



RNSIT-CSE
Bengaluru

Divide and Conquer Module 2



Reshma Jakabal

Assistant Professor

Department of CS&E, RNSIT

Contents

- Binary Search
- Recurrence equation for Divide and Conquer
- Finding Maximum and Minimum
- Merge Sort
- Quick Sort
- Strassen's matrix multiplication
- Advantages and Disadvantages of Divide and Conquer
- Decrease and Conquer Approach: Topological Sorting

Introduction

- Divide-and-Conquer (DaC) is probably the best-known general algorithm design technique.
- Given a function to compute on n inputs the divide-and-conquer strategy suggests splitting the inputs into k distinct subsets, $1 < k < n$, yielding k subproblems.
- These subproblems must be solved, and then a method must be found to combine subsolutions into a solution of the whole.
- If the subproblems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
- Often the subproblems resulting from a divide-and conquer design are of the same type as the original problem.
- For those cases the reapplication of the divide-and-conquer principle is naturally expressed by a recursive algorithm.

- **Example 1** [detecting a counterfeit coin]
- **Statement:** you are given a bag with 16 coins. Told that one in that is counterfeit, which even is lighter than genuine coin. Now, task is to determine whether the bag contains a counterfeit coin or not. [Machine is supported to weigh the coins].

- **Solution using divide and conquer strategy**

1. Divide the original instance in to two or more instances.

16 is divided in to 2 each sets A and B.

2. Determining A or B contains counterfeit coin use machine to compare the weights. If both have different weights' then counterfeit coin is present.

3. Take the result from step2 and generate the answer for the original 16-coin instance.

Algorithm Control abstraction using divide-and-conquer

Algorithm DAndC(P)

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

Introduction

- If the size of **P** is **n**, And the sizes of the **k** subproblems are **n1, n2, n3... nk**, respectively, then the computing time of DAndC is described by the recurrence relation

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

Where,

- **T(n)** is the time for DAndC on any input of size n
- **g(n)** is the time to compute the answer directly for small inputs
- **f(n)** is the time for dividing **P** and combining the solutions to subproblems.

Introduction

- The complexity of many divide and conquer is given by recurrences of the form

$$\bullet T(n) = \begin{cases} T(1) & n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & n > 1 \end{cases}$$

Where,

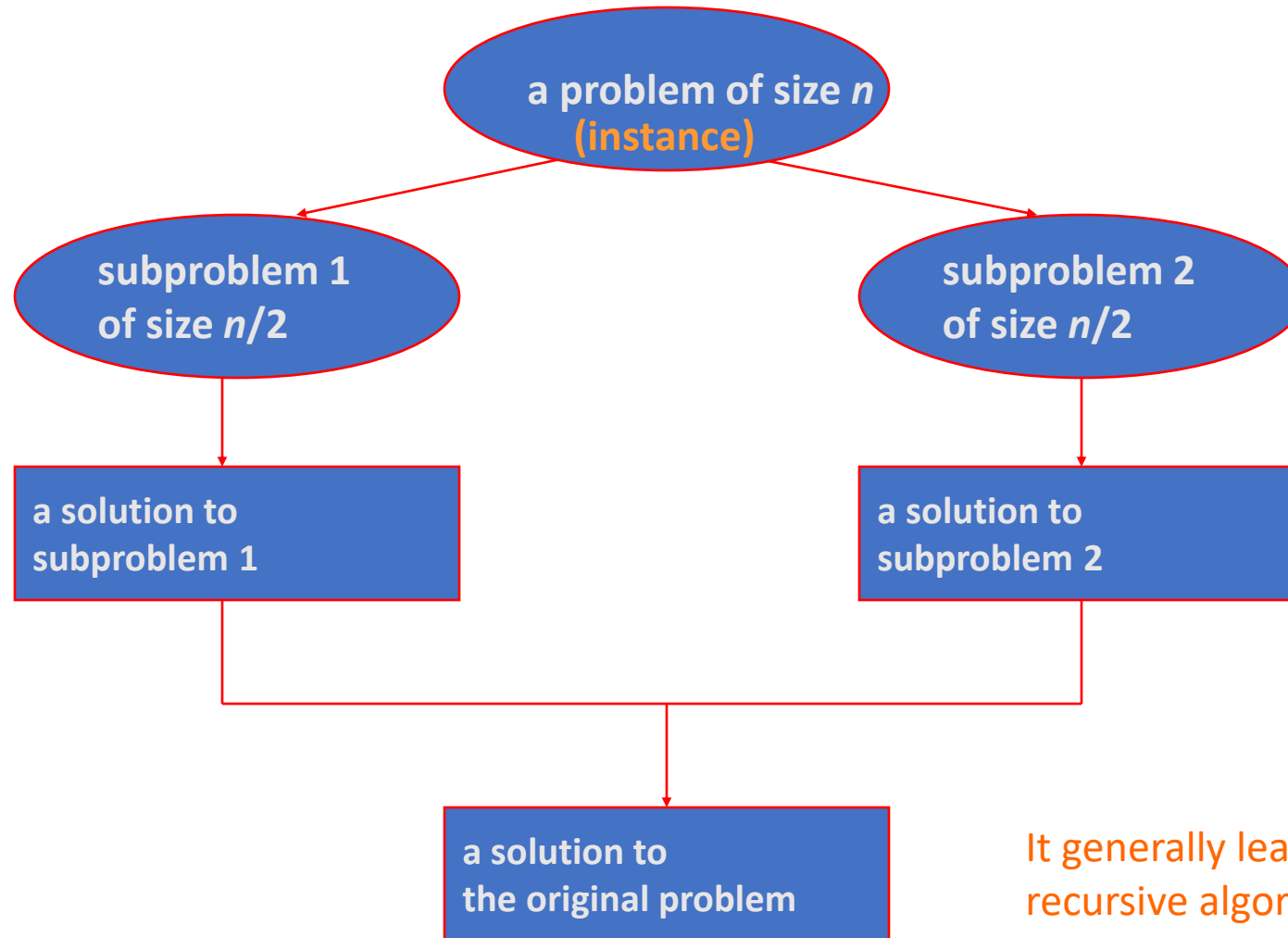
- **a** and **b** are constants
- **T(1)** is known
- **n** is a power of **b** (i.e., **n=b^k**)

Divide-and-Conquer

The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique (cont.)



It generally leads to a recursive algorithm!

Divide-and-Conquer Examples

- Sorting: Mergesort and Quicksort
- Binary tree traversals
- Binary search
- Multiplication of large integers
- Matrix multiplication: Strassen's algorithm

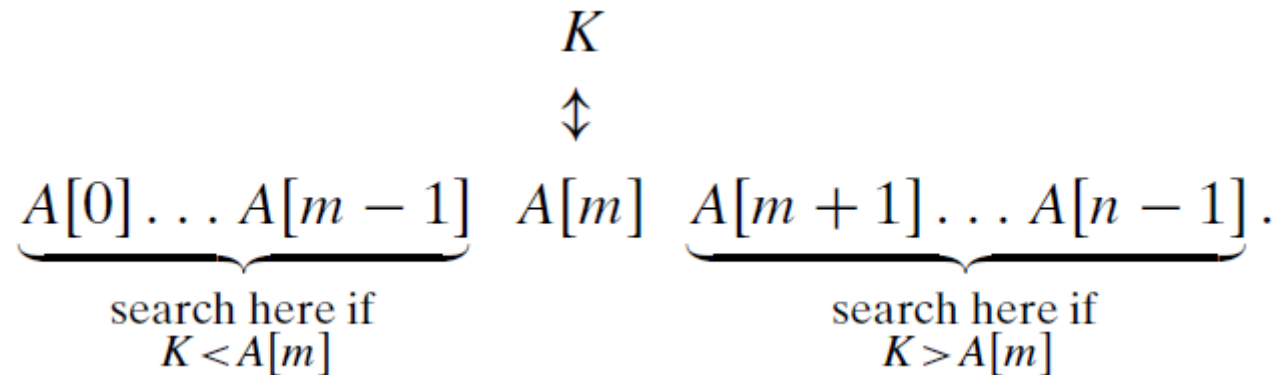
Binary Search

- Binary search is a well known instance of divide and conquer method. For binary search divide and conquer strategy is applied recursively for a given sorted array is as follows:
- **Divide**: Divide the selected array at the middle. It creates two sub-array, one left sub-array and other right sub-array.
- **Conquer**: Find out the appropriate sub-array.
- **Combine**: Check for the solution to key element.

- For a given sorted array of N element and for a given key element (value to be searched in the sorted array), the basic idea of binary search is as follows –
 1. First find the middle element of the array
 2. Compare the middle element with the key element.
 3. There are three cases
 - If it is the key element then search is successful.
 - If it is less than key element then search only the lower half of the array.
 - If it is greater than key element then search only the upper half of the array.
 4. Repeat 1, 2 and 3 until the key element found or sub-array sizes become one

Binary Search

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



Binary Search

Example

- Apply binary search algorithm to the following set of numbers considering **70** as the key

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

index 0 1 2 3 4 5 6 7 8 9 10 11 12

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

iteration 1	l						m						r
-------------	-----	--	--	--	--	--	-----	--	--	--	--	--	-----

iteration 2								l		m			r
-------------	--	--	--	--	--	--	--	-----	--	-----	--	--	-----

iteration 3								l, m		r			
-------------	--	--	--	--	--	--	--	--------	--	-----	--	--	--

Binary Search : Non-Recursive

```

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


Binary Search : Recursive

```

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

Binary Search : Analysis

Recurrence Relation

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

Solution

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

$$= [T(n/4) + 1] + 1 = T(n/4) + 2$$

$$= [T(n/8) + 1] + 2 = T(n/8) + 3$$

- - - - -

$$- = T(n/2^k) + k \quad n = b^k, n = 2^k, \log n = \log 2^k, k = \log n$$

$$- = T(n/n) + k$$

$$- = 1 + k$$

$$- = 1 + \log n$$

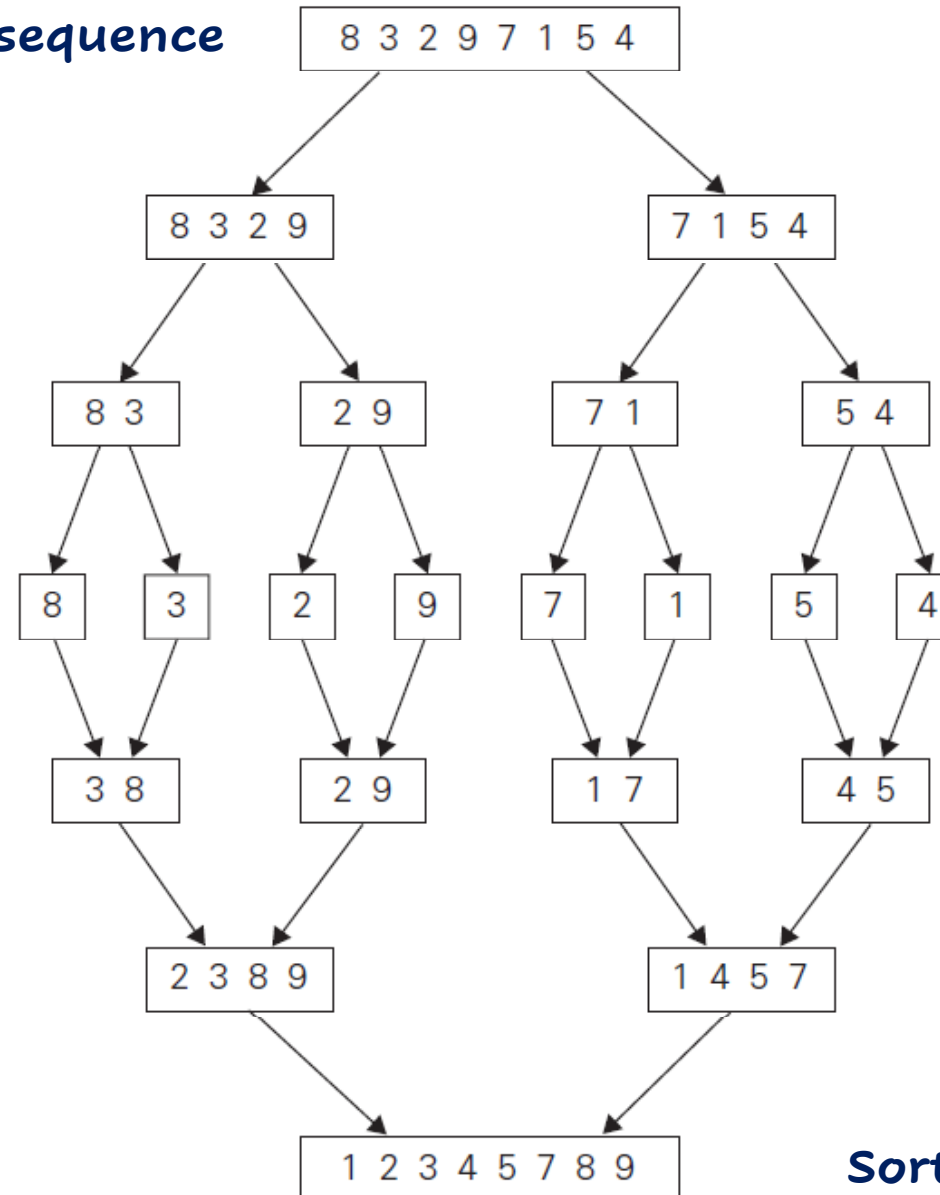
$$- \mathbf{T(n) = O(\log n)}$$

Merge Sort

- Merge sort is a perfect example of a successful application of the divide-and conquer technique.
- Let us assume that the set of elements are to be sorted in **non-decreasing** order that is in **ascending** order.
- Given a sequence of n elements , $a[1].....a[n]$, the idea is to imagine them split into two sets $a[1]..... a[\lfloor n/2 \rfloor]$ and $a[\lfloor n/2 \rfloor + 1]..... a[n]$.
- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of n elements
- This is an ideal example of the divide-and-conquer strategy in which the **splitting** is into two equal-sized sets and the combining operation is the merging of two sorted sets into one

Merge Sort

Input sequence



Divide

Divide

Divide

Divide

Combine

Combine

Sorted sequence

Merge Sort

Do it yourself

- Consider the input sequence

11, 44, 22, 99, 66, 33, 88, 55, 77, 00

Obtain the merge sort tree representation showing the divide and combine phase

Merge Sort – algorithm

```

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

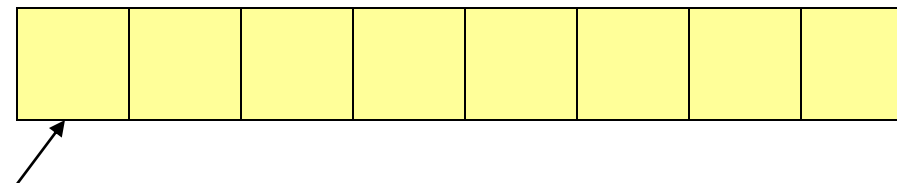
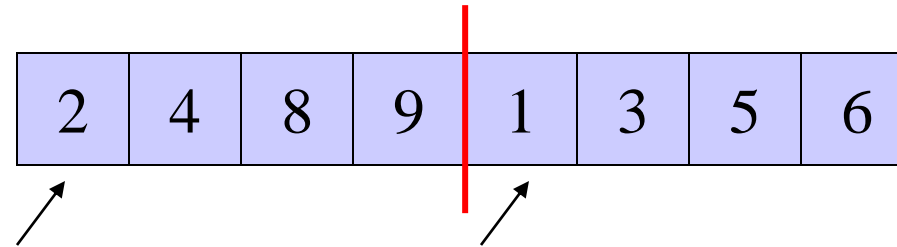
Merge Sort – algorithm

```

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

Working of Merge

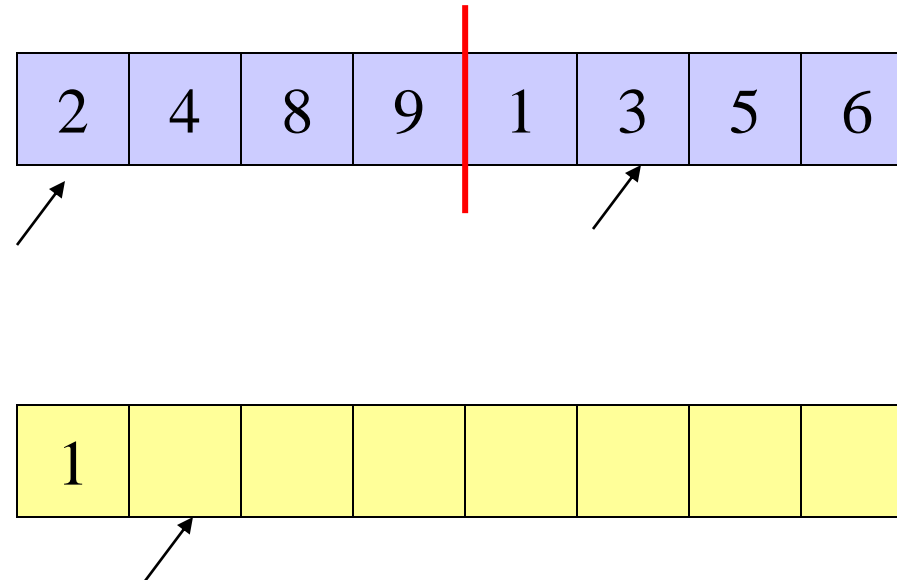
- The merging requires an auxiliary array.



Auxiliary array

Working of Merge

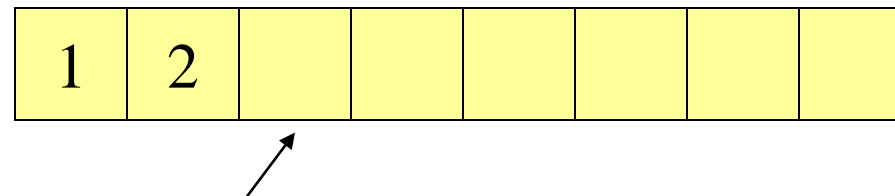
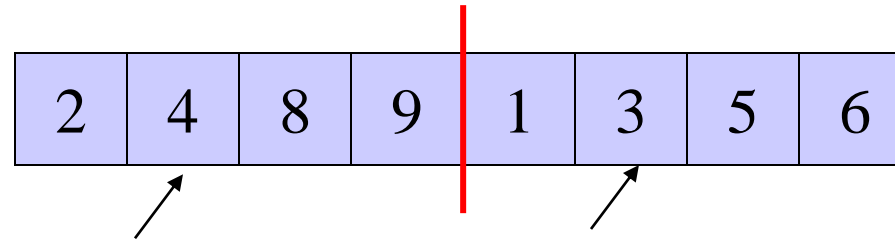
- The merging requires an auxiliary array.



Auxiliary array

Working of Merge

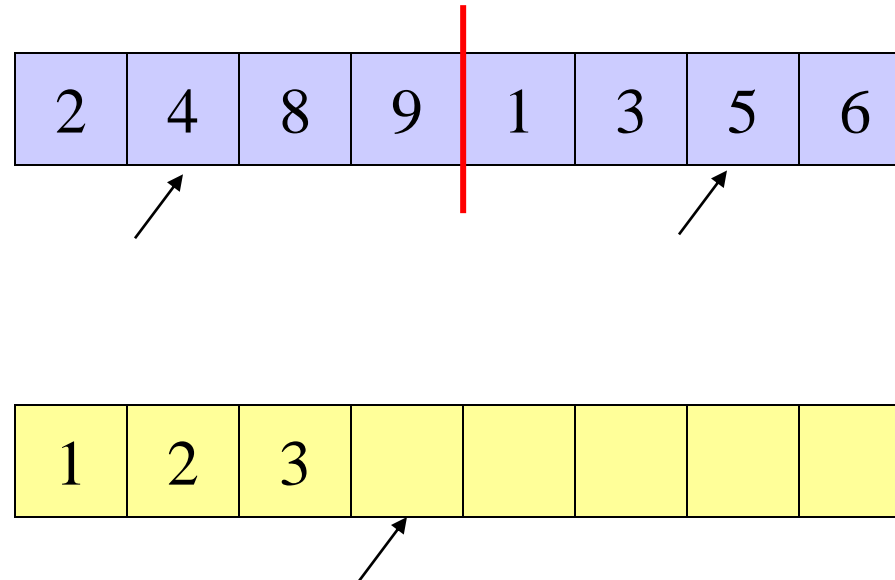
- The merging requires an auxiliary array.



Auxiliary array

Working of Merge

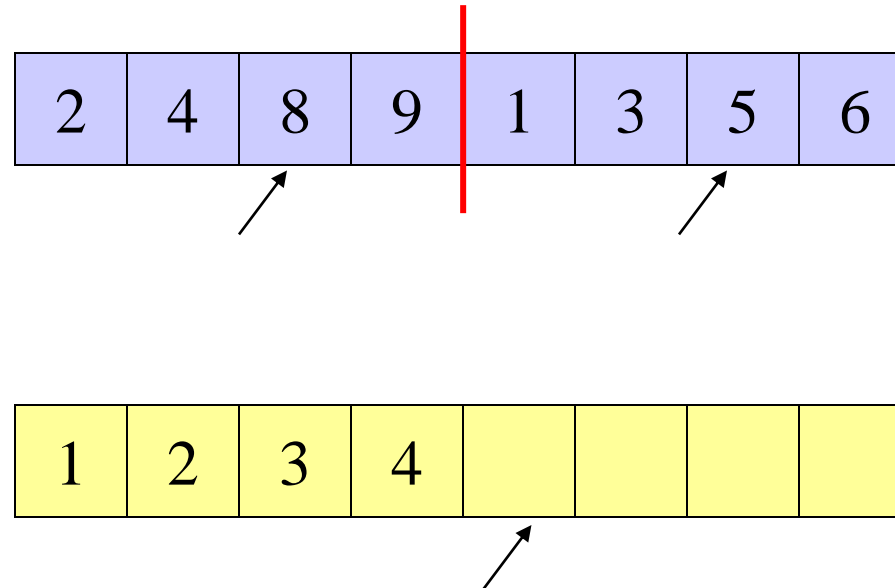
- The merging requires an auxiliary array.



Auxiliary array

Working of Merge

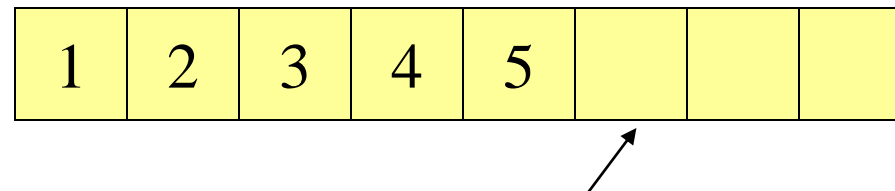
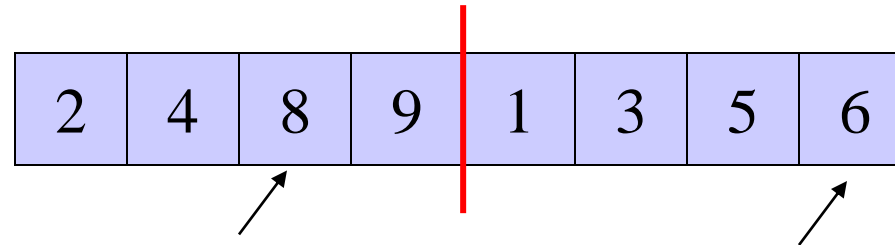
- The merging requires an auxiliary array.



Auxiliary array

Working of Merge

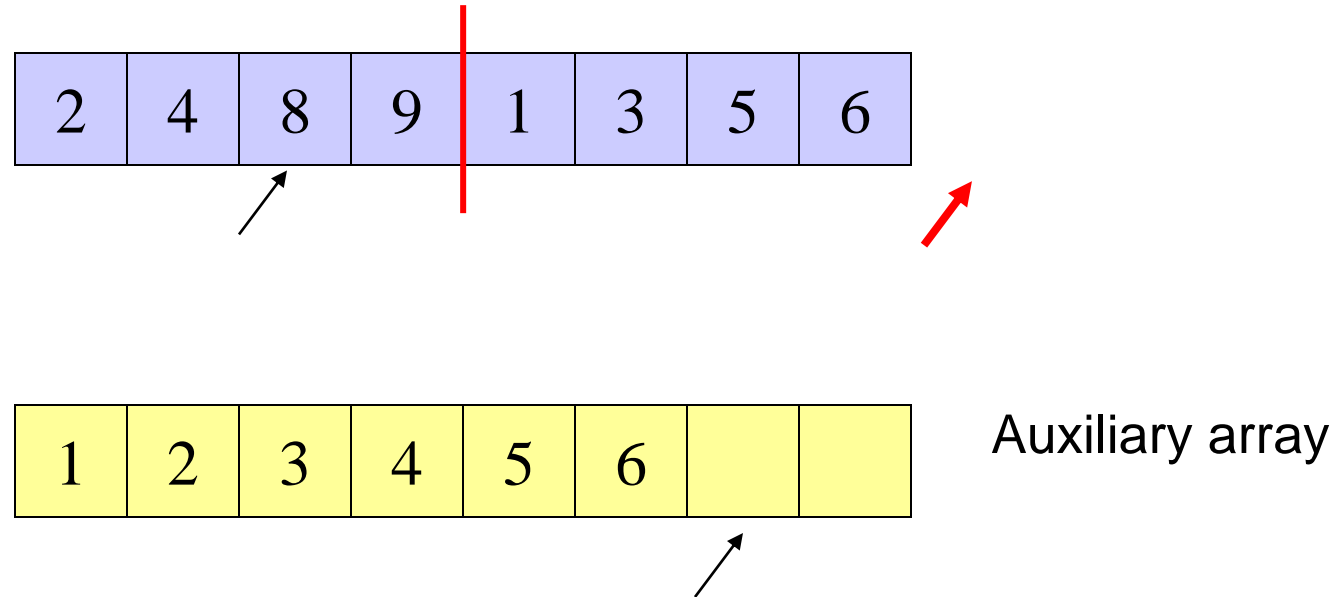
- The merging requires an auxiliary array.



Auxiliary array

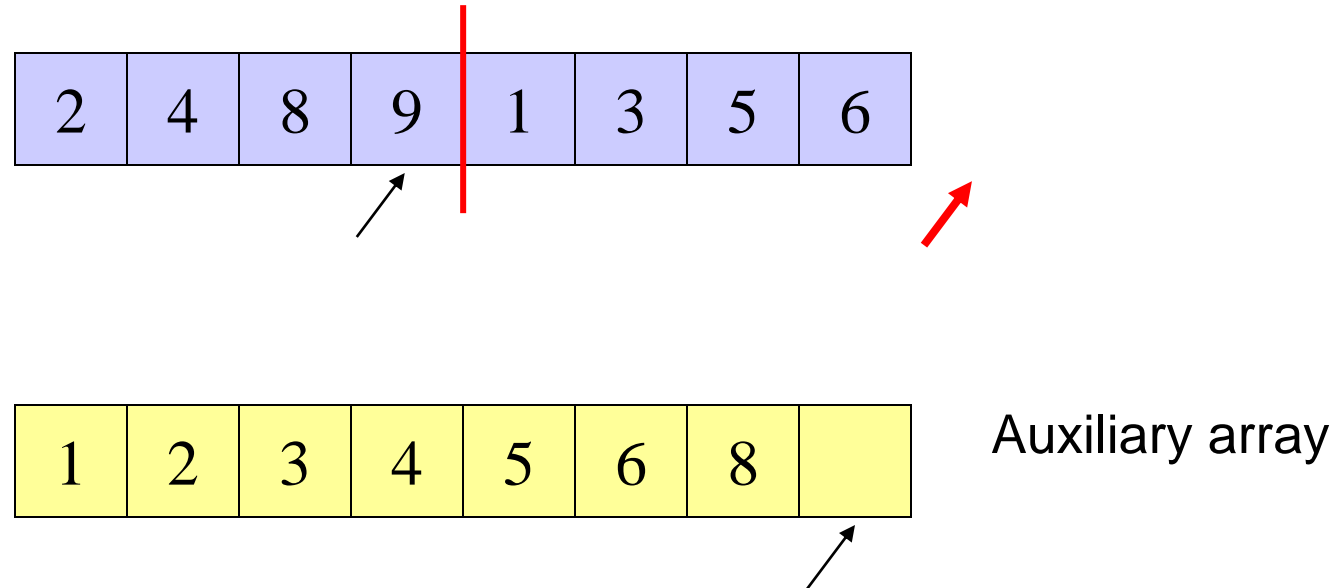
Working of Merge

- The merging requires an auxiliary array.



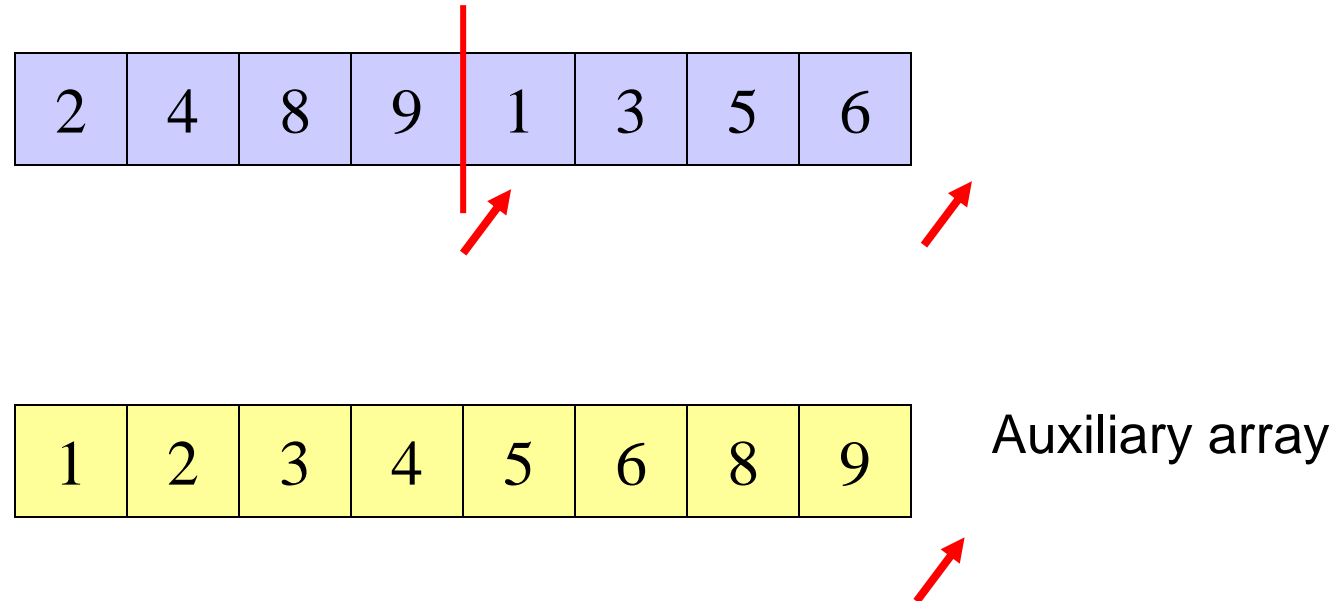
Working of Merge

- The merging requires an auxiliary array.



Working of Merge

- The merging requires an auxiliary array.



- The complexity of many divide and conquer is given by recurrences of the form

$$\bullet T(n) = \begin{cases} T(1) & n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & n > 1 \end{cases}$$

Where,

- **a** and **b** are constants
- **T(1)** is known
- **n** is a power of **b** (i.e., **n=b^k**)

Merge Sort - Analysis

- The recurrence relation for Merge sort is given by

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

- Solution**

In the given relation $a=2, b=2, f(n)=n$, n is power of b so $n=b^k, n=2^k$

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

The general pattern ?

$$= 2^k T(n/2^k) + kn \quad n=2^k, \quad k=\log n$$

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

$$= n \cdot 0 + n \log n \quad \text{considering only leading term and ignoring constants we get}$$

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

Pros of Mergesort

- Supports Large size list
- Suitable for linked list
- Supports external sorting
- Stable

Cons of Mergesort

- Requires extra space (not in place)
- Recursion- Uses more memory on stack

Merge Sort - **Analysis**

Properties summarized

- Merge Sort is useful for sorting linked lists.
- Merge Sort is a stable sort which means that the same element in an array maintain their original positions with respect to each other.
- Overall time complexity of Merge sort is $\Theta(n \log n)$.
 - i.e. its best, worst and average case time complexity is $\Theta(n \log n)$.
- It is more efficient as it is in worst case also the runtime is $\Theta(n \log n)$
- The space complexity of Merge sort is $O(n)$. This means that this algorithm takes a lot of space and may slower down operations for the large data sets.
- Merge sort is not **in-place** sorting

Quick Sort

- Quicksort is the other important sorting algorithm that is based on the **divide-and conquer** approach.
- Unlike **merge sort**, which divides its input elements according to their **position** in the array, **quicksort** divides them according to their **value**.
- The idea of array **partition** is used in this sorting.
- 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:

$$\underbrace{A[1] \dots A[s-1]}_{\text{All are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n]}_{\text{All are } \geq A[s]}$$

- 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**

Quick Sort

- Now note the difference between the working of Merge sort and Quick sort
- In **Merge sort** the division of the problem into two subproblems is immediate and the entire work happens in combining their solutions;
- In **Quick sort**, the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

Quick Sort

Pivot Element

- There are a number of ways to pick the pivot element.
- In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100
----	----	----	----	----	----	---	----	-----

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

40	20	10	80	60	50	7	30	100
----	----	----	----	----	----	---	----	-----

Pivot

Quick Sort

Let the pivot element be v , i.e., $v=a[1]$, $i=1, j=n$

Following are the rules

Increment i as long as $a[i] \leq v$

Decrement j as long as $a[j] \geq v$

Compare i and j ,

i.e., **if** ($i < j$) then interchange $a[i]$ and $a[j]$

else interchange v and $a[j]$

1	2	3	4	5	6	7	8	9
40	20	10	80	60	50	7	30	100

v
 \nearrow
 i

\nwarrow
 j

Quick Sort

Let the pivot element be v , i.e., $v=a[1]$, $i=1, j=n$

Following are the rules

Increment i as long as $a[i] \leq v$

Decrement j as long as $a[j] \geq v$

Compare i and j ,

i.e., **if** ($i < j$) then interchange $a[i]$ and $a[j]$

else interchange v and $a[j]$

1	2	3	4	5	6	7	8	9
40	20	10	80	60	50	7	30	100

v points to the first element (40).
 i points to the second element (20).
 j points to the ninth element (100).

Quick Sort

Let the pivot element be v , i.e., $v=a[1]$, $i=1, j=n$

Following are the rules

Increment i as long as $a[i] \leq v$

Decrement j as long as $a[j] \geq v$

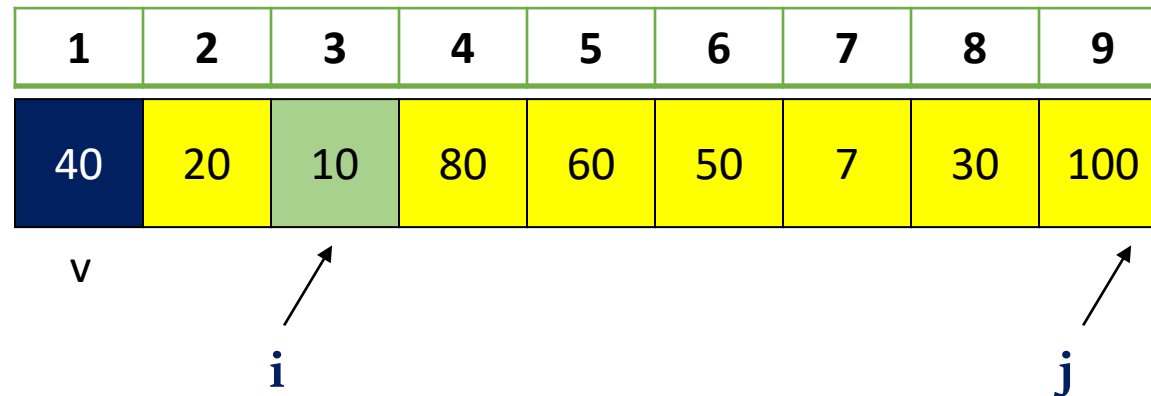
Compare i and j ,

i.e., **if** ($i < j$) then interchange $a[i]$ and $a[j]$

else interchange v and $a[j]$

1	2	3	4	5	6	7	8	9
40	20	10	80	60	50	7	30	100

v i j



Quick Sort

Let the pivot element be v , i.e., $v=a[1]$, $i=1, j=n$

Following are the rules

Increment i as long as $a[i] \leq v$

Decrement j as long as $a[j] \geq v$

Compare i and j ,

i.e., **if**($i < j$) then interchange $a[i]$ and $a[j]$

else interchange v and $a[j]$

1	2	3	4	5	6	7	8	9
40	20	10	80	60	50	7	30	100

v
 i
 j

STOP !!!!

Start moving j

Quick Sort

Let the pivot element be v , i.e., $v=a[1]$, $i=1, j=n$

Following are the rules

Increment i as long as $a[i] \leq v$

Decrement j as long as $a[j] \geq v$

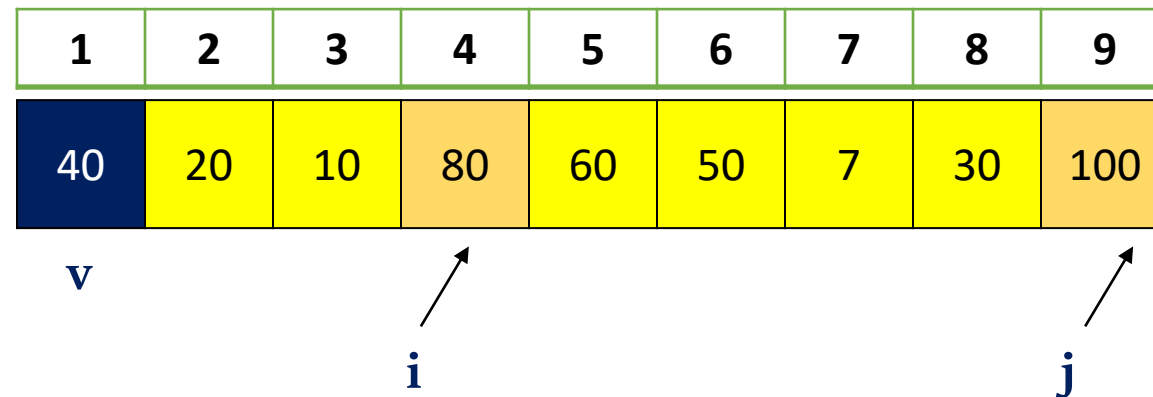
Compare i and j ,

i.e., **if**($i < j$) then interchange $a[i]$ and $a[j]$

else interchange v and $a[j]$

1	2	3	4	5	6	7	8	9
40	20	10	80	60	50	7	30	100

v
 i
 j



Following are the rules

Increment **i** as long as **a[i] ≤ v**

Decrement **j** as long as **a[j] ≥ v**

Compare **i** and **j**,

i.e., **if**($i < j$) then interchange **a**[i] and **a**[j]

```
else interchange v and a[j]
```

V

i

j

STOP !!!

$i < j$?

Yes !!! Exchange $a[i]$ and $a[j]$

Quick Sort

Let the pivot element be v , i.e., $v=a[1]$, $i=1, j=n$

Following are the rules

Increment i as long as $a[i] \leq v$

Decrement j as long as $a[j] \geq v$

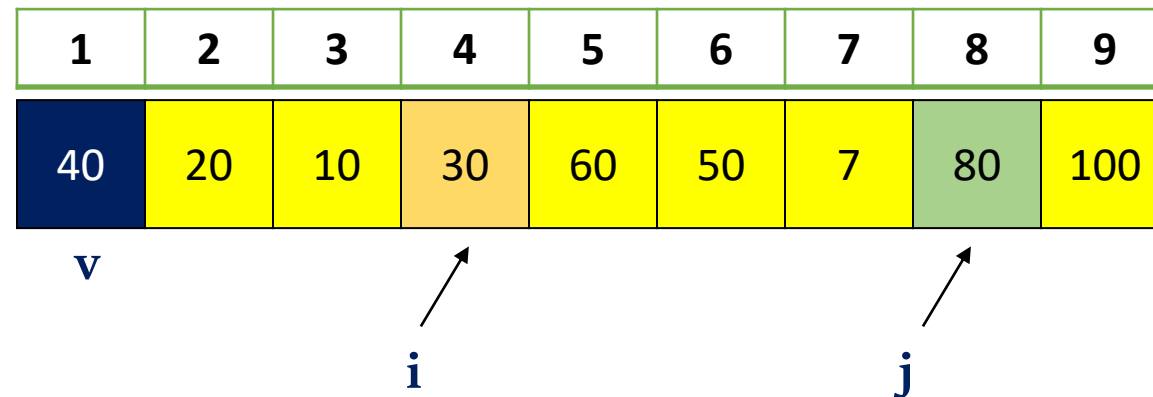
Compare i and j ,

i.e., **if**($i < j$) then interchange $a[i]$ and $a[j]$

else interchange v and $a[j]$

1	2	3	4	5	6	7	8	9
40	20	10	30	60	50	7	80	100

v
 i
 j



Quick Sort

Let the pivot element be v , i.e., $v=a[1]$, $i=1, j=n$

Following are the rules

Increment i as long as $a[i] \leq v$

Decrement j as long as $a[j] \geq v$

Compare i and j ,

i.e., **if**($i < j$) then interchange $a[i]$ and $a[j]$

else interchange v and $a[j]$

1	2	3	4	5	6	7	8	9
40	20	10	30	60	50	7	80	100

v

i

j

STOP !!!!

Start moving j

Quick Sort

Let the pivot element be v , i.e., $v=a[1]$, $i=1, j=n$

Following are the rules

Increment i as long as $a[i] \leq v$

Decrement j as long as $a[j] \geq v$

Compare i and j ,

i.e., **if**($i < j$) then interchange $a[i]$ and $a[j]$

else interchange v and $a[j]$

1	2	3	4	5	6	7	8	9
40	20	10	30	60	50	7	80	100

v
 i
 j

STOP !!!!

$i < j$?

Yes !!! Exchange $a[i]$ and $a[j]$

Quick Sort

Let the pivot element be v , i.e., $v=a[1]$, $i=1, j=n$

Following are the rules

Increment i as long as $a[i] \leq v$

Decrement j as long as $a[j] \geq v$

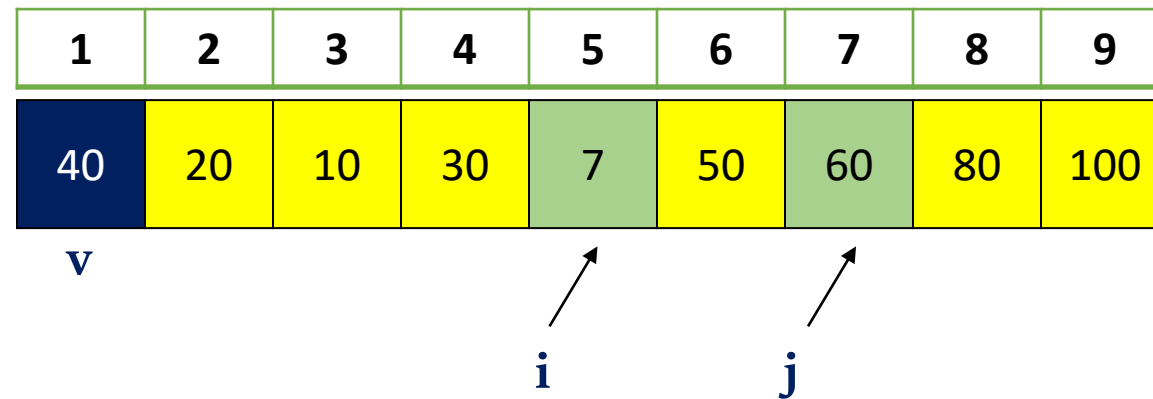
Compare i and j ,

i.e., **if**($i < j$) then interchange $a[i]$ and $a[j]$

else interchange v and $a[j]$

1	2	3	4	5	6	7	8	9
40	20	10	30	7	50	60	80	100

v
 i
 j



Quick Sort

Let the pivot element be v , i.e., $v=a[1]$, $i=1, j=n$

Following are the rules

Increment i as long as $a[i] \leq v$

Decrement j as long as $a[j] \geq v$

Compare i and j ,

i.e., **if**($i < j$) then interchange $a[i]$ and $a[j]$

else interchange v and $a[j]$

1	2	3	4	5	6	7	8	9
40	20	10	30	7	50	60	80	100

v

i

j

STOP !!!! Start moving j

Quick Sort

Let the pivot element be v , i.e., $v=a[1]$, $i=1, j=n$

Following are the rules

Increment i as long as $a[i] \leq v$

Decrement j as long as $a[j] \geq v$

Compare i and j ,

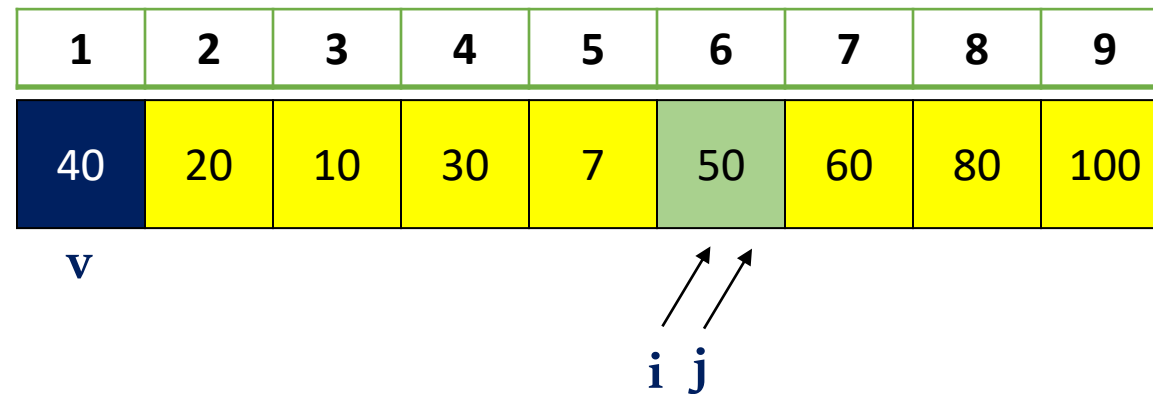
i.e., **if**($i < j$) then interchange $a[i]$ and $a[j]$

else interchange v and $a[j]$

1	2	3	4	5	6	7	8	9
40	20	10	30	7	50	60	80	100

v

 i j



Quick Sort

Let the pivot element be v , i.e., $v=a[1]$, $i=1, j=n$

Following are the rules

Increment i as long as $a[i] \leq v$

Decrement j as long as $a[j] \geq v$

Compare i and j ,

i.e., **if**($i < j$) then interchange $a[i]$ and $a[j]$

else interchange v and $a[j]$

1	2	3	4	5	6	7	8	9
40	20	10	30	7	50	60	80	100

v

j i

STOP !!!!

$i < j$?

No !!!!

Exchange $a[j]$ with pivot element

Quick Sort

Let the pivot element be v , i.e., $v=a[1]$, $i=1, j=n$

Following are the rules

Increment i as long as $a[i] \leq v$

Decrement j as long as $a[j] \geq v$

Compare i and j ,

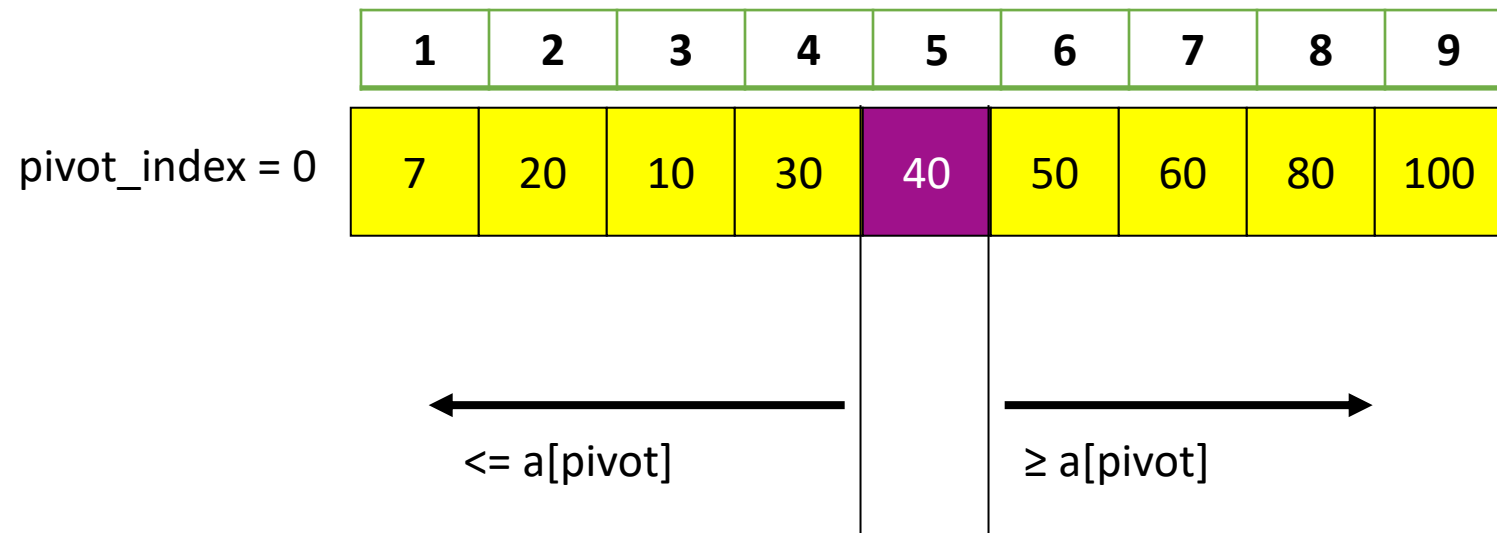
i.e., **if**($i < j$) then interchange $a[i]$ and $a[j]$

else interchange v and $a[j]$

1	2	3	4	5	6	7	8	9
7	20	10	30	40	50	60	80	100

Quick Sort

Got the first partition !!!!!!!



Quick Sort

Do it yourself

- Obtain the first partition for the following set of elements considering the first element as the pivot element

65, 70, 75, 80, 85, 60, 55, 50, 45

- Apply quicksort to sort the list **E, X, A, M, P, L, E** in alphabetical order

Quick Sort

```

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

Quick Sort

```
1  Algorithm Partition( $a, m, p$ )
2  // Within  $a[m], a[m + 1], \dots, a[p - 1]$  the elements are
3  // rearranged in such a manner that if initially  $t = a[m]$ ,
4  // then after completion  $a[q] = t$  for some  $q$  between  $m$ 
5  // and  $p - 1$ ,  $a[k] \leq t$  for  $m \leq k < q$ , and  $a[k] \geq t$ 
6  // for  $q < k < p$ .  $q$  is returned. Set  $a[p] = \infty$ .
7  {
8       $v := a[m]; i := m; j := p;$ 
9      repeat
10     {
11         repeat
12              $i := i + 1;$ 
13         until ( $a[i] \geq v$ );
14
15         repeat
16              $j := j - 1;$ 
17         until ( $a[j] \leq v$ );
18
19         if ( $i < j$ ) then Interchange( $a, i, j$ );
20     } until ( $i \geq j$ );
21
22      $a[m] := a[j]; a[j] := v;$  return  $j$ ;
23 }
```

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

Quick Sort – Analysis (Best Case)

- The recurrence relation is given by

$$T(n) = \begin{cases} a & n = 1, a \text{ is a constant} \\ 2T\left(\frac{n}{2}\right) + n & n > 1, c \text{ is a constant} \end{cases}$$

• Solution

In the given relation $a=2$, $b=2$, $f(n)=cn$, n is power of b so $n=b^k$, $n=2^k$

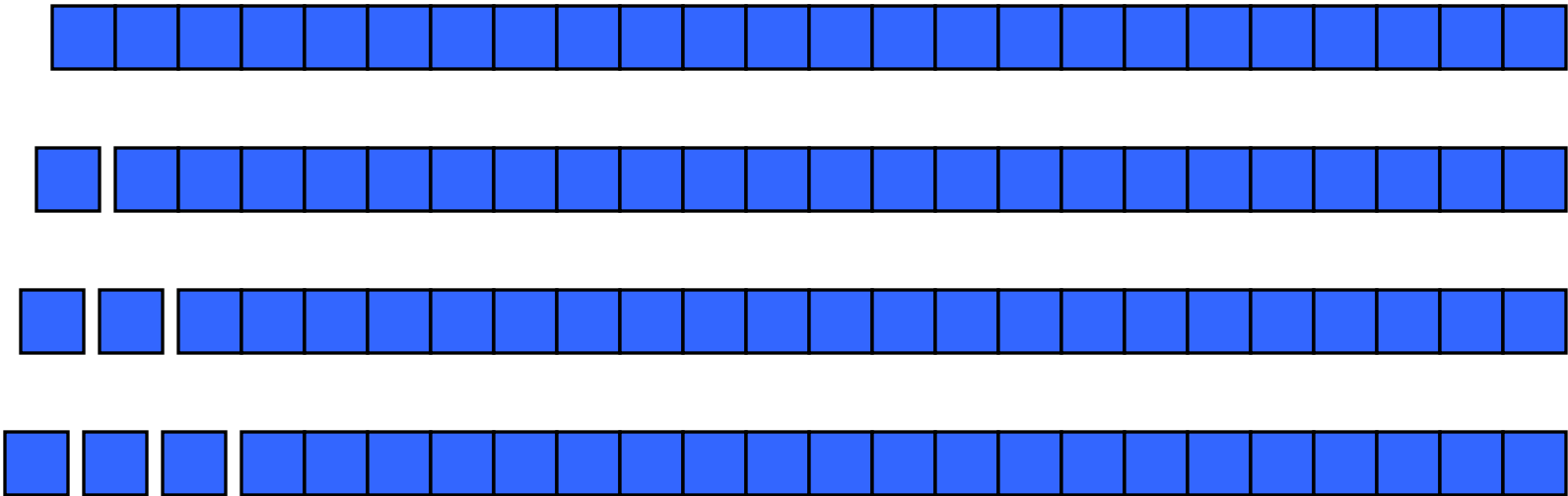
$$\begin{aligned} T(n) &= 2T(n/2) + n && \text{substitute } T(n/2) = 2T(n/4) + (n/2) \\ &= 2[2T(n/4) + n/2] + n \\ &= 4T(n/4) + 2n && \text{substitute } T(n/4) = 2T(n/8) + (n/4) \\ &= 4[2T(n/8) + n/4] + 2n \\ &= 8T(n/8) + 3n \end{aligned}$$

The general pattern ?

$$\begin{aligned} &= 2^k T(n/2^k) + kn && n=2^k, \quad k=\log n \\ &= nT(1) + \log n \cdot n \\ &= n \log n \end{aligned}$$

- $T(n)_{\text{Best}} = \Omega(n \log n)$**

Quick Sort- Analysis (Worst Case)



Quick Sort- Analysis (Worst Case)

- The recurrence relation for worst case analysis is given by

$$T(n) = 0 + T(n-1) + n$$

Advantages

- Algorithm is in-place since it uses small auxiliary stack
- Quick sort requires time complexity of $O(n \log n)$ in the best case and average case to sort n items.

Disadvantages

- Algorithm is not stable
- Time complexity is quadratic $O(n^2)$ in worst case.
- It is fragile.



RNSIT-CSE
Bengaluru

Divide and Conquer Module 2



Reshma Jakabal

Assistant Professor

Department of CS&E, RNSIT

Strassen's Matrix Multiplication

- Let A and B be two $n \times n$ matrices
- The product matrix $C = AB$ is also an $n \times n$ matrix whose i, j^{th} element is formed by taking the elements in the i^{th} row of A and j^{th} column of B and multiplying them to get
- $C(i, j) = \sum_{1 \leq k \leq n} A(i, k)B(k, j)$ for all i and j between 1 and n
- To compute $C(i, j)$ using the formula above how many multiplications are needed?
- Consider an example

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \quad \text{then } C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

- Where,

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}$$

$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$

$$c_{21} = a_{21}b_{11} + a_{22}b_{21}$$

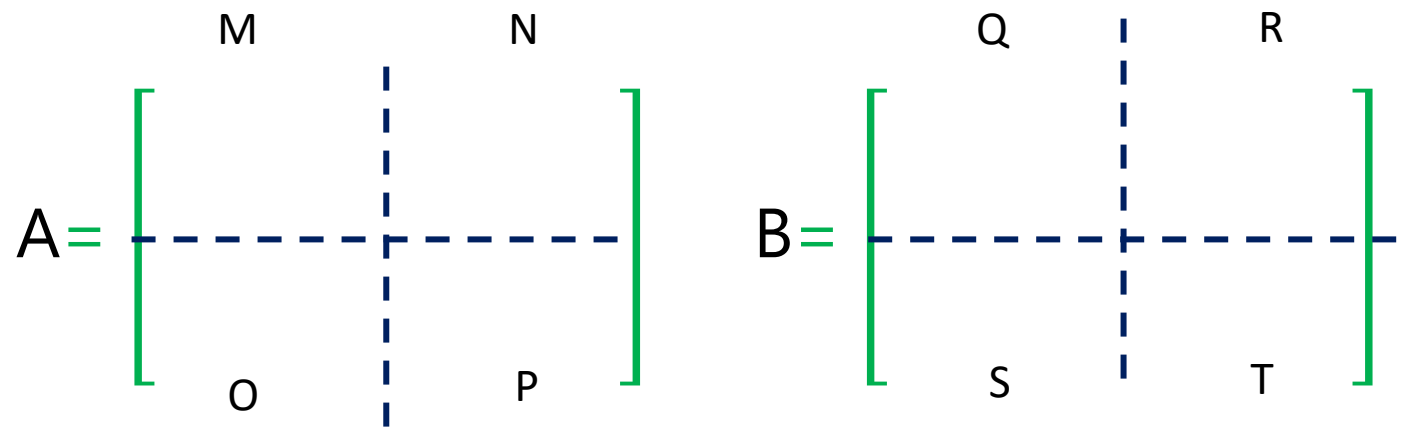
$$c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

8 multiplications

Time complexity ? $\Theta(n^3)$

Strassen's Matrix Multiplication

- Can we use **Divide and Conquer** approach to multiply two **$n \times n$** matrices ?
- Let's assume that **n** is power of **2**, i.e., there exists a non-negative constant **k** such that **$n=2^k$**
- If **n** is not power of **2** then add enough rows and columns of **zeros** to both **A** and **B** so that the resultant dimensions are power of **2**.
- Here is the application of Divide and Conquer approach
- In this method matrix A and B are splitted into 4 squares sub-matrices where each sub-matrices has dimension of $n/2$.



Strassen's Matrix Multiplication

- Consider the following situation

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

- Then

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned}$$

- Where

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

- Consider the following example:

- 1 2 6 8
- 4 5 * 7 9

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

- Now we want to find C12, this can be done as shown:-

- $R = 1 * (8 - 9) = -1,$
- $T = (1 + 2) * 9 = 27,$
- $C_{12} = -1 + 27 = 26$

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned}$$

- Similarly, C21 can be found as shown
- $Q = (4 + 5) * 6 = 54$
- $S = 5 * (7 - 6) = 5$
- $C_{21} = 54 + 5 = 59$

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

Strassen's Matrix Multiplication

- Consider the following matrices and compute the product using Strassen's Method
- $A = \begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix}$ $B = \begin{bmatrix} 1 & 3 \\ 4 & 7 \end{bmatrix}$

Analysis of Strassen's Multiplication

$$\bullet T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 7T\left(\frac{n}{2}\right) & \text{otherwise} \end{cases}$$

Finding minimum and maximum

- Problem statement: The problem is to find the maximum and minimum items in a set of n elements.

```
max=min =A[0]
  for i ←1 to n – 1 do
    if (A[i]>max)
      max=a[i];
    else if (A[i]<min)
      min=a[i];
  end for
```

Straight forward Approach

```
1  Algorithm StraightMaxMin(a, n, max, min)
2  // Set max to the maximum and min to the minimum of a[1 : n].
3  {
4      max := min := a[1];
5      for i := 2 to n do
6          {
7              if (a[i] > max) then max := a[i];
8              if (a[i] < min) then min := a[i];
9          }
10 }
```

Divide and Conquer Approach

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

Divide and Conquer for finding Max and Min

1. Write a recursive function accepting the array and its start and end index as parameters
2. The base cases will be
 - If array size is 1, return the element as both max and min
 - If array size is 2, compare the two elements and return maximum and minimum
3. The recursive part is
 - Recursively calculate and store the maximum and minimum for left and right parts
 - Determine the maximum and minimum among these by 2 comparisons
4. Return max and min.

- Method 1: if we apply the **general approach** to the array of size n , the number of comparisons required are $2n-2$.
- Method-2: In divide n conquer approach, we will divide the problem into sub-problems and find the max and min of each group, now max. Of each group will compare with the only max of another group and min with min.
- Let n = is the size of items in an array
- Let $T(n)$ = time required to apply the algorithm on an array of size n . Here we divide the terms as $T(n/2)$.

- $T(n) = T(n/2) + T(n/2) + 2$
- $T(n) = 2 T(n/2) + 2$
- $T(2) = 1$ time required to compare two elements/items
- $T(1) = 0$
- We can solve this recurrence relation if n is a power of 2.
- $T(n) = 2 T(n/2) + 2 \quad \rightarrow \text{Eq (i)}$
- $T\left(\frac{n}{2}\right) = 2 T\left(\frac{n}{2^2}\right) + 2 \quad \rightarrow \text{Eq (ii)}$
- Put Eq (ii) in Eq (i)

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

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

Similarly, apply the same procedure recursively on each subproblem

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + 2^i + 2^{i-1} + \dots + 2 \dots \dots \text{(Eq. 3)}$$

$$\frac{n}{2^i} = 2 \Rightarrow n = 2^{i+1}$$

$$\begin{aligned}
 T(n) &= 2^i T(2) + 2^i + 2^{i-1} + \dots + 2 \\
 &= 2^i \cdot 1 + 2^i + 2^{i-1} + \dots + 2 \\
 &= 2^i + \frac{2(2^i - 1)}{2 - 1} \\
 &= 2^{i+1} + 2^i - 2 \\
 &= n + \frac{n}{2} - 2 \\
 &= \frac{3n}{2} - 2
 \end{aligned}$$

- Number of comparisons requires applying general approach on n elements = $2n-2$
- We can analyze, that how to reduce the number of comparisons by using this technique.
- Analysis: suppose we have the array of size 8 elements.
- Method1: requires $(2n-2)$, $(2 \times 8) - 2 = 14$ comparisons
- Method2: $\frac{3 \times 8}{2} - 2 = 10$ *comparisons*

Advantages:

- The difficult problem is broken down to sub problems and each problem is solved separately
- and independently. This is useful for obtaining solutions in easier way for difficult problems.
- This technique facilitates the discovers of new efficient algorithms. Example: Quick sort, Merge sort etc.
- The sub problems can be executed on parallel processor.
- Hence time complexity can be reduced.

Disadvantages

- Large number of sub lists are created and need to be processed
- This algorithm makes use of recursive methods and the recursion is slow and complex.
- Difficulties in solving larger size of inputs

Decrease and Conquer

- This technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance of the same problem.
- Once such relationship is established, it can be exploited either top down (recursively) or bottom up (without a recursion).
- There are three major variations of decrease-and-conquer:
 - Decrease by a constant.
 - Decrease by a constant factor.
 - Variable size decrease.

Decrease by a constant.

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

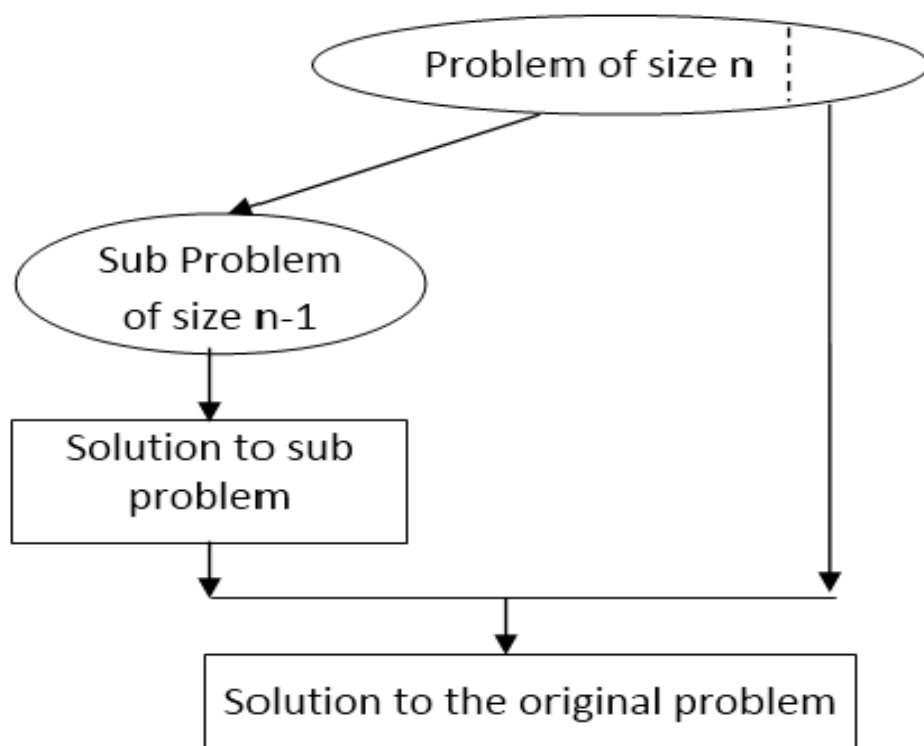


Figure: Decrease-(by one)-and-conquer technique

Example: $a^n = a^{n-1} \times a$

Decrease by a constant factor

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

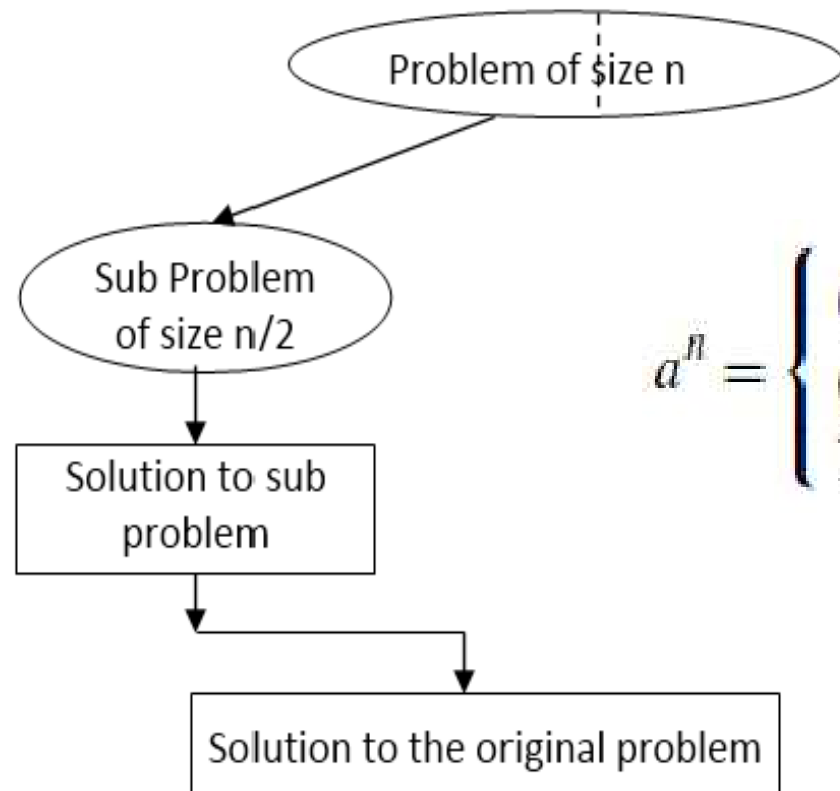


Figure: Decrease-(by half)-and-conquer technique

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$

Variable size decrease

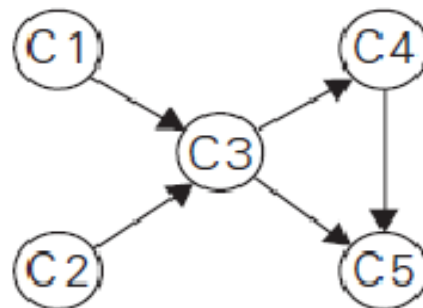
- In the variable-size-decrease variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another.
- Example: Euclid's algorithm for computing the greatest common divisor. It is based on the formula.
- $\text{gcd}(m, n) = \text{gcd}(n, m \bmod n)$.

Topological Sorting

- A directed graph, or digraph , is a graph with directions specified for all its edges.
- For a directed acyclic graph $G=(V,E)$
- A topological sort is an ordering of all of G 's vertices $v_1, v_2, v_3, \dots, v_n$ such that ...
- Vertex u comes before vertex v if edge (u,v) belongs to G

Motivation for topological sorting

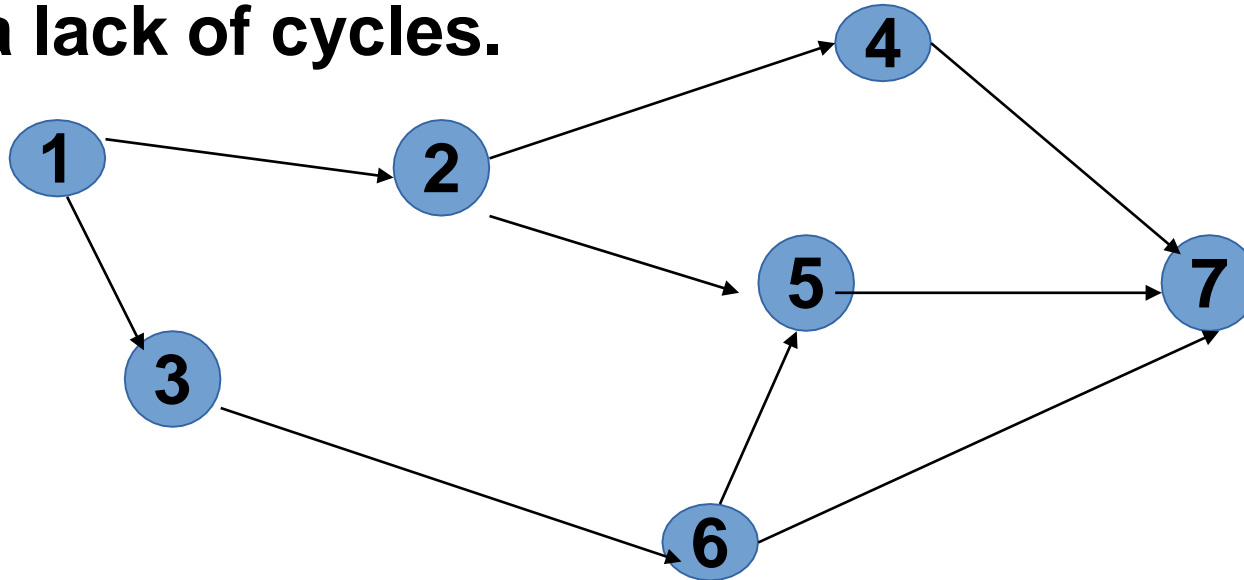
- Consider a set of five required courses $\{C1, C2, C3, C4, C5\}$ a part-time student has to take in some degree program. The courses can be taken in any order as long as the following course prerequisites are met: C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4. The student can take only one course per term. In which order should the student take the courses? The situation can be modeled by a digraph in which vertices represent courses and directed edges indicate prerequisite requirements.



- In terms of this digraph, the question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends this problem is called topological sorting.
- Topological Sort: For topological sorting to be possible, a digraph in question must be a DAG. i.e., if a digraph has no directed cycles, the topological sorting problem for it has a solution.
- There are two efficient algorithms that both verify whether a **digraph** is a **dag** and, if it is, produce an ordering of vertices that solves the topological sorting problem. The first one is based on depth-first search; the second is based on a direct application of the decrease-by-one technique.

Directed Acyclic Graph

A directed acyclic graph is an acyclic graph that has a direction as well as a lack of cycles.



Vertices Set:
 $\{1,2,3,4,5,6,7\}$

Edge Set: $\{(1,2),(1,3),(2,4),$
 $(2,5),(3,6),(4,7),$
 $(5,7),(6,7)\}$

A directed acyclic graph has a topological ordering. This means that the nodes are ordered so that the starting node has a lower value than the ending node.

Definition:

A topological sort or topological ordering of a directed acyclic graph is a linear ordering of its vertices such that for every directed edge (u,v) from vertex u to v , u comes before v in the ordering.

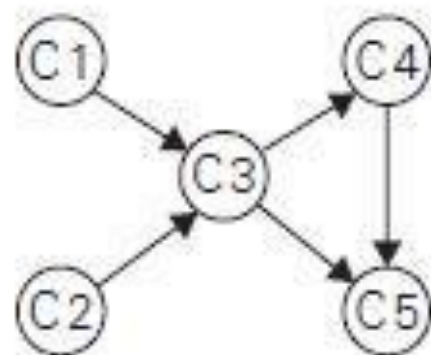
- There are two commonly used algorithms implementing using topological sort method.

1. DFS based algorithm

2. Source Removal Algorithm

Algorithm1: (DFS method)

- ***Procedure:***
- Perform a DFS traversal and note the order in which vertices become dead-ends (i.e., popped off the traversal stack). Reversing this order yields a solution to the topological sorting problem, provided, ofcourse, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.



(a)

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

(b)

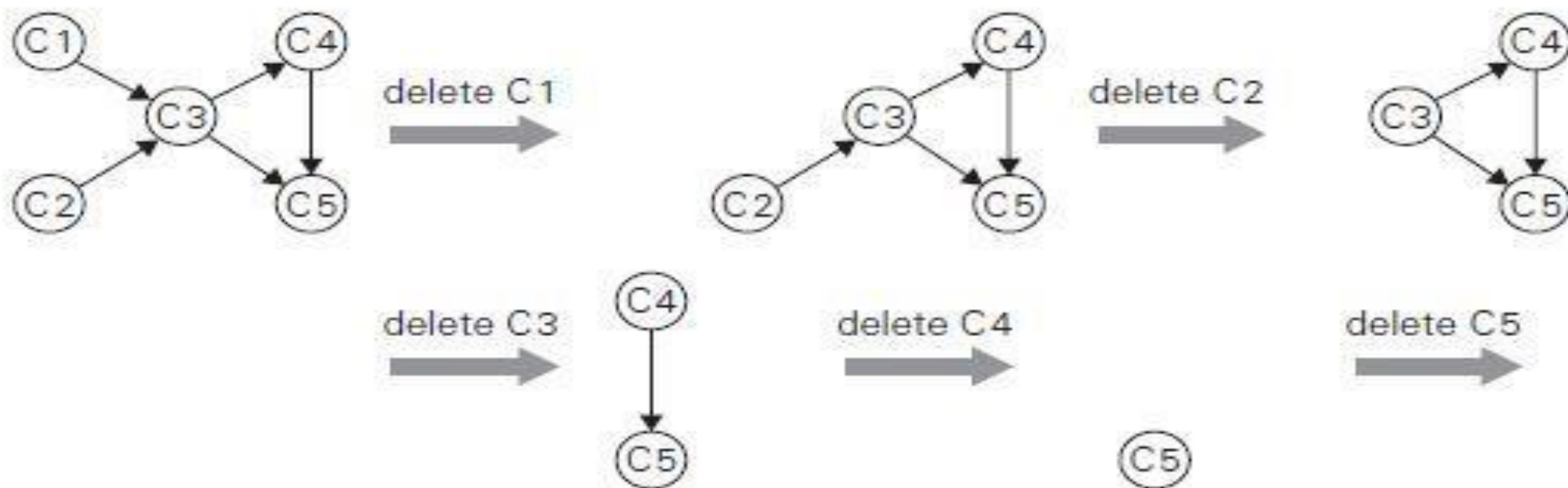
The popping-off order:
 C5, C4, C3, C1, C2
 The topologically sorted list:
 $C2 \rightarrow C1 \rightarrow C3 \rightarrow C4 \rightarrow C5$

(c)

FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

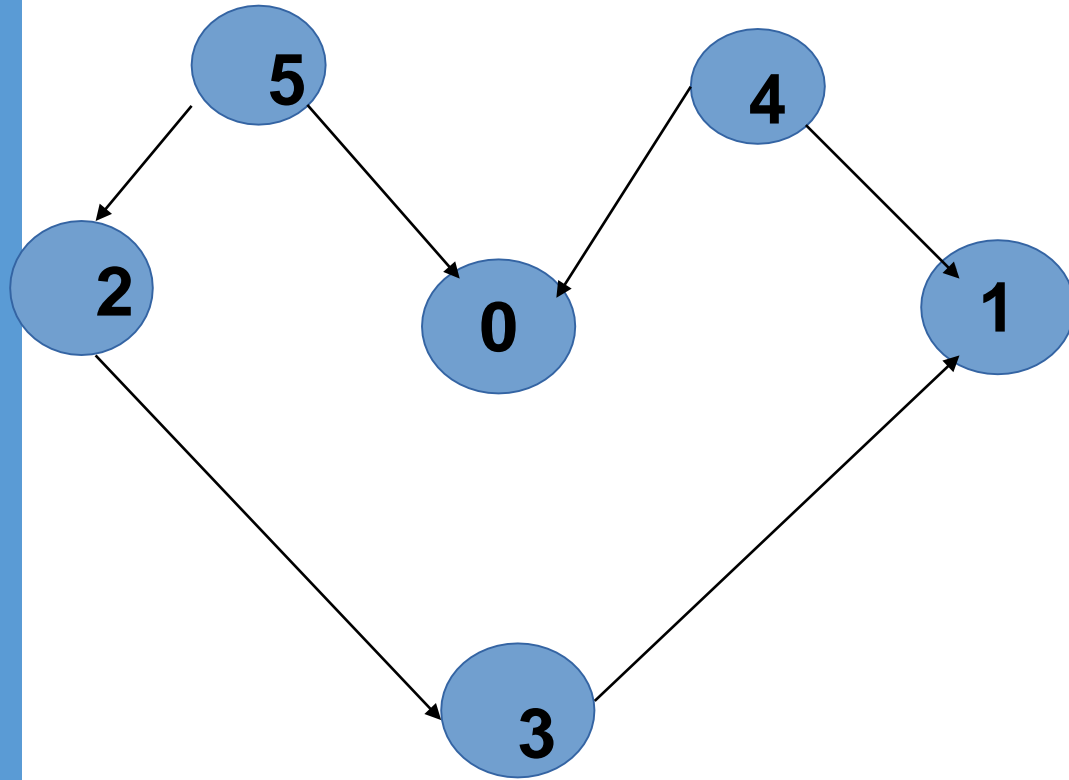
Algorithm2: (Source removal method)

- ***Procedure:***
- Repeatedly, identify in a remaining digraph a source, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. The order in which the vertices are deleted yields a solution to the topological sorting problem.



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

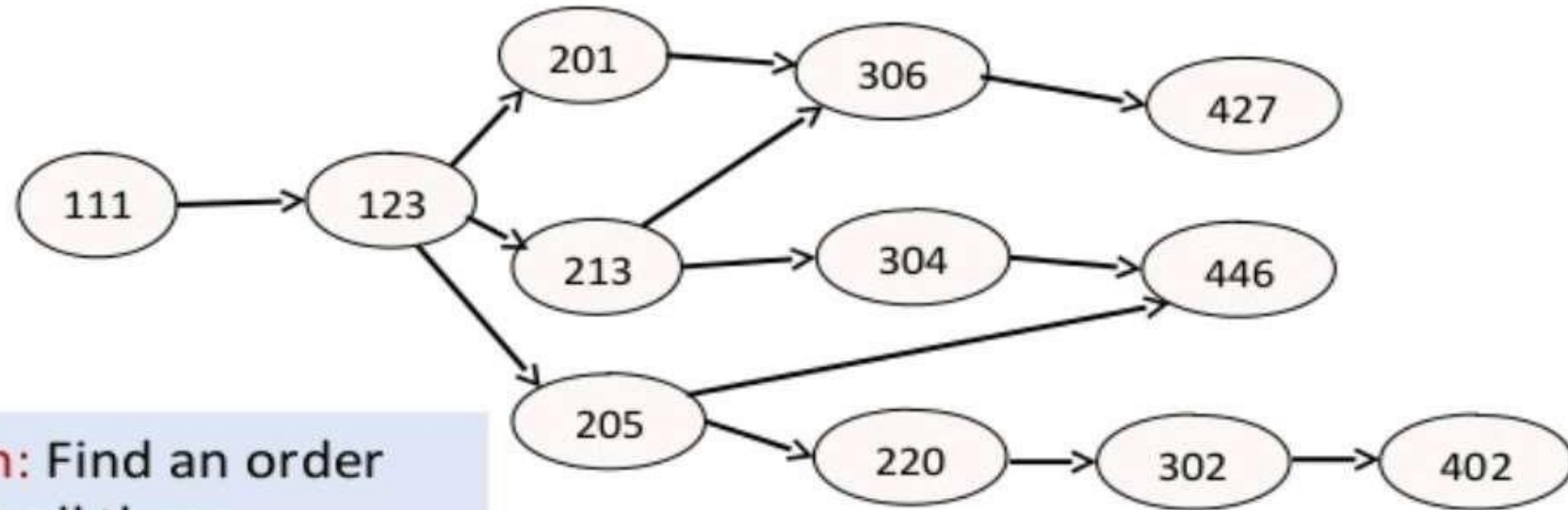
FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.



For example, a topological sorting of the given graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example , another topological sorting for the graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as “0” (a vertex with no incoming edges).

- So, the Topological Sorting is NOT UNIQUE.
- Also it can only be applied to Directed Acyclic Graphs(DAG).

- Consider the following graph of course prerequisites



Problem: Find an order in which all these courses can be taken.

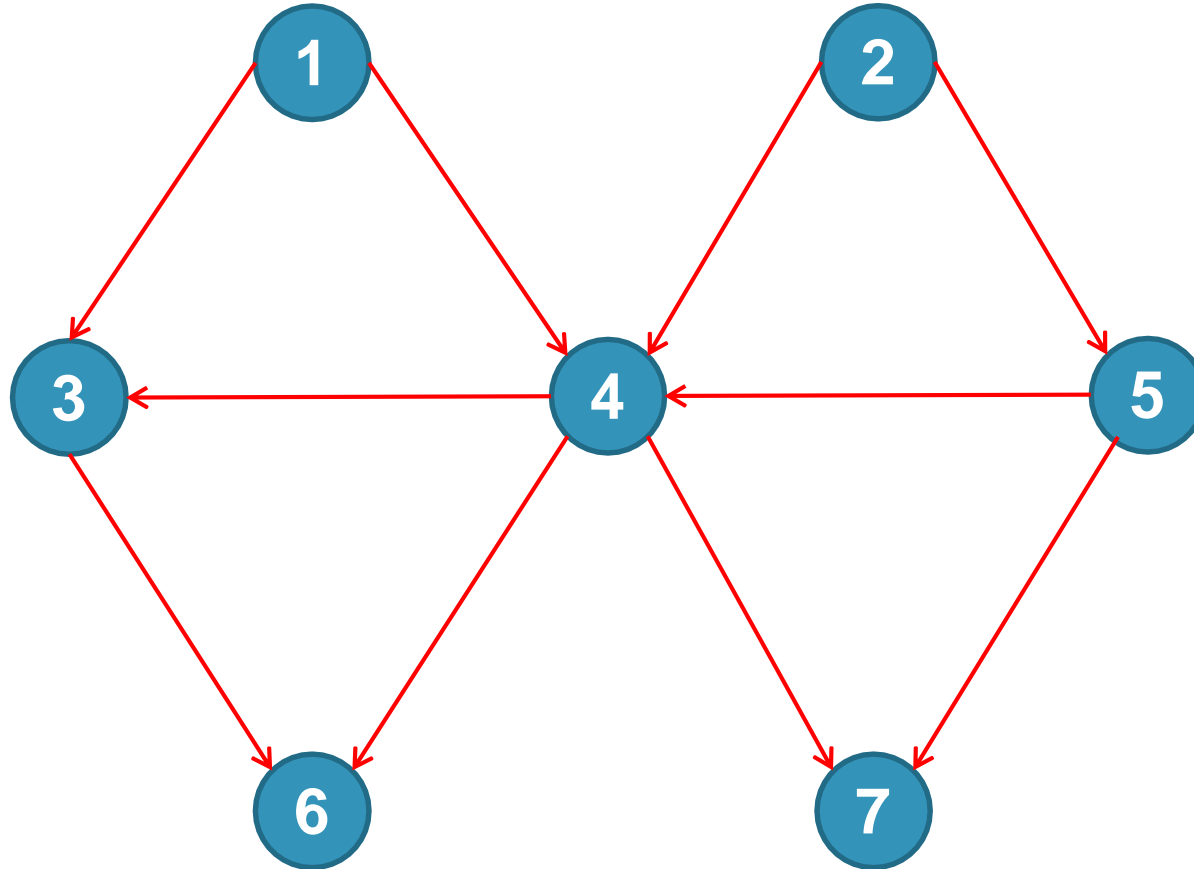
Example: 111, 123, 201, 213, 304, 306, 427, 205, 446, 220, 302, 402

- To take a course, **all** of its prerequisites must be taken first

Algorithm

- 1 - Compute the indegrees of all vertices
- 2 - Find a vertex U with indegree 0 and print it (store it in the ordering) If there is no such vertex then there is a cycle and the vertices cannot be ordered. Stop.
- 3 - Remove U and all its edges (U, V) from the graph.
- 4 - Update the indegrees of the remaining vertices.
- 5 - Repeat steps 2 through 4 while there are vertices to be processed

Example



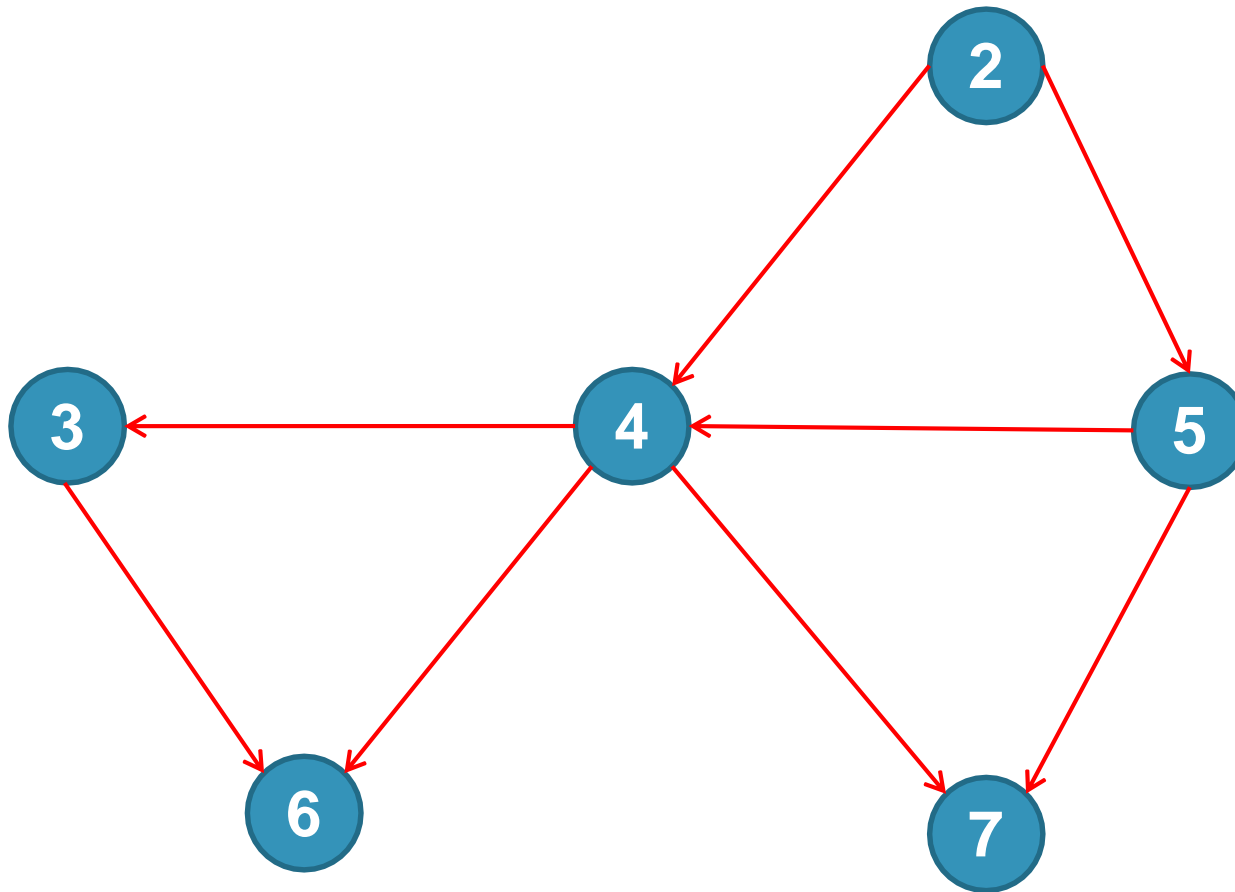
Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

Select Node : 1

Output :

1



Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

Select Node : 2

Output :



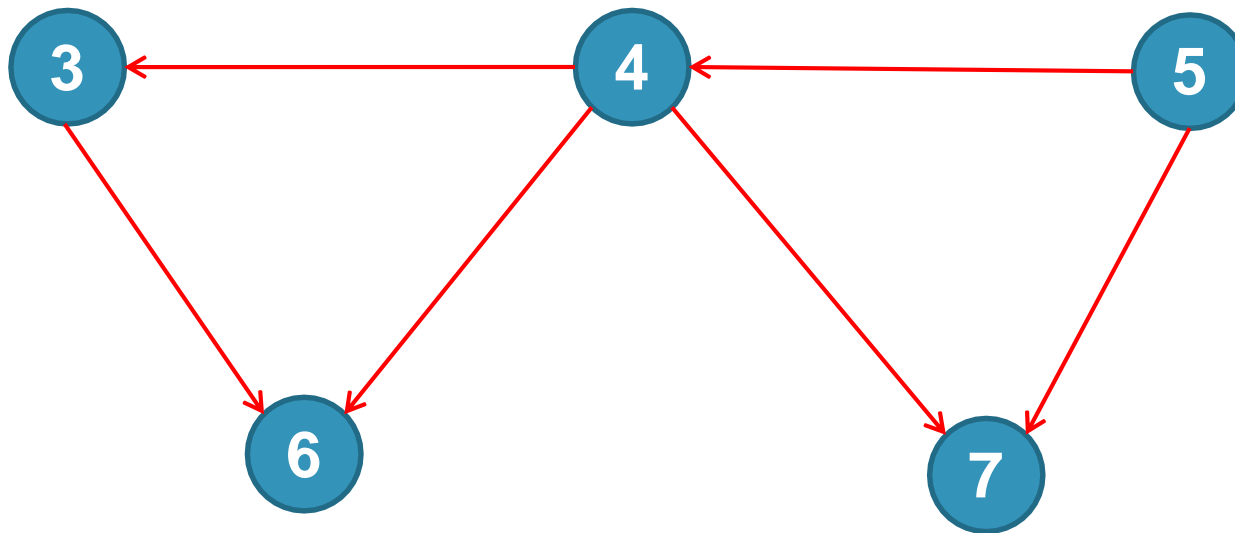
Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

Select Node : 5

Output :

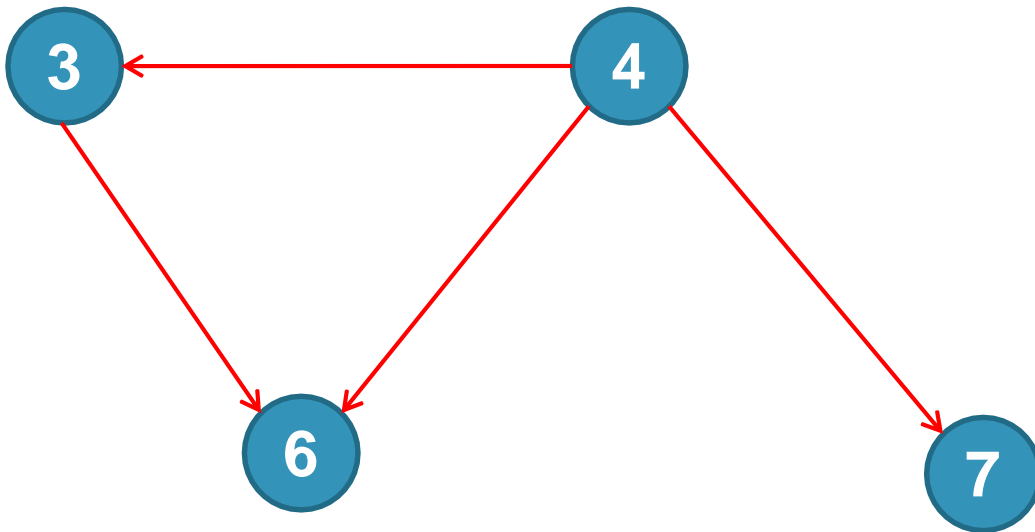


Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

Select Node : 4

Output :



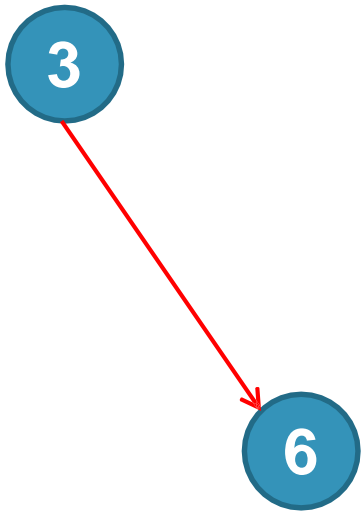
Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

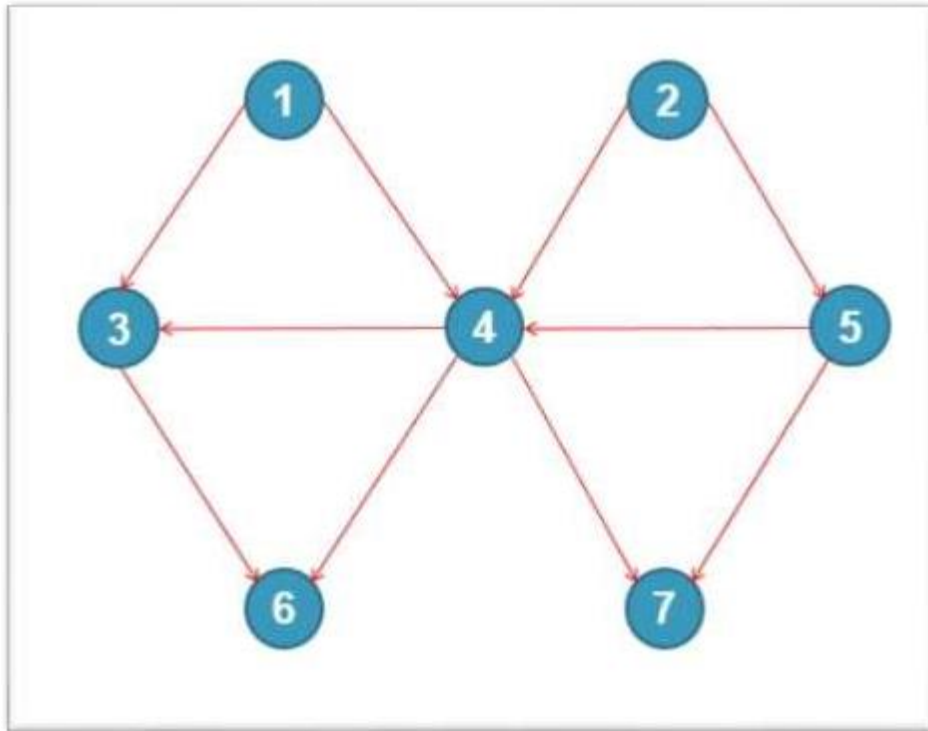
Select Node : 3

Output :

1 2 5 4 3



Contd...



6

7

Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

Select Node :6 7

Output :

1 2 5 4 3 6 7

1] Build Systems :

- We have various IDE's like Eclipse , Netbeans etc. We have to build a project which has many libraries dependent on each other then IDE uses Topological Sort to decide which library to include or build first.

2] Task Scheduling :

- Topological Sort is helpful in scheduling interdependent task to know which task should proceed which one.

31 Pre- Requisite Problems:

- We often come across situations where we need to finish one job in order to proceed the next one. For ex, In University structure, we need to complete basic Algorithm course to study an Advance Algorithm course. So, there exist a pre- requisite and we can know this by doing a topological sort on all.



THANK
YOU