

# Chapter 3

## Sorting Selection and Sequencing

Prepared By:  
Er. Amrit Poudel

# Divide and Conquer

A **divide and conquer** algorithm is a strategy of solving a large problem by

1. breaking the problem into smaller sub-problems
2. solving the sub-problems, and
3. combining them to get the desired output.

To use the divide and conquer algorithm, **recursion** is used.

# How Divide and Conquer Algorithms Work?

Here are the steps involved:

1. **Divide:** Divide the given problem into sub-problems using recursion.
2. **Conquer:** Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
3. **Combine:** Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

# Divide and Conquer Algorithm

**Algorithm** DAndC( $P$ )

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

- **Small( $P$ )** is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without splitting

## Computing time of **DAndC** Algorithm

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

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

# Computing time of DAndC Algorithm

The complexity of the divide and conquer algorithm is calculated using the [master theorem](#).

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

where,

- $n$  = size of input
- $a$  = number of subproblems in the recursion
- $n/b$  = size of each subproblem. All subproblems are assumed to have the same size.
- $f(n)$  = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solution

# Divide and Conquer(Merge Sort)

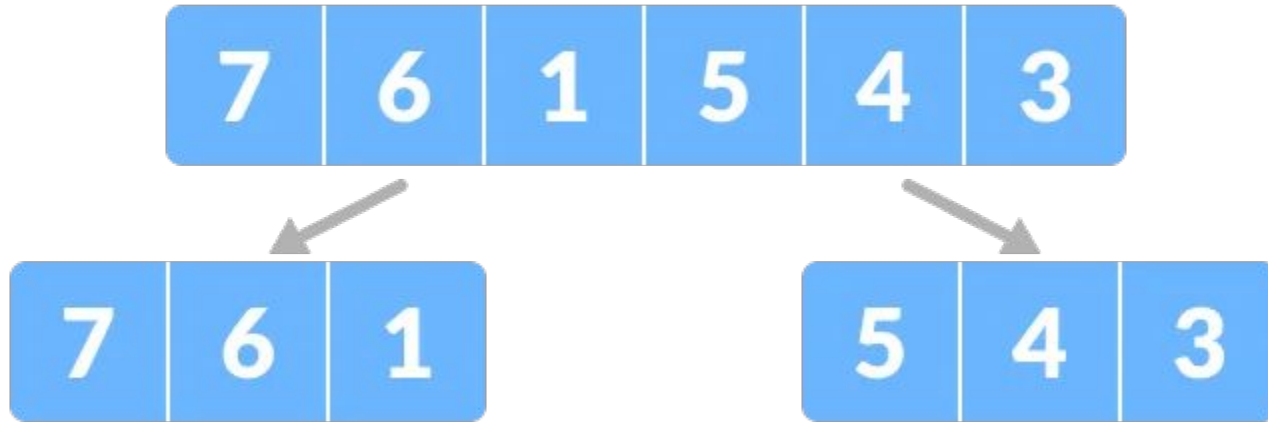
Here, we will sort an array using the divide and conquer approach (ie. [merge sort](#)).

1. Let the given array be:



# Divide and Conquer(Merge Sort)

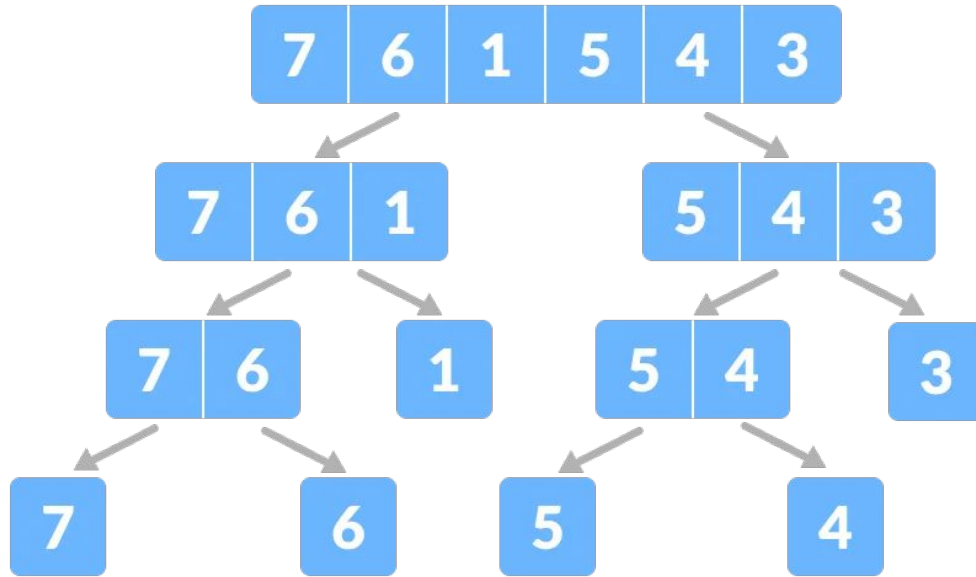
2. **Divide** the array into two halves.





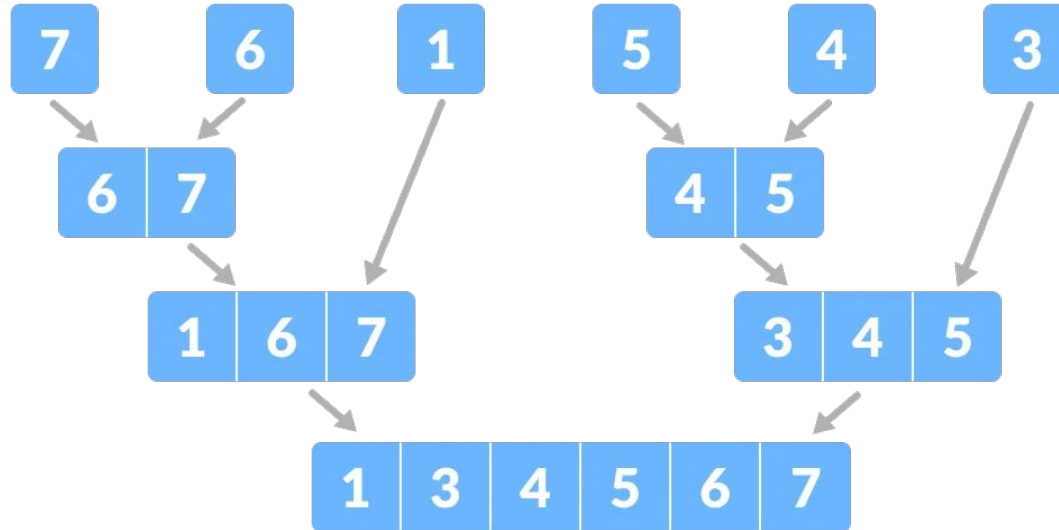
# Divide and Conquer(Merge Sort)

Again, divide each subpart recursively into two halves until you get individual elements.



# Divide and Conquer(Merge Sort)

3. Now, combine the individual elements in a sorted manner. Here, conquer and combine steps go side by side.



# Time Complexity

For a **merge sort**, the equation can be written as:

**Divide:** It is  $O(1)$  operation because the middle index can be calculated in constant time.

**Conquer:** It is the time complexity of recursively solving the one sub-problem of size  $n/2$  i.e.  $T(n/2)$ .

**Combine:** The time complexity of this part is  $O(n)$  because merging two sorted array or lists requires  $O(n)$  operation

The recurrence relation for the above is:  $T(n) = 2T(n/2) + O(n)$

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

$$\approx O(n \log n)$$

# Time Complexity

## Time Complexity of merge sort further explanation:

The list of size **N** is divided into a max of **logn** parts, and the merging of all sublists into a single list takes **O(N)** time, the worst-case run time of this algorithm is **O(nlogn)**

# Binary Search

- Let  $a_i, 1 \leq i \leq n$ , be a list of elements that are sorted in non decreasing order.
- Consider the problem of determining whether a given element  $x$  is present in the list. If  $x$  is present, we are to determine a value  $j$  such that  $a_j = x$ . If  $x$  is not in the list, then  $j$  is to be set to zero
- Let  $P = (n, a_1, \dots, a_n, x)$  denote an arbitrary instance of this search problem  
( $n$  is the number of elements in the list,  $a_1, \dots, a_n$  is the list of elements and  $x$  is the element searched for).
- Divide-and-conquer can be used to solve this problem.

# Recursive binary search

**Algorithm** BinSrch( $a, i, l, x$ )

// Given an array  $a[i : l]$  of elements in nondecreasing  
// order,  $1 \leq i \leq l$ , determine whether  $x$  is present, and  
// if so, return  $j$  such that  $x = a[j]$ ; else return 0.

```
{  
    if ( $l = i$ ) then // If Small( $P$ )  
    {  
        if ( $x = a[i]$ ) then return  $i$ ;  
        else return 0;  
    }  
    else  
    { // Reduce  $P$  into a smaller subproblem.  
         $mid := \lfloor (i + l) / 2 \rfloor$ ;  
        if ( $x = a[mid]$ ) then return  $mid$ ;  
        else if ( $x < a[mid]$ ) then  
            return BinSrch( $a, i, mid - 1, x$ );  
        else return BinSrch( $a, mid + 1, l, x$ );  
    }  
}
```

# Iterative Binary Search

**Algorithm** BinSearch( $a, n, x$ )

// Given an array  $a[1 : n]$  of elements in nondecreasing  
// order,  $n \geq 0$ , determine whether  $x$  is present, and  
// if so, return  $j$  such that  $x = a[j]$ ; else return 0.

```
{  
     $low := 1; high := n;$   
    while ( $low \leq high$ ) do  
    {  
         $mid := \lfloor (low + high)/2 \rfloor;$   
        if ( $x < a[mid]$ ) then  $high := mid - 1;$   
        else if ( $x > a[mid]$ ) then  $low := mid + 1;$   
        else return  $mid;$   
    }  
    return 0;  
}
```

# Example

- Let us select the 14 entries:

**-15, -60, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151**

- Place them in `a[1:14]`, and simulate the steps that BinSearch goes through as it searches for different values of `x`.
- Only the variable `slow`, `high`, and `mid` need to be traced as we simulate the algorithm
- Try the following values for `x`: 151, -14 and 9 for two successful searches and one unsuccessful search



## Binary Search Example

$x = 151$	<i>low</i>	<i>high</i>	<i>mid</i>		$x = -14$	<i>low</i>	<i>high</i>	<i>mid</i>
	1	14	7			1	14	7
	8	14	11			1	6	3
	12	14	13			1	2	1
	14	14	14			2	2	2
			found			2	1	not found
				$x = 9$	<i>low</i>	<i>high</i>	<i>mid</i>	
					1	14	7	
					1	6	3	
					4	6	5	
							found	

---

**Table 3.2** Three examples of binary search on 14 elements

# Time Complexity Calculations

- Comparisons between  $x$  and elements of  $a[]$  are referred to *element comparisons*.
- We assume that only one comparison is needed to determine which of the three possibilities of the if statement holds.
- The number of element comparisons needed to find each of the 14 elements is:

$a:$	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
Elements:	-15	-6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons:	3	4	2	4	3	4	1	4	3	4	2	4	3	4

# Time Complexity

- No element requires more than 4 comparisons to be found.
- The average is obtained by summing the comparisons needed to find all 14 items and dividing by 14; this yields  $45/14$ , or approximately 3.21 comparisons per successful search on the average.
- There are 15 possible ways that an unsuccessful search may terminate depending on the value of  $x$ .
- If  $x < a[1]$ , the algorithm requires 3 element comparisons to determine that  $x$  is not present.
- For all the remaining possibilities in, Search requires 4 element comparisons.
- Thus the average number of element comparisons for an unsuccessful search is  $(3 + 14 * 4)/15 = 59/15 = 3.93$ .

# Time Complexity for n elements

- **At Iteration 1**
  - Length of array =  $n$
- **At Iteration 2,**
  - Length of array =  $n/2$
- **At Iteration 3,**
  - Length of array =  $(n/2)/2 = n/2^2$
- **Therefore, after Iteration k,**
  - Length of array =  $n/2^k$
- **Also, we know that after**
  - After k iterations, the **length of array becomes 1**
- **Therefore**
  - **Length of array =  $n/2^k=1$**
  - **$\Rightarrow n = 2^k$**
- **Applying log function on both side**
  - $\Rightarrow \log_2 (n) = \log_2 (2^k)$
  - $\Rightarrow \log_2 (n) = k \log_2 (2)$
- **Therefore**
  - **$k = \log_2 (n)$**

## Time Complexity

In conclusion we are now able to completely describe the computing time of binary search by giving formulas that describe the best, average, and worst cases:

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

# Space Complexity

- Storage is required for the  $n$  elements of the array
- The storage for variables low, mid , high and x is needed
- Total space complexity=  $n+4$

# Finding the maximum and minimum

**Problem:** find the maximum and minimum items in a set of  $n$  elements

```
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 }
```

---

**Algorithm 3.5** Straightforward maximum and minimum

# Time Complexity

- The time complexity of straightforward algorithm is determined by the total cost of the element comparisons
- **StraightMaxMin** requires  $2(n - 1)$  element comparisons in the best average, and worst cases
- An immediate improvement is possible by realizing that the comparison  **$a[i] < min$**  is necessary only when  **$a[i] > max$**  is false. Hence we can replace the contents of the for loop by

```
if ( $a[i] > max$ ) then  $max := a[i]$ ;  
else if ( $a[i] < min$ ) then  $min := a[i]$ ;
```

- Now the best case occurs when the elements are in increasing order. The number of element comparisons is  **$n - 1$** .
- The worst case occurs when the elements are in decreasing order. In this case the number of element comparisons
- The number of comparisons can be reduced by using divide and conquer method



# Divide and Conquer Approach

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

## Time Complexity

$$\begin{aligned}T(n) &= 2T(n/2) + 2 \\&= 2(2T(n/4) + 2) + 2 \\&= 4T(n/4) + 4 + 2 \\&\vdots \\&= 2^{k-1}T(2) + \sum_{1 \leq i \leq k-1} 2^i \\&= 2^{k-1} + 2^k - 2 = 3n/2 - 2\end{aligned}$$

- $3n/2 - 2$  is the best, average and worst case number of comparisons when  $n$  is the power of two
- Compared with the  $2n - 2$  comparisons for the straightforward method, this is a saving of 25% in comparison

# Space Complexity

- MaxMin is worse than the straightforward algorithm because it requires stack space for  $i, j, max, min, max1$  and  $min1$ .
- Given  $n$  elements, there will be  $\lceil \log n \rceil + 1$  levels of recursion and we need to save seven values for each recursive call.

# Merge Sort

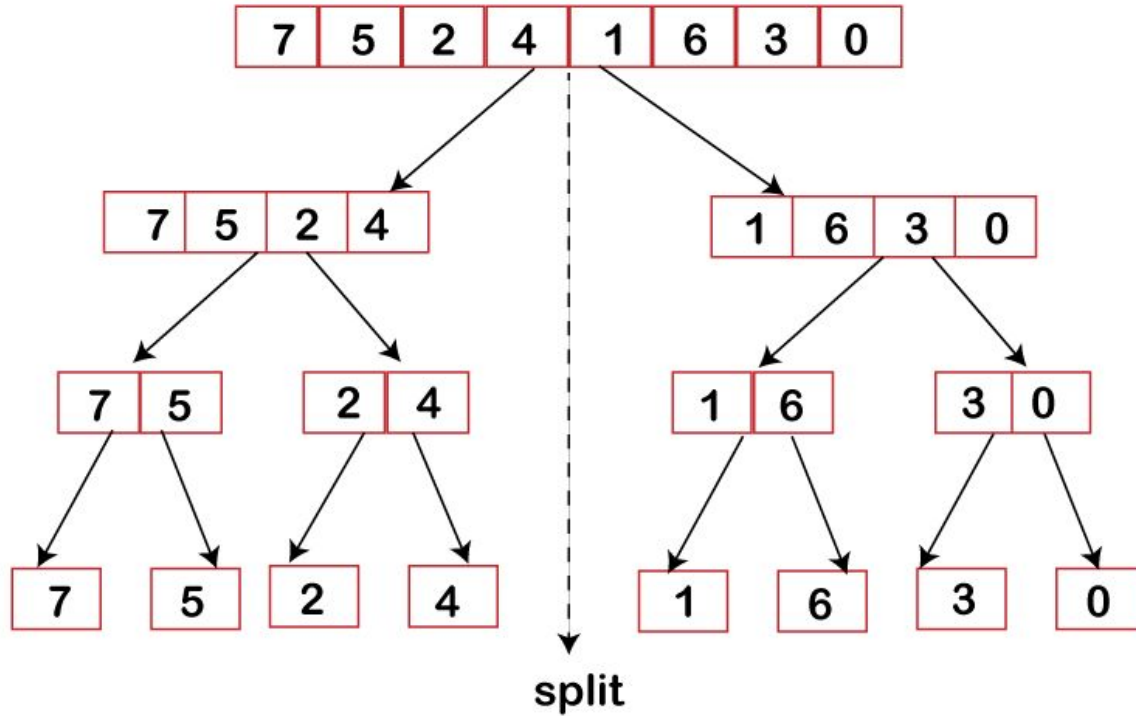


Figure 1: Merge Sort Divide Phase

# Merge Sort

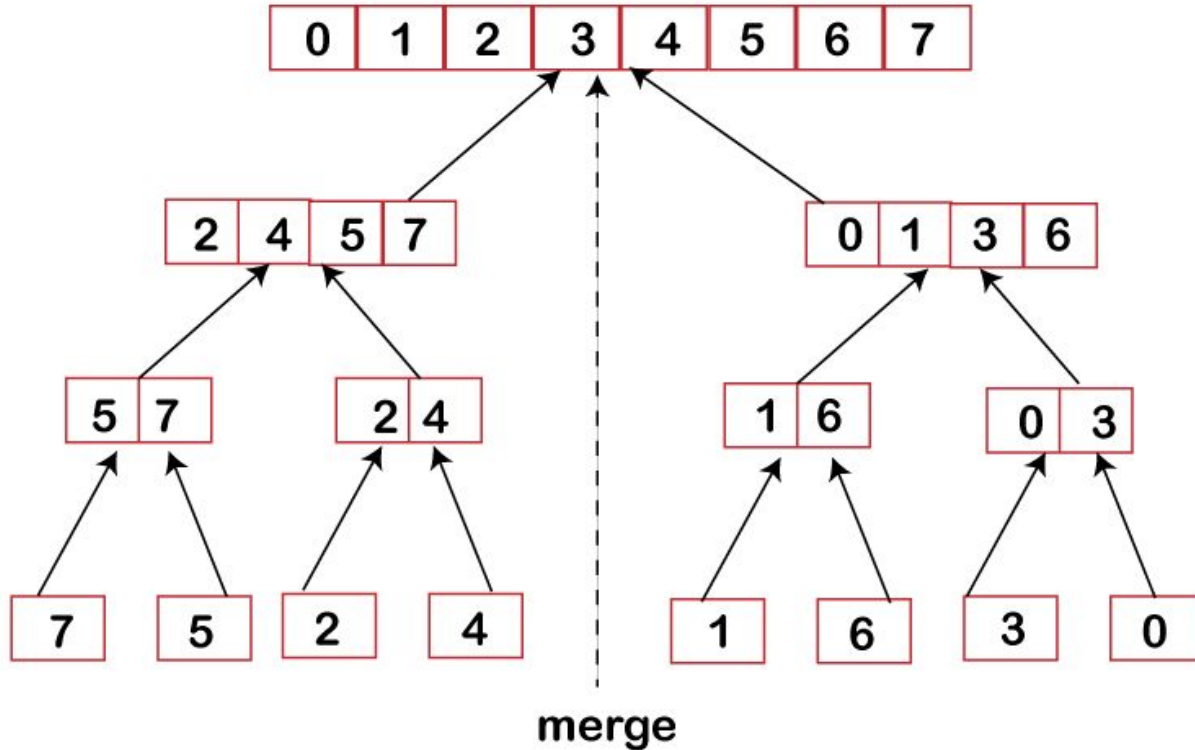


Figure 2: Merge Sort Combine Phase

# Merge Sort

**Algorithm** MergeSort(*low*, *high*)

// *a*[*low* : *high*] is a global array to be sorted.

// Small(*P*) is true if there is only one element

// to sort. In this case the list is already sorted.

```
{
    if (low < high) then // If there are more than one element
    {
        // Divide P into subproblems.
        // Find where to split the set.
        mid :=  $\lfloor (low + high) / 2 \rfloor$ ;
        // Solve the subproblems.
        MergeSort(low, mid);
        MergeSort(mid + 1, high);
        // Combine the solutions.
        Merge(low, mid, high);
    }
}
```

## Time Complexity

- Time complexity of Merge Sort is  $\theta(n \log n)$  in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

# Quick Sort

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



# Quicksort

- Partition(a, 1, 10)
  - The element  $a[1]=65$  is the partitioning element and it is eventually (in the sixth row) determined to be the fifth smallest element of the set. Notice that the remaining elements are unsorted but partitioned about  $a[5] = 65$ .

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	$i$	$p$
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

# Time Complexity

Case	Time Complexity
Best Case	$O(n \cdot \log n)$
Average Case	$O(n \cdot \log n)$
Worst Case	$O(n^2)$

# Strassen's matrix multiplication

- Naive Method of Matrix Multiplication:

```
Algorithm multiply(A, B, C){  
    for (int i = 0; i < N; i++) {  
        for (int j = 0; j < N; j++) {  
            C[i][j] = 0;  
            for (int k = 0; k < N; k++) {  
                C[i][j] += A[i][k]*B[k][j];  
            }  
        }  
    }  
}
```

Time Complexity of above method is  $O(N^3)$ .

# Strassen's matrix multiplication

- Divide and Conquer method of matrix multiplication

- Following is simple Divide and Conquer method to multiply two square matrices.

- Divide matrices A and B in 4 sub-matrices of size  $N/2 \times N/2$  as shown in the below diagram.
- Calculate following values recursively.  $ae + bg$ ,  $af + bh$ ,  $ce + dg$  and  $cf + dh$ .

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A                      B                      C

A, B and C are square matrices of size  $N \times N$   
a, b, c and d are submatrices of A, of size  $N/2 \times N/2$   
e, f, g and h are submatrices of B, of size  $N/2 \times N/2$

# Strassen's matrix multiplication

- Divide and Conquer method of matrix multiplication

In the above method, we do 8 multiplications for matrices of size  $n/2 \times n/2$  and 4 additions. Addition of two matrices takes  $O(n^2)$  time. So the time complexity can be written as

$$T(N) = 8T(n/2) + O(n^2)$$

From [Master's Theorem](#),  $a=8$ ,  $b=2$ ,  $f(n) = n^2$ ,  $\log_2 8 = 3$ ,  $n^k = n^2$  time complexity of above method is  **$O(n^3)$**  which is unfortunately same as the above naive method.

- Hence there is no improvement over the conventional method.

# Strassen's matrix multiplication

In strassen's method, number of multiplications is reduced from 8 to 7.

$$\begin{aligned}p1 &= a(f - h) & p2 &= (a + b)h \\p3 &= (c + d)e & p4 &= d(g - e) \\p5 &= (a + d)(e + h) & p6 &= (b - d)(g + h) \\p7 &= (a - c)(e + f)\end{aligned}$$

The A x B can be calculated using above seven multiplications.

Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A                      B                      C

A, B and C are square matrices of size  $N \times N$

a, b, c and d are submatrices of A, of size  $N/2 \times N/2$

e, f, g and h are submatrices of B, of size  $N/2 \times N/2$

p1, p2, p3, p4, p5, p6 and p7 are submatrices of size  $N/2 \times N/2$

# Strassen's matrix multiplication

- Time Complexity of Strassen's Method

- Addition and Subtraction of two matrices takes  $O(N^2)$  time. So time complexity can be written as

$$T(N) = 7T(N/2) + O(N^2)$$

From [Master's Theorem](#), time complexity of above method is  $O(N^{\log 7})$  which is approximately  $O(N^{2.8074})$

# THE GREEDY METHOD

- Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the immediate benefit.
- It is used to solve variety of problems
- Most of these problems have  $n$  inputs and require us to obtain a subset that satisfies some constraints.
- **Feasible Solution:** Any subset that satisfies these constraints are called feasible solutions
- **Objective Function:** A function is used to assign the value to the solution
- **Optimal Solution:** Any solution that maximize or minimize a given objective function



# Greedy Algorithm

```
Algorithm Greedy( $a, n$ )  
//  $a[1 : n]$  contains the  $n$  inputs.  
{  
     $solution := \emptyset$ ; // Initialize the solution.  
    for  $i := 1$  to  $n$  do  
    {  
         $x := \text{Select}(a)$ ;  
        if Feasible( $solution, x$ ) then  
             $solution := \text{Union}(solution, x)$ ;  
    }  
    return  $solution$ ;  
}
```

- The function **Select** selects an input from  $a[ ]$  and removes it
- The selected input value is assigned to  $x$ .
- **Feasible** is a Boolean-valued function that determines whether  $x$  can be included into the solution vector
- The function **Union** combines  $x$  with the solution and updates the objective function
-

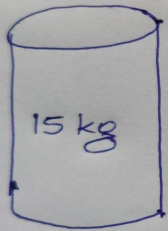
# KNAPSACK PROBLEM

- Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

# Fractional Knapsack Problem ( $m=15$ )

Objects	1	2	3	4	5	6	7
Profit	10	5	15	7	6	18	3
Weights	2	3	5	7	1	4	1
P/W	5	1.3	3	1	6	4.5	3
$x$	(1	$\frac{2}{3}$	1	0	1	1	1)

Let  $x$  be the variable determining whether the object is included. If included, it determines how much fraction it is included.



$$\begin{aligned}
 15 - 1 &= 14 \\
 14 - 2 &= 12 \\
 12 - 4 &= 8 \\
 8 - 5 &= 3 \\
 3 - 1 &= 2 \\
 2 - 2 &= 0
 \end{aligned}$$

<u>Constraint</u>
$\sum x_i w_i \leq m$
<u>Objective</u>
$\max \sum x_i p_i$

$$\begin{aligned}
 \text{Total Weight } \sum x_i w_i &= 1 \times 2 + \frac{2}{3} \times 3 + 1 \times 5 + 0 \times 7 + 1 \times 1 \\
 &\quad + 1 \times 4 + 1 \times 1 \\
 &= 15
 \end{aligned}$$

$$\begin{aligned}
 \text{Total Profit } \sum x_i p_i &= 1 \times 10 + \frac{2}{3} \times 5 + 1 \times 15 + 1 \times 6 + 1 \times 18 \\
 &\quad + 1 \times 3 \\
 &= 10 + 2 + 15 + 6 + 18 + 3 \\
 &= 55.2
 \end{aligned}$$

Disregarding the time to initially sort the objects the algorithm requires only  $O(n)$  time

# JOB SEQUENCING WITH DEADLINES

**The problem is stated as below.**

- There are  $n$  jobs to be processed on a machine.
- Each job  $i$  has a deadline  $d_i \geq 0$  and profit  $p_i \geq 0$ .
- $P_i$  is earned iff the job is completed by its deadline.
- The job is completed if it is processed on a machine for unit time.
- Only one machine is available for processing jobs.
- Only one job is processed at a time on the machine.

# Job Sequencing with Deadlines

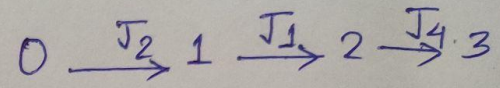
- High level description of job sequencing algorithm
  - **Assuming the jobs are ordered such that  $p[1] \geq p[2] \geq \dots \geq p[n]$**

```
GreedyJob(int d[], set J, int n)
// J is a set of jobs that can be
// completed by their deadlines.
{
    J = {1};
    for (int i=2; i<=n; i++) {
        if (all jobs in J  $\cup$  {i} can be completed
            by their deadlines) J = J  $\cup$  {i};
    }
}
```

---

# Job Sequencing with deadlines:

$n=5$



Jobs	$J_1$	$J_2$	$J_3$	$J_4$	$J_5$
Profit	20	15	10	5	1
Deadlines	2	2	1	3	3

Job Consider	Slot Assign	Solution	Profit
—	—	$\phi$	0
$J_1$	[1, 2]	$J_1$	20
$J_2$	[0, 1] [1, 2]	$J_1, J_2$	20+15
$J_3(x)$	[0, 1] [1, 2]	$J_1, J_2$	20+15
$J_4$	[0, 1] [1, 2] [2, 3]	$J_1, J_2, J_4$	20+15+5
$J_5$	[0, 1] [1, 2] [2, 3]	$J_1, J_2, J_4$	20+15+5 = 40

$\Sigma \text{Profit} = 40.$

∴ Job Sequence with Max Profit is  $J_2 \rightarrow J_1 \rightarrow J_4.$

## Greedy Algorithm for job sequencing with deadlines

1. Sort  $p_i$  into decreasing order. After sorting  $p_1 \geq p_2 \geq p_3 \geq \dots \geq p_i$ .
2. Add the next job  $i$  to the solution set if  $i$  can be completed by its deadline. Assign  $i$  to time slot  $(r-1, r)$ , where  $r$  is the largest integer such that  $1 \leq r \leq d_i$  and  $(r-1, r)$  is free.
3. Stop if all jobs are examined. Otherwise, go to step 2.

# Optimal Merge Pattern

- Optimal merge pattern is a pattern that relates to the merging of two or more sorted lists in a single sorted list.
- If we have two sorted lists, containing  $n$  and  $m$  records respectively then they could be merged together, to obtain one sorted file in time  **$O(n+m)$** .
- There are many ways in which pairwise merge can be done to get a single sorted list.
- Different pairings require a different amount of computing time.
- The main thing is to pairwise merge the  $n$  sorted files so that the number of comparisons will be less.



## Optimal Merge Pattern

- Suppose there are two sorted lists A and B, which are merged two get a single sorted merge list 'C'.

A	B	C
3	5	3
8	9	5
12	11	8
20	16	9

Size:  $(m)=4$   $(n)=4$

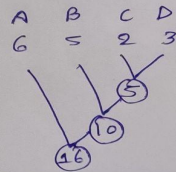
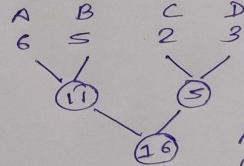
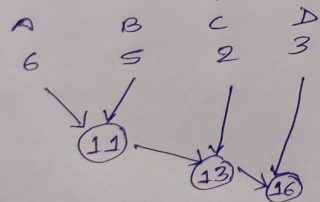
11  
12  
16  
20

Size:  $(m+n): 4+4=8$ .

- Similarly, let us suppose 4 lists A, B, C, and D with size:

List	A	B	C	D
Size	6	5	2	3

- There are different ways to merge them:



∴ Total amount of merging =  $16+13+11$   
= 40

$16+11+5$   
= 32

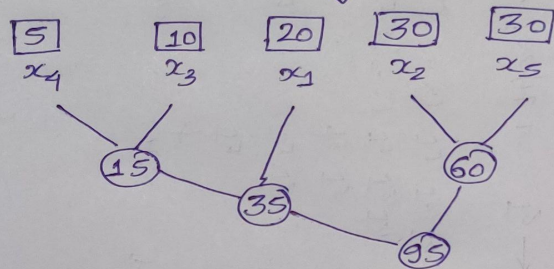
⇒  $16+10+5 = 31$

→ We see there are no other methods (patterns), to get optimal pattern, always merge two smallest size lists.

Example 2:

lists:	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$
Sizes:	20	30	10	5	30

First take list in increasing order of size.



Total Cost of =  $15 + 35 + 95 + 60 = 205$

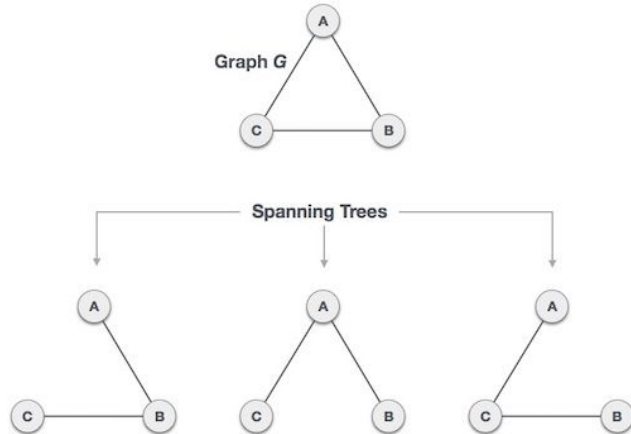
Merging using  
Optimal Pattern

~~If  $d$  is the number of times a n.~~  
If  $d_i$  is the distance from the root to the external node for list  $x_i$  and  $q_i$ : Then

$$3 \times 5 + 10 \times 3 + 2 \times 20 + 30 \times 2 + 30 \times 2 = 205.$$

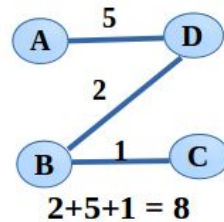
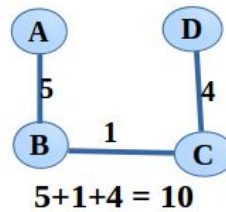
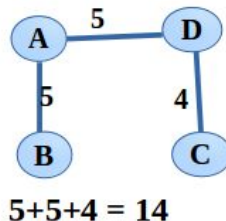
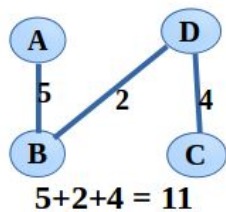
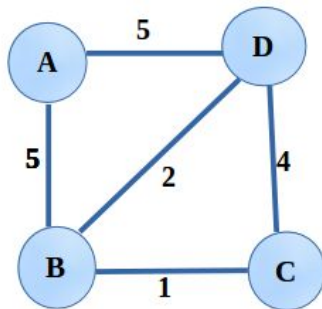
# Minimum Spanning Tree

- A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges.
- If a vertex is missed, then it is not a spanning tree.
- A single graph can have many different spanning trees.



# Minimum Spanning Tree

- A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree.
- The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

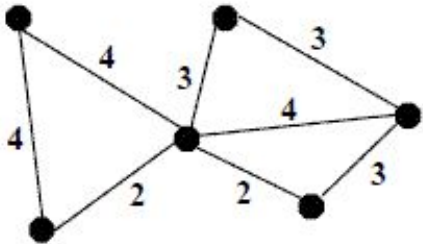
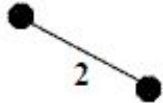
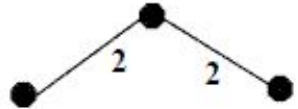
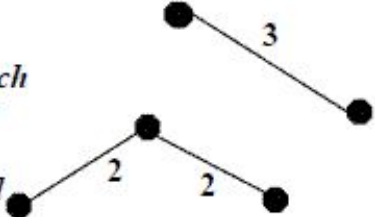
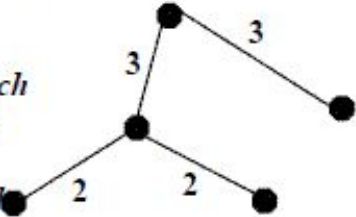
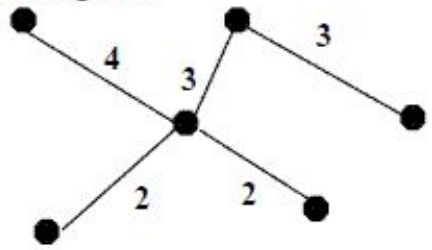


# Kruskal's Minimum Spanning Tree Algorithm

steps for finding MST using Kruskal's algorithm:

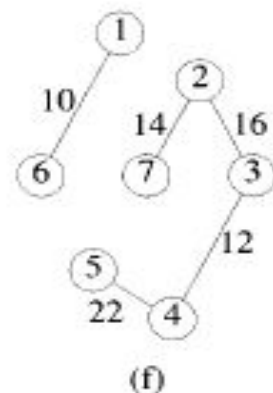
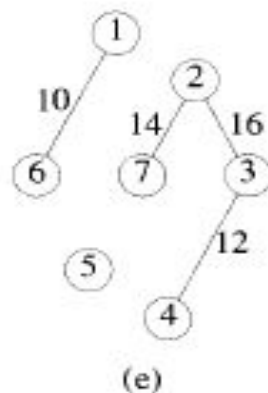
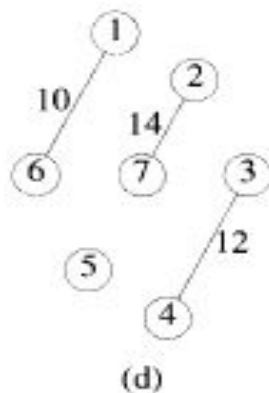
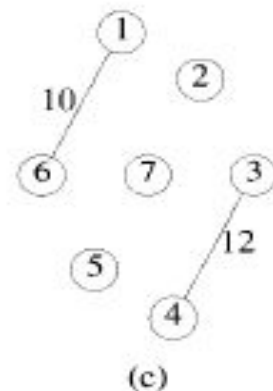
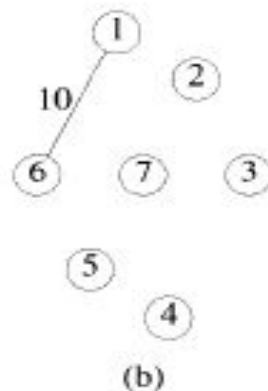
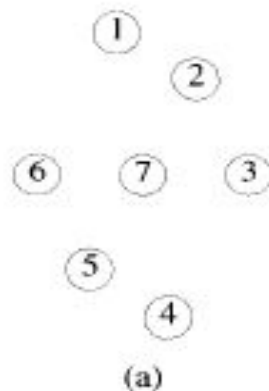
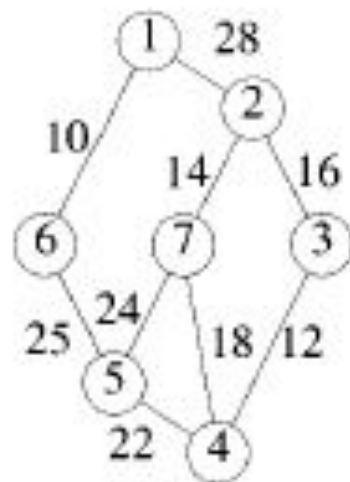
1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are  $(V-1)$  edges in the spanning tree.

# Kruskal's Algorithm

<p>1 Given a network.....</p> 	<p>2 Choose the shortest edge (if there is more than one, choose any of the shortest).....</p> 	<p>3 Choose the next shortest edge and add it.....</p> 
<p>4 Choose the next shortest edge which wouldn't create a cycle and add it.</p> 	<p>5 Choose the next shortest edge which wouldn't create a cycle and add it.</p> 	<p>6 Repeat until you have a minimal spanning tree.</p> 

The resulting tree has cost 14

# Kruskal's Algorithm



```

1  Algorithm Kruskal( $E, cost, n, t$ )
2  //  $E$  is the set of edges in  $G$ .  $G$  has  $n$  vertices.  $cost[u, v]$  is the
3  // cost of edge  $(u, v)$ .  $t$  is the set of edges in the minimum-cost
4  // spanning tree. The final cost is returned.
5  {
6      Construct a heap out of the edge costs using Heapify;
7      for  $i := 1$  to  $n$  do  $parent[i] := -1$ ;
8      // Each vertex is in a different set.
9       $i := 0$ ;  $mincost := 0.0$ ;
10     while  $((i < n - 1)$  and  $(\text{heap not empty}))$  do
11     {
12         Delete a minimum cost edge  $(u, v)$  from the heap
13         and reheapify using Adjust;
14          $j := \text{Find}(u)$ ;  $k := \text{Find}(v)$ ;
15         if  $(j \neq k)$  then
16         {
17              $i := i + 1$ ;
18              $t[i, 1] := u$ ;  $t[i, 2] := v$ ;
19              $mincost := mincost + cost[u, v]$ ;
20             Union $(j, k)$ ;
21         }
22     }
23     if  $(i \neq n - 1)$  then write ("No spanning tree");
24     else return  $mincost$ ;
25 }
```



# How Prim's algorithm works

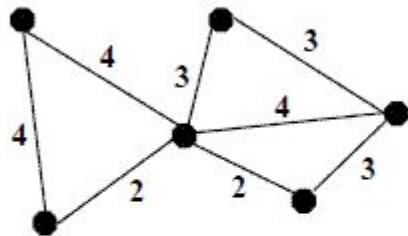
The steps for implementing Prim's algorithm are as follows:

- Initialize the minimum spanning tree with a vertex chosen at random.
- Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
- Keep repeating step 2 until we get a minimum spanning tree

# Prim's Algorithm

1

*Given a network.....*



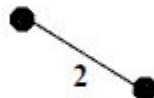
2

*Choose a vertex*



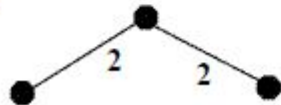
3

*Choose the shortest edge from this vertex.*



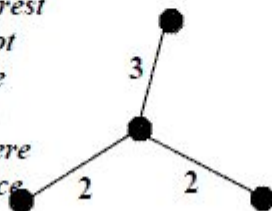
4

*Choose the nearest vertex not yet in the solution.*



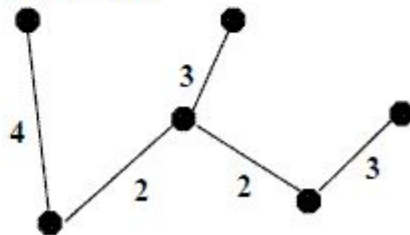
5

*Choose the next nearest vertex not yet in the solution, when there is a choice choose either.*



6

*Repeat until you have a minimal spanning tree.*



## **Minimum Connector Algorithms**

### **Kruskal's algorithm**

1. Select the shortest edge in a network
2. Select the next shortest edge which does not create a cycle
3. Repeat step 2 until all vertices have been connected

### **Prim's algorithm**

1. Select any vertex
2. Select the shortest edge connected to that vertex
3. Select the shortest edge connected to any vertex already connected
4. Repeat step 3 until all vertices have been connected

# Prim's Vs Kruskal's Algorithm

Prim's algorithm gives connected component as well as it works only on connected graph.

Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components

# Tree vertex splitting

- Directed and weighted binary tree
- Consider a network of power line transmission
- The transmission of power from one node to the other results in some loss, such as drop in voltage
- Each edge is labeled with the loss that occurs (edge weight)
- Network may not be able to tolerate losses beyond a certain level
- You can place boosters in the nodes to account for the losses



# Tree Vertex Splitting

- **Definition** Given a network and a loss tolerance level, the tree vertex splitting problem is to determine the optimal placement of boosters.
- You can place boosters only in the vertices and nowhere else

## Greedy solution for TVSP(steps)

- ➡ We want to minimize the number of booster stations ( $X$ )
- ➡ For each node  $u \in V$ , compute the maximum delay  $d(u)$  from  $u$  to any other node in its subtree
- ➡ If  $u$  has a parent  $v$  such that  $d(u) + w(v, u) > \delta$ , split  $u$  and set  $d(u)$  to zero
- ➡ Computation proceeds from leaves to root



- Delay for each leaf node is zero
- The delay for each node  $v$  is computed using the formula

$$d(v) = \max_{u \in C(v)} \left[ d(u) + w(\text{parent}(v), v) \right]$$

$u$  is child of  $v$

If  $d(v) > \delta$ , split  $v$

Here, u is child of j

```
for i=1 to n
{
  d[i]=0;
}
for j=n to 2
{
  d[j] = max  $\left[ \begin{array}{l} u \in \text{Child}(j) \\ d(u) + w(\text{parent}(j), j) \end{array} \right]$ 
  if(d[j]> δ)
    return j;
}
```



$$\delta = 6$$

for  $i = 1$  to  $10$

$$i = 1 \quad d[1] = 0$$

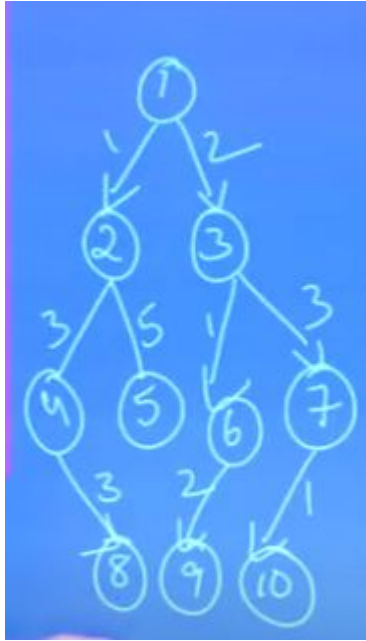
$$i = 2 \quad d[2] = 0$$

⋮

⋮

⋮

$$i = 10 \quad d[10] = 0$$



for  $j = n$  to 2  
 $= 10$  to 2

when  $j = 10$

$$d[9] = d[10] = \max\{0 + w(7, 10)\}$$

$$= \max\{0 + 1\}$$

$$= 1$$

$$d[8] = \max\{0 + w(6, 9)\} = 2$$

$$d[7] = \max\{0 + w(4, 8)\} = 3$$

$$d[6] = \max\{0 + w(5, 9)\} = 3$$

$$d[5] = \max\{d(10) + w(3, 7)\}$$

$$= d(4)$$

child of 7

$$= \max\{1 + 3\} = 4$$

$$d[4] = \max\{d(u) + w(3, 6)\}$$

$$= \max\{d(9) + 1\}$$

$$= \max\{2 + 1\} = 3$$

$$d[3] = \max\{d(u) + w(2, 5)\}$$

$$= \max\{0 + 5\} = 5$$

$$d[2] = \max\{d(u) + w(2, 4)\}$$

$$= \max\{d(8) + w(2, 4)\}$$

$$= \max\{3 + 3\} = 6$$

$$d[3] = \max\{d(6) + w(1, 3)$$

$$d(7) + w(1, 3)\}$$

$$= \max\{3 + 2$$

$$4 + 2\} = 6$$

$$d[2] = \max\{d(4) + w(2, 4)$$

$$d(5) + w(2, 5)\}$$

$$= \max\{6 + 3$$

$$5 + 3\} = 9 (> 8) \text{ i.e. } 9 > 6$$

So, at node 2 boost ~~is~~ is needed.

At last part,  $d(2)$  calculation needs to be corrected, Its correct value is 7

## Optimal storage on Tapes

- There are  $n$  programs that are to be stored on a computer tape of length  $l$ .
- Program  $i$  has length  $l_i$ ,  $1 \leq i \leq n$
- Let Programs are stored in the order  $I = i_1, i_2, \dots, i_n$ , Time  $t_j$  needed to retrieve program  $i_j$  is proportional to

$$\sum_{k=1}^j l_{i_k}$$



## Optimal storage on tapes

If all the programs retrieved often the **Expected or Mean Retrieval Time (MRT)** is

$$MRT = \frac{1}{n} \sum_{j=1}^n t_j$$

**Objective is to find permutation for n programs so that Mean Retrieval Time (MRT) is minimized.**

Minimizing MRT is equivalent to minimizing

$$d(I) = \sum_{j=1}^n \sum_{k=1}^j l_{ik}$$

## Optimal storage on Tapes

Let  $n=3$   $(l_1, l_2, l_3) = (5, 10, 3)$



For these 3 programs, how many orderings are possible?



## Possible solutions ( $n!$ )

Ordering No.	Program order		
1	1	2	3
2	1	3	2
3	2	1	3
4	2	3	1
5	3	1	2
6	3	2	1



## Possible Solutions

No.	Program order			d(I)	MRT
1	1	2	3	$5 + (5+10) + (5+10+3) = 38$	$38/3 = 12.66$
2	1	3	2	$5 + (5+3) + (5+3+10) = 31$	$31/3 = 10.33$
3	2	1	3	$10 + (10+5) + (10+5+3) = 43$	$43/3 = 14.33$
4	2	3	1	$10 + (10+3) + (10+3+5) = 41$	$41/3 = 13.33$
5	3	1	2	$3 + (3+5) + (3+5+10) = 29$	$29/3 = 9.66$
6	3	2	1	$3 + (3+10) + (3+10+5) = 34$	$34/3 = 11.33$

## Approach to obtain optimal solution

1. Find all the permutations of programs and find optimal Solution (minimum MRT).

Drawback : for large value of  $n$ , time will be high to find all possible permutations.

## Greedy Approach to obtain optimal solution

2. Store programs in **increasing order of their lengths** using any sorting order. (Time required for sorting:  $n \log n$ )



i.e. in given example, increasing order of lengths are : 3,5,10 and hence ordering of programs is 3,1,2.