LECTURE NOTES PREPARED BY Mrs. SWARNALATA SAHOO

3rd Semester, Computer Science & Engineering

ALGORITHMS(TH-5)

UNIT: 1

INTRODUCTION TO ALGORITHM

Algorithm : An algorithm is a step-by-step procedure or set of rules used to solve a specific problem or perform a particular task. It's a finite sequence of well-defined instructions that transforms input into output.

Criteria of algorithm:

1. **Input:** A good algorithm must accept inputs. It defines what data the algorithm expects for processing. *Example:* In a sorting algorithm, the input might be an unsorted list of numbers.
2. **Output:** A good algorithm should produce output, which is the result of the input processing. *Example:* In a sorting algorithm, the output would be a sorted list of numbers.
3. **Finiteness:** The algorithm must terminate after a finite number of steps. An infinite loop or non-terminating algorithm is not valid. *Example:* The **bubble sort** algorithm terminates once all elements are sorted, ensuring finiteness.
4. **Definiteness:** Each step in the algorithm must be clear and unambiguous. The instructions should specify exactly what needs to be done. *Example:* In the **binary search** algorithm, the step to check if the middle element is equal to the target should be clear and well-defined.
5. **Effectiveness:** The steps must be feasible and simple enough to be carried out with the available computational resources.

Iterative and recursive algorithm:

Iterative algorithm uses loops (for, while, do-while) to repeat the steps.

Recursive algorithm calls itself to break the problems into smaller sub problems.

Algorithm and pseudo code :

An algorithm is a step-by-step procedure developed for solving a given problem. An algorithm follows a systematic and a logical approach, where the procedure is defined step-wise.

Whereas pseudo code is an informal method of developing an algorithm. It does not have any specific syntax to follow. Pseudo code represents an algorithm to solve a problem in natural language and mathematical notations.

Difference between algorithm and program :

Algorithm	Program
A step-by-step procedure to solve a problem or perform a task.	A set of instructions written in a programming language for execution by a computer.
Abstract and conceptual.	Concrete and implemented in code.
It is written using plain English and can be understood by those from a non- programming	It could be written in any programming language, such as Python, Java, C++, JavaScript, or any other language, depending on the task the program is designed for.

background.	
Describes the logic to solve a problem.	Implements the logic to perform tasks.
Independent of any programming language or platform.	Depends on the programming language and the target platform.

Write an algorithm for Finding sum of natural number

Step 1: Start
Step 2: Read n.
Step 3: initialize i=1 , sum=0
Step 4: if(i>n) go to step 8
Step 5: sum = sum + i
Step 6: i = i + 1
Step 7: Go to step 4
Step 8: Display the result sum Step 9: Stop

Write an algorithm to find the sum of digits of a number.

Step 1: Start
Step 2: Read n.
Step 3: initialize sum=0
Step 4: remainder=n%10
Step 5: sum = sum + remainder
Step 6: n=n / 10
Step 7: if (n>0) repeat steps 4, 5 and 6 Step 8: else go to step 9
Step 9: print sum
Step 10: Stop

UNIT-2
ALGORITHMIC COMPLEXITY

Concept of algorithm complexity:

Time complexity : The time complexity of a program is defined as the amount of computer time needed to complete the execution of a program. It depends on the type of problem to be solved. It comprises of two types:

1. Compile time
2. Run time

space complexity : Space complexity is defined as the amount of memory space consumed by the program during execution. Generally three types of spaces are considered for determining the amount of space used by the algorithm.

1. Instruction space
2. Data space
3. Environmental space

Best-case time complexity: The minimum time required by an algorithm for the most favorable input. **Example:** In a **linear search**, the best case occurs when the target element is at the first position, so the algorithm terminates after 1 comparison. The best- case time complexity is **O(1)**.

Worst-case time complexity: The maximum time required by an algorithm for the least favorable input. **Example:** In **linear search**, the worst case occurs when the target element is at the last position or not in the list at all, requiring **O(n)** comparisons.

Average-case time complexity: The expected time required for typical input. It is usually computed by averaging the time complexities over all possible inputs, assuming a uniform distribution of inputs. **Example:** For **quick sort**, the average-case time complexity is **O(n log n)**, as the algorithm typically splits the input into approximately equal parts.

Big-O notation:

Big-O notation describes the upper bound of the time or space complexity of an algorithm, providing an asymptotic analysis of its performance as the input size grows. It expresses the worst-case scenario of an algorithm's efficiency.

Example:

Linear Search:

The time complexity of a linear search is **O(n)** because in the worst case, it might need to check each element in a list of n elements.

```
def linear_search(arr, target): for i in range(len(arr)):
```

```
    if arr[i] == target: return i
return -1
```

The algorithm's time complexity is proportional to n, so
it is **O(n)**.

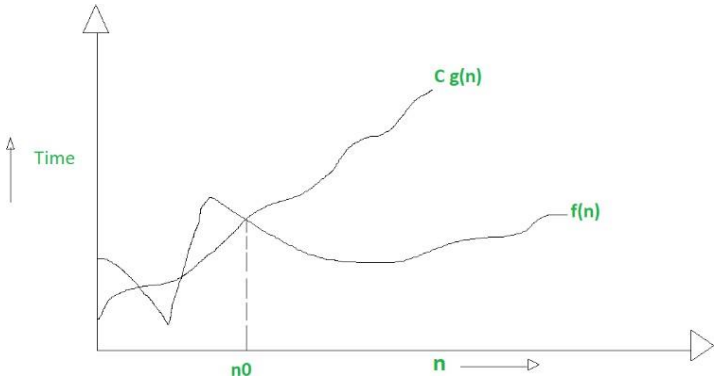
Big-O helps evaluate how an algorithm's runtime grows
with the input size, allowing comparisons between algorithms.

Big O notation (O):

It defines an upper bound on order of growth of time taken by an algorithm or code with input size. Mathematically, if **f(n)** describes the running time of an algorithm; **f(n)** is **O(g(n))** if there exist positive constant **C** and **n0** such that,

0 <= f(n) <= Cg(n) for all n >= n0

n = used to give upper bound a function.



<u>Asymptotic Notation</u> :

Asymptotic Notations are mathematical tools used to analyze the performance of algorithms by understanding how their efficiency changes as the input size grows.

These notations provide a concise way to express the behavior of an algorithm's time or space complexity as the input size approaches infinity.

Following asymptotic notations are used to calculate the running time complexity of an algorithm.

1. Big Oh Notation(O)
2. Big omega Notation(Ω)
3. Big theta Notation(Θ)
4. Little Oh Notation(o)
5. Little omega Notation(ω)

Big Oh(O): This notation is the formal way to express the upper bound of an algorithm's running time. is the most commonly used notation. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

A function f(n) can be represented is the order of g(n)that is O(g(n)), if there exists a value of positive integer n as n_0 and a positive constant c such that: $f(n) \leqslant c.g(n)$ for n>n0 in all case

example:

$f(n)=4n^3+ 5n^2+6n+7$

Considering $g(n)=n^3$

$f(n) \leqslant 5.g(n)$ for all the values of n>2

Hence, the complexity of f(n) can be represented as O(g(n)), i.e. $O(n^3)$

Big Omega (Ω): This notation Ω(n) is the formal way to express the lower bound of an algorithm's running

time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

$f(n)=\Omega(g(n))$ when there exists constant c that $f(n)\geqslant c.g(n)$ for all sufficiently large value of n. Here c is a positive integer. It means function g is a lower bound for function f ; after a certain value of n, f will never go below g .

Example: $f(n)=4n^3+5n^2+6n+7$

Considering $g(n)=n^3$, $f(n)\geqslant 4.g(n)$ for all the values of $n>0$.

Hence, the complexity of f(n) can be represented as $\Omega(g(n))$, i.e. $\Omega(n^3)$

Theta (Θ): The notation (n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. Some may confuse the theta notation as the average case time complexity; while big theta notation could be almost accurately used to describe the average case, other notations could be used as well.

$f(n)=\theta(g(n))$ when there exist constants c_1 and c_2 that $c_1.g(n)\leqslant f(n)\leqslant c_2.g(n)$ for all sufficiently large value of n. Here n is a positive integer.

Example: Let $f(n)=4n^3+5n^2+6n+7$

Considering $g(n)=n^3$, $4.g(n)\leqslant f(n)\leqslant 5.g(n)$ for all the large values of n.

Hence, the complexity of f(n) can be represented as $\theta(g(n))$, i.e. $\theta(n^3)$

Little Oh(o): Littleo notation is used to denote an upper bound that is not asymptotically tight.o(g(n) (little-oh of g of n) is defined as the set f(n)=o(g(n)) for any positive constant c>0 and there exists a value n0>0, such that $0\leqslant f(n)\leqslant c.g(n)$.

In the o-notation, the function f(n) becomes insignificant relative to g(n)as n approaches infinity; that is,

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)=0$$

Example: let $f(n)=4n^3+5n^2+6n+7$ Considering $g(n)=n^4$

$$\lim_{n\to\infty}\left(\frac{4n^3+5n^2+6n+7}{n^4}\right)=0$$

Hence, the complexity of f(n) can be represented as o(g(n)), i.e. $o(n^4)$.

Little Omega(ω)

ω notation is used to denote a lower bound that is not asymptotically tight. We define ω(g(n)) (little-omega of g of n) as the set f(n)= ω(g(n)) for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that f(n)

> c · g(n) for every integer n ≥ n0

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)=\infty$$

Example: let $f(n)=4n^3+5n^2+6n+7$ Considering $g(n)=n^4$

$$\lim_{n\to\infty}\left(\frac{4n^3+5n^2+6n+7}{n^2}\right)=\infty$$

Hence, the complexity of f(n) can be represented as o(g(n)), i.e. $\omega(n^2)$

UNIT- 3
RECURSIVE ALGORITHMS

Recursive algorithm :

A recursive algorithm solves a problem by calling itself with smaller inputs. Recursive solutions are often more elegant for problems that can be broken down into smaller, similar sub problems, but they can be less memory-efficient due to function call overhead.

Distinguish between iteration and recursion :

Iteration uses loops (like for or while) to repeat a block of code, while recursion uses function calls. Both can solve problems that involve repetition, but recursion can be more elegant for problems with a natural recursive structure, while iteration might be more efficient in terms of memory because it avoids the overhead of the call stack.

Write an algorithm to print the Fibonacci series up to n terms

Step 1: Start
Step 2:Declare variables i, t0,t1,t2
Step 3: Initialize the variables, t0=0, t1=1, and t2=0
Step 4: Enter the number of terms of Fibonacci series to be printed
Step 5: Print First two terms of series t0 and t1
Step 6:Use loop for the following steps t2=t0+t1
t0=t1 t1=t2
step 7: increase value of i each time by 1 print the value of t2
Step 8: Stop

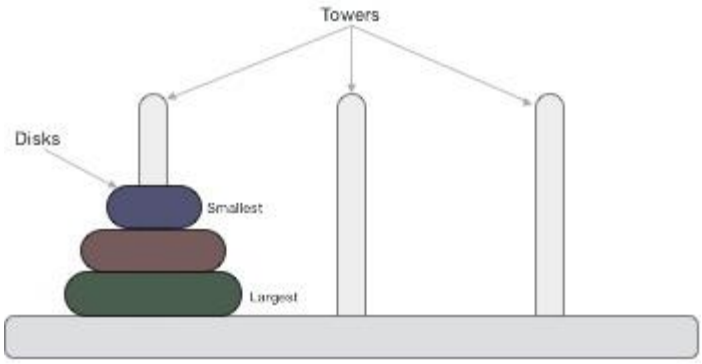
Write an algorithm to find the factorial of a number using recursion

Step 1: Start
Step 2: Read number *n*
Step 3: Call factorial(n)
Step 4: Print factorial f
Step 5: Stop

factorial(n)
Step 1: If n==1 then return 1
Step 2: Else
f=n*factorial(n-1)
Step 3: Return f

TOWER-OF-HANOI

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted –

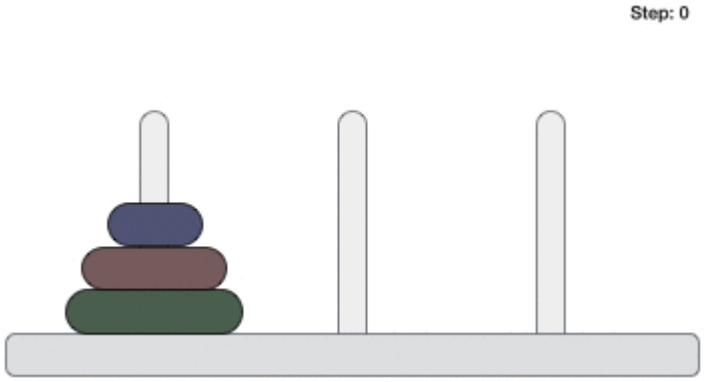


These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are –

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.



Tower of Hanoi puzzle with n disks can be solved in minimum $\mathbf{2^n-1}$ steps. This presentation shows that a puzzle with 3 disks has taken $\mathbf{2^3 - 1 = 7}$ steps.

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks –

- First, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to destination peg.
- And finally, we move the smaller disk from aux to destination peg.

So now, we are in a position to design an algorithm for Tower of Hanoi with more than

two disks. We divide the stack of disks in two parts. The largest disk (n^{th} disk) is in one part and all other (n-1) disks are in the second part.
Our ultimate aim is to move disk **n** from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.
The steps to follow are

```
Step 1 – Move n-1 disks from source to aux Step 2 – Move nth disk from
source to dest Step 3 – Move n-1 disks from aux to dest
```

A recursive algorithm for Tower of Hanoi can be driven as follows –

```
START
Procedure Hanoi(disk, source, dest, aux)
 IF disk == 1, THEN
 move disk from source to dest
 ELSE
Hanoi(disk - 1, source, aux, dest) // Step 1
move disk from source to dest // Step 2
 Hanoi(disk - 1, aux, dest, source) // Step 3
END IF END Procedure
 STOP
```

Advantages and disadvantages of recursion

Advantages of recursion

- **Simpler and more elegant code:** For problems that can be broken down into smaller, self-similar subproblems, recursion can provide a more intuitive and easier-to-understand solution compared to other methods.
- **Reduced code length:** Recursive solutions often require fewer lines of code, making the code more concise.
- **Better for certain data structures:** It is particularly useful for problems involving recursive data structures like trees, linked lists, and graphs.
- **Easier to debug:** For some developers, the logical flow of a recursive function can be easier to trace than a complex iterative one.
- **Avoids side effects:** It can be more functional, as it doesn't require changing the value of a variable in the same way an iterative loop might.

Disadvantages of recursion

- **Higher memory consumption:** Each recursive call adds a new frame to the call stack, which can lead to higher memory usage than iterative solutions.
- **Slower performance:** The overhead from function calls can make recursion slower than an equivalent iterative solution.
- **Stack overflow:** If the base case is not properly defined, the function may call itself indefinitely, leading to a stack overflow error that crashes the program.
  Difficult to trace: For those not familiar with the pattern, the logic of a recursive function can be difficult to follow.

UNIT-4
ALGORITHM PARADIGM

Greedy algorithm:

The **Greedy algorithm paradigm** is an approach where decisions are made by choosing the best available option at each step without considering the global context. This choice is made with the hope that local optimization leads to global optimization.

Example:

Coin Change Problem (Greedy approach): Given a set of coin denominations (say {1, 5, 10, 25}) and a target amount, the greedy approach chooses the largest coin denomination that is less than or equal to the remaining amount, repeatedly, until the amount is reduced to zero.

Steps:

1. Start with the target amount (e.g., 63).
2. Choose the largest denomination that doesn't exceed the target (in this case, 25).
3. Subtract 25 from 63 (remaining 38).
4. Choose 25 again, subtracting from 38 (remaining 13).
5. Choose 10 (remaining 3).
6. Choose 1 (remaining 2).
7. Choose 1 again (remaining 1).
8. Choose 1 once more (remaining 0).

Result: **63 = 25 + 25 + 10 + 1 + 1 + 1** (using 6 coins).
Greedy Algorithms are optimal for problems that exhibit the "greedy choice property" and "optimal substructure."

Divide and Conquer approach :

The **Divide and Conquer** paradigm solves a problem by dividing it into smaller subproblems, solving each subproblem recursively, and then combining the results of these subproblems to solve the original problem.

Example:

Merge Sort (Divide and Conquer):

1. Divide the list into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves to produce a sorted list.

Steps of Merge Sort:

- Split the list into two halves.
- Recursively divide the halves until you reach lists of one element.
- Merge the lists back together in sorted order.
- Continue merging until the entire list is sorted.

Time Complexity:O(n log n), where n is the size of the list.

Difference from Greedy:

Greedy makes choices at each step based on the immediate benefit, while divide and conquer breaks down the problem into subproblems, solves them independently, and combines them for the final solution.

Branch and bound algorithms

The general process of a Branch and Bound algorithm can be summarized as follows:

1. Initialize the problem with the entire solution space.
2. Select a branching variable and create subproblems (branches).
3. Calculate bounds for each subproblem.
4. Prune branches that cannot lead to a better solution than the current best.
5. Select the most promising branch to explore further.
6. Repeat steps 2-5 until the optimal solution is found or all branches are explored.

Advantages of Branch and Bound approach:

Branch and Bound algorithms offer several advantages in problem- solving:

- **Optimality:** They guarantee finding the optimal solution if one exists.
- **Efficiency:** By pruning unnecessary branches, they can significantly reduce the search space.
- **Flexibility:** The algorithm can be adapted to various problem types and domains.
- **Anytime property:** They can provide increasingly better solutions as the algorithm progresses, even if interrupted before completion.

Common Applications of Branch and Bound

Branch and Bound algorithms find applications in various fields and problem types:

1. Traveling Salesman Problem (TSP)

The TSP is a classic optimization problem where the goal is to find the shortest possible route that visits each city exactly once and returns to the starting city. Branch and Bound is often used to solve TSP instances efficiently.

2. Knapsack Problem

In the Knapsack Problem, we need to select a subset of items with maximum value while respecting a weight constraint. Branch and Bound can be applied to find the optimal solution.

3. Job Scheduling

Branch and Bound can be used to optimize job scheduling problems, where tasks need to be assigned to resources in the most efficient manner.

4. Integer Programming

Many integer programming problems can be solved using Branch and Bound techniques, especially when combined with linear programming relaxations.

Dynamic Programming:

Dynamic Programming (DP): Dynamic programming is a method used to solve problems by breaking them into smaller subproblems, solving each subproblem just once, and storing the results for reuse (memoization or tabulation). DP is used when the problem has overlapping subproblems, meaning the same subproblems are solved multiple times.

Example:Fibonacci Sequence (DP)

A naive recursive solution to calculate the nth Fibonacci number would involve recalculating the same values many times. Dynamic programming solves this by storing the results of subproblems.

Recursive Fibonacci:

```
deffib(n):
if n <= 1:
return n
return fib(n-1) + fib(n-2)
```

Dynamic Programming Version:

```
deffib(n):
dp = [0] * (n + 1) dp[1] = 1
foriinrange(2, n+1): dp[i] = dp[i-1] + dp[i-2] returndp[n]
```

Difference from Divide and Conquer:

Divide and con

quer breaks the problem into smaller, independent subproblems and solves them recursively, while DP stores solutions to subproblems and avoids redundant calculations.
Time Complexity (DP):O(n), as it only computes each Fibonacci number once.

Longest common sub sequence with an example:

Longest Common subsequence

The longest common subsequence (LCS) problem can be formulated as follows

"Given twosequences $X = \langle x_1, x_2 \ldots\ldots x_n \rangle$ and $Y = \langle y_1, y_2 \ldots\ldots y_n \rangle$ and the objective is to find the LCS $Z = \langle z_1, z_2, \ldots\ldots z_n \rangle$ that is common to x and y".

Given two sequences X and Y, we say Z is a common sub sequence of X and Y if Z is a subsequence of both X and Y. For example, X = < A, B, C, B, D, A, B > and Y = < B, D, C, A, B, A>, the sequence <B, C, A> is a common subsequence. Similarly, there are many common

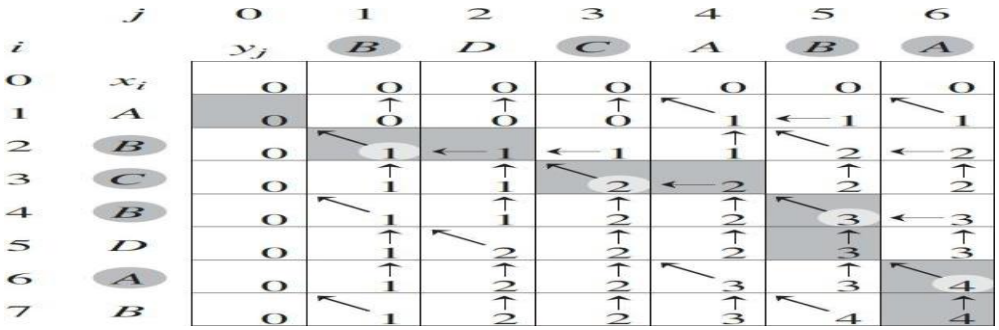
subsequences in the two sequences X and Y. However, in the longest common subsequence problem, we wish to find a maximum length common subsequence of X and Y, that is < B, C, B, A> or < B, D, A, B>.

Dynamic Approach algorithm:

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tabl
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
8  else PRINT-LCS(b, X, i, j − 1)
```

Example:

Given X=<ABCBDAB> and Y=< BDCABA>



i		j	0	1	2	3	4	5	6
			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	0	0	0	1	1	1
2	B		0	1	1	1	1	2	2
3	C		0	1	1	2	2	2	2
4	B		0	1	1	2	2	3	3
5	D		0	1	2	2	2	3	3
6	A		0	1	2	2	3	3	4
7	B		0	1	2	2	3	4	4

Here the Length of the LCS is 4 and the LCS is B C B A

Differenciate between Dynamic programming and Greedy Approach

Dynamic Programming	Greedy Method
1. Dynamic Programming is used to obtain the optimal solution.	1. Greedy Method is also used to get the optimal solution.
2. In Dynamic Programming, we choose at each step, but the choice may depend on the solution to sub- problems.	2. In a greedy Algorithm, we make whatever choice seems best at the moment and then solve the sub- problems arising after the choice is made.
3. Less efficient as compared to a greedy approach	More efficient as compared to dynamic programming approach.
4. Example: 0/1 Knapsack	4. Example: Fractional Knapsack
5. It is guaranteed that Dynamic Programming will generate an optimal solution using Principle of Optimality.	5. In Greedy Method, there is no such guarantee of getting Optimal Solution.

UNIT-5
SORTING

Sorting and searching:

Sorting refers to the operation of arranging data in some given order. Searching refers to the operation of searching the particular record from the existing information.

linear search:

In Linear Search the list is searched sequentially and the position is returned if the key element to be searched is available in the list, otherwise -1 is returned. The search in Linear Search starts at the beginning of an array and move to the end, testing for a match at each item.

Binary search :

Binary search is simpler and faster than linear search. Binary search the array to be searched is divided into two parts, one of which is ignored as it will not contain the required element One essential condition for the binary search is that the array which is to be searched, should be arranged in order.

Bubble Sort :

Bubble Sort works by repeatedly comparing adjacent elements of a list and swapping them if they are in the wrong order. This process is repeated until no more swaps are needed, indicating the array is sorted. Each pass ensures that the largest unsorted element bubbles up to its correct position at the end of the list.

Step1: start

Step2: Check if the first element in the input array is greater than the next element in the array.

Step3: If it is greater, swap the two elements; otherwise move the pointer forward in the array.

Step4: Repeat Step 2 until we reach the end of the array.

Step5: Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.

Step6: Display the sorted array Step7: stop

Code

```
voidbubbleSort(int numbers[], intarray_size)
{
inti, j, temp;
 for (i = (array_size - 1); i>= 0; i--) for (j = 1; j <= i; j++)
     if (numbers[j-1] > numbers[j])
       {
         temp = numbers[j-1]; numbers[j-1] = numbers[j]; numbers[j] =
```

```
        temp;
      }
}
```

Selection sort :

In the selection sort we first find the smallest value in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and we continue the process in this way until the entire array is sorted.

1. Set MIN to location 0.

2. Search the minimum element in the list.

3. Swap with value at location MIN.

4. Increment MIN to point to next element.

5. Repeat until the list is sorted.

Algorithm: Selection-Sort (A)

For i← 1 to n-1 do

min j ←i; min x ← A[i]

for j ←i + 1 to n do if A[j] < min x then min j ← j

min x ← A[j]

A[min j] ← A [i] A[i] ← min x

The time complexity of the selection sort algorithm is **O(n^2).**

Insertion Sort:

It builds the sorted array one element at a time by repeatedly inserting the next element in its correct position in the sorted portion of the list. It has a **time complexity of O(n^2)** in the worst case but **O(n)** in the best case when the array is already sorted. **Insertion Sort** is **stable** because equal elements retain their relative order.

Merge Sort

Input: array of numbers to be sorted

Output: sorted array
Step1: start
Step2: if(p<r)
Step3: q = (p + r) / 2
Step4: Call merge_sort(A,p,q)
Step5: Call merge_sort(A,q+1,r)
Step6: Call merge(A,p,q, r)
[end of if] Step7: stop

Merge(A,p,q,r)
Step1: i=p, j = q + 1, k = 1
Step2: while (i < q && j <r)
Step3: if (A[i] < A[j])
Step4: B[k] = A[i]
i + + k + +
Step5: else
Step6: B[k] = A[j]
j + +
k + +
/* end of while loop*/
Step7: if (i < q)
Step8: repeat from m = i to q
Step9: B [k] = A[m]
k + +
Step10: else
Step11: repeat from m = j to r
Step12: B [k] = A[m]
k + +
step 13: repeat from m = p to r
Step14: A [m] = B[m]
Step15: stop

Quick sort:

Algorithm Quick_sort(A,p,r)

Input: array of numbers to be sorted
Output: sorted array
Step1: start
Step2: if(p<r)

a. Partition(A,p,r,q)
b. Call Quick_sort(A,p,q)
c. Call Quick_sort(A,q+1,r) Step3: stop

Partition(A,p,r)

Step1: set i=p,j=r,x=A[p]
Step2: while(i<j)
Step3: repeat until(A[i]<x) j=j+1
Step4: repeat until (A[i]>=x) i=i+1
step5: if(i<j)
temp=A[i]
A[i] = A[j]
A[j] = temp
Step6: else
temp=A[j] A[j]=A[p]
A[p] = temp return (j)
(end of while)

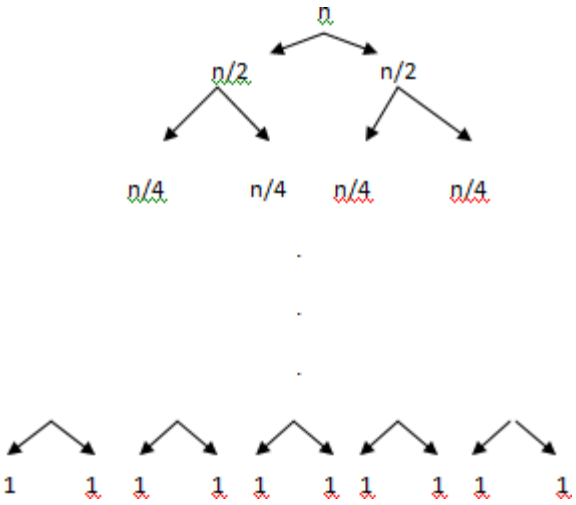
Performance of Qick_sort

The performance of Quick sort depends upon the selection of pivot element.
Worst Case:

The worst case performance of Quick sort occurs if the pivot divides the array in such a way that the first part contains 1 element and the other part contains (n-1) elements.
The worst case time complexity is $\theta(n^2)$

Best case

If the partitioning procedure produces 2-sublists of size n/2 Quick sort runs much faster. In each step the original array is divided into two equal half.



Suppose k number of steps are required to split the array such that each sub array will contain only 1-element

Hence, $n/2^k = 1$

$\Rightarrow n = 2^k$

$\Rightarrow \log_n = \log 2^k = k\log 2$

$\Rightarrow k = \lg n$

As there will be n numbers sub arrays each containing only one element and to produse the sub array K numbers of steps are required . hence we can write T(n) = n*k

$T(n) = \theta(n\lg n)$

Average case

Since most of the time the pivot element divides the array to produce 2-sub arrays, hence the total time is O(nlgn)

T(n) = O(nlgn)

Algorithm for linear search :

Step1: Start from the 0th index of the input array, compare the key value with the value present in the 0th index.

Step2: If the value matches with the key, return the position at which the value was found.

Step3:If the value does not match with the key, compare the next element in the array.

Step4:Repeat Step 3 until there is a match found. Return the position at which the match was found.

Step5: If it is an unsuccessful search, print that the element is not present in the array and exit the program.

```
linear_search (list, value)

for each item in the list

if match item == value
return the item's location
end if
end for end procedure
```

searching :

Searching refers to the operation of searching the particular record from the existing information.

Algorithm for binary search:

Binary search looks for a particular key value by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. But if the middle item has a value greater than the key value, the right sub-array of the middle item is searched. Otherwise, the left sub-array is searched. This process continues recursively until the size of a subarray reduces to zero.

Algorithm binary_search

```
A ← sorted array
n ← size of array
x ← value to be searched
Set lowerBound = 1
Set upperBound = n
while x not found
        if upperBound < lowerBound
        EXIT: x does not exists.
set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
 if( A[midPoint] < x )
   set lowerBound = midPoint + 1
   if (A[midPoint] > x)
    set upperBound = midPoint – 1
    if A[midPoint] = x
      EXIT: x found at location midPoint
end while
```

Binary search tree:

A Binary Search Tree is a Binary Tree where every node's left child has a lower value, and every node's right child has a higher value.

What is the difference between a Binary Search Tree and a Balanced Search Tree

In a **Binary Search Tree (BST)**, there is no guarantee that the tree will remain balanced, which can result in a worst-case time complexity of **O(n)** for search operations. A **Balanced Search Tree**, such as an AVL tree or Red-Black Tree, maintains balance, ensuring logarithmic time complexity for operations.

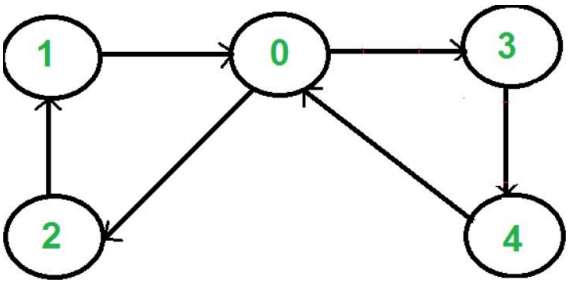
Hash Table :

A **Hash Table** is a data structure that uses a hash function to map keys to values. It allows for efficient insertion, deletion, and search operations, with average time complexity of **O(1)**.

UNIT-6
GRAPH

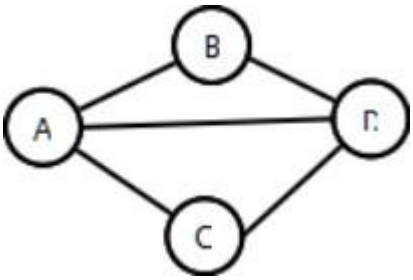
Directed graph :

A **directed graph (or digraph)** is a graph where edges have a direction. Each edge is an ordered pair of vertices, meaning there is a specific start and end vertex. It is represented as (u,v), where u is the starting vertex and v is the ending vertex.



Undirected graph :

An **undirected graph** is a graph in which the edges do not have any direction. Each edge is an unordered pair of vertices, meaning the relationship is bidirectional.



Differences between a directed and an undirected graph in terms of edges, degree, and applications

Edges: In a **directed graph**, edges have direction, represented as ordered pairs (u,v), meaning there is a specific starting vertex u and an ending vertex v. In an **undirected graph**, edges are unordered pairs {u,v}, meaning the relationship is bidirectional, and the direction of traversal doesn't matter.

Degree: In a directed graph, each vertex has **in-degree** (number of edges directed toward it) and **out-degree** (number of edges directed away from it). For example, if a vertex has edges pointing to it from 3 vertices and edges pointing out to 2 vertices, its in-degree would be 3, and its out-degree would be 2. In an undirected graph, each vertex has only a **degree**, which is the number of edges connected to it, regardless of direction.

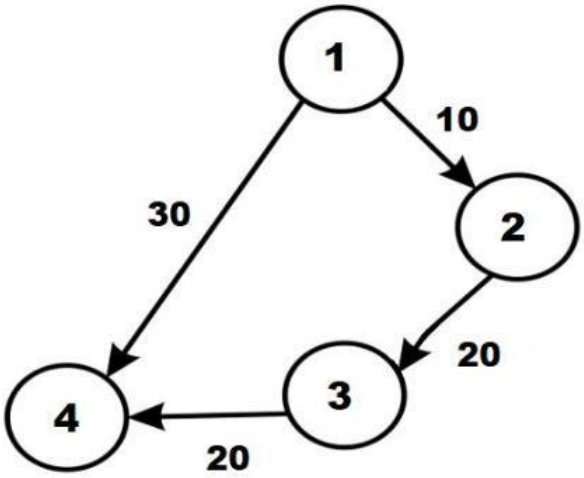
Applications:

Directed Graphs: Used for **web page linking**, where hyperlinks have direction, **social media networks**,

where relationships like "follows" or "likes" are directional, and **task scheduling**, where tasks have precedence (task A must be done before task B).
Undirected Graphs: Used in **social networks** (e.g., friendship connections where the relationship is bidirectional), **computer networks** (to represent communication between nodes that is bidirectional), and **geographical mapping**
(e.g., roads between cities).

weighted graph :
A graph with a value associated with every edge is called a weighted graph. The values corresponding to the edges are called weights. A value in a weighted graph can represent quantities such as cost, distance, and time, depending on the graph. We typically use weighted graphs in modelling computer networks.
An edge in a weighted graph is represented as (u, v, w), where:

- u is the source vertex
- v is the destination vertex
- w represents the weight associated with going from u to v

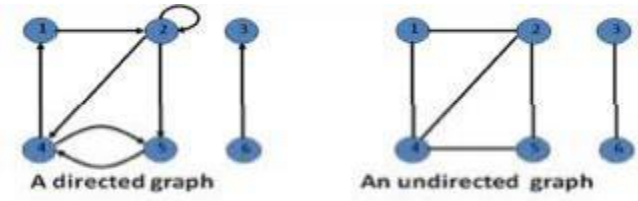


complete graph :
A complete graph is a simple undirected graph where every pair of distinct vertices is connected by a unique edge. This means each vertex is directly connected to every other vertex in the graph.
Graph terminologies :
Edge: An edge is a line connecting two vertices.
Vertex: A vertex is a point where edges meet.
A **directed graph** (or **digraph**) is a graph with **directed edges**.

- Edges have directions so they are represented by arrows.
- Each edge **leaves** a vertex and **enters** a vertex.

- An **undirected graph** is a graph with **undirected edges**.
  - Edges have no directions so they are represented by lines.
  - Self-loops are forbidden.

Edge (u,v) is the same as edge (v,u).



- **Adjacency**
  - If (u,v) is an edge, vertex v is **adjacent** to vertex u.
  - In an undirected graph, adjacency relation is symmetric. If u is adjacent to v, v is adjacent to u.

– In a directed graph, it is not symmetric.
  - Vertex 2 is adjacent to 1.
  - But vertex 1 is not adjacent

Degree (Directed Graph)

  – The **out-degree** of a vertex is the number of edges leaving it.

- The out-degree of vertex 2 is 3.

  – The **in-degree** of a vertex is the number of edges entering it.

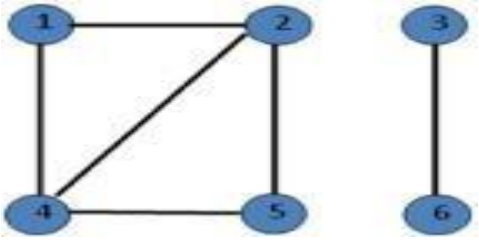
- The in-degree of vertex 2 is 2.

  – **degree=out-degree+ in-degree.**

- **Degree**

– In an undirected graph,
  - The out-degree and the in-degree are not defined.
  - Only the degree of a vertex is defined.

– The degree of vertex 2 is 3.



Loop

A loop occurs when an edge connects a vertex to itself.

Weighted graph: Weighted graphs assign numerical values (weights) to edges. **Unweighted graph:** An unweighted graph has no edge weights. It focuses solely on connectivity between nodes.

Connected graph: A graph is connected if there is a path between any pair of nodes.

Disconnected graph: A disconnected graph has isolated components that are not connected to each other. These components are separate subgraphs.

Acyclic graph An acyclic graph contains no cycles (closed loops). In other words, you cannot start at a node and follow edges to return to the same node.

Cyclic graph: A cyclic graph has at least one cycle. You can traverse edges and eventually return to the same node.

<u>Difference between a path and a cycle</u>:

A **path** is a sequence of vertices where each edge is traversed exactly once, and no vertex is repeated.

A **cycle** is a path that starts and ends at the same vertex, with at least one edge, and no vertex (except the start and end) is repeated.

Difference: A path may or may not return to the starting point, while a cycle always returns to the starting vertex.

spanning tree :

A **spanning tree** of a connected, undirected graph is a subgraph that includes all the vertices of the original graph, with the minimum number of edges needed to keep the graph connected, i.e., **V - 1 edges** for **V vertices**. It is a tree because it has no cycles.

Directed Acyclic Graph:

A **Directed Acyclic Graph (DAG)** is a directed graph with no cycles, meaning there is no way to start at a vertex and follow a sequence of edges that leads back to the same vertex.

Topological sorting:

Topological sorting is a linear ordering of the vertices in a Directed Acyclic Graph (DAG) such that for every directed edge **(u, v)**, vertex **u** comes before **v** in the ordering. It is used for scheduling tasks or resolving dependencies.

The two main steps involved in the topological sort algorithm include:

- Selecting a node with zero in-degree
- Deleting N from the graph along with its edges

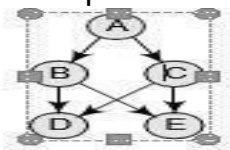
Step 1: Find the in-degree INDEG(N) of
every node in the graph
Step 2: Enqueue all the nodes with a zero in-degree
Step 3: Repeat Steps 4 and 5 until the QUEUE is empty

Step 4: Remove the front node N of the QUEUE by setting FRONT = FRONT + 1
Step 5: Repeat for each neighbour M of node N:

a) Delete the edge from N to M by setting INDEG(M) = INDEG(M) – 1

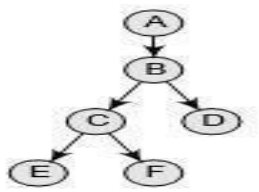
b) IF INDEG(M) = , then Enqueue M, that is, add M to the rear of the queue [END OF INNER LOOP]
[END OF LOOP]

Example:



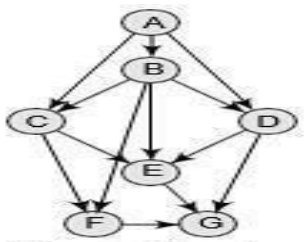
Topological sort can be given as:

- A, B, C, D, E
- A, B, C, E, D
- A, C, B, D, E
- A, C, B, E, D



Topological sort can be given as:

- A, B, D, C, E, F
- A, B, D, C, F, E
- A, B, C, D, E, F
- A, B, C, D, F, E
- A, B, F, E, D



Topological sort can be given as:

- A, B, C, F, D, E, C
- A, B, C, D, E, F, G
- A, B, C, D, F, E, G
- A, B, D, C, E, F, G
- A, B, D, C, F, E, G

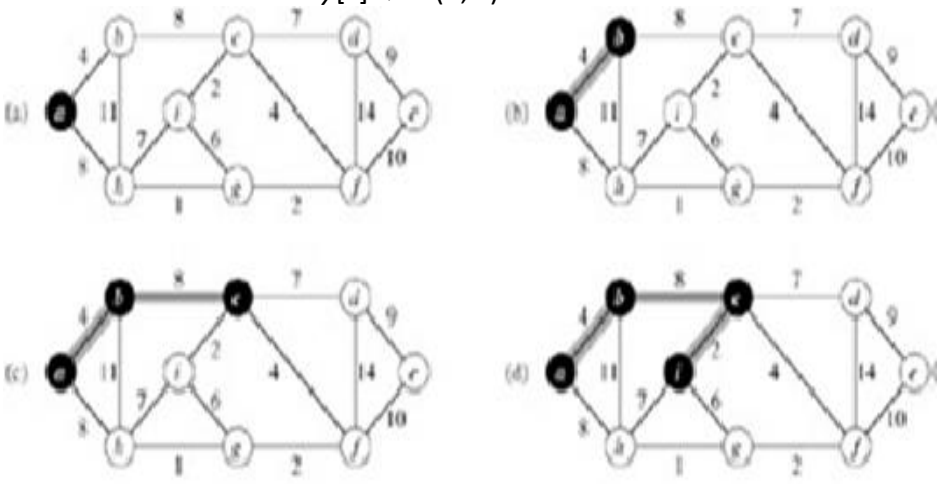
prim's algorithm :

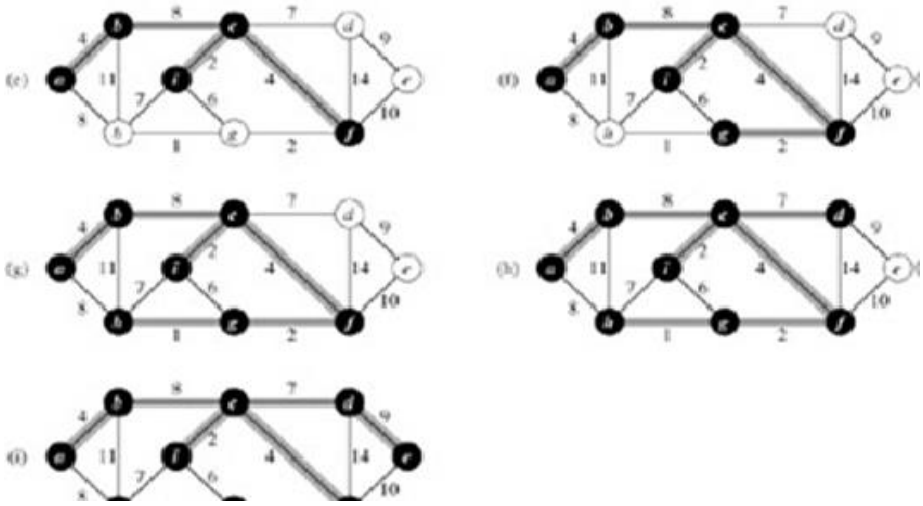
The tree starts from an arbitrary root vertex r and grows until the tree spans all the vertices in V. Each step adds to the tree *A*a light edge that connects A to an isolated vertex—one on which no edge of *A* is incident.

- Select any vertex
- Choose the minimum cost adjacent edge of selected vertex
- Repeat step2 until all node are not explored.

MST-PRIM(*G*, *w*, *r*)

1. **for** each *u* ϵ *V* [*G*]
2. **do** *key*[*u*] ← ∞
3. π[*u*] ← NIL
4. *key*[*r*] ← 0
5. *Q* ← *V* [*G*]
6. **while** *Q* ≠ Ø
7. **do** *u* ← EXTRACT-MIN(*Q*)
8. **for** each *v* ϵ *Adj*[*u*]
9. **do if** *v* ϵ *Q* and *w*(*u*, *v*) < *key*[*v*]
10. **then** π[*v*] ← *u*
11. *key*[*v*] ← *w*(*u*, *v*)





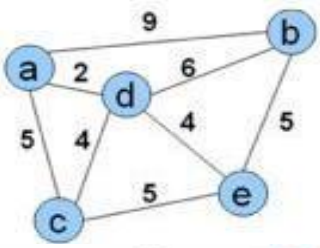
Kruskal's algorithm:

It uses a disjoint-set data structure to maintain several disjoint sets of elements. Each set contains the

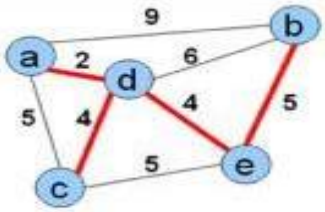
vertices in one tree of the current forest. The operation FIND-SET(u) returns a representative element from the set that contains u. Thus, we can determine whether two vertices u and v belong to the same tree by testing whether FIND-SET(u) equals FIND- SET(v). To combine
trees, Kruskal's algorithm calls the UNION procedure.

MST-KRUSKAL(*G*, *w*)

1. $A \leftarrow \emptyset$
2. **for** each vertex $v \_ V[G]$
3. **do** MAKE-SET(v)
4. sort the edges of E into nondecreasing order by weight w
5. **for** each edge $(u, v) \_ E$, taken in non decreasing order by weight
6. **do if** FIND-SET(u) ≠ FIND-SET(v)
7. **then** $A \leftarrow A \_ \{(u, v)\}$
8. UNION(u, v)
9. **return** A



$F = \{a\}, \{b\}, \{c\}, \{d\}, \{e\}$
$A = \emptyset$
$E = \{(a,d), (c,d), (d,e), (a,c), (b,e), (c,e), (b,d), (a,b)\}$



$E = \{(a,d), (c,d), (d,e), (a,c), (b,e), (c,e), (b,d), (a,b)\}$

Forest	A
{a}, {b}, {c}, {d}, {e}	∅
{a,d}, {b}, {c}, {e}	{(a,d)}
{a,d,c}, {b}, {e}	{(a,d), (c,d)}
{a,d,c,e}, {b}	{(a,d), (c,d), (d,e)}
{a,d,c,e,b}	{(a,d), (c,d), (d,e), (b,e)}

Minimum spanning tree:

A minimum spanning tree is a sub graph of an undirected weighted graph *G*, such that

- it is a tree (i.e., it is acyclic). it covers all the vertices *V* ,contains *|E| - 1* edges
- the total cost associated with tree edges is the minimum among all possible spanning trees not necessarily unique. Two algorithms used to find MST

Kruskal's Algorithm

Prim's Algorithm

Properties of Minimum Spanning Tree**:**

- A minimum spanning tree connects all the vertices in the graph, ensuring that there is a path between any pair of nodes.
- An **MST** is **acyclic**, meaning it contains no cycles. This property ensures that it remains a tree and not a graph with loops.
- An **MST** with **V** vertices (where **V** is the number of vertices in the original graph) will have exactly **V – 1** edges, where **V** is the number of vertices.

- An **MST** is optimal for minimizing the total edge weight, but it may not necessarily be unique.
- The cut property states that if you take any cut (a partition of the vertices into two sets) in the original graph and consider the minimum- weight edge that crosses the cut, that edge is part of the **MST**.