MODULE - 1

(1) Characteristics of Algorithm

(1) **INPUT** → Every algo. takes input. This input could be anything from no. to strings to complex data str., depending on the problems the algo. is designed to solve.

(2) **OUTPUT** → An algo. produces an output based on the input and its internal logic.

(3) **DEFITENESS** → must be precisely defined and unambiguous.

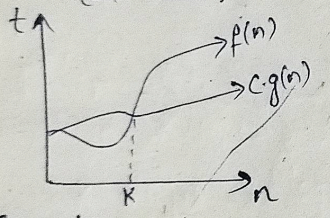
(4) **FINITENESS** → must terminate after a finite no. of steps.

(5) **CORRECTNESS** → Should produce correct output for all possible inputs.

(6) **EFFICIENCY** → Should be designed to be efficient in terms of time and space.

(2) Analysis of Algorithm

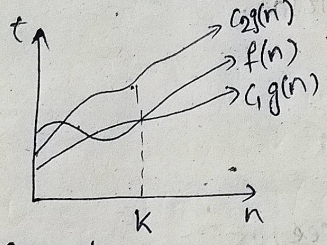
Best case
Big-omega (Ω)
Lower bound
(at least)



Example
for a sorting algo., best case scenario occurs when input is already sorted.

$f(n) = \Omega g(n)$
$f(n) \geq c \cdot g(n)$
$2n^2+n \geq c \cdot n^2$
$2n^2+n \geq 2n^2 \quad (c=2)$

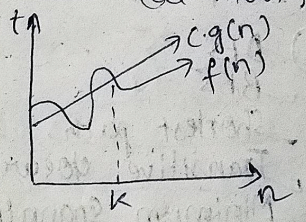
Average Case
Theta (Θ)
Exact time



Example
considers inputs of various distributions, not just the best or worst scenario.

$c_1 g(n) \leq f(n) \leq c_2 g(n)$
$2n^2 \leq 2n^2+n \leq 3n^2$

Worst case
Big-Oh (O)
Upper bound
(at most)



Example
might occur when the input is in reverse order or contains elements with eg. keys.

$f(n) = O g(n)$
$f(n) \leq c \cdot g(n)$
$c > 0$
$n \geq k$
$k \geq 0$

(3) Time and Space trades off

Time-Space trade-off refers to the concept in CSE where a ↓ in the amount of memory (Space) used by an algo. results in an ↑ in the amount of time req. to execute the algo. and vice-versa.

- **Time complexity** → represents the amount of time an algo. takes to complete as a function of the size of its input.
- **Space complexity** → represents amount of memory an algo. uses as a func. of size of its input.

Ex. of Time-Space
→ caching
→ Iterative vs.
→ DS choices
→ Compression

Real world App
#. software de
embedded s
performance

(4) Recursive A

Recursive Algor
itself recursive
subproblems u
directly.

Recurrence Re
recursive algo
inputs.

$T(n) = f(n)$
where,
T(n) → time
n → size
f(n) → time
g(n) → size
T(g(n)) → time

Ex:- Pseudocod
function
if n=
re
else:
rel

Recurrence r
O(1) → rep.
Opera
T(n-1) → rep
In this case,
and solving
algorithm.

Ex. of Time-Space trades-off
→ caching
→ Iterative vs. Recursive Approaches
→ DS choices
→ Compression Algorithm.

Real world Applications
* software dev., database systems, network protocols, and embedded systems, where resources are limited, and performance is critical.

(4) Recursive Algorithm and recurrence relation

Recursive Algorithm → algo. that solves a problem by calling itself recursively, breaking the probl. down into smaller subproblems until they become simple enough to be solved directly.

Recurrence Relation → describes the time complexity of a recursive algo. in terms of its own values for smaller inputs.

$$T(n) = f(n) + T(g(n))$$

where,
$T(n)$ → time complexity of a recursive algo
n → size of input
$f(n)$ → time taken for operations other than recursive call(s).
$g(n)$ → size of subproblem(s) generated by recursive call(s).
$T(g(n))$ → time taken to solve subproblem(s).

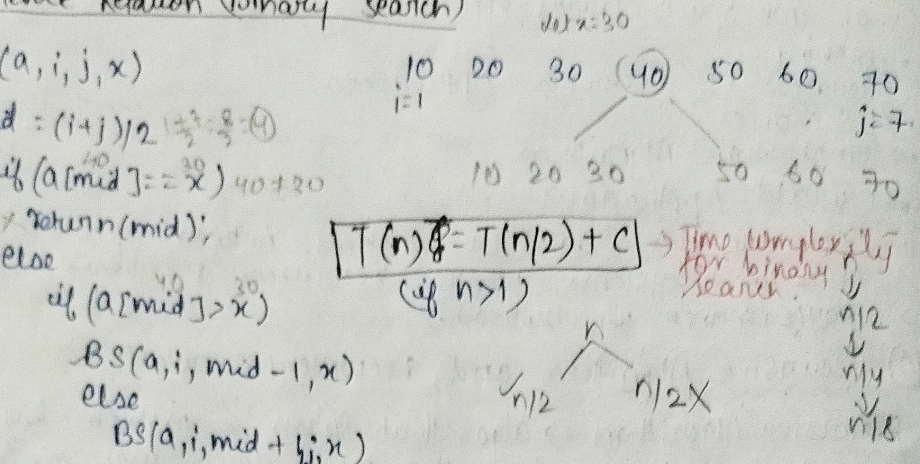
Ex:- Pseudocode for a recursive algo.

```
function factorial(n):
    if n == 0;
        return 1
    else:
        return n * factorial(n-1)
```

Recurrence relation:- $T(n) = O(1) + T(n-1)$
$O(1)$ → rep. the const. time taken for comparison & return operation
$T(n-1)$ → rep. time taken to compute the factorial of $n-1$.
In this case, recurrence rel. describes linear time complexity and solving it would give us exact time complexity of the algorithm.

Recurrence Relation (Binary search)

```
BS(a, i, j, x)
mid = (i+j)/2
if (a[mid] == x)
    return(mid);
else
    if (a[mid] > x)
        BS(a, i, mid-1, x)
    else
        BS(a, i, mid+1, x)
```



$T(n) = T(n/2) + c$ (if $n>1$) → Time complexity for binary search

$n/2 \to n/4 \to n/8$

Substitution Method (some probl. ko solve krta hai)

① $T(n) = \begin{cases} T(n/2)+c & \text{if } n>1 \\ 1 & \text{if } n=1 \end{cases}$

$T(n) = T(n/2) + c$ — (i)

$T(n/2) = T(n/4) + c$ — (ii)

$T(n/4) = T(n/8) + c$ — (iii)

(ii) in (i)

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

$T(n) = T(n/2^2) + 2c$ — (iv)

(iii) in (iv)

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

⋮ (k times)

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

if $n=1$

$T(n) = T(n/n) + kc$

$T(n) = T(1) + kc$

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

$n = 2^k$

$\log n = k \log 2$

$\Rightarrow k = \log n \quad (\because \log 2 = 1)$

② $T(n) = \begin{cases} 1 & \text{if } n=1 \\ n * T(n-1) & \text{if } n>1 \end{cases}$

$T(n) = n * T(n-1)$ — (i)

$T(n-1) = (n-1) * T(n-1-1) = (n-1) * T(n-2)$ — (ii)

$T(n-2) = (n-2) * T(n-3)$ — (iii)

(ii) in (i)

$T(n) = n * (n-1) * T(n-2)$ — (iv)

(iii) in (iv)

$T(n) = n * (n-1) * (n-2) * T(n-3)$

⋮ (n-1) steps.

$T(n) = n * (n-1) * (n-2) * \ldots * 1$

$T(n) = n!$

$T(n) = O(n!)$

③ $T(n) = \begin{cases} 1 & \\ 2T(\frac{n}{2}) & \end{cases}$

$T(n) = 2T($

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

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

(ii) in (i)

$T(n) = 2[2T($

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

(iii) in (

$T(n) = 2^2[2$

$T(n) = 2^3 T$

⋮ k

$T(n) = 2^k T$

Let

Now,

$T(n) = n*(n-1)*(n-2)*(n-3) \cdots T(n-(n-1))$

$T(n) = n*(n-1)*(n-2)*(n-3) \cdots T(n-n+1)$, $T(1)=1$

$T(n) = n*(n-1)*(n-2)*(n-3) \cdots *3*2*1$

$T(n) = n * n(1-\frac{1}{n}) * n(1-\frac{2}{n}) * n(1-\frac{3}{n}) \cdots * n(\frac{3}{n}) * n(\frac{2}{n}) * \frac{1}{n})$

$\boxed{T(n) = O(n^n)}$

(3) $T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(\frac{n}{2})+n & \text{otherwise} \end{cases}$

$T(n) = 2T(\frac{n}{2}) + n$ — (i)

$T(n/2) = 2T(n/4) + \frac{n}{2}$ — (ii)

$T(n/4) = 2T(n/8) + n/4$ — (iii)

(ii) in (i)

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

$T(n) = 2^2 T(n/2^2) + 2n$ — (iv)

(iii) in (iv)

$T(n) = 2^2\left[2T(n/8) + \frac{n}{4}\right] + 2n$

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

⋮ k times

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

$T(1)$

Let $n/2^k = 1 \Rightarrow n = 2^k$

$\Rightarrow \log n = k \log 2$

$\Rightarrow k = \log n.$

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

Now, $T(n) = n \cdot 1 + n \log n$

$\boxed{T(n) = O(n \log n)}$

$T(n) = 4T(n/4) + \frac{n}{2} + n$ — (iv)

(iii) in (iv)

$T(n) = 8T(n/8) + \frac{n}{4} + \frac{n}{2} + n$

$T(n) = 8T(n/8) + \frac{7n}{4}$

70
j=7
70
n
n/2
n/4
n/8

(4) $T(n) = \begin{cases} 1 & \text{, if } n = 1 \\ T(n-1) + \log n & \text{, if } n > 1 \end{cases}$

$T(n) = T(n-1) + \log n$ — (i)

$T(n-1) = T(n-2) + \log(n-1)$ — (ii)

$T(n-2) = T(n-3) + \log(n-2)$ — (iii)

(ii) in (i)

$T(n) = T(n-2) + \log(n-1) + \log n$ — (iv)

(iii) in (iv)

$T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log n$

$\vdots$ k times

$T(n) = \underbrace{T(n-k)}_{T(1)} + \log(n-(k-1)) + \log(n-(k-2)) + \dots + \log(n)$

Let $n-k = 1$

$\Rightarrow n = k$

$T(n) = T(n-n) + \log(n-(n-1)) + \log(n-(n-2)) + \dots + \log(n)$

$T(n) = T(0) + \log(1) + \log(2) + \dots + \log(n)$

$T(n) = 1 + \log(1 \cdot 2 \cdot 3 \dots n)$

$T(n) = 1 + \log(n!)$

$T(n) = 1 + \log(n^n)$

$\boxed{T(n) = O(n \log n)}$

$\log(n!)$

$= n \times (n-1) \times (n-2) \times (n-3) \dots$

In worst case.

$= n \times n \times n \times \dots$

Masters Theorem (Specific probl. ko hi solve krta hai)

→ analyzing the time complexity of the divide and conquer algorithm.

→ Standard form:- $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ $\{a \geq 1, b > 1\}$

where,

$T(n)$ → time complexity of algo.

a → no. of subproblems in recursion

b → factor by which the probl. size is reduced in each recursive call

$f(n)$ → time complexity of work done outside of recursive calls.

→ $T(n) = aT(n/b) + f(n)$
$a \geq 1, b > 1$

→ solution is :- $T(n) = n^{\log_b a}[U(n)]$
$U(n)$ depends on $h(n)$

→ $h(n) = \frac{f(n)}{n^{\log_b a}}$

→ Relation b/w $h(n)$ and $U(n)$ is :-

if $h(n)$	$U(n)$
$n^r, r>0$	$O(n^r)$
$n^r, r<0$	$O(1)$
$(\log n)^i, i \geq 0$	$\frac{(\log_2 n)^{i+1}}{i+1}$

① $T(n) = 8T(n/2) + n^2$
$a=8, b=2, f(n) = n^2$
soln :- $T(n) = n^{\log_b a}[U(n)]$
$= n^{\log_2 8}[U(n)]$
$= n^3[U(n)]$

$h(n) = \frac{f(n)}{n^3} = \frac{n^2}{n^3} = \frac{1}{n} = (n^{-1}) \rightarrow n^r, r<0 \rightarrow$ case (iii)
so, $U(n) = O(1)$

$T(n) = n^3 \cdot 1$
$\boxed{T(n) = O(n^3)}$

② $T(n) = T(n/2) + c$
$a=1, b=2, f(n) = c$
soln $T(n) = n^{\log_b a}[U(n)]$
$= n^{\log_2 1}[U(n)]$
$= n^0[U(n)]$
$= U(n)$

$h(n) = \frac{f(n)}{n^{\log_b a}} = \frac{c}{1} = c = (\log_2 n)^0 \cdot c$

so, $U(n) = \frac{(\log_2 n)^{0+1}}{0+1} \cdot c = (\log_2 n) \cdot c$
$= O(\log_2 n)$

so, $T(n) = O(\log_2 n)$

Master Theorem → Decreasing
→ Dividing

Decreasing

$T(n) = aT(n-b) + f(n)$

Case 1:- If $a=1$, then, $O(n^k)$ or $O(n * f(n))$

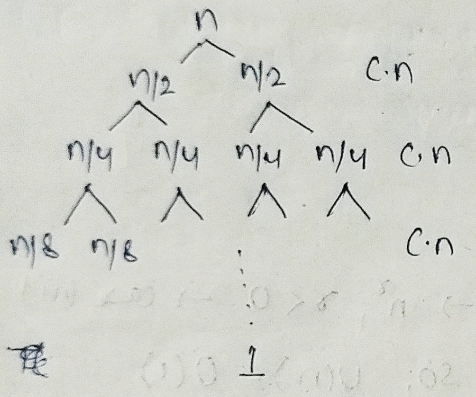
If $a>1$, then, $O(n^k$

Recursive Tree Method

→ Analyze time complexity of recursive algorithm.

↳ Visualizing the recursion as a tree, where each node in the tree represents a subprobl. and the edges represent the recursive calls b/w subprbl.

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



At each level, 2^k nodes

each node has $n/2^k$ cost

height of tree $= \log_2(n)$

∴ Total cost

$C_k = 2^k \cdot \frac{n}{2^k} = n.$

Total cost = sum

$T(n) = n + n + \ldots + n(\log_2 n)$ times

$T(n) = n(\log_2 n)$

$\boxed{T(n) = O(n \log n)}$

Time complexities:-

1) Binary search - $\log_2 n$
2) Sequential search - $O(n)$
3) Quick sort - $O(n \log n)$
4) Merge sort - $O(n \log n)$
5) Insertion sort - $O(n^2)$
6) Bubble sort - $O(n^2)$
7) Heap sort - $O(n \log n)$
8) Selection sort - $O(n^2)$

Performance measurement of algorithm

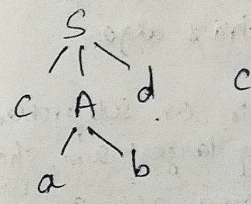
An algo. is said to be efficient and fast if it takes less time to execute and consume less memory space. Analyzing performance → Time complexity
→ Space complexity.

Brute force:- St. forw... cally tries all pos... used when problem... to try every possi... inefficient for lar...

Whenever a non-t... 1st alt. & compa... the 2nd alt. a... go with the 3r...

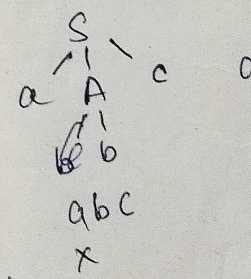
* If matching occ... successful, oth...

eg. S → cAd
A → ab | a
W_1 = cad



cabd
X

eg. S → aAc | aB
A → b | c
B → cd | dd
W = addc.



abc
X

Brute force

If grammer
↳ w

REDMI 9 — POWER
SANJANA MATHUR

2024.5.23 10:47

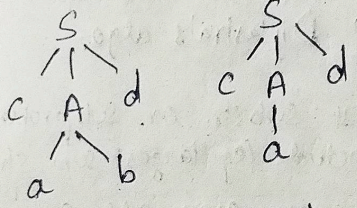
MODULE-2

Brute force:- St. forward approach to problem-solving that systematically tries all possible soln to find the best one.
used when problem size is small enough that it's feasible to try every possible solution.
inefficient for larger problems sizes. due to exhaustive nature.

Whenever a non-terminal is expanding 1st time, then go with 1st alt. & compare with i/p string. if doesn't matches, go for the 2nd alt. and compare with i/p string if doesn't matches go with the 3rd alt. & continue with each & every alt.

* If matching occurs for atleast 1 alt., then parsing successful, otherwise parsing fail.

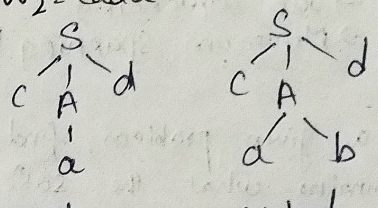
eg. $S \to cAd$
$A \to ab/a$
$W_1 = cad$



cabd
X

cad
(valid)

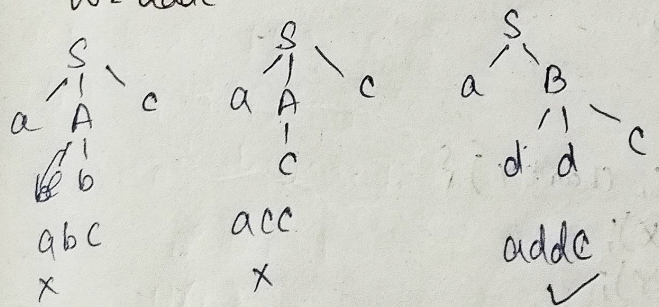
$W_2 = cada$ → invalid



cad
X

cabd
X

eg. $S \to aAc/aB$
$A \to b/c$
$B \to ccd/ddc$
$W = addc$.



abc
X

acc
X

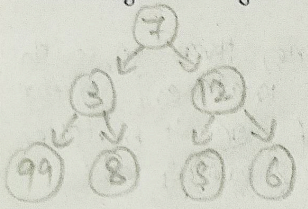
addc
✓

Brute force requires lots of back tracking $O(2^n)$
↓
very costly
reduces performance of parser
debugging is very difficult.

If grammer is non-deterministic
↳ we use brute force technique.

Greedy Approach.

Greedy Algorithm is a problem 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.



Greedy doesn't always work.
In this case, we choose 12 1st and the we wind at 6.
But, if we had choosen 3 then we had got 99.
So, Greedy fails.

→ follows local optimal choice at each stage with intent of finding global optimum.
→ Min. cost and Max. profit. Min. Risk.

Applications → Knapsack problem
→ Job sequencing
→ Minimum Spanning tree
→ Optimal merge pattern
→ Huffman coding
→ Dijkstra's algo.

Steps:-

Step 1:- In a given problem, find the best substr. or subproblem.

Step 2: Determine what the soln will include (eg. largest sum, shortest path)

Step 3:- Create an iterative process for going over all subpr. and creating an optimum soln.

Limitation - Greedy algo. makes judgements based on the info. at each iteration without considering the broader prob, hence it doesn't produce the best ans. for every probl.

Optimization problems (Dijkstra's Algo.) with -ve graph edges can't be solved using a greedy algorithm.

LCS Using Greedy Algo.

```
#include <stdio.h>
#include <string.h>
void lcs_greedy(char *x, char *y){
    int m = strlen(x);
    int n = strlen(y);
    int i = 0, j = 0;
    while(i<m && j<n){
        if(x[i]==y[j]){
            printf("%c", x[i]);
            i++;
            j++;
        }
```

```
else if(m>n
    i++;
}
else{
    j++;
}
}
int main(
    char
    char
    prin
    lcs_
    prin
    retu
}
```

Dynamic Appro

Dynamic Programm
to solve comple
simpler subpro
and storing th
leading to mo
Problems.

How it work

Ex:- Fibonac

To fi
simply
Thus
larger

```
        else if (m>n){
            i++;
        }
        else{
            j++;
        }
    }
}
int main(){
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";
    printf("LCS of %s and %s is: ", X, Y);
    lcs_greedy(X, Y);
    printf("\n");
    return 0;
}
```

Dynamic Approach

Dynamic Programming (DP) → method used in maths & CSE to solve complex problems by breaking them down into simpler subproblems. By solving each sub-probl. only once and storing the results, it avoids redundant computation, leading to more efficient sol? for a wide range of problems.

How it works:- ① Identify subproblems → Divide main probl. into smaller independent subprobl.

② Store Solutions → Solve each subpr. & store the sol? in a table or array.

③ Build up solutions → use the stored sol? to build up the sol?. to main probl.

④ Avoid Redundancy → By storing sol?, DP ensures that each subprob. is solved only once, reducing computation time.

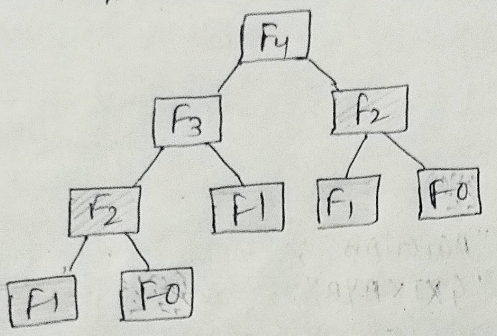
Ex:- Fibonacci series

To find it using brute force approach, you would simply add the (n-1) th & (n-2) th fibonacci no. This would work but it would be inefficient for larger values of n.

So, dynamic programming approach :-

N^{th} term of Fibonacci series.

Here, F_n denotes N^{th} term of Fibonacci



Overlapping Subproblems.

- Subproblems :- F(0), F(1), F(2), F(3), ...
- Store solutions :- Create a table to store the values of F(n) as they are calculated.
- Build up solutions :- For F(n), look up F(n-1) and F(n-2) in the table & add them.
- Avoid Redundancy :- The table ensures that each subproblem (eg. F(2)) is solved only once.)

Approaches of DP : ⟷ Top-down approach
Bottom-up Approach.

Top-Down Approach (Memoization) :-

- Start with final soln & recursively breaks it down into smaller subproblems.
- Stores the soln. to subproblems in a table to avoid redundant calculations.
- Suitable when no. of subprob. is large & many of them are used.

Bottom-Up Approach (Tabulation) :-

- Starts with smallest subproblems & gradually builds up final soln.
- Fills a table with soln. to subpr. in bottom-up manner.
- Suitable when no. of subprob. is small & the optimal soln. can be directly computed from soln. to smaller subpr.

Common Algo. that use

- Longest common sub
  Finds the longest com
- Shortest Path in a
  nodes in a Graph.
- Knapsack problem
  that can be plac
- Matrix Chain Multipl
  multiplication to mi
- Fibonacci Sequence:

Applications of DP

→ Optimization → Knaps
Subar

→ Computer Science →
→ Operations Research

LCS :-

```
#include <stdio.h>
#include<string.h>

int max(int a, int b
    return (a > b)
}
int lcs(char *x,
    int L[m+1][
    int i, j;
    for (i=0; i<
        for (j=
            if (

            els

            els

        }
    }
    retu
}
```

Common Algo. that use Dynamic Programming:-

- Longest common subsequence (LCS):- Finds the longest common subsequence b/w 2 strings.
- Shortest Path in a Graph:- Find shortest path b/w 2 nodes in a Graph.
- Knapsack problems:- Determine max. value of items that can be placed in a knapsack with given capacity.
- Matrix Chain Multiplication:- Optimizes the order of matrix multiplication to minimize the no. of operations.
- Fibonacci sequence:- Calculates the nth Fibonacci number.

Applications of DP

→ Optimization → Knapsack probl., shortest path prbl., max. subarray probl.

→ Computer Science → LCS, edit distance, string matching

→ Operations Research → Inventory management, Scheduling, resource allocation.

LCS:-

```
#include <stdio.h>
#include<string.h>

int max(int a, int b){
    return (a>b)? a: b;
}
int lcs(char *X, char *Y, int m, int n){
    int L[m+1][n+1];
    int i, j;
    for(i=0; i<=m; i++){
        for(j=0; j<=n; j++){
            if(i==0 || j==0)
                L[i][j]=0;
            else if(X[i-1]==Y[j-1])
                L[i][j]=L[i-1][j-1]+1;
            else
                L[i][j]=max(L[i-1][j], L[i][j-1]);
        }
    }
    return L[m][n];
```

```
int main( ){
    char X[ ] = "AGGTAB" ;
    char Y[ ] = "GXTXAYB";
    int  m = strlen(X);
    int  n = Strlen(Y);
    printf("Length of LCS is %d \n", lcs(X, Y, m, n));
    return 0;
}
```

Branch and Bound Algorithm

is a State space search algo., which is used for optimisation problems. This involves partitioning a probl. into sub-problems (branching) & solving these subprobl. to optimal level. It uses bounds to eliminate the need to consider suboptimal sol". (bounding).

Principles → Branching
→ Bounding
→ Selection.

- Branching → involves dividing a problem into smaller sub-problems in a systematic way.
- Bounding → This entails ~~into~~ calculating bounds on the optimal sol". to discard any subproblem that can't give a better sol". than the current best.
- Selection → involves choosing one subprobl. at a time for further exploration.

Pseudocode

function branch_and_bound:
1. Start with an empty pool of subproblems;
2. Add the original problem to the pool;
3. While the pool is not empty;
4. Select ~~the~~ & remove ~~the~~ a subpr. from pool
5. If subproblem has an optimal sol". & it is better than best known sol. update the best known sol.
6. If sub.. can't be pruned & it has branches - Variables, create new subpr. by branching & add them to the pool
7. return the best known sol".

TSP cost matrix

0	
10	
15	
20	

Back Tracking
are like problem
options to find
paths and if
can may find
resulting diff. pr

Example → find
Input:- A ma
0 → re
1 → re

Algorithm:-
1. Start at
2. For each
moving
3. If moving
path take
4. If movi
backtrack
5. Repeat
all po

Applicatio

How it
(i) choose
(ii) Expl
(iii) Back

TSP cost matrix :-

0	10	15	20
10	0	35	25
15	35	0	30
20	25	30	0

Backtracking Algorithm

are like problem-solving strategies that help explore diff. options to find the best soln. They work by trying out diff. paths and if one doesn't work, they backtrack & try another until they find the right one. It's like solving puzzles by testing diff. pieces until they fit together perfectly.

Example → Finding the shortest path through a maze.

Input:- A maze represented as a 2D array where,
0 → represent open space
1 → represents a wall.

Algorithm :-

1. Start at the staring point.
2. For each of the 4 possible direcn. (up, down, L, R), try moving in that direction.
3. If moving in that direcn. ~~fata~~ leads to ending point, return path taken.
4. If moving in that direcn. doesn't lead to ending point, backtrack to previous posn. & try a diff. direcn.
5. Repeat steps 2-4 until the ending point is reached or all possible paths have been explored.

Applications → ① Solving puzzles (Sudoku)
② Finding shortest path through a maze.
③ Scheduling problems.
④ Resource allocation problems

How it works

(i) Choose → Make a choice from the available options for the next part of the soln.

(ii) Explore → Recursively explore all the possible choices from the current state.

(iii) Backtrack → If a soln. can't be found using the current choice, undo the choice & go back to the previous state to explore other options.

Heuristics

Problem solving techniques or rules of thumb that simply complex decision-making processes, often by using practical & expt. knowledge rather than formal algorithms.

Characteristics:-

① Simplicity → simple & easy-to-understand rule that guide decision-making.

② Efficiency → designed to provide quick sol^n or decisions without exhaustive analysis.

③ Experience-based → rely on past experiences, knowledge or intuition rather than explicit calculations or data analysis.

④ Adaptability → can be adapted/modified based on specific context or problem at hand.

⑤ Risk → may involve risks or biases, as they don't guarantee optimal solutions.

⑥ Domain Specific → tailored to specific domains or types of problems.

Applications :-

① Cognitive psychology → understand human decision-making processes.

② Artificial intelligence → for problem-solving, search algo., decision-making processes

③ Design & Engineering → designers & engineers often rely on rules of thumb to generate & evaluate sol^n efficiently.

④ Medical decision making → diagnosing illnesses, formulating treatment plans.

⑤ Education → instructional design to facilitate learning processes.

Graph and Tree

Traversal Algo. are

process, all the n

Tree Traversal Alg

Depth-First Traver

Pre-order In-order

Pre-order :- algo

the left subtree

In-order :- algo.

root node and

used for-bina

Post order :- algo.

subtree, and fi

useful for-

Breadth-First

a level before

to keep track

Graph Traversa

DFS → stack : is a

possible along

used to find

maze problems

BFS → queue : explores

before moving

commonly used

Dijkstra's al

Dijkstra's algo

is often used

weighted grap

order of ↑ d

REDMI 9 —P-OWER

SANJANA MATHUR

2024.5.23 10:48

Graph and Tree Algorithms: Traversal Algorithms

Traversal Algo. are fundamental techniques used to visit and process all the nodes in a graph or a tree data str.

Tree Traversal Algorithms:

Depth-First Traversal:- DFT explores as far as possible along each branch before backtracking.

Pre-order ↙ In-order ↓ Post-order ↘

Pre-order:- algo. visits root node, then recursively traverses the left subtree & finally the right subtree.

In-order:- algo. recursively traverses the left subtree, then visits the root node and finally traverses the right subtree.
used for - binary search trees to get nodes in sorted order.

Post-order:- algo. recursively traverses left subtree, then the right subtree, and finally visits the root node.
useful for - deleting nodes in a tree.

Breadth-First Traversal:- (BFT) :. visits all the nodes of a level before going to the next level. It uses queue ds to keep track of nodes to be visited.

Graph Traversal Algo.:-

DFS (stack): is a graph traversal algo. that explores as far as possible along each branch before backtracking. It can be used to find connected components, detect cycles, solve maze problems.

BFS (queue): explores all the neighbour nodes at the present depth before moving on to the nodes at the next depth level.
commonly used for finding shortest paths & in algo. like Dijkstra's algo.

Dijkstra's algo.:- not strictly a traversal algo., Dijkstra's algo. is often used for finding the shortest paths btw nodes in a weighted graph. uses priority queue to explore nodes in order of ↑ distance from source node.

Module - 3

DFS

→ explores the graph by going as deep as possible along each branch before backtracking.

→ it explores one branch of the graph as deeply as possible before moving on to explore other branches.

→ DFS uses stack DS to keep track of the nodes to be visited.

→ used for topological sorting, cycle detection, path finding.

Pseudocode

```
DFS(Graph, start_node):
    Create a stack S
    push start_node onto S
    mark start_node as visited
    while S is not empty:
        current_node = pop from S
        visit current_node
        for each neighbour of current_node:
            if neighbour is not visited:
                mark neighbour as visited
                push neighbour onto S.
```

BFS

→ Start with a queue ds & enqueue the starting node.

→ while the queue is not empty:

• Dequeue a node from the queue.

• visit the dequeued node.

• Enqueue all of its neighbours (not yet visited) into queue.

→ Repeat step 2 until queue is empty.

Pseudocode

```
BFS(Graph, start_node)
    create a queue Q
    enqueue start_node into Q
    mark start_node as visited
    while Q is not empty:
        current_node = dequeue from Q
        visit current_node
        for each neighbour of current_node:
            if neighbour is not visited:
                mark neighbour as visited
                enqueue neighbour into Q.
```

Shortest Path Algorithm

Refers to a set of al
tes nodes in a graph
The most well-kno
Dijkstra's algo., whic
a single source no
graph.

Dijkstra's algo. g
as long as the
If -ve edge, w
used.

other variants o

① A* algo. :- co
search, used
graph.

② Floyd-Warshal
graph including
for finding Si

③ Bellman-Ford
node to all o
-ve edge weigh

Dijkstra's algo.

1> Initialisat

2> Selecting

3> Exploring

4> Marking

5> Termin

6> Iteration

Shortest Path Algorithm

Refers to a set of algo. used to find the shortest path b/w two nodes in a graph.

The most well-known & widely used shortest path algo. is Dijkstra's algo., which efficiently finds the shortest path from a single source node to all other nodes in a weighted graph.

Dijkstra's algo. guarantees finding the shortest path as long as the graph doesn't contain -ve edge weights. If -ve edge weights are there, Bellman-Ford algo. can be used.

Other variants of shortest path:-

1. A* algo.:- combination of Dijkstra's algo. and heuristic search, used for finding shortest path b/w 2 nodes in a graph.
2. Floyd-warshall algo. - b/w all pairs of nodes in a weighted graph including -ve edge weights but it's less efficient for finding single-source shortest paths.
3. Bellman-Ford algo. - used to find ___ from a single source node to all other nodes in a graph, even if it contains -ve edge weights.

Dijkstra's algo.

1. Initialization - Set initial node's distance to 0 and all other node's dist. to ∞. Mark all nodes as unvisited.
2. Selecting the node - Choose unvisited node with smallest tentative dist. For initial node, this value is 0.
3. Exploring neighbours - For the selected node, consider all its neighbours & calculate their tentative dist. through the current node.
4. Marking as visited - Once all the neighbours have been considered, mark the current node as visited.
5. Termination - If the destination node has been visited or if the smallest tentative dist. among the unvisited nodes is ∞, the algo terminates.
6. Iteration - Repeat 2 through 5 until either destination node has been visited or if smallest tentative dist. is ∞.

Transitive Closure

- It's a concept used in graph theory that deals with determining the reachability b/w pairs of vertices in a directed graph.
It helps answer the question: "Can I reach vertex B starting from vertex A by following directed edges?"

Given a directed graph $G = (V, E)$ with vertices V and edges E, the T.C. of G is another directed graph $G' = (V, E')$ such that there is an edge from vertex u to vertex v in G' if and only if there is a directed path from u to v in G.

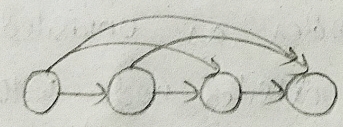
Ways to compute T.C. of a graph :-

1. **Floyd-Warshall Algo.** → used to find shortest path b/w all pairs of vertices in a weighted graph, Time complexity = $O(v^3)$
2. **Warshall's Algo.** → Specifically designed to compute t.c. of a graph. Time complexity $O(V^3)$
3. **Depth-First Search (DFS)** → By performing a DFS from each vertex & marking all vertices reachable from it, we can construct t.c. of the graph. $T(n) = O(V * (V+E))$ V → vertices, E → Edges.

Transitive closure constructs the output graph from input graph.

Input

Output



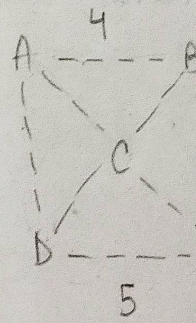
Main

Minimum Spa
(MST) of a con
a tree (a graph
together with

Algo. for findin
Kruskal's Algo

1. Sort edges
2. Initialize Gr
3. Iterate throu
4. Add Edges
5. Union-Find
6. Continue un
7. Output M

ex:-
Undirected g



Minimum Spanning Tree :-

(MST) of a connected, undirected graph is a subgraph that is a tree (a graph with no cycles) and connects all the vertices together with the min. possible total edge weight.

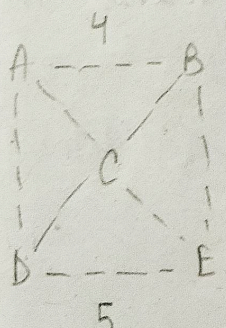
Algo. for finding Minimum Spanning Tree :- Prim's algo. O(ElogV)

Kruskal's Algorithm :⇒ O(ElogE)

1. Sort edges by weight :- Sort all the edges in non-decreasing order of their weights.
2. Initialize Empty MST :- Create an empty min. spanning tree, initially containing no edges.
3. Iterate through sorted edges :- Starting from the smallest edge, iterate through the sorted list of edges.
4. Add Edges if No cycle :- For each edge, if adding it to the MST doesn't create a cycle, add it to the MST.
5. Union-Find (Disjoint set Data Str.) :- Keeps ~~check~~ track of disjoint sets & efficiently determines whether adding an edge would create a cycle in the MST.
6. Continue until MST is formed :- Repeat 4 & 5 until MST contains V-1 edges.
7. Output MST :- Resulting tree is Min. Spanning Tree.

Ex :-

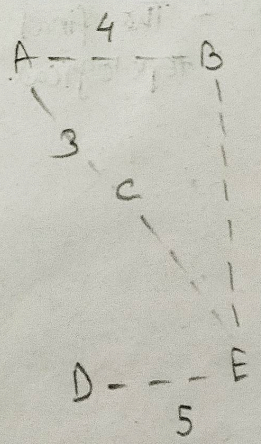
Undirected graph



Sorted list of edges with their weights.

(A,B) : 4
(A,D) : 3
(A,C) : 6
(B,C) : 2
(B,E) : 5
(C,D) : 1
(C,E) : 7
(D,E) : 5

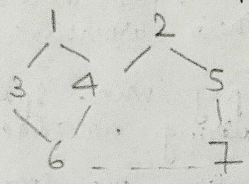
Resulting MST :-



Topological Sorting

It is a linear ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge $u \to v$, vertex u comes before vertex v in the ordering.
It is used primarily in scheduling tasks where some tasks must be completed before others can begin.

Ex:- Consider a DAG representing dependencies btw tasks.



Task 1 must be completed before tasks 3 & 4 can start.

Steps :-

1> Identify Unvisited Nodes - Begin by marking all vertices as unvisited.

2> DFS (Depth-first Search) - Perform DFS on the graph starting from any unvisited node.

3> Visit and Mark Visited - When visiting a node during DFS, recursively visit its neighbors if they are unvisited.

4> Ordering :- After visiting all neighbors of a node, add it to the front of a list or stack.

5> Repeat for unvisited Nodes :- If there are any unvisited nodes remaining, repeat the DFS process from one of them.

6> Output - The final ordered list or stack represents the topological sorting of the graph.

Network Flow
are a class of
min. cut-) in a flo
where each edge
data, or any other
used in → trans
→ telec
→ com

Ford-Fulkerson
finds the ma
1> Initialization →
2> Augmenting Pa
3> Augmenting
adde
any
capa
4> Update Res
5> Repeat → Re
be
6> Output →
f

REDMI 9 —POWER
SANJANA MATHUR

6064.5.23 10:48

Network Flow Algorithm

are a class of algorithms used to find the max. flow (or min. cut) in a flow network, which is a directed graph where each edge has a capacity & represents a flow of goods, data, or any other entity.

used in → transportation
→ telecommunications
→ computer network.

Ford-Fulkerson algorithm :-

finds the max. flow in a flow network.

1) Initialization → Initialize the flow on each edge to 0.

2) Augmenting paths → It is a simple path from source to sink in which all edges have residual capacity > 0.

3) Augmenting flow → find the max. possible flow that can be added without violating the capacity constraints of any edge. This max. flow value is called bottleneck capacity of the augmenting path.

4) Update Residual capacities → Update the residual capacities of the edges & reverse edges along the augmenting path.

5) Repeat → Repeat steps 2-4 until no augmenting paths can be found in the residual graph.

6) Output → The max. flow is the total flow out of the source vertex, which is also eq. to the total flow into the sink vertex.

Module-4

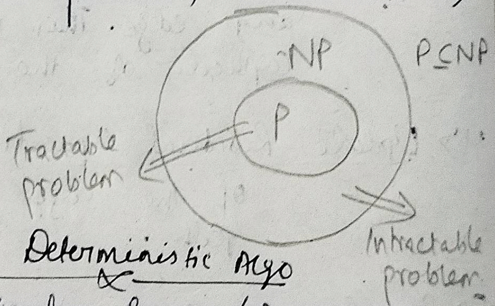
Algorithm

Polynomial time
- → Linear Search (O(n))
- → Primary Search (O(log n))
- → Insertion sort (O(n²))
- → Merge sort (O(n log n))

Non-polynomial time / Exponential time
- → 0/1 knapsack - 2^n
- → Traveling salesman - 2^n
- → Graph colouring - 2^n
- → Su-Do-Ku - 2^n
- → Scheduling - 2^n

P class Problem - A problem which can be solved in polynomial time is known as P-class problem (Sorting / Searching).

NP Class (Non deterministic & polynomial time) - A problem which can't be solved in polynomial time but can be verified in polynomial time (eg. 0/1 knapsack problem, Su-do-ku, TSP)



Deterministic Algo

The algo. which we generally use like linear search, 1° search, where we know that how each & every step works.

Non Deterministic Algo

```
Kundra Search (A, n, key)
{
   i = choice ( );
   if (i == key)
   {
      print (i)
      // success
   }
   print (0)
   // failure
}
```

A, key = 6

9	8	7	6	4	11
1	2	3	4	5	6

Computability of A...
is concerned ...
An algo. is said ...
Turning machine ...

computability cla...

1. P (Polynomial Ti...
   . Problems for whi... polynomial time ... Time taken by th... size of the input ...
   ex - Sorting algo.
2. NP (Nondetermin...
   . A problem is ... be verified in ...
   Ex - The Travelli...
3. NP-hard (Nond...
   . A problem is N... reduced to it in ...
   Ex:- Boolean Sa...
4. NP-complete (N...
   . A problem is ... NP-hard. NP-c... NP in terms ...
   ex - Subset Sum ...

Summary

P → represents e...

NP → represents ... verified

NP-hard → repres... proble...

NP-complete → ...

Relationship ... the complexity ...

Computability of Algo.

It is concerned with what can & can't be computed. An algo. is said to be computable if there exists a Turning machine that can solve the probl. it addresses.

computability classes.

① P (Polynomial Time):-

Problems for which there exists an algo. that runs in polynomial time in terms of the size of the input. Time taken by the algo. is bounded by poly. func" of size of the input.

ex- Sorting algo. like Merge sort & Quick sort.

② NP (Nondeterministic Polynomial Time):-

A problem is in NP if, given a soln., its correctness can be verified in polynomial time.

ex- The Travelling Salesman Problem.

③ NP-hard (Nondeterministic Polynomial-hard):-

A problem is NP-hard if every problem in NP can be reduced to it in polynomial time.

ex:- Boolean Satisfiability Problem (SAT)

④ NP-complete (Nondeterministic Polynomial-complete):-

A problem is NP-complete if it is both in NP and NP-hard. NP-complete problems are the hardest problems in NP in terms of computational complexity.

ex- Subset Sum Problem.

Summary

P → represents efficiently solvable problems

NP → represents problems whose solutions can be efficiently verified

NP-hard → represents problems atleast as hard as the hardest problems in NP.

NP-complete → represents hardest problems within NP.

Relationship btw these classes is fundamental to understanding the complexity of computational problems.

COOK'S Theorem

Also called COOK'S Levin theorem.

States that - any decision problem in NP, a class of decision problems for which a soln. can be verified in polynomial time, can be transformed into an equivalent problem known as SAT (Boolean Satisfiability Problem) in polynomial time.

It states that the Boolean Satisfiability problem is NP-complete.

Standard NP-Complete problems and Reduction techniques

(1) **Boolean Satisfiability Problem (SAT):**
Given a boolean formula, is there an assignment of truth values to the variables such that the formula evaluates to true?

Reduction technique: Many NP-complete problems can be reduced to SAT, such as 3SAT where each clause contains at most 3 literals.

(2) **Vertex cover:-**
Given an undirected graph & a +ve integer k, is there a set of at most k vertices such that each edge of the graph is incident to at least one vertex in the set?

Reduction technique: Reduction from SAT or 3SAT, where each variable corresponds to a vertex & each clause corresponds to an edge.

(3) **Subset Sum:-**
Given a set of integers & a target sum, is there a subset of the integers that adds up to the target sum?

Reduction Technique: Often reduced from Partition problem, where goal is to partition a set of no. into 2 subsets with eq. sums.

(4) **Travelling Salesman Problem (TSP):-**
Given a list of cities & the dist. b/w each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the original city?

Reduction technique: Reduction from Hamiltonian cycle, where the goal is to find a cycle that visits each vertex exactly once.

(5) **Knapsack Problem:-**
Given a set of items, ea
determine the total weight
in a collection so that
or eq. to a given limit
possible.

Reduction Technique: Often
where
integer
corresp

Modu

Approximation Algorithm

is an algo. designed
optimization problems
These problems are ofte

key points

(1) Approximation Ratio →
case ratio b/w the
the optimal soln.

(2) Polynomial time com
given in polynomial
large instances of NP-

(3) Performance Guarant
that the soln. they
optimal soln.

(4) Greedy Algorithm
selecting the locally

(5) Heuristic Approach
and find near-opt

(6) Trade-off b/w so
Approximation algo

③ Knapsack Problem:
Given a set of items, each with a weight & a value, determine the total weight no. of each item to include in a collection so that the total weight is less than or eq. to a given limit & the total value is as large as possible.

Reduction Technique: Often reduced from Subset Sum, where each item corresponds to an integer weight, and the knapsack capacity corresponds to the largest sum.

Module 5

Approximation Algorithm
is an algo designed to find near-optimal soln. to optimization problems in a reasonable amount of time. These problems are often NP-hard

Key points

① Approximation Ratio → This of an algo. is the worst-case ratio b/w the value of its soln. and the value of the optimal soln.

② Polynomial time complexity - Approx. Algo. are designed to run in polynomial time ensuring that they can handle large instances of NP-hard probl. in reasonable amnt. of time

③ Performance Guarantee - They provide perf. guar., ensuring that the soln they find is reasonably close to the optimal soln.

④ Greedy Algorithm - decisions are made iteratively by selecting the locally optimal choice at each step.

⑤ Heuristic Approach - to efficiently explore the soln space and find near-optimal soln.

⑥ Trade-off b/w soln. Quality and Efficiency - Approximation algo. trade off soln quality for efficiency.

Randomized algorithms

Randomized algorithm is an algorithm that incorporate random input, often using random no. or probability distributions, in its operation.

Advantages :-

1. Efficiency → more efficient in terms of time or space complexity compared to their deterministic counterparts.
2. Approximation → can provide approximate sol^n to complex problems efficiently.
3. Parallelism → making them ~~efficient~~ suitable for execution on parallel & distributed systems.
4. Security → Secure communication & other security applications

Example → Randomized Quick sort —
Avg. time complexity of $O(n \log n)$.