2 n \Rightarrow n = 2^k$
Substitute these values in equation (2)

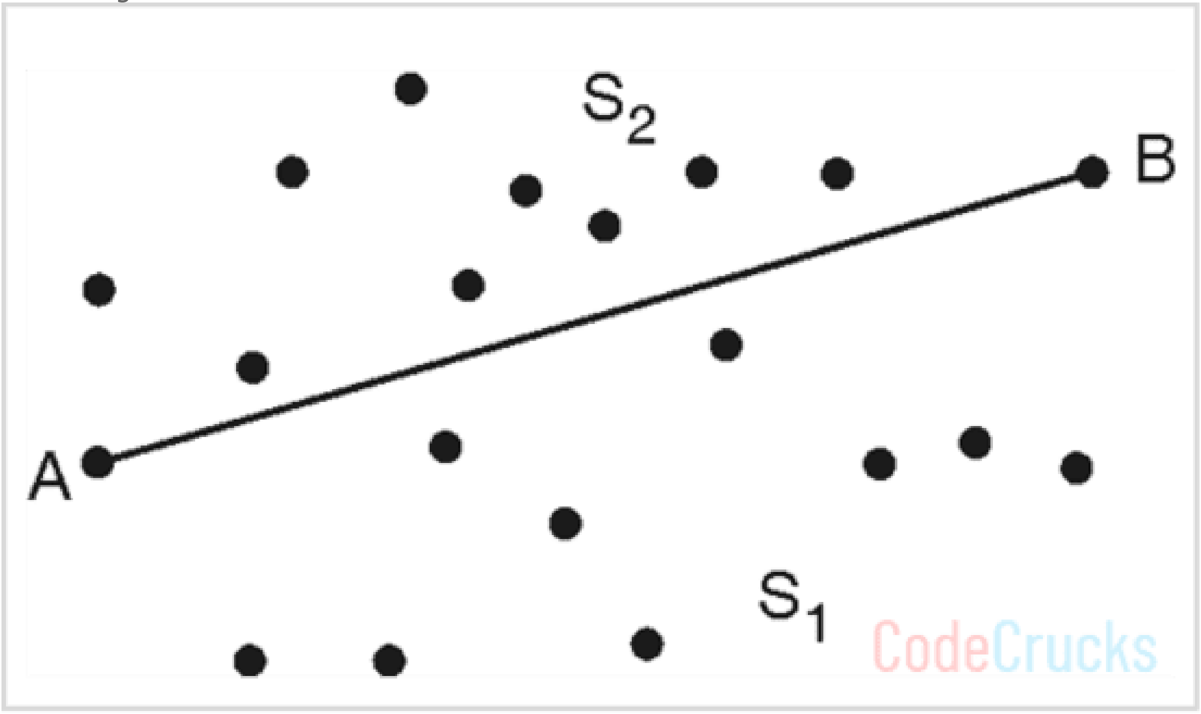
$T(n) = n.T(n/n) + \log_2 n \cdot n$
$T(n) = O(n.\log_2 n)$

Example of Convex Hull

Problem: Find the convex hull for a given set of points using divide and conquer approach.

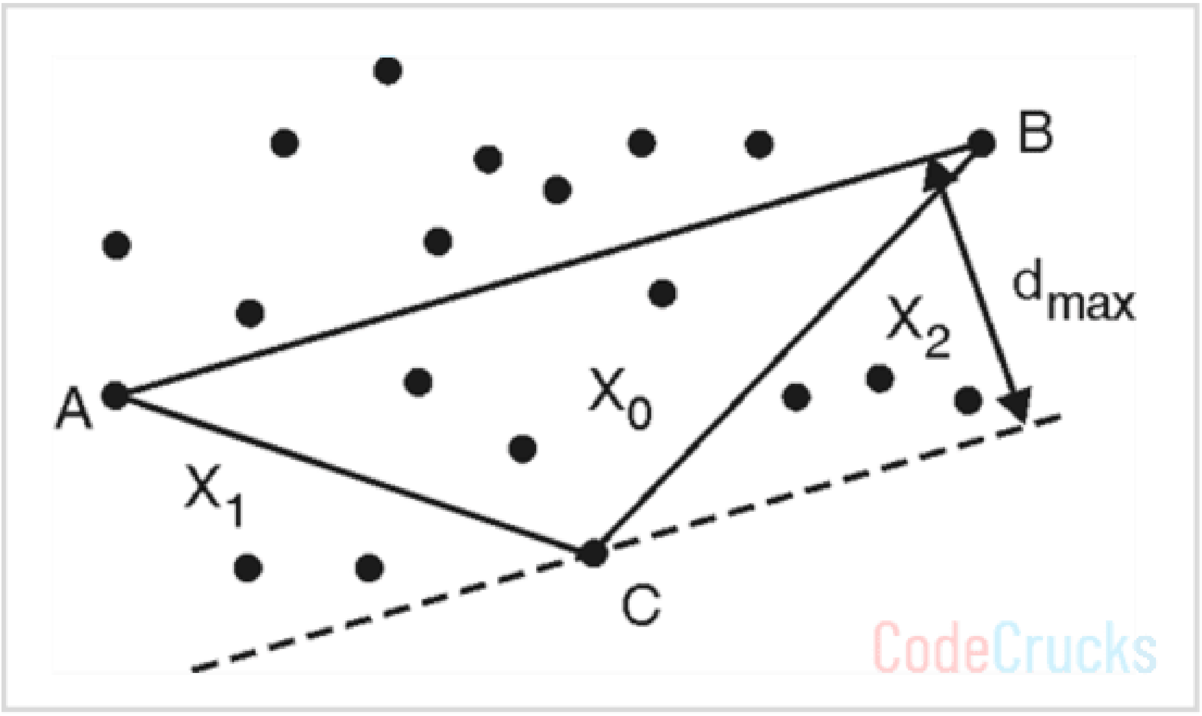


Solution:
Step 1: According to the algorithm, find left most and rightmost points from the set P and label them as A and B. Label all the points on the right of AB as S_1 and all the points on the right of BA as S_2.



Solution = {AB, BA}

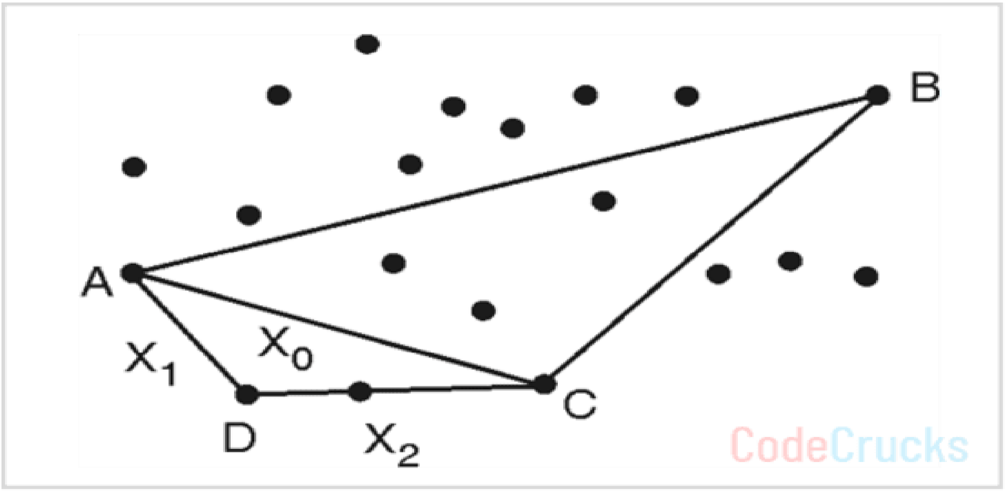
Make a recursive call to FindHull (S_1, A, B) and FindHull(S_2 ,B, A)
Step 2 : FindHull(S_1 ,A, B)
Find point C orthogonally farthest from line AB



Solution = Solution – { AB } ∪ {AC, CB}

= {AC, CB, BA}

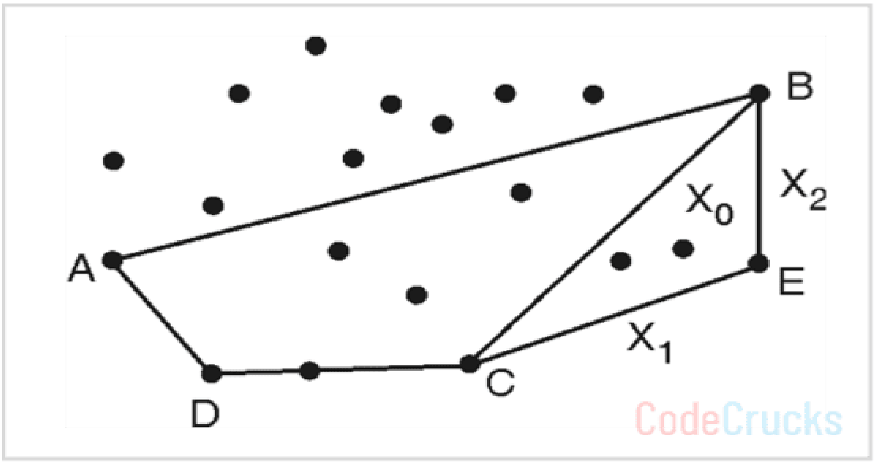
Label regions X_0, X_1 and X_2 as shown in above figure
Make recursive calls: FindHull (X_1, A, C) and FindHull (X_2, C, B)
Step 3 : FindHull(X_1, A, C)
Find point D orthogonally farthest from line AC



Solution = Solution – {AC} ∪ {AD, DC}

= {AD, DC, CB, BA}

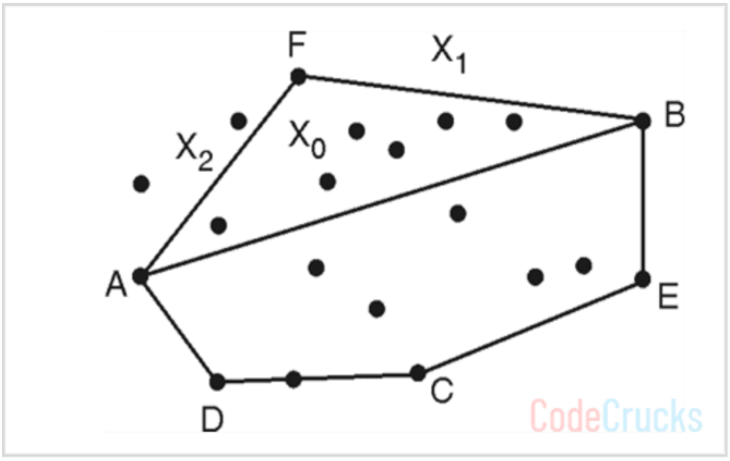
Label regions X_0, X_1 and X_2 as shown in above figure
Make recursive calls: FindHull (X_1, A, D) and FindHull (X_2, D, C)
But X_1 and X_2 sets are empty, so algorithm returns
Step 4 : FindHull(X_2, C, B)
Find point E orthogonally farthest from line CB



Solution = Solution – {CB} ∪ {CE, EB}

= {AD, DC, CE, EB, BA}

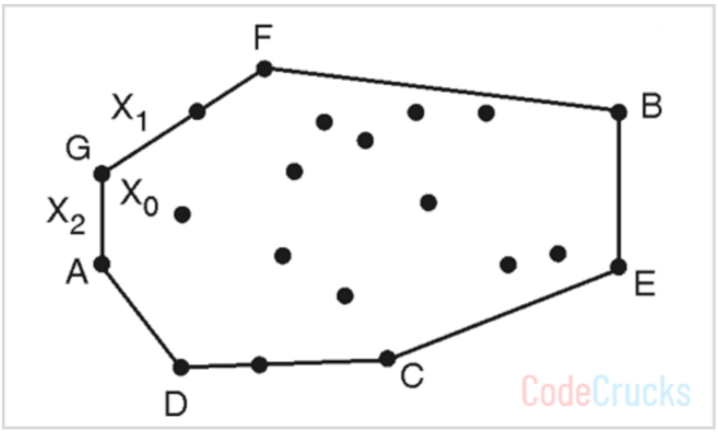
Label regions X_0, X_1 and X_2 as shown in Fig. P.3.6.1(d).
Make recursive calls: FindHull (X_1, C, E) and FindHull (X_2, E, B).
But X_1 and X_2 sets are empty, so algorithm returns Now we will explore the points in S2, on the right-hand side of the line BA
Step 5 : FindHull(S_2 ,B, A)
Find point F orthogonally farthest from line BA



Solution = Solution – {BA} ∪ {BF, FA}

= {AD, DC, CE, EB, BF, FA}

Label regions X_0, X_1 and X_2 as shown in above figure.
Make recursive calls : FindHull (X_1, B, F) and FindHull (X_2, F, A)
But X_1 set is empty, so call to FindHull (X_1, B, F) returns
Step 6 : FindHull (X_2, F, A)
Find point G orthogonally farthest from line FA



Solution = Solution – {FA} ∪ {FG, GA}

= {AD, DC, CE, EB, BF, FG, GA}

Label regions X_0,
Collision avoidance: If an automobile's convex hull avoids collisions with obstructions, so does the car. Because calculating collision-free routes is considerably easier with a convex vehicle, it is frequently used to design paths.

The lowest area rectangle that encloses a polygon has at least one side flush with the polygon's convex hull, and hence the hull is computed in the first step of minimum rectangle methods. Finding the smallest three-dimensional box enclosing an item is also dependent on the 3D-convex hull.

X_1 and X_2 as shown in last figure
Make recursive calls: FindHull (X_1, F, G) and FindHull (X_2, G, A).
But X_1 and X_2 sets are empty, so algorithm returns. And no more recursive calls are left.
So polygon with edges (AD, DC, CE, EB, BF, FG, GA) is the convex hull of given points.

Merge Sort

Merge sort algorithm is a classic example of divide and conquer. To sort an array, recursively, sort its left and right halves separately and then merge them. The time complexity of merge mort in the *best case, worst case* and *average case* is O(n log n) and the number of comparisons used is nearly optimal.

This strategy is so simple, and so efficient but the problem here is that there seems to be no easy way to merge two adjacent sorted arrays together in place (The result must be build up in a separate array).

The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list.

The basic merging algorithm takes two input arrays 'a' and 'b', an output array 'c', and three counters, *a ptr, b ptr* and *c ptr,* which are initially set to the beginning of their respective arrays. The smaller of *a[a ptr]* and *b[b ptr]* is copied to the next entry in 'c', and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to 'c'.

To illustrate how merge process works. For example, let us consider the array ‘a’ containing 1, 13, 24, 26 and ‘b’ containing 2, 15, 27, 38. First a comparison is done between 1 and 2. 1 is copied to ‘c’. Increment *a ptr* and *c ptr.*

1	*2*	*3*	*4*
1	13	24	26
h pt			

5	*6*	*7*	*8*
2	15	27	28
i pt			

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1							
i pt							

and then 2 and 13 are compared. 2 is added to ‘c’. Increment *b ptr* and *c ptr.*

1	*2*	*3*	*4*
1	13	24	26
	h pt		

5	*6*	*7*	*8*
2	15	27	28
i pt			

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2						
	i pt						

then 13 and 15 are compared. 13 is added to ‘c’. Increment *a ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
	h pt		

5	*6*	*7*	*8*
2	15	27	28
	i pt		

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13					
		i pt					

24 and 15 are compared. 15 is added to ‘c’. Increment *b ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
		h pt	

5	*6*	*7*	*8*
2	15	27	28
	i pt		

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15				
			i pt				

24 and 27 are compared. 24 is added to ‘c’. Increment *a ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
		h pt	

5	*6*	*7*	*8*
2	15	27	28
		i pt	

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15	24			
				i pt			

26 and 27 are compared. 26 is added to ‘c’. Increment *a ptr* and *c ptr.*

1	*2*	*3*	*4*
1	13	24	26
			h pt

5	*6*	*7*	*8*
2	15	27	28
		i pt	

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15	24	26		
					i pt		

As one of the lists is exhausted. The remainder of the b array is then copied to ‘c’.

1	2	3	4
1	13	24	26

h
pt

5	6	7	8
2	15	27	28

i
pt

1	2	3	4	5	6	7	8
1	2	13	15	24	26	27	28

i
pt

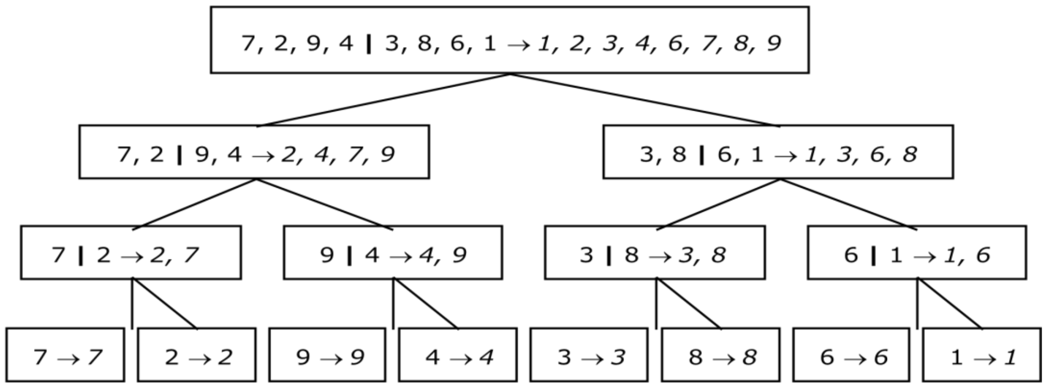
Algorithm

```
    Algorithm MERGESORT (low, high)
// a (low : high) is a global array to be sorted.
{ if (low < high)
        { mid := □(low  + high)/2□      //finds where to split the set
                MERGESORT(low,  mid)                //sort one subset
                MERGESORT(mid+1, high) //sort the other subset
                MERGE(low, mid, high)               // combine the results
        }
}
```

```
Algorithm MERGE (low, mid, high)
// a (low : high) is a global array containing two sorted subsets //
in a (low : mid) and in a (mid + 1 : high).
// The objective is to merge these sorted sets into single sorted //
set residing in a (low : high). An auxiliary array B is used.
{ h :=low; i := low; j:= mid + 1; while
        ((h ≤ mid) and (J ≤ high)) do
        { if (a[h] ≤ a[j]) then
                {    b[i] := a[h]; h := h + 1;
                }  else
                {
                        b[i] :=a[j]; j := j + 1;
                } i := i +
                1;
        }
        if (h > mid) then for k := j
                to high do
                { b[i] := a[k]; i := i + 1;
                } else for k := h to
        mid do
                { b[i] := a[K]; i := i + l;
                } for k := low
        to high do a[k] :=
        b[k];
}
```

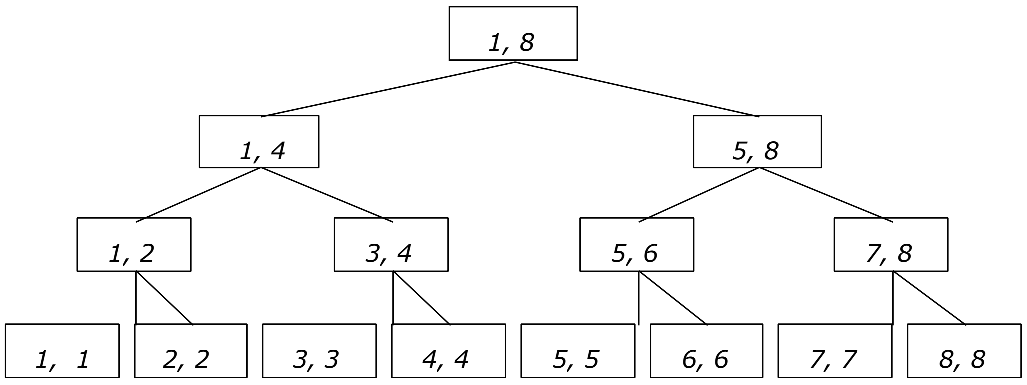
Example

For example let us select the following 8 entries 7, 2, 9, 4, 3, 8, 6, 1 to illustrate merge sort algorithm:


7, 2, 9, 4 | 3, 8, 6, 1 → 1, 2, 3, 4, 6, 7, 8, 9
7, 2 | 9, 4 → 2, 4, 7, 9
3, 8 | 6, 1 → 1, 3, 6, 8
7 | 2 → 2, 7
9 | 4 → 4, 9
3 | 8 → 3, 8
6 | 1 → 1, 6
7 → 7
2 → 2
9 → 9
4 → 4
3 → 3
8 → 8
6 → 6
1 → 1

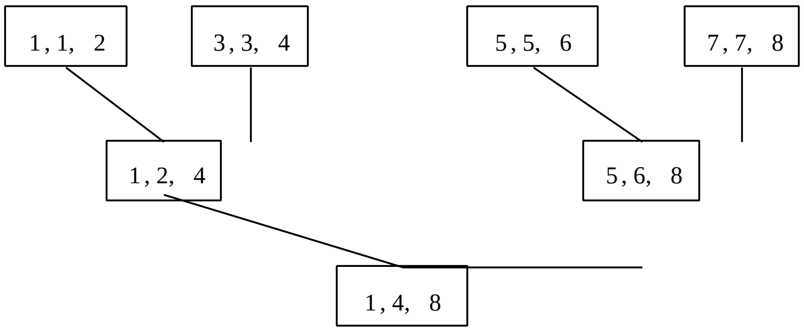
Tree Calls of MERGESORT(1, 8)

The following figure represents the sequence of recursive calls that are produced by MERGESORT when it is applied to 8 elements. The values in each node are the values of the parameters low and high.



Tree Calls of MERGE()

The tree representation of the calls to procedure MERGE by MERGESORT is as follows:



Analysis of Merge Sort

We will assume that ‘n’ is a power of 2, so that we always split into even halves, so we solve for the case $n = 2^k$.

For n = 1, the time to merge sort is constant, which we will be denote by 1. Otherwise, the time to merge sort ‘n’ numbers is equal to the time to do two recursive merge sorts of size n/2, plus the time to merge, which is linear. The equation says this exactly:

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

This is a standard recurrence relation, which can be solved several ways. We will solve by substituting recurrence relation continually on the right–hand side.

We have, $T(n) = 2T(n/2) + n$

Since we can substitute n/2 into this main equation

$$\begin{aligned} 2\,T(n/2) &= 2\,(2\,(T(n/4)) + n/2) \\ &= 4\,T(n/4) + n \end{aligned}$$

We have,

$$\begin{aligned} T(n/2) &= 2\,T(n/4) + n \\ T(n) &= 4\,T(n/4) + 2n \end{aligned}$$

Again, by substituting n/4 into the main equation, we see that

$$\begin{aligned} 4T\,(n/4) &= 4\,(2T(n/8)) + n/4 \\ &= 8\,T(n/8) + n \end{aligned}$$

So we have,

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

Continuing in this manner, we obtain:

$$T(n) = 2^k\,T(n/2^k) + K.\,n$$

As $n = 2^k$, $K = \log_2 n$, substituting this in the above equation

T (*n*) □ 2log 2*n* □□ *k*2□ *k* □
□ □ log2 *n* .
n T
2 □
□ □

$$\begin{aligned} &= n\,T(1) + n \log n \\ &= n \log n + n \end{aligned}$$

Representing this in O notation:

T(n) = **O(n log n)**

We have assumed that $n = 2^k$. The analysis can be refined to handle cases when 'n' is not a power of 2. The answer turns out to be almost identical.

Although merge sort's running time is O(n log n), it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably. *The Best and worst case time complexity of Merge sort is O(n log n).*

Strassen's Matrix Multiplication:

The matrix multiplication of algorithm due to Strassens is the most dramatic example of divide and conquer technique (1969).

The usual way to multiply two n x n matrices A and B, yielding result matrix 'C' as follows :

```
    for i := 1 to n do for j :=1 to n do c[i, j] := 0; for
K: = 1 to n do c[i, j] := c[i, j] + a[i, k] * b[k, j];
```

This algorithm requires n^3 scalar multiplication's (i.e. multiplication of single numbers) and n^3 scalar additions. So we naturally cannot improve upon.

We apply divide and conquer to this problem. For example let us considers three multiplication like this:

☐A 11 A 12 ☐ ☐B 11 B 12 ☐C 11 C 12
☐ A A ☐ ☐ B ☐□ ☐ ☐ ☐☐
B C C
☐ 21 22 ☐ ☐ 21 22 ☐ ☐ 21 22 ☐

Then c_{ij} can be found by the usual matrix multiplication algorithm,

C11 = A11 . B11 + A12 . B21

C12 = A11 . B12 + A12 . B22

C21 = A21 . B11 + A22 . B21

C22 = A21 . B12 + A22 . B22

This leads to a divide–and–conquer algorithm, which performs nxn matrix multiplication by partitioning the matrices into quarters and performing eight (n/2)x(n/2) matrix multiplications and four (n/2)x(n/2) matrix additions.

$$T(1) = 1$$
$$T(n) = 8\,T(n/2)$$

Which leads to T (n) = O (n^3), where n is the power of 2.

Strassens insight was to find an alternative method for calculating the C_{ij}, requiring seven (n/2) x (n/2) matrix multiplications and eighteen (n/2) x (n/2) matrix additions and subtractions:

$P = (A_{11} + A_{22})\,(B_{11} + B_{22})$

$Q = (A_{21} + A_{22})\,B_{11}$

$R = A_{11}\,(B_{12} - B_{22})$

$S = A_{22}\,(B_{21} - B_{11})$

$T = (A_{11} + A_{12})\,B_{22}$

$U = (A_{21} - A_{11})\,(B_{11} + B_{12})$

$V = (A_{12} - A_{22}) (B_{21} + B_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U.$

This method is used recursively to perform the seven (n/2) x (n/2) matrix multiplications, then the recurrence equation for the number of scalar multiplications performed is:

$T(1) = 1$
$T(n) = 7\ T(n/2)$

Solving this for the case of $n = 2^k$ is easy:

$T(2^k) = 7\ T(2^{k-1})$

$= 72\ T(2k-2)$

$= - - - - -$
$= - - - - -$

$= 7i\ T(2k-i)$

Put i = k

$= 7^k\ T(1)$

$= 7^k$

That is, $T(n) = 7^{\log_2 n}$

$= n \log 7\ _2$

$= O(n\ )^{\log_2 7} = O(^2 n^{.81})$

So, concluding that Strassen's algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until 'n' revolves the hundreds.

Quick Sort

The main reason for the slowness of Algorithms like SIS is that all comparisons and exchanges between keys in a sequence $w_1, w_2, \ldots, w_n$ take place between adjacent pairs. In this way it takes a relatively long time for a key that is badly out of place to work its way into its proper position in the sorted sequence.

Hoare his devised a very efficient way of implementing this idea in the early 1960's that improves the O(n^2) behavior of SIS algorithm with an expected performance that is O(n log n).

In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value.

The chosen value is known as the *pivot element*. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

- Repeatedly increase the pointer 'i' until a[i] >= pivot.
- Repeatedly decrease the pointer 'j' until a[j] <= pivot.
- If j > i, interchange a[j] with a[i]
- Repeat the steps 1, 2 and 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and place pivot element in 'j' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

- It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.
- Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . .x[high].
- It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).
- It calls itself recursively to sort the right sub-array x[j+1], x[j+2], ... x[high] between positions j+1 and high.

Algorithm Algorithm

```
 QUICKSORT(low, high)
/* sorts the elements a(low), . . . . . , a(high) which reside in the global array
         A(1 : n) into ascending order a (n + 1) is considered to be defined and must be
greater than all elements in a(1 : n); A(n + 1) = + □ */
    { if low < high then
          { j := PARTITION(a, low, high+1);
```

```
                              // J is the position of the partitioning element
            QUICKSORT(low, j – 1);
            QUICKSORT(j + 1 , high);
        }
}
```

```
    Algorithm PARTITION(a, m, p)
{
        V ☐  a(m); i ☐  m; j ☐ p;         // A (m) is the partition element do
        { loop  i  := i   + 1  until  a(i) ≥ v // i moves left to right loop  j  := j  –
            1   until   a(j)   ≤ v  //  p  moves  right  to  left  if  (i  <  j)  then
            INTERCHANGE(a, i, j)
         } while (i ≥ j); a[m] :=a[j]; a[j] :=V; // the partition element
        belongs at position P return j;
}
```

```
    Algorithm INTERCHANGE(a, i, j)
{
        P:=a[i];
        a[i] :=
        a[j]; a[j]
        := p;
}
```

Example

Select first element as the pivot element. Move 'i' pointer from left to right in search of an element larger than pivot. Move the 'j' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped. This process continues till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and interchange pivot and element at 'j' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				i						j			swap i & j
				04						79			
					i			j					swap i & j
					02			57					
						j	i						

(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	swap pivot & j
pivot					j, i								swap pivot & j
(02	08	16	06	04)	**24**								
pivot, j	i												swap pivot & j
02	(08	16	06	04)									
	pivot	i		j									swap i & j
		04		16									
			j	i									
	(06	04)	**08**	(16)									swap pivot & j
	pivot, j	i											
	(04)	**06**											swap pivot & j
	04 pivot, j, i												
				16 pivot, j, i									
(02	**04**	**06**	**08**	**16**	**24)**	38							
							(56	57	58	79	70	45)	

							pivot	i				j	swap i & j
								45				57	
								j	i				
							(45)	**56**	(58	79	70	57)	swap pivot & j
							45 pivot, j, i						swap pivot & j
									(58 pivot	79 i	70	57) j	swap i & j
										57		79	
										j	i		
									(57)	**58**	(70	79)	swap pivot & j
									57 pivot, j, i				
											(70	79)	

											pivot, j	i	swap pivot & j
											70		
												79 ivot, j, i	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

Analysis of Quick Sort:

Like merge sort, quick sort is recursive, and hence its analysis requires solving a recurrence formula. We will do the analysis for a quick sort, assuming a random pivot (and no cut off for small files).

We will take T (0) = T (1) = 1, as in merge sort.

The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (The pivot selection takes only constant time). This gives the basic quick sort relation:

$$T(n) = T(i) + T(n - i - 1) + C\,n \qquad - \qquad (1)$$

Where, $i = |S_1|$ is the number of elements in S_1.

Worst Case Analysis

The pivot is the smallest element, all the time. Then i=0 and if we ignore T(0)=1, which is insignificant, the recurrence is:

$$T(n) = T(n - 1) + C\,n \qquad n > 1 \qquad - \qquad (2)$$

Using equation – (1) repeatedly, thus

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

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

- - - - - - - -

$$T(2) = T(1) + C(2)$$

Adding up all these equations yields

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

= **O (n²)** - (3)

Best Case Analysis

In the best case, the pivot is in the middle. To simply the math, we assume that the two sub-files are each exactly half the size of the original and although this gives a slight over estimate, this is acceptable because we are only interested in a Big – oh answer.

T (n) = 2 T (n/2) + C n - (4)

Divide both sides by n

T(*n*) *T*(*n* / 2)

____ □ □ *C* - (5) *n* *n* / 2

Substitute n/2 for ‘n’ in equation (5)

T(*n* / 2) *T*(*n* / 4)

______ □ □ *C* - (6) *n* / 2 *n* / 4

Substitute n/4 for ‘n’ in equation (6)

T(*n* / 4) *T*(*n* / 8)

______ □ □ *C* - (7) *n* / 4 *n* / 8

- - - - - - - -

- - - - - - - -

Continuing in this manner, we obtain:

T(2) □ *T*(1) □ *C* - (8)

2 1

We add all the equations from 4 to 8 and note that there are log n of them:

____ *T*(*n*) *T*(1) *C* log *n*

□ □ *n* 1 - (9)

Which yields, T (n) = C n log n + n = **O(n log n)** - (10)

This is exactly the same analysis as merge sort, hence we get the same answer.

Average Case Analysis

The number of comparisons for first call on partition: Assume left_to_right moves over k smaller element and thus k comparisons. So when right_to_left crosses left_to_right it has made n-k+1 comparisons. So, first call on partition makes n+1 comparisons. The average case complexity of quicksort is

T(n) = comparisons for first call on quicksort
+
{Σ 1<=nleft,nright<=n [T(nleft) + T(nright)]}n = (n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-1)]/n

nT(n) = n(n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-2) + T(n-1)]

(n-1)T(n-1) = (n-1)n + 2 [T(0) +T(1) + T(2) + ----- + T(n-2)] \

Subtracting both sides:

nT(n) –(n-1)T(n-1) = [n(n+1) – (n-1)n] + 2T(n-1) = 2n + 2T(n-1) nT(n)
= 2n + (n-1)T(n-1) + 2T(n-1) = 2n + (n+1)T(n-1)
T(n) = 2 + (n+1)T(n-1)/n
The recurrence relation obtained is:
T(n)/(n+1) = 2/(n+1) + T(n-1)/n

Using the method of subsititution:

T(n)/(n+1) = 2/(n+1) + T(n-1)/n
T(n-1)/n = 2/n + T(n-2)/(n-1)
T(n-2)/(n-1) = 2/(n-1) + T(n-3)/(n-2)
T(n-3)/(n-2) = 2/(n-2) + T(n-4)/(n-3)
. .
. .
T(3)/4 = 2/4 + T(2)/3
T(2)/3 = 2/3 + T(1)/2 T(1)/2 = 2/2 + T(0)
Adding both sides:
T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + ------------- + T(2)/3 + T(1)/2]
= [T(n-1)/n + T(n-2)/(n-1) + ------------- + T(2)/3 + T(1)/2] + T(0) +
[2/(n+1) + 2/n + 2/(n-1) + ---------- +2/4 + 2/3]
Cancelling the common terms:
T(n)/(n+1) = 2[1/2 +1/3 +1/4+--------------+1/n+1/(n+1)]

T(n) = (n+1)2[□₂□*k* □*n* □1 1/ *k*
=2(n+1) [–]
=2(n+1)[log (n+1) – log 2]
=2n log (n+1) + log (n+1)-2n log 2 –log 2
T(n)= O(n log n)

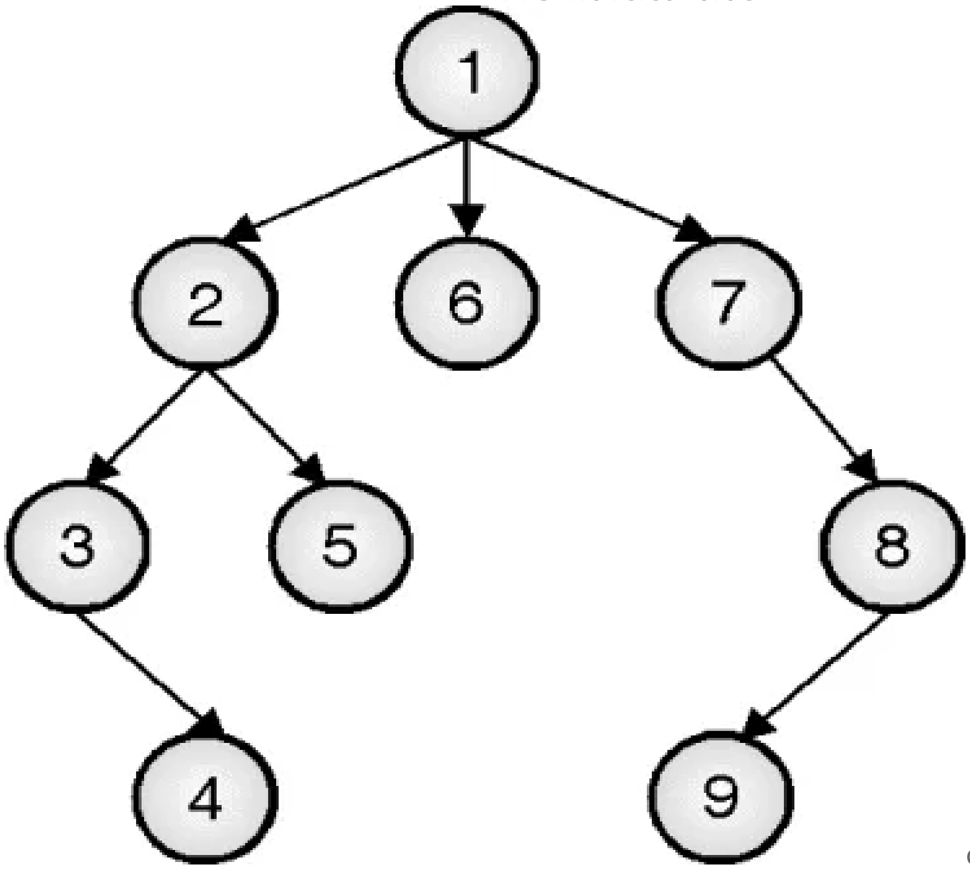
Depth First Search

Depth First Search is a very useful way of graph traversal in many computer science applications. It uses a stack to traverse the graph.

Working Mechanism of Depth First Search

- In graph traversal tree, root of the tree would be the node from where we started traversal process. It is closely related to pre-order traversal of tree. It explores the branch as deep as possible before backtracking and exploring another branch.
- Following figure shows the order of visited nodes in DFS traversal. Starting from root node, it explores the left branch and recursively calls its left child until there is no more left child. After reaching the leaf node, it visits the right child in same order and then backtracks to its parent.

DFS Traversal order



of nodes

- To simplify the procedure and to keep track of already visited vertices, we use time stamps. Two time stamps are associated with each vertex, value of time stamps vary from 1 to 2|v|. When vertex is encountered first time, it is called discovery of the vertex.

- On discovery of vertex u, time stamp d[u] is associated with that vertex, which is called discovery time of u.
- Similarly, when algorithm backtracks from any vertex u, it is assigned finishing time f[u].
- It is very obvious that each vertex v ∊ G, d[u] < f[u]. We will use color coding to indicate states for the nodes.
    - Vertex u will be white before it is discovered, that is before it is assigned d[u].
    - Vertex u will be gray between its discovery and finishing, that is between d[u] and f [u].
    - Vertex u will be black after algorithm backtracks from u, that is after assignment of f [u].
- DFS searches deeper and deeper in graph whenever it is possible.
- Unlike BFS, the sub graph created by DFS might have several trees. In other words, DFS traversal may produce DFS tree or DFS forest.

Algorithm for Depth First Search

Algorithms for DFS traversal is shown below:

```
Algorithm DFS
// Graph G = <V, E> is the input to DFS

time ← 0
for each vertex u ∊ V do
    setColor(u) ← WHITE
    π(u) ← NIL
end

for each vertex u ∊ V do
    if Color(u) == WHITE then
        DFS_TRAVERSAL(u)
    end
end
```

DFS_TRAVERSAL routine is described below :

```
DFS_TRAVERSAL(u)
Time ← time + 1
d[u] ← time
setColor(u) ← GRAY
for each v ∈ Adjacent[u] do
    if Color(v) == WHITE then
        π(u) ← u
        DFS_TRAVERSAL(v)
    end
end
setColor(u) ← BLACK
time ← time + 1
f[u] ← time
```

Complexity Analysis of Depth First Search

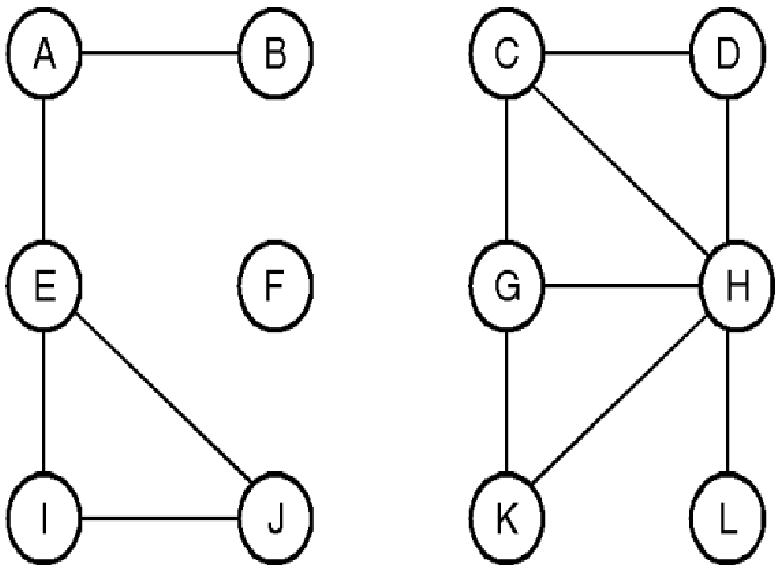
- In worst case, every edge and vertex will be explored by algorithm. So time complexity of algorithm is O(|V| + |E|). Number of edged depends on how sparse the graph is? In worst case, |E| would be $O(|V|^2)$.
- In worst case, DFS traversal need stack of size V to store vertices. Mostly this is required in case of skew tree.
- With adjacency list representation, it requires O(|V| + |E|) space. Space complexity in case of adjacency matrix representation would be $O(|V|^2)$.

Applications of DFS

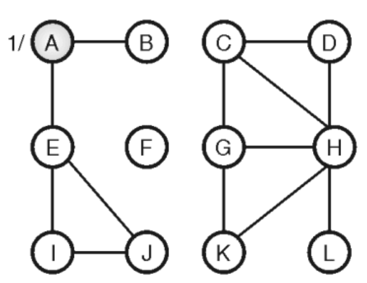
- Finding connected components of graph.
- Topological sorting.
- Finding strongly connected component.
- Check the cycle in undirected graph.
- Game playing.
- Checking biconnectivity of graph.
- Finding articulation point.

Example

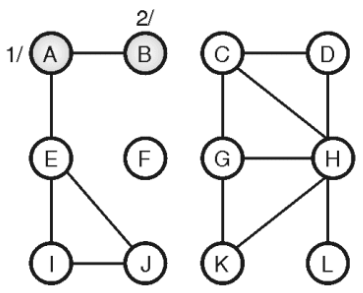
Example: Create a DFS forest for the following graph



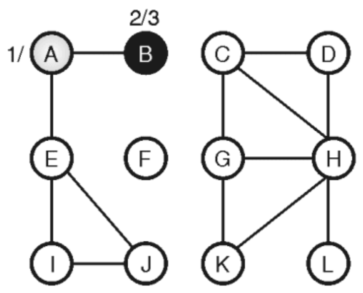
Solution:



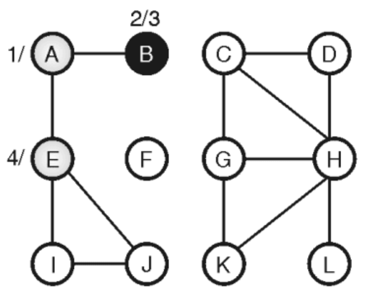
(a) A is discovered



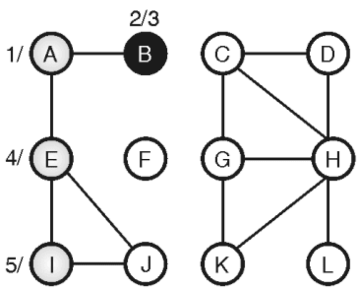
(b) B is discovered



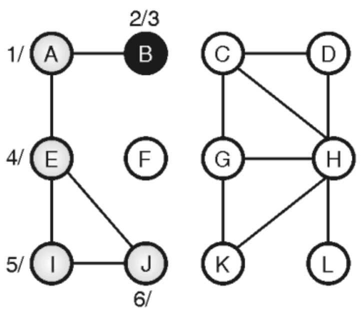
(c) B is finished



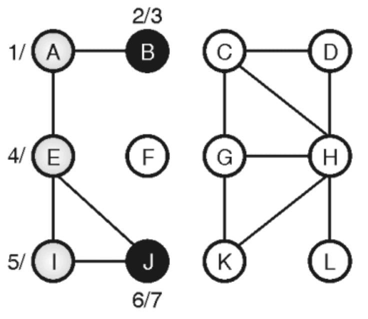
(d) E is discovered



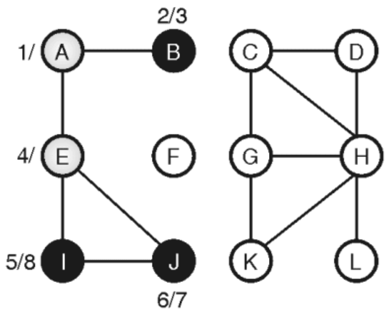
(e) I is discovered



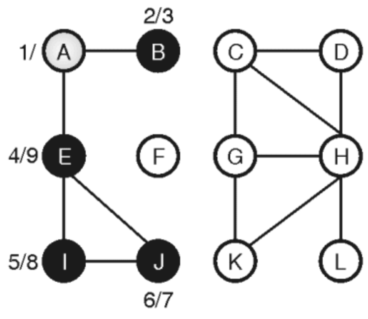
(f) J is discovered



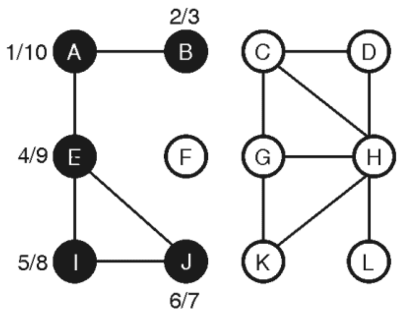
(g) J is finished



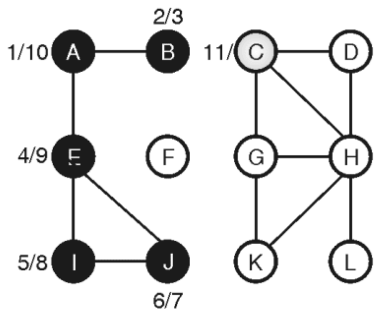
(h) I is finished



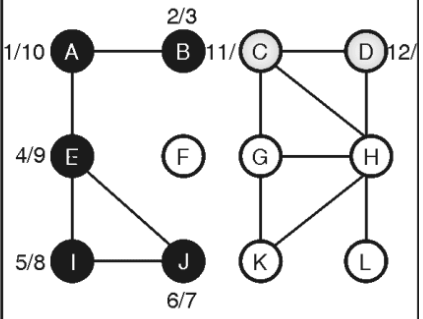
(i) E is finished



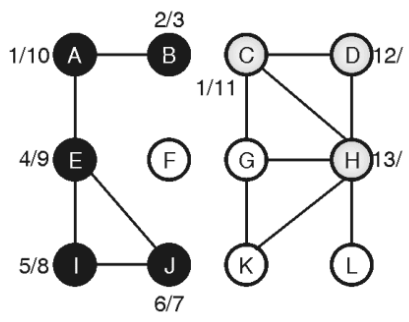
(j) A is finished



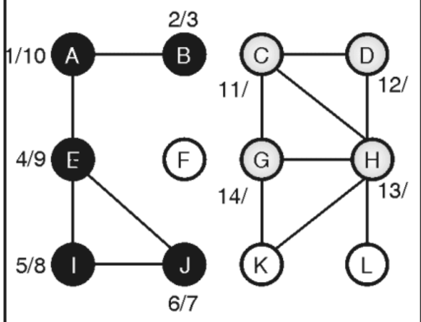
(k) C is discovered



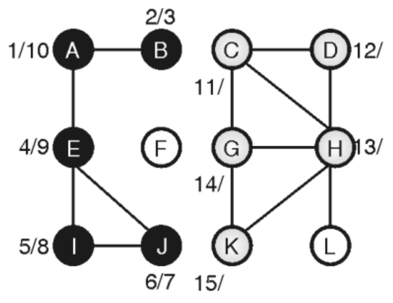
(l) D is discovered



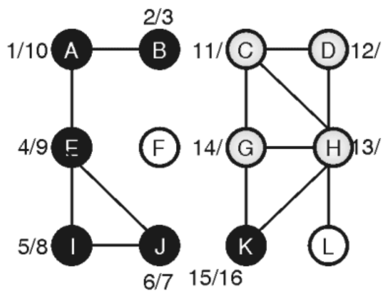
(m) H is discovered



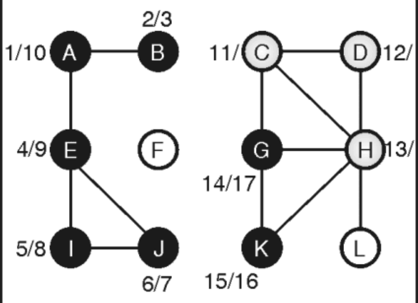
(n) G is discovered



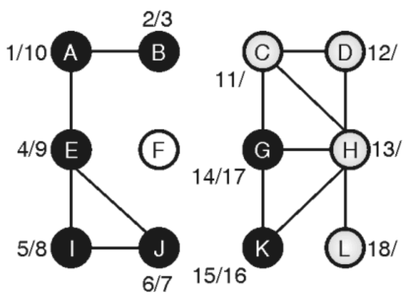
(o) K is discovered



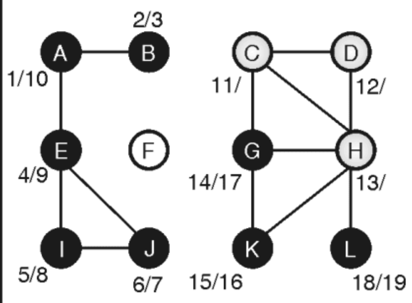
(p) K is finished



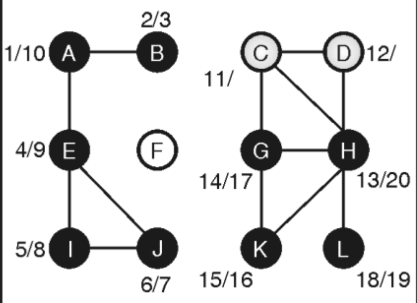
(q) G is finished



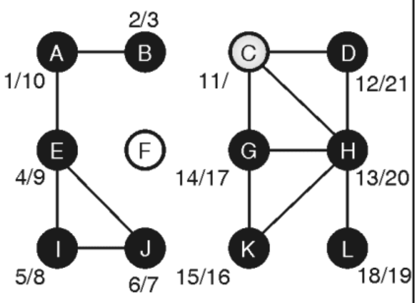
(r) L is discovered



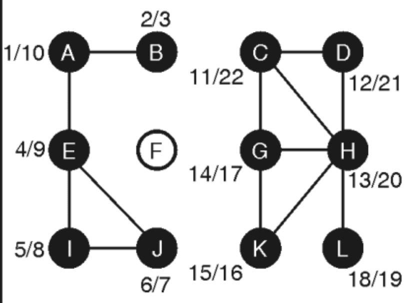
(s) L is finished



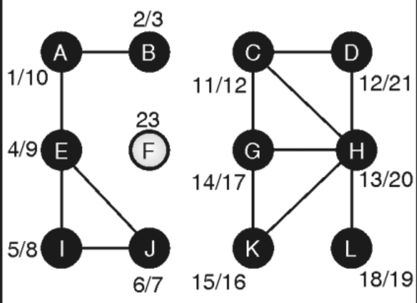
(t) H is finished



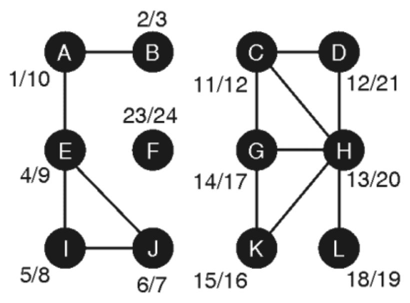
(u) D is finished



(v) C is finished



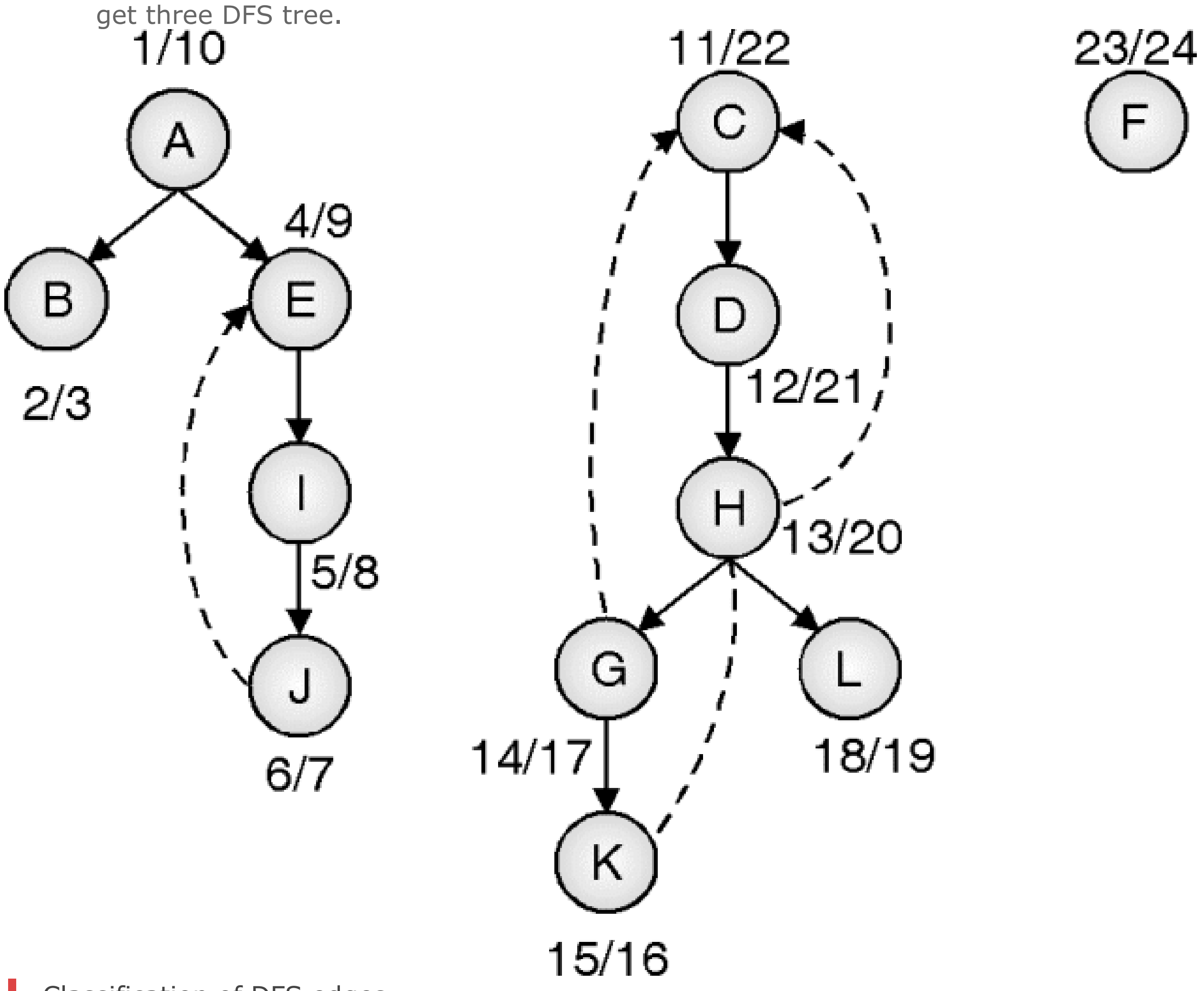
(w) F is visited



(x) F is finished

- Initially,none of the vertex is visited, so they are white. On discovering vertex first time, it is converted to gray. When no more unvisited child exists for vertex, it is considered finished and converted to black.
- Discovery time and finish time is written beside each vertex separated by slash (/).

- DFS Forest of above graph is shown below. Dotted edge indicates back edges. We get three DFS tree.



Classification of DFS edges

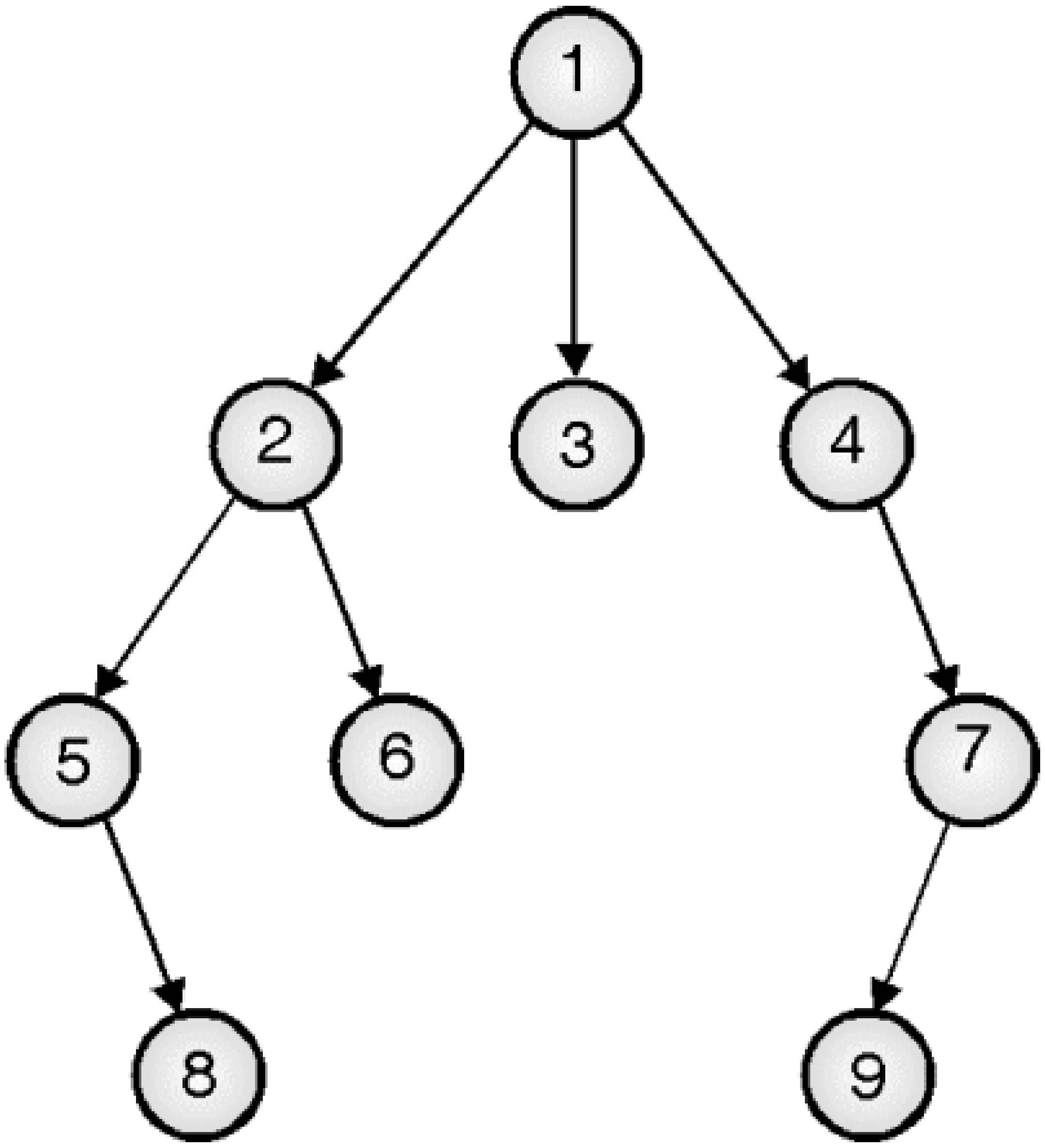
Edges of DFS forest are classified in one for the four categories :

Tree edges : In the depth-first forest, edge (u, v) is considered tree edge if v was first discovered by exploring edge (u, u).
Back edge : In the depth-first tree, edge (u, v) is called back edge if edge (u, v) connects a vertex u to an ancestor v. Self-loop is considered as back edge.
Forward edge : Forward edge (u, v) is the non-tree edge that connects vertex v to its descendent in the depth-first tree.
Cross edge : Remaining all edges is classified as cross edges.

Breadth First Search

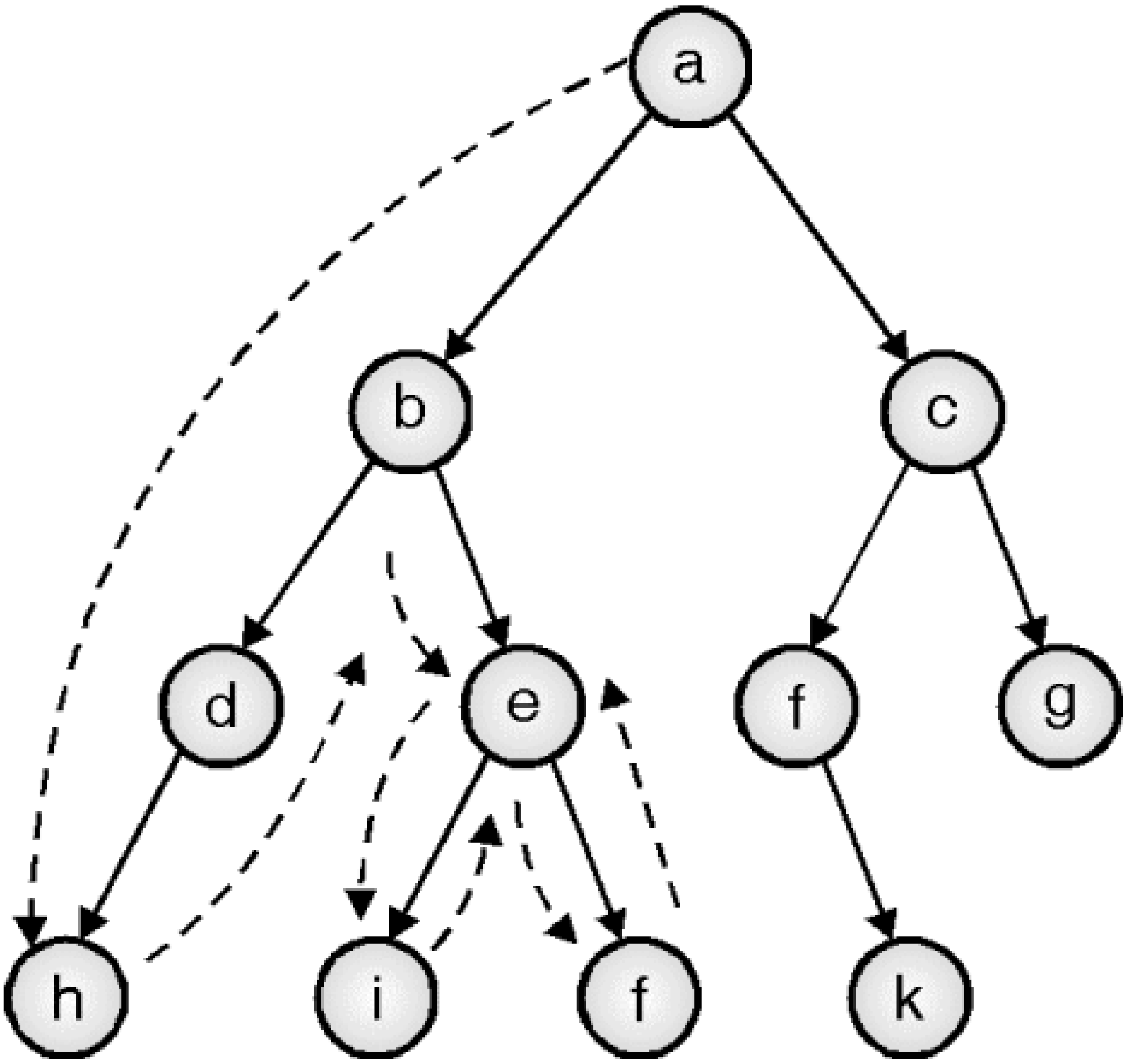
Breadth first search is as widely used as Depth-first search in problem-solving.

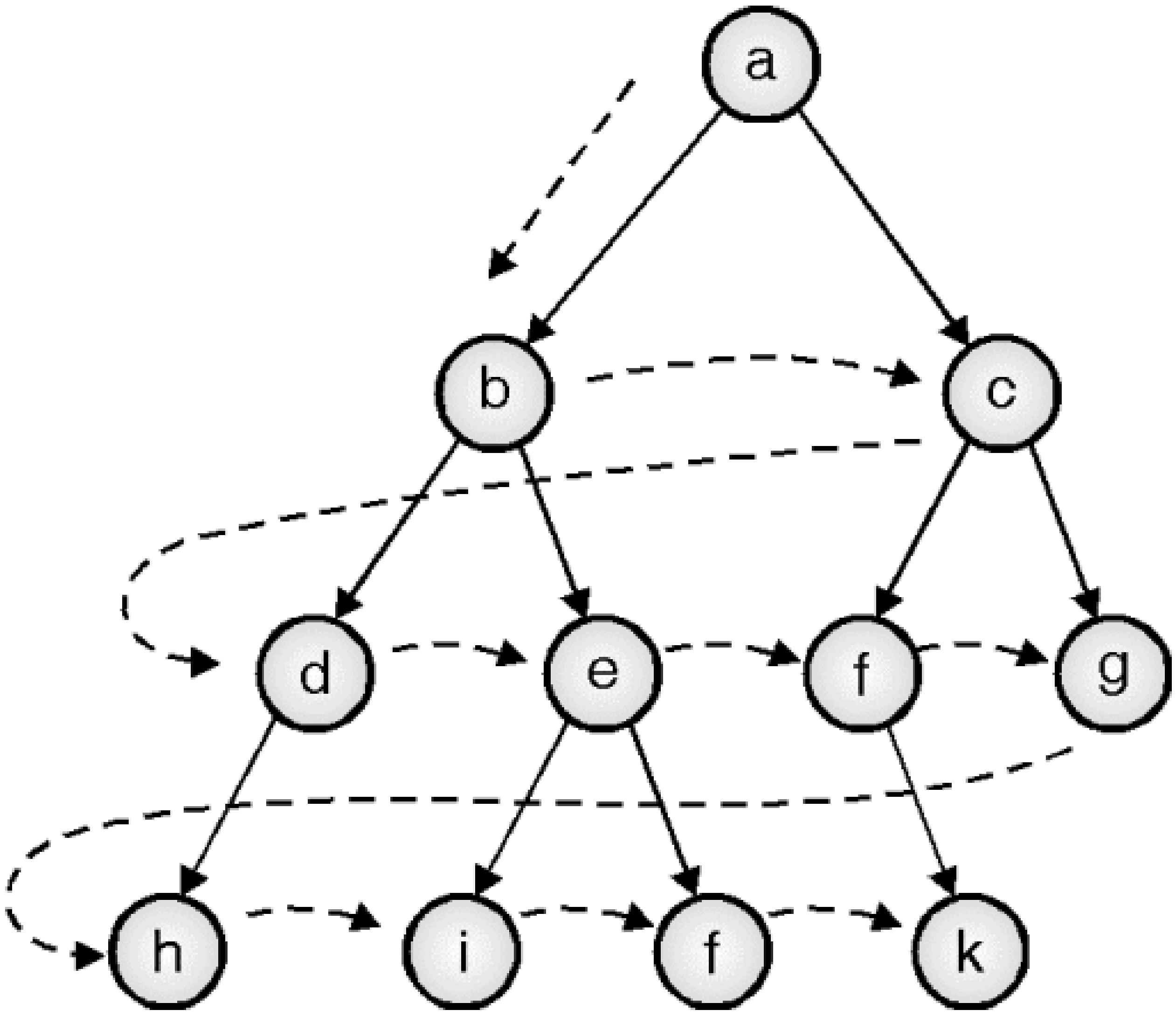
- Concept of **Breadth First Search (BFS)** algorithm is used in many applications. Prim's algorithm to find minimum spanning tree and Dijakstra's single source shortest path algorithm uses the concept of BFS.
- For graph G = (V, E), BFS starts form some source vertex s and explores it's all neighbors reachable from s. Any path in breadth first tree from vertex u to v corresponds to shortest path in graph G.



- This approach discovers all the vertices at distance k from the root of tree before exploring any vertex at distance k + 1.
- Figure (A) shows the order of breadth first traversal.

- Figure (B) abd Figure (C) shows the traversing order of DFS and BFS respectively


a
b
c
d
e
f
g
h
i
f
k



- DFS traversal of above graph: h – d – i – f – e – b – k – f – g – c – a
- BFS traversal of above graph: a – b – c – d – e – f – g – h – i – j – k
- Breadth first search traversal explores all neighbors of current node before going to next level of neighbors.

Algorithm for Breadth First Search

The algorithm for BFS traversal is described below :

Algorithm

```
BFS(v, Q)

// v is the root vertex or initial vertex

// Q is the Queue data structure to store unexplored nodes


ENQUEUE(v, Q)           // Insert v at the end of queue Q

label v as discovered
```

while

```
Q is not empty
```

do

```
  v ← DEQUEUEU(Q)       // remove front node from queue Q
for
each edge from v to u
do
if
u is not labeled as discovered
then
     ENQUQUE(w, Q)
end
end
end
```

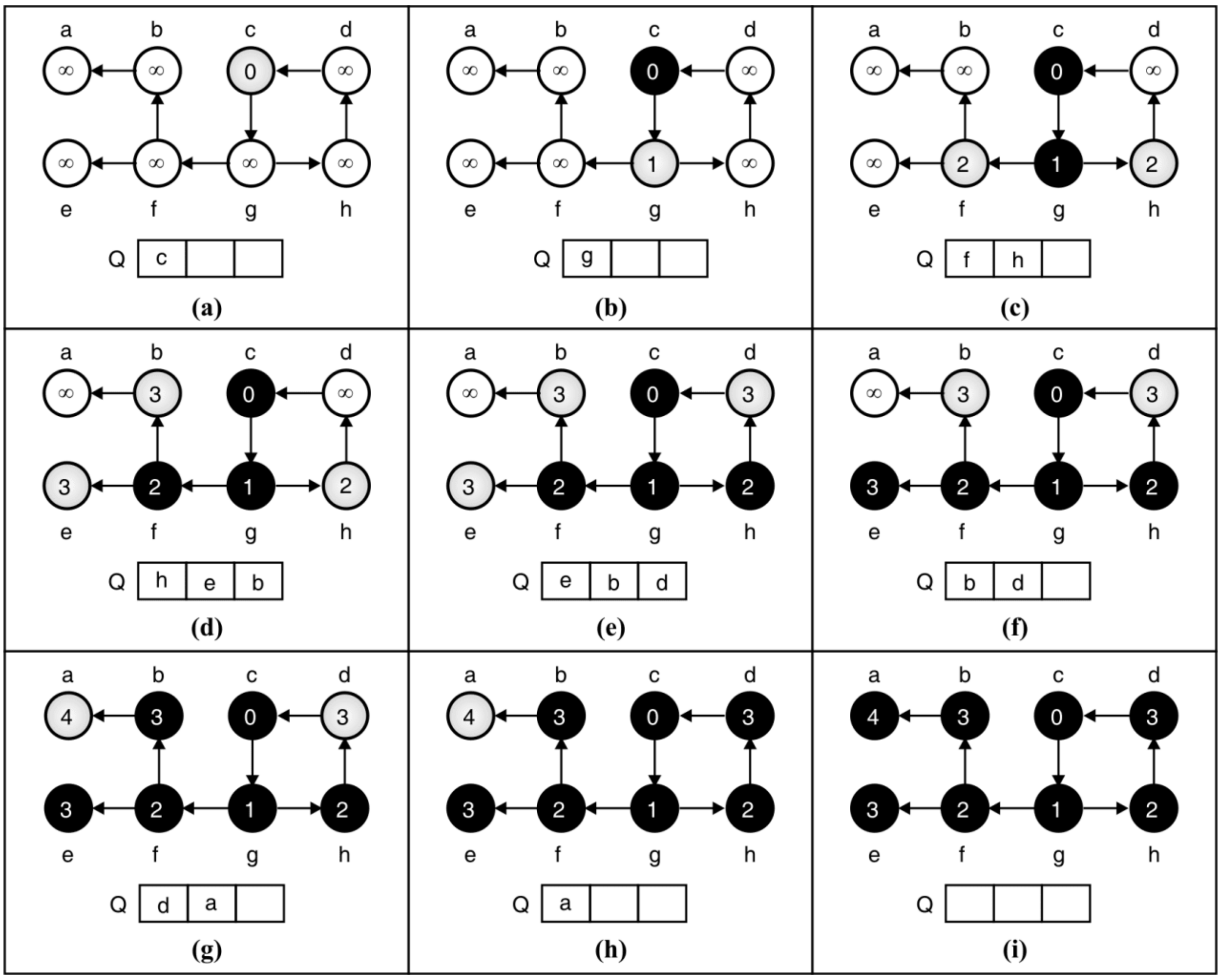
Complexity Analysis of Breadth First Search

- In worst case, every edge and vertex will be explored by algorithm. So time complexity of algorithm is O(|V| + |E|). Number of edged depends on how sparse the graph is? In worst case, |E| would be $O(V^2)$.
- With adjacency list representation, it requires O(|V| + |E|) space. Space complexity in case of adjacency matrix representation would be $O(|V|^2)$.

Example

DFS used stack whereas BFS uses a queue for traversing the graph. Figure 1 simulates the function of BFS. Following color conventions are used for simplicity :

- **White :** Undiscovered state. Initially all nodes are in undiscovered state and hence white.
- **Gray :** Discovered but not fully explored. When algorithm encounters node first time, it is inserted in queue and converted white to gray.
- **Black :** Fully explored / visited. When node v is removed from queue and all its neighbours are inserted in queue, node v is called fully explored / visited and converted gray to black.
- Let us start with arbitrary node c. Initial node c is inserted in queue labelled as Q. Node c is in queue, so it is gray and it is in discovered but not fully explored state. Figure 1(a)
- Now, Front node c is removed from queue and labelled as visited, so it is converted from gray to black. All its neighbours are inserted in to queue. Here, g is the only neighbor of c. Node g is inserted in Q and colored gray. Figure 1(b).
- Next, the front node g is removed from queue and labelled as visited, and all its unvisited neighbours are inserted in queue. Figure 1(c).
- Node f is at the front of queue and hence it is removed from queue and labelled as visited. All its unvisited neighbours are inserted in queue (Figure 1(d).
- This process continues until Q is empty.

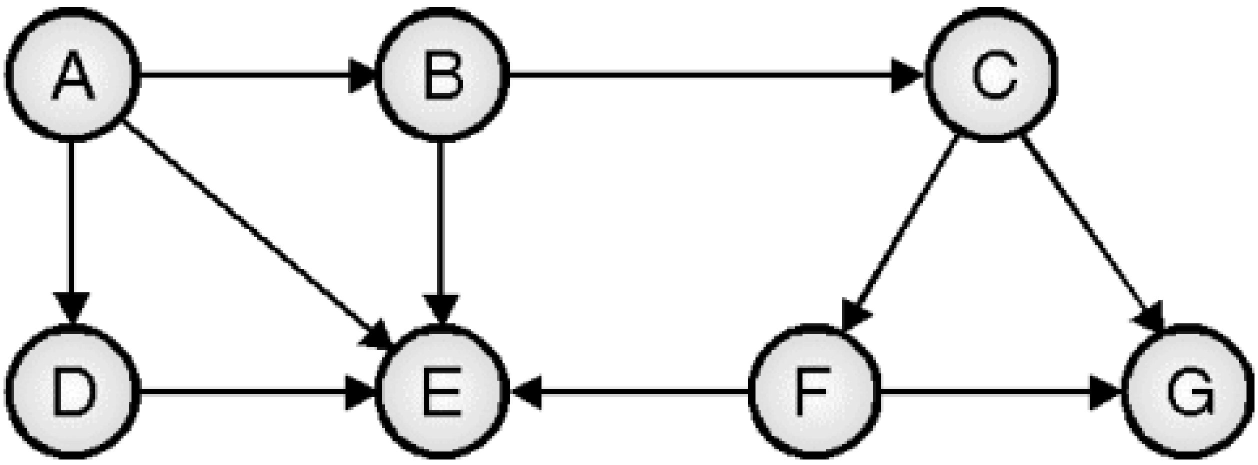


Topological Sort

Topological sort is applicable on directed acyclic graphs only. So let us first understand what we mean by the directed acyclic graph.

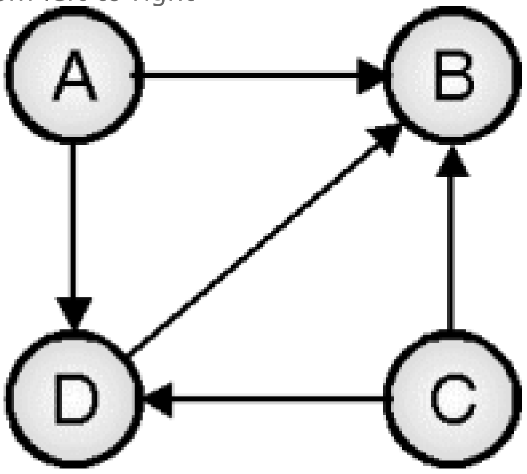
Directed Acyclic Graph

A graph is called a Directed Acyclic Graph (DAG) if the graph is a directed graph and it has no directed cycles. DAG is used to find common subexpressions in optimizing compilers. The graph shown below is an example of a directed acyclic graph

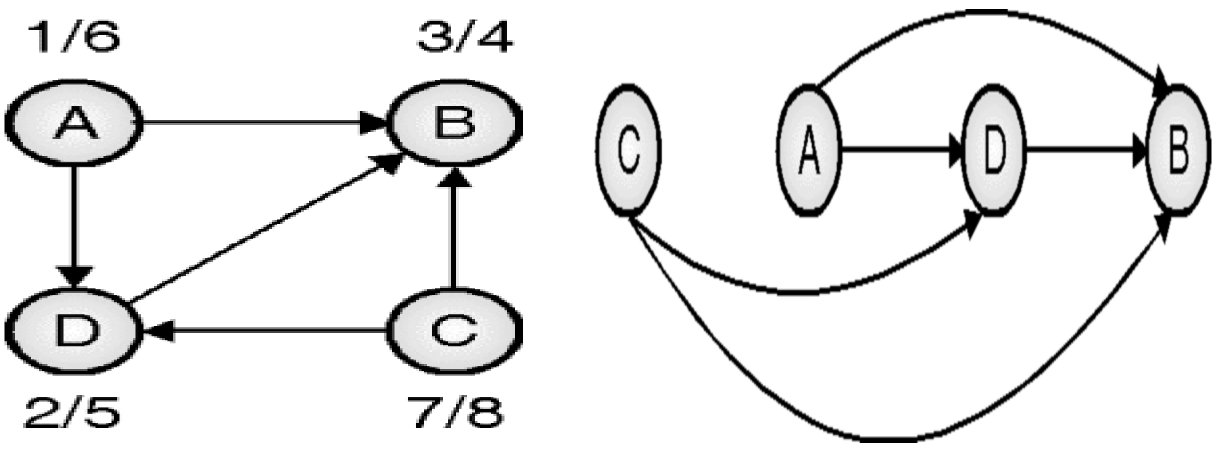


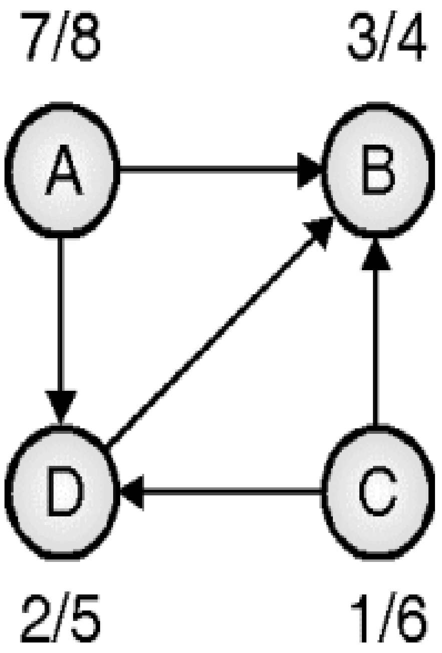
Working Mechanism

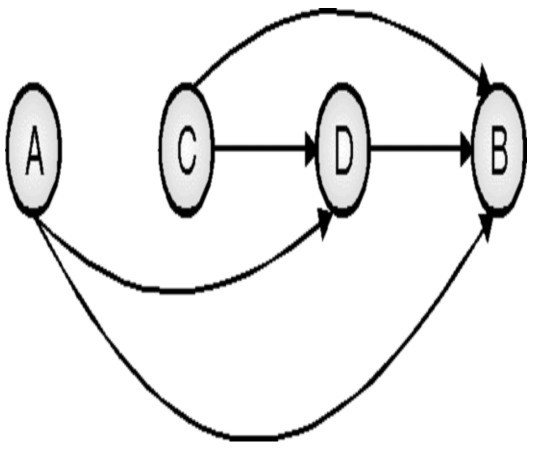
- Topological sorting of directed graph is linear ordering of its vertices. In this sorting, for every directed edge from u to v, u comes before v in ordering. This is used to simulate tasks and their ordering. In graph, each node may represent one task and every edge may represent constraint that one task should be complete before another.
- Topological sorting is possible only if graph is DAG. Every DAG has at least one topological sorting sequence. If graph is not DAG, topological sorting is not possible.
- Topological sorting is different than DFS. In topological sorting, vertex is print before its adjacent vertices.
- In simplest way, to find topological ordering of graph, we should perform following steps :
    - Visit graph in DFS order and label the vertices with their discovery and finish time.
    - Linearly arrange vertices in decreasing order of their finish time from left to right



- There may exist more than one DFS forest of graph, so multiple topological ordering for same graph is possible. Arc can only go in forward direction. For above graph, two different DFS traversal and their linear ordering is shown below :







Algorithm for Topological Sort

The algorithm for topological sorting is shown below:

Algorithm

```
TOPOLOGICAL_SORT

// Input to algorithm is graph G = <V, E>


Call DFS

Display vertices in decreasing order of their finish time
```

Complexity Analysis

This algorithm scans edges and nodes once to order them. So running time is linear in a number of edges and vertices, i.e. O(|V| + |E|).

Applications of Topological Sort

- Project management.
- Event management.
- Job scheduling.
- Instruction scheduling.
- Order of compilation tasks.
- Data serialization.
- Ordering of cell evaluation in formulas in spread sheet.

Stable and Unstable Sorting Algorithms

Stability is mainly essential when we have key-value pairs with duplicate keys possible (like people's names as keys and their details as values). And we wish to sort these objects by keys.

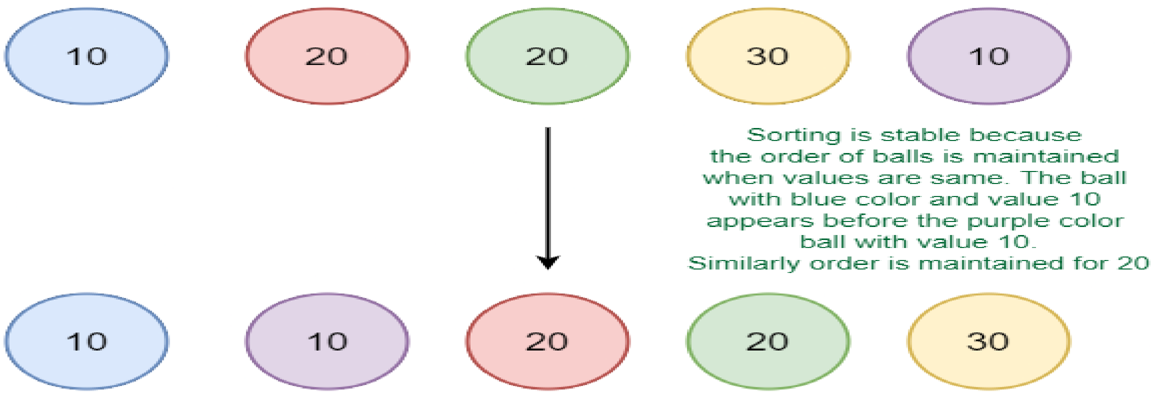
What is a stable sorting algorithm?

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input data set

Formally stability may be defined as, how the algorithm treats equal elements.
Let **A[]** be an array, and let **'<'** be a strict weak ordering on the elements of A[]. A sorting algorithm is stable

if: where is

the sorting permutation (sorting moves to position) .
Informally, stability means that equivalent elements retain their relative positions, after sorting.


10
20
20
30
10
Sorting is stable because the order of balls is maintained when values are same. The ball with blue color and value 10 appears before the purple color ball with value 10. Similarly order is maintained for 20
10
10
20
20
30

Example of stable sort