

SEMESTER EXAM SERIES

FREE NOTES

9 HOURS



ALGORITHM

Syllabus

- **Chapter-1 (Introduction):** Algorithms, Analysing Algorithms, Efficiency of an Algorithm, Time and Space Complexity, Asymptotic notations: Big-Oh, Time-Space trade-off Complexity of Algorithms, Growth of Functions, Performance Measurements.
- **Chapter-2 (Sorting and Order Statistics):** Concept of Searching, Sequential search, Index Sequential Search, Binary Search Shell Sort, Quick Sort, Merge Sort, Heap Sort, Comparison of Sorting Algorithms, Sorting in Linear Time. Sequential search, Binary Search, Comparison and Analysis Internal Sorting: Insertion Sort, Selection, Bubble Sort, Quick Sort, Two Way Merge Sort, Heap Sort, Radix Sort, Practical consideration for Internal Sorting.
- **Chapter-3 (Divide and Conquer):** with Examples Such as Sorting, Matrix Multiplication, Convex Hull and Searching.
- **Chapter-4 (Greedy Methods):** with Examples Such as Optimal Reliability Allocation, Knapsack, Huffman algorithm
- **Chapter-5 (Minimum Spanning Trees) –** Prim's and Kruskal's Algorithms,
- **Chapter-6 (Single Source Shortest Paths):** - Dijkstra's and Bellman Ford Algorithms.
- **Chapter-7 (Dynamic Programming)** with Examples Such as Knapsack. All Pair Shortest Paths – Warshal's and Floyd's Algorithms, Resource Allocation Problem. Backtracking, Branch and Bound with Examples Such as Travelling Salesman Problem, Graph Coloring, n-Queen Problem, Hamiltonian Cycles and Sum of Subsets.
- **Chapter-8 (Advanced Data Structures):** Red-Black Trees, B – Trees, Binomial Heaps, Fibonacci Heaps, Tries, Skip List, Introduction to Activity Networks Connected Component.
- **Chapter-9 (Selected Topics):** Algebraic Computation, Fast Fourier Transform, String Matching, Theory of NPCompleteness, Approximation Algorithms and Randomized Algorithms.

Chapter-1 (Introduction)

Algorithms, Analysing Algorithms,
Efficiency of an Algorithm, Time and Space
Complexity, Asymptotic notations: Big-Oh,
Time-Space trade-off Complexity of
Algorithms, Growth of Functions,
Performance Measurements.

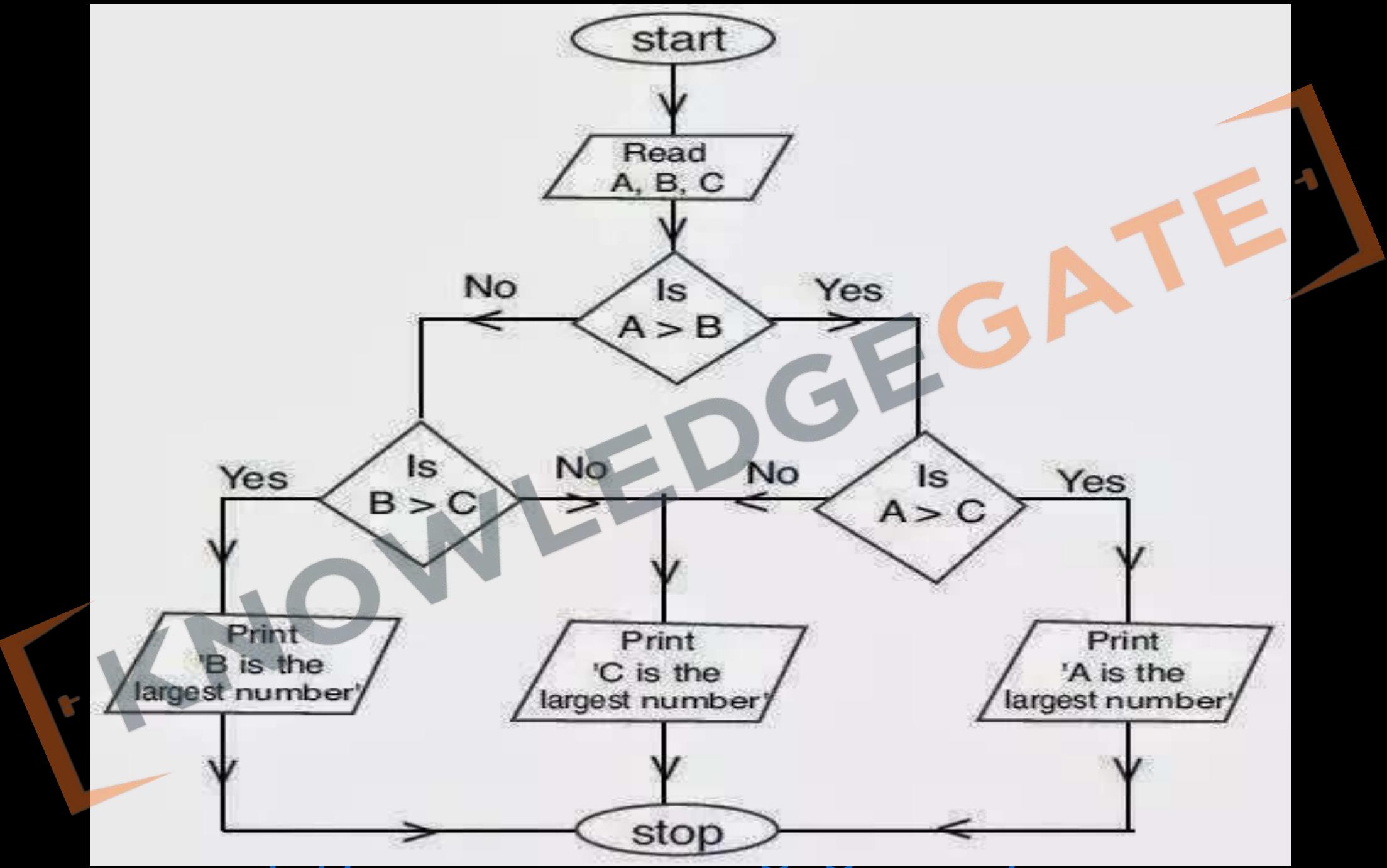
Q Find the Largest Number Among Three Numbers ?



<http://>

'GATE

1. Start
2. Read the three numbers to be compared, as A, B and C.
3. Check if A is greater than B.
 - 3.1 If true, then check if A is greater than C.
 - 3.1.1 If true, print 'A' as the greatest number.
 - 3.1.2 If false, print 'C' as the greatest number.
 - 3.2 If false, then check if B is greater than C.
 - 3.2.1 If true, print 'B' as the greatest number.
 - 3.2.2 If false, print 'C' as the greatest number.
4. End



```
#include <stdio.h>
int main()
{
    int A, B, C;

    printf("Enter the numbers A, B and C: ");
    scanf("%d %d %d", &A, &B, &C);

    if (A >= B && A >= C)
        printf("%d is the largest number.", A);

    if (B >= A && B >= C)
        printf("%d is the largest number.", B);

    if (C >= A && C >= B)
        printf("%d is the largest number.", C);

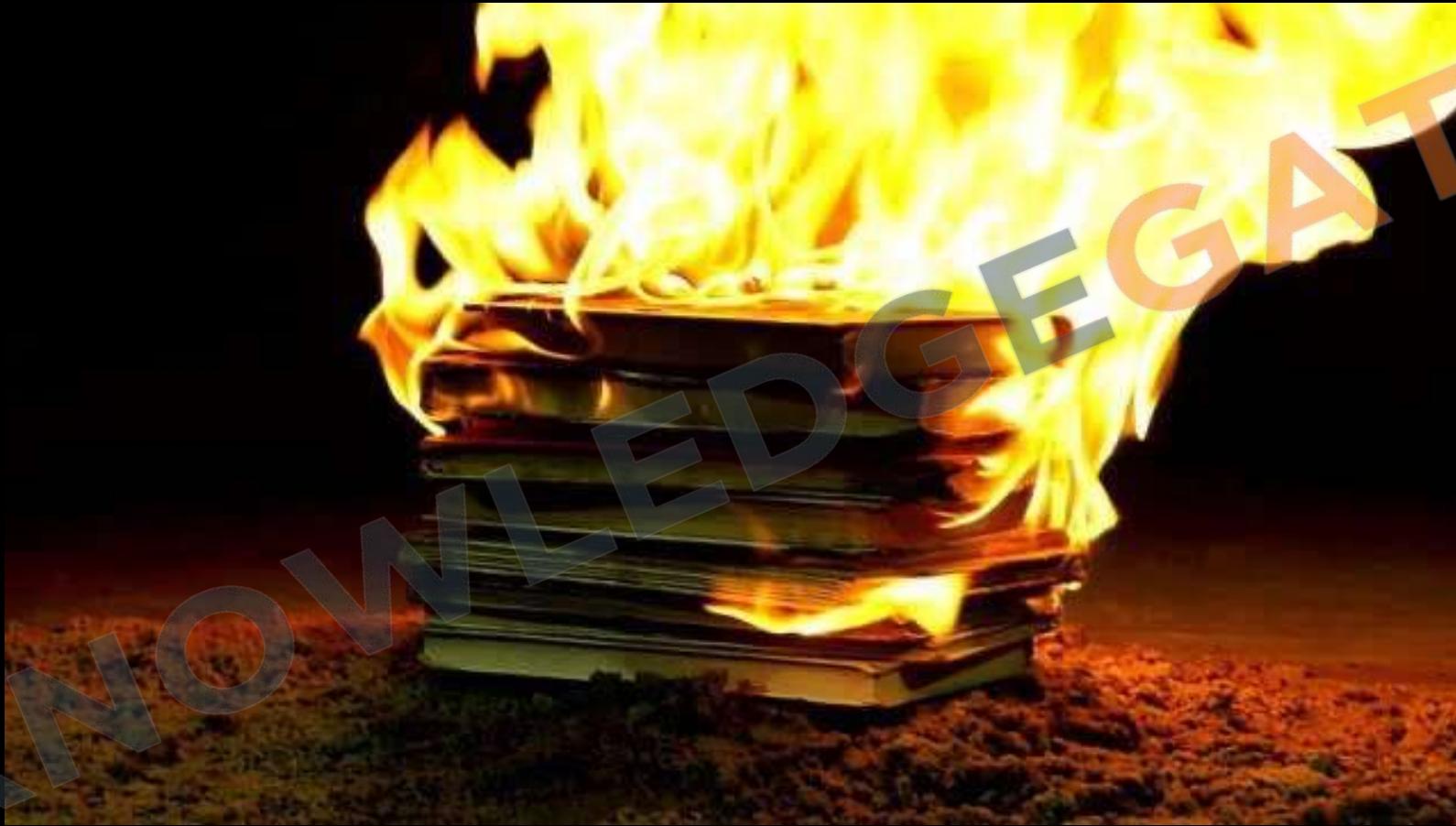
    return 0;
}
```

Introduction to Algorithm

- In mathematics and computer science, an algorithm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation. A step by step Procedure.



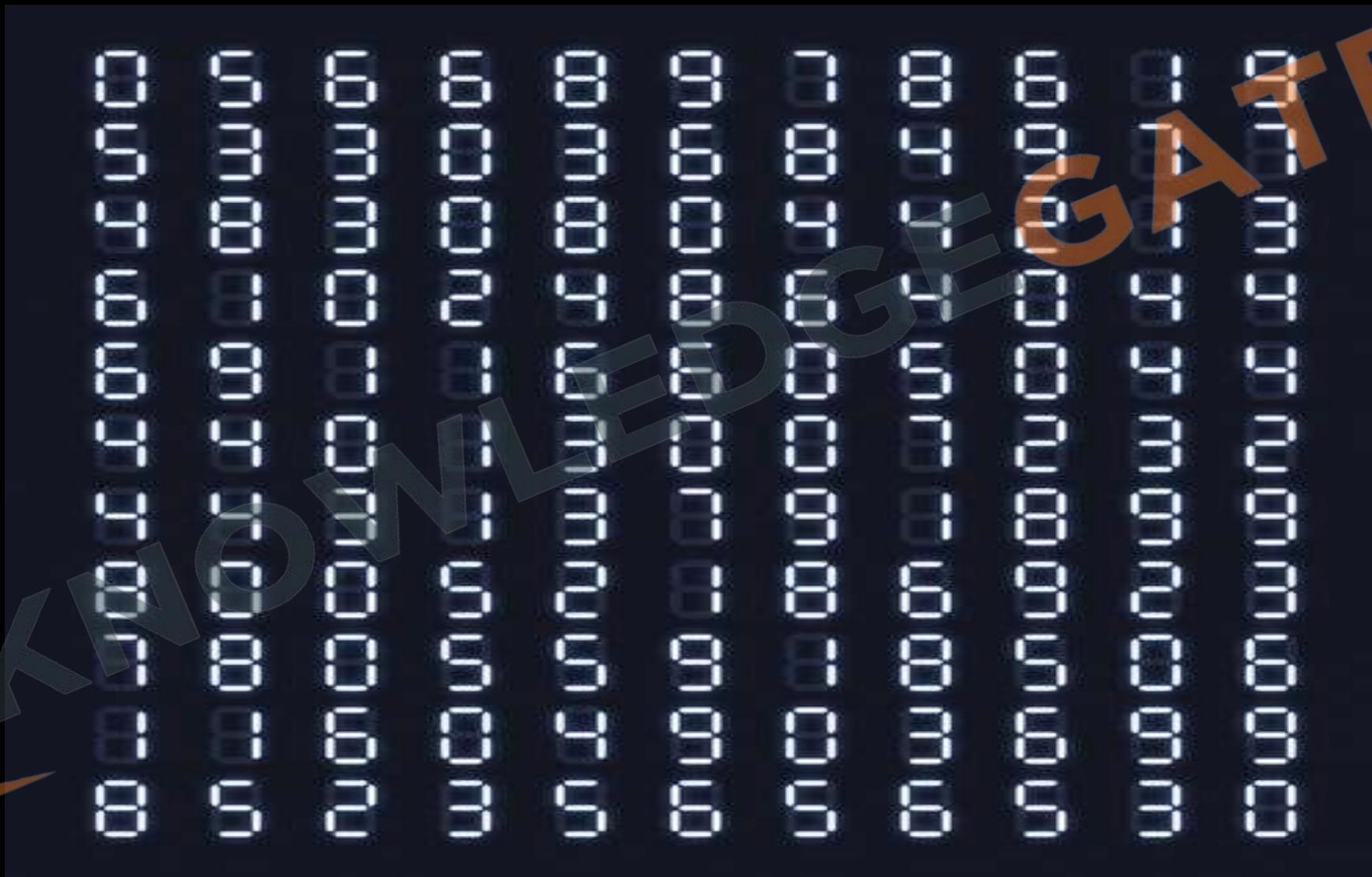
- Algorithms are unambiguous specifications for performing calculation, data processing, automated reasoning, and other tasks.



बेटा किताबों को आग लगा दो

<http://www.knowledgegate.in/GATE>

- Will accept Zero or more input, but generate at least one output.



Properties of Algorithm

- Should terminate in finite time
- Unambiguous
- Input Zero or more output at least one
- Every instruction in algo should be effective
- It should be deterministic

Algorithm

Written in design Phase

Needs Domain Knowledge

Can be Written in Any Language

Independent of H/w & S/w

We analysis Algorithm for time &
space
<http://www.knowledgegate.in/GATE>

Program

Written in Implementation Phase

Needs Programmer Knowledge

Can be Written in Programming
Language

Dependent of H/w & S/w

We test programs for faults
<http://www.knowledgegate.in/GATE>

Problem Solving Cycle

- Problem Definition: Understand Problem
- Constraints & Conditions: Understand constraints if any
- **Design Strategies (Algorithmic Strategy)**
- Express & Develop the algo
- Validation (Dry run)
- **Analysis (Space and Time analysis)**
- Coding
- Testing & Debugging
- Installation
- Maintenance

Need for Analysis

- We do analysis of algorithm to do a performance comparison between different algorithm to figure out which one is best possible option.



- What parameters can be considered for comparison between cars?



For Algo?

- Following are the parameters which can be considered while analysis of an algorithm
 - Time
 - Space
 - Bandwidth
 - Register
 - Battery power

- Out of all **time** is the most important Criteria for analysis of algorithm



- How to analysis time?



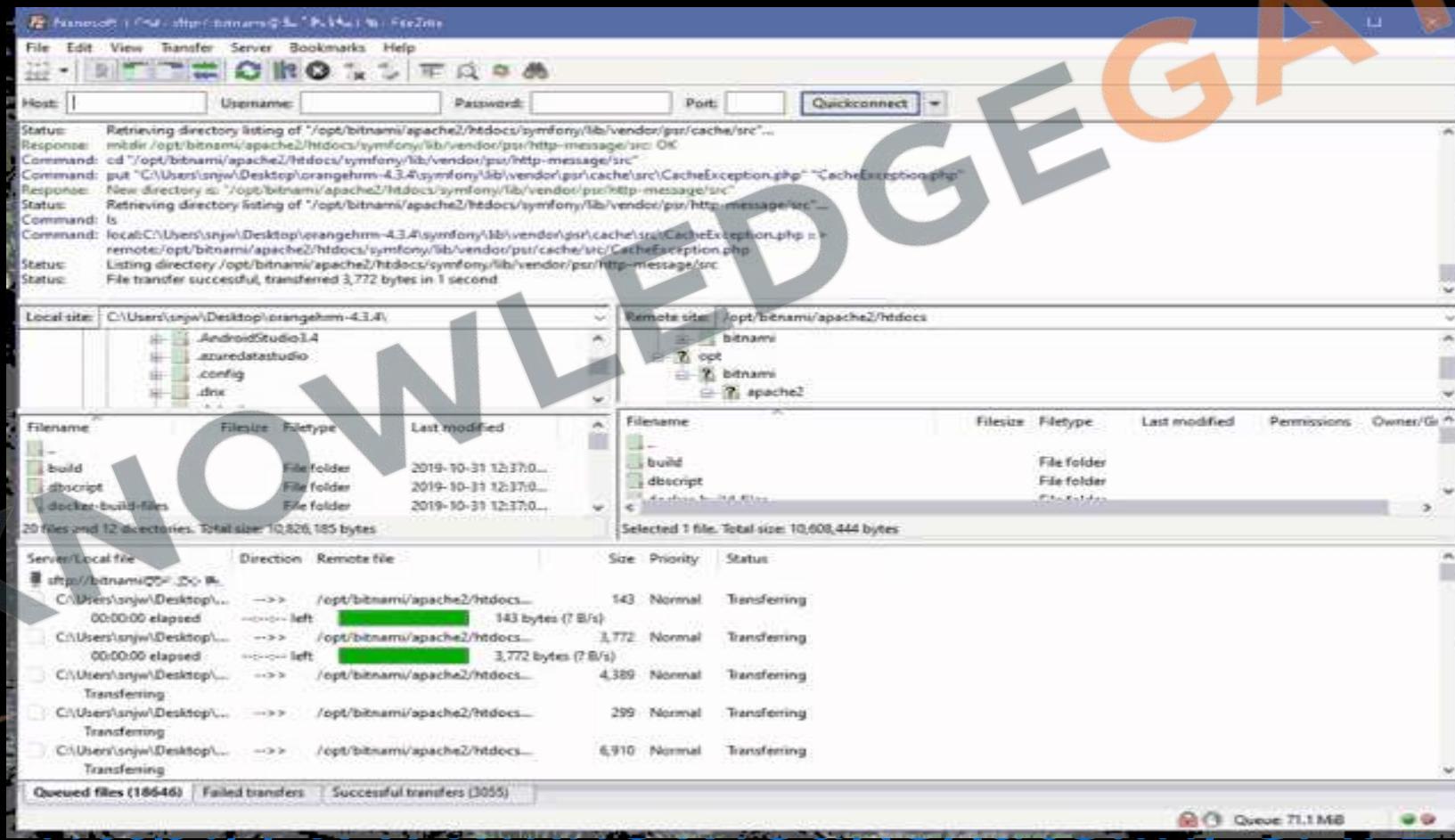
<http://www.knowledgegate.in/GATE>

Types of Analysis

- Experimental or
- Apostrium or
- Relative analysis
- Apriori Analysis or
- Independent analysis or
- Absolute analysis



- **Experimental or Apostrium or relative analysis** : Means analysis of algorithm after it is converted to code. Implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time.



- **Advantage**: Exact values no rough
- **Disadvantage**: final result instead of depending only algorithm depends on many other factors like background software & hardware, programming language, even the temperature of the room.

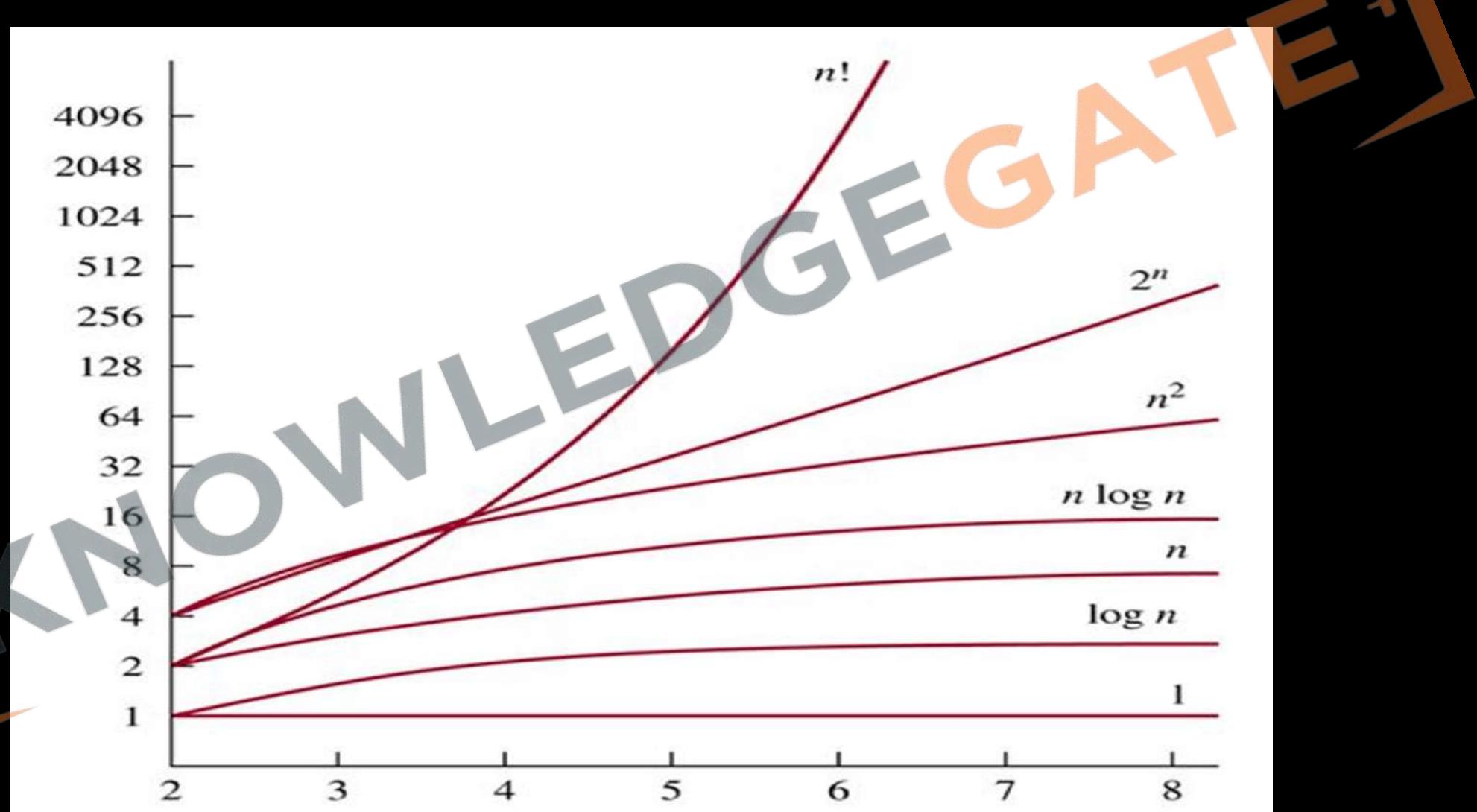
- **Apriori Analysis or Independent analysis or Absolute analysis:** we do analysis using asymptotic notations and mathematical tools of only algorithm, i.e. before converting it into program of a particular programming language.
- It is a determination of order of magnitude of a statement.



<http://>

in/GATE

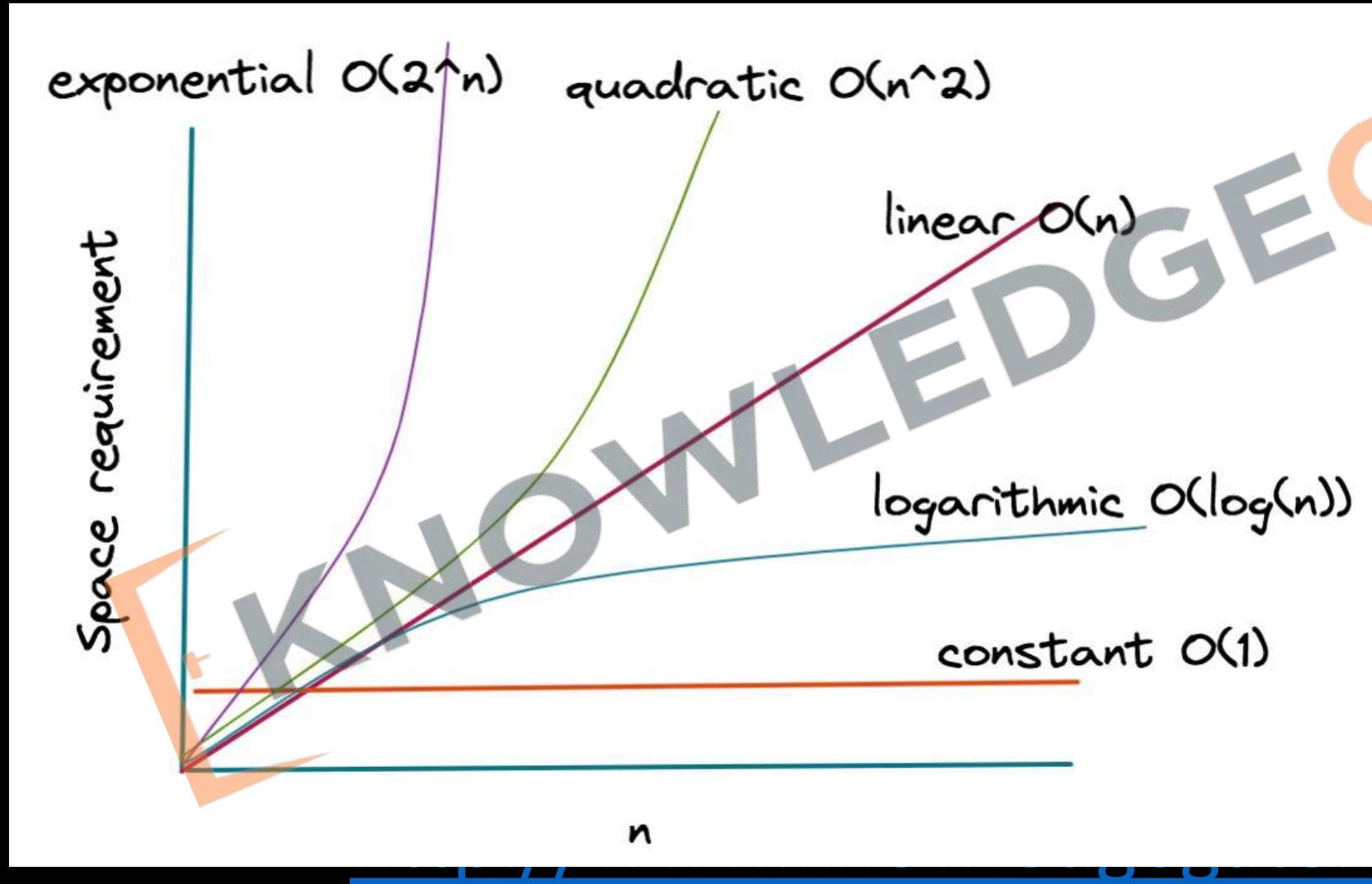
- In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.



- Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms.
- It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower, always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.
 - **Advantage:** Uniform result depends only on algorithm.
 - **Disadvantage:** Estimated or approximate value no accurate and precise value.

Asymptotic Notations

- Asymptotic notations are Abstract notation for describing the behavior of algorithm and determine the rate of growth of a function.



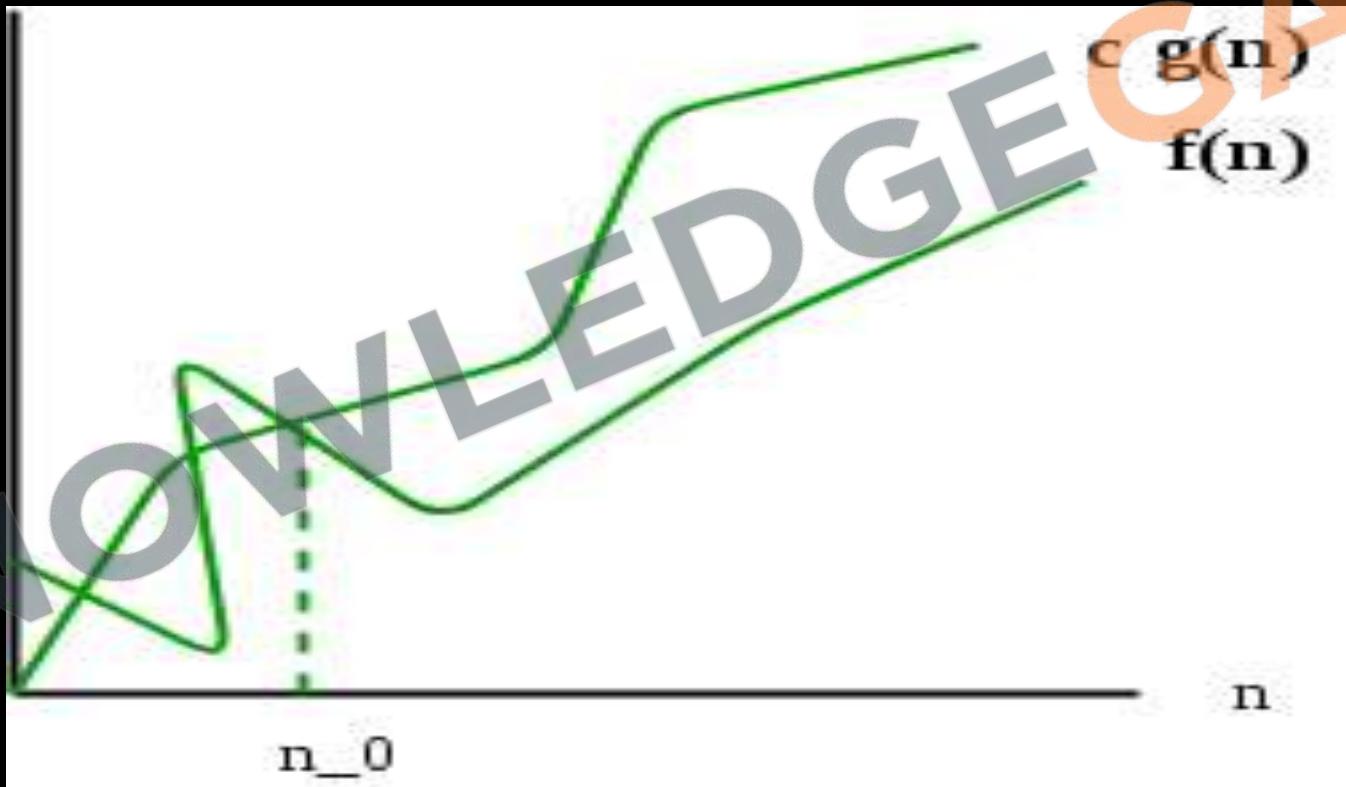


KNOWLEDGE GATE

<http://www.knowledgegate.in/GATE>

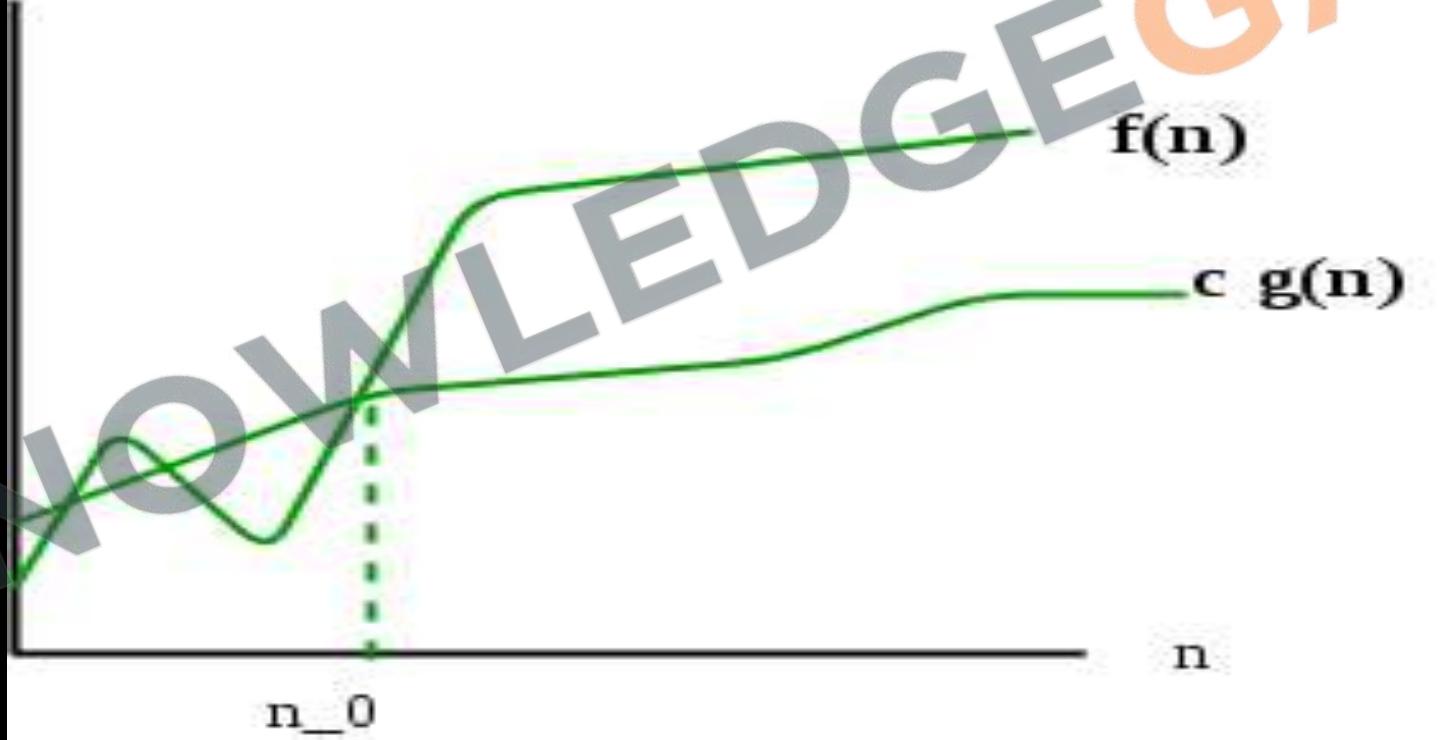
Big O Notation

- The Big O notation defines an upper bound of an algorithm, it bounds a function only from above.
- The Big O notation is useful when we only have upper bound on time complexity of an algorithm.
- Many times, we easily find an upper bound by simply looking at the algorithm.
- $O(g(n)) = \{f(n): \text{there exist positive constants } C \text{ and } N_0 \text{ such that } 0 \leq f(n) \leq C \cdot g(n) \text{ for all } n \geq N_0\}$



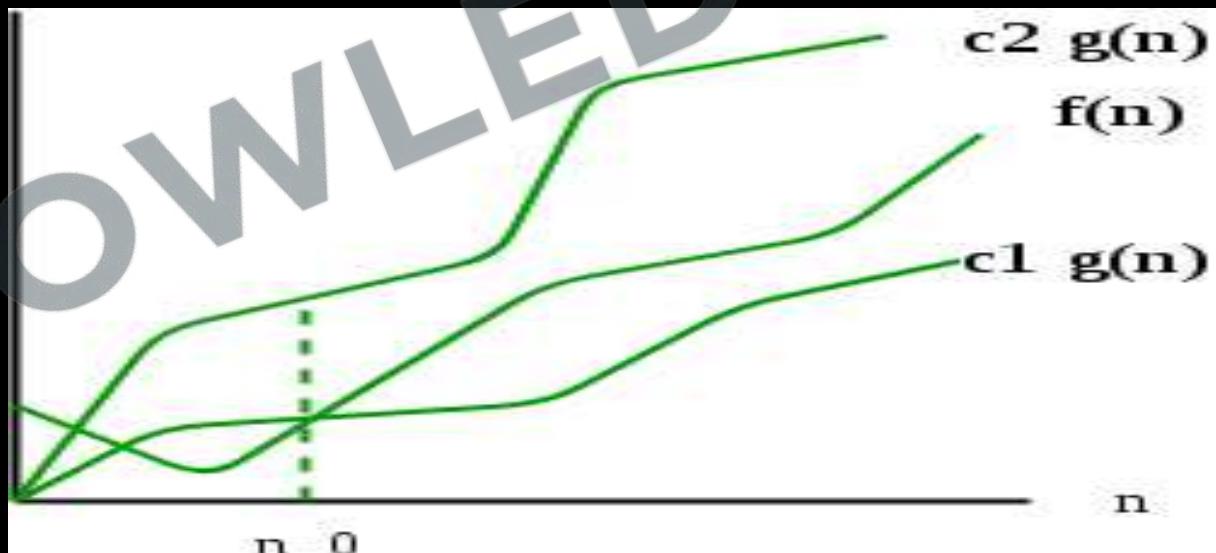
Ω Notation

- Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.
- Ω Notation can be useful when we have lower bound on time complexity of an algorithm.
- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions.
- $\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq c*g(n) \leq f(n) \text{ for all } n \geq n_0\}$.



Theta Notation

- **Θ Notation:** The theta notation bounds a function from above and below, so it defines exact asymptotic behaviour.
- For a given function $g(n)$, we denote $\Theta(g(n))$ is following set of functions.
- $\Theta(g(n)) = \{f(n): \text{there exist positive constants } C_1, C_2 \text{ and } n_0 \text{ such that } 0 \leq C_1 * g(n) \leq f(n) \leq C_2 * g(n) \text{ for all } n \geq n_0\}$
- The above definition means, if $f(n)$ is theta of $g(n)$, then the value $f(n)$ is always between $C_1 * g(n)$ and $C_2 * g(n)$ for large values of n ($n \geq n_0$).
- The definition of theta also requires that $f(n)$ must be non-negative for values of n greater than n_0 .



<http://www.educatech.in/GATE>

Small notations

- Every thing is same as big notations just, just we take strictly increasing or monotonically increasing case and equal case is not allowed.
 - Analogy of asymptotic notation with real numbers
- | | |
|--------------------------|------------|
| $f(n)$ is $O(g(n))$ | $a \leq b$ |
| $f(n)$ is $\Omega(g(n))$ | $a \geq b$ |
| $f(n)$ is $\Theta(g(n))$ | $a = b$ |
| $f(n)$ is $o(g(n))$ | $a < b$ |
| $f(n)$ is $\omega(g(n))$ | $a > b$ |

Conclusion

- Most useful notation is Theta, followed by Big O
- the Omega notation is the least used notation among all three



h

E

$$f(n) = 100 * 2^n + n^5 + n \quad \text{then show } f(n) = O(2^n)$$

$$f(n) = 100 * 2^n + n^5$$

$$f(n) = 100 * 2^n$$

$$f(n) = 2^n$$

$$f(n) = O(2^n)$$

$$n^5 \gg n$$

$$2^n \gg n^5$$
$$2^n \approx 100 \times n$$

Master Theorem

- In the analysis of algorithms, the master theorem for divide-and-conquer recurrences provides an asymptotic analysis (using Big O notation) for recurrence relations of types that occur in the analysis of many divide and conquer algorithms.

- The approach was first presented by Jon Bentley, Dorothea Haken, and James B. Saxe in 1980, where it was described as a "unifying method" for solving such recurrences. The name "master theorem" was popularized by the widely used algorithms textbook Introduction to Algorithms by Cormen, Leiserson, Rivest, and Stein.



<http://www.tu-mit-india.org> TU MASTER AADMI HAI YAAR !

- The above equation divides the problem into ‘a’ number of subproblems recursively, $a \geq 1$
- Each subproblem being of size n/b . the subproblems (of size less than k) that do not recurse. ($b > 1$)
- where $f(n)$ is the time to create the subproblems and combine their results in the above procedure.

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

Case 1

- If $f(n) = O(n^{\log_b a} - \epsilon)$ for some constant $\epsilon > 0$,
- then $T(n) = \Theta(n^{\log_b a})$

- Case1
 - If $f(n) = O(n^{\log_b a} - \epsilon)$ for some constant $\epsilon > 0$,
 - then $T(n) = \Theta(n^{\log_b a})$
- $Q T(n) = 4T(n/2) + n?$

- Case1
 - If $f(n) = O(n^{\log_b a} - \epsilon)$ for some constant $\epsilon > 0$,
 - then $T(n) = \Theta(n^{\log_b a})$
- Q $T(n) = 9T(n/3) + n?$

- Case1
 - If $f(n) = O(n^{\log_b a} - \epsilon)$ for some constant $\epsilon > 0$,
 - then $T(n) = \Theta(n^{\log_b a})$
- Q $T(n) = 9T(n/3) + n^2?$

- Case1
 - If $f(n) = O(n^{\log_b a} - \epsilon)$ for some constant $\epsilon > 0$,
 - then $T(n) = \Theta(n^{\log_b a})$
- Q $T(n) = 8T(n/2) + n^2?$

- Q T(n) = 7T(n/2) + n²?



<http://www.knowledgegate.in/GATE>

Case 2

- If $f(n) = \Theta(n^{\log_b a})$,
 - then $T(n) = \Theta(n^{\log_b a} \lg n)$

- Case1
 - If $f(n) = O(n^{\log_b a} - \epsilon)$ for some constant $\epsilon > 0$,
 - then $T(n) = \Theta(n^{\log_b a})$
- Case2
 - If $f(n) = \Theta(n^{\log_b a})$,
 - then $T(n) = \Theta(n^{\log_b a} \lg n)$
- Q $T(n) = 2T(n/2) + n$?

- Case1
 - If $f(n) = O(n^{\log_b a} - \epsilon)$ for some constant $\epsilon > 0$,
 - then $T(n) = \Theta(n^{\log_b a})$
- Case2
 - If $f(n) = \Theta(n^{\log_b a})$,
 - then $T(n) = \Theta(n^{\log_b a} \lg n)$
- Q $T(n) = T(2n/3) + 1?$

Case 3

- If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$,
- and if $a f(n/b) \leq c f(n)$ for some constant $c < 1$ and all sufficiently large n ,
- then $T(n) = \Theta(f(n))$

- Case1
 - If $f(n) = O(n^{\log_b a} - \epsilon)$ for some constant $\epsilon > 0$,
 - then $T(n) = \Theta(n^{\log_b a})$
- Case2
 - If $f(n) = \Theta(n^{\log_b a})$,
 - then $T(n) = \Theta(n^{\log_b a} \lg n)$
- Case3
 - $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$,
 - and if $a f(n/b) \leq c f(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$
- $T(n) = T(n/3) + n$?

- Case1
 - If $f(n) = O(n^{\log_b a} - \epsilon)$ for some constant $\epsilon > 0$,
 - then $T(n) = \Theta(n^{\log_b a})$
- Case2
 - If $f(n) = \Theta(n^{\log_b a})$,
 - then $T(n) = \Theta(n^{\log_b a} \lg n)$
- Case3
 - $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$,
 - and if $a f(n/b) \leq c f(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$
- $T(n) = 3T(n/4) + n \lg n$?

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

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

$$a=2 \quad b=2 \quad f(n)=n^2$$

Case 3:

$$\begin{aligned}f(n) &= \Omega(n^{\log_b a + \epsilon}) \\&= \Omega(n^{1+\epsilon})\end{aligned}$$

$$n^2 = \Omega(n^2)$$

$$2f(\frac{n}{2}) \leq c \cdot f(n)$$

$$c = 2$$

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

$$\text{http://www.knowledgigate.in/GATE}$$

$$T(n) = T(\sqrt{n}) + O(\log n)$$

Let $m = \log n$

$$2^m = n$$
$$2^{m/2} = n^{1/2}$$

$$T(2^m) = T(2^{m/2}) + O(\log 2^m)$$

$$\hookrightarrow x(m) = T(2^m)$$

$$x(m) = x(m/2) + O(m)$$

$$a=1 \quad b=2 \quad f(n) = O(m)$$

$$m^{\log_b a} = m^{\log_2 1} + c$$

$x(m) = O(\log m) \rightarrow T(n) = O(\log \log n)$

$$T(n) = T(n-1) + n^4$$

$$T(n) \leq \begin{cases} c, & \text{if } n \leq 1, \\ aT(n-b) + f(n), & n > 1, \end{cases}$$

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

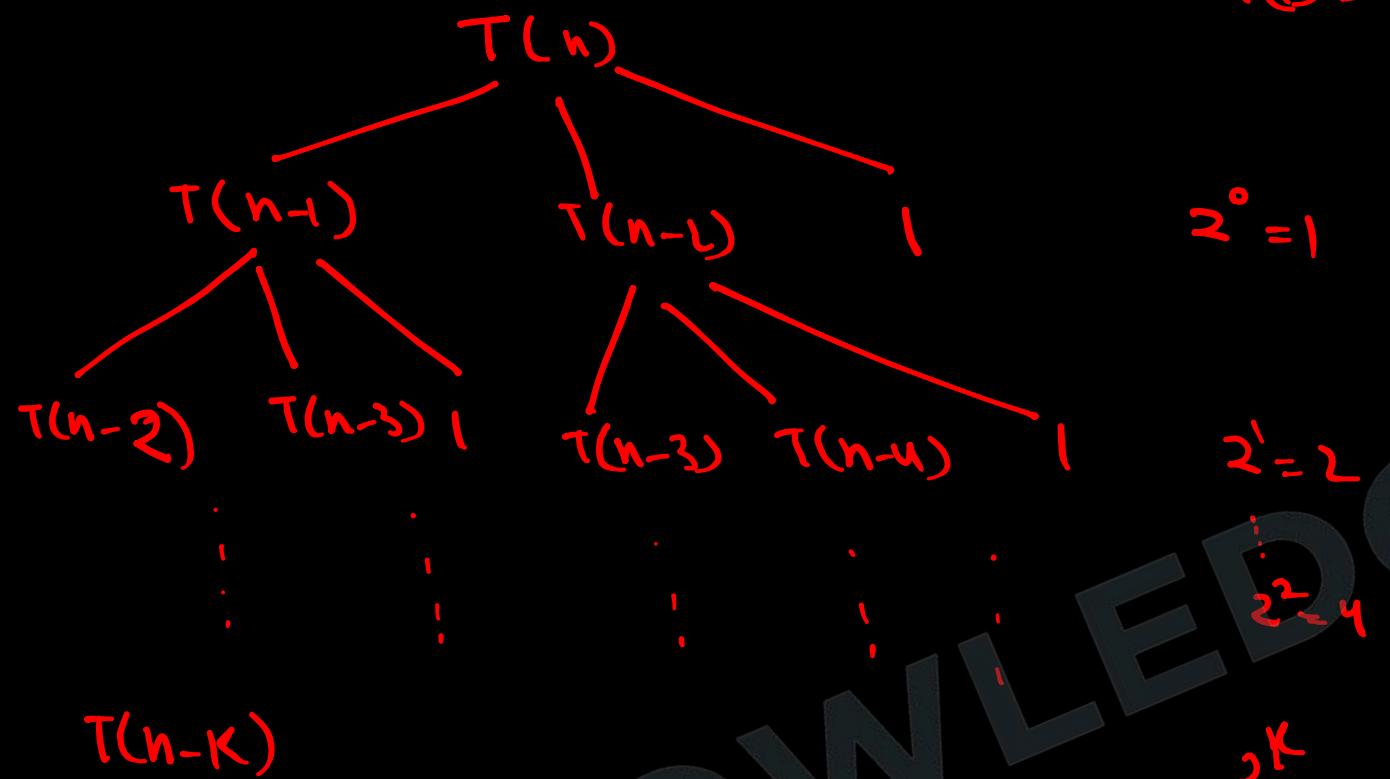
If $f(n)$ is $O(n^k)$, then

1. If $a < 1$ then $T(n) = O(n^k)$
2. If $a = 1$ then $T(n) = O(n^{k+1})$
3. if $a > 1$ then $T(n) = O(n^ka^{n/b})$

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

$$T(0) = 0$$

$$T(1) = 1$$



$$2^0 = 1$$

$$2^1 = 2$$

$$2^2 = 4$$

$$2^K$$

$$n-1 = 1$$

$$K = n-1$$

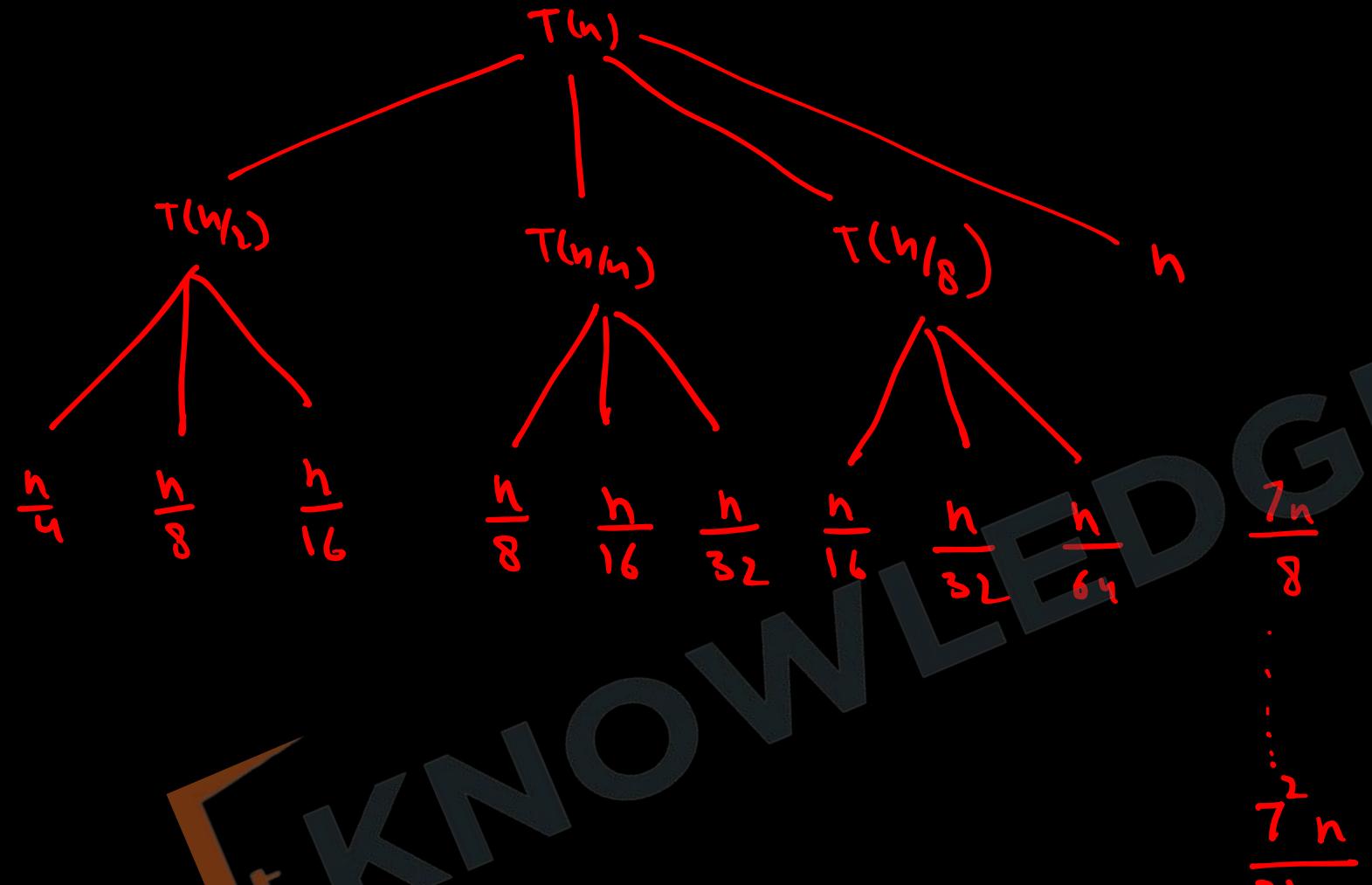
$$= 2^0 + 2^1 + 2^2 + \dots + 2^{n-1}$$

Sum of n terms of geometric progression

$$\frac{a(r^n - 1)}{r-1} = 2^0(2^{n-1} - 1)$$

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

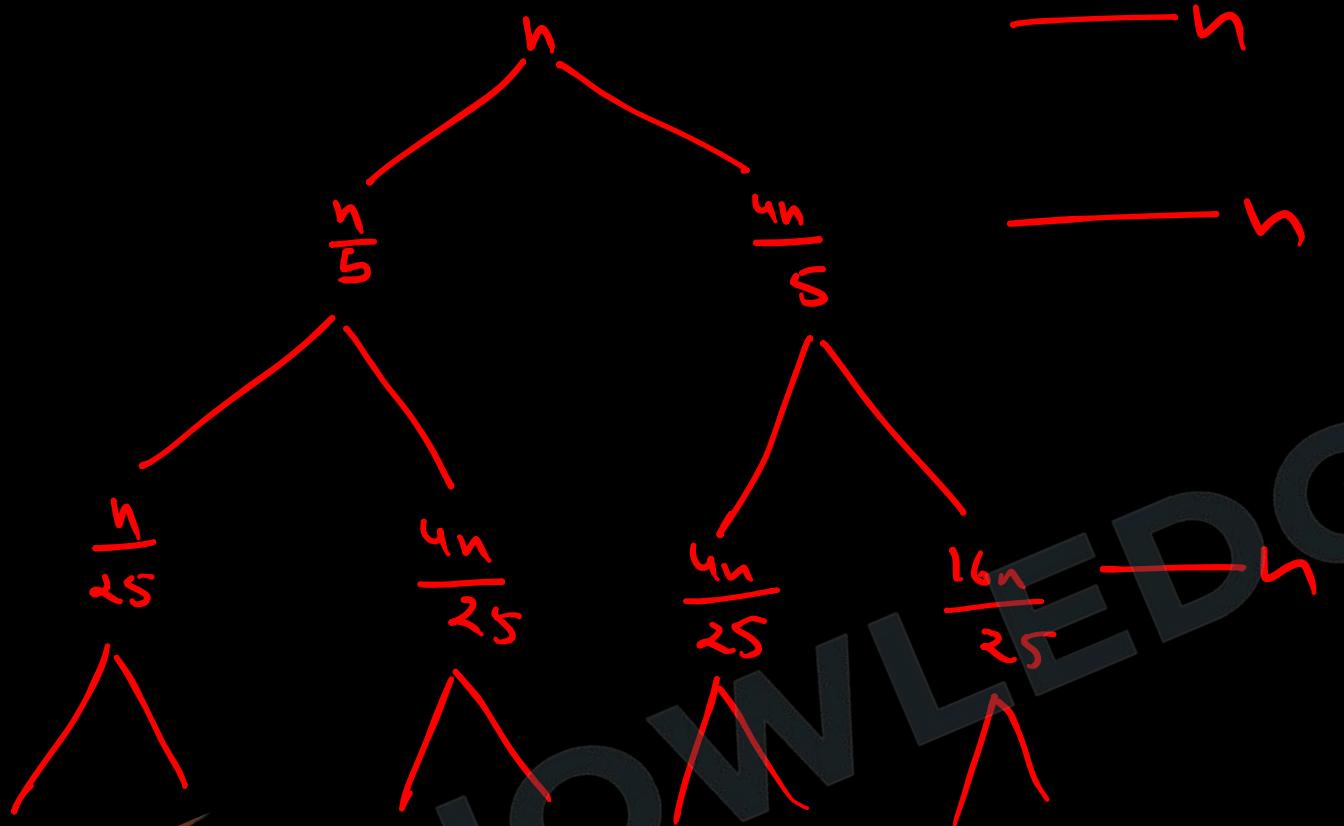
$$T(n) = T(n/2) + T(n/4) + T(n/8) + n$$



$$T(n) = n + \frac{7n}{8} + \frac{7n}{8^2} + \dots + \frac{7n}{8^{\lfloor \log_2 n \rfloor}} + \frac{7n}{8^{\lceil \log_2 n \rceil}}$$

<http://www.knowledgegate.in/GATE>

$$T(n) = n + T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right)$$



$$T(n) = \underbrace{n + n + n + \dots + n}_{\log n} = \Theta(n \log n)$$

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

Iteration method

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

$$= 2^2 \left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2} \right] + n + n = 2^3 T\left(\frac{n}{2^3}\right) + n + n + n$$

$$= 2^2 \left[2T\left(\frac{n}{2^4}\right) + \frac{n}{2^3} \right] + n + n + n = 2^4 T\left(\frac{n}{2^4}\right) + n + n + n + n$$

⋮

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

$$T\left(\frac{n}{2^k}\right) = T(1)$$

$$T(n) = 2^k T(1) + kn$$

$$T(n) = n \cdot \log n + kn$$

$$\underline{\underline{http://www.knowledgegate.in/GATE}}$$

Q Consider the following three functions.

$$f_1 = 10^n$$

$$f_2 = n^{\log n}$$

$$f_3 = n^{\sqrt{n}}$$

Which one of the following options arranges the functions in the increasing order of asymptotic growth rate?

$$f_2 < f_3 < f_4$$

Q Consider the following functions from positives integers to real numbers 10 , \sqrt{n} , n , $\log_2 n$, $100/n$. The CORRECT arrangement of the above functions in increasing order of asymptotic complexity

$$100/n < 10 < \log_2 n < \sqrt{n} < n$$

Ch-2

Sorting and Order Statistics

Concept of Searching, Sequential search, Index Sequential Search, Binary Search Shell Sort, Quick Sort, Merge Sort, Heap Sort, Comparison of Sorting Algorithms, Sorting in Linear Time. Sequential search, Binary Search, Comparison and Analysis Internal Sorting: Insertion Sort, Selection, Bubble Sort, Quick Sort, Two Way Merge Sort, Heap Sort, Radix Sort, Practical consideration for Internal

Sorting.

<http://www.knowledgegate.in/GATE>

The figure displays a performance matrix for nine sorting algorithms (Insertion, Selection, Bubble, Shell, Merge, Heap, Quick, Quick3) against four data distribution types (Random, Nearly Sorted, Reversed, Few Unique). Each cell in the matrix contains a horizontal bar chart representing the execution time of a specific algorithm on a specific data set. The length of each bar corresponds to the time taken for that algorithm to sort the data. A large diagonal watermark 'KNOWLEDGE GATE' is overlaid across the matrix.

	Insertion	Selection	Bubble	Shell	Merge	Heap	Quick	Quick3
Random								
Nearly Sorted								
Reversed								
Few Unique								

School Morning Assembly



KNOWLEDGE GATE

Sorting

- The process of arranging data items (numeric or char) in a specific order either increasing or decreasing order is called sorting. It is an important process which is used in a number of applications as many times we require sorted data.
- There are number of approaches available for sorting and some parameter based on which we judge the performance of these algorithm.

Sorting Algorithm

Best Case

Worst Case

Selection

$O(n^2)$

$O(n^2)$

Bubble

$O(n^2) / O(n)$

$O(n^2)$

Insertion

$O(n)$

$O(n^2)$

Merge

$O(n \log n)$

$O(n \log n)$

Heap

$O(n \log n)$

$O(n \log n)$

Quick

$O(n \log n)$

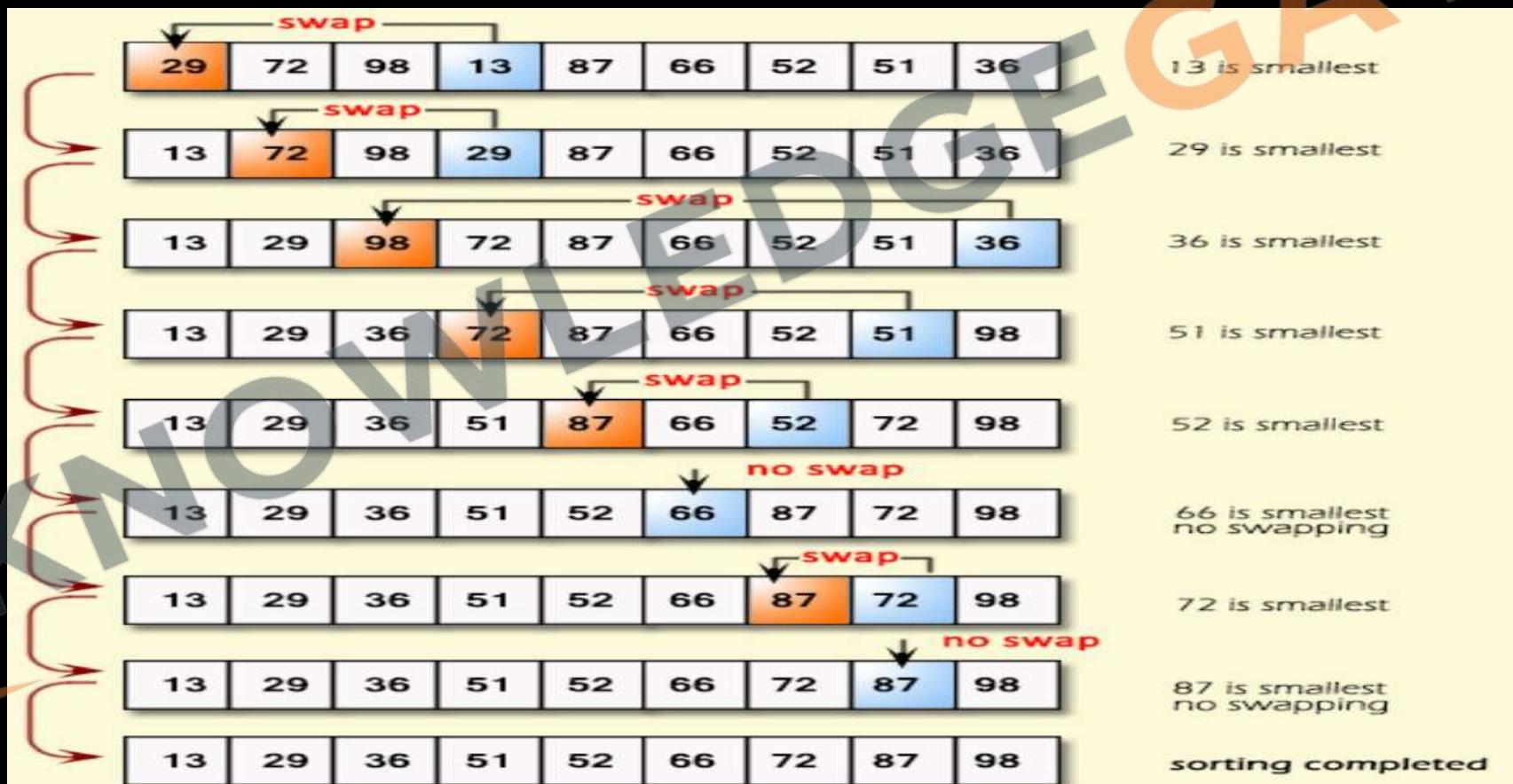
$O(n^2)$

- **Space complexity**: - Internal sorting and External sorting
- **Internal/Inplace sorting** means when a sorting algorithm does not require any additional memory apart from the space acquired by the problem. Eg Heap Sort
- While on the other hand some sorting algo requires additional space called **external sorting** algorithm. Eg Merge Sort

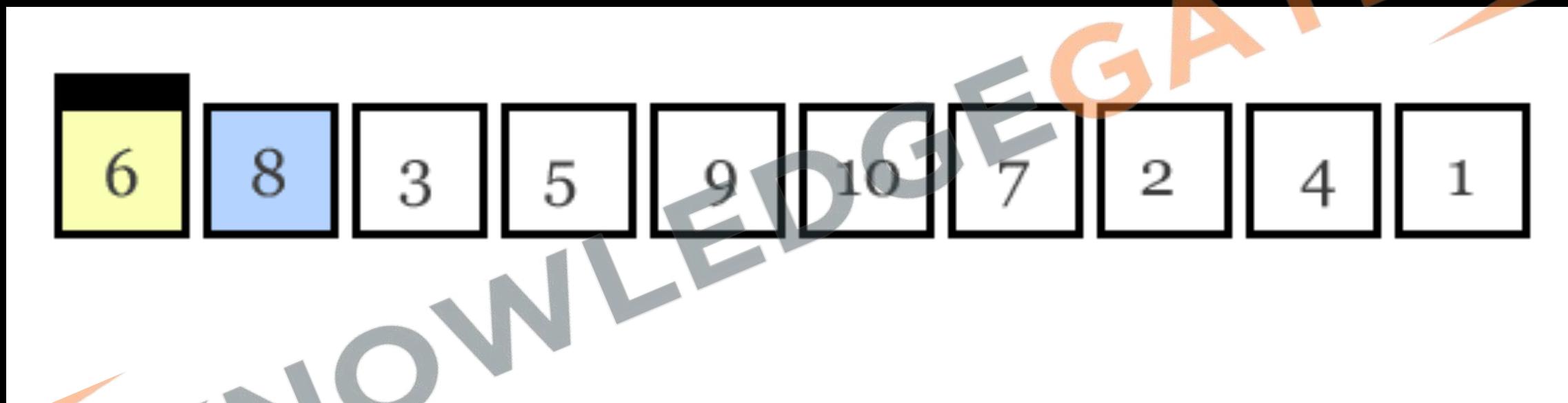
- **Stable or Unstable**: - When the input sequence contains copies then we check whether the output sequence preserve the order of the respective copies or not?
 - Stable(Bubble Sort), Unstable(Insertion Sort)
- Even studying all the criterion, we understand that the different sorting algo have different advantages in different scenarios therefor we must understand the actual logic of the algorithm and then can solve problems based on them.

Selection Sort

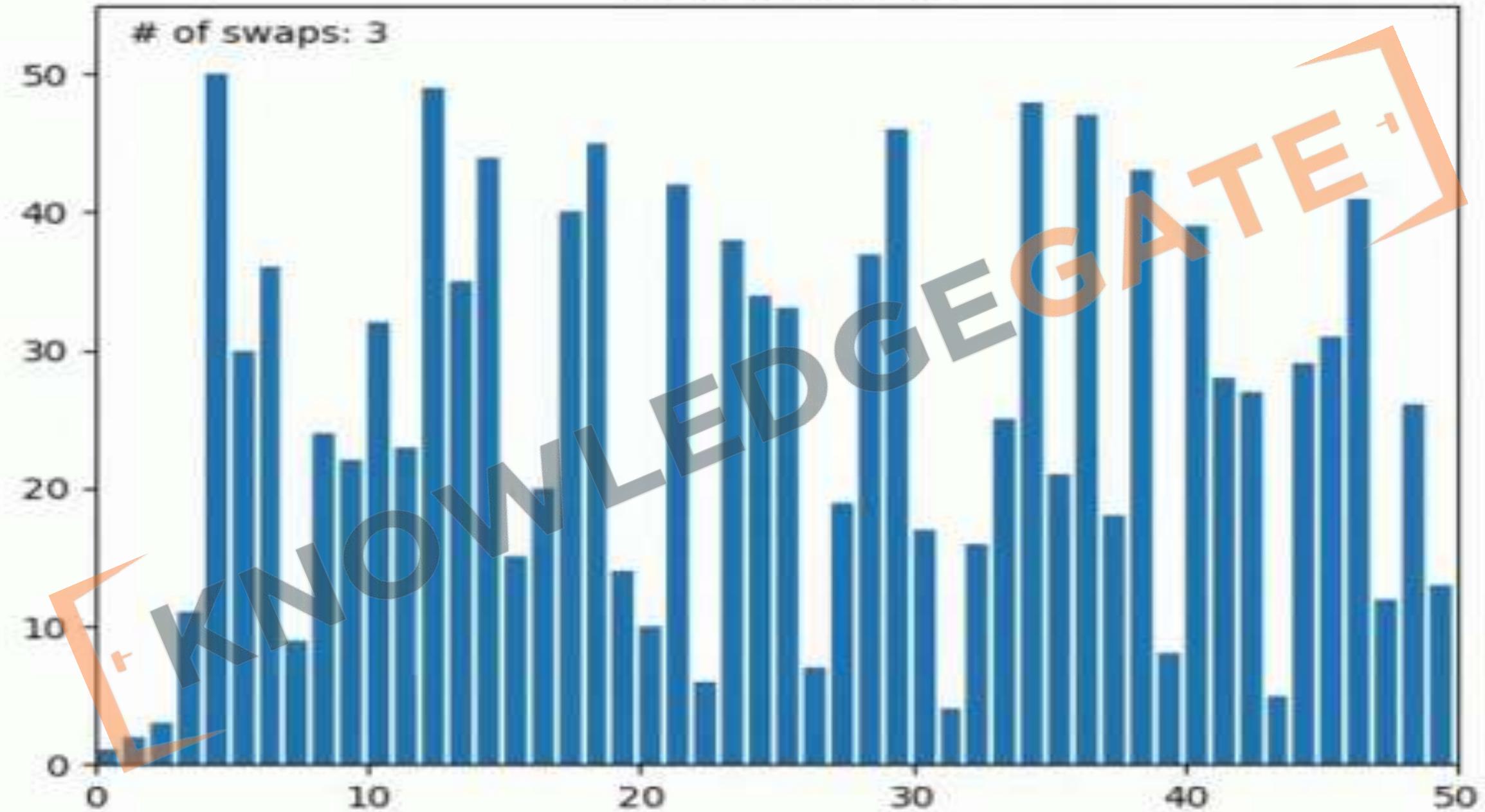
- The algorithm divides the input list into two parts: a sorted sublist of items which is built up from left to right at the front (left) of the list and a sublist of the remaining unsorted items that occupy the rest of the list.



- Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.



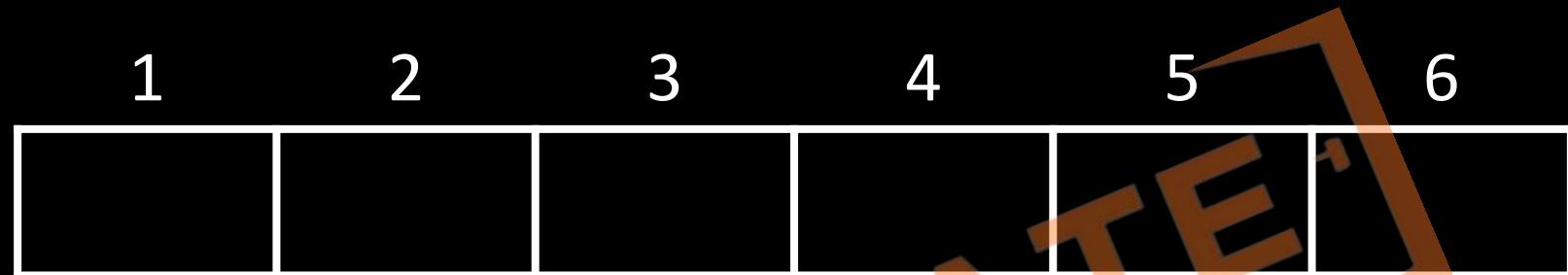
Selection Sort



Selection Sort(Algo)

Selection sort (A, n)

```
{  
    for k ← 1 to n-1  
    {  
        min = A[k]  
        Loc = k  
        for j ← k+1 to n  
        {  
            if(min > A[j])  
            {  
                min = A[j]  
                Loc = j  
            }  
        }  
        swap(A[k],A[Loc])  
    }  
}
```



Selection Sort(Analysis)

Selection sort (A, n)

```
{  
    for k ← 1 to n-1  
    {  
        min = A[k]  
        Loc = k  
        for j ← k+1 to n  
        {  
            if(min > A[j])  
            {  
                min = A[j]  
                Loc = j  
            }  
        }  
        swap(A[k],A[Loc])  
    }  
}
```

- Depends on structure or content ?
 - Structure
- Internal/External sort algorithm ?
 - Internal
- Stable/Unstable sort algorithm ?
 - Unstable
- Best case time complexity ?
 - $O(n^2)$
- Worst case time complexity ?
 - $O(n^2)$
- Algorithmic Approach?
 - Subtract and Conquer

Selection Sort (Conclusion)

- Selection sort is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations(number of swaps, which is $n - 1$ in the worst case).
- It has an $O(n^2)$ time complexity, which makes it inefficient on large lists.

Bubble / Shell / Sinking Sort

6 5 3 1 8 7 2 4



KNOWLEDGE GATE

Bubble / Shell / Sinking Sort

- Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.

Bubble / Shell / Sinking Sort(Algo without flag)

Bubble sort (A, n)

{

for k \leftarrow 1 to n-1

{

ptr = 1

while(ptr <= n-k)

{

if(A[ptr] > A[ptr+1])

{

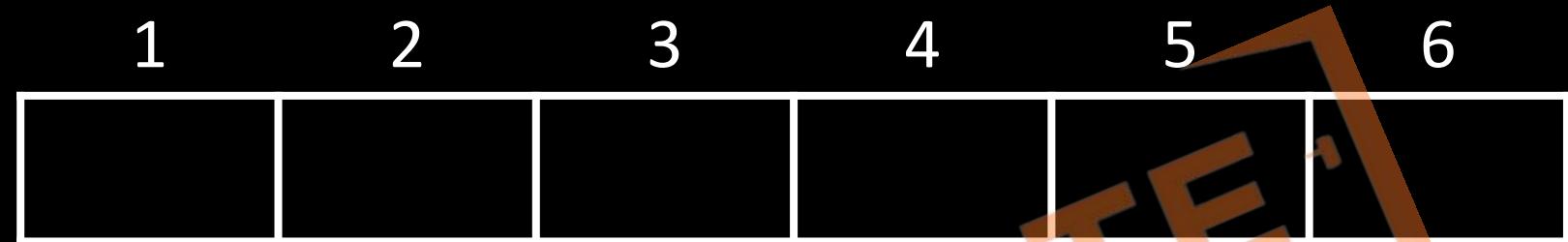
exchange(A[ptr],A[ptr+1])

}

ptr = ptr+1

}

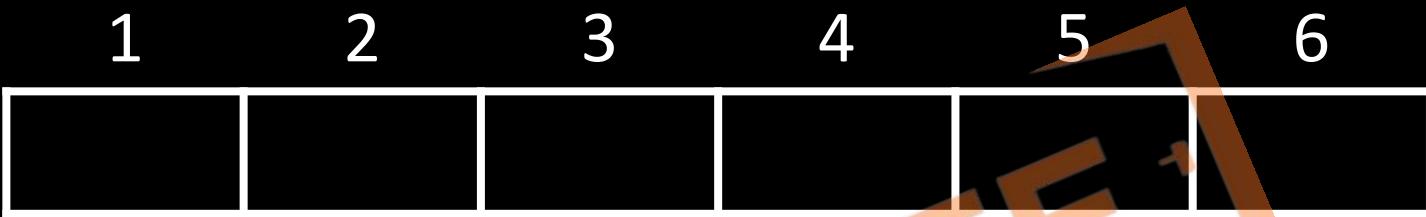
}



Bubble / Shell / Sinking Sort (Algo with flag)

```
Bubble sort (A, n)
{
    for k ← 1 to n-1
    {
        ptr = 1
        while(ptr <= n-k)
        {
            if(A[ptr] > A[ptr+1])
            {
                exchange(A[ptr],A[ptr+1])
            }
            ptr = ptr+1
        }
    }
}

Bubble sort (A, n)
{
    for k ← 1 to n-1
    {
        ptr = 1
        while(ptr <= n-k)
        {
            if(A[ptr] > A[ptr+1])
            {
                exchange(A[ptr],A[ptr+1])
                flag = 1
            }
            ptr = ptr+1
        }
        if(!flag)
        {
            break;
        }
    }
}
```



Bubble / Shell / Sinking Sort (Analysis with flag)

Bubble sort (A, n)

```
{  
    for k ← 1 to n-1  
    {  
        ptr = 1  
        while(ptr <= n-k)  
        {  
            if(A[ptr] > A[ptr+1])  
            {  
                exchange(A[ptr],A[ptr+1])  
                flag = 1  
            }  
            ptr = ptr+1  
        }  
        if(!flag)  
        {  
            break;  
        }  
    }  
}
```

- Depends on structure or content ?
 - Both
- Internal/External sort algorithm ?
 - Internal
- Stable/Unstable sort algorithm ?
 - Stable
- Best case time complexity ?
 - $O(n)$
- Worst case time complexity ?
 - $O(n^2)$
- Algorithmic Approach?
 - Subtract and Conquer

Bubble / Shell / Sinking Sort (Conclusion)

- Even other $O(n^2)$ sorting algorithms, such as insertion sort selection sort, generally run faster than bubble sort, and are no more complex. Therefore, bubble sort is not a practical sorting algorithm. This simple algorithm performs poorly in real world use and is used primarily as an educational tool. More efficient algorithms such as heap sort, or merge sort are used by the sorting libraries built into popular programming languages such as Python and Java.
- When the list is already sorted (best-case), the complexity of bubble sort is only $O(n)$. By contrast, most other algorithms, even those with better average-case complexity, perform their entire sorting process on the set and thus are more complex. However, not only does insertion sort share this advantage, but it also performs better on a list that is substantially sorted (having a small number of inversions).

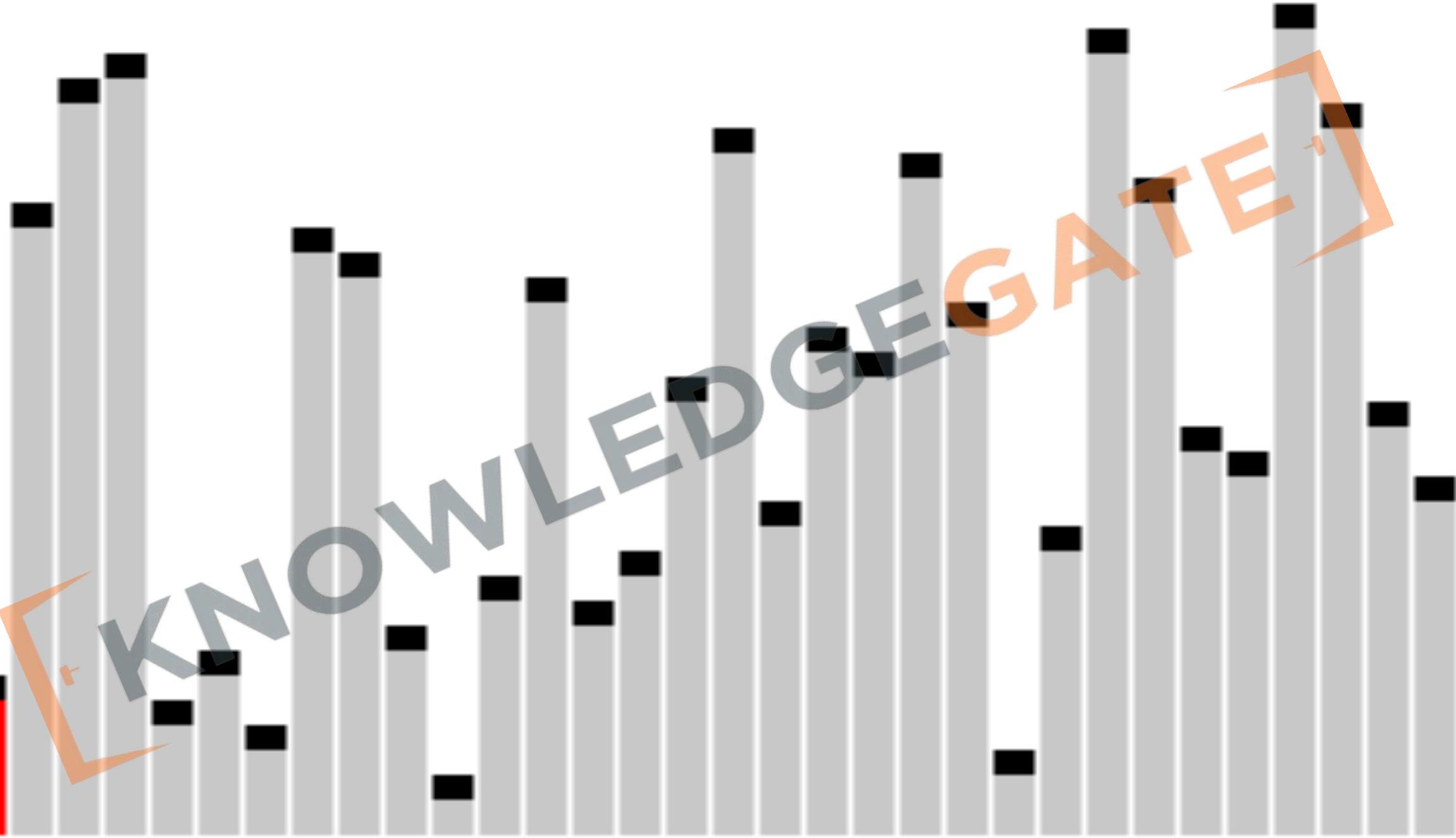


Insertion Sort

6 5 3 1 8 7 2 4



<http://www.knowledgegate.in/GATE>

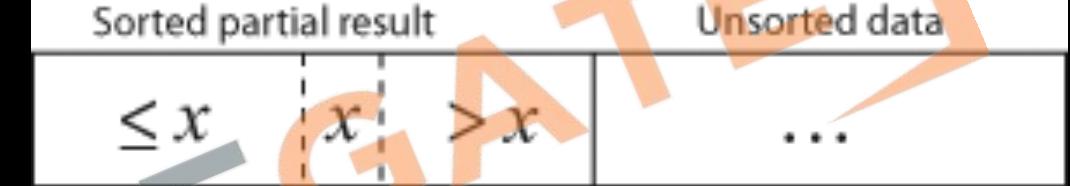
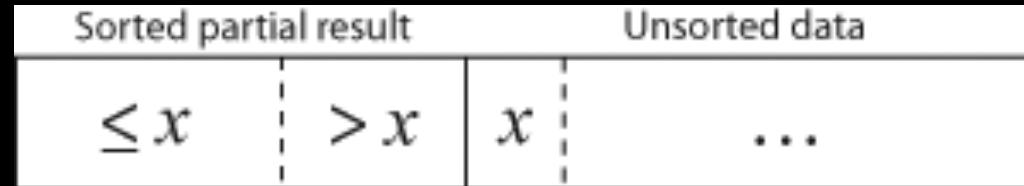


KNOWLEDGE AGGREGATE

Insertion Sort

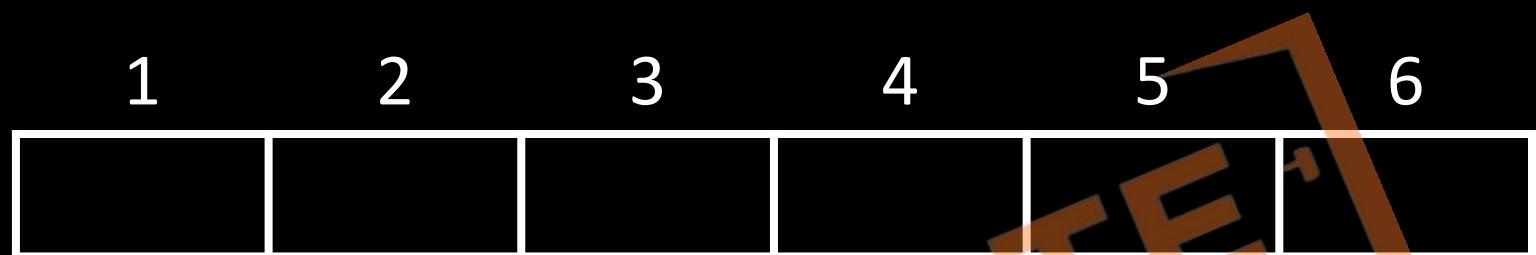
- At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.
- At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked).
- If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

- The resulting array after k iterations has the property where the first $k + 1$ entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result:



Insertion Sort (Algo)

```
Insertion sort (A, n)
{
    for j ← 2 to n
    {
        key = A[j]
        i = j - 1
        while(i>0 and A[i] > key)
        {
            A[i+1] = A[i]
            i = i-1
        }
        A[i+1]=key
    }
}
```



Insertion Sort (Analysis)

```
Insertion sort (A, n)
{
    for j ← 2 to n
    {
        key = A[j]
        i = j - 1
        while(i>0 and A[i] > key)
        {
            A[i+1] = A[i]
            i = i-1
        }
        A[i+1]=key
    }
}
```

- Depends on structure or content ?
 - Both
- Internal/External sort algorithm ?
 - Internal
- Stable/Unstable sort algorithm ?
 - Stable
- Best case time complexity ?
 - $O(n)$
- Worst case time complexity ?
 - $O(n^2)$
- Algorithmic Approach?
 - Subtract and Conquer

Insertion Sort (Conclusion)

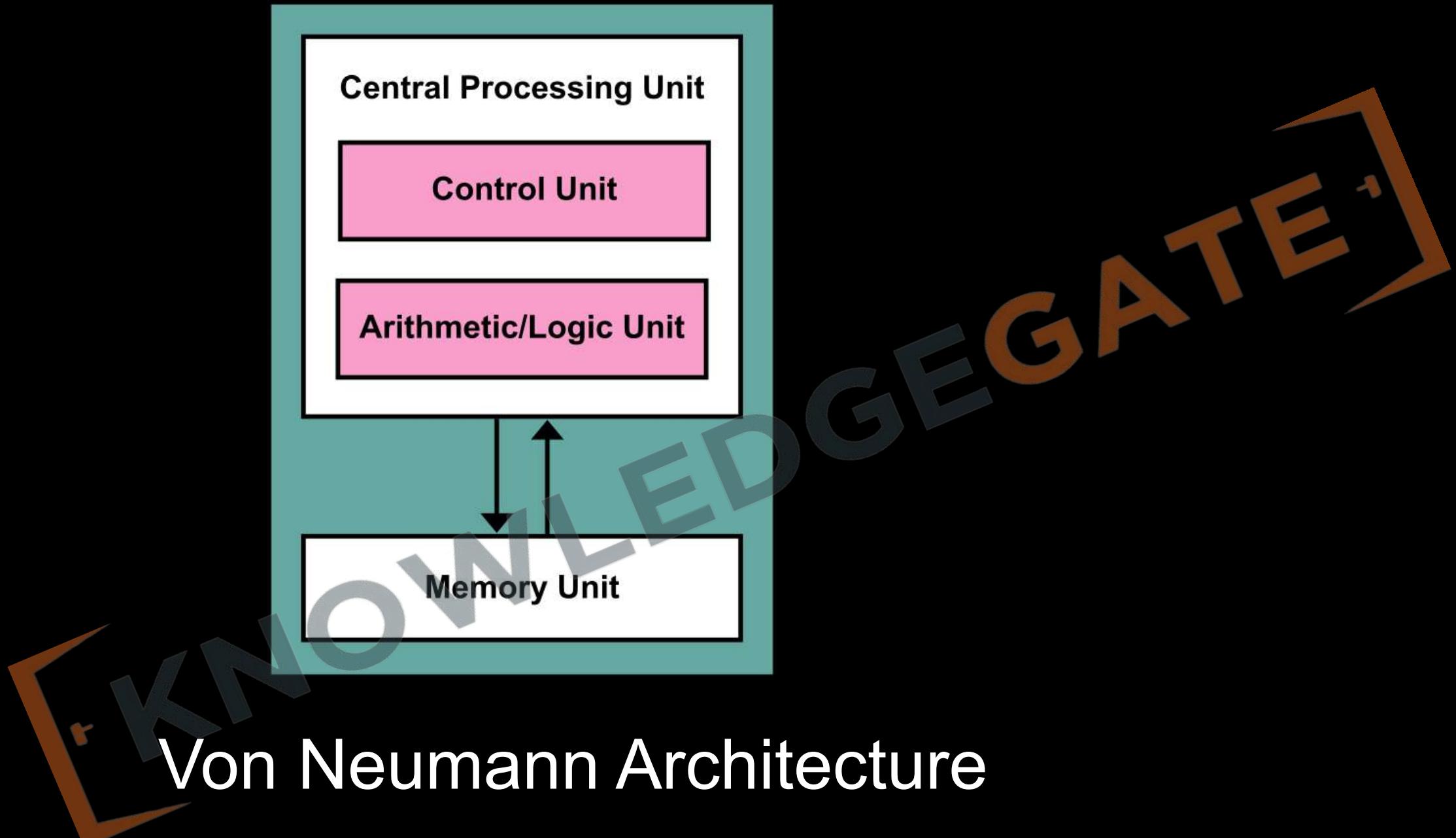
- **Insertion sort** is much less efficient on large lists than more advanced algorithms such as heapsort($O(n\log n)$), or merge sort ($O(n\log n)$). However, insertion sort provides several advantages:
 - Efficient for (quite) small data sets, much better other quadratic sorting algorithms such as selection and bubble sort.

Merge Sort

- In computer science, merge sort is an efficient, general-purpose, comparison-based sorting algorithm.
- Merge sort is a divide and conquer algorithm that was invented by John von Neumann in 1945.



<http://www.knowledgegate.in/GATE>



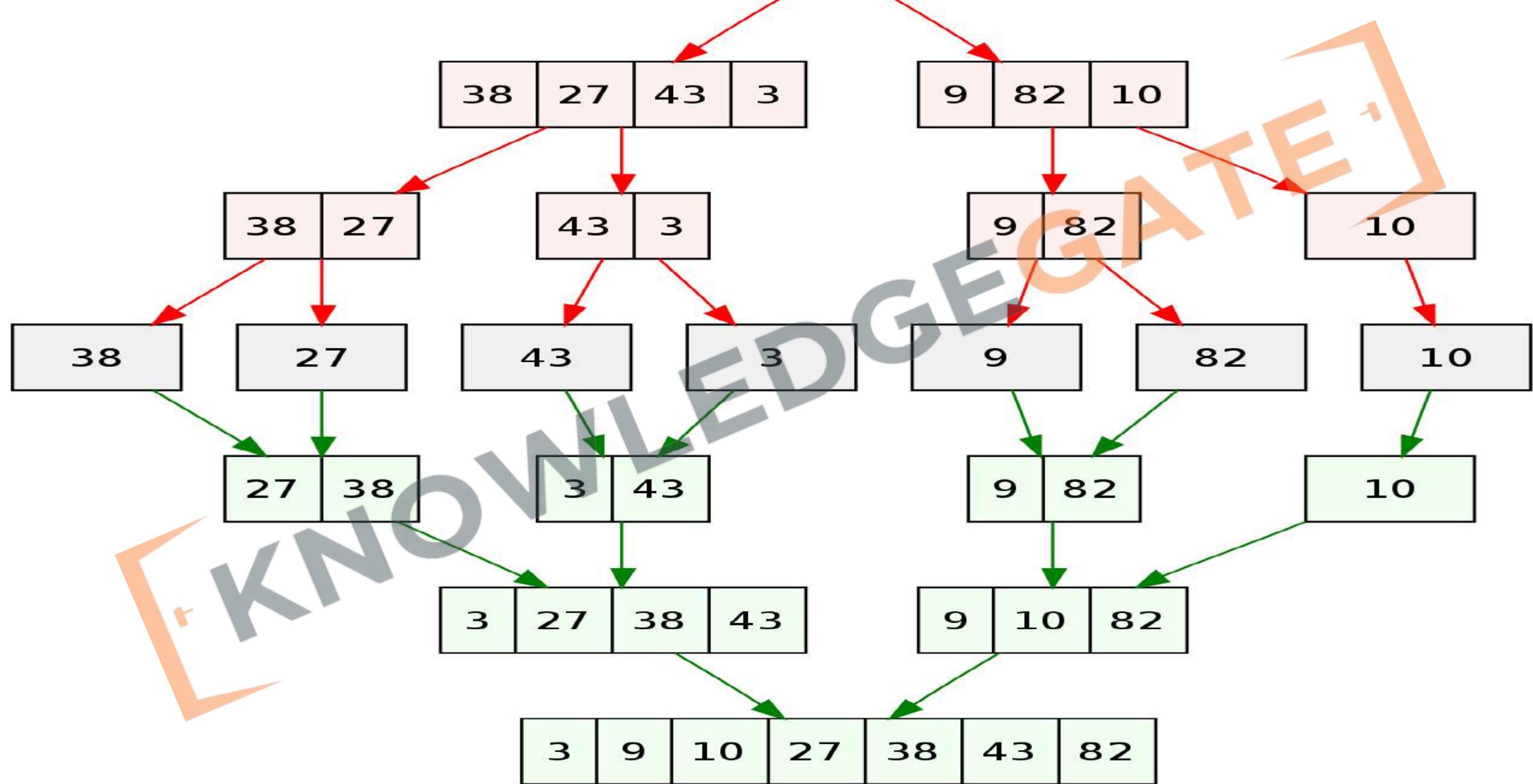
Von Neumann Architecture

<http://www.knowledgegate.in/GATE>

Merge Sort

- Conceptually, a merge sort works as follows:
 - Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted).
 - Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.

38	27	43	3	9	82	10
----	----	----	---	---	----	----





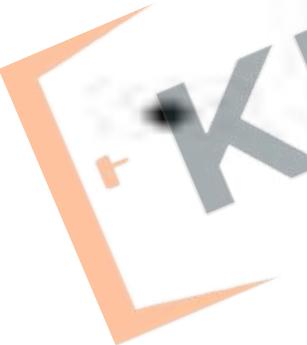
WHEN THE

KNOWLEDGESEGATE[®]

WHEN THE MERGE IS SORTING

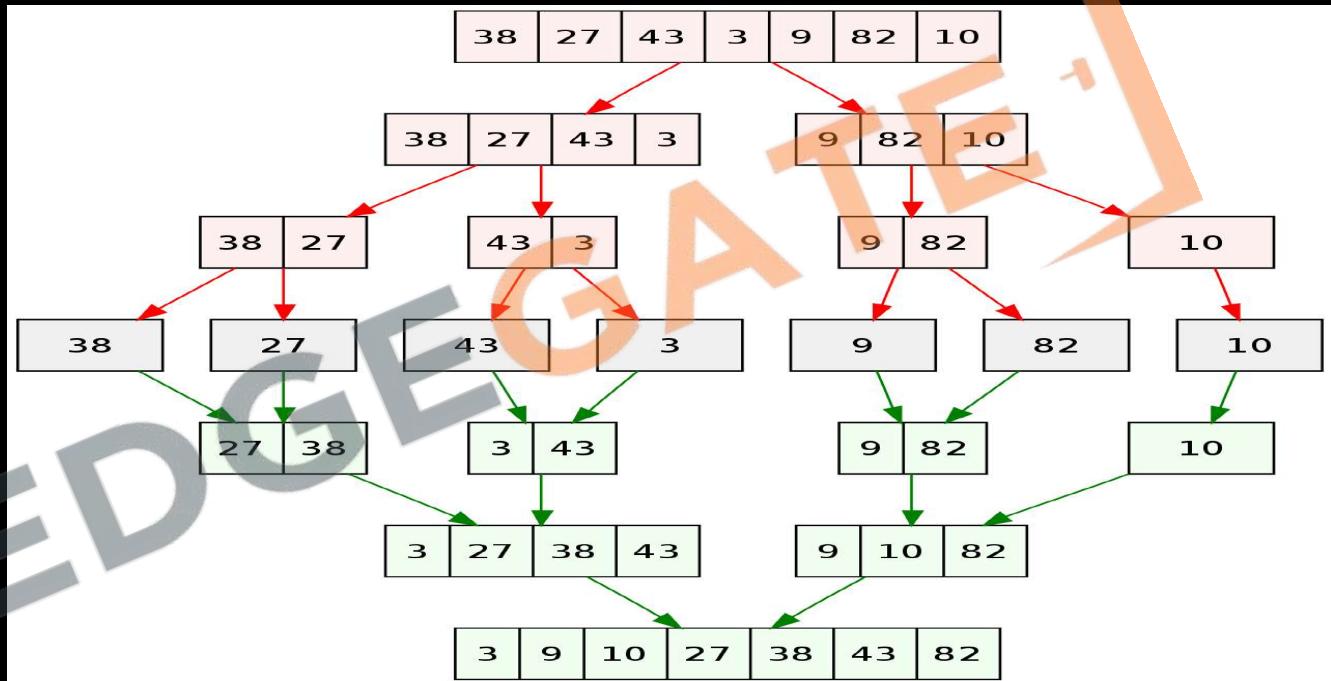
Merge Sort



 KNOWLEDGEGATE[®]

Merge Sort(Algo)

```
Merge_Sort(A, p, r)
{
    if(p < r)
    {
        q ← ⌊ (p + r)/2 ⌋
        Merge_Sort (A, p, q)
        Merge_Sort (A, q + 1, r)
        Merge (A, p, q, r)
    }
}
```



```
Merge (A, p, q, r)
```

```
{
```

```
    n1 ← q - p + 1
```

```
    n2 ← r - q
```

```
Create array L [1.....n1+1] and R [1.....n2+1]
```

```
for i ← 1 to n1
```

```
    do L[i] = A [p + i - 1]
```

```
for j ← 1 to n2
```

```
    do R[j] = A [j + q]
```

```
L[n1+1] ← ∞
```

```
R[n2+1] ← ∞
```

```
i ← 1
```

```
j ← 1
```

```
for k ← p to r
```

```
{
```

```
    if(L[i] <= R[j])
```

```
    {
```

```
        A[k] = L[i]
```

```
        i = i + 1
```

```
    }
```

```
    Else
```

```
    {
```

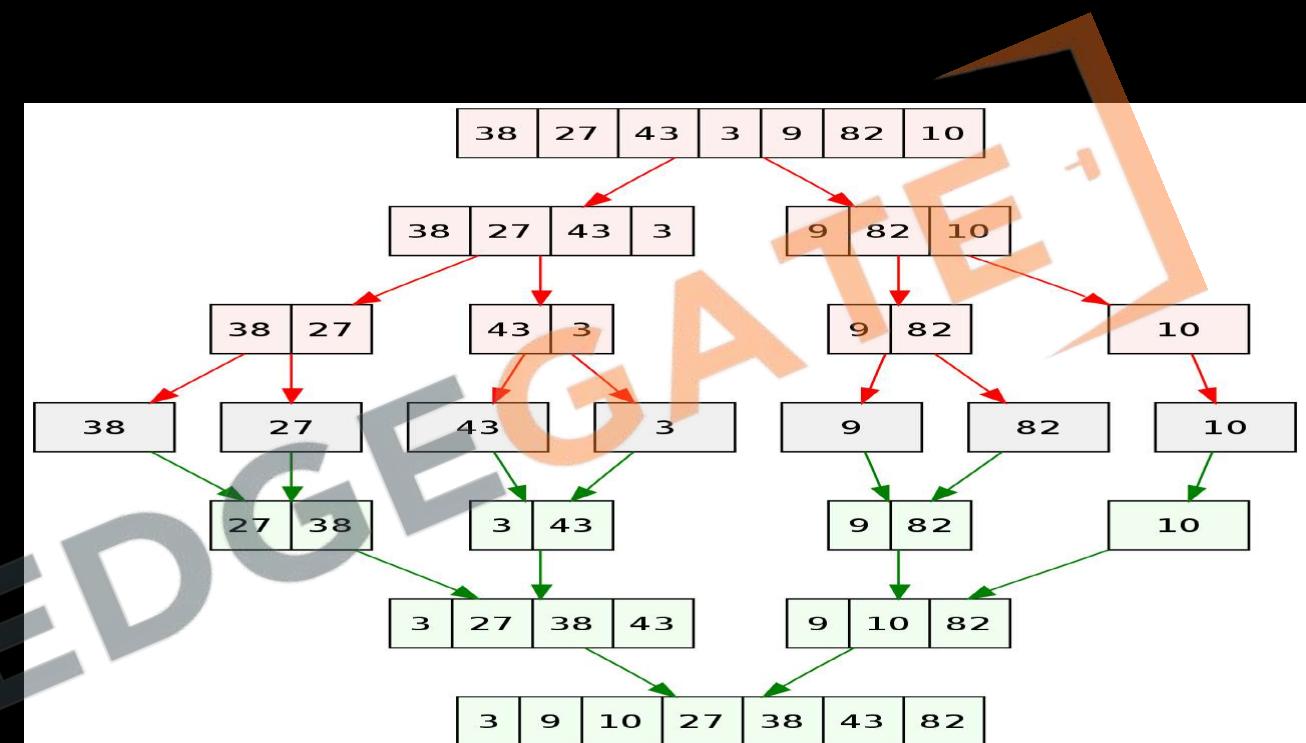
```
        A[k] = R[j]
```

```
        j = j + 1
```

```
}
```

```
}
```

Merge Sort(Algo)



Merge Sort(Analysis)

- Depends on structure or content ?
 - Structure
- Internal/External sort algorithm ?
 - External
- Stable/Unstable sort algorithm ?
 - Stable
- Best case time complexity ?
 - $O(n \log n)$
- Worst case time complexity ?
 - $O(n \log n)$
- Algorithmic Approach?
 - Divide and Conquer

Merge Sort(Conclusion)

- If the running time of merge sort for a list of length n is $T(n)$, then the recurrence
 - $T(n) = 2T(n/2) + n$
- In the *worst* case, merge sort does about 39% fewer comparisons than quicksort does in the *average* case.
- Merge sort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as Lisp, where sequentially accessed data structures are very common.
- Merge sort's requires $O(n)$ space complexity.

Heap Sort

- Heapsort was invented by J. W. J. Williams in 1964. This was also the birth of the heap, presented already by Williams as a useful data structure in its own right.



Heap Sort

- In computer science, heapsort is a comparison-based sorting algorithm.
- Heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region.
- Heapsort does not waste time with a linear-time scan of the unsorted region; rather, heap sort maintains the unsorted region in a heap data structure to more quickly find the largest element in each step.

- The heapsort algorithm can be divided into two parts.
 - In the first step, a heap is built out of the data. The heap is often placed in an array with the layout of a complete binary tree.
 - In the second step, a sorted array is created by repeatedly removing the largest element from the heap (the root of the heap), and inserting it into the array. The heap is updated after each removal to maintain the heap property. Once all objects have been removed from the heap, the result is a sorted array.



Heap Sort(Algo)

```
Heap_Sort(A)
{
    Build_Max_heap(A)
    for i ← length[A] down to 2
    {
        do exchange (A[1] ↔ A[i])
        Heap-size[A] ← Heap-size[A] - 1
        Max-Heapify(A,1)
    }
}
```



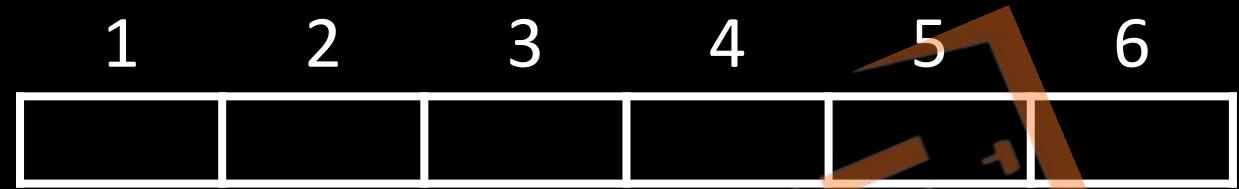
Heap Sort(Algo)

```
Build_Max_Heap(A)
{
    Heap-size[A] ← length[A]
    for i ← ⌊length[A]/2⌋ down to 1
    {
        do Max-Heapify (A, i)
    }
}
```



Heap Sort(Algo)

```
Max-Heapify(A, i)
{
    L ← Left[i]
    R ← Right[i]
    if( L <= Heap_size[A] and A[L] > A[i])
        Largest ← L
    Else
        Largest ← i
    if(R <= Heap_size[A] and A[r] > A[Largest])
        Largest ← R
    if(Largest != i)
    {
        Exchange( A[i] ↔ A[Largest])
        Max-Heapify(A, Largest)
    }
}
```

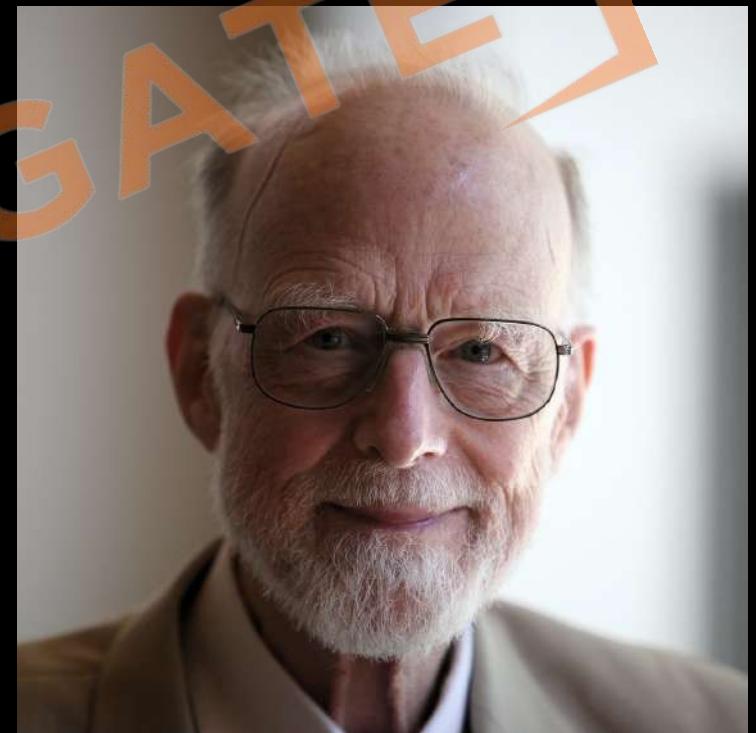


Heap Sort(Analysis)

- Depends on structure or content ?
 - Both
- Internal/External sort algorithm ?
 - Internal
- Stable/Unstable sort algorithm ?
 - Unstable
- Best case time complexity ?
 - $O(n \log n)$
- Worst case time complexity ?
 - $O(n \log n)$
- Algorithmic Approach?
 - Mixed Approach

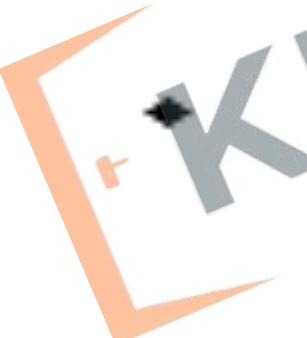
Quick Sort

- **Quicksort** is an developed by British computer scientist Tony Hoare and published in 1961, it is still a commonly used algorithm for sorting. When implemented well, it can be somewhat faster than merge sort and about two or three times faster than heapsort.
- His work earned him the Turing Award, usually regarded as the highest distinction in computer science, in 1980.
- Along with Edsger Dijkstra, formulated the dining philosophers problem. He is also credited with development of the null pointer.



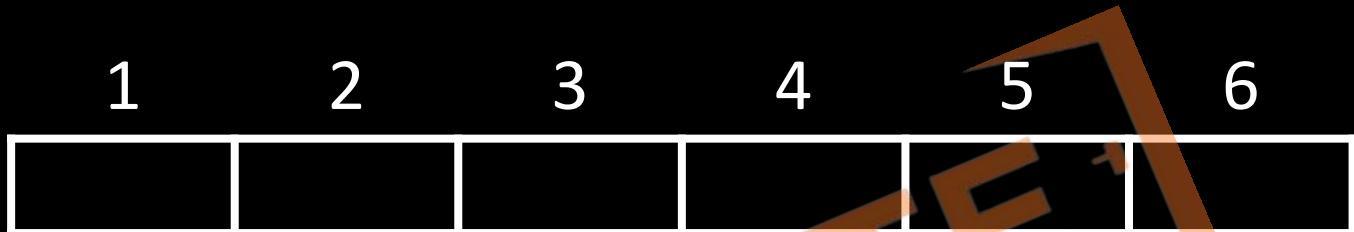
- Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
- The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.



 KNOWLEDGE¹GATE¹

Quick Sort(Algo)

```
Quick_Sort(A, p, r)
{
    if(p < r)
    {
        q ← partition (A, p, r)
        quick_Sort(A, p, q - 1)
        quick_Sort(A, q + 1, r)
    }
}
```



Quick Sort(Algo)

Partition (A, p, r)

{

 x \leftarrow A[r]

 i \leftarrow p - 1

 for j \leftarrow p to r - 1

{

 if(A[j] \leq x)

{

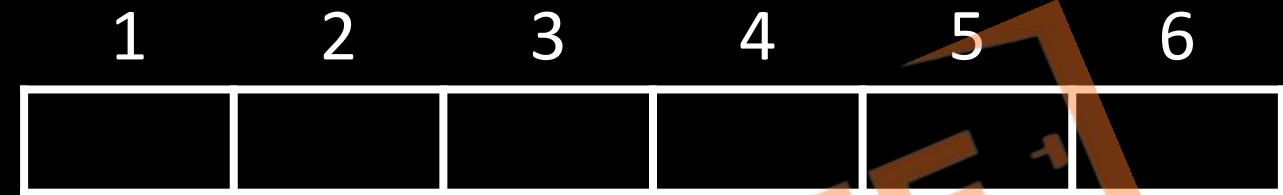
 i \leftarrow i + 1

 Exchange(A[i] \leftrightarrow A[j])

}

 Exchange(A[i + 1] \leftrightarrow A[r])

return i+1



Quick Sort(Analysis)

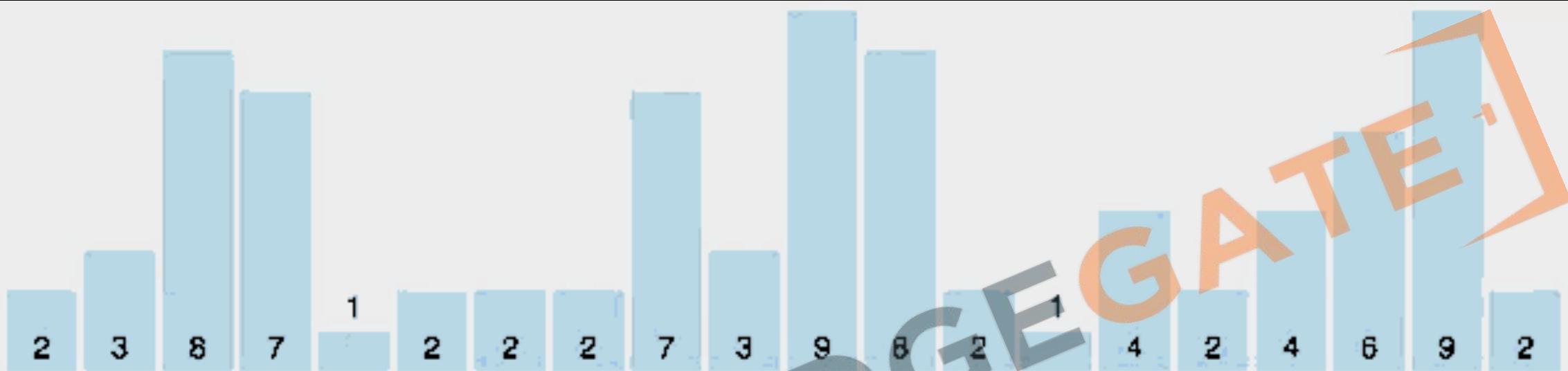
- Depends on structure or content ?
 - Both
- Internal/External sort algorithm ?
 - Internal
- Stable/Unstable sort algorithm ?
 - Unstable
- Best case time complexity ?
 - $O(n \log n)$
- Worst case time complexity ?
 - $O(n^2)$
- Algorithmic Approach?
 - Divide and Conquer

Counting Sort

- Although radix sorting itself dates back far longer, counting sort, and its application to radix sorting, were both invented by Harold H. Seward in 1954.



Counting Sort



KNOWLEDGE GATE

Counting Sort

- Counting sort is a non-comparative stable sorting algorithm suitable for sorting elements within a specific range. It counts the number of objects that have distinct key values, and then does some arithmetic to calculate the position of each object in the output sequence.
- Counting sort is efficient if the range of the input data is not significantly greater than the number of objects to be sorted. It's not a comparison-based sort, and its time complexity is $O(n+k)$, where n is the number of elements in the input array and k is the range of the input. Space complexity is $O(k)$.

Counting Sort

```
Counting_Sort(A, B, k)
{
    let C[0..k] be a new array
    For i ← 0 to k
        do C[i] ← 0
    for j ← 1 to length[A]
        do C[A[j]] ← C[A[j]] + 1
    for i ← 1 to k
        do C[i] ← C[i] + C[i-1]
    for j ← length[A] down to 1
        do B[C[A[j]]] ← A[j]
        C[A[j]] ← C[A[j]] - 1
}
```

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

Given array : Time complexity of counting sort is $O(n)$.

1	2	3	4	5	6	7	8	9	10	
A	1	6	3	3	4	5	6	3	4	5

Step 1 : $i = 0 \text{ to } 6$ $k = 6$ (largest element in array A)

$$C[i] \leftarrow 0$$

0	1	2	3	4	5	6
C	0	0	0	0	0	0

Step 2 : $j = 1 \text{ to } 10$ $(\because \text{length } [A] = 10)$

$$C[A[j]] \leftarrow C[A[j]] + 1$$

For $j = 1$

$$C[A[1]] \leftarrow C[1] + 1 = 0 + 1 = 1 \quad C \begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \boxed{0} & 1 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$C[1] \leftarrow 1$$

For $j = 2$

$$C[A[2]] \leftarrow C[6] + 1 = 0 + 1 = 1 \quad C \begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \boxed{0} & 1 & 0 & 0 & 0 & 0 & 1 \end{array}$$

$$C[6] \leftarrow 1$$

$$\text{Similarly for } j = 5, 6, 7, 8, 9, 10 \quad C \begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \boxed{0} & 1 & 0 & 3 & 2 & 2 & 2 \end{array}$$

Step 3 :

For $i = 1 \text{ to } 6$

$$C[i] \leftarrow C[i] + C[i - 1]$$

For $i = 1$

$$C[1] \leftarrow C[1] + C[0] \quad C \begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \boxed{0} & 1 & 0 & 3 & 2 & 2 & 2 \end{array}$$

$$C[1] \leftarrow 1 + 0 = 1$$

For $i = 2$

$$C[2] \leftarrow C[2] + C[1] \quad C \begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \boxed{0} & 1 & 1 & 3 & 2 & 2 & 2 \end{array}$$

$$C[1] \leftarrow 1 + 0 = 1$$

$$\text{Similarly for } i = 4, 5, 6 \quad C \begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \boxed{0} & 1 & 1 & 4 & 6 & 8 & 10 \end{array}$$

Step 4 :

For $j = 10 \text{ to } 1$

$$B[C[A[j]]] \leftarrow A[j]$$

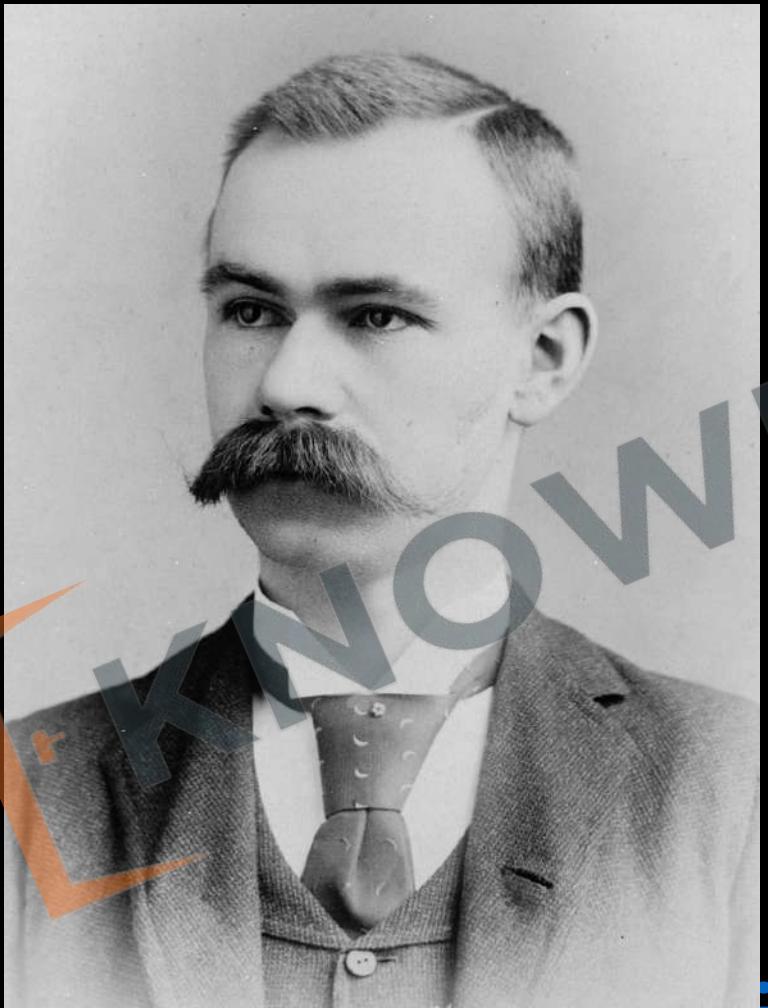
$$C[A[j]] \leftarrow C[A[j] - 1]$$

j	$A[j]$	$C[A[j]]$	$B[C[A[j]]] \leftarrow A[j]$	$C[A[j]] \leftarrow C[A[j] - 1]$
10	5	8	$B[8] \leftarrow 5$	$C[5] \leftarrow 7$
9	4	6	$B[6] \leftarrow 4$	$C[4] \leftarrow 5$
8	3	4	$B[4] \leftarrow 3$	$C[3] \leftarrow 3$
7	6	10	$B[10] \leftarrow 6$	$C[6] \leftarrow 9$
6	5	7	$B[7] \leftarrow 5$	$C[5] \leftarrow 6$
5	4	5	$B[5] \leftarrow 4$	$C[4] \leftarrow 4$
4	3	3	$B[3] \leftarrow 3$	$C[3] \leftarrow 2$
3	3	2	$B[2] \leftarrow 3$	$C[3] \leftarrow 1$
2	6	9	$B[9] \leftarrow 6$	$C[6] \leftarrow 8$
1	1	1	$B[1] \leftarrow 1$	$C[1] \leftarrow 0$

1	2	3	4	5	6	7	8	9	10	
B	1	3	3	3	4	4	5	5	6	6

Radix Sort

- Radix sort dates back as far as 1887 to the work of Herman Hollerith on tabulating machines. Radix sorting algorithms came into common use as a way to sort punched cards as early as 1923.



170
45
75
90
802
24
2
66

KNOWLEDGE GATE
<http://www.knowledgegate.in/GATE>

Radix Sort

- An IBM card sorter performing a radix sort on a large set of punched cards. Cards are fed into a hopper below the operator's chin and are sorted into one of the machine's 13 output baskets, based on the data punched into one column on the cards. The crank near the input hopper is used to move the read head to the next column as the sort progresses. The rack in back holds cards from the previous sorting pass.



Radix Sort

- Radix Sort is an integer sorting algorithm that organizes data by individual digits which have the same position and value. It starts by sorting integers based on their least significant digit using a stable sorting method like counting sort to keep the same relative order for similar key values. After sorting by the least significant digit, it progresses to the next digit to the left, continuing this process until it has sorted by the most significant digit.
- The time complexity is generally $O(nk)$, where n is the number of elements and k is the number of passes needed, which depends on the number of digits in the largest number.
- Radix Sort excels at sorting fixed-length number sequences like phone numbers or dates and may outperform comparison-based sorts such as quicksort or mergesort if the numbers aren't much longer than the array size. It's particularly adept at handling large data sets because its speed depends more on digit count rather than the actual size of the numbers being sorted.

Radix Sort

```
radixSort(arr)
```

```
{
```

max = largest element in the given array

d = number of digits in the largest element (or, max)

Now, create d buckets of size 0 - 9

for i -> 0 to d

sort the array elements using counting sort (or any stable sort) according to the digits at the ith place

```
}
```

Bucket Sort

- Bucket sort, also known as bin sort, is an effective sorting algorithm ideal for uniformly distributed data. Here are the condensed key points:
 - **Element Distribution:** Elements are distributed across several buckets based on their value.
 - **Sorting Buckets:** Each bucket is sorted individually, either using another sorting algorithm or by applying bucket sort recursively.
 - **Combining Buckets:** Sorted buckets are merged back into a single array for the final sorted order.
 - **Performance:** Best for data evenly distributed over a range. Average and best-case complexity is $O(n+k)$, with n being the number of elements and k the number of buckets.
 - **Space Requirement:** Occupies $O(n \cdot k)$ space due to the buckets.

- **Stability**: Maintains the relative order of equal elements.
- **Ideal Scenarios**: Most efficient for large sets of floating-point numbers or uniformly spread data.
- **Drawbacks**: Performance drops for non-uniformly distributed data and depends on the input distribution and bucket count.
- In brief, bucket sort is fast and stable, best for evenly distributed datasets. It segregates elements into buckets, sorts these, and then merges them into a sorted array.

Bucket Sort

.79	.43	.60	.11	.32	.29	.57	.82	.94	.07
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

0
1
2
3
4
5
6
7
8
9

KNOWLEDGE GATE

BUCKET-SORT (A)

{

let B[0 .. n - 1] be a new array

n = A.length

for i = 0 to n - 1

 make B[i] an empty list

for i = 1 to n

 Insert A[i] into list B[nA[i]]

for i = 0 to n - 1

 sort list B[i] with insertion sort

concatenate the lists together

B[0], B[1],...B[n - 1] (in order)

}

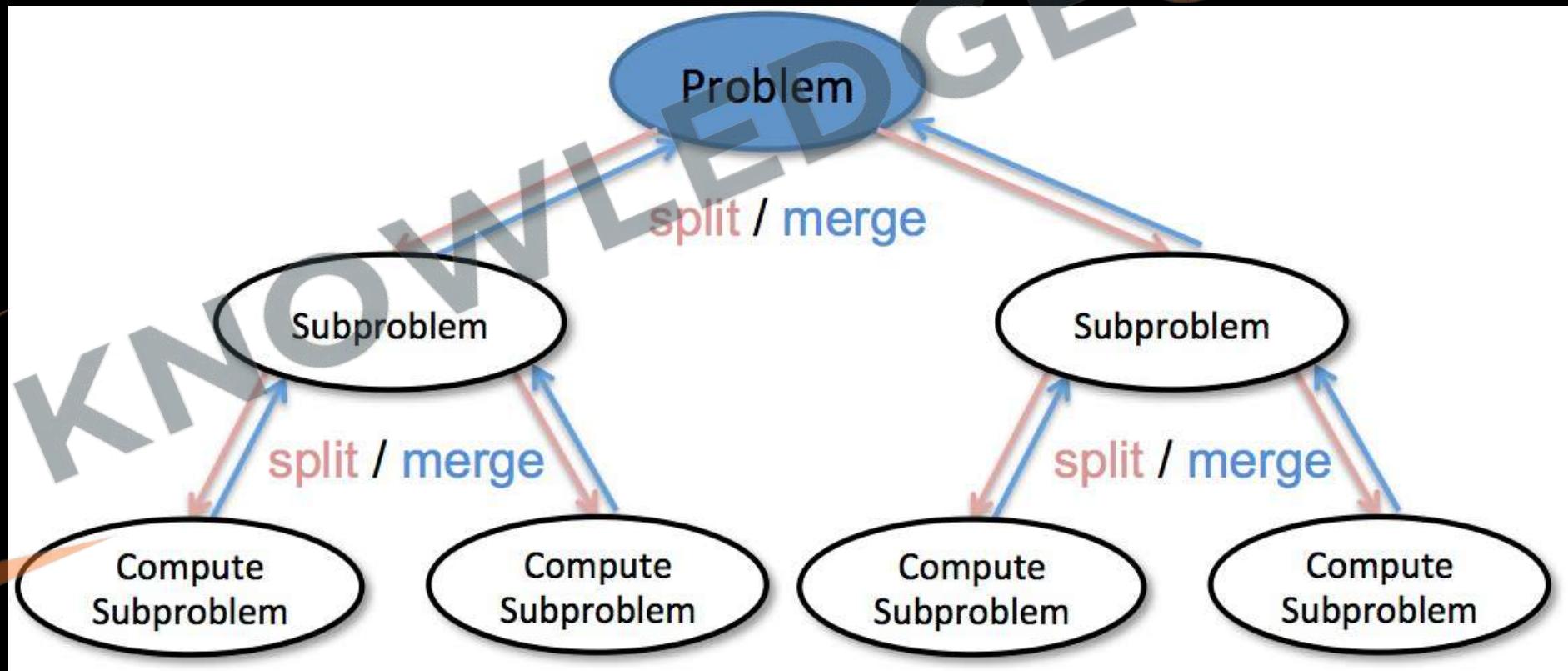
Ch-3

Divide and Conquer

with Examples Such as Sorting,
Matrix Multiplication, Convex
Hull and Searching.

Divide and conquer

- Divide and conquer is a fundamental algorithm design paradigm in computer science. It works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.



- Here are the main steps involved in a divide and conquer algorithm:
 - **Divide**: Split the problem into several sub-problems that are smaller instances of the same problem.
 - **Conquer**: Solve the sub-problems recursively. If the sub-problems are small enough, solve them in a straightforward manner.
 - **Combine**: Combine the solutions of the sub-problems into the solution for the original problem.
- The efficiency of a divide and conquer algorithm depends on the size reduction at each division and the efficiency of the combine step. When properly designed, such algorithms can lead to significant reductions in time complexity, often achieving logarithmic growth in computational cost.

Merge Sort



<http://www.knowledgegate.in/GATE>

QuickSort



<http://www.knowledgegate.in/GATE>

What is Matrix Multiplication

3 × 3 Matrix Multiplication

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} (aj + bm + cp) & (ak + bn + cq) & (al + bo + cr) \\ (dj + em + fp) & (dk + en + fq) & (dl + eo + fr) \\ (gj + hm + ip) & (gk + hn + iq) & (gl + ho + ir) \end{bmatrix}$$

SQUARE-MATRIX-MULTIPLY(A, B)

```
1   $n = A.rows$ 
2  let  $C$  be a new  $n \times n$  matrix
3  for  $i = 1$  to  $n$ 
4      for  $j = 1$  to  $n$ 
5           $c_{ij} = 0$ 
6          for  $k = 1$  to  $n$ 
7               $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
8  return  $C$ 
```

Matrix Multiplication with Divide and Conquer

$$A = [a_{ij}]$$

$$B = [b_{ij}]$$

$$AXB = [a_{ij} * b_{ij}]$$

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

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

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

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

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

Matrix Multiplication with Divide and Conquer

$$A = \begin{bmatrix} a_{11} & a_{12} & | & a_{13} & a_{14} \\ a_{21} & a_{22} & | & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & | & a_{33} & a_{34} \\ a_{41} & a_{42} & | & a_{43} & a_{44} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & b_{12} & | & b_{13} & b_{14} \\ b_{21} & b_{22} & | & b_{23} & b_{24} \\ \hline b_{31} & b_{32} & | & b_{33} & b_{34} \\ b_{41} & b_{42} & | & b_{43} & b_{44} \end{bmatrix}$$

4x4

$$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$$

$$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$$

$$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$$

$$C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$$

$$f(n) = \begin{cases} 1 & n \leq 2 \\ 8T(n/2) + n^2 & n > 2 \end{cases}$$

MM(A, B, n)

{ if ($n \leq 2$)

$$C_{11} = a_{11} * b_{11} + a_{12} * b_{21}$$

$$C_{12} = a_{11} * b_{12} + a_{12} * b_{22}$$

$$C_{21} = a_{21} * b_{11} + a_{22} * b_{21}$$

$$C_{22} = a_{21} * b_{12} + a_{22} * b_{22}$$

else

{ Divide A & B of $n \times n$ into 4 submatrices
each of order $n/2 \times n/2$

$$MM(A_{11}, B_{11}, n/2) + MM(A_{12}, B_{21}, n/2)$$

$$MM(A_{11}, B_{12}, n/2) + MM(A_{12}, B_{22}, n/2)$$

$$MM(A_{21}, B_{11}, n/2) + MM(A_{22}, B_{21}, n/2)$$

$$MM(A_{21}, B_{12}, n/2) + MM(A_{22}, B_{22}, n/2)$$

Matrix Multiplication with Divide and Conquer

- Volker Strassen first published this algorithm in 1969 and thereby proved that the n^3 general matrix multiplication algorithm was not optimal. The Strassen algorithm's publication resulted in more research about matrix multiplication that led to both asymptotically lower bounds and improved computational upper bounds.



Strassens Matrix Multiplication

Input: $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \in \mathbb{R}^{n \times n}$

```

1: if  $n=1$  then
2:    $C = A \cdot B$ 
3: else
4:    $M_1 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$ 
5:    $M_2 = (A_{21} + A_{22}) \cdot B_{11}$ 
6:    $M_3 = A_{11} \cdot (B_{12} - B_{22})$ 
7:    $M_4 = A_{22} \cdot (B_{21} - B_{11})$ 
8:    $M_5 = (A_{11} + A_{12}) \cdot B_{22}$ 
9:    $M_6 = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$ 
10:   $M_7 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$ 
11:   $C_{11} = M_1 + M_4 - M_5 + M_7$ 
12:   $C_{12} = M_3 + M_5$ 
13:   $C_{21} = M_2 + M_4$ 
14:   $C_{22} = M_1 - M_2 + M_3 + M_6$ 

```

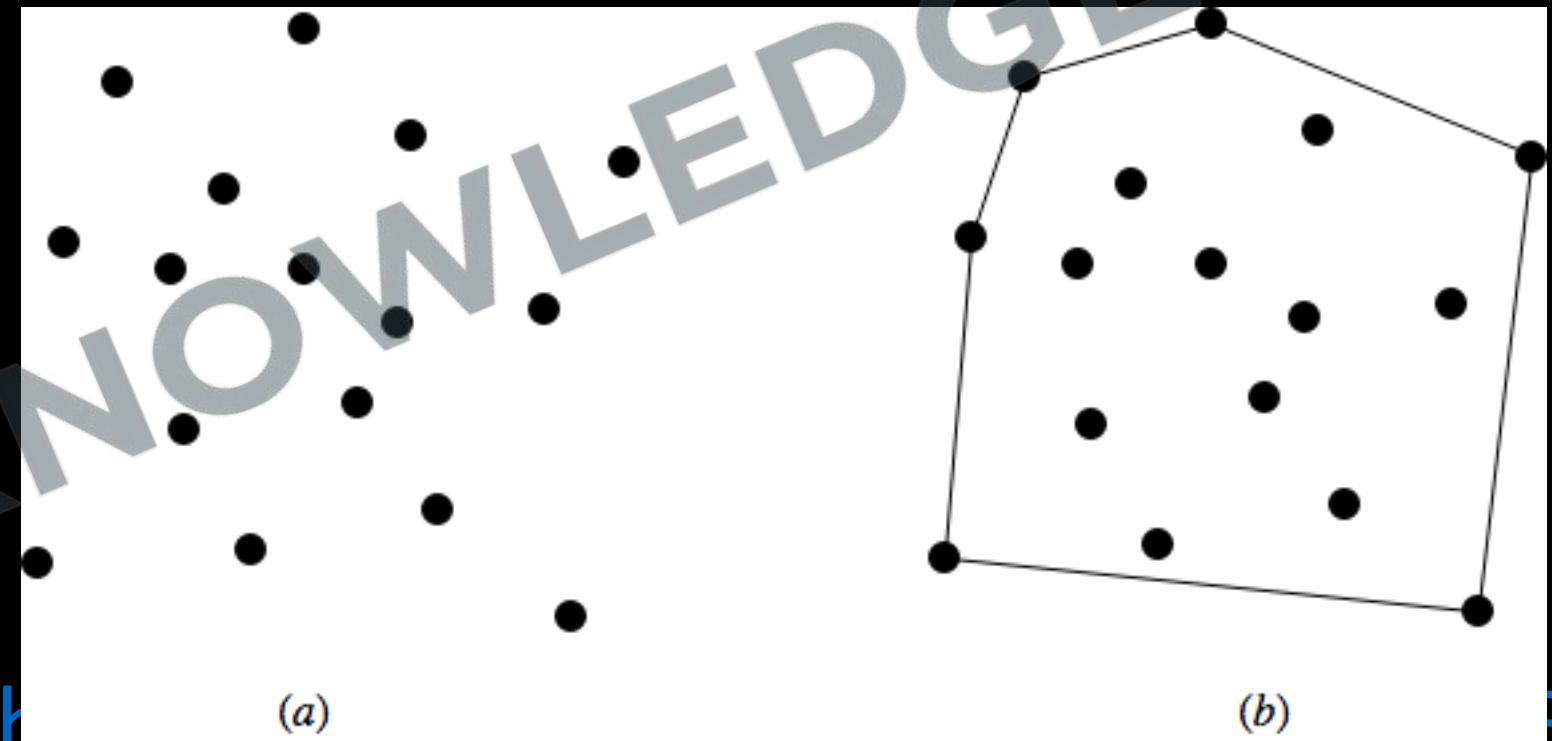
Output: $A \cdot B = C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \in \mathbb{R}^{n \times n}$

$$T(n) = \begin{cases} 1 & n \leq L \\ 7T(n/2) + n^L & n > L \end{cases}$$

$$\mathcal{O}(n^{\log_2 7}) = \mathcal{O}(n^{\log_2 7}) \\ = \mathcal{O}(n^{2.8})$$

Convex Hull

- The convex hull problem is a classic algorithmic problem in the field of computational geometry. The goal is to find the smallest convex polygon that encloses a set of points in a plane. In simple terms, if you imagine a set of nails hammered into a board and you wrap a rubber band around all the nails, the rubber band would outline the convex hull.
- **Convex Hull:** The smallest convex polygon formed by a set of points such that no point from the set lies outside the polygon.



- The convex hull is a fundamental structure in computational geometry, with applications in pattern recognition, image processing, GIS (Geographic Information Systems), and in solving other geometric problems.

Applications:

- Pathfinding and motion planning problems.
- Collision detection in physical simulations and computer games.
- Determining the boundary of an object in machine learning and computer vision.
- Clustering analysis in data mining.
- Supporting GIS operations like creating the boundary for geographical datasets.

GRAHAM-SCAN(Q)

{

Let p_0 be the point in Q with the minimum y-coordinate, or the leftmost such point in case of a tie.

Let $\langle p_1, p_2, \dots, p_m \rangle$ be the remaining points in Q, sorted by polar angle in counter clockwise order around p_0 .

Top[S] $\leftarrow 0$

PUSH(p_0 , S)

PUSH(p_1 , S)

PUSH(p_2 , S)

for i $\leftarrow 3$ to m

do while the angle formed by points NEXT-TO-TOP(S), Top(S), and p_i makes a non-left turn

POP(S)

PUSH(p_i , S)

Return S

}

Binary Search

- Binary search is a classic searching algorithm used to find the position of a target value within a sorted array. It is much more efficient than a linear search, offering $O(\log n)$ time complexity, where n is the number of elements in the array.

- Binary search locates a target value within a sorted array using a divide-and-conquer approach:
 - **Initialization**: Set low and high pointers at the array's start and end.
 - **Middle Index**: Calculate the middle of low and high. Use $\text{low} + (\text{high} - \text{low}) / 2$ to prevent overflow.
 - **Comparison**: Check if the middle element is the target. If yes, **return** its index.
 - **Direction**: If the target is smaller, search the left side (high becomes $\text{middle} - 1$); if larger, search the right (low becomes $\text{middle} + 1$).
 - **Iteration**: Repeat until low exceeds high.
 - **Outcome**: Return the target's index or indicate it's not found.

Ch-4

Greedy Methods

with Examples Such as Optimal
Reliability Allocation, Knapsack,

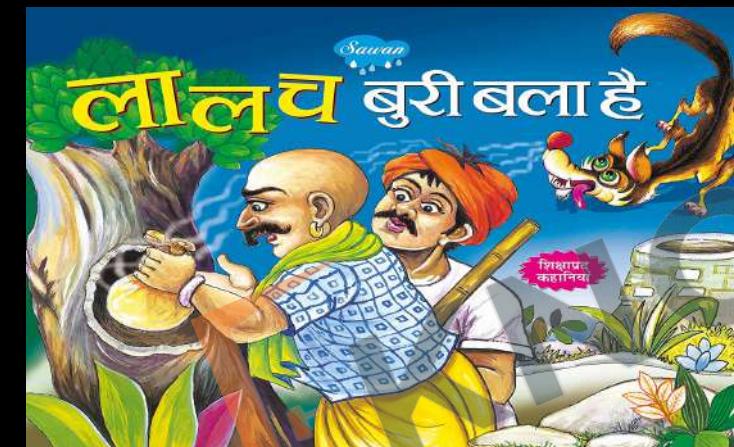
Huffman algorithm

<http://www.knowledgegate.in/GATE>

4K
ULTRAHD



हिंदी कहानी



लालच बुरी बला है,
अगर बुरे काम के लिए किया गया हो तो
<http://www.knowledgedgegate.in/GATE>

Greedy Algorithm

- A **greedy algorithm** is a problem solving approach like Subtract and conquer, divide and conquer and dynamic programming, which is used for solving optimality problem(one Solution), out of all feasible solution.
- Minimum Spanning Tree
- Single source shortest path
- Huffman Coding
- Knapsack Problem
- Optimal Merge Pattern

Greedy Algorithm

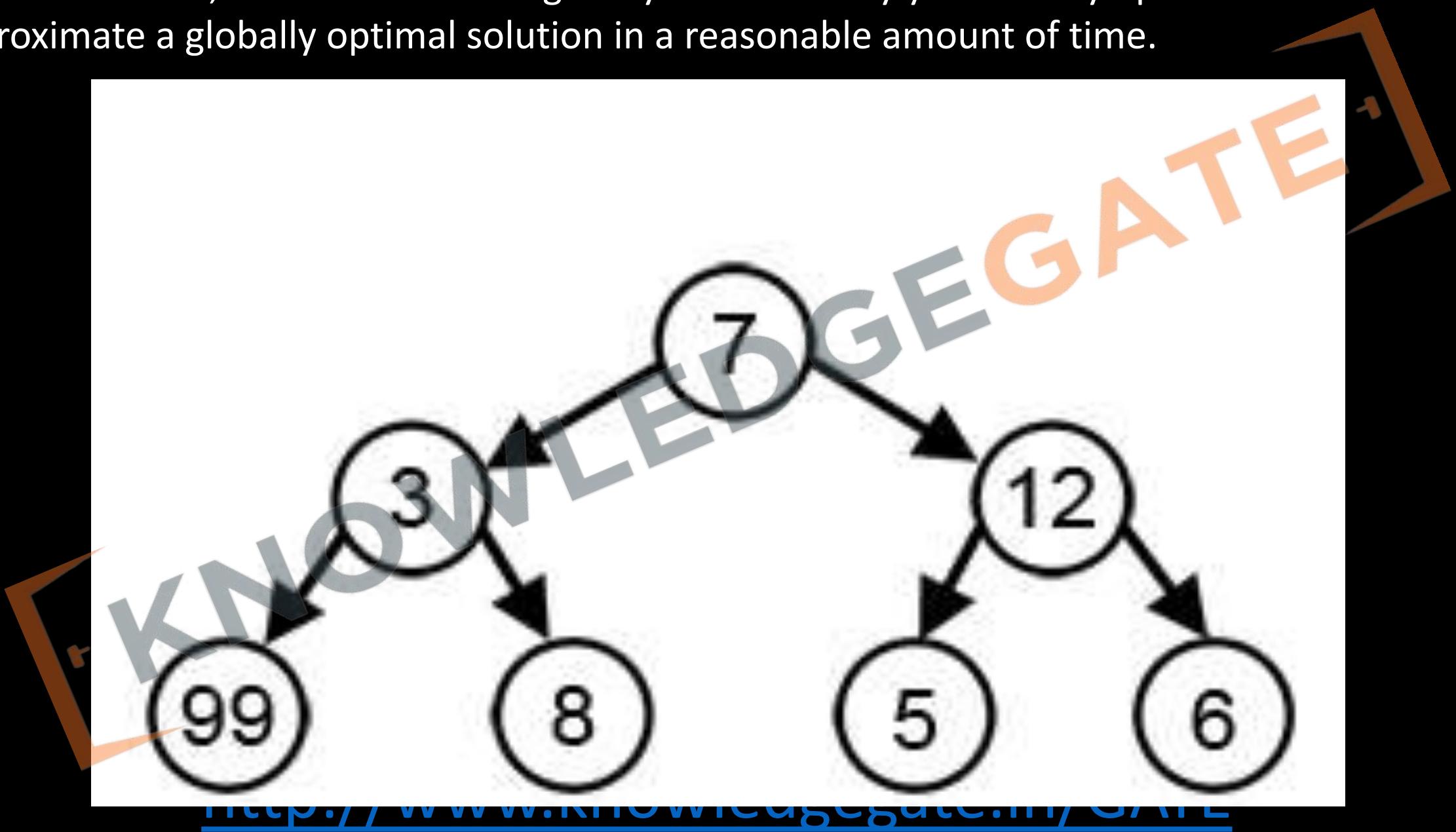
- A **greedy algorithm** is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum.

Bank Officer

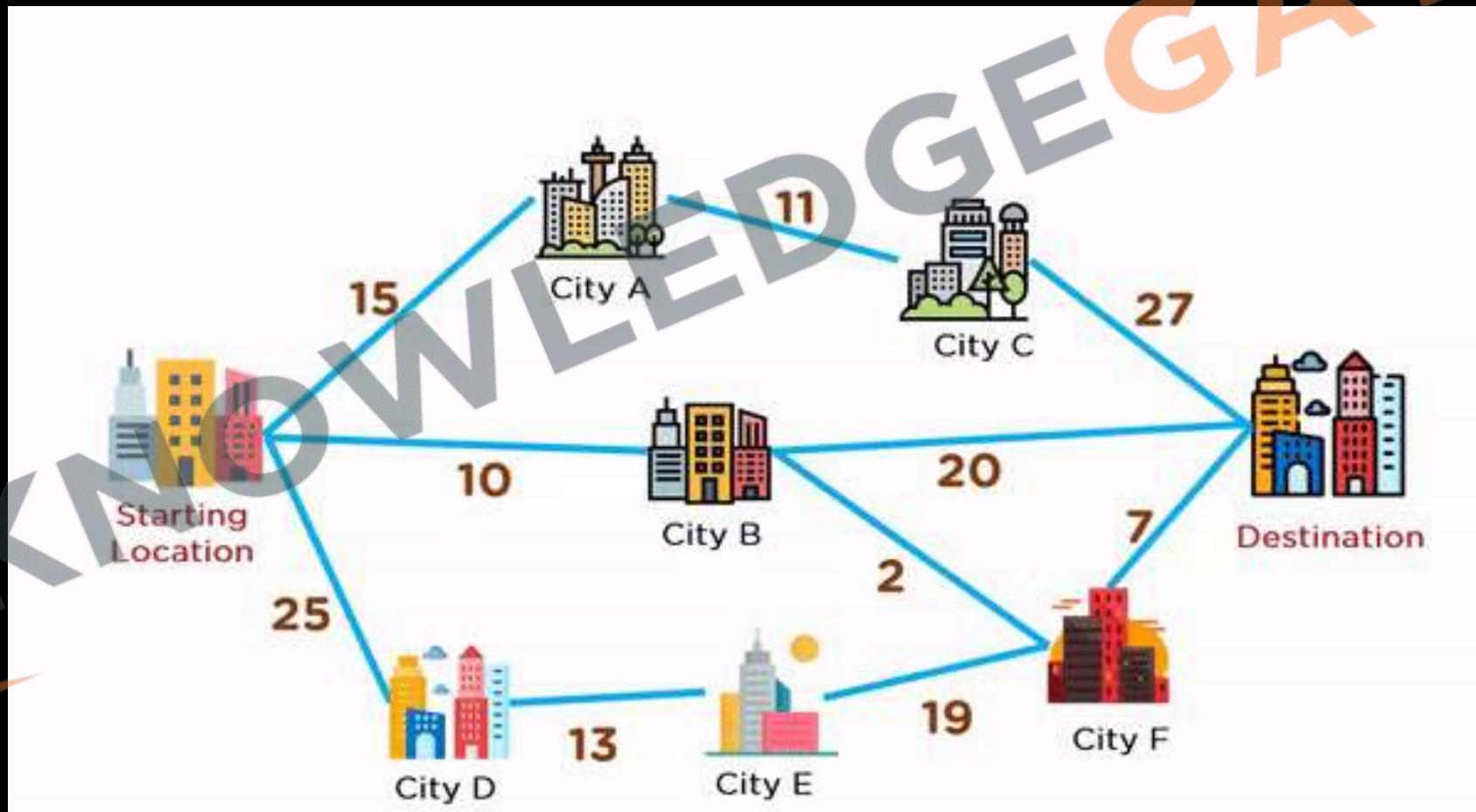


IAS Officer

- **Will greedy always work?** - In many problems, a greedy strategy does not usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.



- For example, a greedy strategy for the traveling salesman problem (which is of a high computational complexity) is the following heuristic:
- "At each step of the journey, visit the nearest unvisited city." This heuristic does not intend to find a best solution, but it terminates in a reasonable number of steps; finding an optimal solution to such a complex problem typically requires unreasonably many steps.



- We can make whatever choice seems best at the moment and then solve the subproblems that arise later.



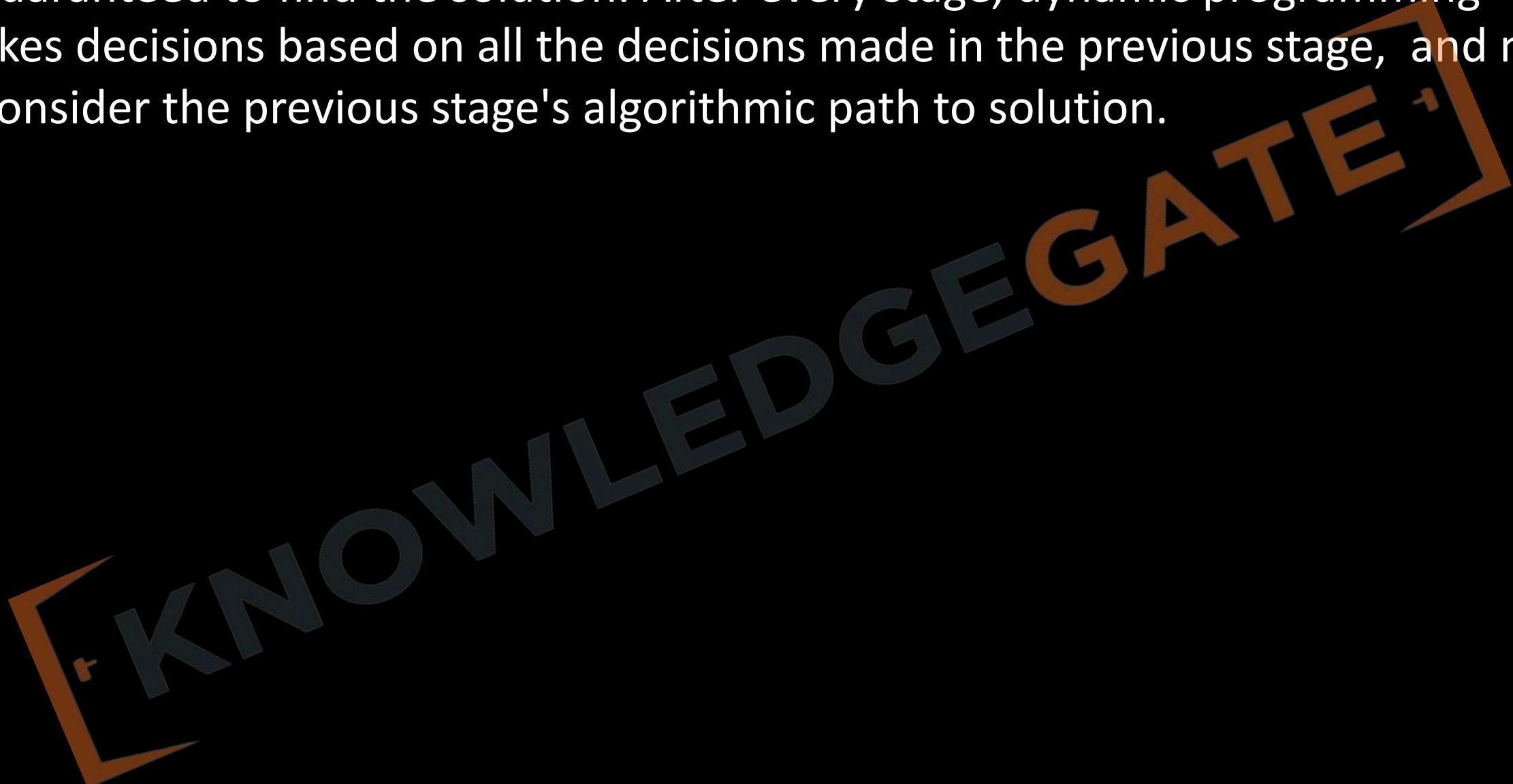
<http://www.knowledgegate.in/GATE>

- It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. In other words, a greedy algorithm never reconsiders its choices.



इस मेट्रो में जाऊँ या अगली में?
<http://www.knowledgigate.in/GATE>

- This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution. After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution.



<http://www.knowledgegate.in/GATE>

Optimization Problem

- An optimization problem is a type of problem that seeks to find the best solution from all feasible solutions. Here's a simple breakdown:
- **Optimization Problems:**
 - **Objective:** Minimize or maximize some quantity (like cost, profit, distance).
 - **Constraints:** Set of restrictions or conditions that the solution must satisfy.
 - **Feasible Solution:** A solution that meets all constraints.
 - **Optimal Solution:** A feasible solution that yields the best value of the objective function.

Activity Selection Problem

- The Activity Selection Problem is a classic problem and often used to illustrate the concept of greedy algorithms. Here's a simplified explanation:
- You have a set of activities, each with a start and an end time. Each activity has s_i a start time, and f_i a finish time. If activity i is selected, the resource is occupied in the intervals (s_i, f_i) . We say i and j are compatible activities if their start and finish time does not overlap i.e., i and j compatible if $s_i \geq f_j$ and $s_j \geq f_i$
- You need to select the maximum number of activities that don't overlap in time. The **Goal is to Maximize** the number of activities selected.

- **Approach:**
 - **Sort Activities:** First, sort all activities by their finish time.
 - **Select First Activity:** Choose the activity that finishes first.
 - **Iterate and Select:** For each subsequent activity, if its start time is after or at the finish time of the previously selected activity, select it.
- Activities (start time, end time): (1, 3), (2, 4), (3, 5), (5, 7)
- Sorted by end time: (1, 3), (3, 5), (2, 4), (5, 7)
- Selected Activities: (1, 3), (3, 5), (5, 7)

- At each step, you make the choice that seems best at the moment (choosing the activity that ends earliest). This local optimization leads to a globally optimal solution.
- **Use Cases:**
 - Scheduling tasks in a single resource environment (like a single meeting room).
 - Allocating time slots for interviews or exams where no overlap is allowed.
- **Complexity:**
 - If the activities are not sorted, the main complexity is in the sorting step, which is typically $O(n \log n)$ for n activities.
 - The selection process is $O(n)$, as it involves iterating through the list once.
 - This problem is an excellent example to teach students about greedy algorithms, showing how a locally optimal choice can lead to a globally optimal solution in certain scenarios.

ii. Pseudo code for iterative greedy algorithm :

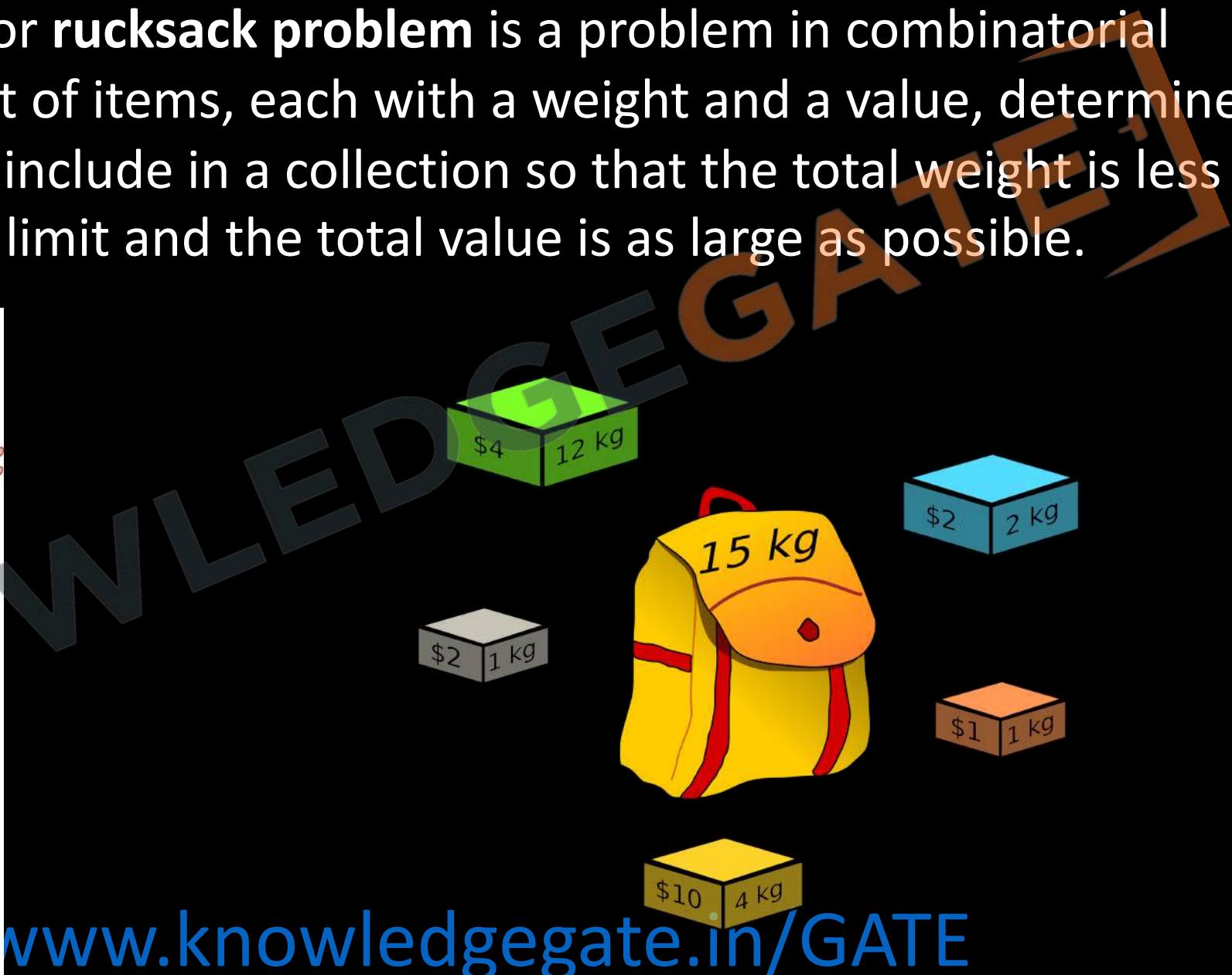
G_A_S (s, f)

1. $n \leftarrow \text{length}[s]$
2. $A \leftarrow [a_1]$
3. $i \leftarrow 1$
4. $m \leftarrow 2 \text{ to } n$
5. do if $s_m \geq f_i$
6. then $A \leftarrow A \cup \{a_m\}$
7. $i \leftarrow m$
8. return A

Sorted activities	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
Starting time	2	1	3	0	5	3	5	6	8	8	12
Finish time	3	4	5	6	7	8	9	10	11	12	14

Knap Sack Problem

- The **knapsack problem** or **rucksack problem** is a problem in combinatorial optimization: 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.



- Knap Sack problem can be studied in two versions fractional Knap Sack and 0/1 Knap Sack, here we will be discussing Fractional Knap Sack and then 0/1 Knap Sack will be solved in Dynamic Programming.



Q Consider the weights and values of items listed below. Note that there is only one unit of each item.

Greedy by Profit

Object	O ₁	O ₂	O ₃
Profit	25	24	15
Weight	18	15	10

Object	O ₁	O ₂	O ₃
Profit	25	24	15
Weight	18	15	10
Solution	http://www.knowledgegate.in/GATE		

Q Consider the weights and values of items listed below. Note that there is only one unit of each item.

Greedy by Weight

Object	O ₁	O ₂	O ₃
Profit	25	24	15
Weight	18	15	10

Object	O ₁	O ₂	O ₃
Profit	25	24	15
Weight	18	15	10
Solution	http://www.knowledgegate.in/GATE		

Q Consider the weights and values of items listed below. Note that there is only one unit of each item.

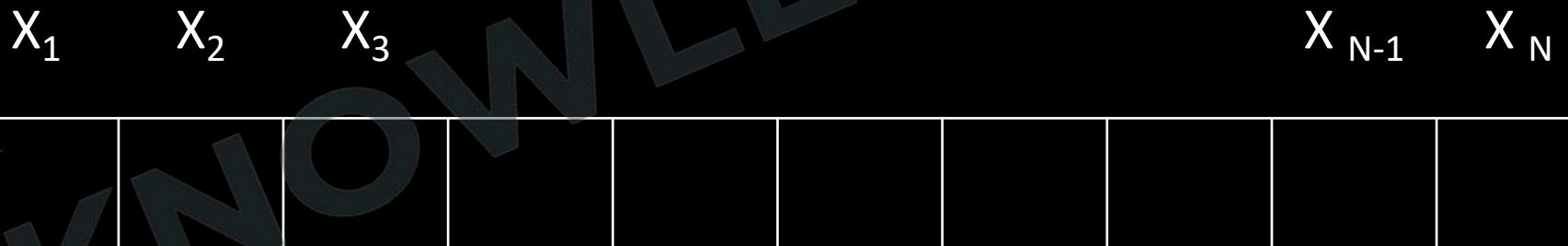
Greedy by Profit/Weight

Object	O ₁	O ₂	O ₃
Profit	25	24	15
Weight	18	15	10

Object	O ₁	O ₂	O ₃
Profit	25	24	15
Weight	18	15	10
Profit/Weight	1.38	1.6	1.5
Solution			

Problem Definition

- More formally there are n number of objects $O_1, O_2, O_3 \dots O_n$, each object has a weight associated with its W_i , and a profit associated with it P_i , we can take x_i fraction of the object ranging from $0 \leq x_i \leq 1$
- Weight Condition $\sum_{i=1}^n W_i \cdot X_i \leq M$
- Profit $\sum_{i=1}^n P_i \cdot X_i$



- The knapsack problem has been studied for more than a century, with early works dating as far back as 1897. The name "knapsack problem" dates back to the early works of mathematician Tobias Dantzig (1884–1956).



Conclusion

- It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.
- The problem often arises in resource allocation where there are financial constraints and is studied in fields such as
 - Combinatorics
 - Computer science
 - Complexity theory
 - Cryptography
 - Applied mathematics.

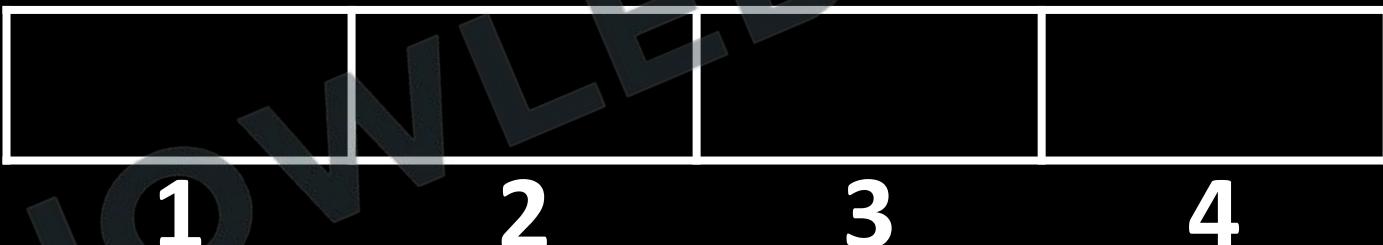
Job sequencing with Deadline

- We are given n-jobs, where each job is associated with a deadline D_i and a profit P_i if the job is finished before the deadline.
- We have single CPU with Non-Primitive Scheduling.
- With each job we assume arrival time is 0, burst time of each job requirement is 1.
- Select a Subset of ‘n’ jobs, such that, the jobs in the subset can be completed within the deadline and generate Max profit.

Q if we have for task T_1, T_2, T_3, T_4 , having Deadline $D_1 = 2, D_2 = 1, D_3 = 2, D_4 = 1$, and profit $P_1 = 100, P_2 = 10, P_3 = 27, P_4 = 15$, find the maximum profit possible?

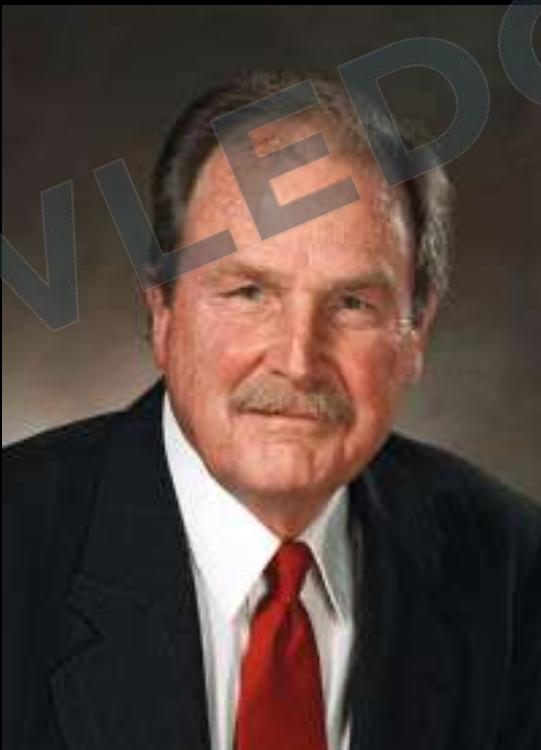
Task	T_1	T_2	T_3	T_4
Profit	100	10	27	15
Deadline	2	1	2	1

Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7
Profit	35	30	25	20	15	12	5
Deadline	3	4	4	2	3	1	2

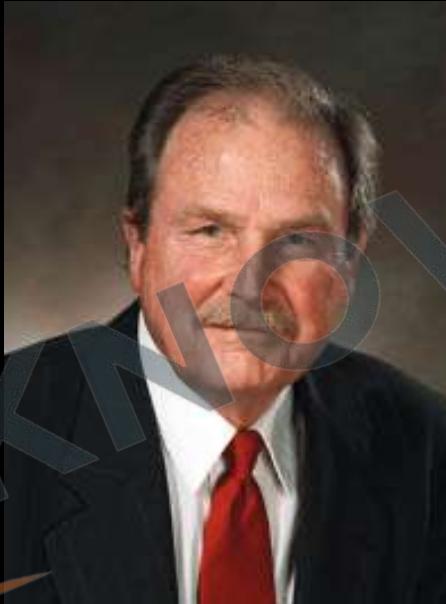


Huffman coding

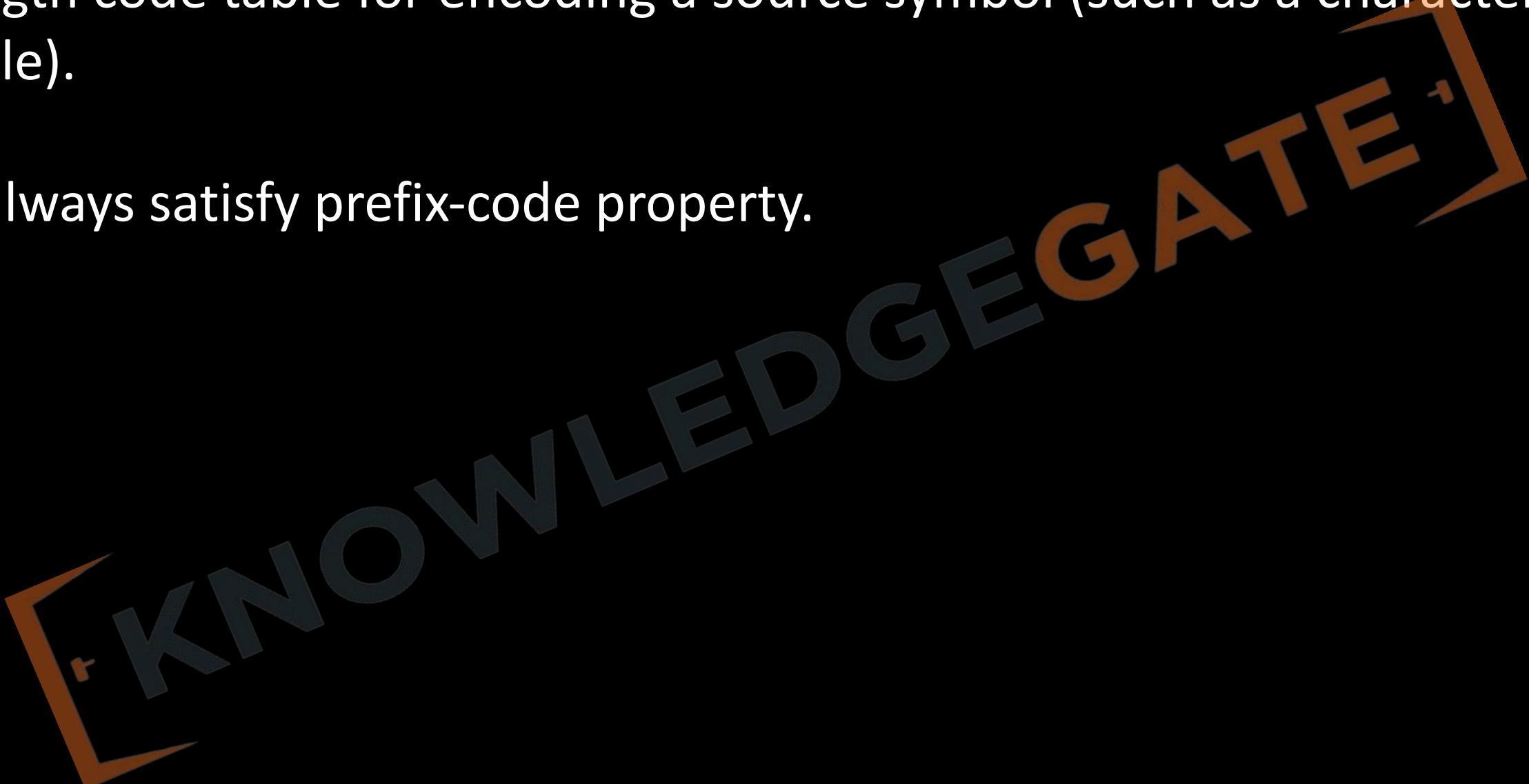
- In computer science and information theory, a **Huffman code** is a particular type of optimal prefix code that is commonly used for lossless data compression.
- The process of finding or using such a code proceeds by means of **Huffman coding**, an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".



- In 1951, David A. Huffman and his MIT information theory classmates were given the choice of a term paper or a final exam. The professor, Robert M. Fano, assigned a term paper on the problem of finding the most efficient binary code.
- Huffman, unable to prove any codes were the most efficient, was about to give up and start studying for the final when he hit upon the idea of using a frequency-sorted binary tree and quickly proved this method the most efficient.
- In doing so, Huffman outdid Fano, who had worked with Claude Shannon to develop a similar code. Building the tree from the bottom up guaranteed optimality, unlike the top-down approach of Shannon–Fano coding.



- The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file).
- It always satisfy prefix-code property.



<http://www.knowledgegate.in/GATE>

- The algorithm derives this table from the estimated probability or frequency of occurrence (*weight*) for each possible value of the source symbol.
- Huffman's method can be efficiently implemented, finding a code in time linear to the number of input weights if these weights are sorted.
- However, although optimal among methods encoding symbols separately, Huffman coding is not always optimal among all compression methods.

In alphabetical order

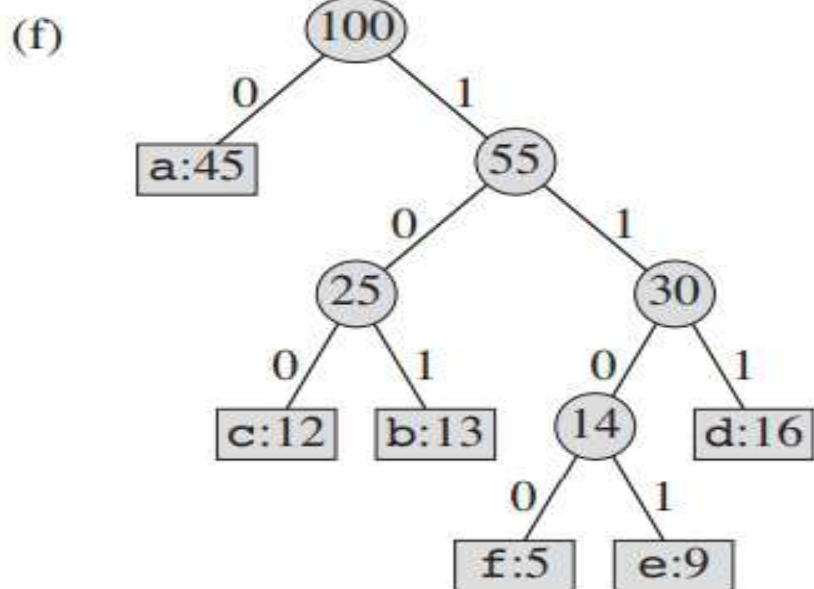
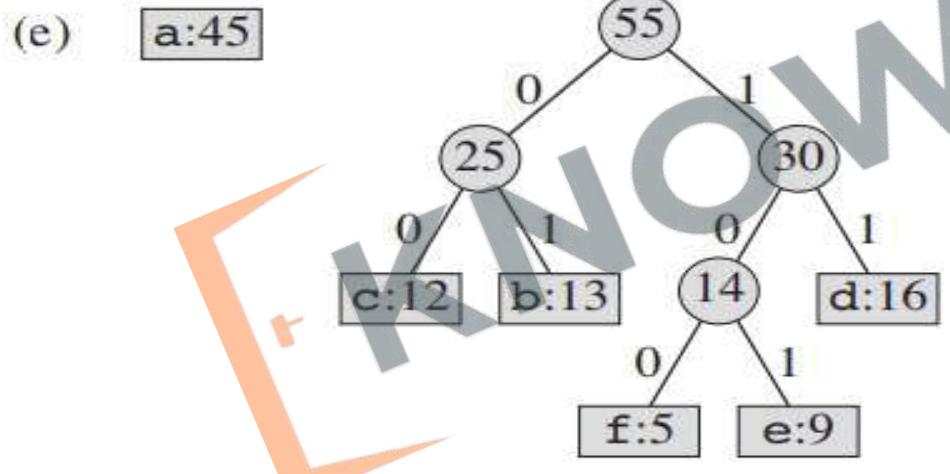
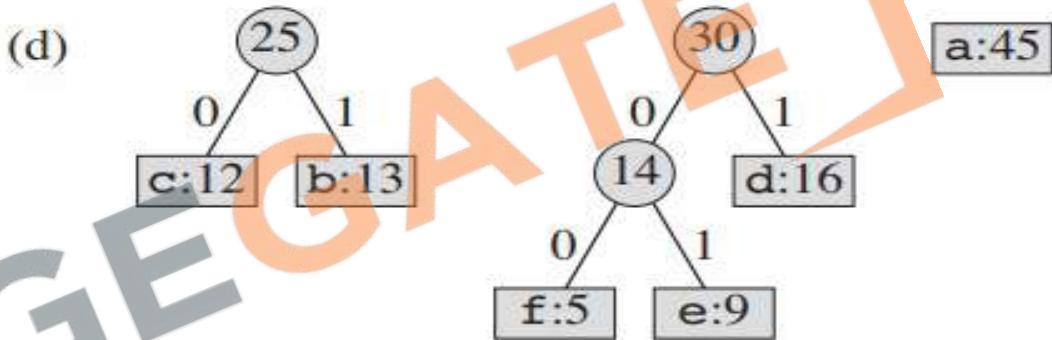
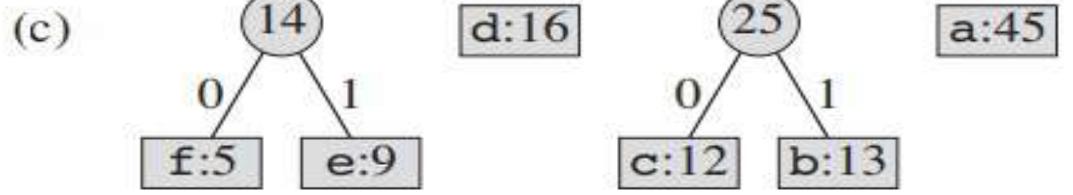
A	8.15 %	N	7.10 %
B	1.44 %	O	8.00 %
C	2.76 %	P	1.98 %
D	3.79 %	Q	0.12 %
E	13.11 %	R	6.83 %
F	2.92 %	S	6.10 %
G	1.99 %	T	10.47 %
H	5.26 %	U	2.46 %
I	6.35 %	V	0.92 %
J	0.13 %	W	1.54 %
K	0.42 %	X	0.17 %
L	3.39 %	Y	1.98 %
M	2.54 %	Z	0.08 %

KNOWLEDGE GATE

```
HUFFMAN( $C$ )
1   $n = |C|$ 
2   $Q = C$ 
3  for  $i = 1$  to  $n - 1$ 
4      allocate a new node  $z$ 
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8       $\text{INSERT}(Q, z)$ 
9  return  $\text{EXTRACT-MIN}(Q)$     // return the root of the tree
```

(a) **f:5** **e:9** **c:12** **b:13** **d:16** **a:45**

(b) **c:12** **b:13** **14** **0** **1** **f:5** **e:9** **d:16** **a:45**



Q Consider the following character with frequency and generate Huffman tree, find Huffman code for each character, find the number of bits required for a message of 100 characters?

Character	Frequency
M ₁	12
M ₂	4
M ₃	45
M ₄	17
M ₅	23

Ch-5

Minimum Spanning Trees

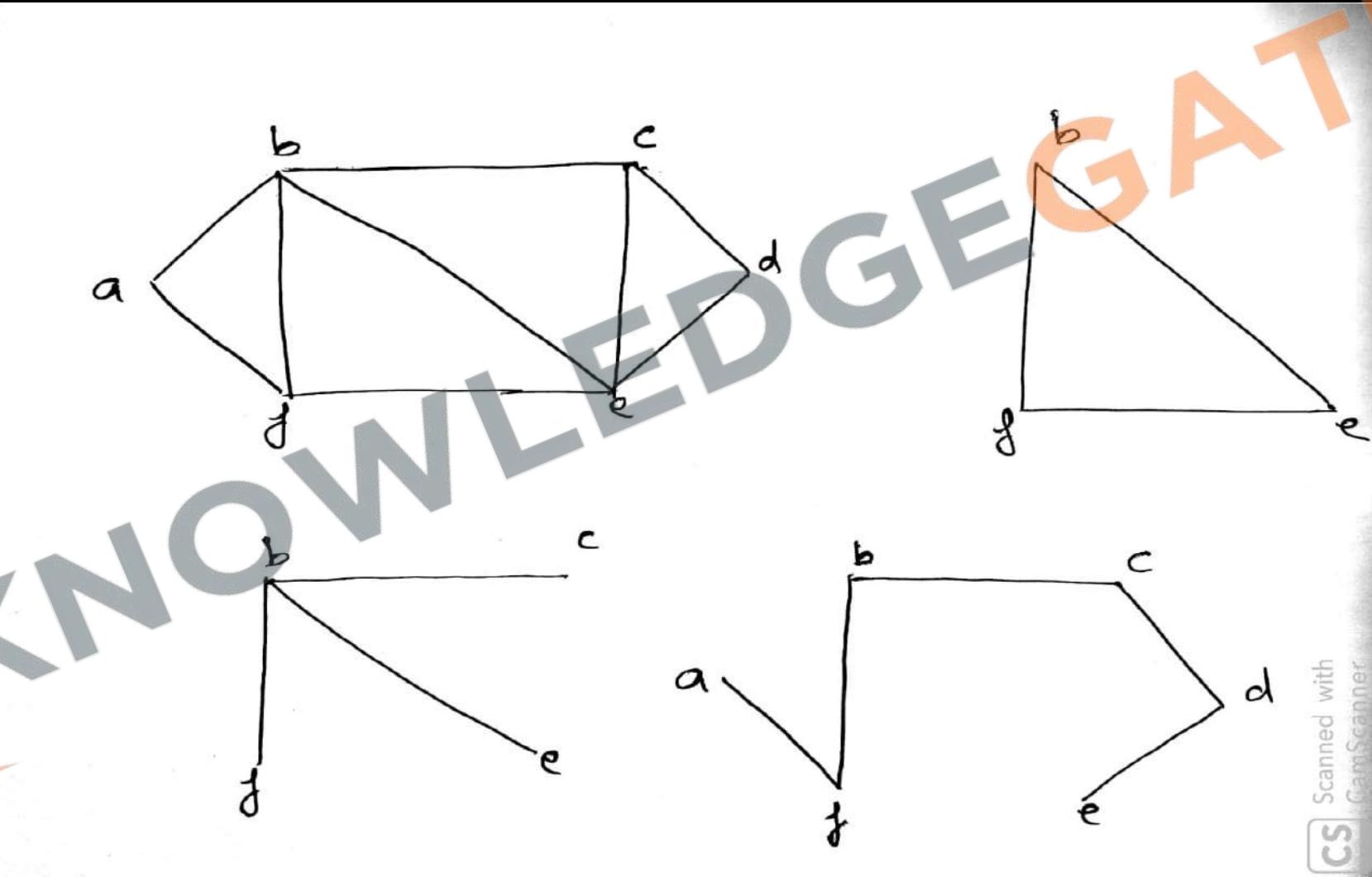
Prim's and Kruskal's

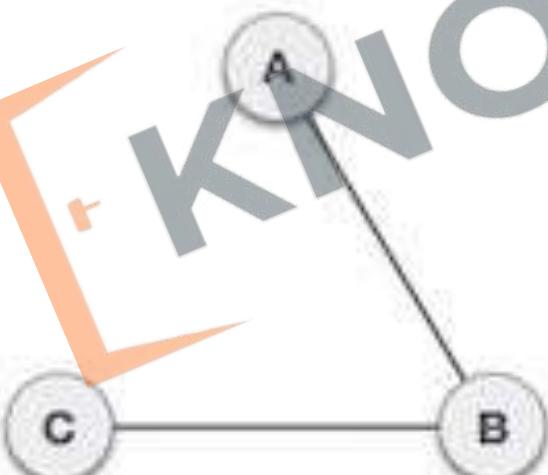
Algorithms

<http://www.knowledgegate.in/GATE>

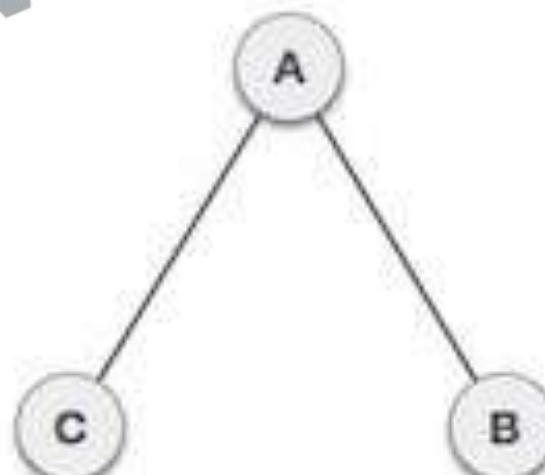
Spanning tree

- A tree T is said to be spanning tree of a connected graph G , if T is a subgraph of G and T contains all vertices of G .

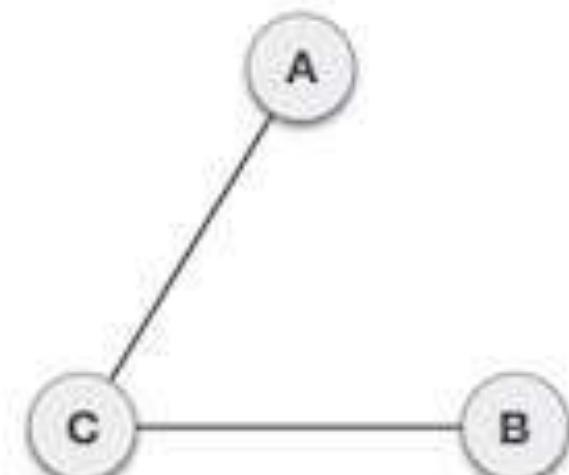




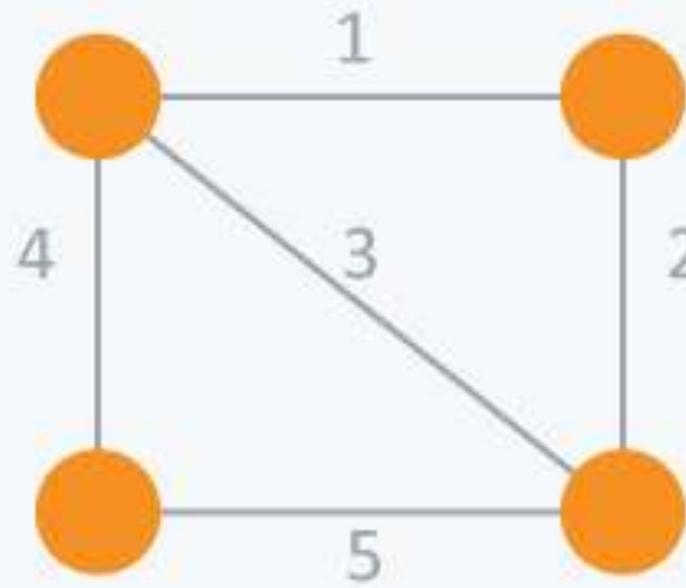
Graph G



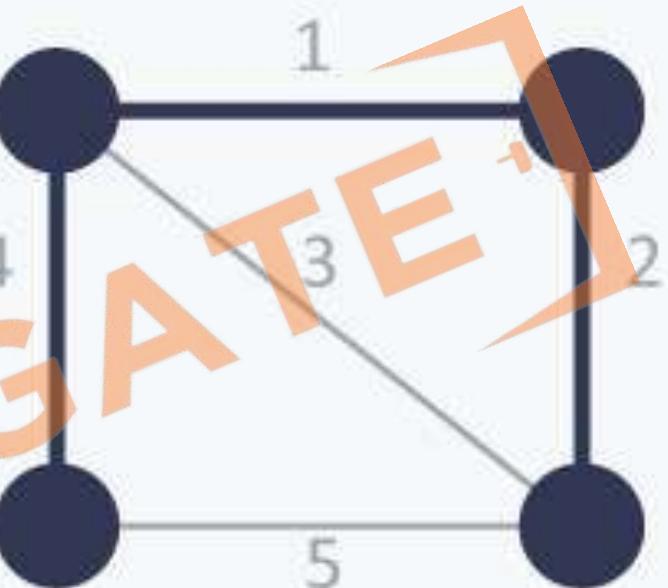
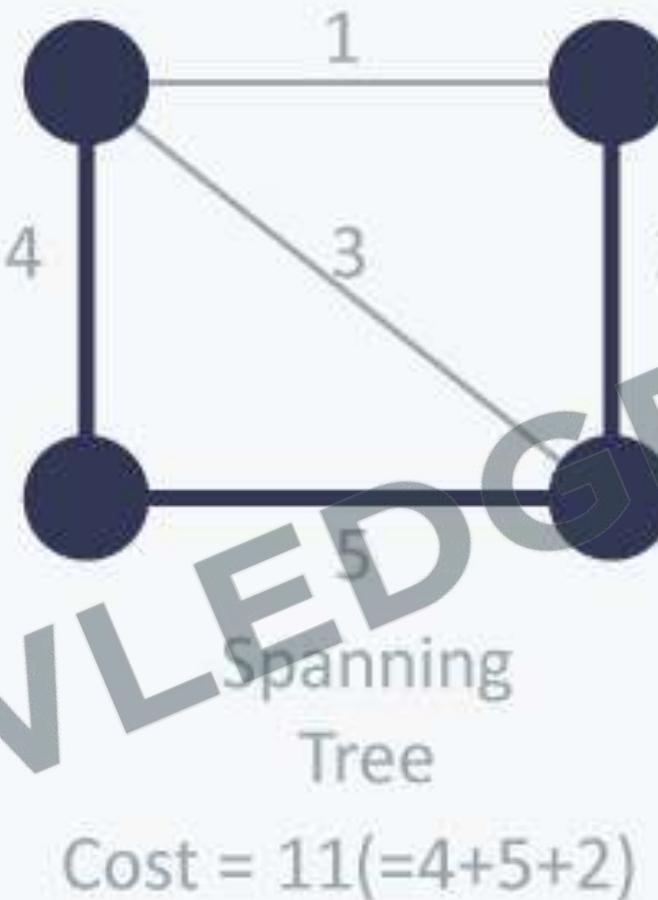
Spanning Trees



KNOWLEDGE GATE

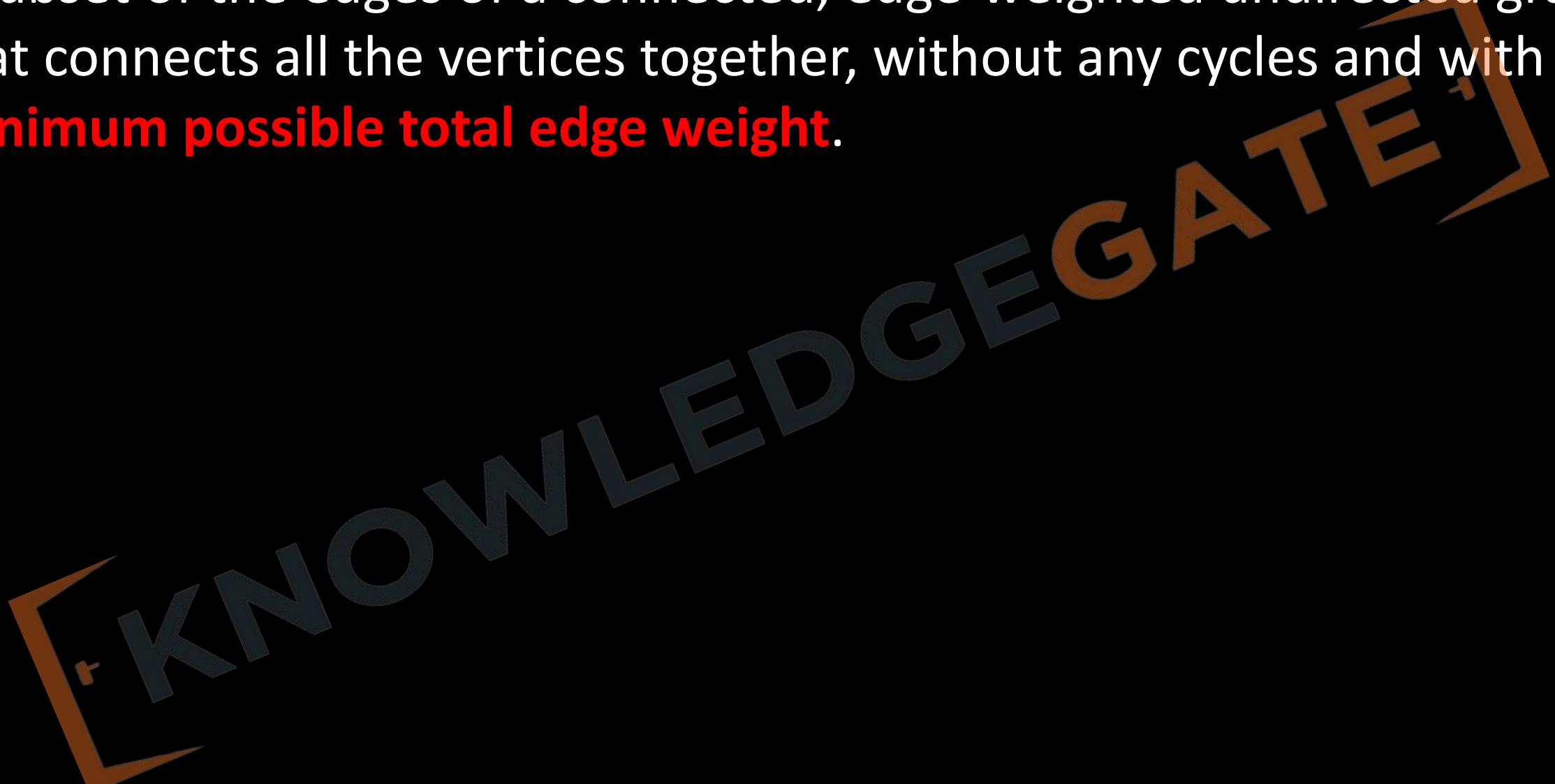


Undirected
Graph



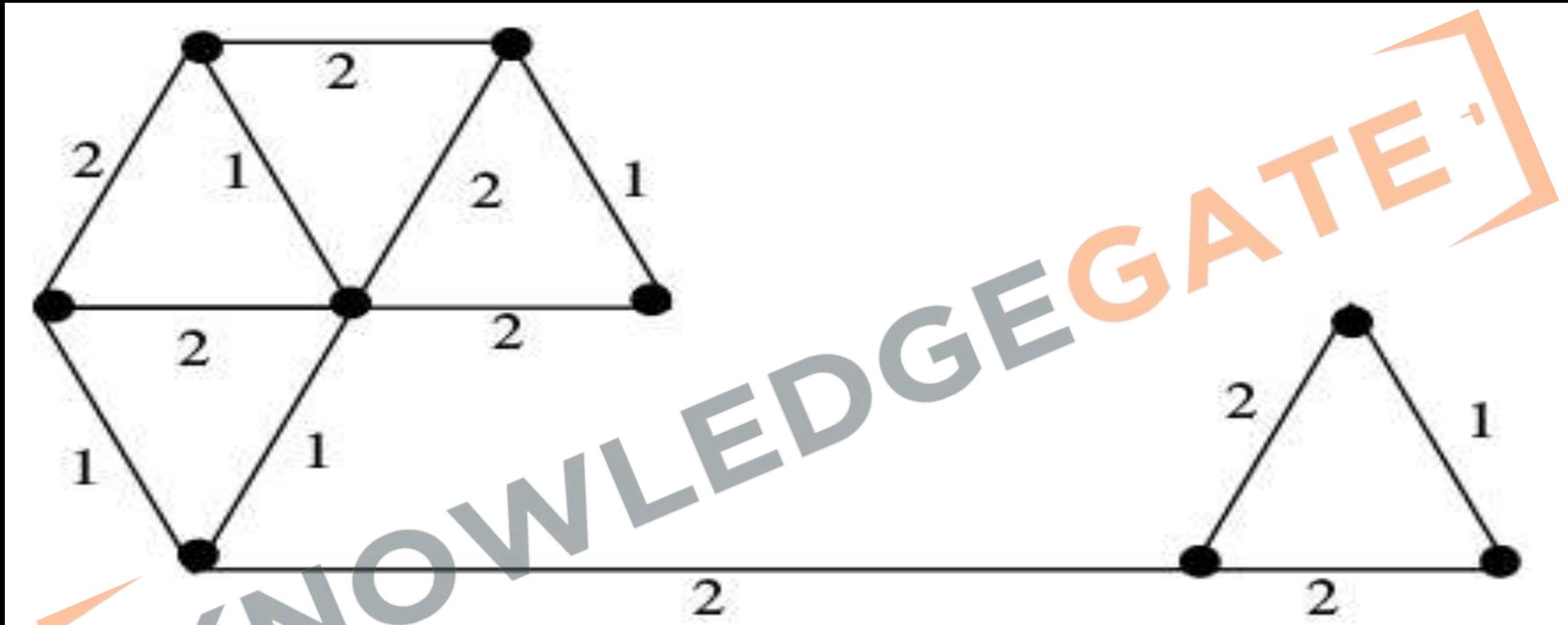
Minimum Spanning
Tree

- A **minimum spanning tree (MST)** or **minimum weight spanning tree** is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the **minimum possible total edge weight**.



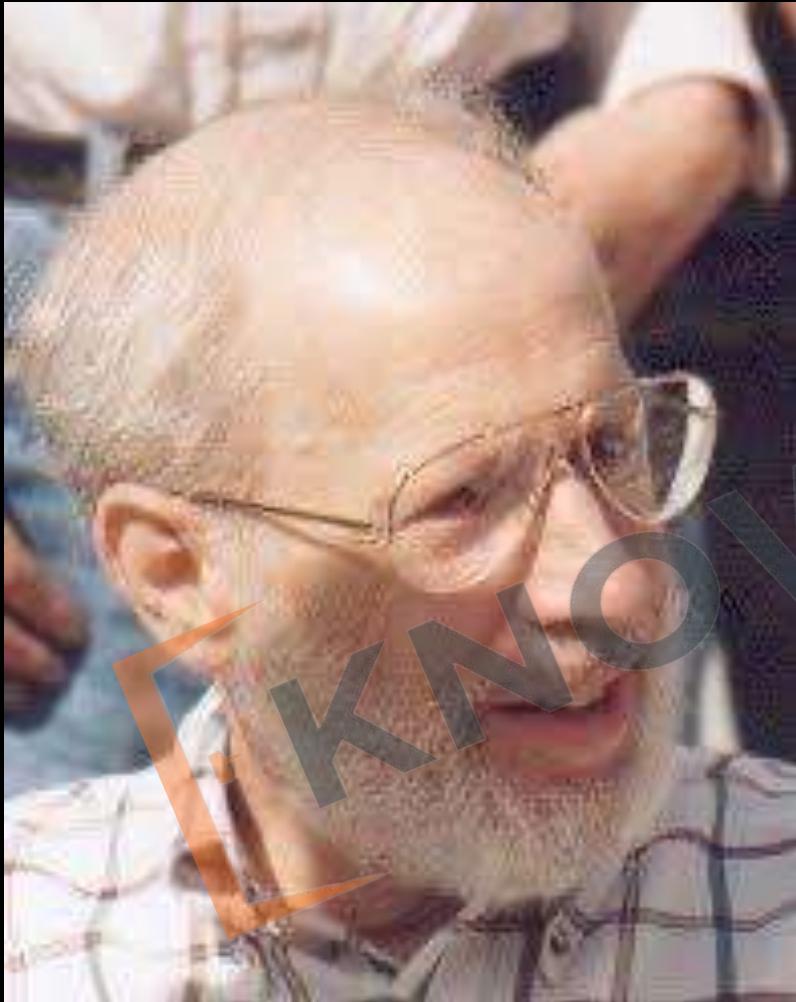
<http://www.knowledgegate.in/GATE>

- Minimum spanning tree (MST) can be more than one

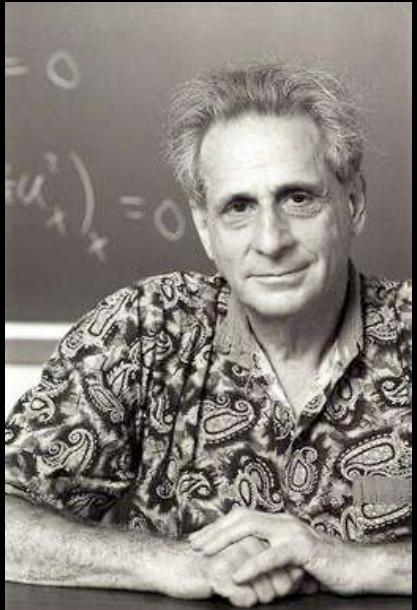


Kruskal Algorithm

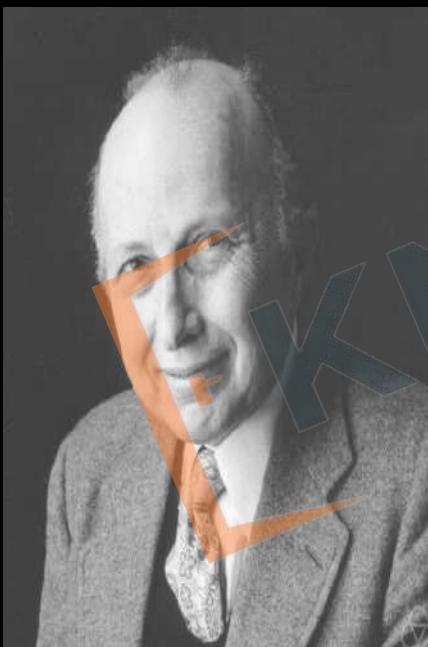
- Joseph Bernard Kruskal, Jr. was an American mathematician, statistician, computer scientist and psychometrician.



- Kruskal had two notable brothers



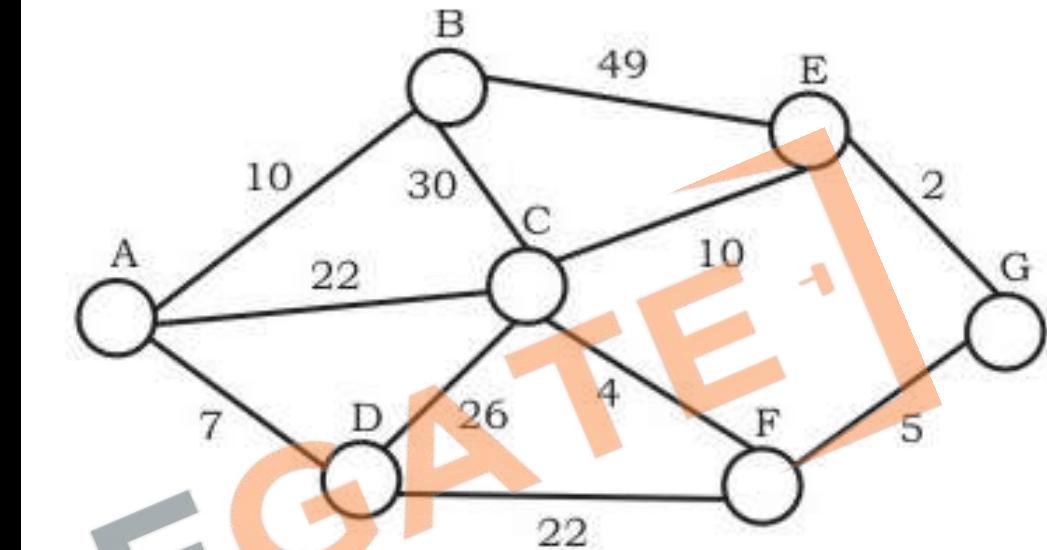
Martin David Kruskal (September 28, 1925 – December 26, 2006) was an American mathematician and physicist. He made fundamental contributions in many areas of mathematics and science, ranging from plasma physics to general relativity and from nonlinear analysis to asymptotic analysis. His most celebrated contribution was in the theory of solitons.



William Henry Kruskal (October 10, 1919 – April 21, 2005) was an American mathematician and statistician. He is best known for having formulated the Kruskal–Wallis one-way analysis of variance (together with W. Allen Wallis), a widely used nonparametric statistical method.

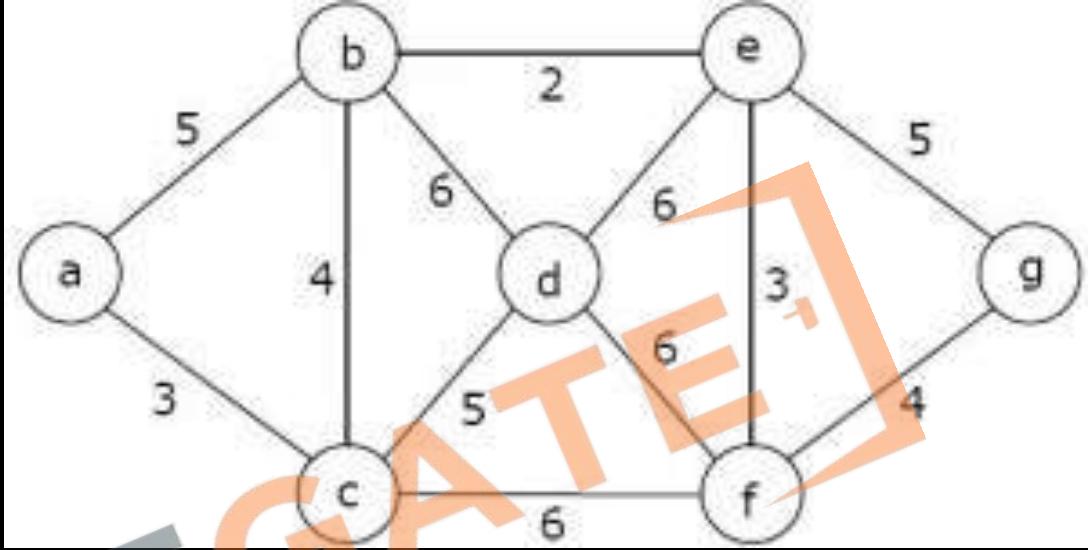
- Kruskal Algorithm Actual idea

- सबसे छोटा Edge 3ठाने का
- Cycle बने तो भगाने का
- जब तक graph connect
ना हो चलते जाने का



Kruskal

```
Minimum_Spanning_Tree (G, w)
{
    A  $\leftarrow \emptyset$ 
    For each vertex  $v \in V(G)$ 
    {
        do Make_Set(v)
    }
    Sort the edges of E into non-decreasing order by weight w
    for each edge  $(u, v) \in E$ , then in non-decreasing order by weights
    {
        if (Find_Set(u) != Find_Set(v))
        {
            A  $\leftarrow A \cup \{(u, v)\}$ 
            UNION (u, v)
        }
    }
    Return A
}
```



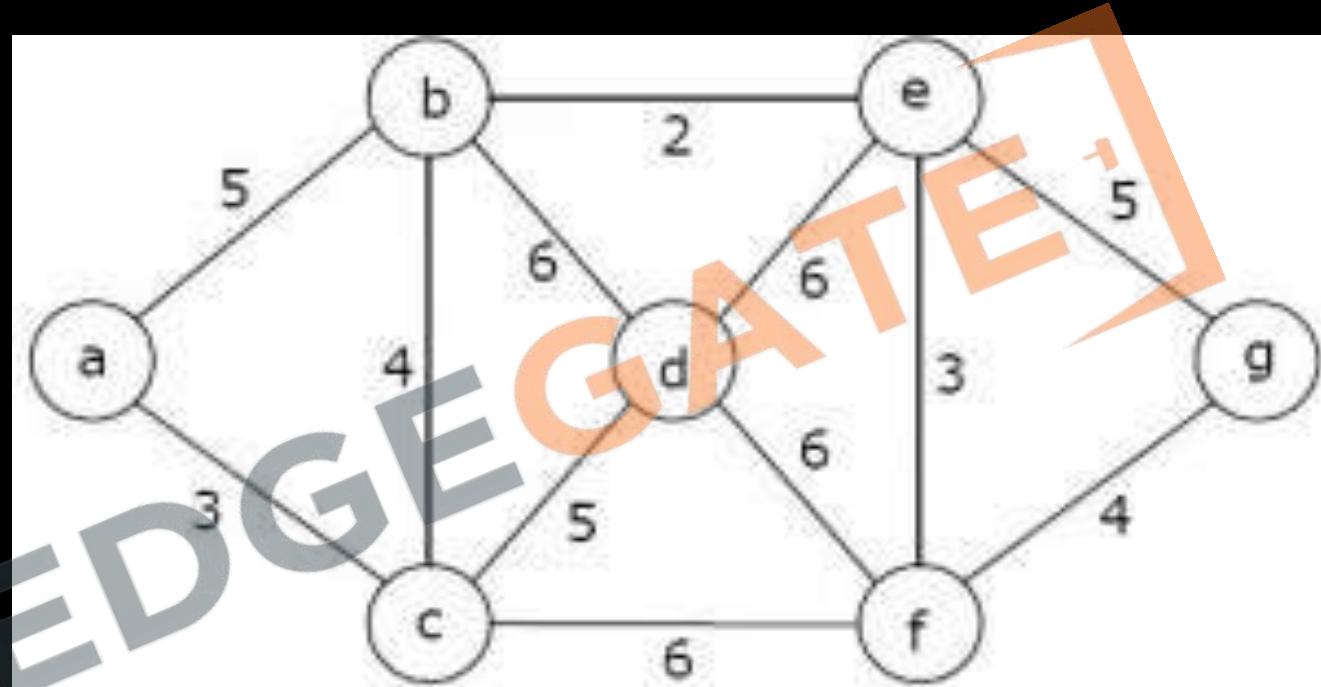
MST-KRUSKAL(G, w)

```
1   $A = \emptyset$ 
2  for each vertex  $v \in G.V$ 
3      MAKE-SET( $v$ )
4  sort the edges of  $G.E$  into nondecreasing order by weight  $w$ 
5  for each edge  $(u, v) \in G.E$ , taken in nondecreasing order by weight
6      if FIND-SET( $u$ )  $\neq$  FIND-SET( $v$ )
7           $A = A \cup \{(u, v)\}$ 
8          UNION( $u, v$ )
9  return  $A$ 
```

The running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on how we implement the disjoint-set data structure. We assume that we use the disjoint-set-forest implementation of Section 21.3 with the union-by-rank and path-compression heuristics, since it is the asymptotically fastest implementation known. Initializing the set A in line 1 takes $O(1)$ time, and the time to sort the edges in line 4 is $O(E \lg E)$. (We will account for the cost of the $|V|$ MAKE-SET operations in the for loop of lines 2–3 in a moment.) The for loop of lines 5–8 performs $O(E)$ FIND-SET and UNION operations on the disjoint-set forest. Along with the $|V|$ MAKE-SET operations, these take a total of $O((V + E) \alpha(V))$ time, where α is the very slowly growing function defined in Section 21.4. Because we assume that G is connected, we have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\alpha(V))$ time. Moreover, since $\alpha(|V|) = O(\lg V) = O(\lg E)$, the total running time of Kruskal's algorithm is $O(E \lg E)$. Observing that $|E| < |V|^2$, we have $\lg |E| = O(\lg V)$, and so we can restate the running time of Kruskal's algorithm as $O(E \lg V)$.

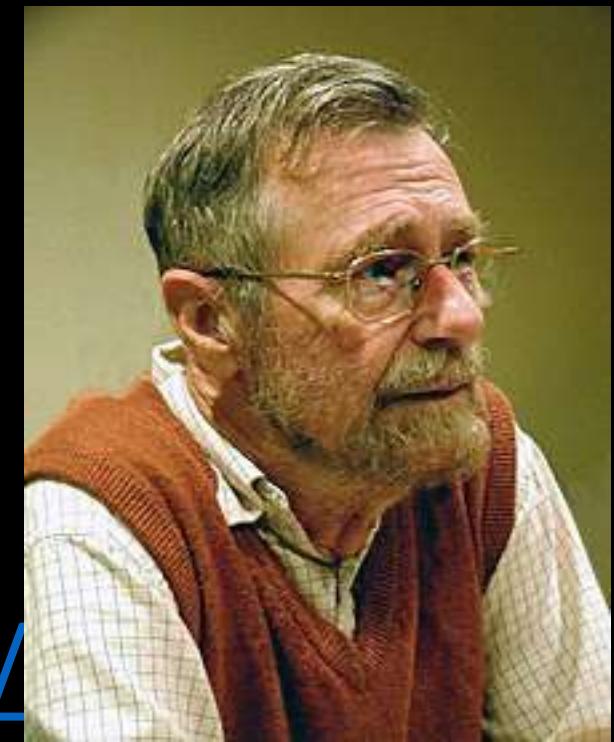
Q Consider the following graph: Which one of the following is NOT the sequence of edges added to the minimum spanning tree using Kruskal's algorithm?

- (A) (b, e),(e, f),(a, c),(b, c),(f, g),(c, d)
- (B) (b, e),(e, f),(a, c),(f, g),(b, c),(c, d)
- (C) (b, e),(a, c),(e, f),(b, c),(f, g),(c, d)
- (D) (b, e),(e, f),(b, c),(a, c),(f, g),(c, d)



Prim's Algorithm

- The algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník
- And later rediscovered and republished by computer scientists Robert C. Prim in 1957
- And Edsger W. Dijkstra in 1959.
- Therefore, it is also sometimes called the **Jarník's algorithm**, **Prim–Jarník algorithm**, **Prim–Dijkstra algorithm** or the **DJP algorithm**.



- Vojtěch Jarník (1897–1970) was a Czech mathematician who worked for many years as a professor and administrator at Charles University, and helped found the Czechoslovak Academy of Sciences. He is the namesake of Jarník's algorithm for minimum spanning trees.

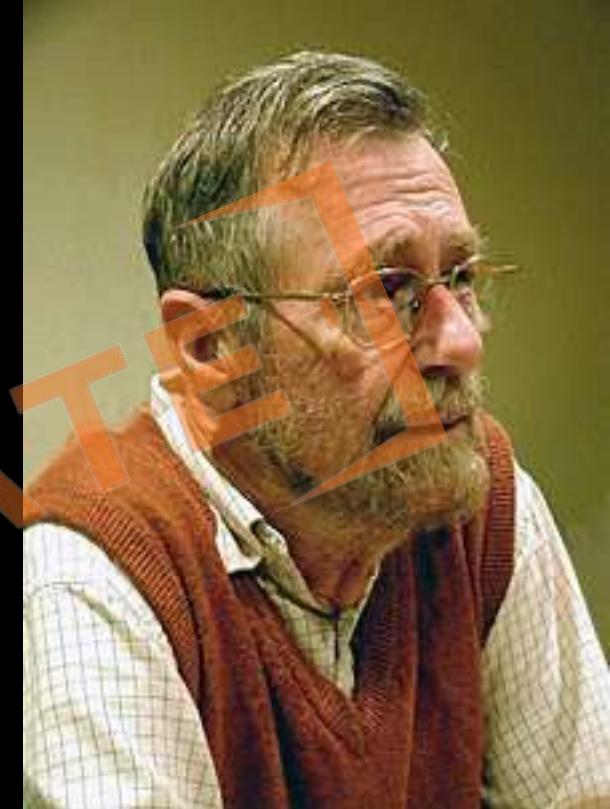


- **Robert Clay Prim** (born September 25, 1921 in Sweetwater, Texas) is an American mathematician and computer scientist.



<http://www.knowledgegate.in/GATE>

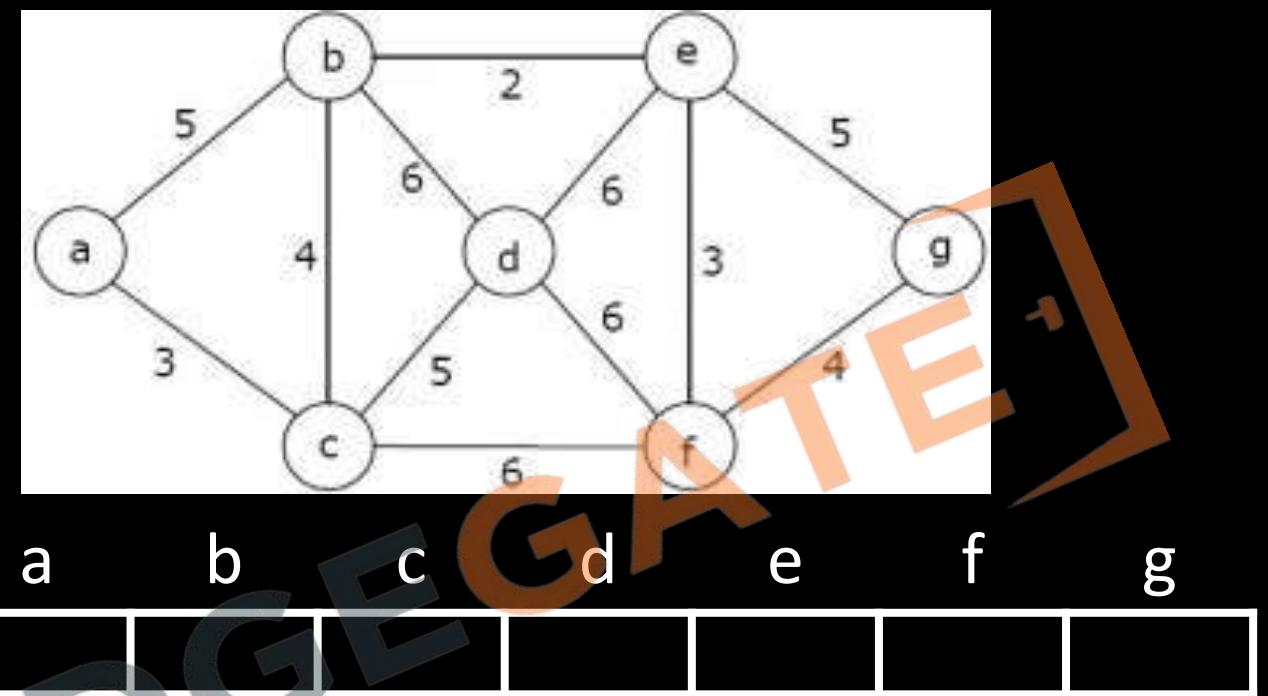
- **Edsger Wybe Dijkstra** (11 May 1930 – 6 August 2002) was a Dutch computer scientist, programmer, software engineer, systems scientist, and science essayist. He received the 1972 Turing Award for fundamental contributions to developing programming languages.



<http://www.knowledgegate.in/GATE>

```
Minimum_Spanning_Tree (G, W, R)
```

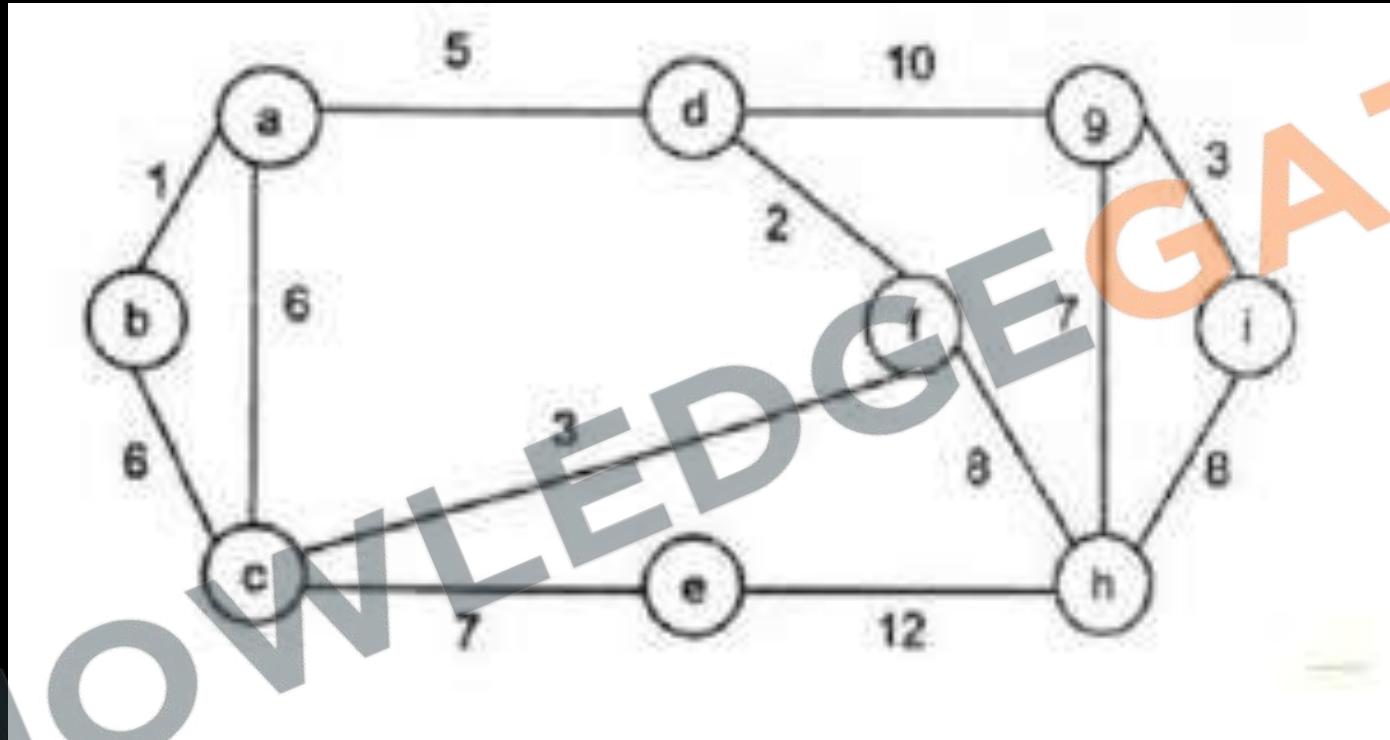
```
{  
    for each  $u \in V(G)$   
    {  
        key[u]  $\leftarrow \infty$   
         $\pi[u] \leftarrow \text{NIL}$   
    }  
    Key[r]  $\leftarrow 0$   
    Q  $\leftarrow V[G]$   
    While ( $Q \neq \emptyset$ )  
    {  
        u  $\leftarrow \text{Extract-Min}(Q)$   
        For each  $v \in \text{adj}[u]$   
        {  
            if ( $v \in Q$  and  $w(u, v) < \text{key}[v]$ )  
            {  
                 $\pi[v] \leftarrow u$   
                key[v]  $\leftarrow w(u, v)$   
            }  
        }  
    }  
}
```



```
1  for each  $u \in G.V$ 
2       $u.key = \infty$ 
3       $u.\pi = \text{NIL}$ 
4   $r.key = 0$ 
5   $Q = G.V$ 
6  while  $Q \neq \emptyset$ 
7       $u = \text{EXTRACT-MIN}(Q)$ 
8      for each  $v \in G.Adj[u]$ 
9          if  $v \in Q$  and  $w(u, v) < v.key$ 
10          $v.\pi = u$ 
11          $v.key = w(u, v)$ 
```

The running time of Prim's algorithm depends on how we implement the min-priority queue Q . If we implement Q as a binary min-heap (see Chapter 6), we can use the BUILD-MIN-HEAP procedure to perform lines 1–5 in $O(V)$ time. The body of the **while** loop executes $|V|$ times, and since each EXTRACT-MIN operation takes $O(\lg V)$ time, the total time for all calls to EXTRACT-MIN is $O(V \lg V)$. The **for** loop in lines 8–11 executes $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$. Within the **for** loop, we can implement the test for membership in Q in line 9 in constant time by keeping a bit for each vertex that tells whether or not it is in Q , and updating the bit when the vertex is removed from Q . The assignment in line 11 involves an implicit DECREASE-KEY operation on the min-heap, which a binary min-heap supports in $O(\lg V)$ time. Thus, the total time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$, which is asymptotically the same as for our implementation of Kruskal's algorithm.

Q For the undirected, weighted graph given below, which of the following sequences of edges represents a correct execution of Prim's algorithm to construct a Minimum Spanning Tree?



- (A) (a, b), (d, f), (f, c), (g, i), (d, a), (g, h), (c, e), (f, h)
- (B) (c, e), (c, f), (f, d), (d, a), (a, b), (g, h), (h, f), (g, i)
- (C) (d, f), (f, c), (d, a), (a, b), (c, e), (f, h), (g, h), (g, i)
- (D) (h, g), (g, i), (h, f), (f, c), (f, d), (d, a), (a, b), (c, e)

Ch-6

Single Source Shortest Paths

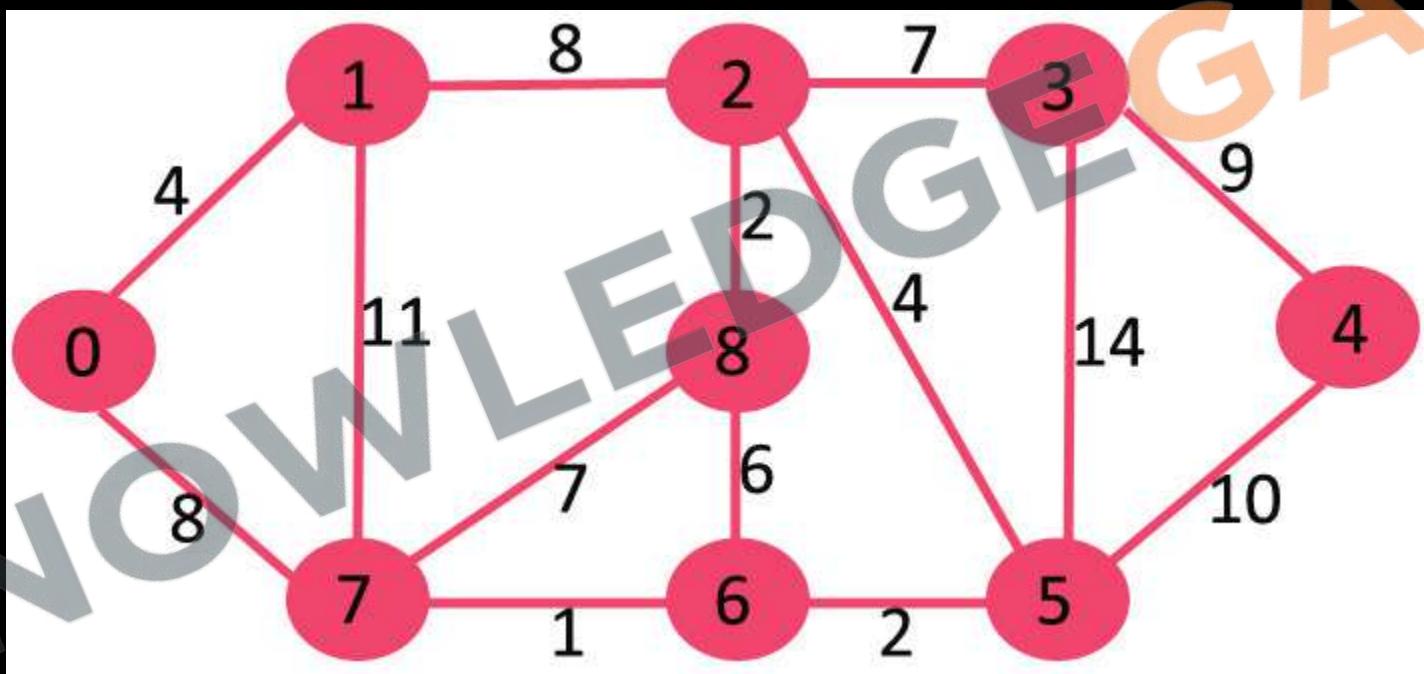
Dijkstra's and Bellman

Ford Algorithms

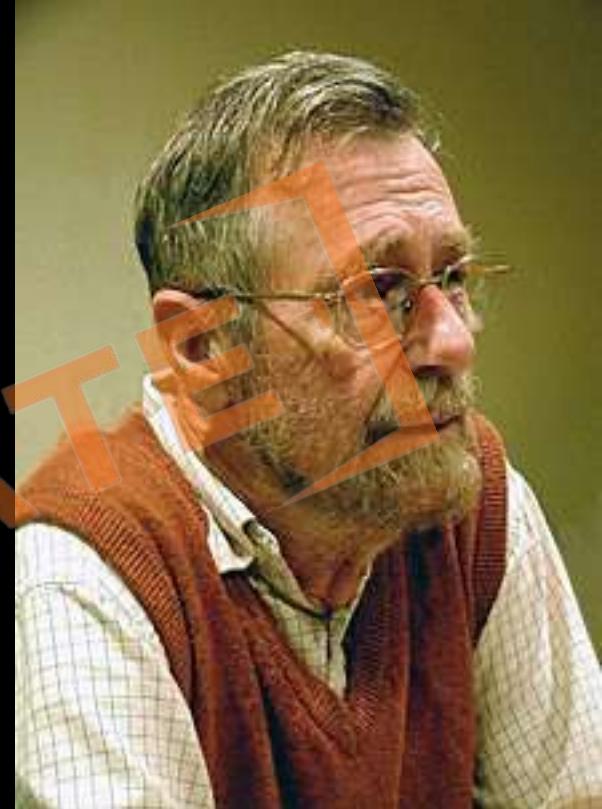
<http://www.knowledgigate.in/GATE>

Single Source Shortest Path

- In graph theory, the **shortest path problem** is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.



- **Edsger Wybe Dijkstra** (11 May 1930 – 6 August 2002) was a Dutch computer scientist, programmer, software engineer, systems scientist, and science essayist. He received the 1972 Turing Award for fundamental contributions to developing programming languages.



<http://www.knowledgegate.in/GATE>

- One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path.
- As I said, it was a twenty-minute invention. In fact, it was published in '59, three years later. The publication is still readable, it is, in fact, quite nice.

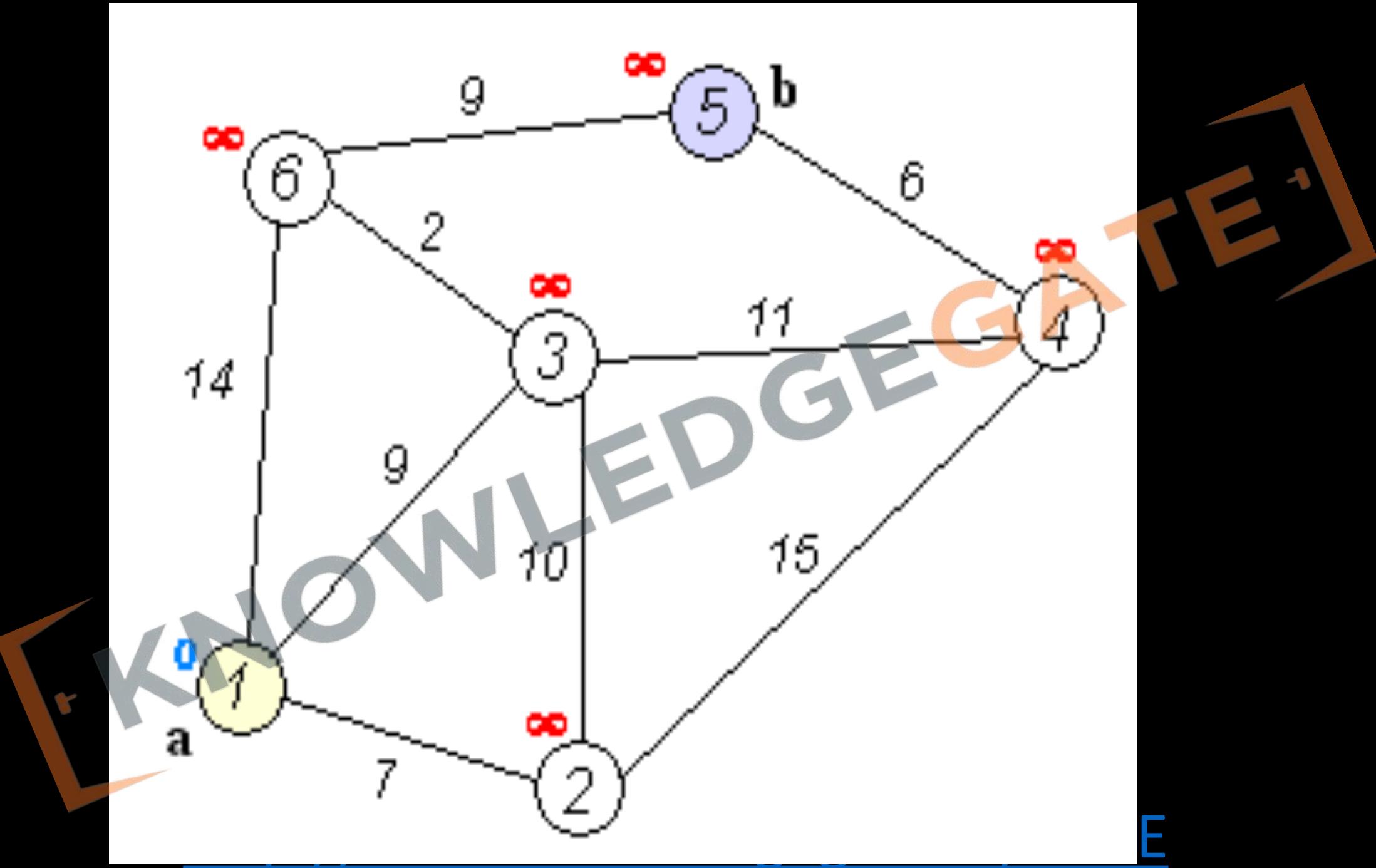




KNOWLEDGE GATE¹

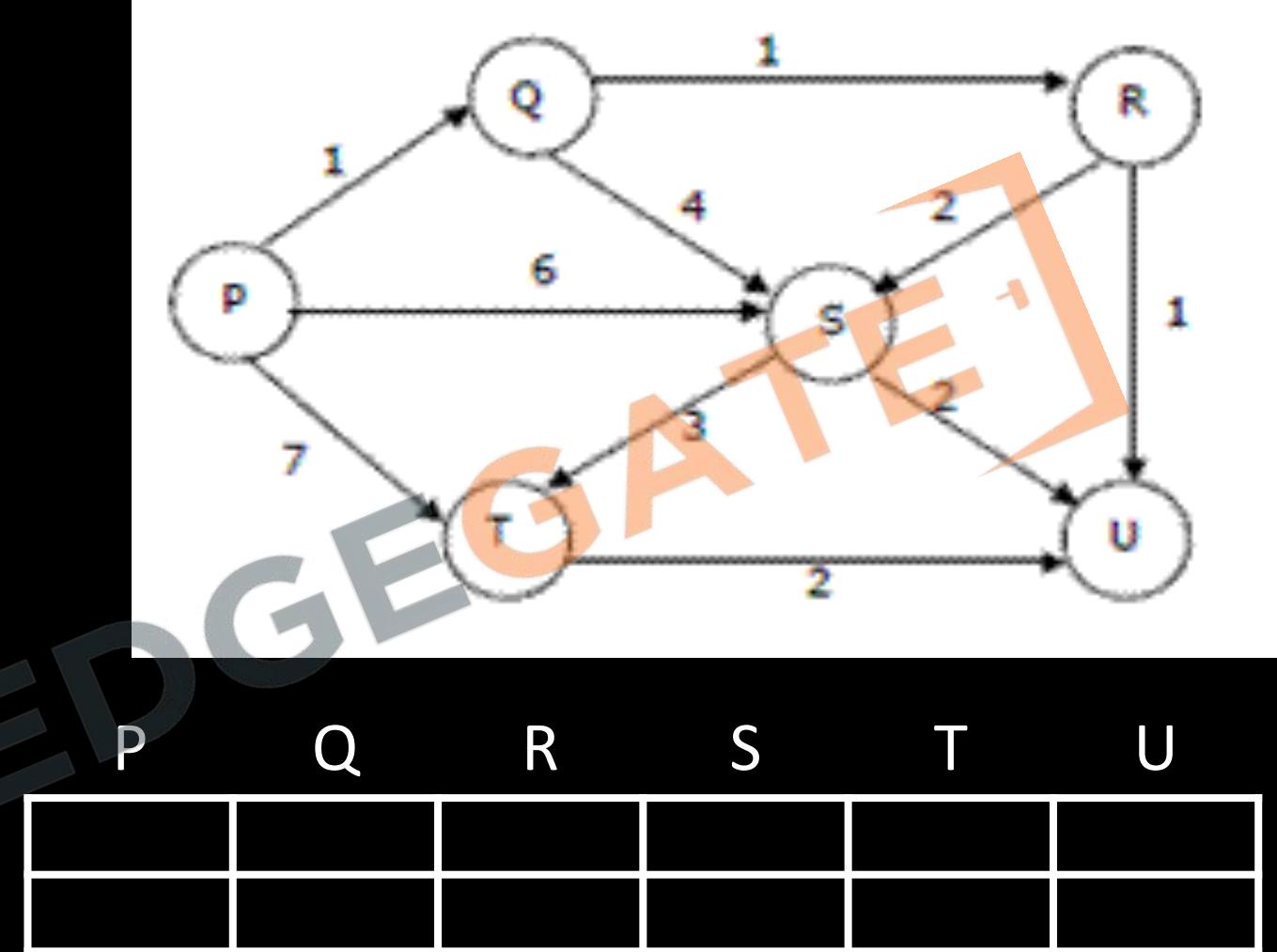


- One of the reasons that it is so nice was that I designed it without pencil and paper. I learned later that one of the advantages of designing without pencil and paper is that you are almost forced to avoid all avoidable complexities.
- Eventually that algorithm became, to my great amazement, one of the cornerstones of my fame.
- A widely used application of shortest path algorithm is network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and Open Shortest Path First (OSPF).



Dijkstra algorithm (G, W, S)

```
{  
    initialize-Single-source ( $G, S$ )  
     $S \leftarrow \emptyset$   
     $Q \leftarrow V[G]$   
    While ( $Q \neq \emptyset$ )  
    {  
         $u \leftarrow \text{extract-min} (Q)$   
         $S \leftarrow S \cup \{u\}$   
        for each vertex  $v \in \text{adj}(u)$   
        {  
            relax ( $u, v, w$ )  
        }  
    }  
}
```



Initialize_Single_Source (G, S)

{

for each vertex $v \in V[G]$

{

$d[v] \leftarrow \infty$

$\Pi[v] \leftarrow \text{NIL}$

}

$d[S] \leftarrow 0$

}

```
Relax (u, v, w)
```

```
{
```

```
    if(d[v] > d[u] + w (u, v))  
    {  
        d[v]  $\leftarrow$  d[u] + w (u, v)  
         $\Pi$ [v]  $\leftarrow$  u  
    }
```

```
}
```

- Guarantee to find optimal solution in a connected graph with positive weights.
- Can fail on graph with negative weights.

DIJKSTRA(G, w, s)

```
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S = \emptyset$ 
3  $Q = G.V$ 
4 while  $Q \neq \emptyset$ 
5    $u = \text{EXTRACT-MIN}(Q)$ 
6    $S = S \cup \{u\}$ 
7   for each vertex  $v \in G.\text{Adj}[u]$ 
8     RELAX( $u, v, w$ )
```

How fast is Dijkstra's algorithm? It maintains the min-priority queue Q by calling three priority-queue operations: INSERT (implicit in line 3), EXTRACT-MIN (line 5), and DECREASE-KEY (implicit in RELAX, which is called in line 8). The algorithm calls both INSERT and EXTRACT-MIN once per vertex. Because each vertex $u \in V$ is added to set S exactly once, each edge in the adjacency list $\text{Adj}[u]$ is examined in the **for** loop of lines 7–8 exactly once during the course of the algorithm. Since the total number of edges in all the adjacency lists is $|E|$, this **for** loop iterates a total of $|E|$ times, and thus the algorithm calls DECREASE-KEY at most $|E|$ times overall. (Observe once again that we are using aggregate analysis.)

DIJKSTRA(G, w, s)

```
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S = \emptyset$ 
3  $Q = G.V$ 
4 while  $Q \neq \emptyset$ 
5    $u = \text{EXTRACT-MIN}(Q)$ 
6    $S = S \cup \{u\}$ 
7   for each vertex  $v \in G.\text{Adj}[u]$ 
8     RELAX( $u, v, w$ )
```

The running time of Dijkstra's algorithm depends on how we implement the min-priority queue. Consider first the case in which we maintain the min-priority queue by taking advantage of the vertices being numbered 1 to $|V|$. We simply store $v.d$ in the v th entry of an array. Each INSERT and DECREASE-KEY operation takes $O(1)$ time, and each EXTRACT-MIN operation takes $O(V)$ time (since we have to search through the entire array), for a total time of $O(V^2 + E) = O(V^2)$.

DIJKSTRA(G, w, s)

```
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S = \emptyset$ 
3  $Q = G.V$ 
4 while  $Q \neq \emptyset$ 
5    $u = \text{EXTRACT-MIN}(Q)$ 
6    $S = S \cup \{u\}$ 
7   for each vertex  $v \in G.\text{Adj}[u]$ 
8     RELAX( $u, v, w$ )
```

If the graph is sufficiently sparse—in particular, $E = o(V^2 / \lg V)$ —we can improve the algorithm by implementing the min-priority queue with a binary min-heap. (As discussed in Section 6.5, the implementation should make sure that vertices and corresponding heap elements maintain handles to each other.) Each EXTRACT-MIN operation then takes time $O(\lg V)$. As before, there are $|V|$ such operations. The time to build the binary min-heap is $O(V)$. Each DECREASE-KEY operation takes time $O(\lg V)$, and there are still at most $|E|$ such operations. The total running time is therefore $O((V + E) \lg V)$, which is $O(E \lg V)$ if all vertices are reachable from the source. This running time improves upon the straightforward $O(V^2)$ -time implementation if $E = o(V^2 / \lg V)$.

DIJKSTRA(G, w, s)

```
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S = \emptyset$ 
3  $Q = G.V$ 
4 while  $Q \neq \emptyset$ 
5    $u = \text{EXTRACT-MIN}(Q)$ 
6    $S = S \cup \{u\}$ 
7   for each vertex  $v \in G.\text{Adj}[u]$ 
8     RELAX( $u, v, w$ )
```

We can in fact achieve a running time of $O(V \lg V + E)$ by implementing the min-priority queue with a Fibonacci heap (see Chapter 19). The amortized cost of each of the $|V|$ EXTRACT-MIN operations is $O(\lg V)$, and each DECREASE-KEY call, of which there are at most $|E|$, takes only $O(1)$ amortized time. Historically, the development of Fibonacci heaps was motivated by the observation that Dijkstra's algorithm typically makes many more DECREASE-KEY calls than EXTRACT-MIN calls, so that any method of reducing the amortized time of each DECREASE-KEY operation to $o(\lg V)$ without increasing the amortized time of EXTRACT-MIN would yield an asymptotically faster implementation than with binary heaps.

Problem with Dijkstra's algorithm

- Negative Weight Edge



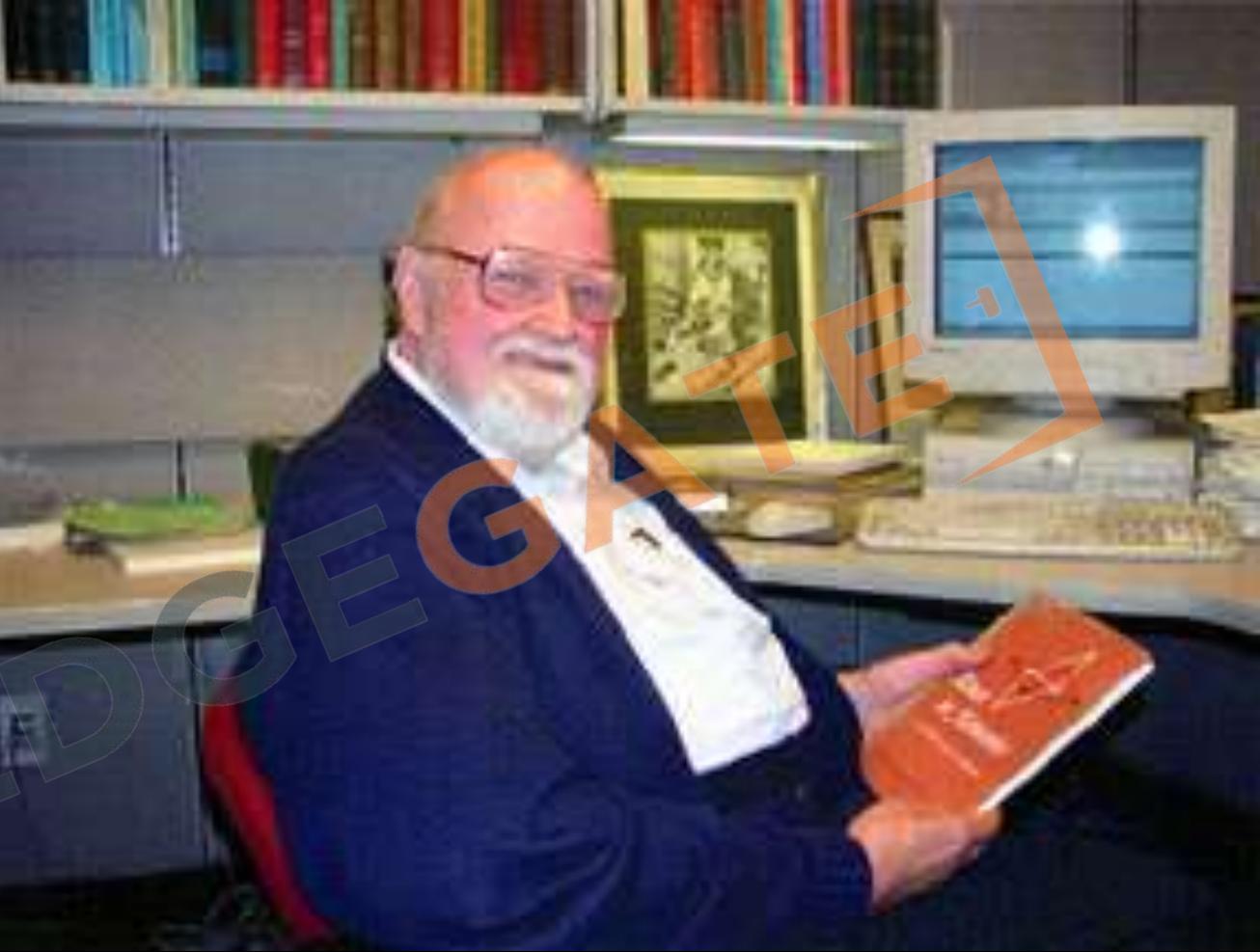
<http://www.knowledgegate.in/GATE>

Bellman–Ford Algorithm

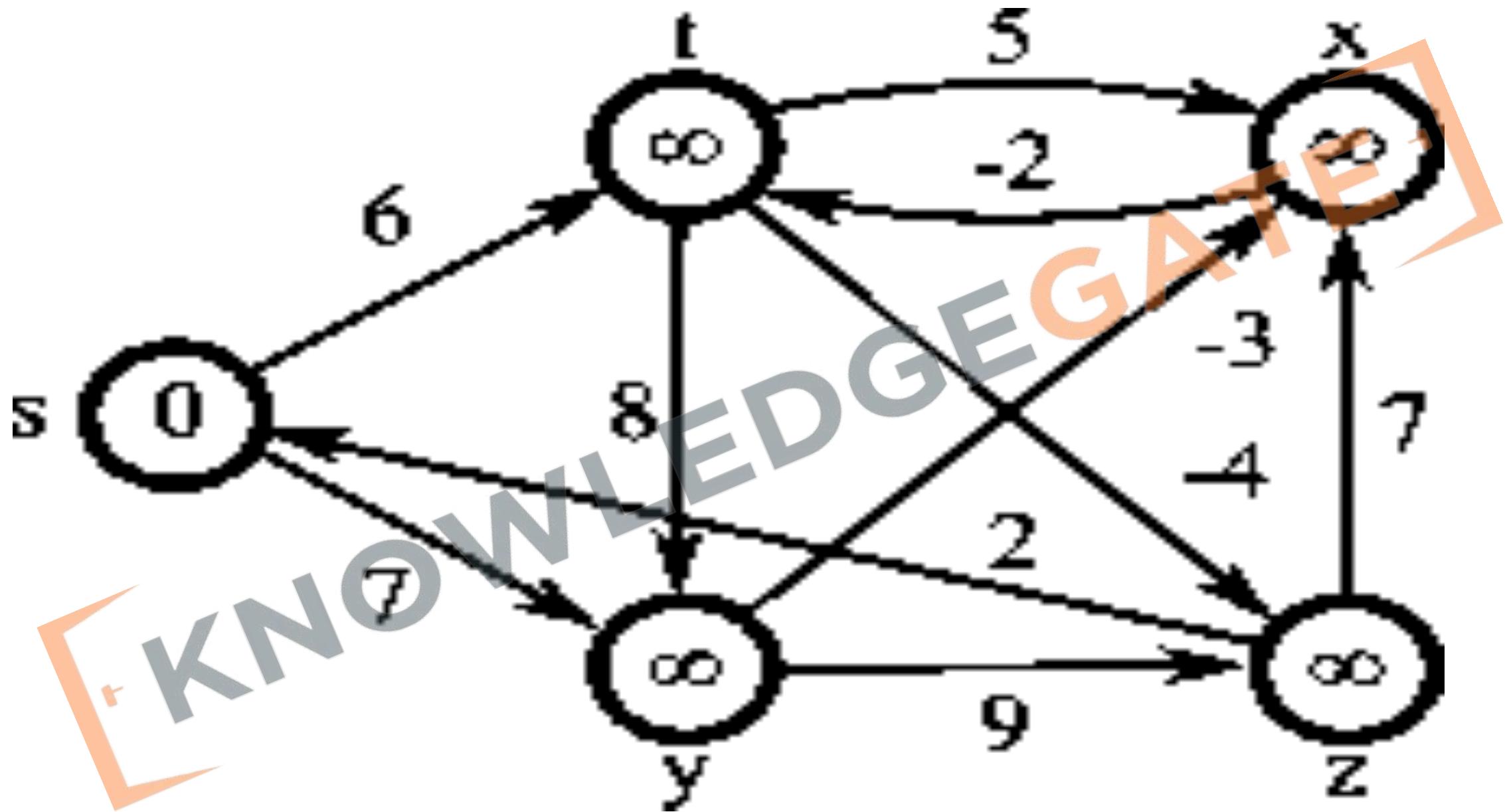
- The **Bellman–Ford algorithm** is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph.
- It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.
- The algorithm was first proposed by Alfonso Shimbel (1955), but is instead named after Richard Bellman and Lester Ford Jr., who published it in 1958 and 1956, respectively.



Richard E. Bellman



L. R. Ford Jr



```
Bellman_ford (G, W, S)
```

```
{
```

```
    initialize-Single-Source (G, S)
```

```
    for i  $\leftarrow$  1 to  $|V(G)| - 1$ 
```

```
{
```

```
    for each edge  $(u, v) \in E(G)$ 
```

```
{
```

```
        Relax( $u, v, w$ )
```

```
}
```

```
}
```

```
    for each edge  $(u, v) \in E(G)$ 
```

```
{
```

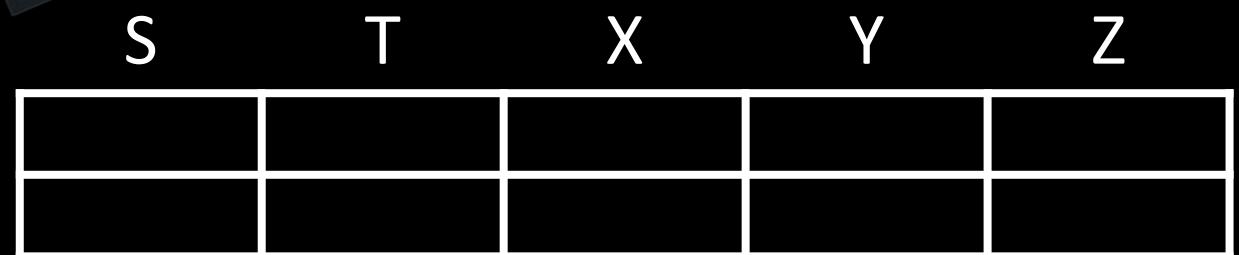
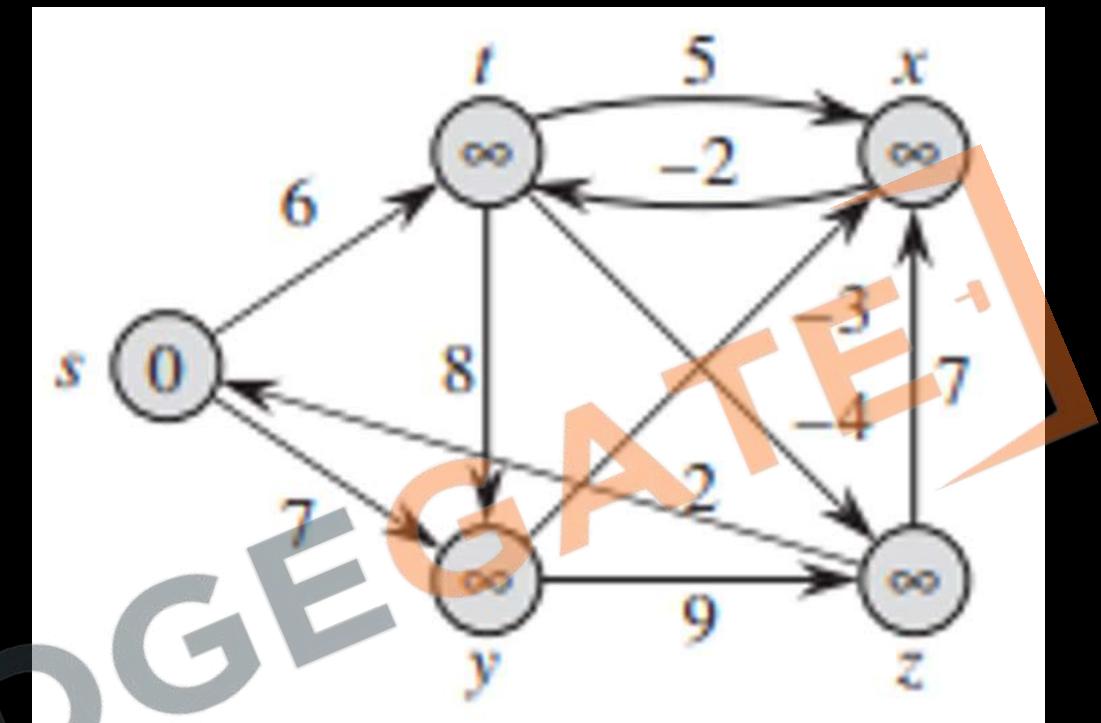
```
        if( $d[v] > d[u] + w(u, v)$ )
```

```
{
```

```
            Return false
```

```
}
```

```
}
```



Initialize_Single_Source (G, S)

{

for each vertex $v \in V[G]$

{

$d[v] \leftarrow \infty$

$\Pi[v] \leftarrow \text{NIL}$

}

$d[S] \leftarrow 0$

}

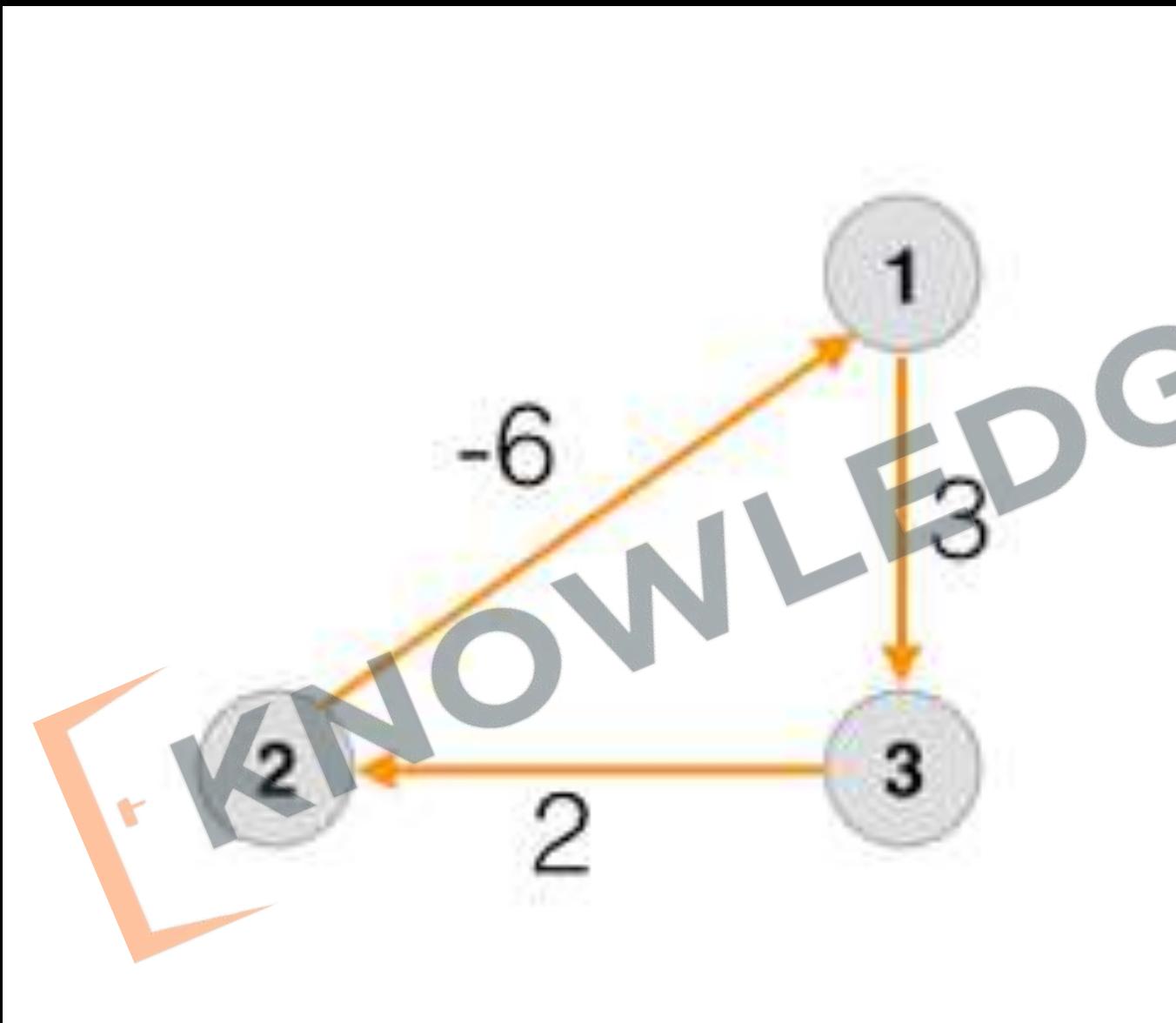
```
Relax (u, v, w)
{
    if(d[v] < d[u] + w (u, v))
    {
        d[v] <- d[u] + w (u, v)
        π[v] <- u
    }
}
```

BELLMAN-FORD(G, w, s)

```
1  INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  for  $i = 1$  to  $|G.V| - 1$ 
3    for each edge  $(u, v) \in G.E$ 
4      RELAX( $u, v, w$ )
5  for each edge  $(u, v) \in G.E$ 
6    if  $v.d > u.d + w(u, v)$ 
7      return FALSE
8  return TRUE
```

The Bellman-Ford algorithm runs in time $O(VE)$, since the initialization in line 1 takes $\Theta(V)$ time, each of the $|V| - 1$ passes over the edges in lines 2–4 takes $\Theta(E)$ time, and the **for** loop of lines 5–7 takes $O(E)$ time.

Bellman–Ford algorithm with negative weight cycle



Ch-7

Dynamic Programming

with Examples Such as Knapsack. All Pair Shortest Paths – Warshal's and Floyd's Algorithms, Resource Allocation Problem. Backtracking, Branch and Bound with Examples Such as Travelling Salesman Problem, Graph Coloring, n-Queen Problem, Hamiltonian Cycles and Sum of Subsets.

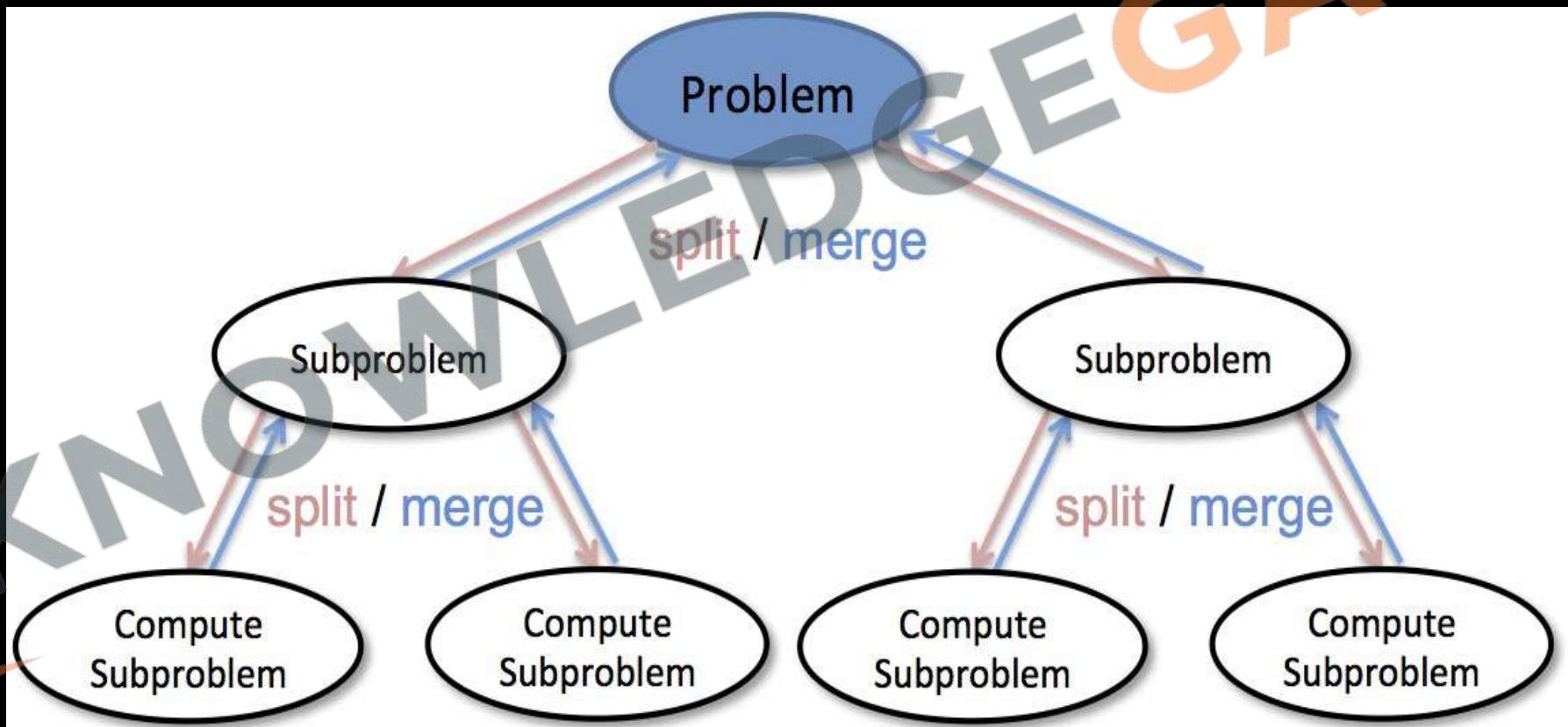
अगर आप अपने Past से कुछ सीख नहीं सकते
तो जीवनभर छोटे काम ही करते रहेंगे।

-Dynamic Programming

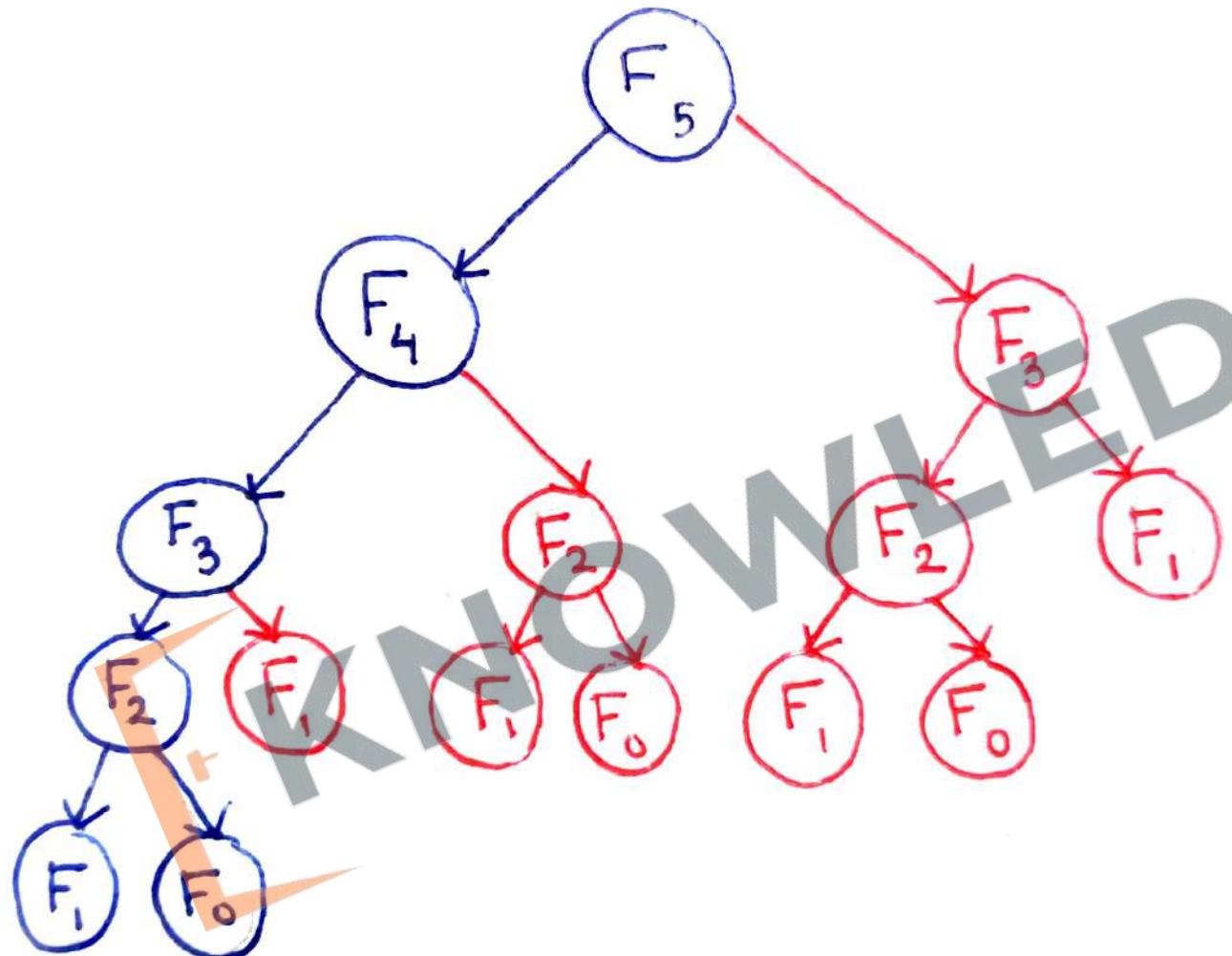


Dynamic Programming

- Divide and conquer partition the problem into independent subproblem, solve the subproblems recursively and then combine their solutions to solve the original problems.



$$\begin{aligned}F(0) &= 0 \\F(1) &= 1 \\F(n) &= F(n - 1) + F(n - 2)\end{aligned}$$



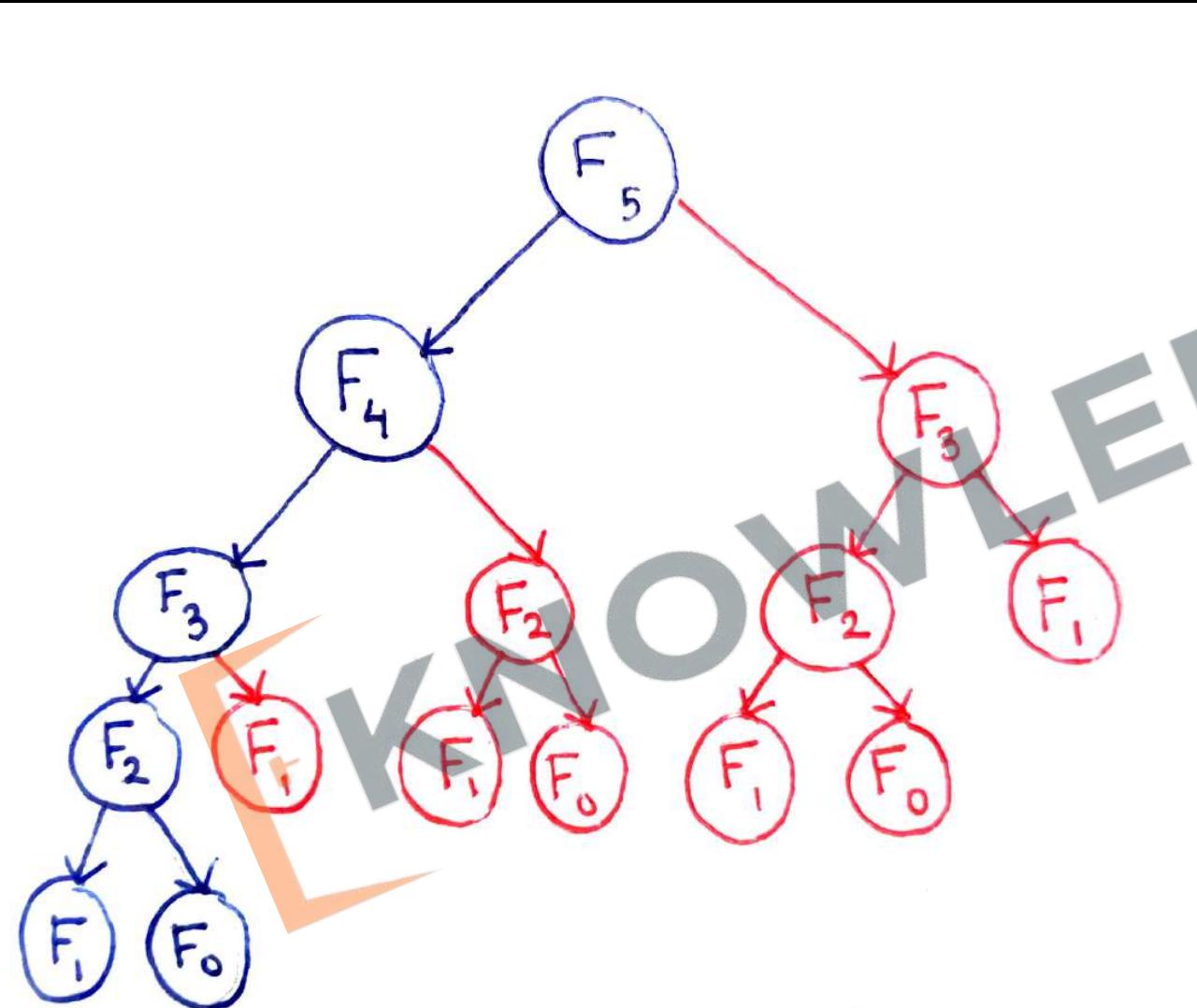
n	0	1	2	3	4	5
F(n)						

KNOWLEDGE GATE

- Dynamic programming is like the divide and conquer method, solve problems by combining the solutions to the subproblems.
- In contrast, dynamic programming is applicable when the subproblems are not independent, i.e. when subproblems share subsubproblems.

<http://www.knowledgegate.in/GATE>

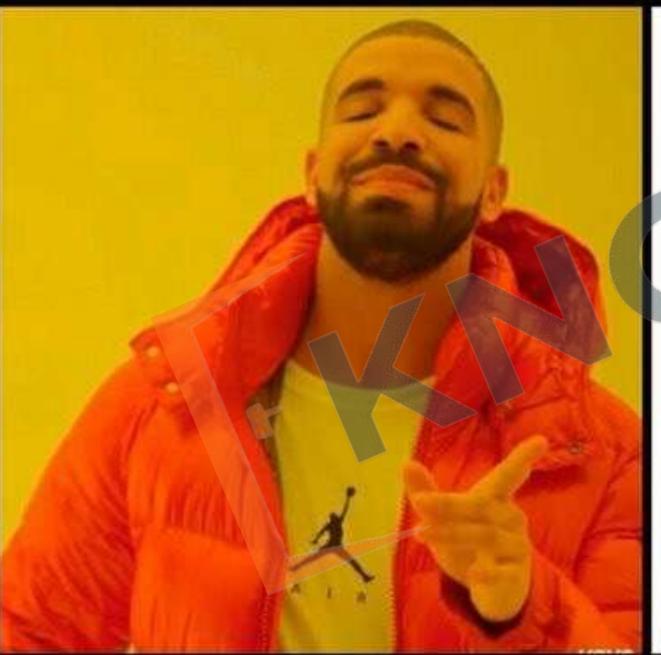
- A dynamic-programming algorithm solves every subsubproblems just one and then saves its answer in a table there by avoiding the work of recomputing the answer every time the subproblem is encountered.



n	$f(n)$
0	1
1	1
2	2
3	3
4	5
5	8
6	13
7	21
8	34
9	55
10	89
11	144
12	233
13	377
14	610
15	987
16	1597
17	2584
18	4181



programming

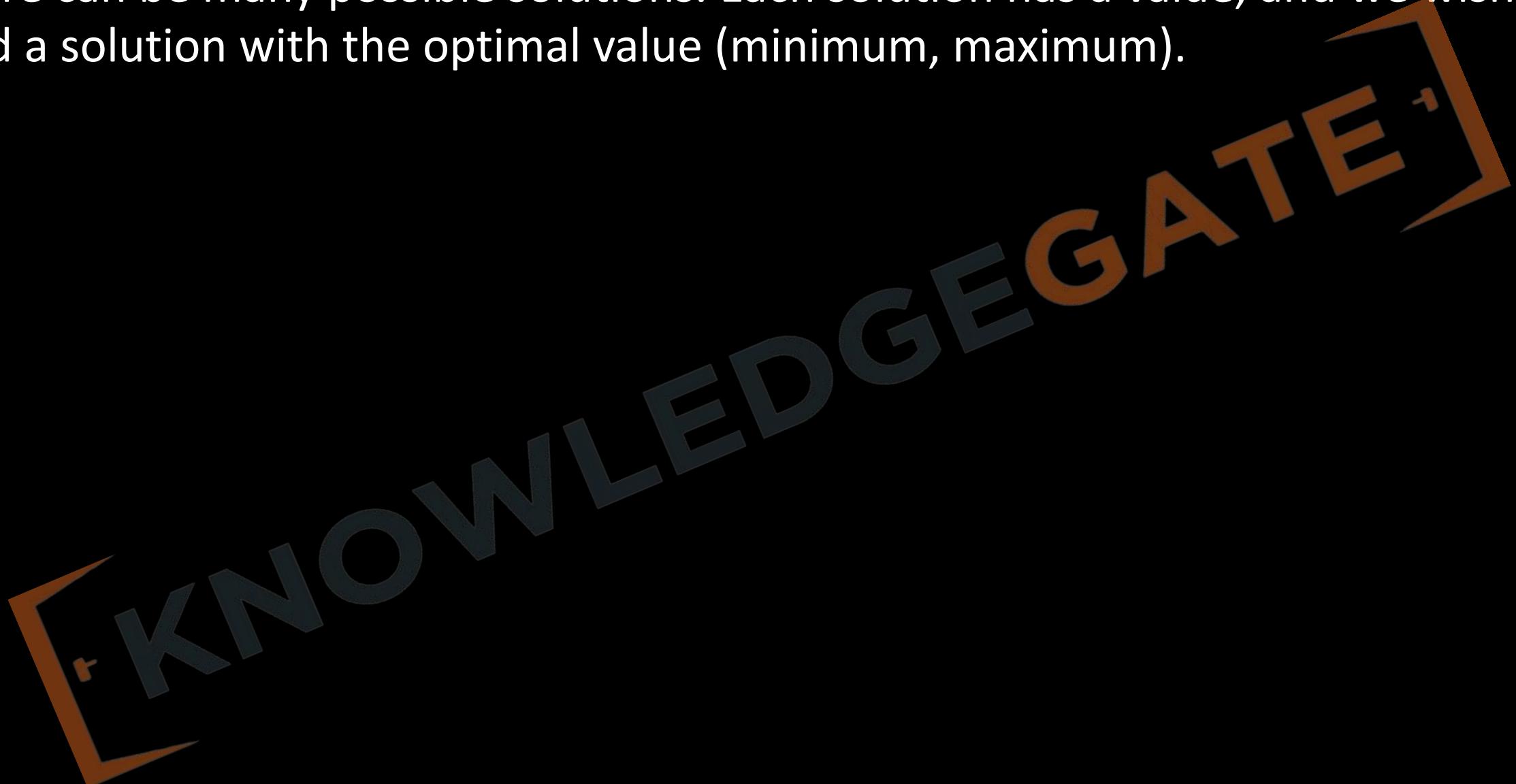


dynamic
programming

KNOWLEDGE GATE¹

n/GATE

- Dynamic programming is typically applied to optimization problems. In such case there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal value (minimum, maximum).



<http://www.knowledgegate.in/GATE>

- There are four steps of dynamic programming
 - Characterize the solution of an optimal solution.
 - Recursively define the value of an optimal solution.
 - Compute the value of an optimal solution in a bottom-up-fashion.
 - Construct an optimal solution from computed information.

भाई असली बात तो example से समझ आएगी

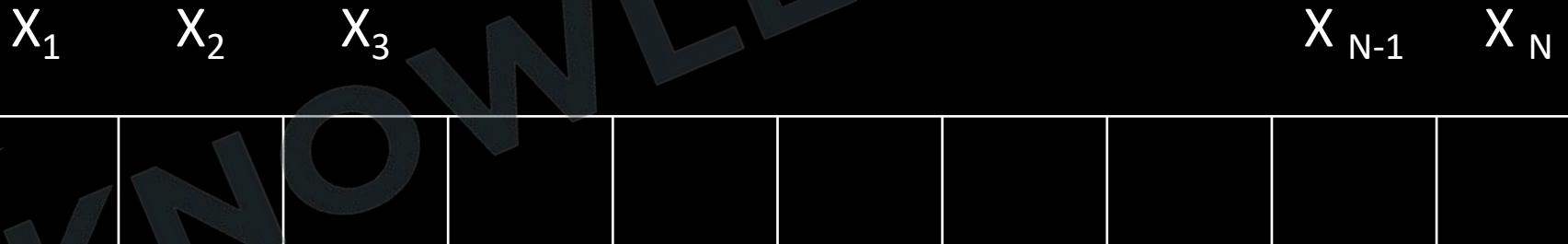
0/1 Knap Sack Problem

- The **knapsack problem** or **rucksack problem** is a problem in combinatorial optimization: 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.



Problem Definition

- More formally there are n number of objects $O_1, O_2, O_3 \dots O_n$, each object has a weight associated with its W_i , and a profit associated with it P_i , we can take x_i either 0 or 1.
- Weight Condition $\sum_{i=1}^n W_i \cdot X_i \leq M$
- Profit $\sum_{i=1}^n P_i \cdot X_i$



Object	O ₁	O ₂	O ₃	O ₄
Profit	1	2	5	6
Weight	2	3	4	5

Θ

$$KS(n, w) = \max \begin{cases} KS(n-1, w - wt[n]) + p[n] \\ KS(n-1, w) \end{cases}$$

$wt[n] > w$

$n = 4$ $w = 8$

wt	p	W=0	W=1	W=2	W=3	W=4	W=5	W=6	W=7	W=8
0	0	0	0	0	0	0	0	0	0	0
1	2	0	0	1	1	1	1	1	1	1
2	3	0	0	1	2	2	3	3	3	3
5	4	0	0	1	2	5	5	6	7	7
6	5	0	0	1	2	5	6	6	7	8

Floyd warshall problem

- The Floyd–Warshall algorithm was published in its currently recognized form by Robert Floyd in 1962. However, it is essentially the same as algorithms previously published by Stephen Warshall in 1962.



Floyd warshall problem

- The Floyd-Warshall algorithm is used to find the shortest paths between all pairs of vertices in a weighted graph.
- It uses a dynamic programming approach to incrementally improve the solution by considering all possible paths.

Floyd warshall problem

a b c

$$D^0 = \begin{matrix} a \\ b \\ c \end{matrix}$$

a b c

$$D^a = \begin{matrix} a \\ b \\ c \end{matrix}$$

a b c

$$D^b = \begin{matrix} a \\ b \\ c \end{matrix}$$

a b c

$$D^c = \begin{matrix} a \\ b \\ c \end{matrix}$$

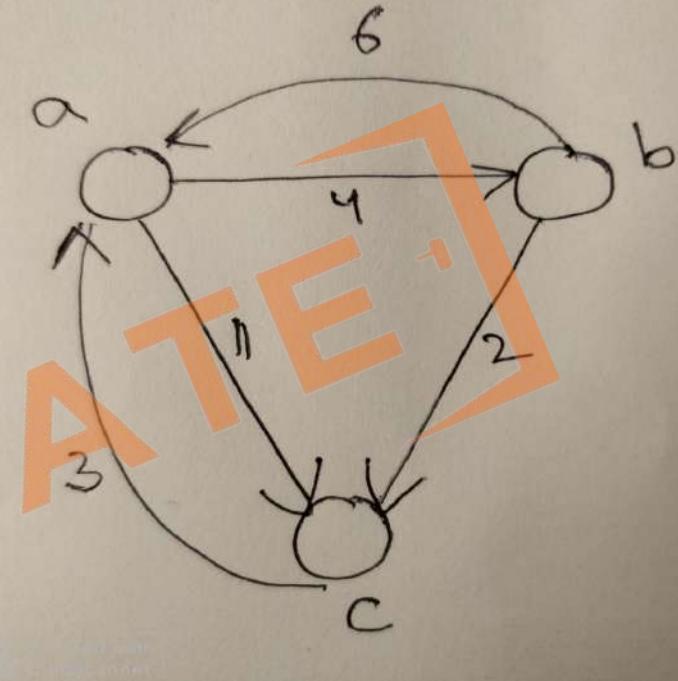
$$\Pi^0 = \begin{matrix} a \\ b \\ c \end{matrix}$$

$$\Pi^a = \begin{matrix} a \\ b \\ c \end{matrix}$$

$$\Pi^b = \begin{matrix} a \\ b \\ c \end{matrix}$$

$$\Pi^c = \begin{matrix} a \\ b \\ c \end{matrix}$$

<http://www.knowledgegate.in/GATE>



- **Algorithm Steps:**
 - Initially, the distance between all pairs of vertices is assumed to be infinity, except for the distance from a vertex to itself, which is zero.
 - The algorithm then iteratively updates the distance matrix to include the shortest path using at most k vertices, where k goes from 1 to the number of vertices in the graph.
 - If a shorter path is found, the distance matrix is updated accordingly.
- **Time Complexity:** It has a time complexity of $O(n^3)$, where n is the number of vertices in the graph.

Sum of subset problem

- Given a set of non-negative integers, and a value sum , determine if there is a subset of the given set with sum equal to given sum .

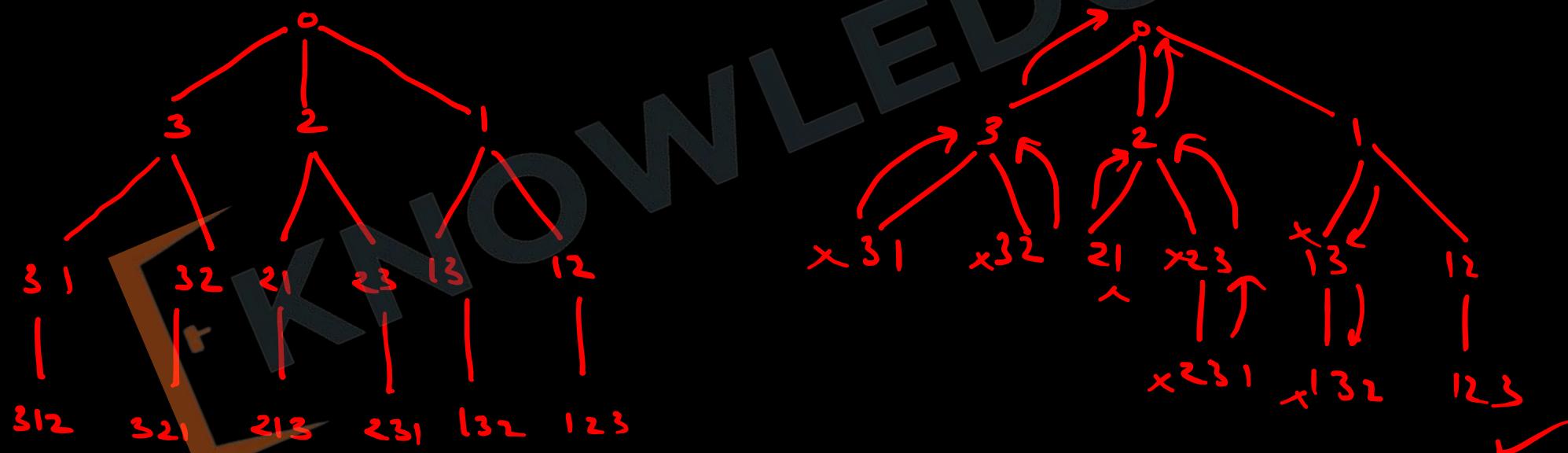
Q Consider a set of non-negative integer $S = \{2, 3, 7, 8, 10\}$, find if there is a sub set of S with sum equal to 14?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2															
3															
7															
8															
10															

- The subset-sum problem is defined as follows. Given a set of n positive integers, $S = \{a_1, a_2, a_3, \dots, a_n\}$ and positive integer W, is there a subset of S whose elements sum to W
- A dynamic program for solving this problem uses a 2-dimensional Boolean array X, with n rows and W+1 column. $X[i, j], 1 \leq i \leq n, 0 \leq j \leq W$, is TRUE if and only if there is a subset of $\{a_1, a_2, \dots, a_i\}$ whose elements sum to j.
- the following is valid for $2 \leq i \leq n$ and $a_i \leq j \leq W$
 - $X[i, j] = X[i - 1, j] \vee X[i - 1, j - a_i]$

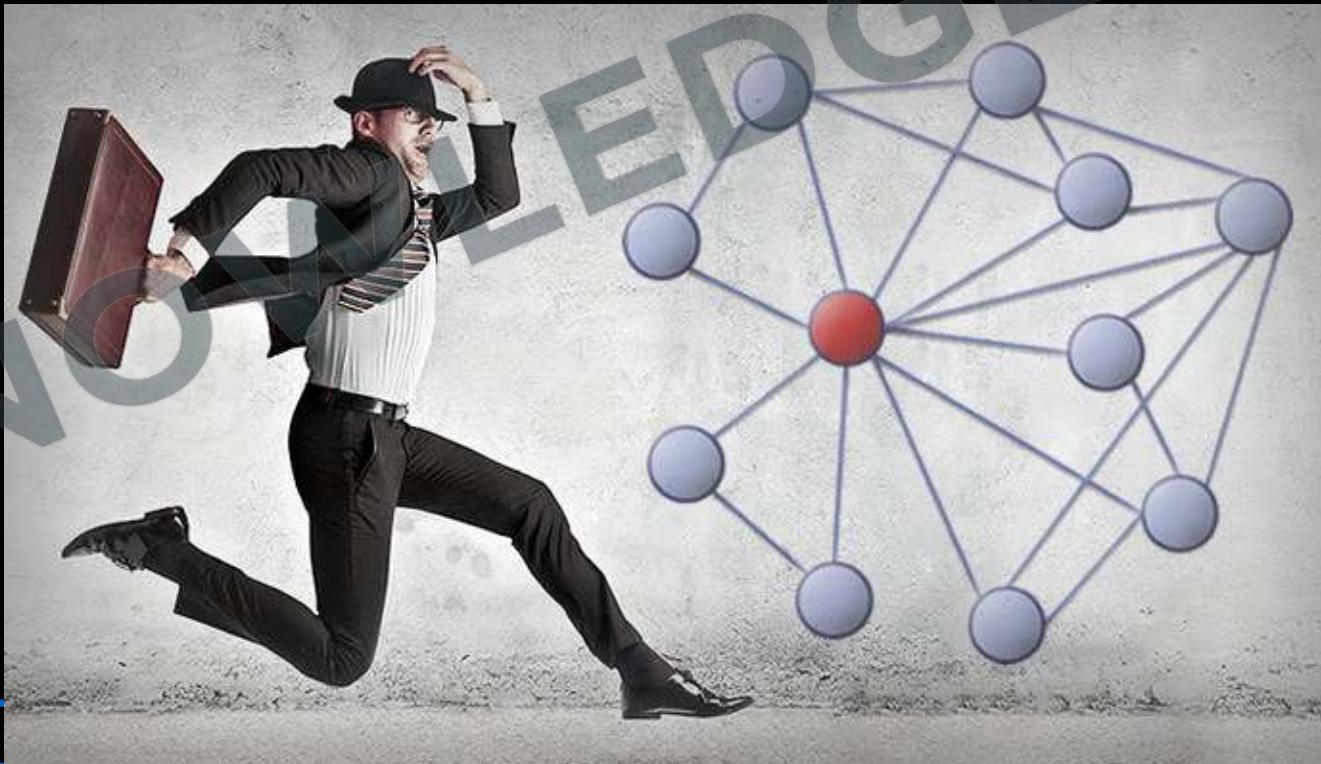
Backtracking

- Backtracking is a programming method used to solve problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints (Bounding function) of the problem at any point of time.
- Backtracking uses brute force approach generating a State space tree, following depth first search.
- Backtracking is not for optimality problems; it is used in general where we have multiple solutions, and we want all those solutions.

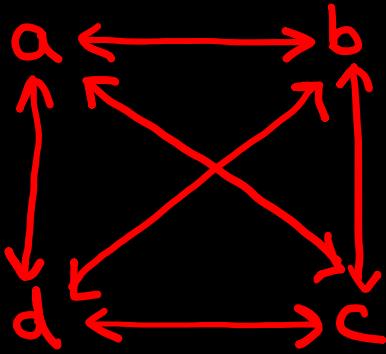


Travelling Salesman Problem

- The Travelling Salesman Problem (TSP) is a classic problem in computer science and operations research.
- **Problem Definition:** The TSP involves finding the shortest possible route that visits a set of cities and returns to the origin city. It's an NP-hard problem in computational complexity theory. As the number of cities increases, the number of possible routes increases factorially, making the problem computationally intensive.

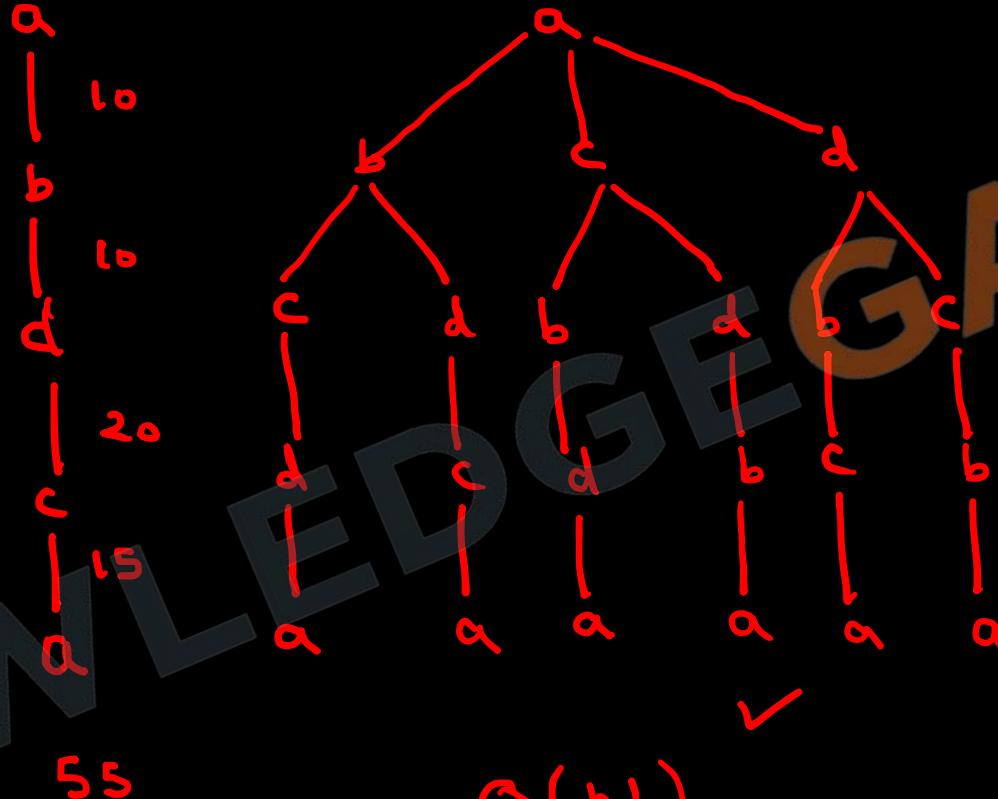


Travelling Salesman Problem



Greedy

	a	b	c	d
a	0	10	15	20
b	5	0	25	10
c	15	30	0	5
d	15	10	20	0

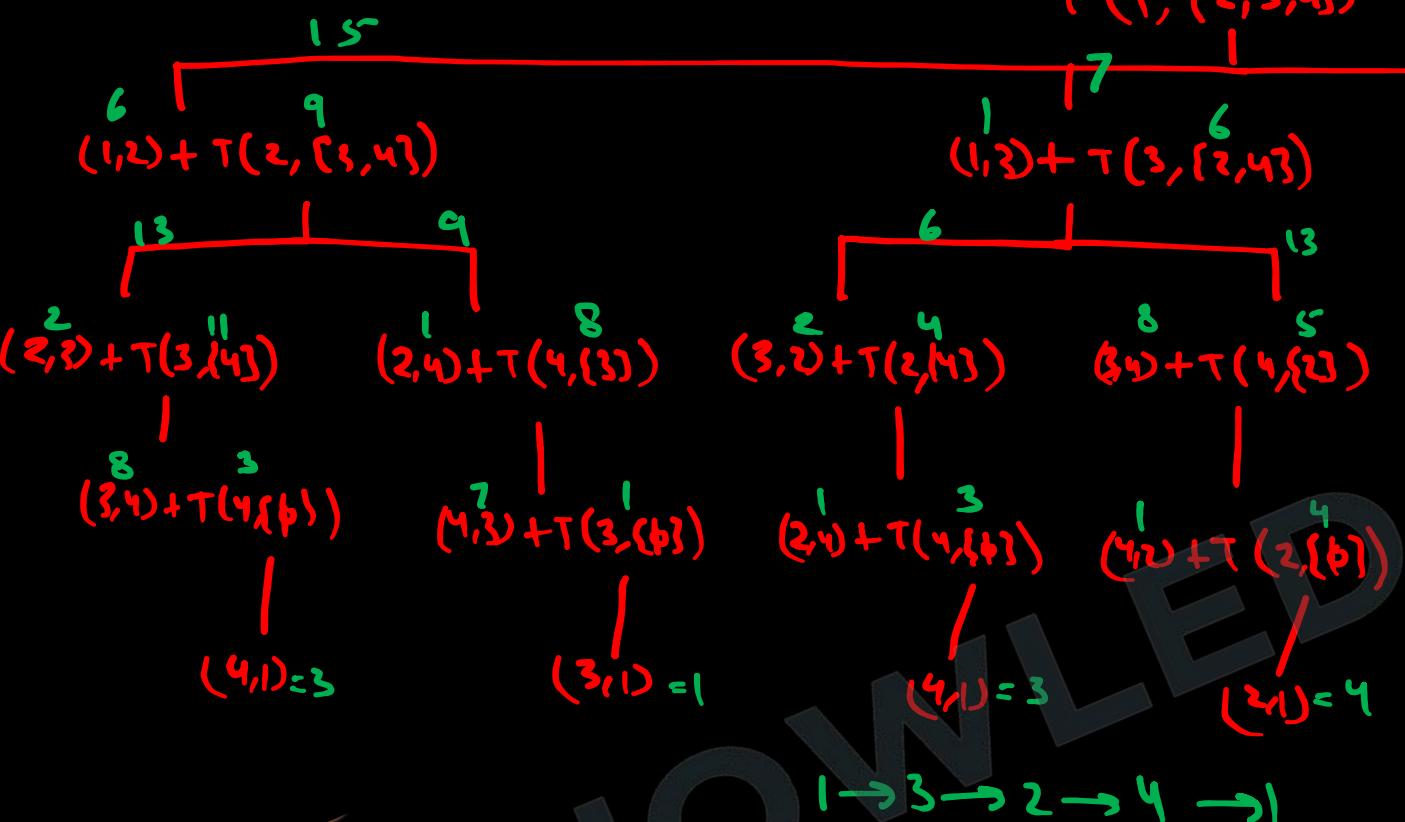


$O(n!)$



$O(n^n)$

Travelling Salesman Problem



$$TSP(i, S) = \min_{k \in S} c(i, k) + TSP(k, S - \{k\})$$

$i \in \{1, 2, 3, 4\}$

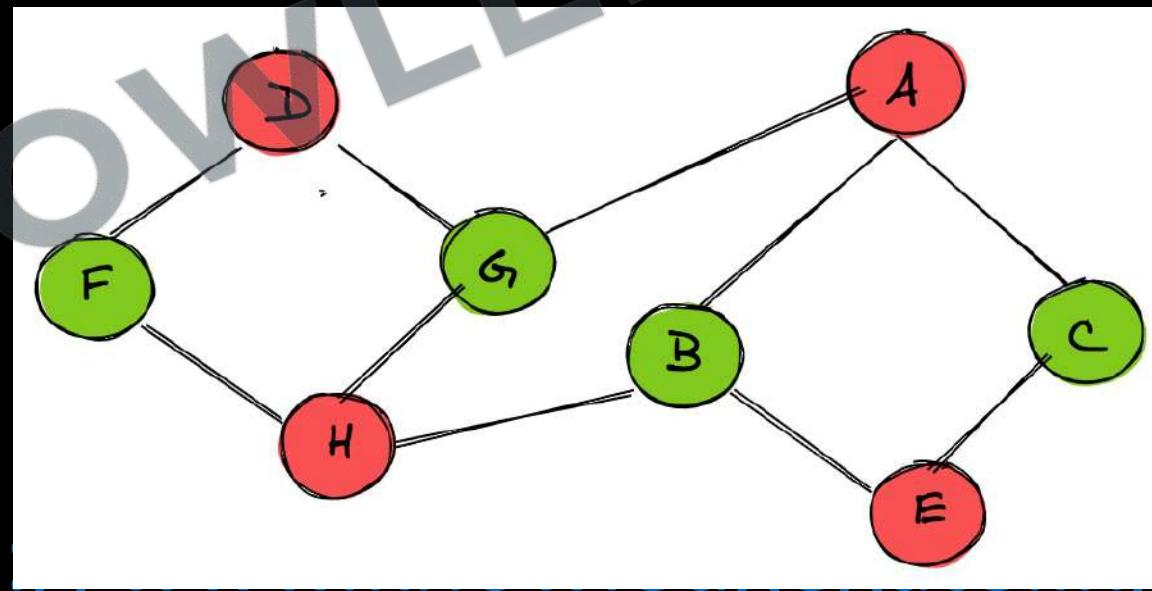
$$(n-1)2^{n-2}$$

$n=4 \rightarrow 12$

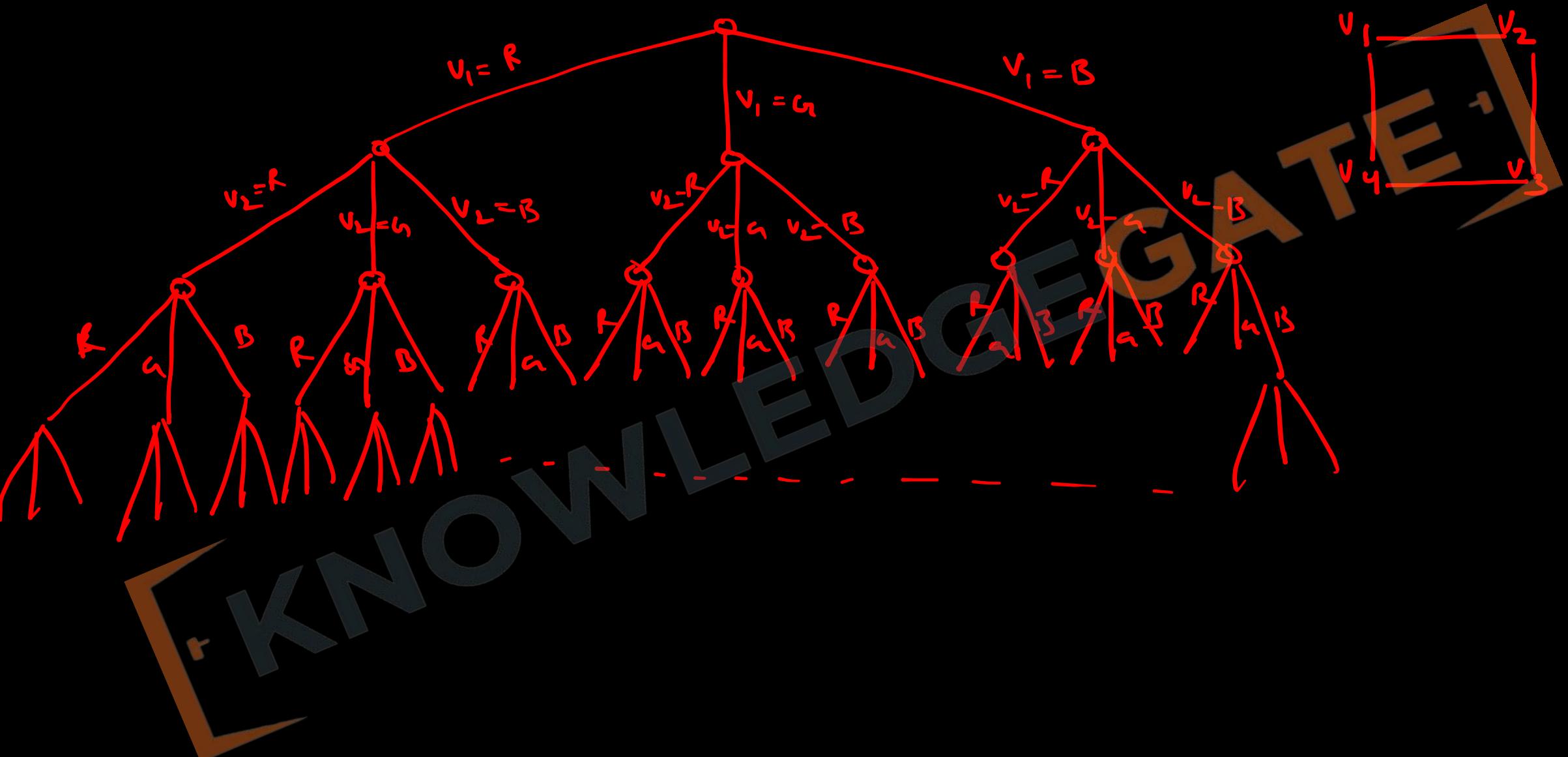


Graph colouring

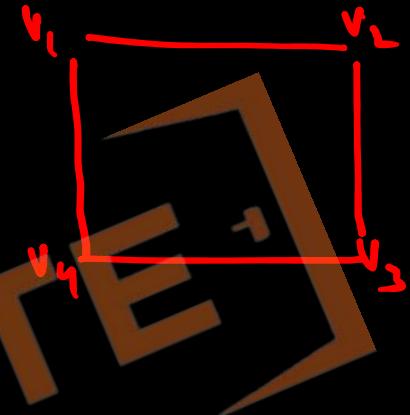
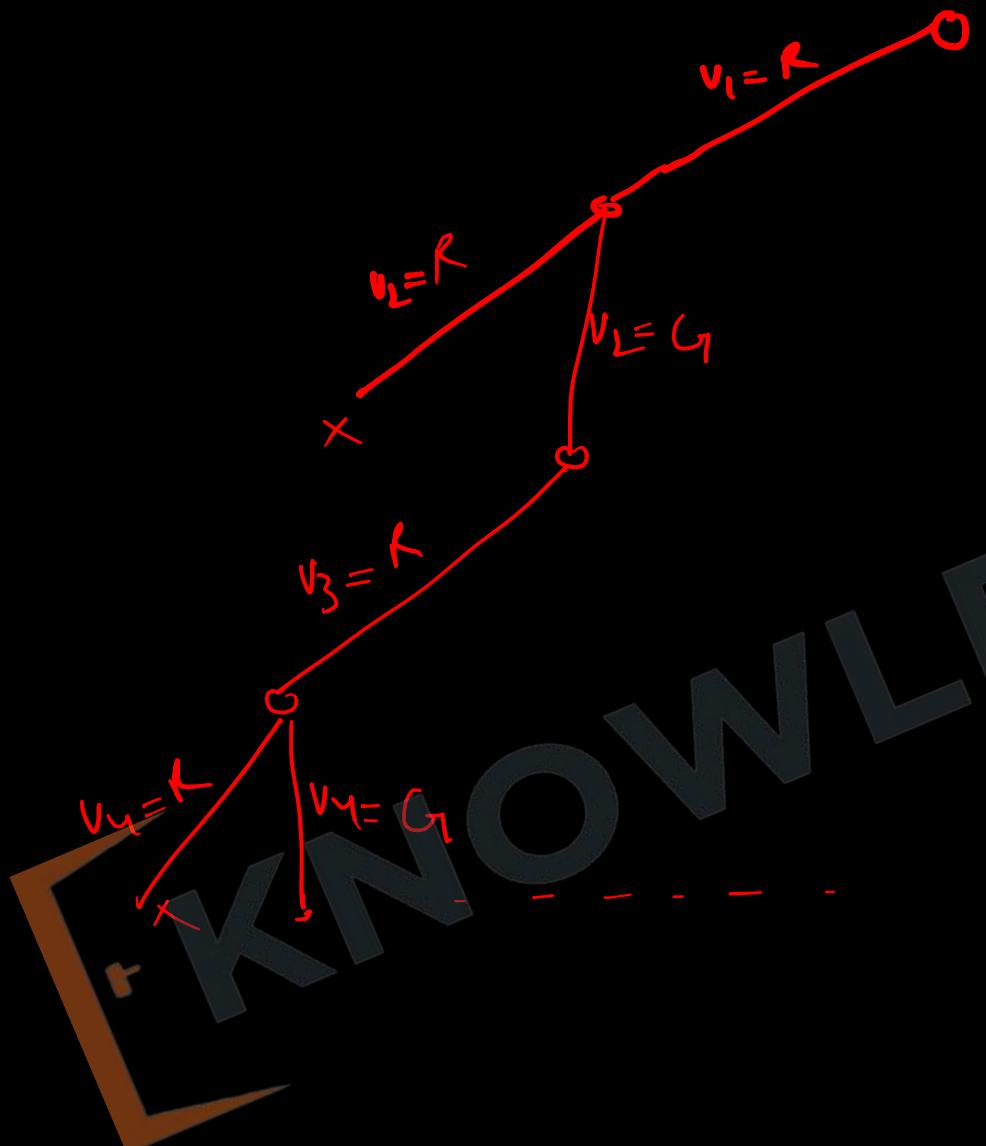
- **Problem Definition:** It involves assigning colors to the vertices of a graph such that no two adjacent vertices share the same color. The goal is to minimize the number of colors used.
- **Applications:** It's used in scheduling, assigning frequencies in mobile networks, map coloring, and solving Sudoku puzzles.
- **Chromatic Number:** This is the minimum number of colors needed to color a graph. Determining this number is a central challenge of the problem.
- **NP-Hard Problem:** Like the Travelling Salesman Problem, graph coloring is NP-Hard for general graphs. This means finding the optimal solution is computationally challenging.
- **Backtracking Algorithms:** These are more exhaustive methods that try different color assignments and backtrack when a conflict is found.



Graph colouring

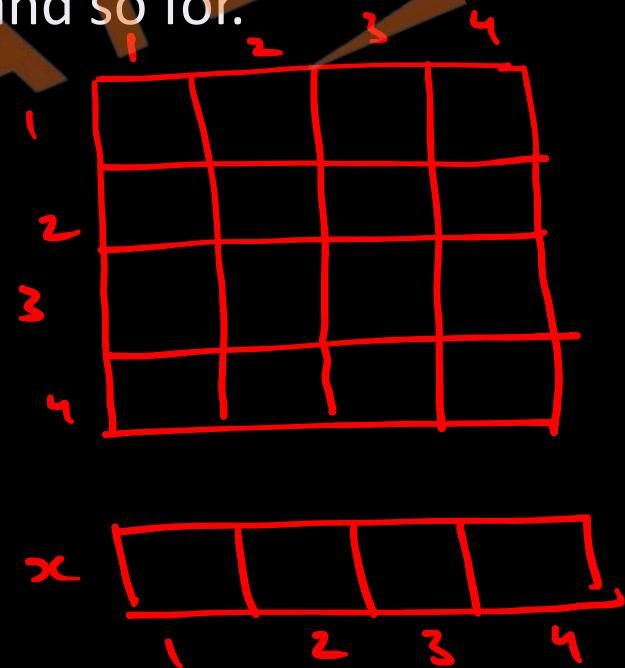
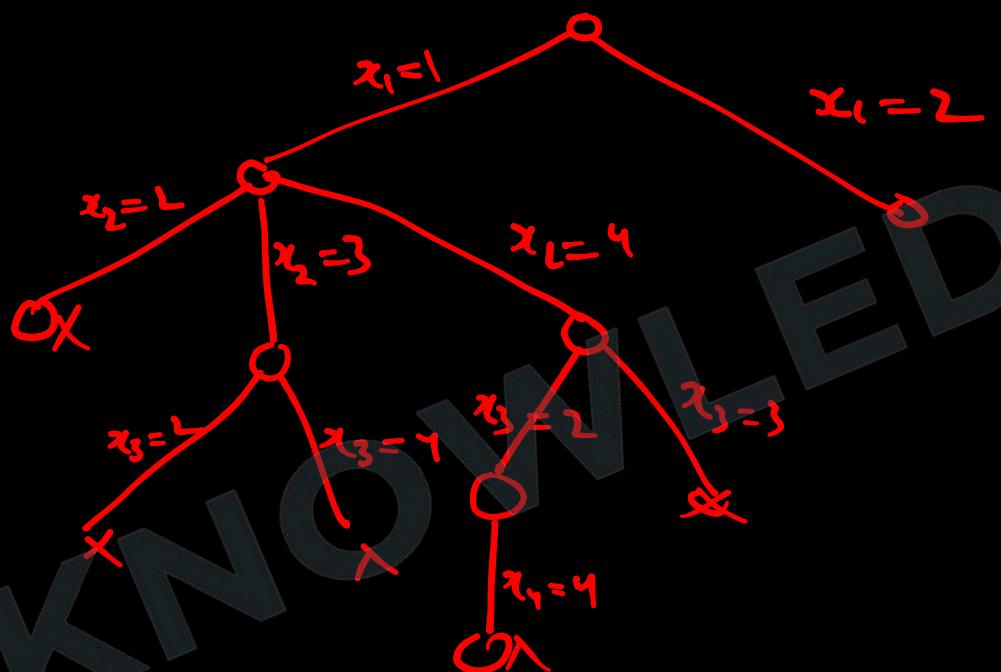


Graph colouring



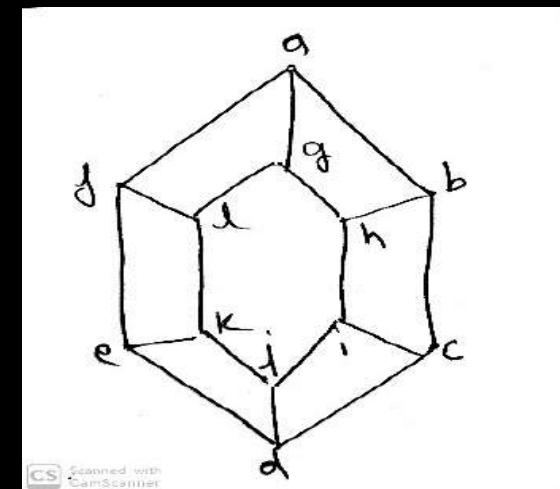
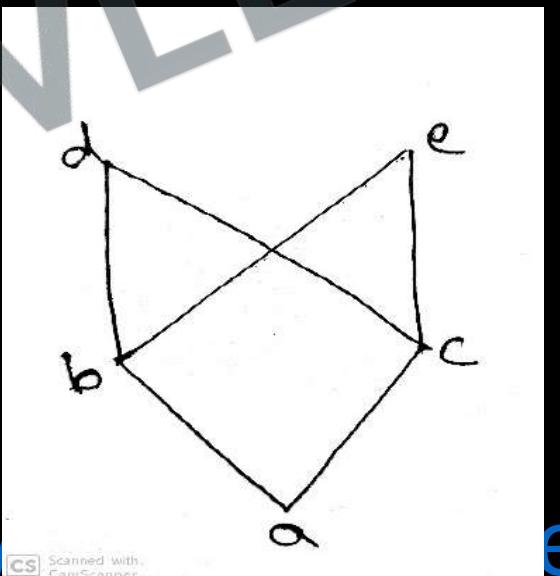
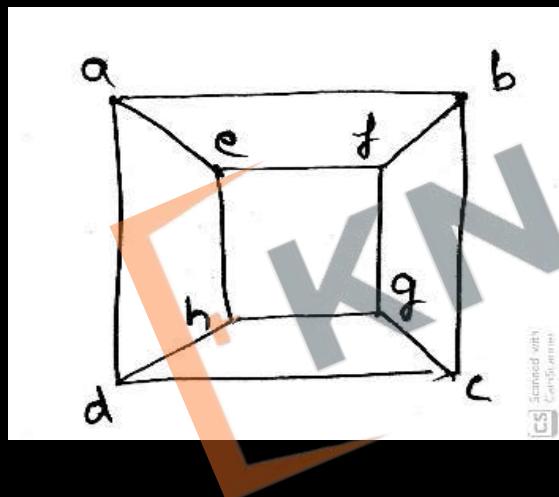
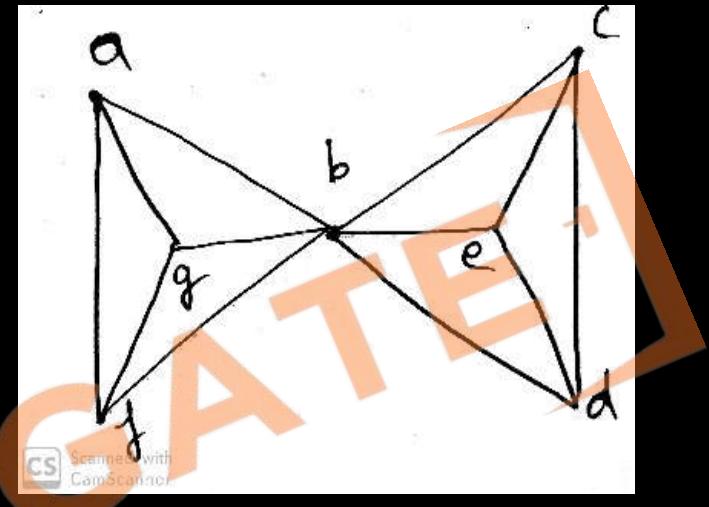
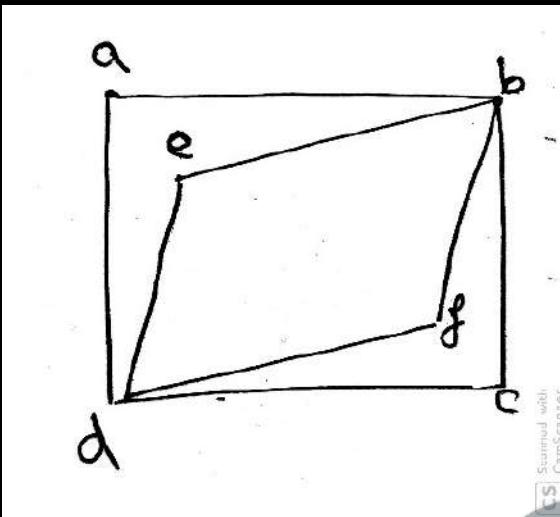
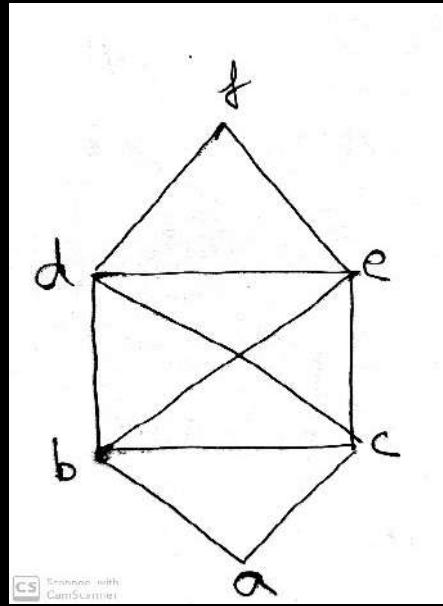
N-Queens problem

- The N-Queens problem is a well-known puzzle in computer science where the aim is to place N chess queens on an $N \times N$ chessboard without any queen threatening another. This means no two queens can be in the same row, column, or diagonal.
 - First queen will in first row, second queen in second row and so on and so for.

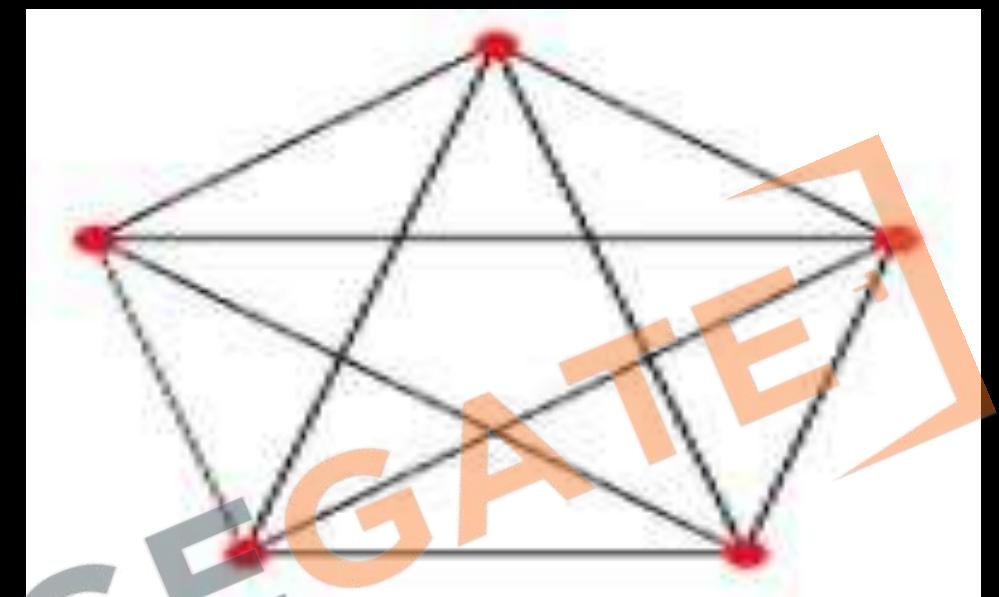


Hamiltonian

1. **Hamiltonian Graph:** - A Hamiltonian circuit in a connected graph is defined as a closed walk that traverses every vertex of G exactly once, except of course the starting vertex, at which the walk also terminates. A graph containing Hamiltonian circuit is called Hamiltonian graph.
2. Finding weather a graph is Hamiltonian or not is a NPC problem.



- **Start at a Vertex**: Begin at any vertex in the graph.
- **Explore Paths**: Choose a neighbor that has not been visited and add it to the current path.
- **Check for Cycle**: If all vertices are included in the path and there is an edge from the last vertex to the first, a Hamiltonian cycle has been found.
- **Backtrack if Stuck**: If you reach a vertex that has no unvisited neighbouring vertices, or you can't form a cycle, backtrack to the previous vertex and try a different path.
- **Repeat**: Continue this process, exploring edges, backtracking when necessary, and checking for cycles until all possibilities are exhausted.



Aspect	Backtracking	Branch and Bound
Approach	Systematic search for solution by trying and eliminating possibilities.	Systematic search for solution using optimization and partitioning.
Goal	To find a feasible solution that satisfies all constraints.	To find the optimal solution with respect to a given objective.
Technique	Removes candidates that fail to satisfy the constraints of the problem.	Uses bounds to estimate the optimality of partial solutions.
State Space Tree	Pruning is used to chop off branches that cannot possibly lead to a solution.	Branches are pruned based on bounds of the cost function (lower and upper bounds).
Application	Suitable for decision problems like puzzles.	Used for optimization problems like integer programming.

Ch-8

Advanced Data Structures

Red-Black Trees, B – Trees, Binomial

Heaps, Fibonacci Heaps, Tries, Skip

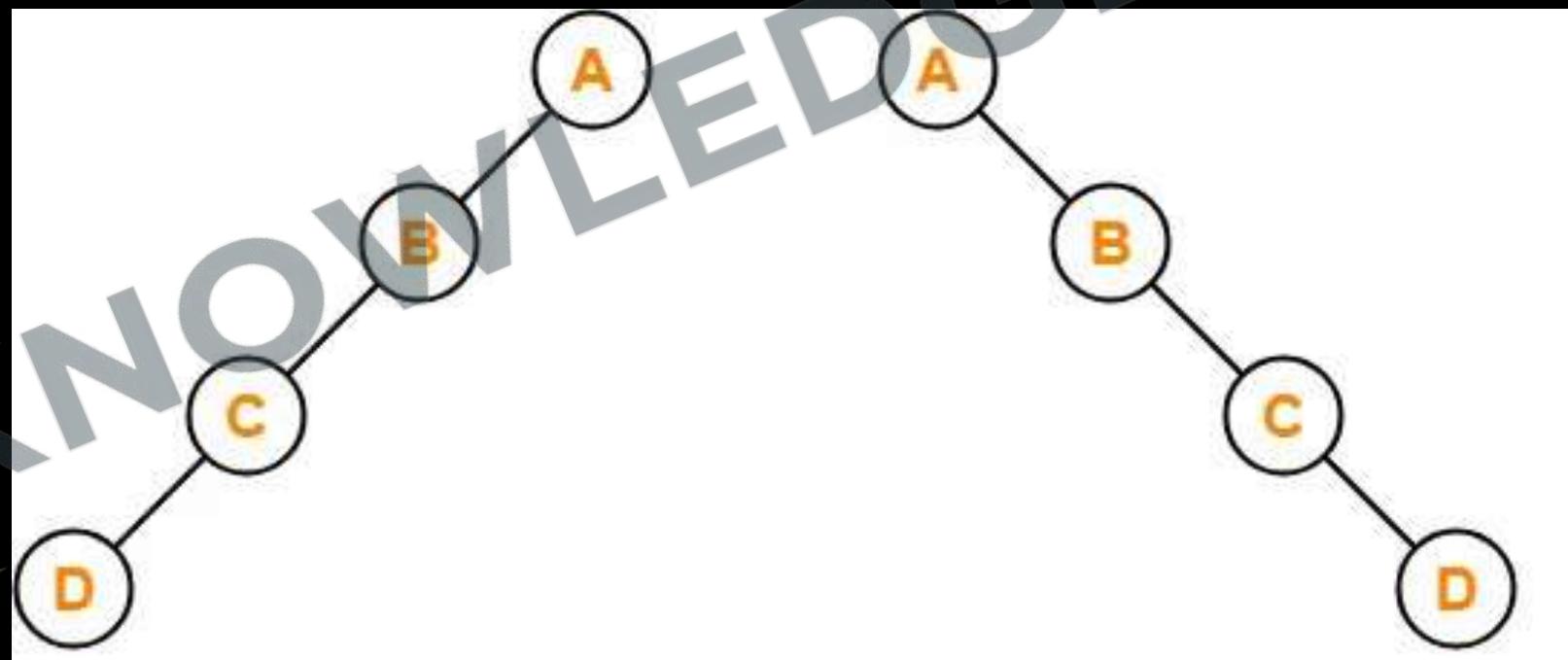
List, Introduction to Activity Networks

Connected Component.

<http://www.knowledgegate.in/GATE>

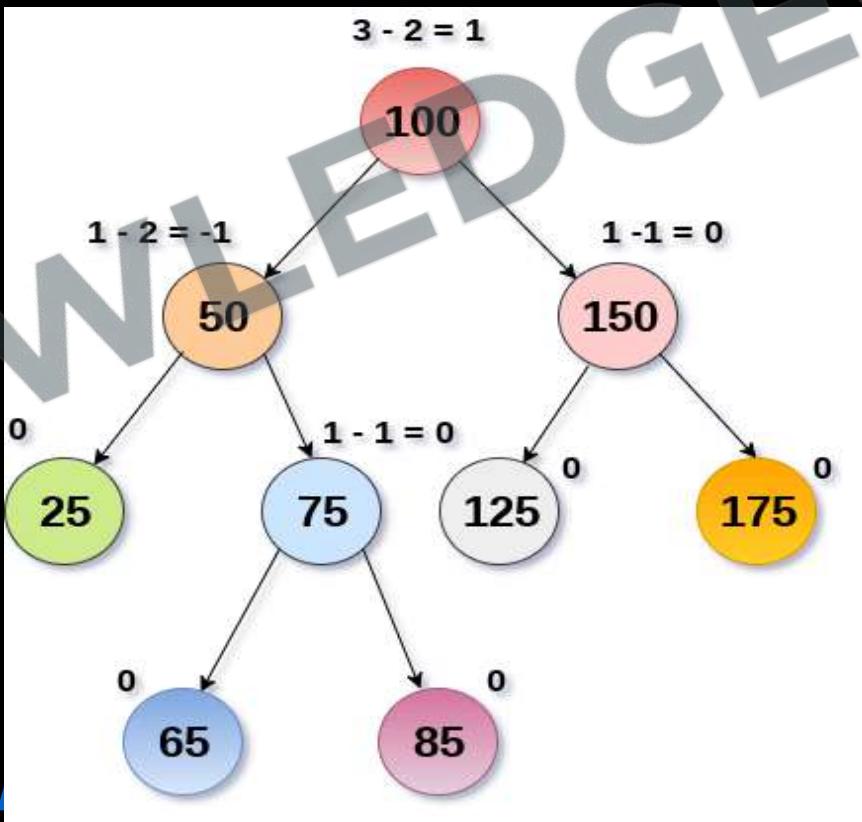
Red-Black Trees

- **Binary Search Trees (BSTs):** The concept of binary search trees has been around informally for a long time, but the binary tree data structure was first structured and named in computer science literature by P.F. Windley, A.D. Booth, A.J.T. Colin, and T.N. Hibbard between 1955 and 1962.
- May be Left-skewed or Right-skewed leading to $O(n)$ worst case time complexity.



Red-Black Trees

- **AVL Trees:** AVL Trees, named after their inventors Georgy Adelson-Velsky and Evgenii Landis, were introduced in 1962. They were the first dynamically balanced trees to be proposed.
- No possibility of being Left-skewed or Right-skewed leading to $O(\log_2 n)$ worst case time complexity.



Red-Black Trees

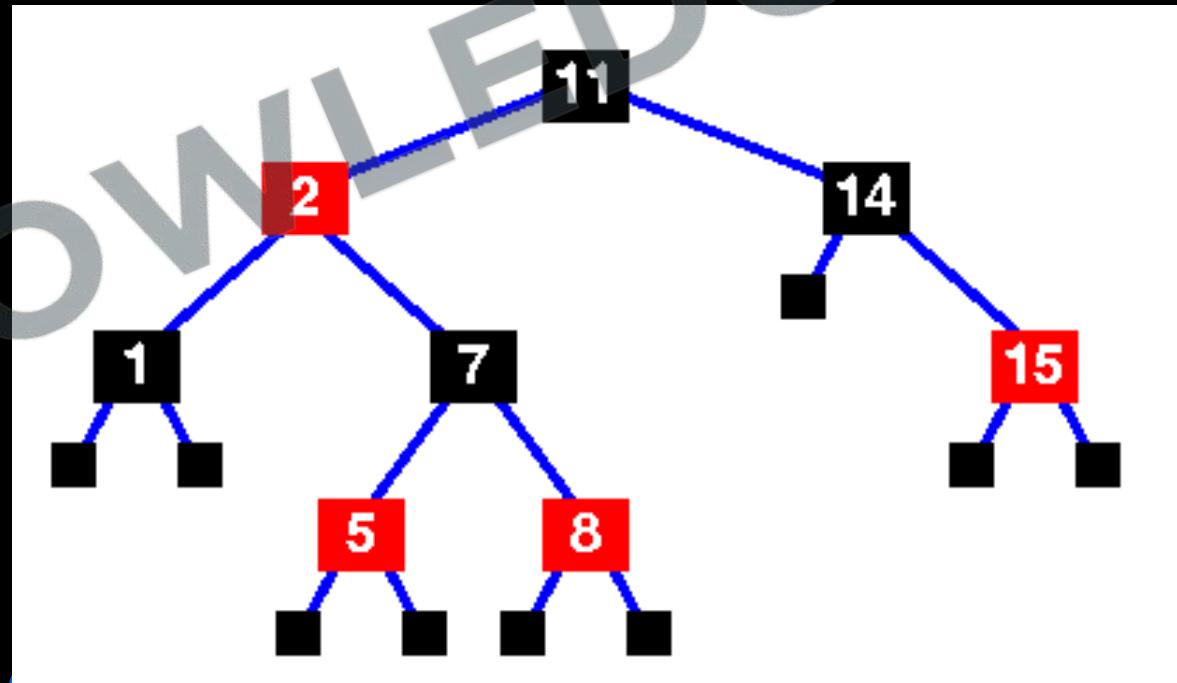
- AVL trees and Red-Black Trees are both self-balancing binary search trees, but there are certain issues with AVL trees that Red-Black Trees address more effectively:
- **Rotations for Balancing:**
 - **AVL Trees:** AVL trees maintain a stricter balance, requiring the heights of two child subtrees of any node to differ by at most one. This can lead to more frequent and potentially complex rotations upon insertions and deletions to maintain the strict height balance.
 - **Red-Black Trees:** Red-Black Trees are more permissive with balance, allowing a greater difference in black heights, which typically leads to fewer rotations needed after insertions and deletions.
- **Use Cases:**
 - **AVL Trees:** AVL trees are more suitable for lookup-intensive applications because they are more strictly balanced and therefore have better search times.
 - **Red-Black Trees:** Red-Black Trees, being less strictly balanced, offer better performance for insertions and deletions and are therefore more suitable for applications where the tree is frequently modified.
- **Algorithmic Complexity:**
 - Both AVL and Red-Black Trees offer $O(\log n)$ search, insertion, and deletion complexities. However, the constants involved in these operations for AVL trees may be higher due to the need for more rotations.

Red-Black Trees

- AVL Trees Example:
 - Database Indexing: Used in databases for efficient search operations, especially when reads are more frequent than writes. For instance, in a library's user database, AVL trees can optimize user data retrieval based on sorted user IDs.
- Red-Black Trees Example:
 - Programming Libraries: Common in standard libraries like C++ STL for implementing std::map and std::set. Suitable for scenarios with frequent additions and deletions, such as maintaining a sorted list of users in real-time applications.

Red-Black Trees

- A Red-Black Tree is a type of self-balancing binary search tree, where each node has an extra attribute: the color, which is either red or black. Here are the defining properties that every Red-Black Tree adheres to:
 - **Node Color**: Every node is colored, either red or black.
 - **Root Property**: The root of the tree is always black.
 - **Leaf Nodes**: Every leaf (NIL node, an empty/black node) is black.
 - **Red Node Rule**: No two red nodes may be adjacent; a red node cannot have a red parent or red child.
 - **Black Height Consistency**: Every path from a given node to any of its descendant NIL nodes must have the same number of black nodes. This is known as the "black-height".



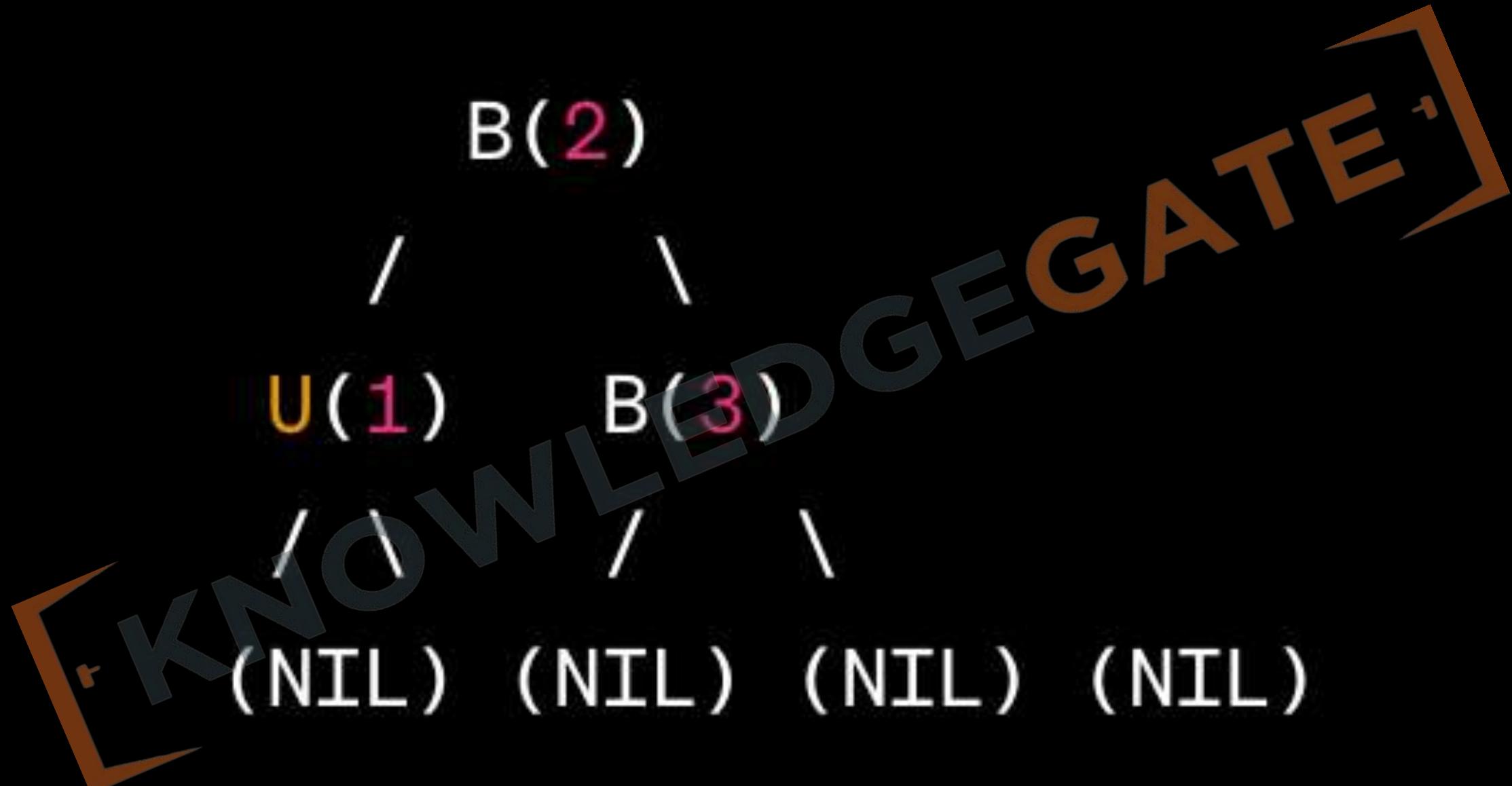
[http:](http://)

- These properties ensure that the tree remains balanced, with the longest path from the root to the farthest leaf not more than twice as long as the shortest path from the root to the nearest leaf.



<http://www.knowledgegate.in/GATE>

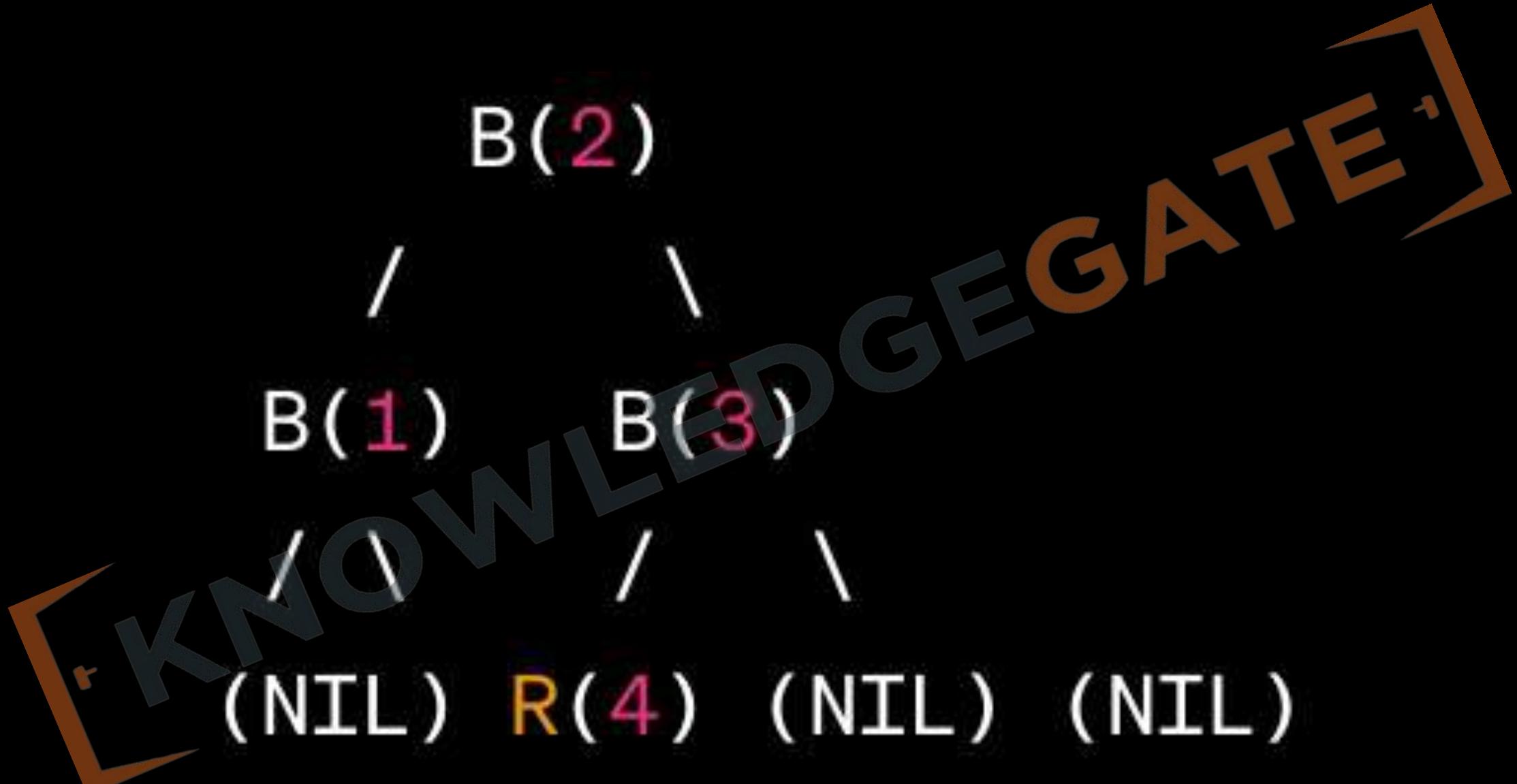
Unspecified Node Color (Violation: Node Color)



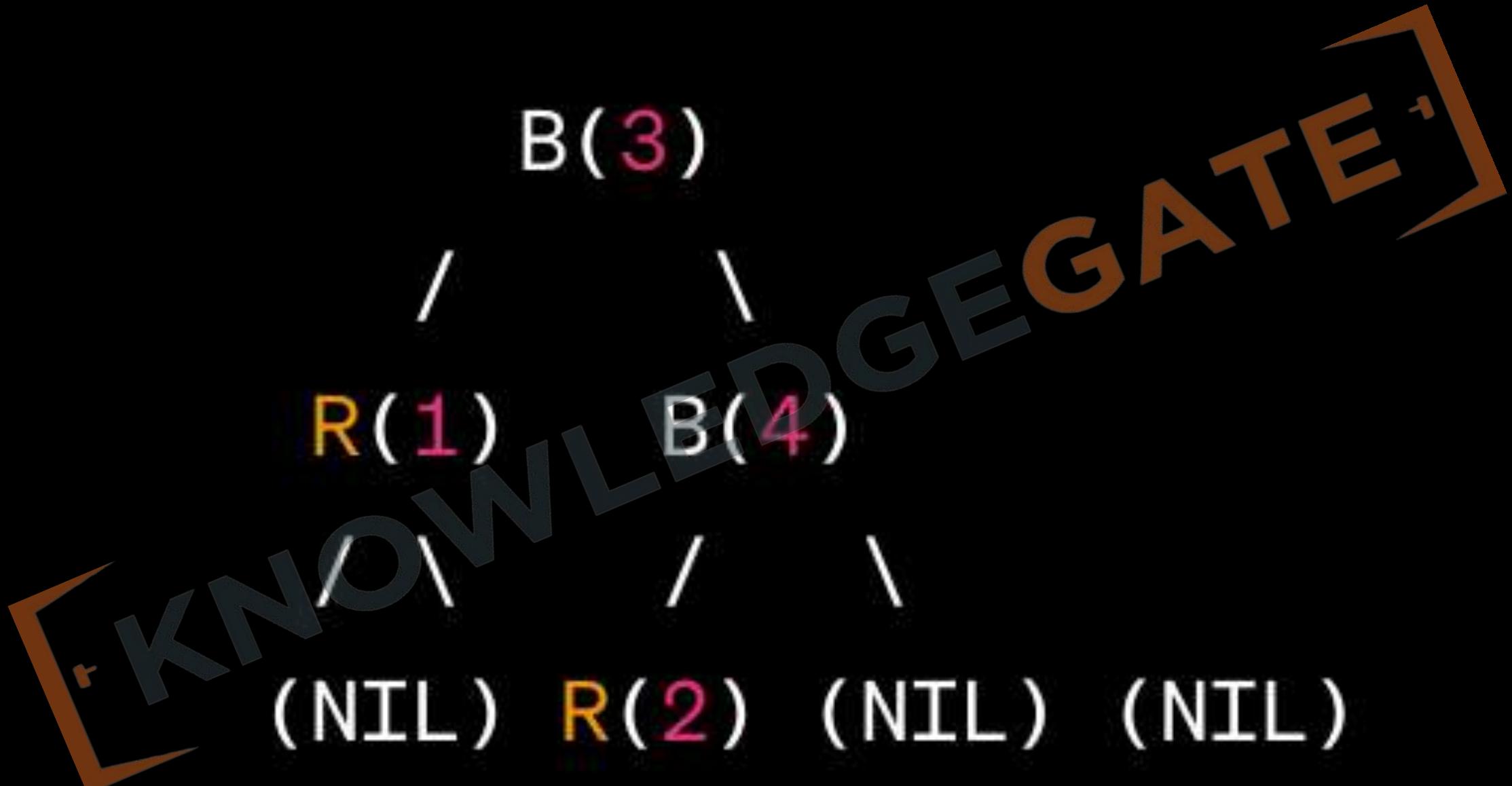
Red Root Node (Violation: Root Property)



Non-Black Leaf Node (Violation: Leaf Nodes)



Consecutive Red Nodes (Violation: Red Node Rule)

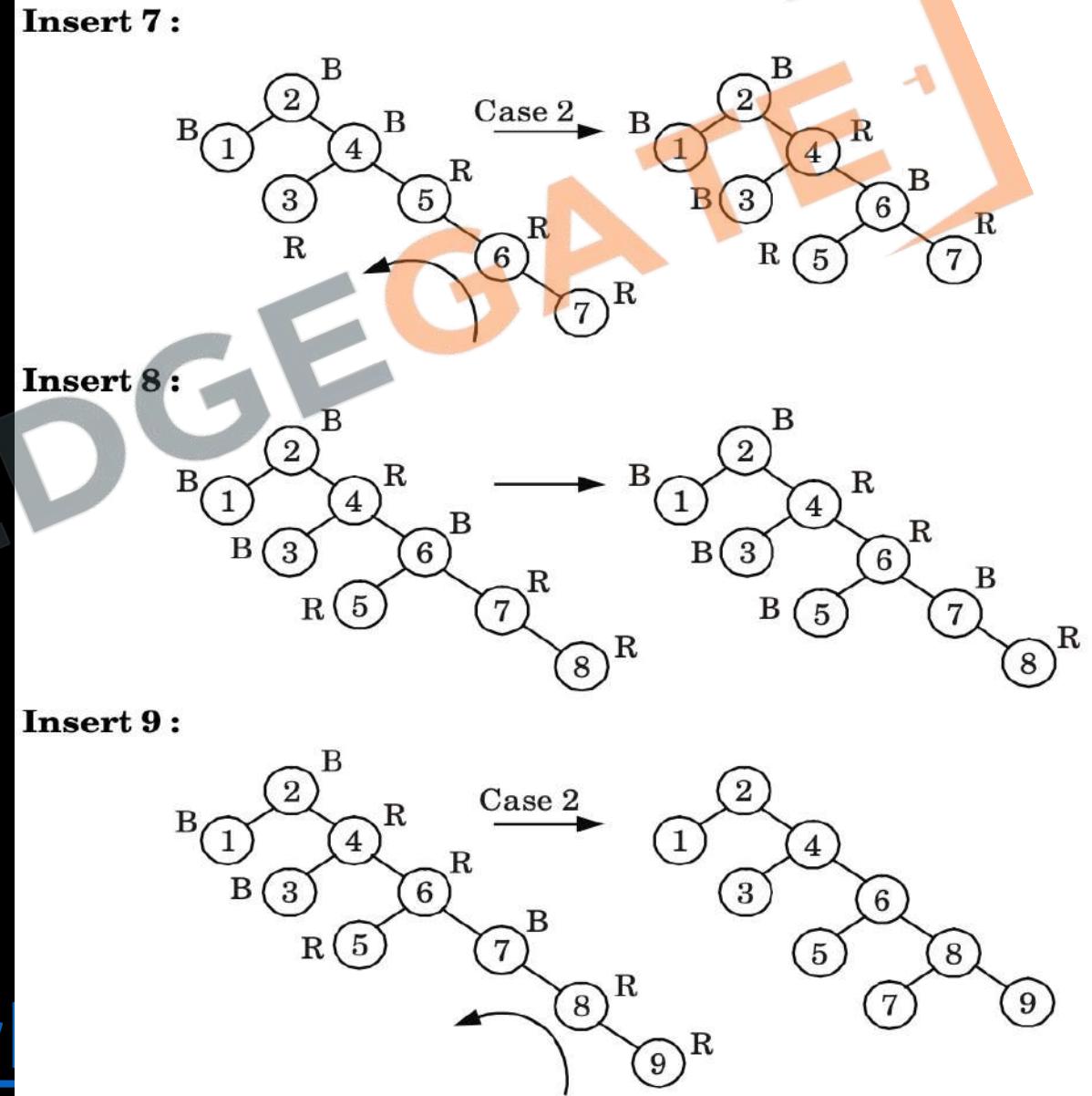
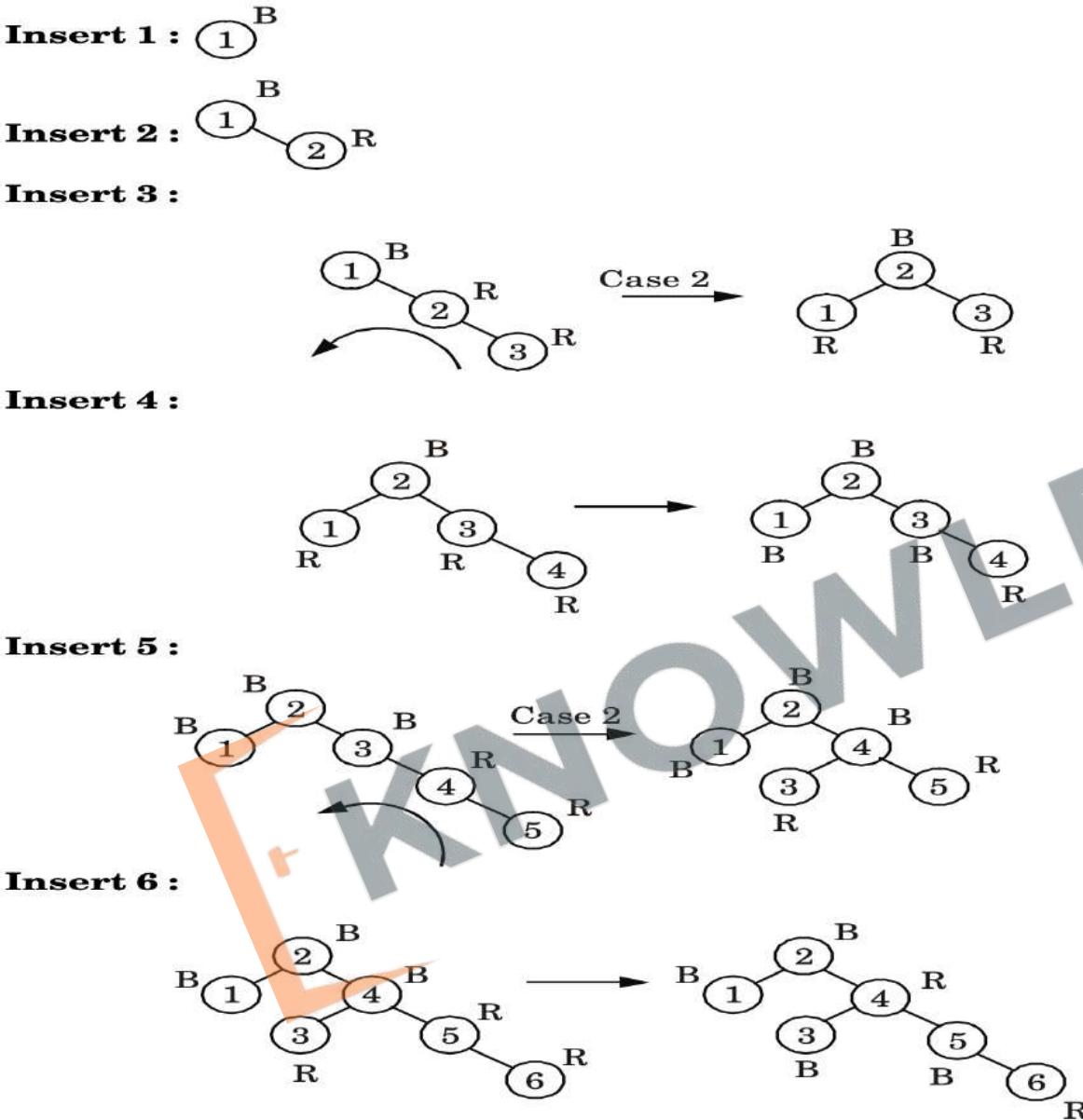


Unspecified Node Color (Violation: Node Color)

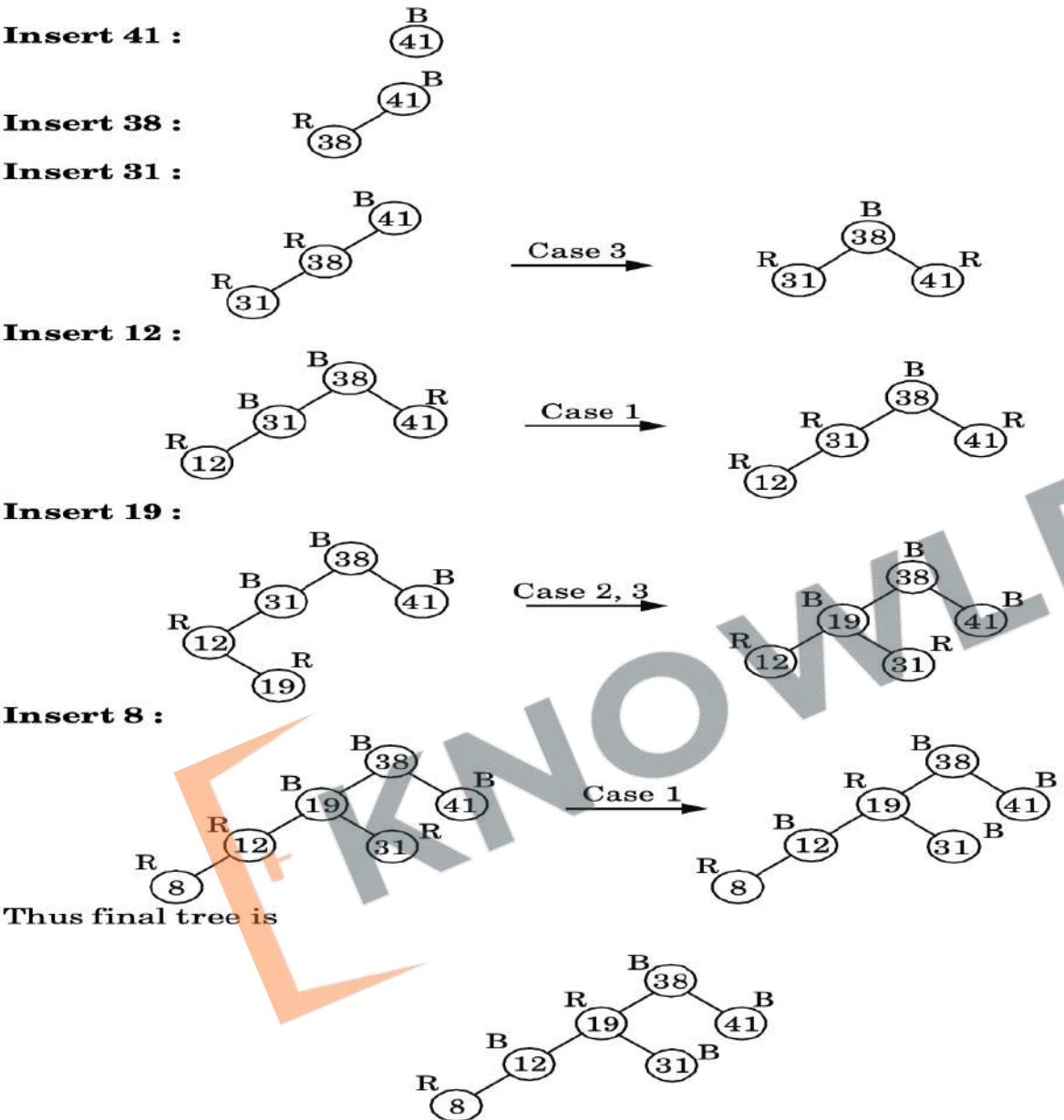


- If tree is empty then create newnode as root node with color black.
- If tree is not empty then insert newnode with color red.
- If the parent of the new node is black, the properties are preserved.
- If the parent of newnode is red, then check the color of the sibling of the new node's parent
 - **Case 1: Uncle is Red (Recoloring needed):**
 - Recolor the parent and uncle to black.
 - Recolor the grandparent to red.
 - Set the grandparent as the new node to be examined and repeat the validation process.
 - **Case 2: Uncle is Black, or null**
 - Perform a rotation on the grandparent and Recolor .
 - former parent to black and the grandparent to red(in LL or RR rotations).
 - In LR or RL rotation re color accordingly
- **Adjust the Root:**
 - If rotations have been performed, the root node might have changed. Ensure the root is still black.
- **Continue Until the Tree is Fixed:**
 - Continue the validation and re-balancing process until all Red-Black Tree properties are satisfied

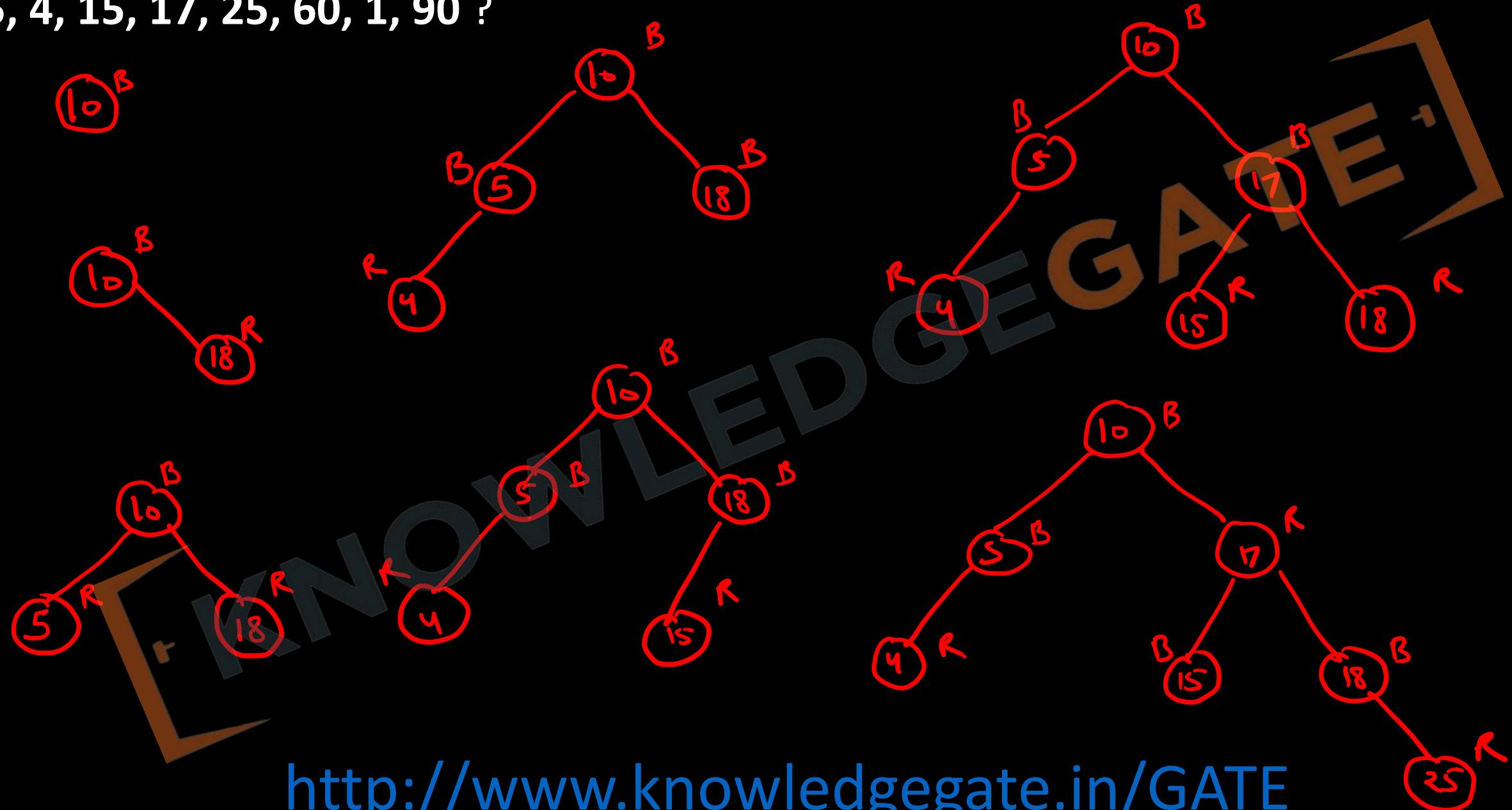
Q Insert the following sequence of information in an empty red-black tree 1, 2, 3, 4, 5, 6, 7, 8 and 9 ?



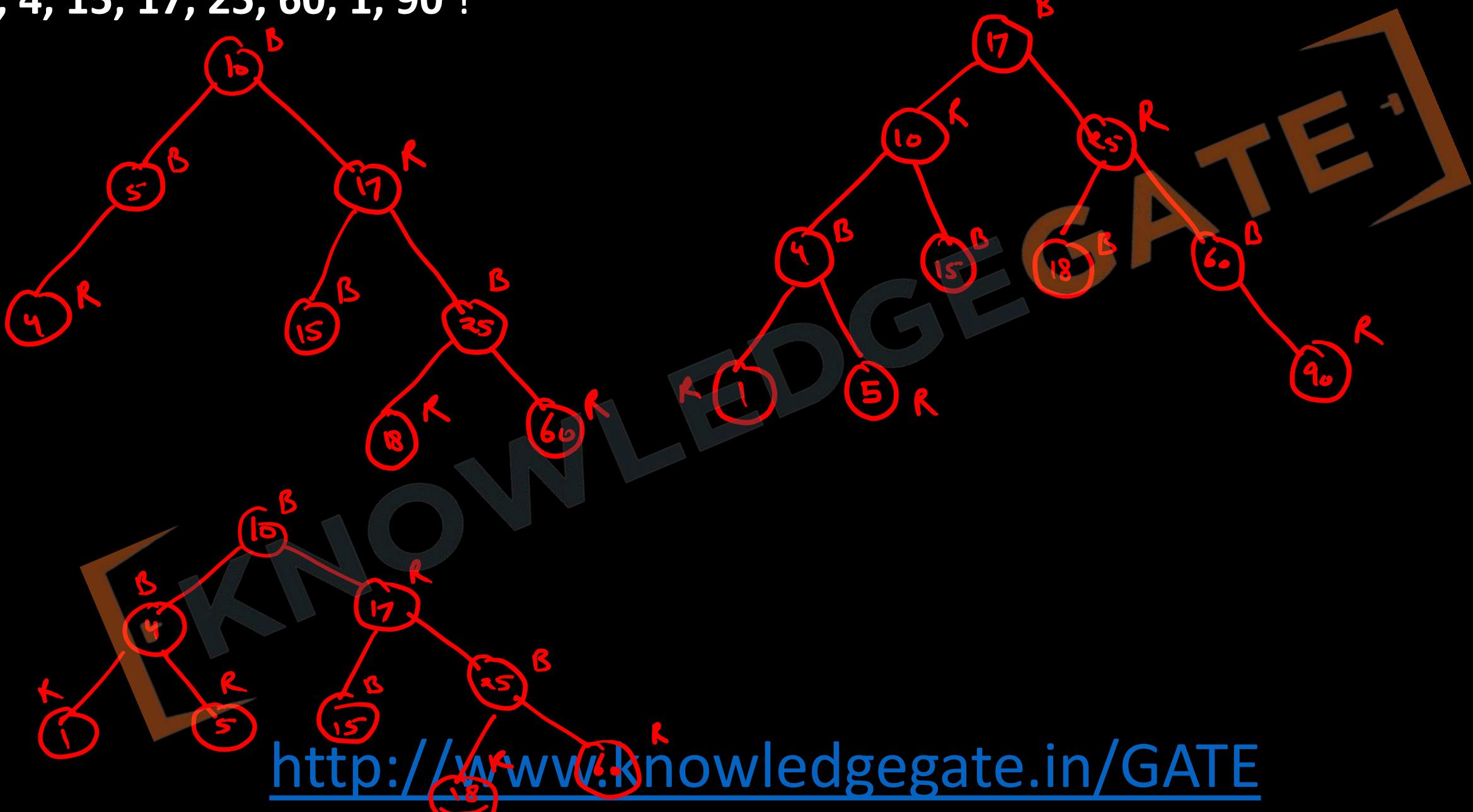
Q Insert the following sequence of information in an empty red-black tree 41, 38, 31, 12, 19, 8 ?



Q Insert the following sequence of information in an empty red-black tree 10, 18, 5, 4, 15, 17, 25, 60, 1, 90 ?

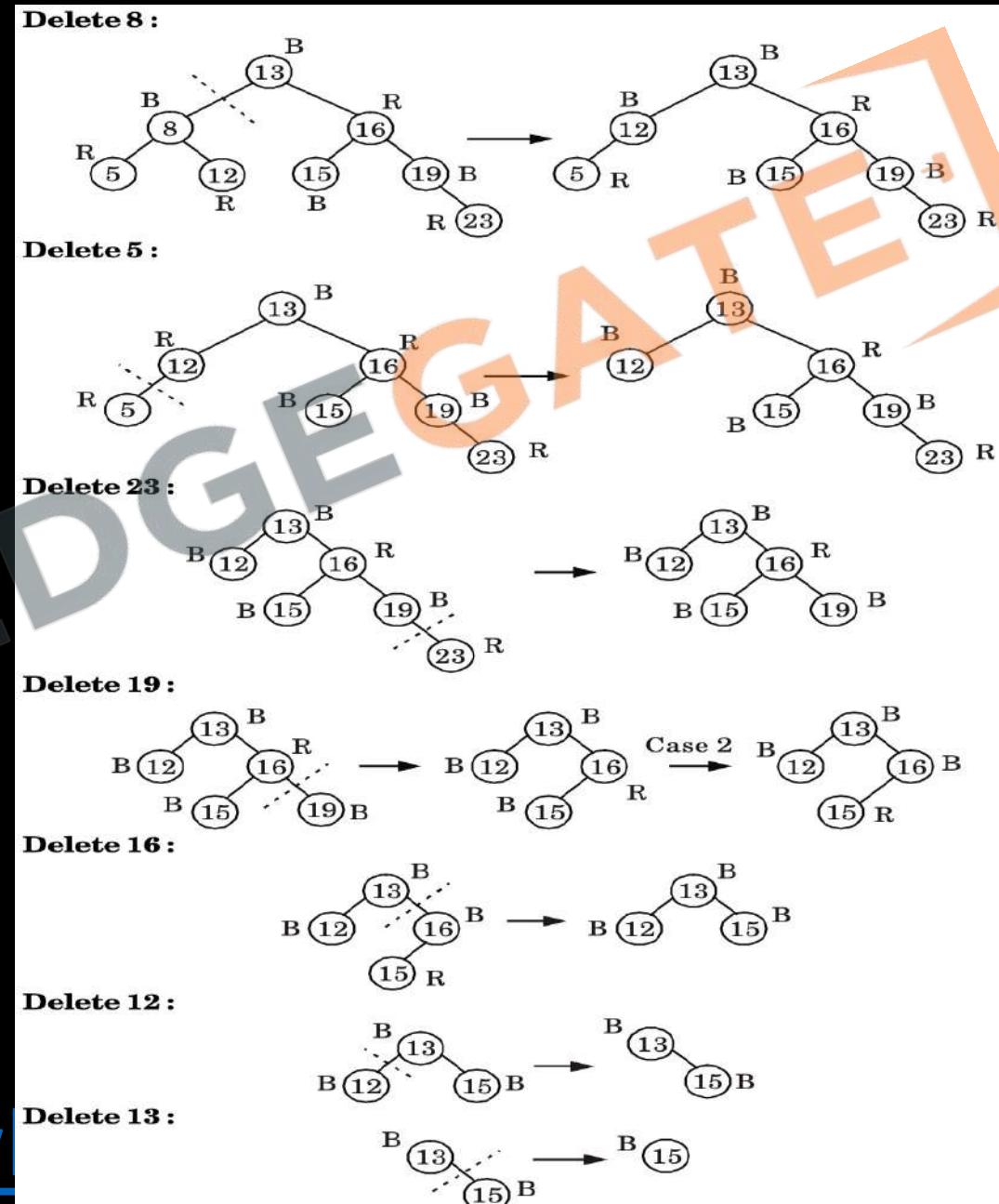
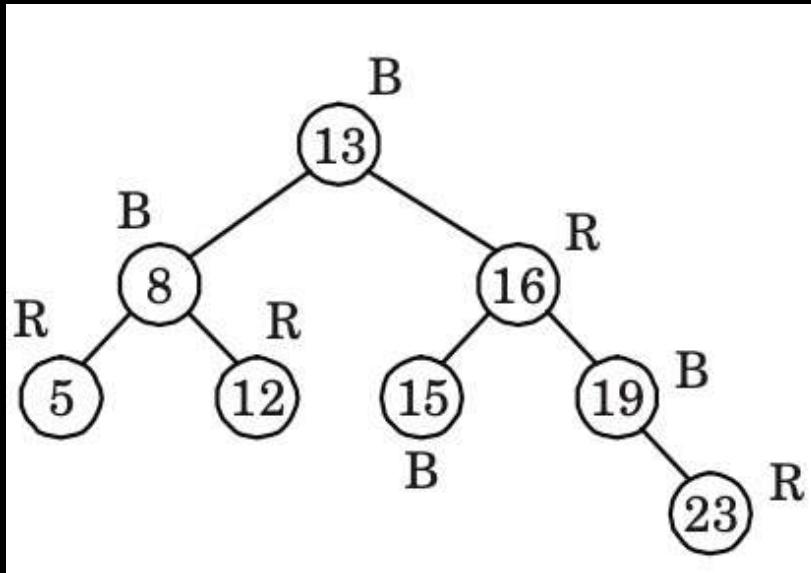


Q Insert the following sequence of information in an empty red-black tree 10, 18, 5, 4, 15, 17, 25, 60, 1, 90 ?

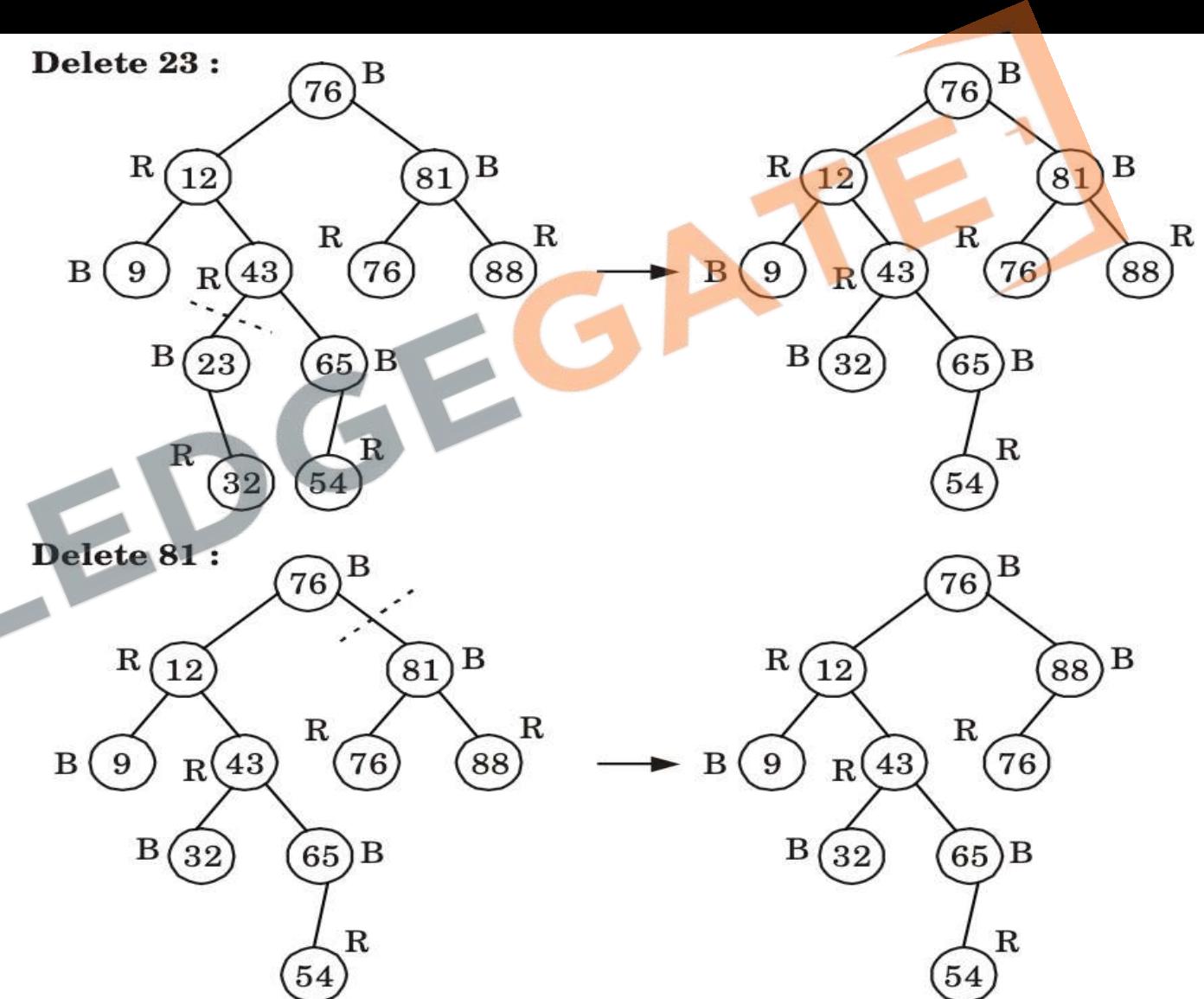
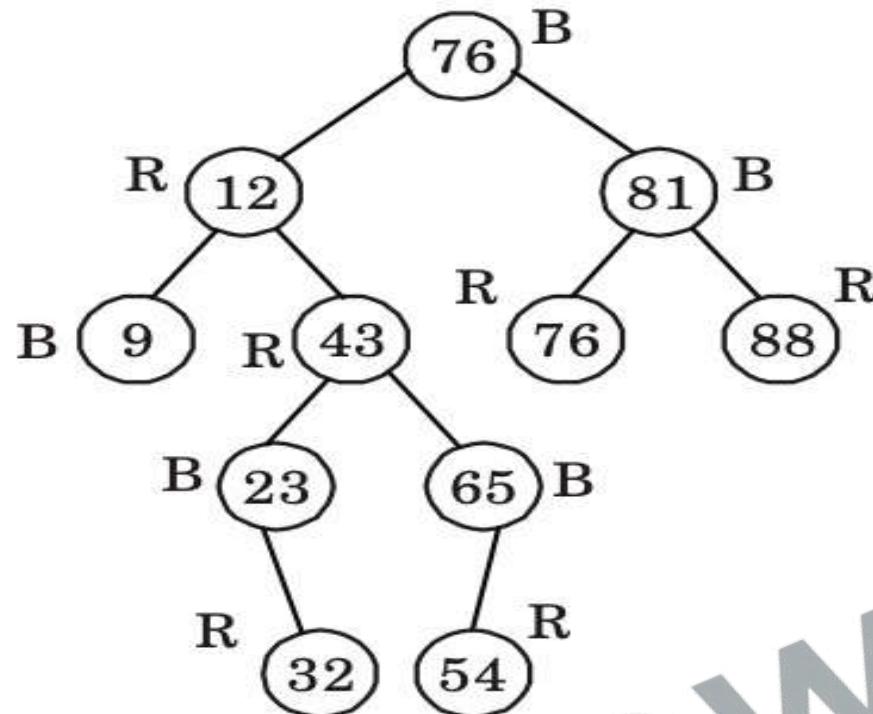


- Deletion in a Red-Black Tree involves several steps and can be more complex than insertion due to the need to maintain the red-black properties post-deletion. Here's a simplified step-by-step process for deleting a node from a Red-Black Tree:
- **Find the Node:** Locate the node you wish to delete using standard binary search tree deletion steps. If the node has two children, find the in-order successor (or predecessor) and swap its value with the node to delete.
- **Remove the Node:** Delete the node from the tree. There are three possible scenarios:
 - Deleting a node with no children: simply remove the node.
 - Deleting a node with one child: replace the node with its child.
 - Deleting a node with two children: replace the node with its in-order successor (or predecessor), which will have at most one child, and then delete the successor.
- **Fixing Double Black Issue:**
 - If you removed a red node, the properties still hold. If a black node was removed, this creates a "double black" issue, where one path has one fewer black node.
 - The node that replaces the deleted node (if any) is marked as "double black," which means it either has an extra black than it should, or it's black and has taken the place of a removed black node.
- **Rebalance the Tree:** To resolve the "double black" issue, several cases need to be handled until the extra blackness is moved up to the root (which can then be discarded) or until re-balancing is done:
 - **Case 1: Sibling is red** - Perform rotations to change the structure such that the sibling becomes black, and then proceed to other cases.
 - **Case 2: Sibling is black with two black children** - Repaint the sibling red and move the double black up to the parent.
 - **Case 3: Sibling is black with at least one red child** - Perform rotations and recoloring so that the red child of the sibling becomes the sibling's parent. This corrects the double black without introducing new problems.
- **Termination:** The process terminates when:
 - The double black node becomes the root (simply remove the extra blackness).
 - The double black node is red (simply repaint it to black).
 - The tree has been adjusted to redistribute the black heights and remove the double blackness.
- **Update Root:** After rotations and recoloring, the root might change, so ensure the root of the tree is still black.

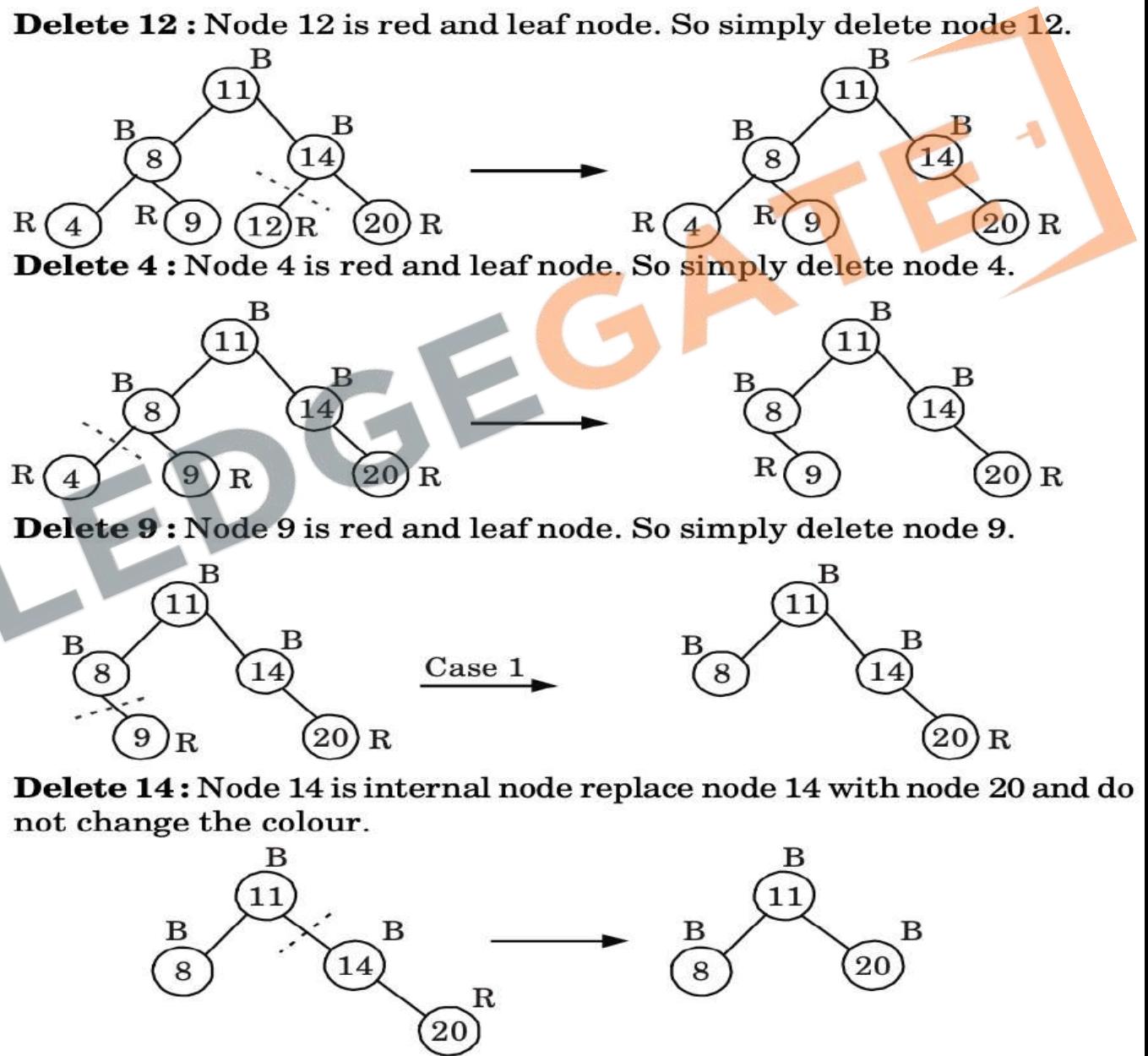
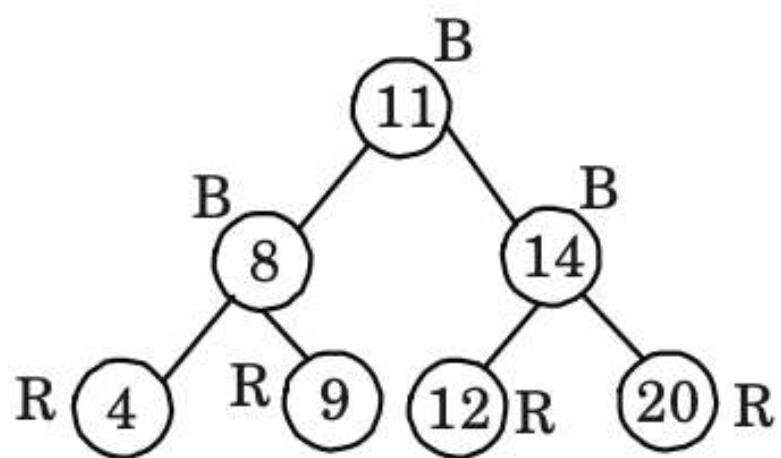
Q Insert the nodes 15, 13, 12, 16, 19, 23, 5, 8 in empty red-black tree and delete in the reverse order of insertion?



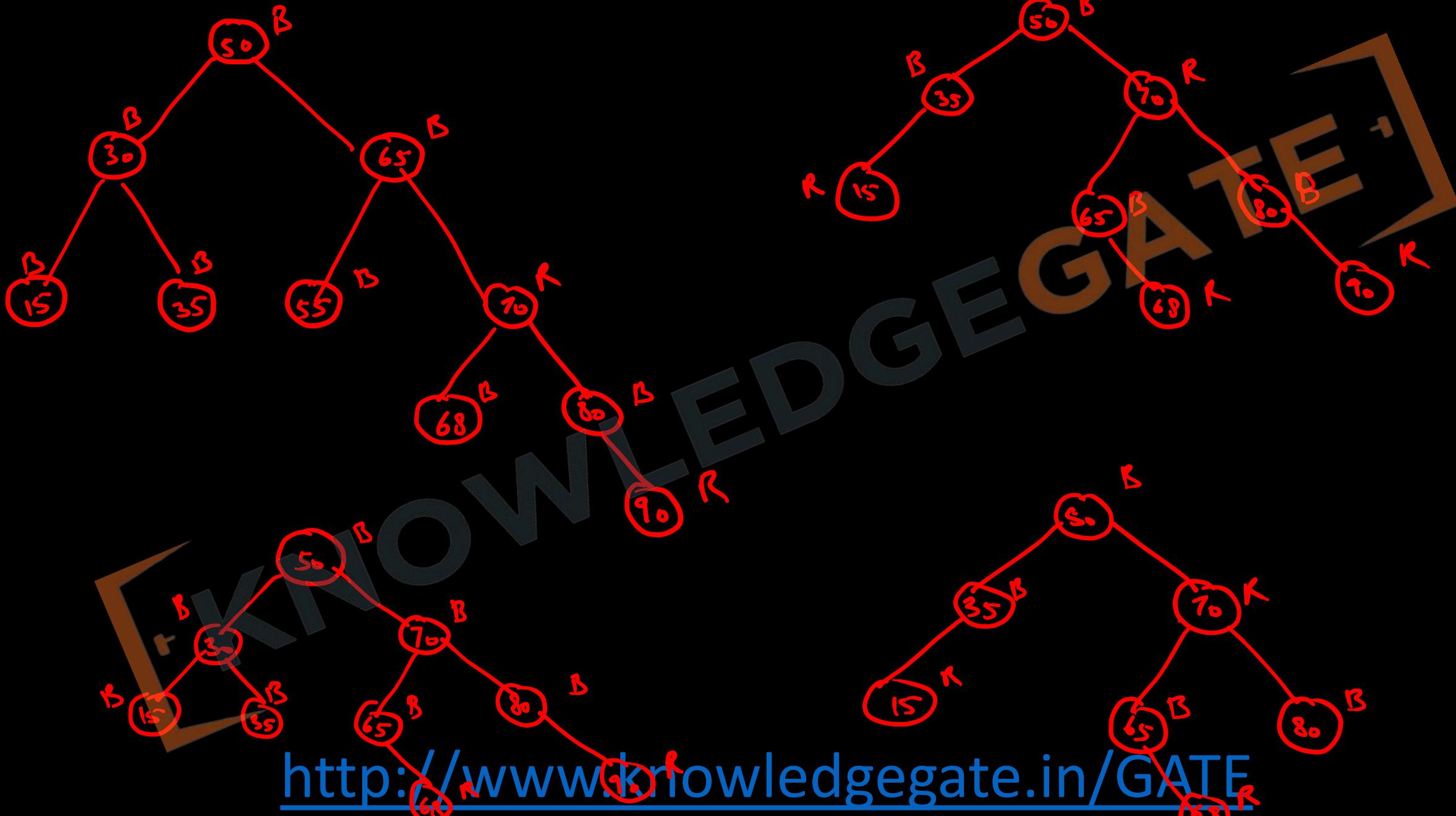
Q Insert the following element in an initially empty RB-Tree 12, 9, 81, 76, 23, 43, 65, 88, 76, 32, 54. Now delete 23 and 81 ?



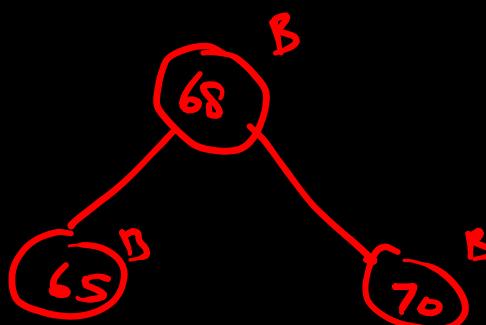
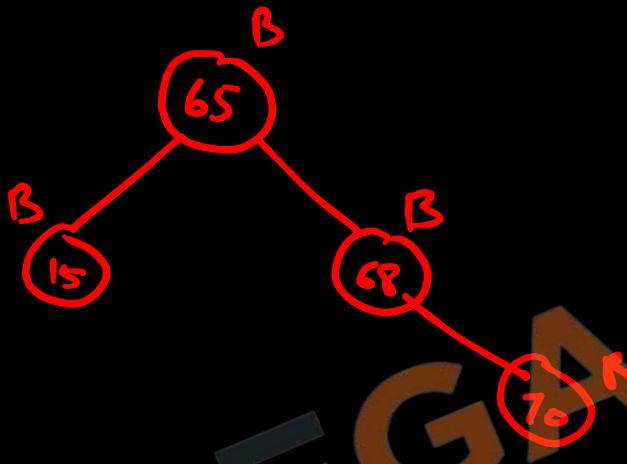
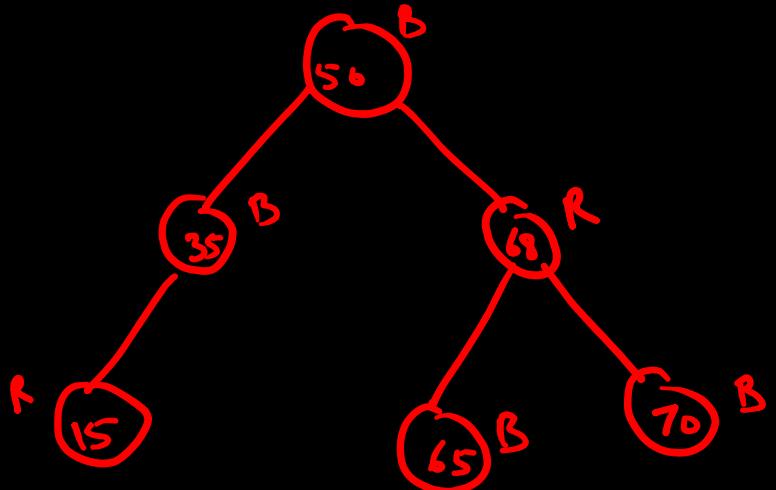
Q Insert the elements 8, 20, 11, 14, 9, 4, 12 in a Red-Black tree and delete 12, 4, 9, 14 respectively ?



Q Consider the following RB Tree and delete 55, 30, 90, 80, 50, 35, 15 respectively ?



Q Consider the following RB Tree and delete 55, 30, 90, 80, 50, 35, 15 respectively ?



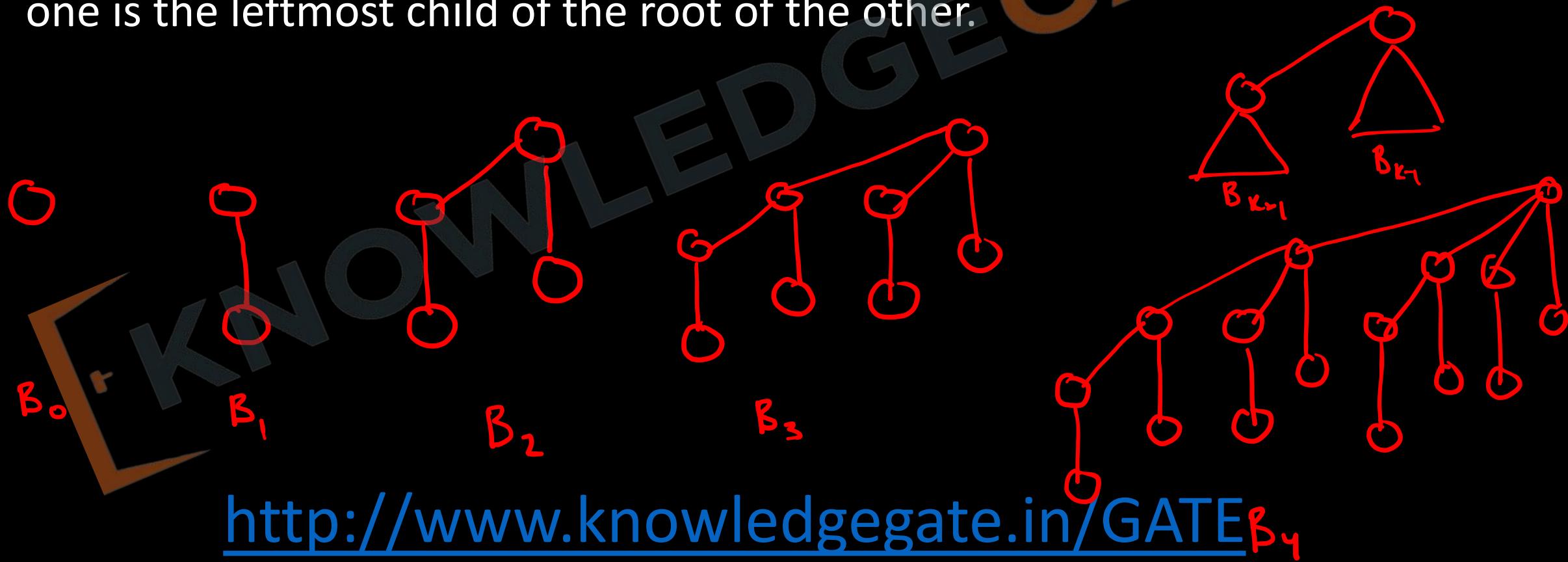
Binomial Tree

- **Development and Context:** Binomial trees as a data structure were developed as part of the broader exploration of efficient data structures for priority queue operations. They were introduced as a key component of binomial heaps.
- **Key Contributors:** The concept of the binomial heap, which uses binomial trees, was introduced by J. W. J. Williams in 1964. Later, it was further developed and popularized by Jean Vuillemin in 1978.
- **Evolution of Heaps:** Prior to binomial heaps, binary heaps were commonly used for priority queues. Binomial trees and heaps were an evolution in this area, offering more efficient solutions for certain operations.



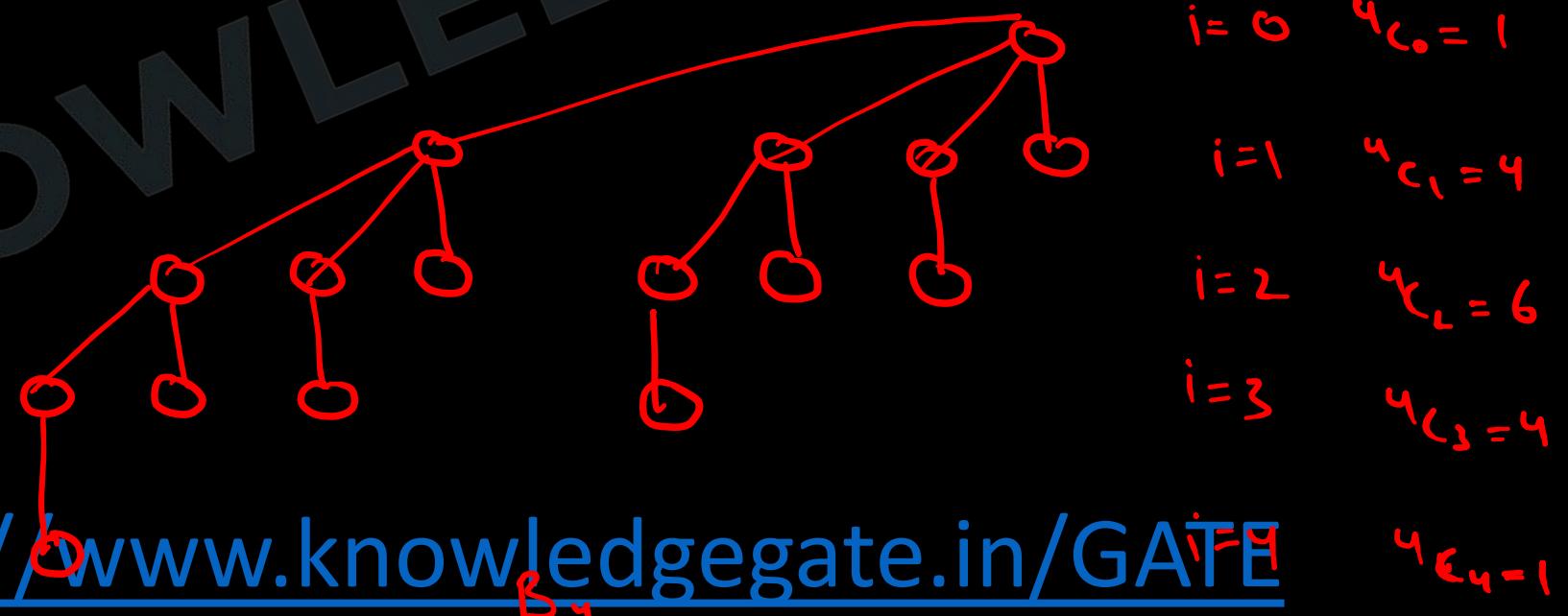
Binomial Tree

- A binomial tree is a specific kind of tree used in certain data structures such as binomial heaps.
- **Recursive Definition:** A binomial tree of order 0 is a single node. A binomial tree of order k consists of two binomial trees of order $k-1$ linked together: one is the leftmost child of the root of the other.



Binomial Tree Properties

- A binomial tree of order k has exactly 2^k nodes. The height of a binomial tree of order k is k .
- The height of a binomial tree B_k is k . That is, the tree has $k+1$ levels, indexed from 0 to k .
- The number of nodes at depth i in a binomial tree of order k is given by the binomial coefficient ${}^k C_i$, $i=0,1,\dots,k$.
- The root of a binomial tree B_k will have a degree k , which will be greater than any other nodes.
- The root node of a binomial tree of order k has a degree of k , and it has k children. The children are themselves root nodes of binomial trees of orders $k-1, k-2, \dots, 0$, respectively.



Binomial Heap

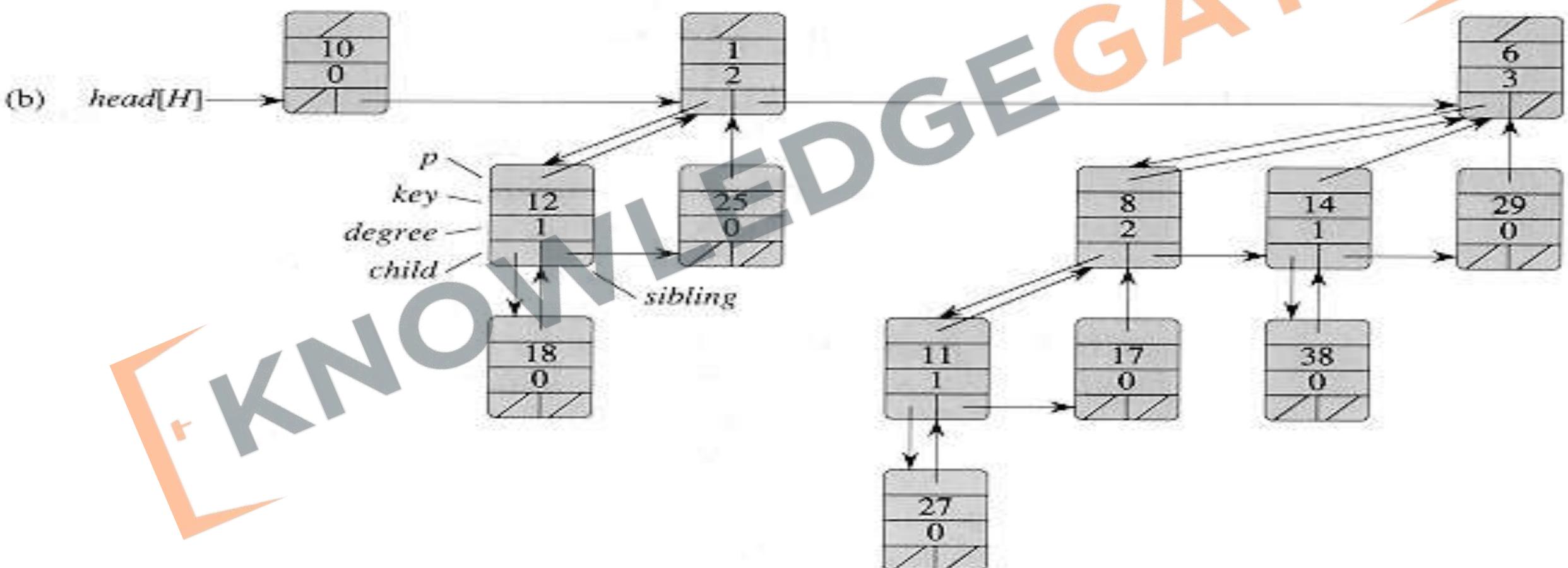
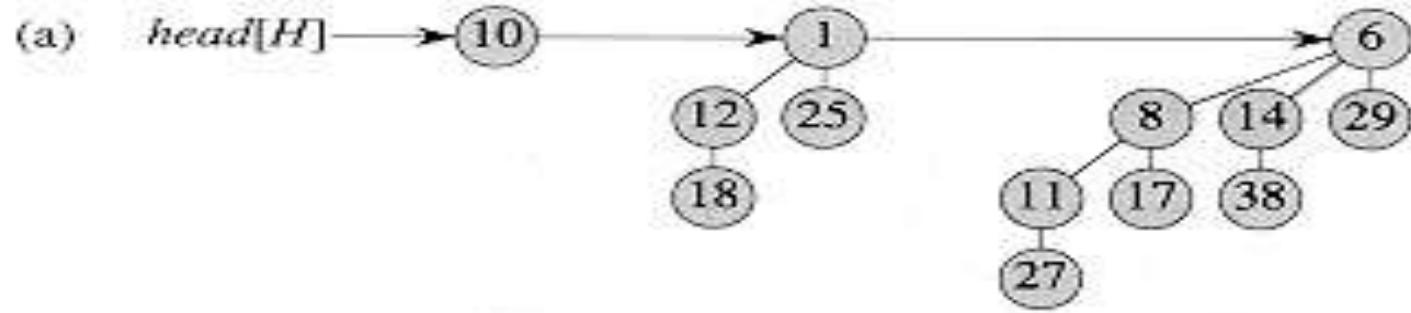
- A binomial heap is implemented as a collection of binomial tree. It keeps data sorted and allows insertion and deletion in amortized time.
- **Properties:**
 - Unlike a binary heap, which is a single tree, a binomial heap is a collection of trees. which are ordered and comply with the min-heap (or max-heap) property.
 - Each binomial tree in a binomial heap has a different order
 - The heap with n elements will have at most $\lceil \log n \rceil + 1$ binomial trees. Because binary representation of n has $\log n + 1$ bits.

$n=15$

1 1 1 1
 $B_3 B_2 B_1 B_0$

$n=9$

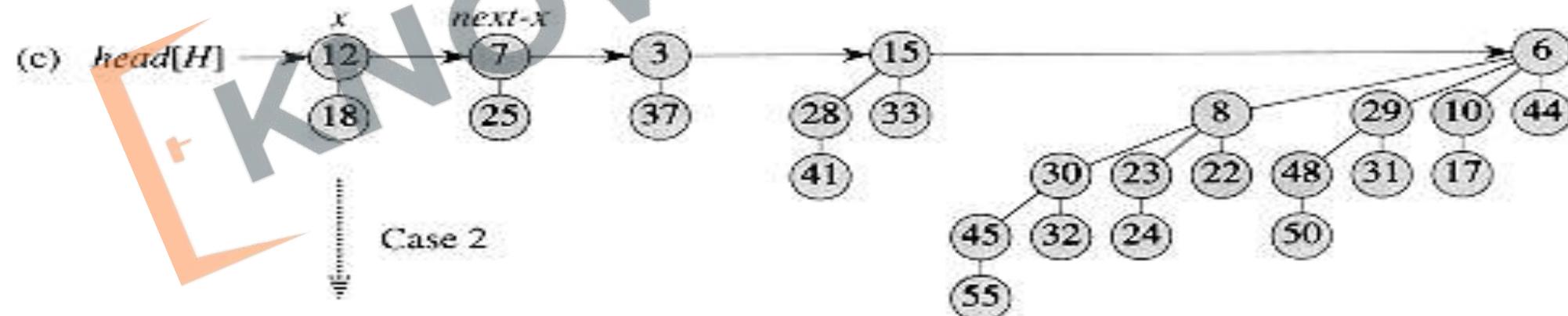
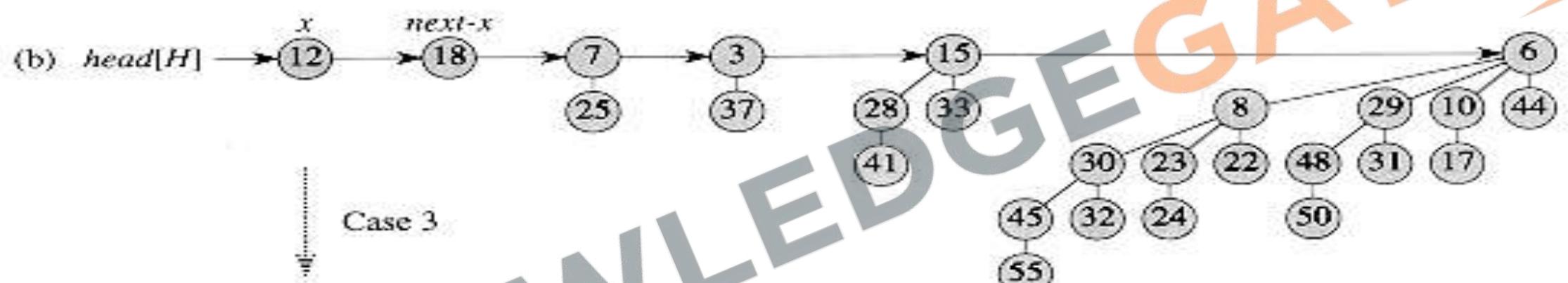
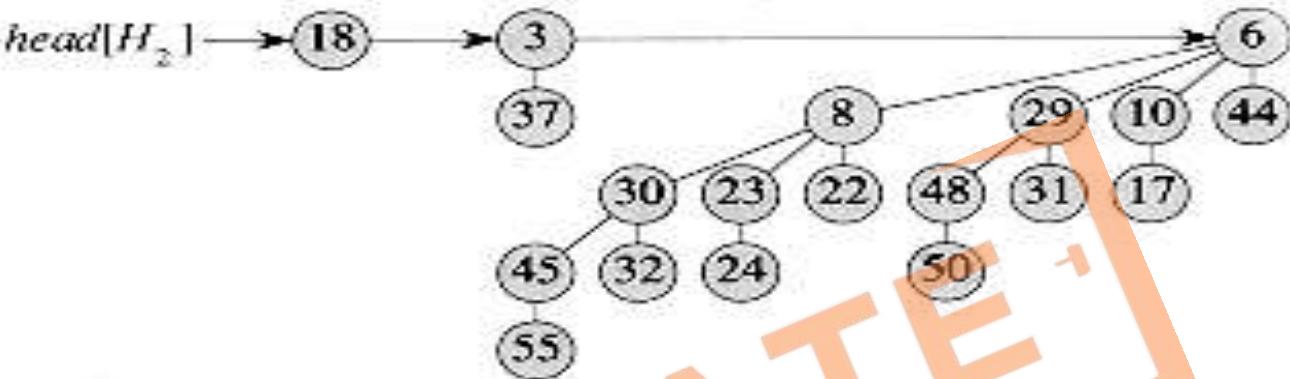
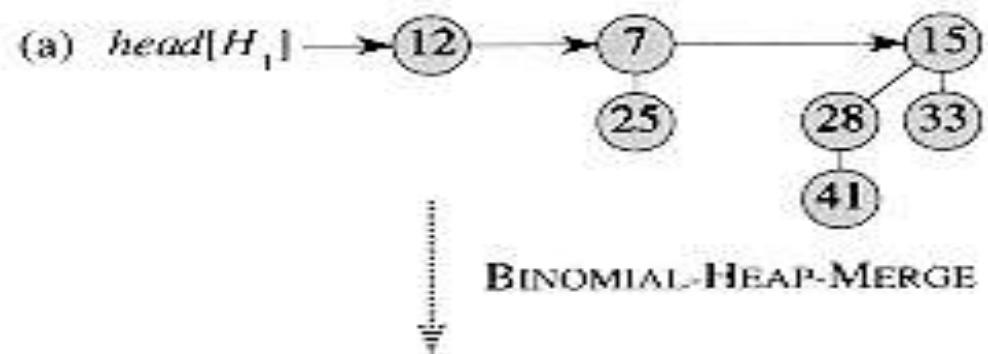
1 0 0 1
 $B_3 X X B_0$

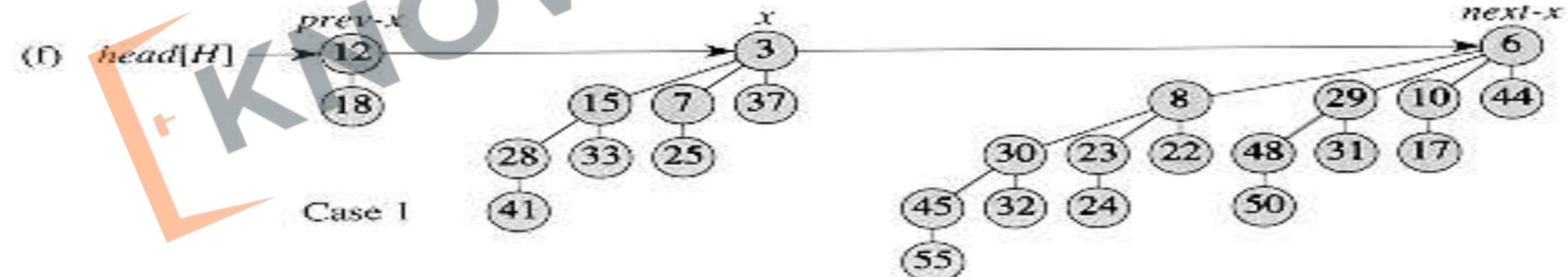
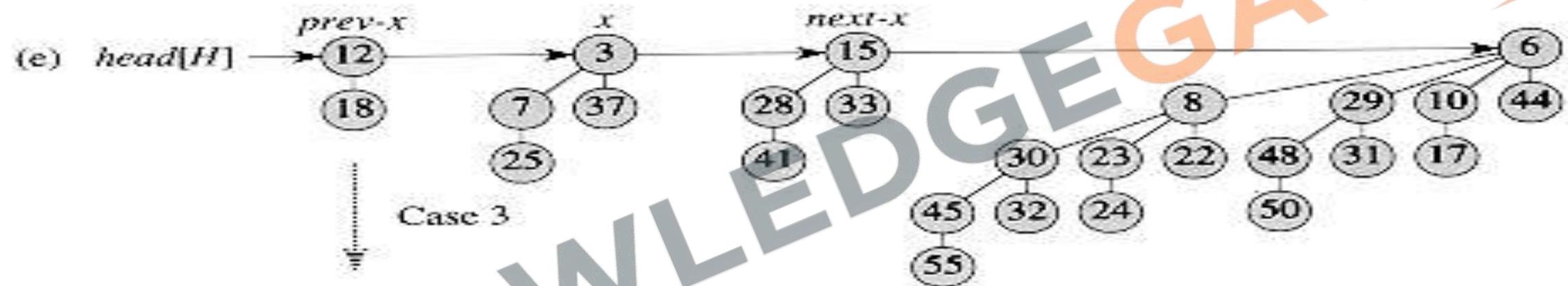
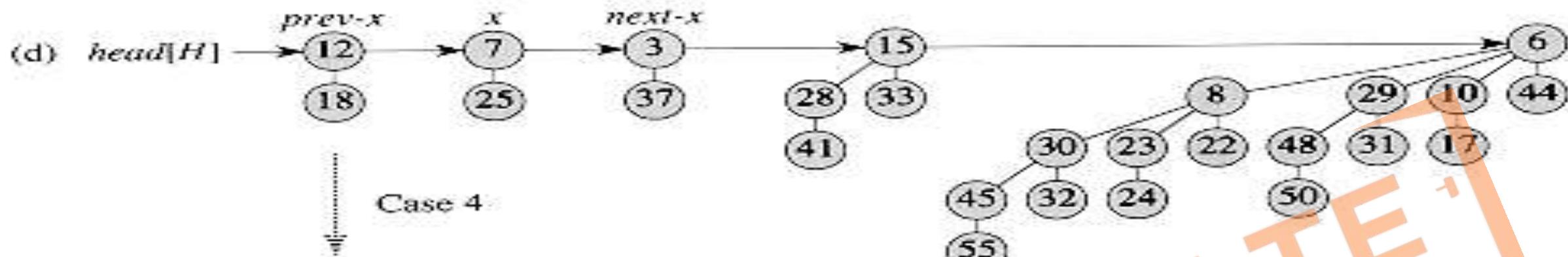


BINOMIAL-HEAP-UNION(H_1, H_2)

```
1   $H \leftarrow \text{MAKE-BINOMIAL-HEAP}()$ 
2   $\text{head}[H] \leftarrow \text{BINOMIAL-HEAP-MERGE}(H_1, H_2)$ 
3  free the objects  $H_1$  and  $H_2$  but not the lists they point to
4  if  $\text{head}[H] = \text{NIL}$ 
5    then return  $H$ 
6   $\text{prev-}x \leftarrow \text{NIL}$ 
7   $x \leftarrow \text{head}[H]$ 
8   $\text{next-}x \leftarrow \text{sibling}[x]$ 
9  while  $\text{next-}x \neq \text{NIL}$ 
10   do if ( $\text{degree}[x] \neq \text{degree}[\text{next-}x]$ ) or
        ( $\text{sibling}[\text{next-}x] \neq \text{NIL}$ 
         and  $\text{degree}[\text{sibling}[\text{next-}x]] = \text{degree}[x]$ )
11     then  $\text{prev-}x \leftarrow x$                                 ▷ Cases 1 and 2
12        $x \leftarrow \text{next-}x$                             ▷ Cases 1 and 2
13     else if  $\text{key}[x] \leq \text{key}[\text{next-}x]$ 
14       then  $\text{sibling}[x] \leftarrow \text{sibling}[\text{next-}x]$       ▷ Case 3
15          $\text{BINOMIAL-LINK}(\text{next-}x, x)$                   ▷ Case 3
16       else if  $\text{prev-}x = \text{NIL}$                          ▷ Case 4
17         then  $\text{head}[H] \leftarrow \text{next-}x$                 ▷ Case 4
18         else  $\text{sibling}[\text{prev-}x] \leftarrow \text{next-}x$     ▷ Case 4
19          $\text{BINOMIAL-LINK}(x, \text{next-}x)$                   ▷ Case 4
20          $x \leftarrow \text{next-}x$                             ▷ Case 4
21    $\text{next-}x \leftarrow \text{sibling}[x]$ 
22  return  $H$ 
```

logn



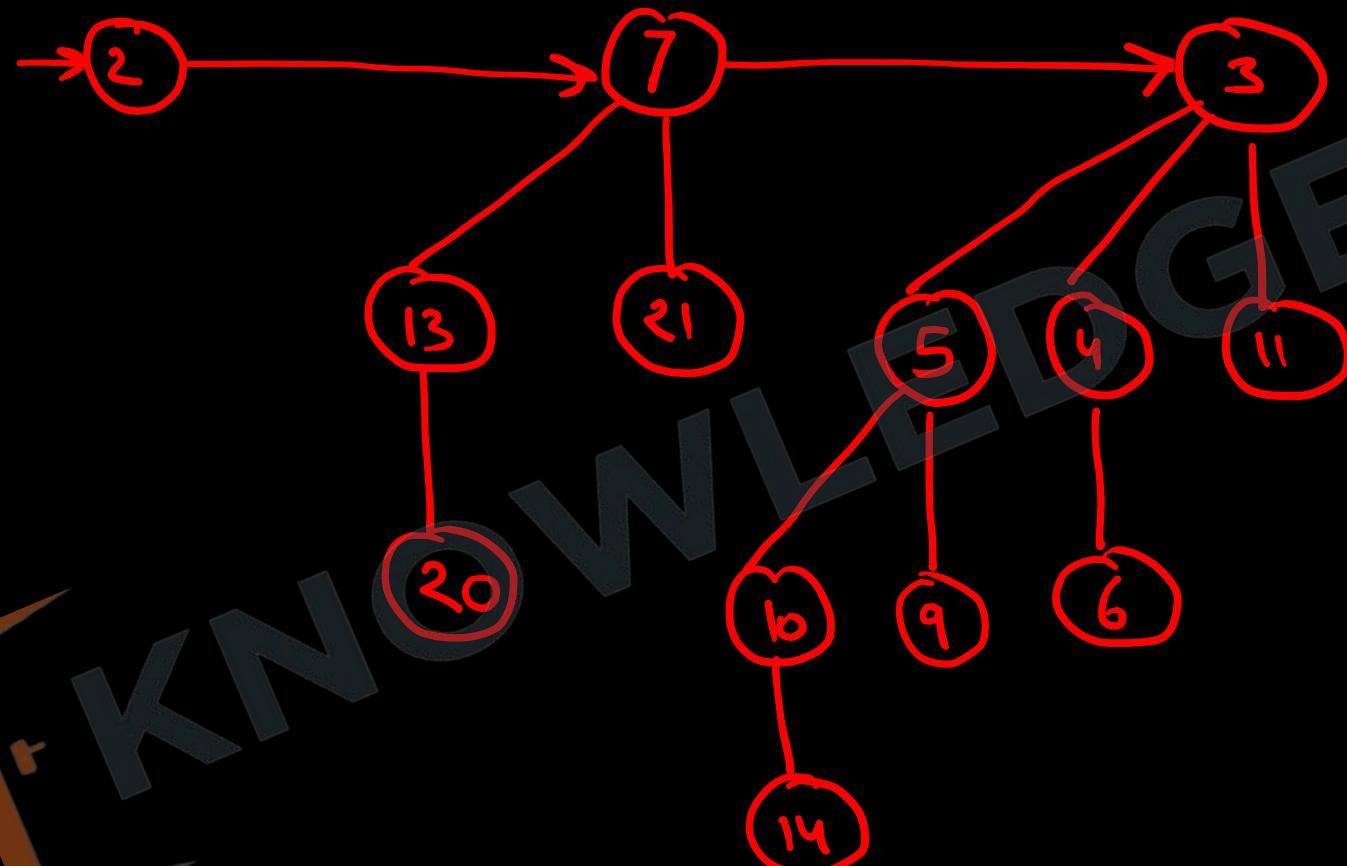


Q Construct the binomial heap for the following sequence of number 4, 6, 3, 11, 9, 5, 14, 10, 21, 7, 13, 20, 2 ?



<http://www.knowledgegate.in/GATE>

Q Construct the binomial heap for the following sequence of number 4, 6, 3, 11, 9, 5, 14, 10, 21, 7, 13, 20, 2 ?

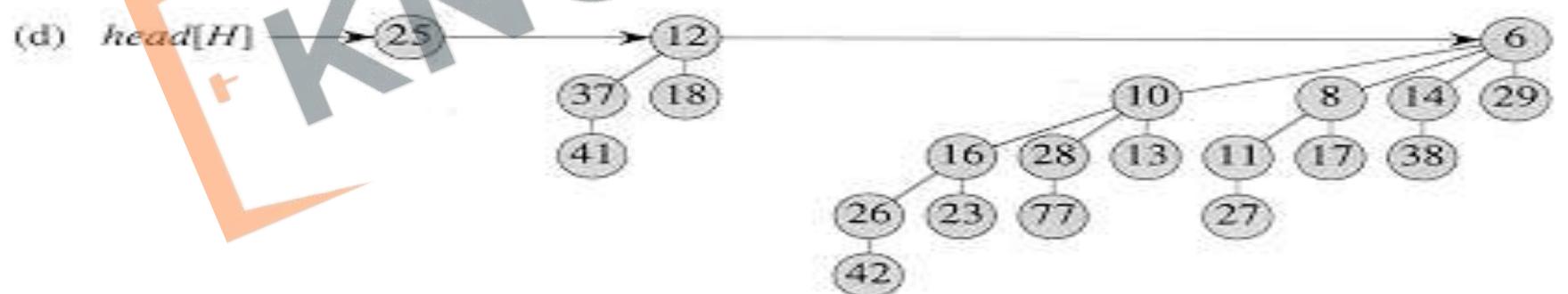
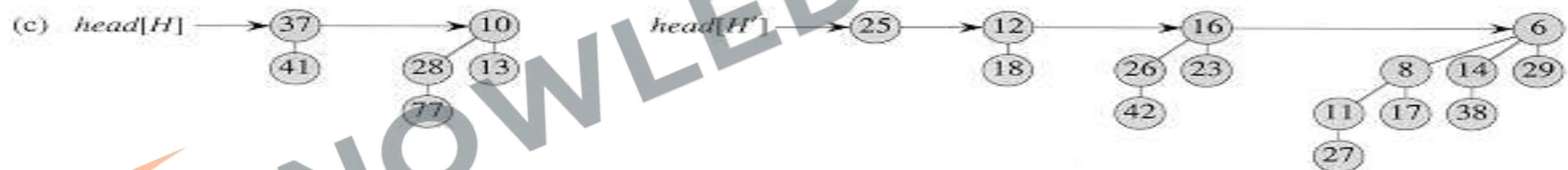
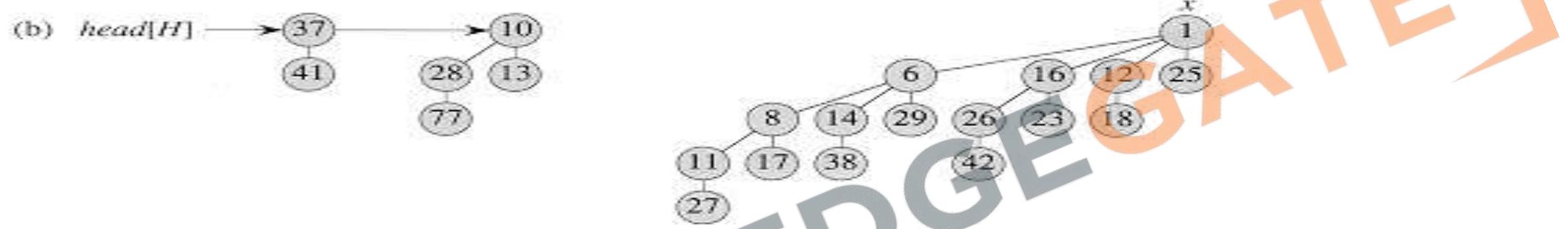
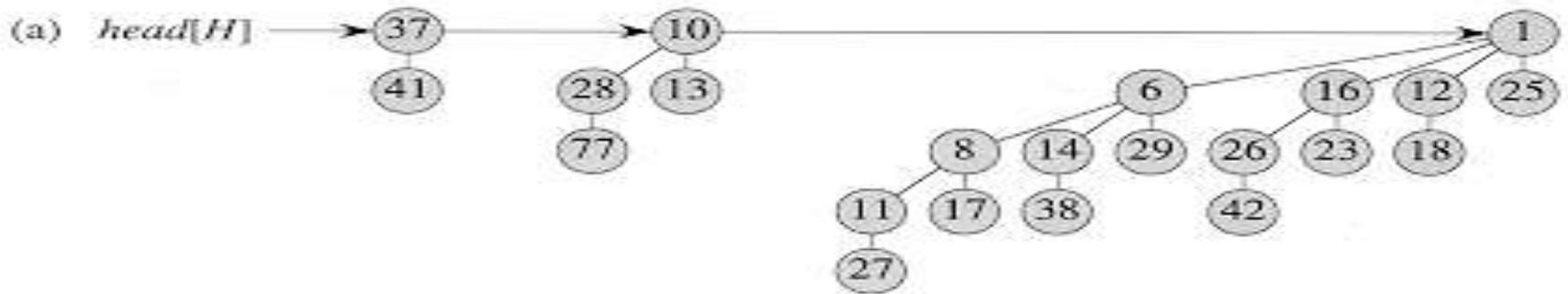


Insertion in Binomial Heap Algorithm

- **Create New Tree:** Form a new single-node binomial tree with the insertion value.
- **Union Heaps:** Merge this new tree with the existing binomial heap.
 - Combine trees of the same degree as necessary.
- **Update Heap:** Set the merged heap as the main heap.

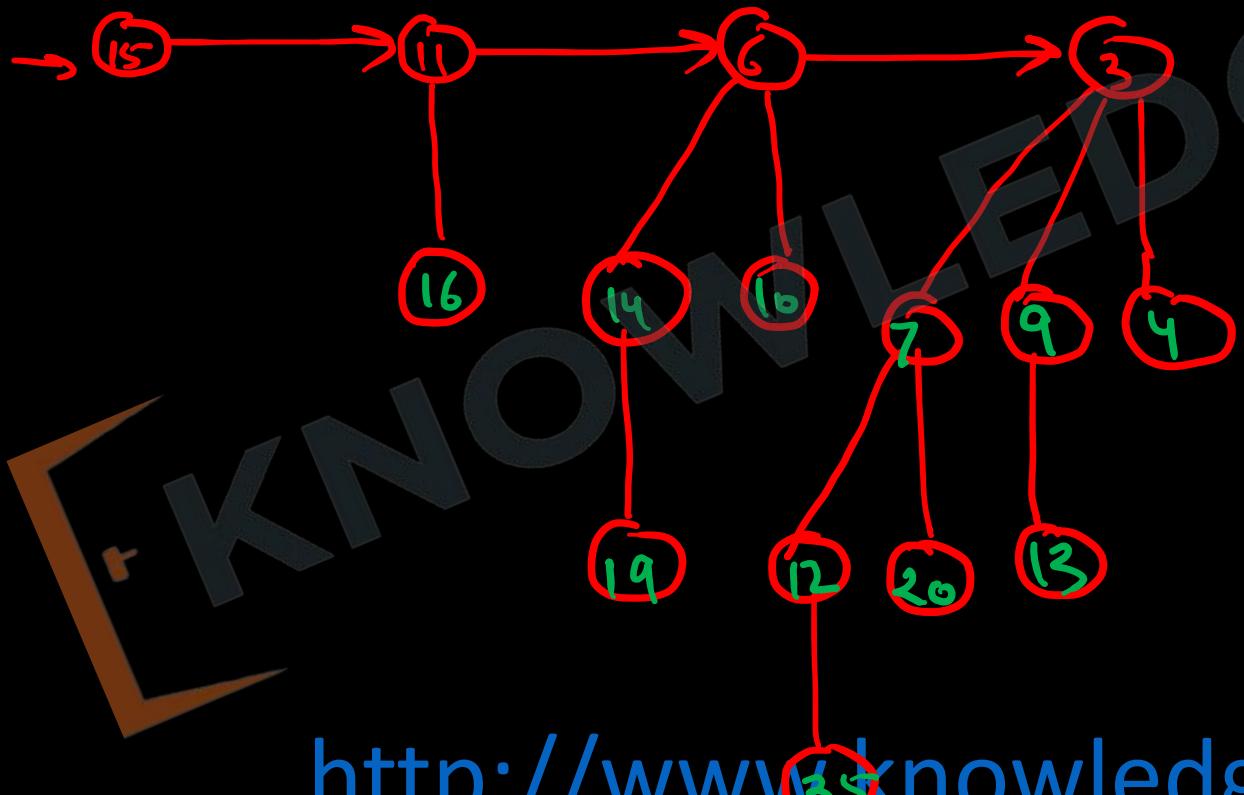
Extract Min in Binomial Heap Algorithm

- **Find Minimum:** Traverse the root list to find the binomial tree with the minimum root key. Let's call this tree B_{\min} .
- **Remove B_{\min} :** Remove B_{\min} from the root list of the heap.
- **Reverse Children of B_{\min} :** Reverse the order of the children of B_{\min} , which are now roots of binomial trees, and create a new binomial heap H' with these trees.
- **Union Heaps:** Perform a union operation between the original heap (without B_{\min}) and H' .
- **Update Pointers:** Update the root list and the minimum element pointer of the heap.



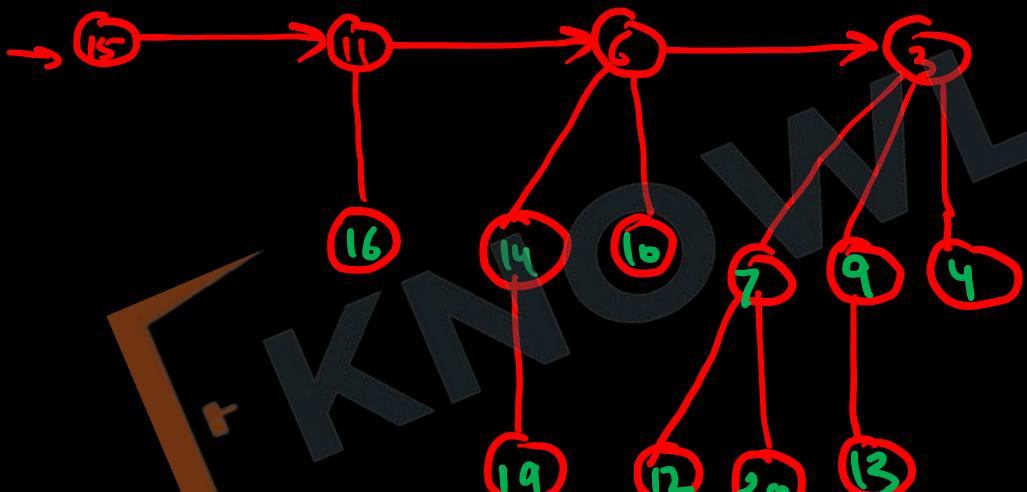
Decrease Key in Binomial Heap Algorithm

- **Update Key:** Assign new, smaller value to node x.
- **Bubble Up:** If 'x' key is less than its parent's, swap x with its parent. Repeat until heap order is restored.
- **Update Min:** Optionally, update the heap's minimum pointer.



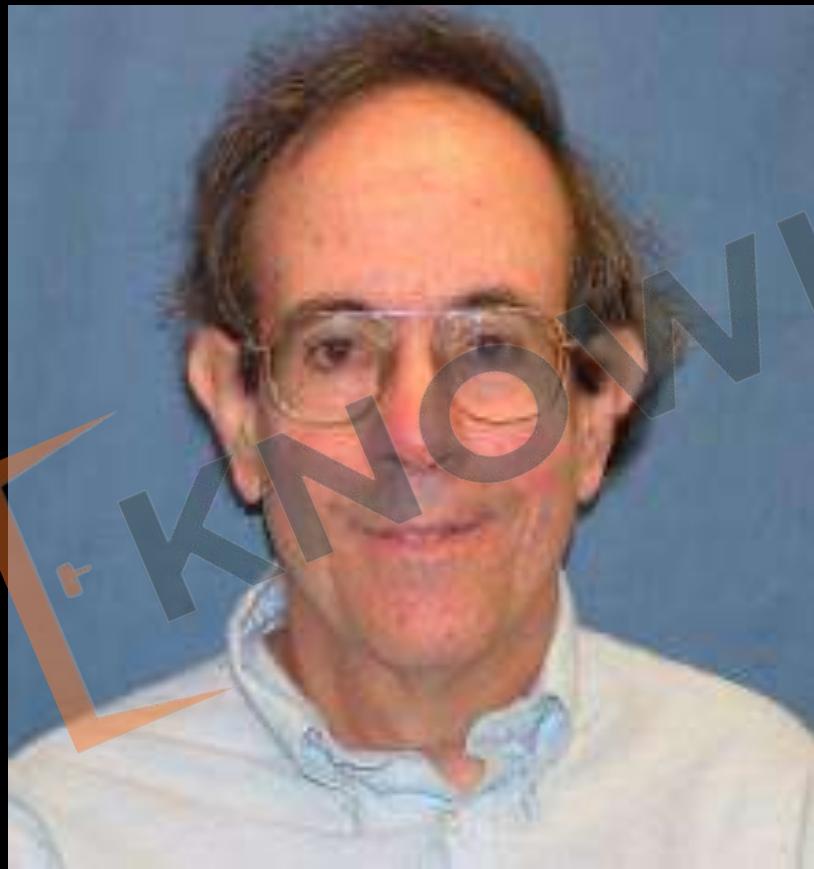
Deletion in Binomial Heap Algorithm

- **Decrease Key:** Decrease the key of the node to be deleted to a value less than or equal to the minimum value in the heap (typically this could be set to negative infinity or a value you know is the smallest possible in your system).
- **Extract Min:** Perform the 'Extract Min' operation to remove the node with the minimum key, which is now the node you wanted to delete.
- **Maintain Heap:** The 'Extract Min' operation will also take care of maintaining the heap properties after deletion.



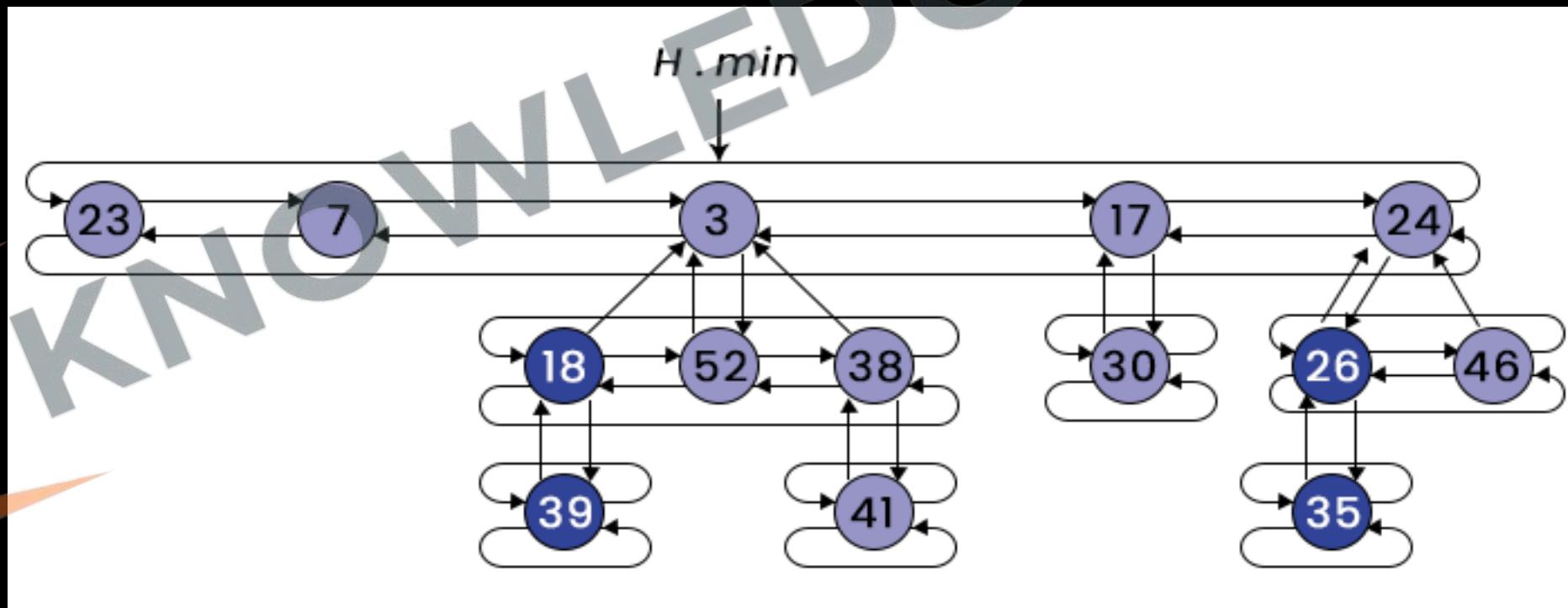
Fibonacci Heaps

- Michael L. Fredman and Robert E. Tarjan developed Fibonacci heaps in 1984 and published them in a scientific journal in 1987. Fibonacci heaps are named after the Fibonacci numbers, which are used in their running time analysis.



Fibonacci Heaps

- Fibonacci Heaps, introduced in 1984, enhance priority queue operations for algorithms like Dijkstra's and Prim's. They feature lazy consolidation and efficient decrease-key operations with $O(1)$ amortized time, and $O(\log n)$ for deletions. Ideal for sparse graphs and network optimization, their complex structure is best for large-scale applications where their amortized time benefits are most impactful.



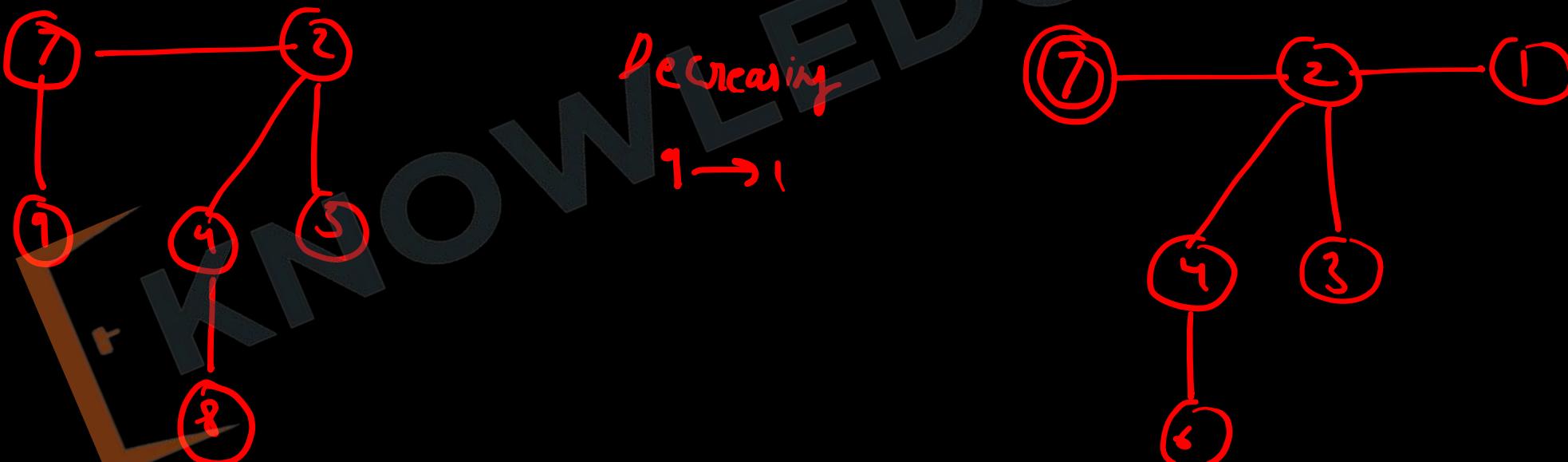
- A Fibonacci Heap is a collection of unordered trees which satisfies the min-heap (or max-heap) property.
- **Structure:** It consists of a set of marked trees. Each tree is a rooted but unordered tree that obeys the min-heap property.
- **Lazy Organization:** Trees within a Fibonacci Heap are not necessarily binomial trees, and the heap does not enforce strict structure. Trees are linked together only as necessary, which usually occurs during extract-min operations.
- **Node Marking:** Nodes in a Fibonacci Heap can be marked, which indicates that a child has been lost since this node was added to its current parent. This is part of the mechanism to limit the degree (number of children) of nodes.
- **Mechanism:** When a child node is removed from a parent (which can happen during certain heap operations), the parent node gets marked. If a marked node loses another child, it's removed from its parent, potentially causing a cascade of changes up the heap.
- **Consolidation:** After the removal of the minimum node, the remaining trees are consolidated into a more structured heap. This operation is "lazy" and is done only when necessary, such as during extract-min operations, to keep the number of trees small.
- **Degree and Number of Children:** The size of a tree rooted at a node is at least F_{k+2} , where k is the degree of the node, and F is the Fibonacci sequence. This ensures that the heap has a low maximum degree.

- **Amortized Costs:**
 - **Insert:** $O(1)$ amortized
 - **Decrease Key:** $O(1)$ amortized
 - **Merge/Union:** $O(1)$ amortized
 - **Extract Min:** $O(\log n)$ amortized
 - **Delete:** $O(\log n)$ amortized
- The Fibonacci Heap is ideal for applications where the number of insertions and decrease-key operations vastly outnumber the number of extract-min and delete operations, thus taking advantage of the low amortized costs for the former operations. Its complexity comes into play in theory more than in practical applications due to constants and lower-order terms involved in its operations.

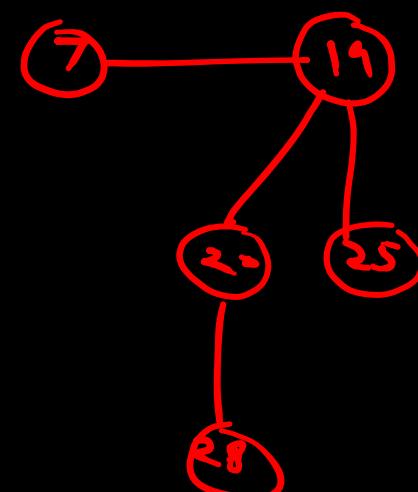
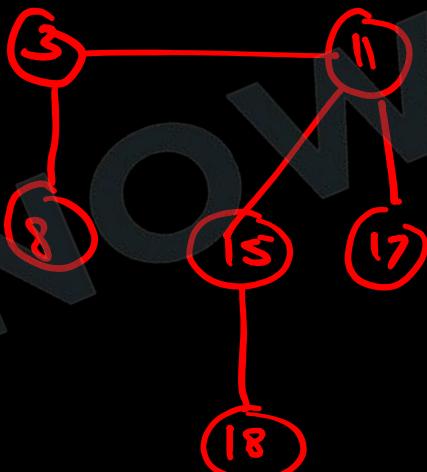
- Insertion in a Fibonacci heap is a simple and efficient operation. Here's how it is typically implemented:
 - **Create a New Node:** A new node containing the key to be inserted is created.
 - **Add to Root List:** This new node is added to the list of roots in the Fibonacci heap.
 - **Update Minimum:** If the new key is smaller than the current minimum key in the heap, the minimum pointer is updated to this new node.
 - **Amortized Cost:** The operation runs in $O(1)$ amortized time, making it very efficient.
- The simplicity of the insertion operation is due to the lazy approach of Fibonacci heaps, where no immediate reorganization or consolidation of the heap is performed at the time of insertion.



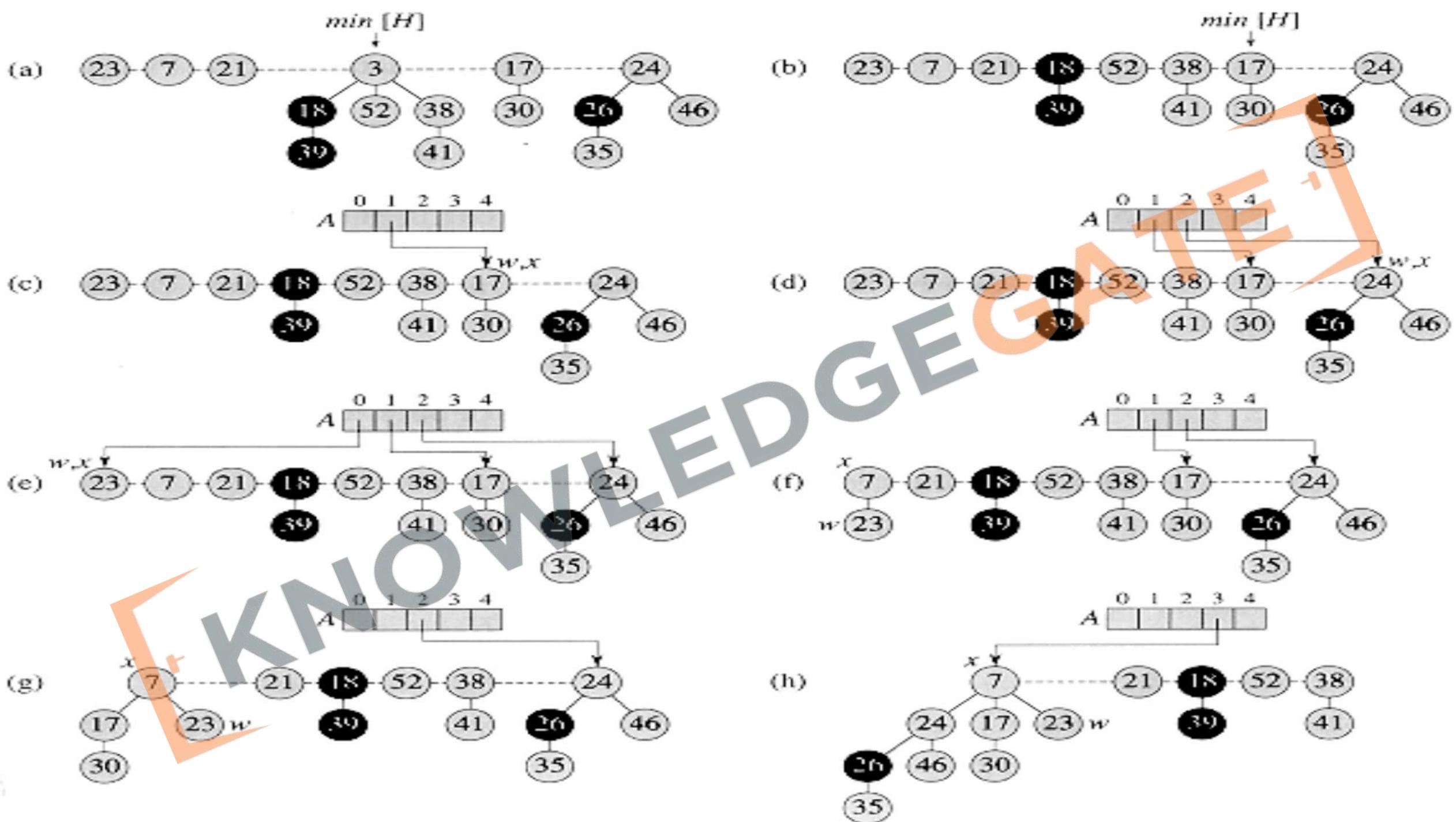
- Decrease key in a Fibonacci heap is an operation to reduce the value of a given node and then adjust the heap to maintain the heap property. Here's how it works:
 - Decrease the Key:** Update the value of the node to the new lower key.
 - Check Heap Property:** If the new key is still greater than or equal to the parent's key, the heap property is maintained, and no further action is required.
 - Cut and Add to Root List:** If the new key violates the heap property (it's now less than the parent's key), cut this node from its parent and add it to the root list.
 - Mark the Parent:** If the parent node was unmarked, mark it. If it was already marked, cut it as well and add it to the root list, then continue this process recursively up the tree.
 - Update Minimum:** If the new key is less than the current heap's minimum key, update the minimum pointer.

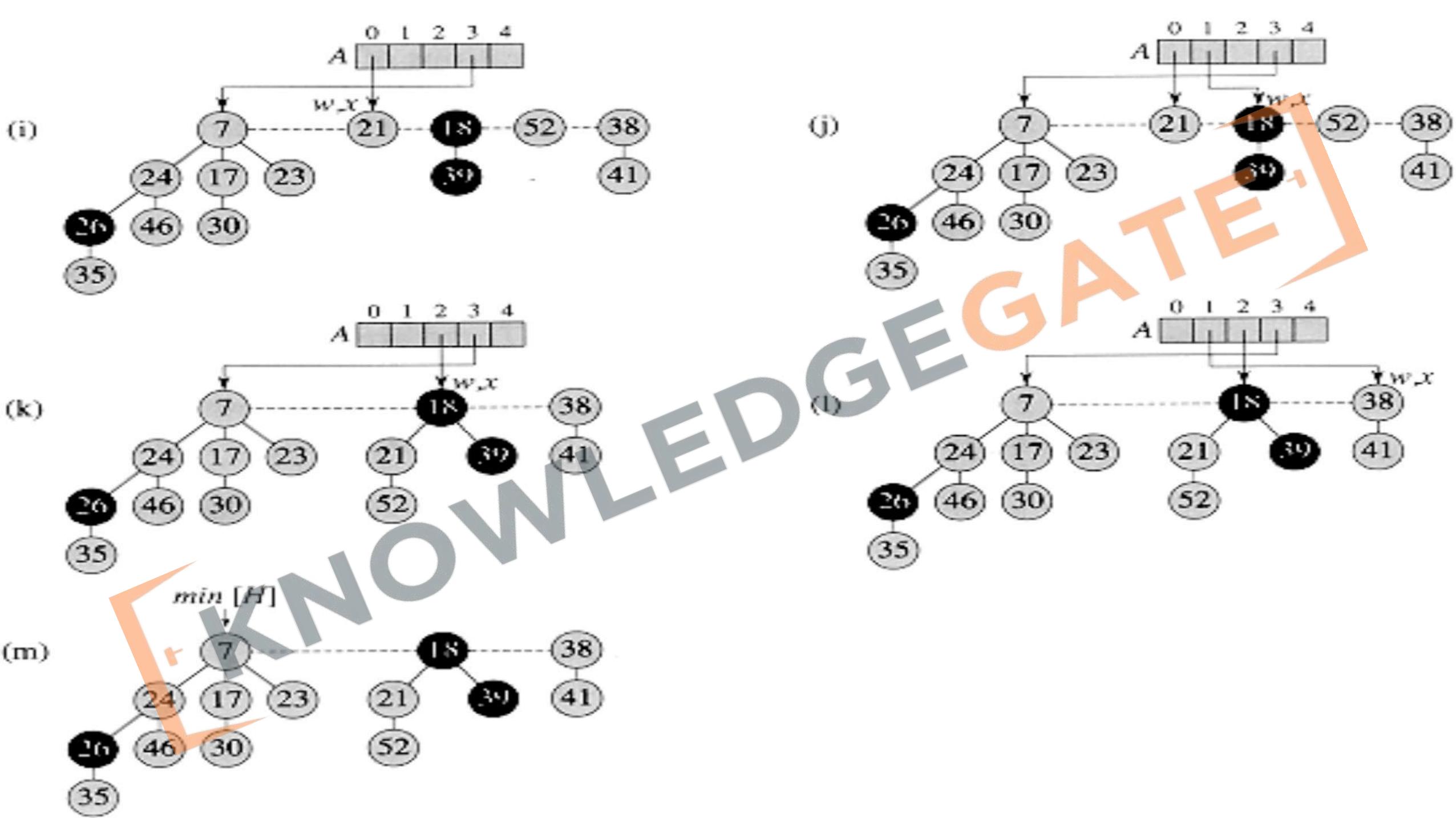


- Merging in a Fibonacci heap, also known as the union operation, is the process of combining two Fibonacci heaps into a single heap. This operation is efficient and is performed in the following way:
 - **Concatenate Root Lists:** Combine the root lists of the two heaps into a single root list, which can be done in $O(1)$ time since the root lists are typically circular doubly linked lists.
 - **Update Minimum:** Check the minimum nodes of both heaps and update the pointer to the minimum node if necessary.
 - **No Immediate Consolidation:** Unlike other heap structures, the trees are not consolidated right away. This is part of the lazy strategy of Fibonacci heaps, which delays work until it's needed (e.g., during an extract-min operation).
 - **Amortized Cost:** The amortized cost of the merge operation is $O(1)$.
- The merge operation in Fibonacci heaps is a foundational operation that supports the efficiency of other heap operations, like insertions and decrease-key, because it enables the heap to maintain a more flexible structure.

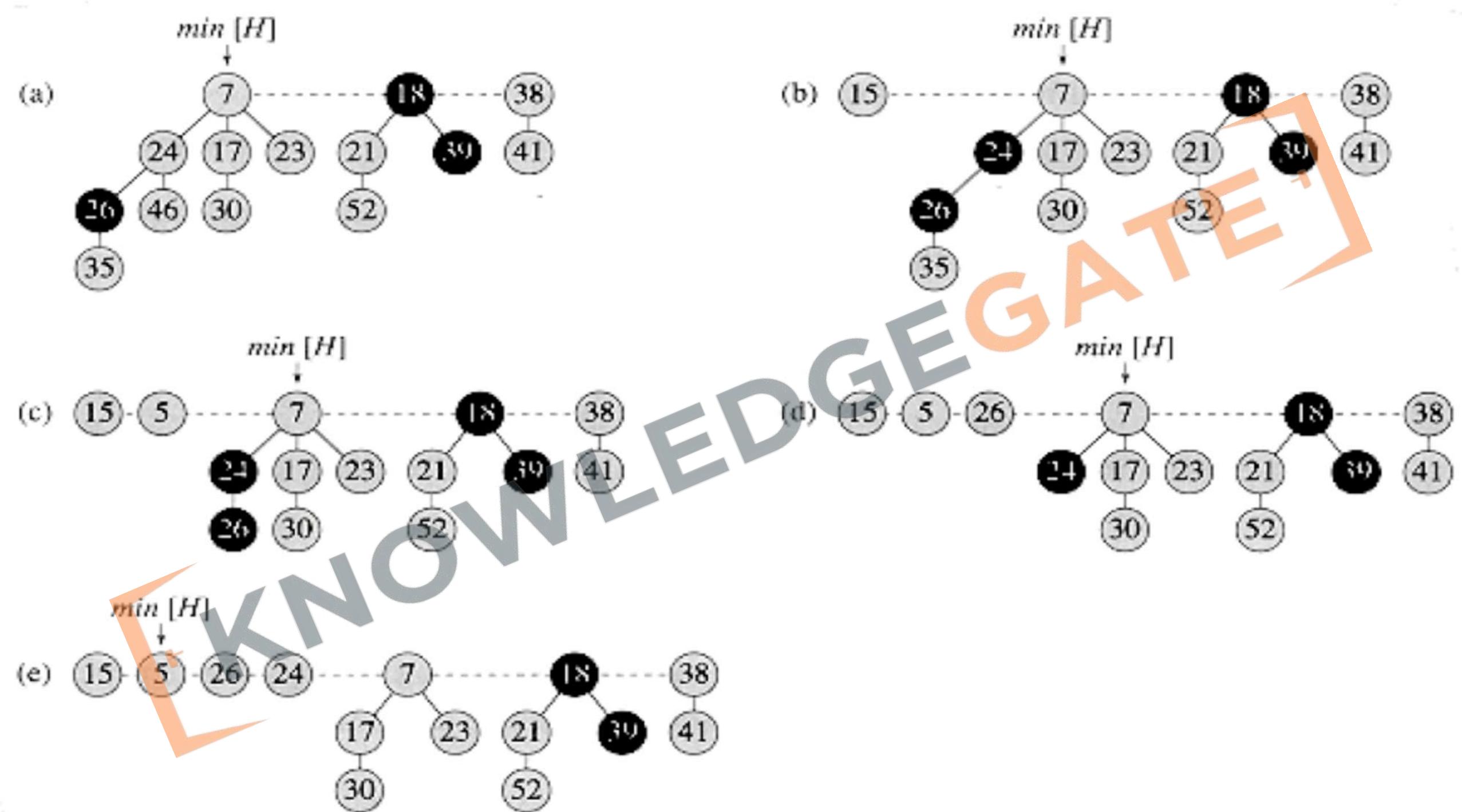


- The "Extract Min" operation in a Fibonacci heap is used to remove and return the smallest key in the heap. It's a key operation that involves more restructuring of the heap compared to other operations. Here's a step-by-step description:
- **Remove the Minimum Node**: Take out the node containing the minimum key from the root list. This node will be returned at the end of the operation.
- **Add Children to Root List**: The children of the minimum node are added to the root list of the heap.
- **Consolidate the Heap**: This step restructures the heap to ensure that no two trees have the same degree in the root list:
 - Pairwise combine trees in the root list that have the same degree until every tree has a unique degree.
 - During this process, link trees by making the one with the larger root a child of the one with the smaller root.
- **Find New Minimum**: Traverse the root list to find the new minimum node and update the heap's minimum pointer.
- **Time Complexity**: The worst-case time complexity for this operation is $O(\log n)$, but this is the amortized complexity because the actual work is done during the consolidation step, which doesn't happen on every operation.
- **Amortized Cost**: The amortized cost is $O(\log n)$ due to the potential increase from the previous operations which pays for the consolidation.





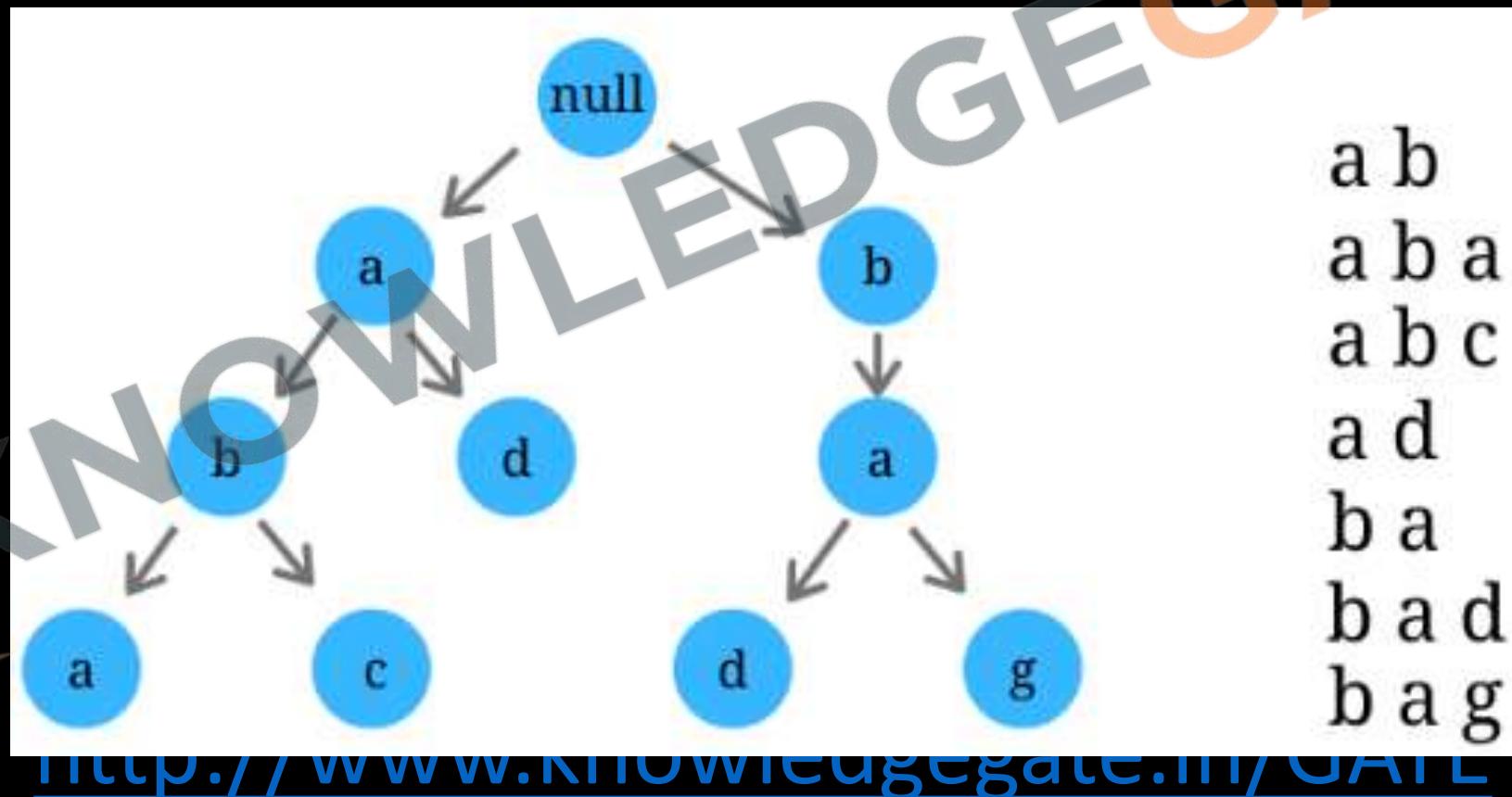
- The delete operation in a Fibonacci heap removes a specific node from the heap. It's more complex than the extract-min operation because it requires locating the node to be deleted, regardless of its position in the heap. Here's how the delete operation works:
 - **Decrease Key to Minus Infinity:** Decrease the key of the node to be deleted to the smallest possible value. This is often done by setting it to negative infinity or a value lower than any other in the heap.
 - **Extract Min:** Perform the extract-min operation, which will now remove the node with the decreased key since it's the smallest in the heap.
 - **Consolidate the Heap:** If necessary, during the extract-min operation, the heap is consolidated to ensure that it maintains the correct structure, merging trees where necessary.
 - **Time Complexity:** The time complexity for the delete operation is $O(\log n)$ amortized, due to the extract-min operation that is invoked.
- By combining the decrease-key and extract-min operations, the delete operation ensures that the heap remains properly structured and that the min-heap property is maintained throughout.



Procedure	Binary Heap	Binomial Heap	Fibonoci Heap
Make Heap	$O(1)$	$O(1)$	$O(1)$
Insert	$O(\log n)$	$O(\log n)$	$O(1)$
Min	$O(1)$	$O(\log n)$	$O(1)$
Extract min	$O(\log n)$	$O(\log n)$	$O(\log n)$
Union	$O(n)$	$O(\log n)$	$O(1)$
Decrease Key	$O(\log n)$	$O(\log n)$	$O(1)$
Delete	$O(\log n)$	$O(\log n)$	$O(\log n)$

Tries

- A Trie, also known as a prefix tree, is a tree-like data structure that stores a dynamic set of strings, usually for retrieval of keys by prefix. It is a type of search tree—an ordered tree data structure used for locating specific keys from within a set.



Requirements:

- **Efficient Storage**: Tries are used to store strings efficiently for retrieval by prefix, which can reduce space requirements compared to naive storage methods.
- **Prefix Lookup**: Tries must support fast lookup to check whether a string is in a set, which is particularly efficient when dealing with prefixes.
- **Dynamic**: They must allow insertion and deletion of keys.
- **Definition:**
 - A Trie is a hierarchical data structure consisting of nodes with pointers to their children along with associated characters. Each node represents a common prefix shared by some keys, and each path down the tree represents a key (usually a string).

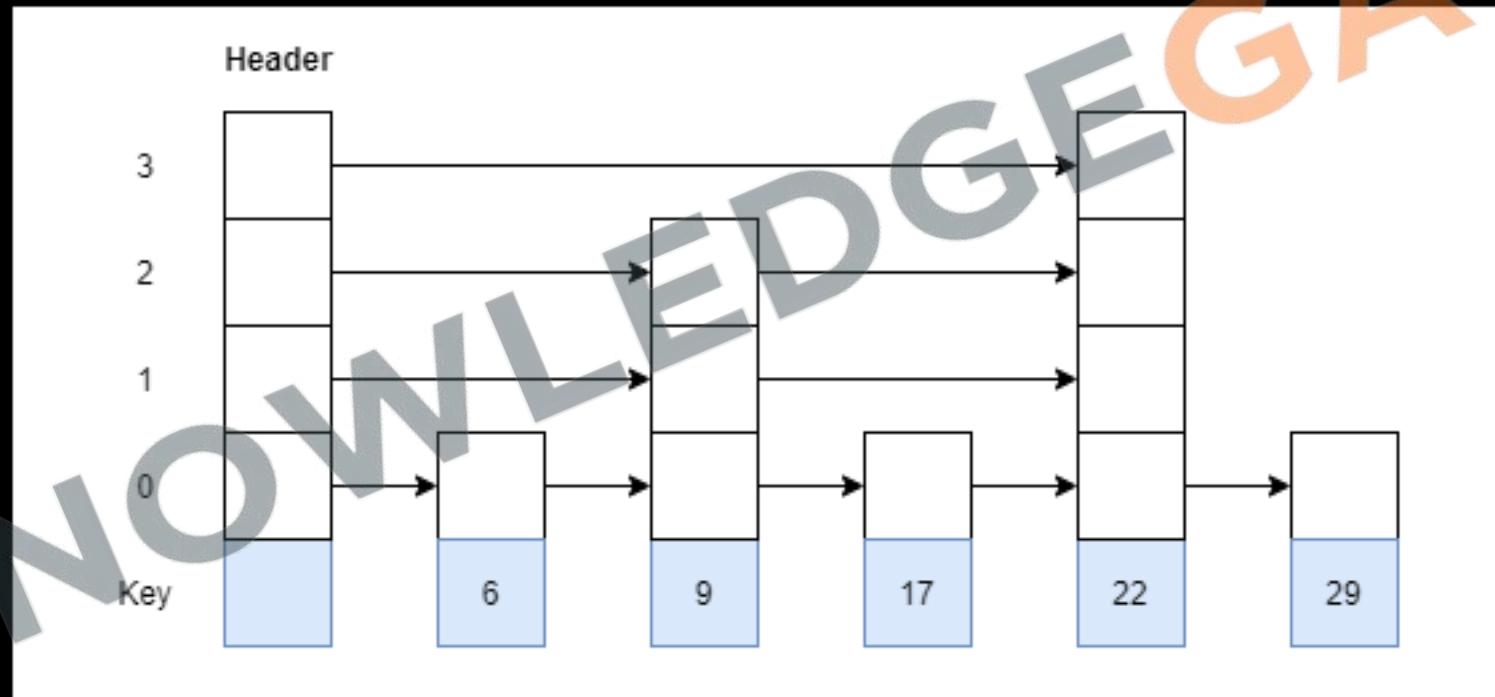
- **Properties:**

- **Edges:** Each edge in a Trie is labelled with a character or a substring.
- **Root Node:** The root node is usually an empty string and does not hold any character.
- **Nodes:** Internal nodes, which can have multiple children, store the common prefixes of the strings. Leaf nodes represent the end of a particular key.
- **Path:** A path from the root to a leaf or a node with a non-null value represents a word or a prefix to a word.
- **Time Complexity:** The time complexity for inserting, searching, and deleting from a Trie is $O(m)$, where m is the length of the key.

- **Autocomplete**: Tries are well-suited for implementing autocomplete features in search engines and text editors due to their efficient prefix searching capabilities.
- **Spell Checking**: They can be used for spell checking as one can quickly check the existence of a word or prefix.
- **IP Routing**: Tries are used in IP routing to store and search IP addresses.
- **Bioinformatics**: In bioinformatics, tries are used for tasks such as genome analysis where sequences of DNA are stored and searched.
- **Text Games**: Games like Scrabble use tries to validate words and search for possible words with given letters.

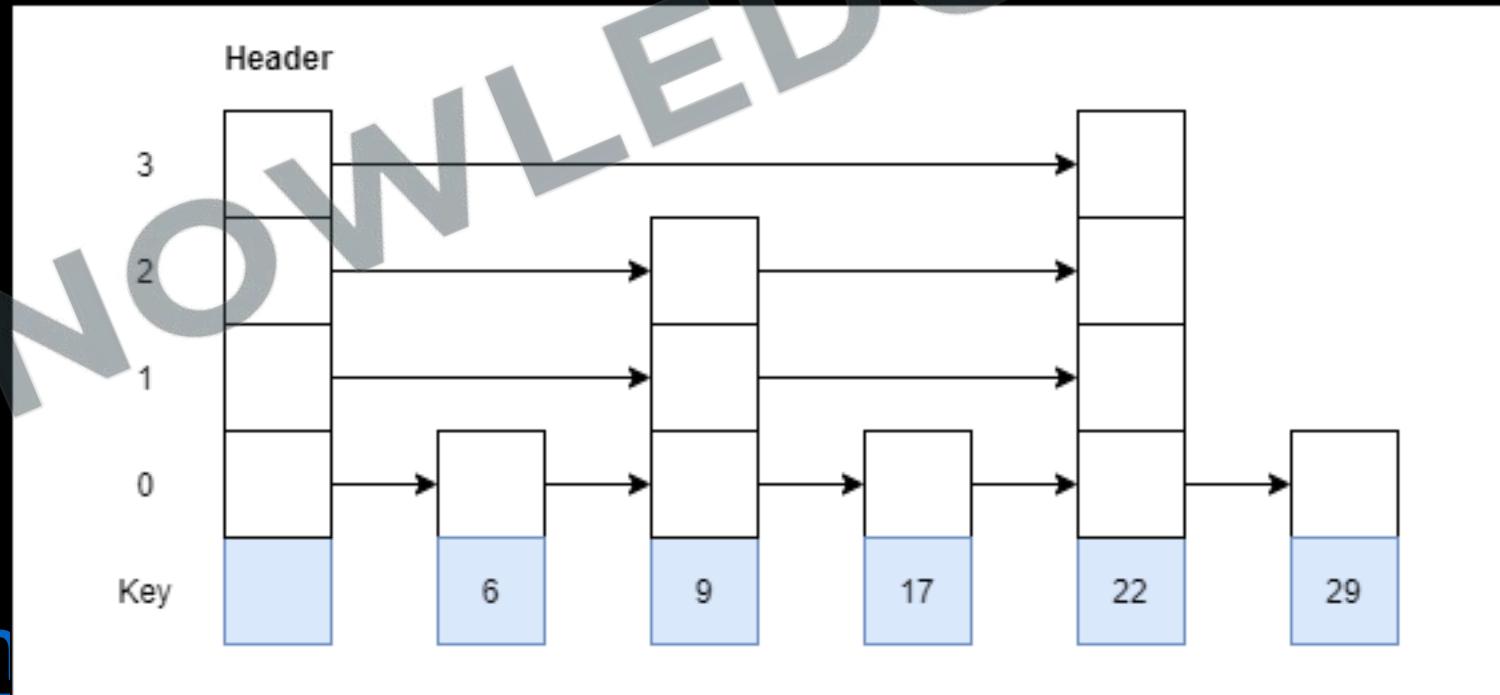
Skip List

- A Skip List is a data structure that allows fast search within an ordered sequence of elements. It does this by maintaining a hierarchy of linked lists that skip over a subset of the elements.



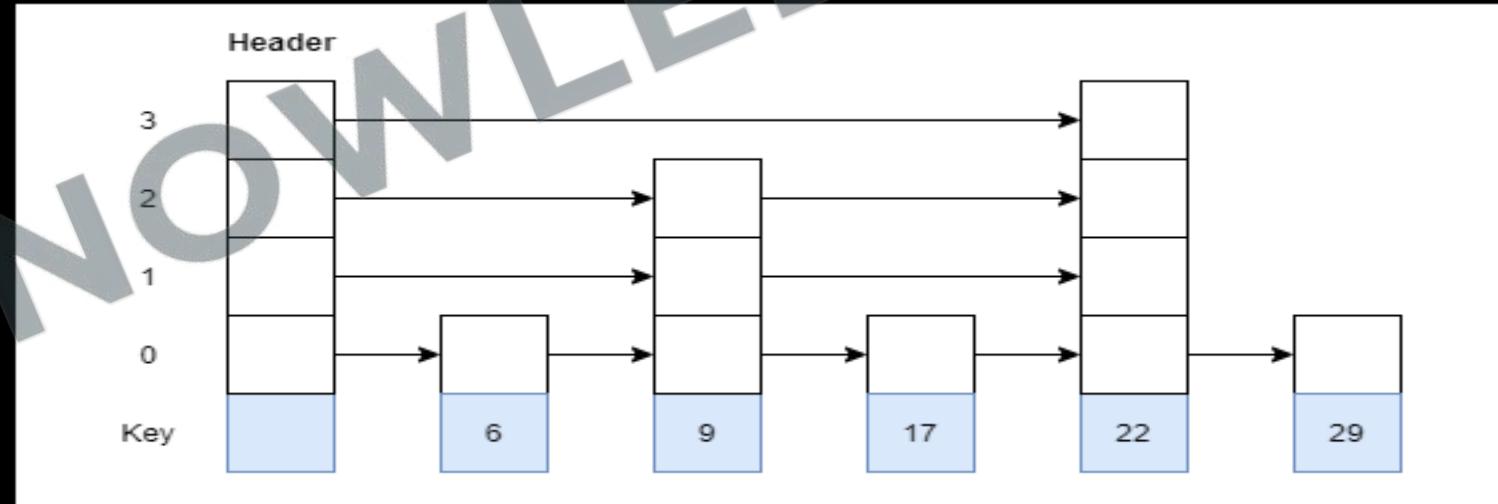
Requirements:

- **Ordered Elements**: Elements must be maintained in a sorted order.
- **Multiple Levels**: Skip lists consist of several layers of linked lists, where the bottom layer is the ordinary list of all elements.
- **Randomization**: The structure of skip lists is probabilistic, typically decided by coin flips or a random number generator to determine the height of each element's "tower."



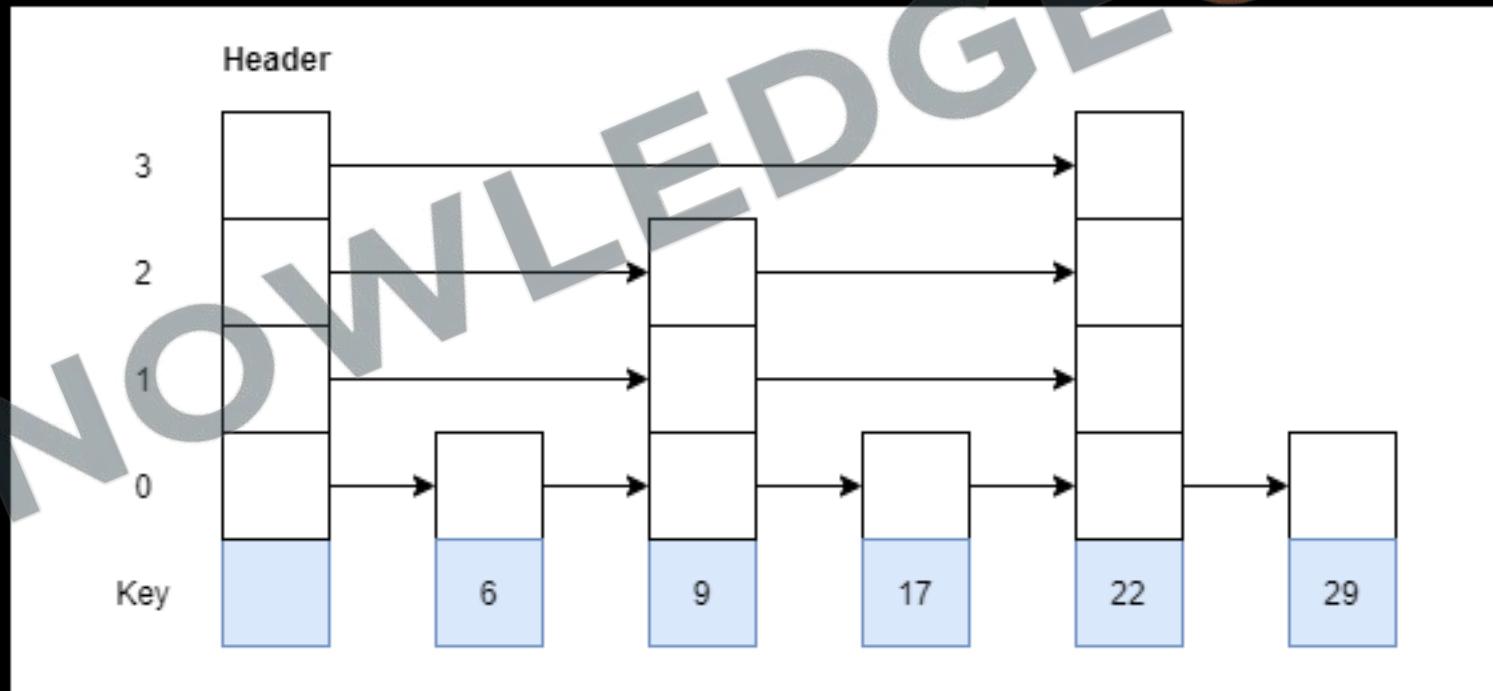
Properties:

- **Efficient Search**: Provides logarithmic search time similar to binary search trees.
- **Probabilistic Balancing**: Unlike trees, skip lists do not require re-balancing operations.
- **Ease of Implementation**: Simpler to implement than balanced trees.
- **Memory Overhead**: May require additional space compared to a standard linked list due to multiple levels.



Applications:

- **Databases**: Skip lists are used in database indices where quick search, insertion, and deletion are necessary.
- **Probabilistic Data Structures**: Can be part of larger data structures that deal with approximations or probabilistic results.

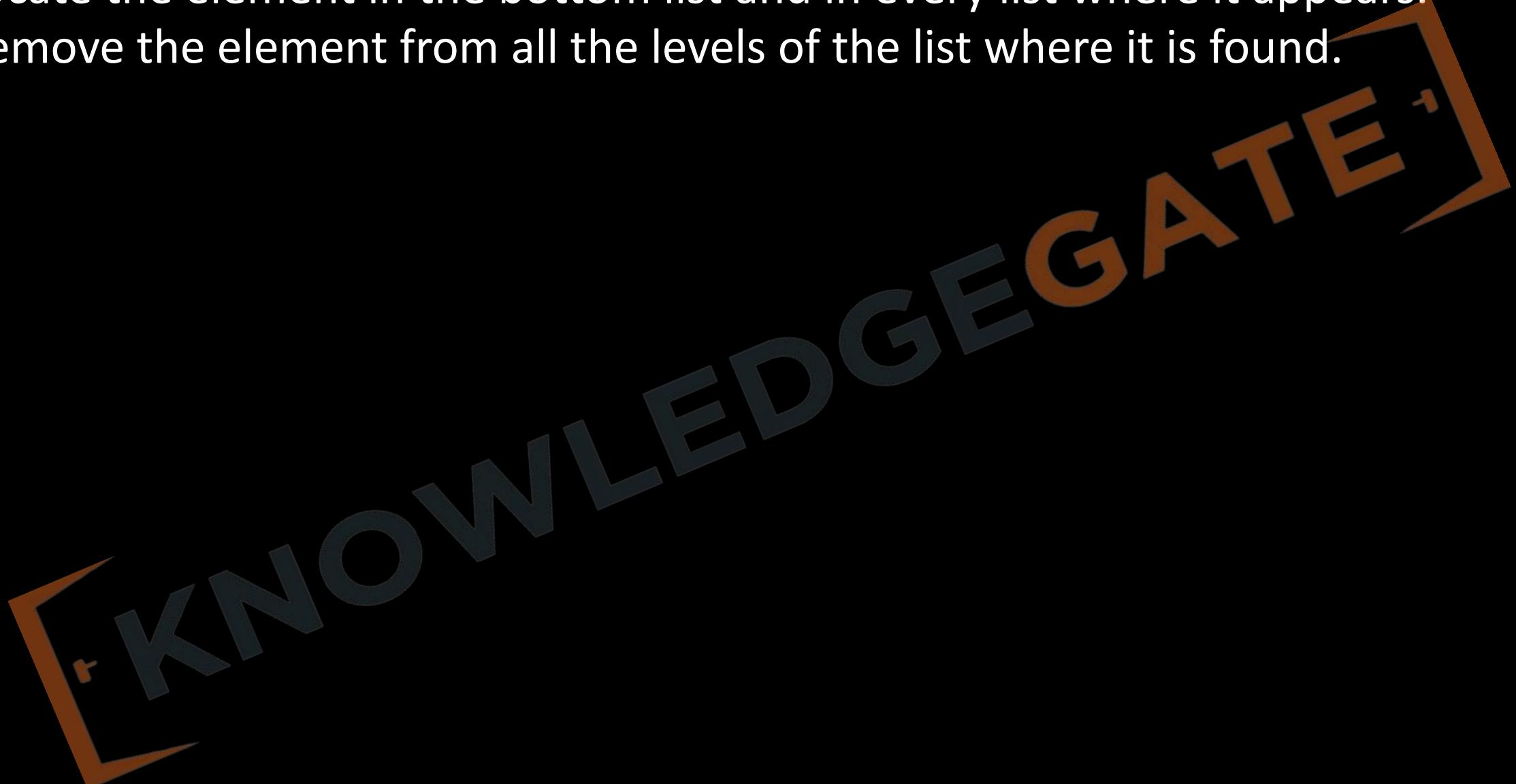


Insertion Algorithm:

- Search for the position where the new element should be inserted, starting from the top level and moving down.
- Insert the element in the lowest level linked list.
- Randomly decide the number of levels the new element should participate in.
- Add the element to each of the higher levels according to the decision in step 3.

Deletion Algorithm:

- Locate the element in the bottom list and in every list where it appears.
- Remove the element from all the levels of the list where it is found.



<http://www.knowledgegate.in/GATE>

Searching Algorithm:

- Start from the head element of the top list.
- Move forward until the next element is greater than the search key.
- Move down one level and continue the search.
- If the bottom level is reached, the search either finds the element or determines it's not in the list.

Complexity:

- **Search**: Average $O(\log n)$, Worst-case $O(n)$ (if unlucky with randomization).
- **Insertion**: Average $O(\log n)$, Worst-case $O(n)$.
- **Deletion**: Average $O(\log n)$, Worst-case $O(n)$.

Ch-9

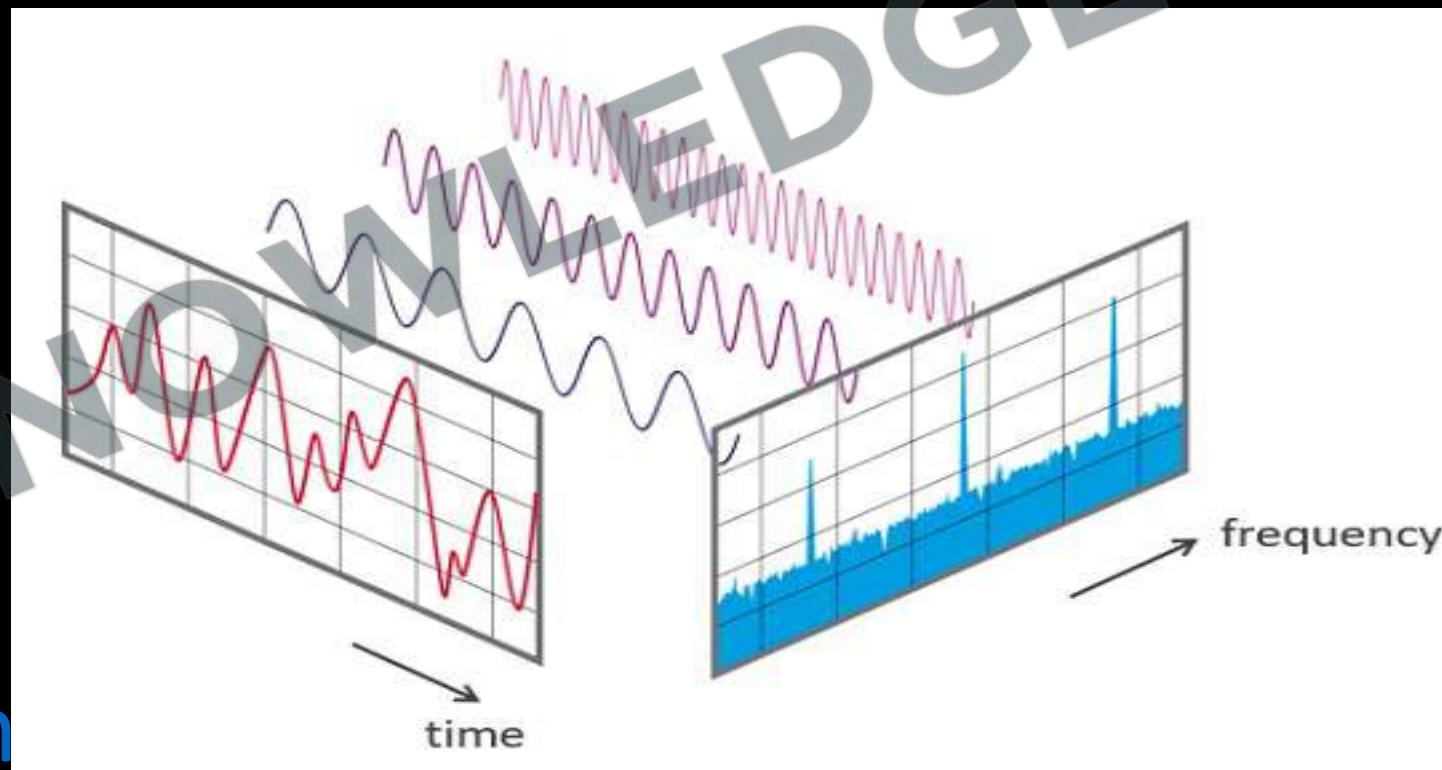
Selected Topics

Algebraic Computation, Fast Fourier Transform, String Matching, Theory of NPCompleteness, Approximation

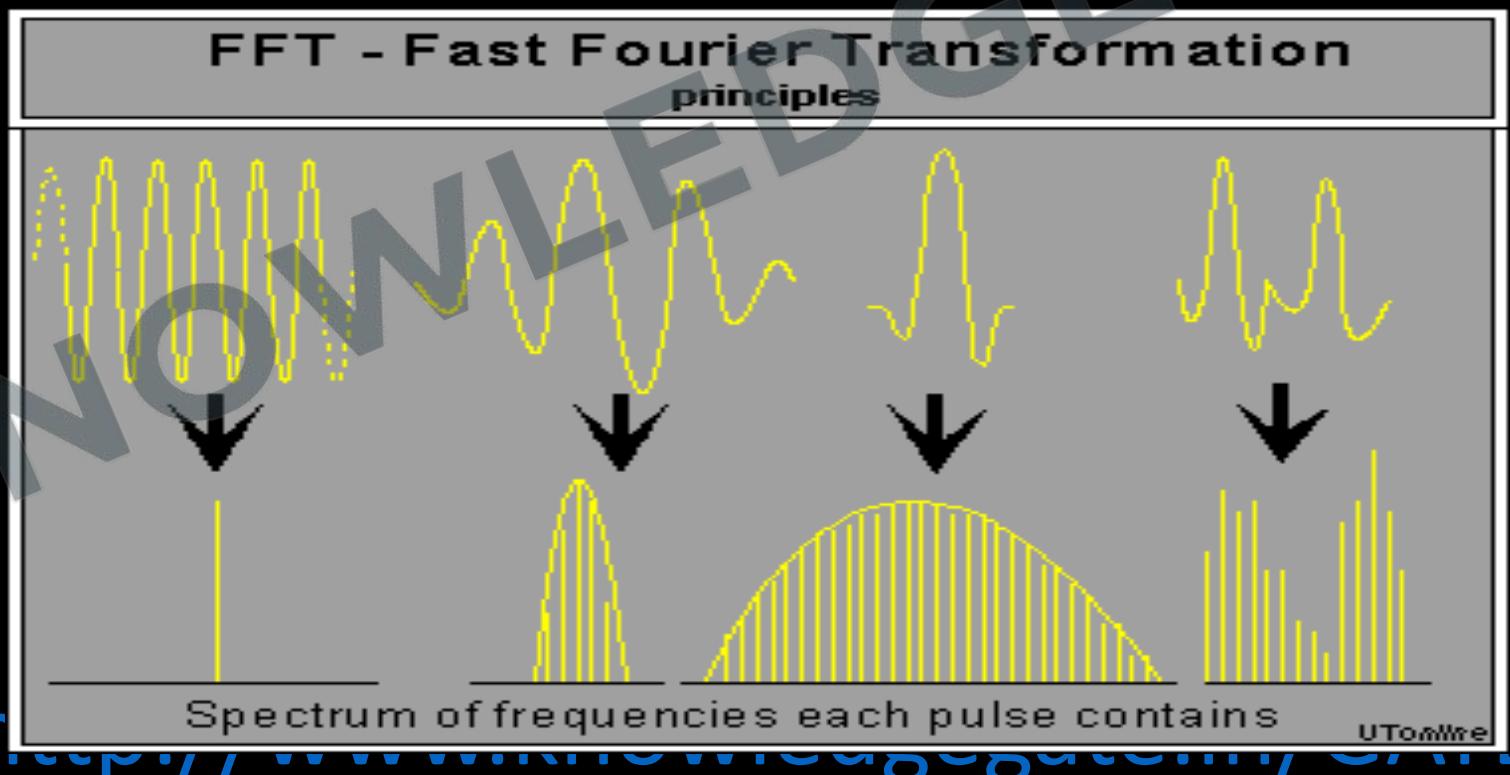
Algorithms and Randomized Algorithms .

Fast Fourier Transformation

- What it is: FFT is a quick way for computers to change how they look at a list of numbers. Instead of seeing them just as a sequence over time, it helps the computer see what cycles or patterns (frequencies) are in that list.
- Why it's special: It does its job really fast compared to older methods. Imagine having to check every book in a library to find one book, versus having a quick way to go right to the shelf where the book is.



- **Where it's used:** It's like a tool in a toolbox that can be used for many things, like making music sound better, getting clear pictures for doctors to see inside the body, and even for phones to send messages without wires.
- **Types:** There's a common kind of FFT that works best when the list of numbers has a total that's a "power of two" (like 2, 4, 8, 16, and so on).
- **What it does:** It helps us understand different pitches in music or bright and dark spots in images by breaking them down into simpler parts that are easier to work with or understand.



String matching

- String matching, often referred to as "pattern matching," is the process of finding one or more instances of a string (pattern) within another string (text). The goal is to find all the positions at which the pattern appears in the text.
- **String Matching Problem:** The string-matching problem is a classic problem in computer science where you want to find all occurrences of a shorter string (a "pattern") within a longer string (a "text"). It's a foundational problem with applications in text editing, DNA sequence analysis, and information retrieval, among others.
- For example, in the string "abracadabra," you might want to find where the substring "abra" appears. The answer would be at positions 1 and 8.
 -

- **String:** A string is a sequence of characters. The characters can be letters, numbers, or symbols. For example, "hello" is a string made of 5 characters.
- **Substring:** A substring is any continuous sequence of characters within a string. For instance, "hello" contains the substrings "he," "ell," "llo," "hello," etc. A substring can be as short as one character or as long as the entire string itself.
- **Proper Substring:** A proper substring is a substring that is strictly contained within a string and is not equal to the string itself. So, for "hello," "hell" or "ello" would be proper substrings, but "hello" would not be a proper substring of itself.

Naive Algorithm

- The Naive algorithm is the simplest method where you slide the pattern over the text one character at a time and check for a match.
- Imagine you lost a small toy in a long hallway. The naive way to find it would be to walk along the hallway and check every single spot on the floor, one by one, until you find the toy.
- **Pros:**
 - Very simple to understand and implement.
 - Works well when the patterns are short and the text is not too large.
- **Cons:**
 - Can be very slow, especially if the pattern occurs frequently but with mismatches.

String: a b c d e f g h

Pattern: d e f

String: a b c d a b c a b c d f — $O(m)$

Pattern: a b c d f — $O(n)$

$O(m \cdot n)$

NAIVE-STRING-MATCHER (T, P)

```
{  
    n = T.length  
    m = P.length  
    for s = 0 to n- m  
        if P[1..m] == T[s + 1 ..s + m]  
            print "Pattern occurs with shift" s  
}
```

Rabin-Karp Algorithm

- In computer science, the **Rabin–Karp algorithm** or **Karp–Rabin algorithm** is a string-searching algorithm created by Richard M. Karp and Michael O. Rabin (1987) that uses hashing to find an exact match of a pattern string in a text.



Rabin-Karp Algorithm

- The Rabin-Karp algorithm uses hashing to find any set of pattern occurrences. Instead of checking all characters of the pattern at every position (like the naive algorithm), it checks a hash value.
- Think of it like looking for a specific page in a book by its unique code instead of by reading every word. If the page number (hash) matches, then you check to make sure it's really the page you're looking for.
- **Pros:**
 - Faster than the naive approach on average.
 - Very efficient for multiple pattern searches at once.
- **Cons:**
 - Requires a good hash function to avoid frequent spurious hits.
 - The worst-case time complexity can be as bad as the Naive algorithm if many hash collisions occur.

Text: a a a a a b

a-1

Pattern: a a b

b-2

c-3

d-4

e-5

f-6

g-7

h-8

i-9

j-10

$$l + l + 2 = 4$$

↑
 $h(p)$

hank code

Text: a b c d a b c e

a-1

ⁿ

Pattern: b c e

b-2

^m

c-3

d-4

e-5

f-6

g-7

h-8

i-9

j-10

$\Theta(n-m+1)$

<http://www.knowledgegate.in/GATE>

Text: c c a c c a a e d b a

a-1

Pattern: d b a

b-2

c-3

d-4

e-5

f-6

g-7

h-8

i-9

j-10

Spurious Hits O(mn)
<http://www.knowledgegate.in/GATE>

Knuth-Morris-Pratt (KMP) Algorithm

- The algorithm was conceived by James H. Morris and independently discovered by Donald Knuth "a few weeks later" from automata theory.^{[1][2]} Morris and Vaughan Pratt published a technical report in 1970.^[3] The three also published the algorithm jointly in 1977.^[1]



Knuth-Morris-Pratt (KMP) Algorithm

- The KMP algorithm is smarter. It pre-processes the pattern to understand its structure and eliminates unnecessary comparisons when a mismatch occurs.
- **Pros:**
 - The algorithm ensures that the characters of the text are never compared more than once, which makes it very efficient.
 - No backtracking is needed, so it's more efficient than the naive approach.
- **Cons:**
 - The pre-processing step requires additional time and memory.
 - The algorithm is more complex to understand and implement.

1 2 3 4

P₁: a b a b

1 2 3 4 5 6 7 8

P₂: a b c d a b c y

1 2 3 4 5 6 7 8 9 10

P₃: a b c d a b e a b f

1 2 3 4 5 6 7 8 9 10 11

P₄: a b c d e a b f a b c

String: a b a b c a b c a b a b a b d

i 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

j 0 1 2 3 4 5

Pattern: a b a b d

```
KMP-MATCHER (T, P)
{
    n = T.length
    m = P.length
    pie = COMPUTE-PREFIX-FUNCTION(P)
    pie = 0
    for i = 1 to n
        while q > 0 and P[q + 1] ≠ T[i]
            q = pie[q]
        if P[q + 1] == T[i]
            q = q + 1
        if q == m
            print "Pattern occurs with shift i-m"
            q = pie[q]
}
```

COMPUTE-PREFIX-FUNCTION(P)

{

M = P.LENGTH

Let pie[1 . . . m] be a new array

pie[1] = 0

k = 0

for q = 2 to m

 while k > 0 and p[k+1] ≠ p[q]

 k= pie[k]

 if p[k+1] == p[q]

 k=k+1

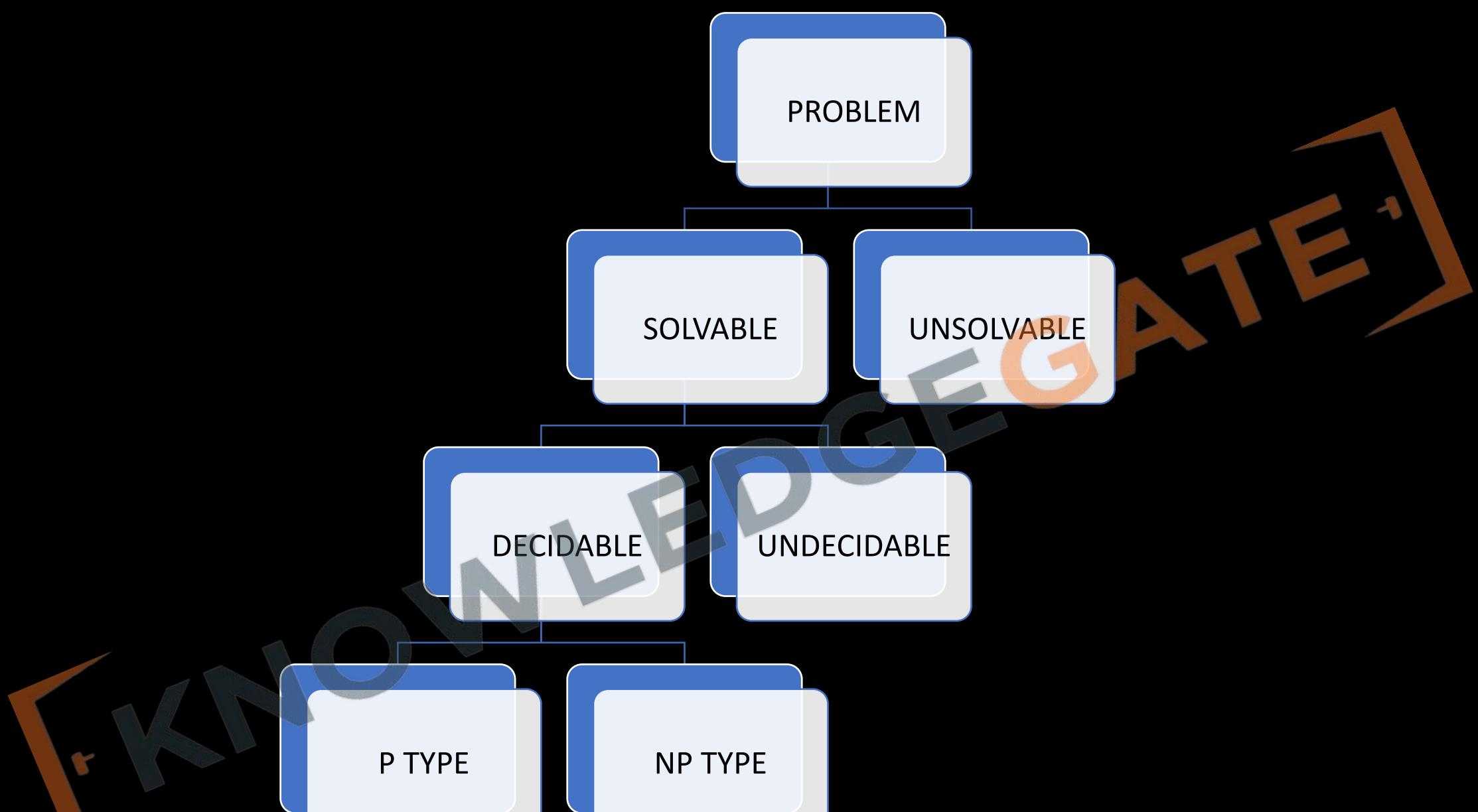
 pie[q] = k

return pie

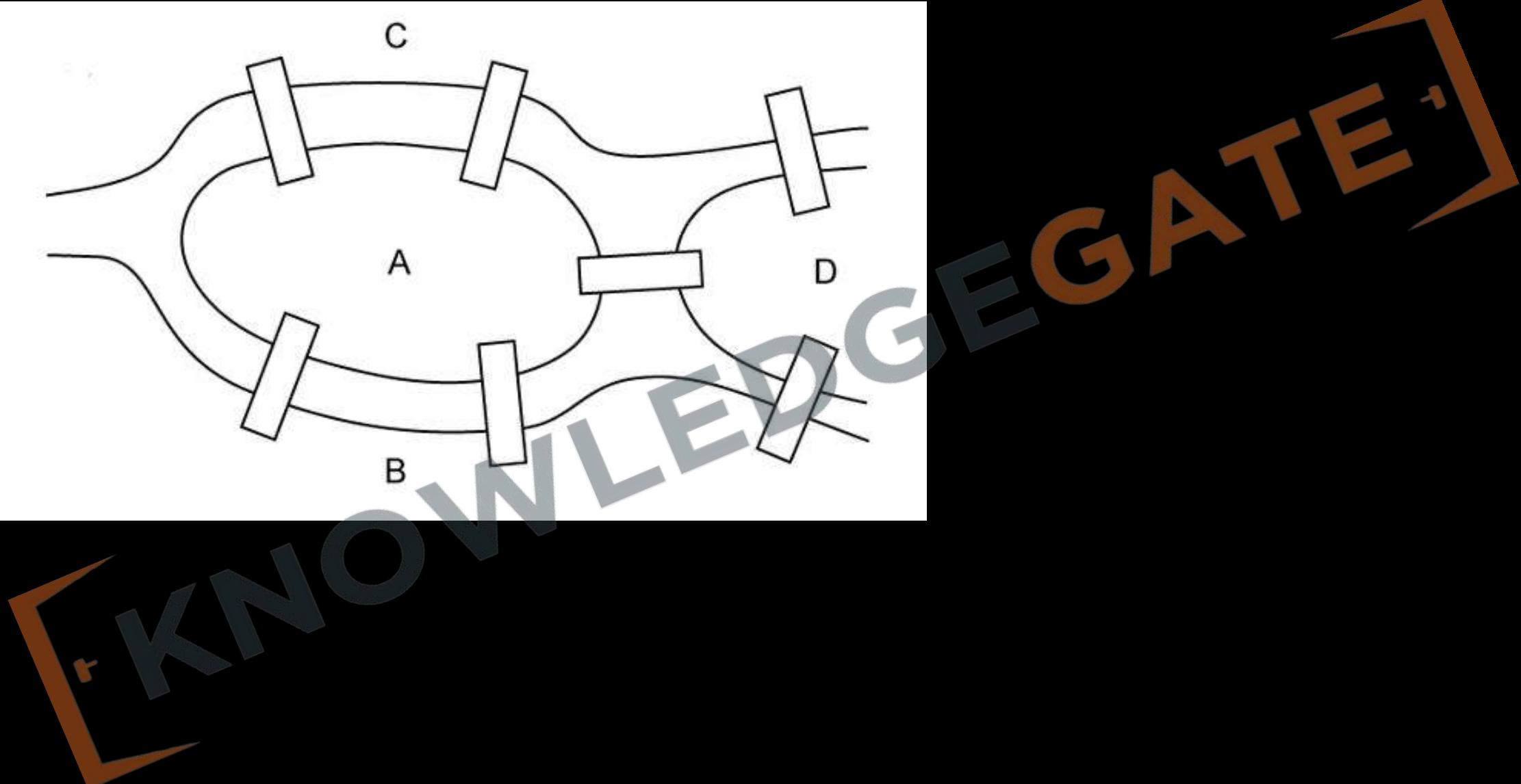
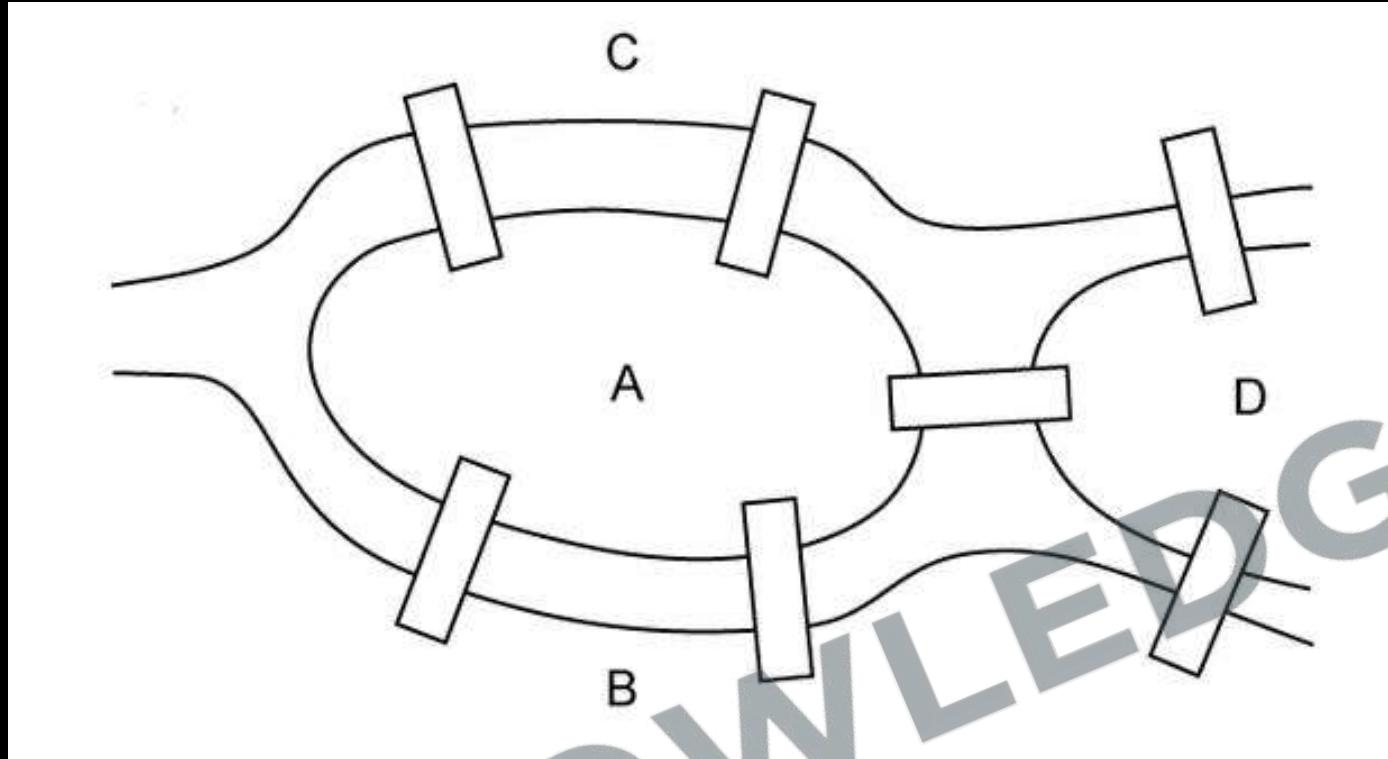
}

What computer science deals with?

- Don't
 - So we do not study how to design a computer
 - We do not study how to run a computer
- Do
 - We deal with problem solving, according to computer science a problem can be divided as follows



Konigsberg Bridge Problem

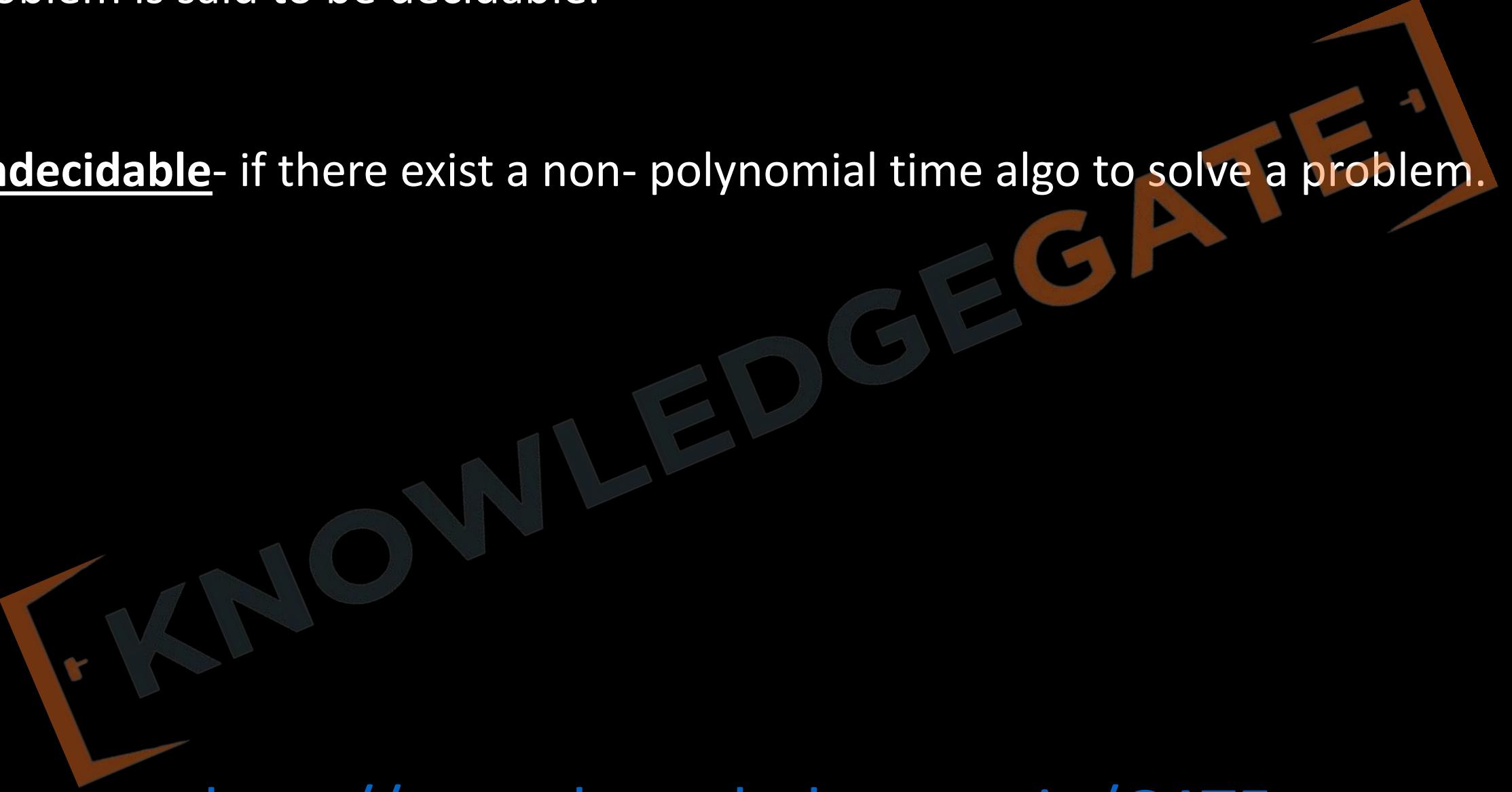


<http://www.knowledgegate.in/GATE>

- **SOLVABLE** - A problem is said to be solvable if either we can solve it or if we can prove that the problem cannot be solved
- **UNSOLVABLE** - A problem is said to be unsolvable if neither we can solve it, nor we can proof that the problem can not be solved

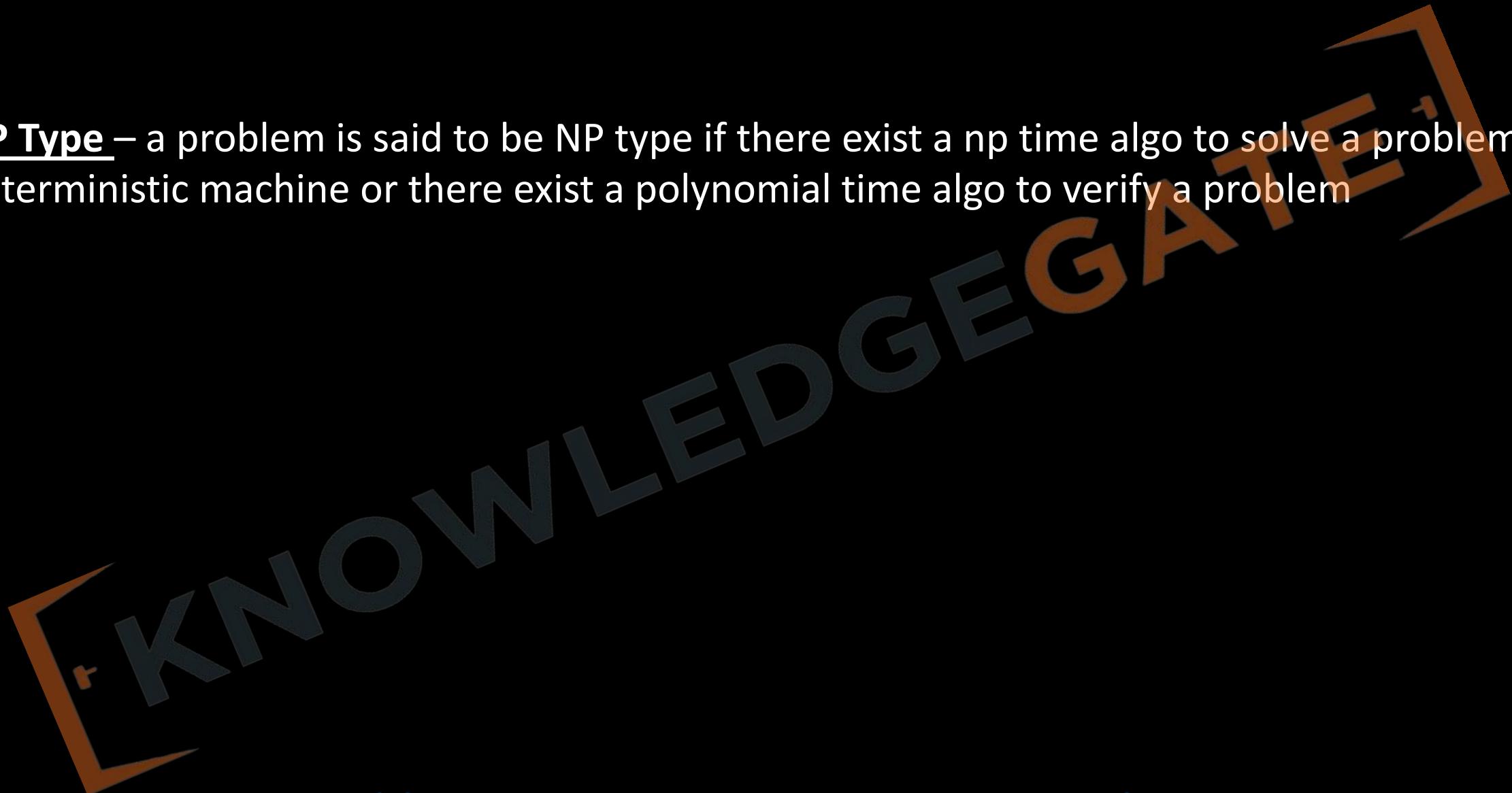
<http://www.knowledgegate.in/GATE>

- **Decidable**- if there exist a polynomial time algorithm to solve a problem then problem is said to be decidable.
- **Undecidable**- if there exist a non- polynomial time algo to solve a problem.

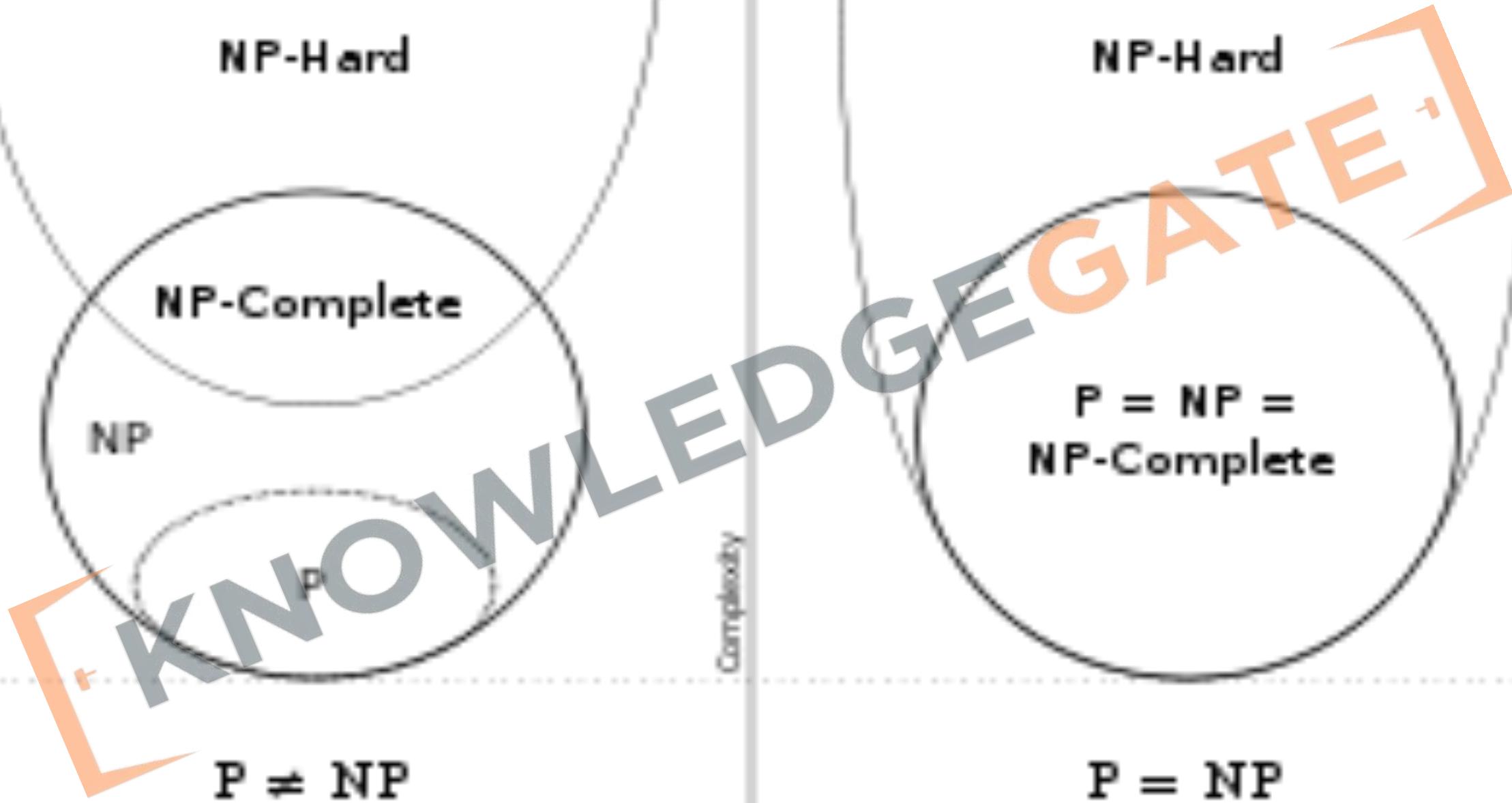


<http://www.knowledgegate.in/GATE>

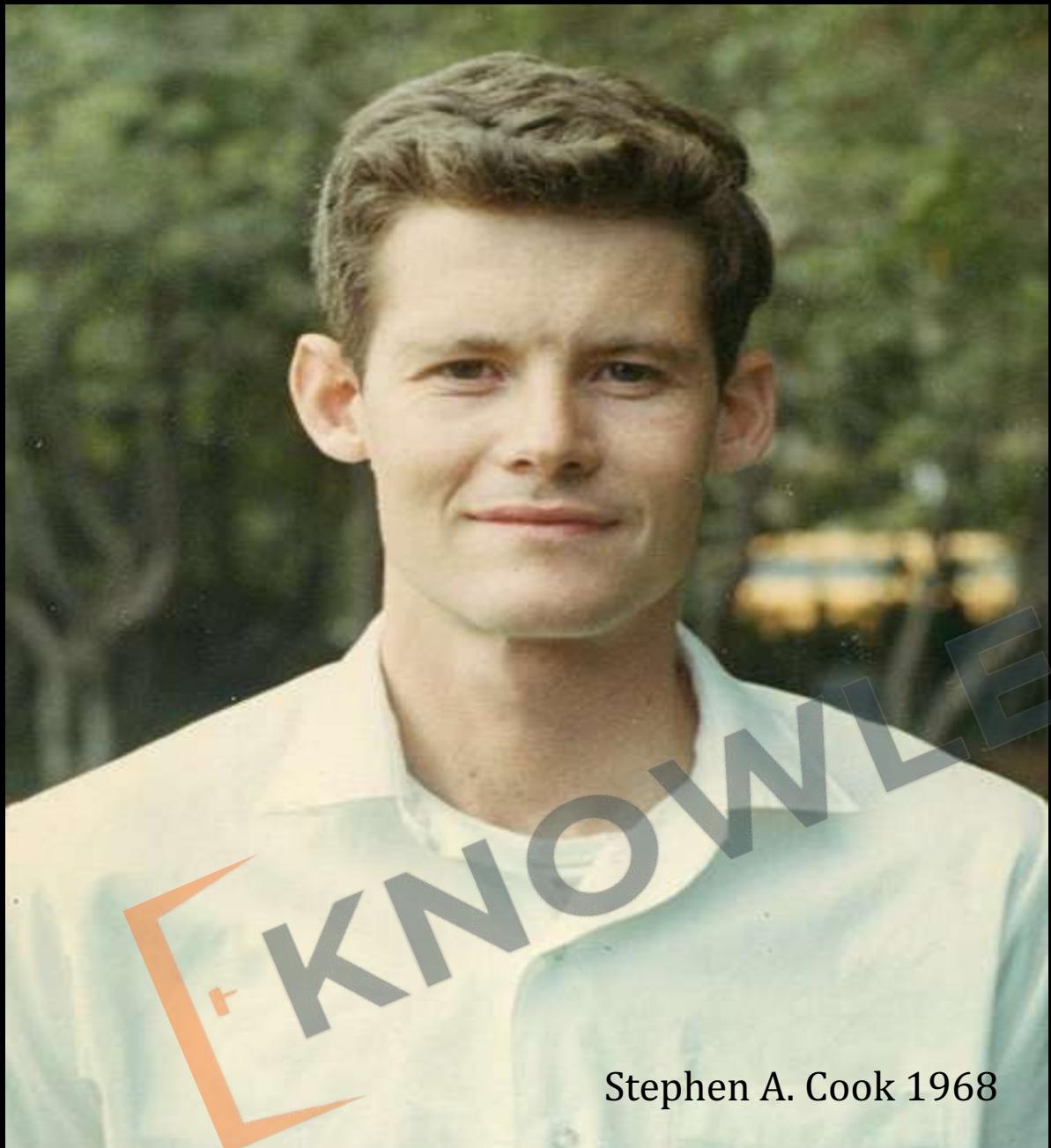
- **P Type** - a problem is said to be P type if there exist a polynomial time algo to solve a problem
- **NP Type** – a problem is said to be NP type if there exist a np time algo to solve a problem on a deterministic machine or there exist a polynomial time algo to verify a problem



<http://www.knowledgegate.in/GATE>



- In computational complexity theory, the **Cook–Levin theorem**, also known as **Cook's theorem**, states that the Boolean satisfiability problem is NP-complete. That is, it is in NP, and any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the Boolean satisfiability problem.
- An important consequence of this theorem is that if there exists a deterministic polynomial time algorithm for solving Boolean satisfiability, then every NP problem can be solved by a deterministic polynomial time algorithm. The question of whether such an algorithm for Boolean satisfiability exists is thus equivalent to the P versus NP problem, which is widely considered the most important unsolved problem in theoretical computer science.



Stephen A. Cook 1968



Stephen A. Cook 2008

<http://www.knowledgegate.in/GATE>

- During his PhD, Cook worked on complexity of functions, mainly on multiplication. In his seminal 1971 paper "The Complexity of Theorem Proving Procedures", Cook formalized the notions of polynomial-time reduction (a.k.a. Cook reduction) and NP-completeness, and proved the existence of an NP-complete problem by showing that the Boolean satisfiability problem (usually known as SAT) is NP-complete.
- This theorem question asks whether every optimization problem whose answers can be efficiently verified for correctness/optimality can be solved optimally with an efficient algorithm.
- Cook conjectures that there are optimization problems (with easily checkable solutions) which cannot be solved by efficient algorithms, i.e., P is not equal to NP. This conjecture has generated a great deal of research in computational complexity theory, which has considerably improved our understanding of the inherent difficulty of computational problems and what can be computed efficiently. Yet, the conjecture remains open.
- In 1982, Cook received the Turing award for his contributions to complexity theory.

- Graph coloring
- 3-Sat
- Cliuqe
- Hamiltonian circuit
- Knapsack problem
- Vertex cover
- Set cover
- Partition problem
- Independent set
- Travelling salesman problem
- Job scheduling

Approximation Algorithm

- An approximation algorithm is a type of algorithm used for solving optimization problems. The key characteristics of an approximation algorithm are:
 - **Purpose:** It is used when finding the exact solution is too time-consuming, complex, or when an exact solution may not even be necessary.
 - **Performance:** It quickly finds a solution that is close to the best possible answer, or "optimal solution."
 - **Guarantee:** It provides a provable guarantee on how close the output is to the optimal solution, usually expressed as a factor of the optimal value.
 - **Complexity:** Often, the problems tackled by approximation algorithms are NP-hard, meaning that no known polynomial-time algorithms can solve these problems to optimality.
 - **Use Cases:** They are commonly used in fields such as operations research, computer science, and engineering for problems like scheduling, packing, and routing where exact solutions are less feasible as the problem size grows.
 - The concept of approximation is fundamental in scenarios where a perfectly precise answer is either impossible or impractical to obtain, and thus a solution that is "good enough" is acceptable given the constraints of time and resources.

Feature	Approximation Algorithm	Deterministic Algorithm
Outcome	Produces a solution close to the optimal, with some error margin.	Produces a consistent and exact output for a given input.
Predictability	The solution quality is predictable, but the exact output may vary.	The output is entirely predictable and does not vary.
Optimization Problems	Often used for optimization problems where exact solutions are hard.	Used for problems where an exact solution is required and feasible.
Nature	Can be either deterministic or non-deterministic.	Always deterministic; follows a single path to a solution.
Resource Usage	Designed to find a good solution within reasonable time and resources.	Time and resources depend on algorithm complexity and input size.

- The approximation ratio of an algorithm for an optimization problem is a measure of how close the algorithm's solution is to the optimal solution.
- Showing that a particular algorithm for TSP is a 2-approximation means proving that the length of the tour produced by the algorithm is no more than twice the length of the optimal tour.
- Vertex cover problem is also 2-approximate.

Randomized algorithms

- Randomized algorithms are algorithms that make random choices during their process to simplify the solution of complex problems. They are particularly useful in situations where the input is large or the algorithm needs to be immune to certain worst-case scenarios that can cause deterministic algorithms to perform poorly.
 - **Simplification of Algorithms:** They can simplify complex algorithms by replacing a deterministic step with a random one.
 - **Performance:** On average, they often perform better than their deterministic counterparts. Their performance is typically analyzed in terms of expected running time or probability of correctness.

- **Las Vegas algorithms**: They always produce the correct result, but the amount of time they take to complete may vary. For example, the Quickselect algorithm for finding the k-th smallest element in an array.
- **Monte Carlo algorithms**: They have a chance of producing an incorrect result but run in guaranteed polynomial time. An example is the Monte Carlo algorithm for primality testing.
- **Applications**: They are used in various fields, including cryptography, numerical analysis, data structures, optimization, and more.
- **Advantages**: They can avoid bad worst-case scenarios by randomizing choices, making it unlikely to encounter the worst case. They are often easier to implement and understand than their deterministic counterparts.

Aspect	Las Vegas Algorithm	Monte Carlo Algorithm
Result Accuracy	Always accurate; results are either correct or the algorithm does not provide an answer.	Results are probabilistically accurate; there's a non-zero chance of error.
Execution Time	Variable; can take a long time to find a solution, but the solution is correct.	Typically faster; provides approximate solutions in a reasonable time frame.
Use Cases	Used in scenarios where correctness is crucial, like cryptographic algorithms.	Suitable for problems where approximate solutions are acceptable, like in simulations and optimizations.
Termination	Guaranteed to terminate with a correct solution or no solution at all.	Usually terminates in a fixed amount of time, but may not always provide a precise solution.
Examples	Algorithms for exact string matching, like the Rabin-Karp algorithm.	Algorithms for numerical integration, simulations in physics, and optimization problems.