Data Structure and Algorithms

Complete Handwitten Notes

Unlock excluive PDF Notes! Join our Telegram Channel to dwonload them now.

CodeNoteBook

DATA STRUCTURE

- **What is Program**

  A set of instructions that can be executed to perform a specific task.

- **Data Structure** → the way of data being organized, managed & its storage format that enables efficient access and modification.

Program.
~~Data Structure~~ = Algorithm + data.

Data structure = A container stores data to perform the tasks like insertion, deletion & modification

Ex Array, Stack, Queue, Tree, Linked list, Heap, Hash Table, Priority Queue.

1) functions of Data Struct.

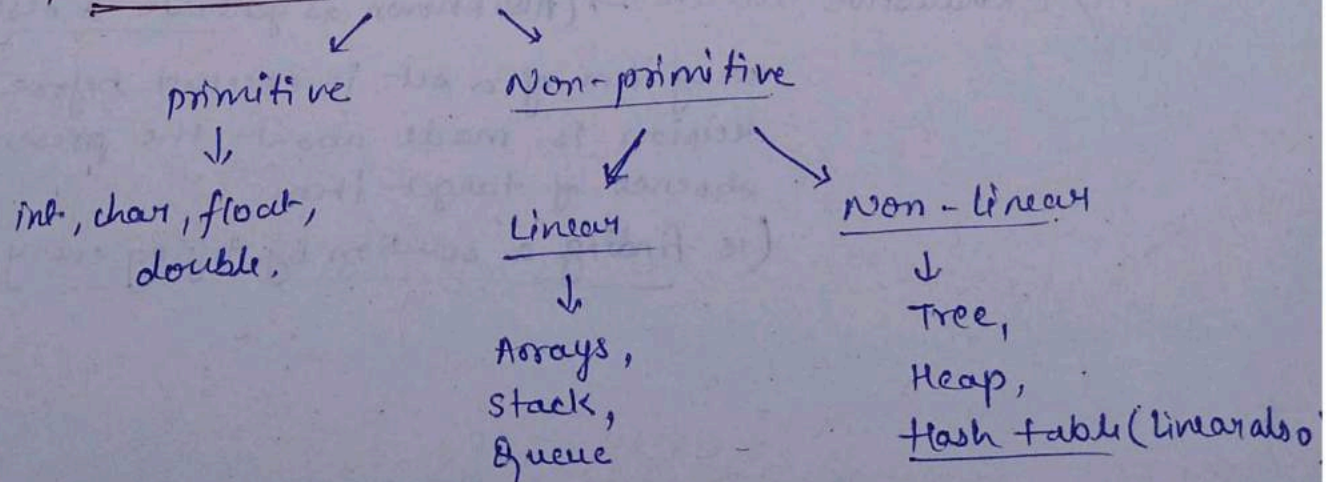
→ Add
- Index — indexing
- key — values
- Position — 1st, 2nd or last etc
- Priority - sorting like asc or desc.

⇒ Get

⇒ change

⇒ Delete

2) Common Data Structures (Examples)



* Linked list on the basis of storage is considered as non-linear however on the basis of access strategies a linked list is considered linear.

• **Algorithm** → set of finite instructions to accomplish a task.

Algorithm = Logic + Control

→ There are countless Algorithms
→ Algorithms are machine dependent.

⟨A⟩ Algorithm Strategies

1) Greedy → to complete the process in best possible way.

2) Divide and conquer → involves dividing the program into
Divide: some sub program.
Conquer: Sub program by calling recursively until sub program solved
Combine: Sub problem solved so that we will get find problem solution.
Eg Merge sort, binary search etc.

3) Dynamic programming → solving an optimization problem by breaking it down into simpler subproblems & utilizing the fact that the optimal solution to the overall problem depends upon the optimal soln to its subproblems.
Eg (Repeating same program with consideration of previous one like factorial program)

4) Exhaustive Search → (Also known as generate & test)
every item of a set is checked before a decision is made about the presence or absence of target item.
(ie finding a solution by trying every possibility)

• How to analyze an Algorithm

High performance requirements

1) Time → it must be time efficient ie faster
→ by analysing how much time it is taking
→ We get a time function on the basis of which we can analyze our different programs ie f(n). in terms of constant value (1 unit time per statement)

2) Space → how much memory space it consume (as the algo is going to be converted into program and run on the machine it requires space)

3) Network → how much data is going to be transferred.

4) Power consumption

5) CPU registers consumption → especially the algorithms for device drivers or system programs (Low Ram footprints are best)

* If f(n) & S(n) ie. Space complexity/function are same ie constant then order of Algo is 1 ie O(1)

S(n) → ~~represents~~ called no. of words ie variables having some space rather than bytes
bcz we don't know which data type they are.

Which data Structure or Algorithm is better?

→ Must Meet Requirements
→ should have high performance in terms of time & space complexity
→ Low RAM footprints (ie use less RAM as much as possible)
→ Easy to implement
* Encapsulated.

* fast execution of program depends on the logic of Algorithm
It should be simple and occupy less space memory so that it makes CPU busy during the whole process resulting in fast execution.

Basic Concepts

1. Overview : System Life cycle.
2. Algorithm Specification
3. Data Abstraction
4. Performance Analysis
5. Performance Measurement.

Overview : System Life Cycle

→ Good programmers regard large-scale computer programs as systems that contain many complex interacting parts.

→ As systems, these programs undergo a development proc called the system life cycle.

→ We consider this cycle as consisting of five phases.

(a) Requirements

(b) Analysis : bottom-up
Vs
top-down

(c) Design : data objects and operations

(d) Refinement and Coding

(e) Verification
- → Program proving
- → Testing
- → Debugging

2) Algorithm Specification

2.1 INTRODUCTION

→ An algorithm is a finite set of instructions that accomplish a particular task.

⇒ Criteria (characteristics of an Algorithm)

(a) Input: zero or more quantities that are externally supplied.

(b) Output: at least one quantity is produced.

(c) Definiteness: clear and unambiguous (not having dou meanings or wrong logics like x/0 infinite loops etc

(d) finiteness: terminate after a finite no. of steps

(e) effectiveness: instruction is basic enough to be carried out.

⇒ Representation

- A natural language like English / chinese
- A graphic, like flowcharts
- A computer language, like C.

⇒ Algorithms + Data structures = Programs

⇒ Recursive algorithms

- Beginning programmers view a function as sth t is invoked (called) by another function.
  - → It executes its code and then return contro to the calling function.
- This perspective ignores the fact that functio can call themselves (Direct recursion)
- They may call other functions that invoke calling function again (Indirect recursion)
  - → extremely powerful
  - → frequently allow us to express an otherwi complex process in very clear term.

• We should express a recursive algorithm when the problem itself is defined recursively.

3) Data Abstraction

⇒ Data type :- It is a collection of objects and a set of operations that acts on those objects.

Eg the data type int consists of objects {0, +1, -1, ... INT_MAX, INT_MIN} and the operations +, -, *, /, and %.

• Data types of C
  a) The basic data types : char, int, float & double
  b) The group data types : array and struct (used defined)
  c) The pointer data type
  d) The user-defined types.

⇒ Abstract Data Type (Eg pointer to a structure)

• An abstract data type (ADT) is a data type that is organized in such a way that the specification of the objects and the operations on the objects is separated from the representation of the objects and the implementation of the operations.

★ We know what it does, but not necessarily how it will do it.

• Specification Vs Implementation
  → An ADT is implementation independent
  → Operation specification
    • function name
    • the type of arguments
    • the type of the results

→ The functions of a data type can be classify into several categories :-

- creator / constructor
- transformers
- observers / reporters.

4) Performance Analysis

- Criteria
  - is it correct ?
  - is it readable ?
  - ⋮
- Performance Analysis (Machine independent)
  - Space complexity : storage requirement
  - time complexity : computing time
- Performance Measurement (Machine dependent)

⟨A⟩ SPACE COMPLEXITY

$$S(P) = C + S_p(I) \longrightarrow \text{Variable space Requirement } (S_p(I))$$

$\downarrow$

fixed Space Requirements (C)

- Fixed space Requirements (C)
  Independent of the characteristics of the inputs & o/ps
  - Instruction space
  - space for simple variables, fixed-size structured variable constants.
- Variable Space Requirements ($S_p(I)$)
  Depends on the instance characteristics I.
  - number, size, values of inputs and outputs associated with I.
  - Recursion stack space, formal parameters, local variables return address.

★ A program is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

Example

$abc = a+b+b*c+(a-b-c)/(a+b)+4.0$

- $abc = a+b+c$

★ Methods to compute the step count-

- Introduce variable count into programs
- Tabular method
  - → Determine the total no. of steps contributed by each statement- Step per execution × frequency.
  - → add up the contribution of all statements.

Iterative summing of a list of numbers.

```
float sum(float list[], int n)
{
  float tsum=0; count++; // for assignment- ✓
  int i;
  for(i=0; i<n; i++)
  { count++;          // for the loop
    tsum+=list[i];
    count++;          // for the assignment
  }                                          } 2n steps.
count++;            // last execution of for  ✓
count++;            // for return.            ✓
return tsum;
}
```

∴ Total 2n+3 steps

Tabular Method

Iterative function to sum a list of numbers Steps / execution

Statement	s/e	Frequency	Total steps (freq X s/e)
float sum (float x[], int n)	0	0	0
{	0	0	0
float sum=0;	1	1	1
int i;	0	0	0
for (i=0; i<n; i++)	1	n+1 (last condn)	(n+1)
sum+ = x[i];	1	n	n
return sum;	1	1	1
}	0	0	0
			2n+3

Step count table for Recursive summing function.

	s/e	frequency	Total Steps
float rsum (float list[], int n)	0	0	0
{ if (n)	1	n+1	n+1
return rsum (list, n-1) + list [n-1];	1	n	n
return list[0];	1	1	1
}	0	0	0
			2n+2

Space-time trade-off & Efficiency

→ Trade-off b/w memory use & runtime performance.

→ Space efficiency & time efficiency reach at two opposite ends

→ The more time efficiency you have, the less space efficiency you have and vice-versa.

Eg Merge Sort & Bubble sort algorithm.

Analysis Vs Design

Analysis: Predict the cost of an algorithm in terms of resources and performance.

Design: Design algorithms which minimize the cost.

Time Complexity

Real time:- To analyze the real time complexity of a program we need to determine two numbers for each statement in it:

(1) Amount of time a single statement will take (s/e)

(2) No. of times it is executed (frequency).

→ Product of these two, will be the total time taken by the statement.

* 1st number depends upon the machine and compiler used, hence the real time complexity is machine dependent.

· Machine Model : Generic RAM

→ Executes operations sequentially

→ Set of primitive operations:

- Arithmetic
- Logical
- Comparisons,
- function calls.

→ Simplifying assumption : all operations cost 1 unit

- Eliminates dependence on the speed of our computer, otherwise impossible to verify and to compare

① Linear Loop.

1. $i = 1$
2. loop ($i <= n$)
   - 2.1 Application code
   - 2.2 $i = i + 2$

For this code time is proportional to n

$f(n) = n$

② Logarithmic Loops

Multiply loops

1. $i = 1$
2. loop ($i < n$)
   - 2.1 Application code
   - 2.2 $i = i * 2$

$f(n) = [\log n]$

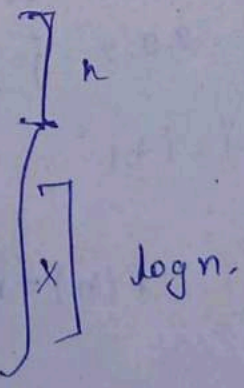
Divide Loops.

1. $i = n$
2. loop ($i >= 1$)
   - 2.1 Application code
   - 2.2 $i = i / 2$

$f(n) = [\log n]$

③ Nested loop - linear logarithmic

1. $i = 1$
2. loop ($i <= n$)
   - 2.1 $j = 1$
   - 2.2 loop ($j <= n$)
     - 2.2.1 application code
     - 2.2.2 $j = j * 2$
   - 2.3 $i = i + 1$



$$f(n) = [n \log n]$$

④ Dependent Quadratic

(bcz j dependent i)

1. i=1
2. loop(i<=n)

    2.1 j=1

    2.2 loop(j<=i)

        2.2.1 Application code

        2.2.2 j=j+1

    2.3 i=i+1

no. of iterations in this body of inner loop is $1+2+3+\cdots+n$ ie $\frac{n(n+1)}{2}$

∴ On an average $\frac{(n+1)}{2}$

∴ Total no. of iterations $= n\frac{(n+1)}{2}$

⑤ Quadratic

(i & j both dependent on n)

1. i=1
2. loop(i<=n)

    2.1 j=1

    2.2 loop(j<=n)

        2.2.1 Application code

        2.2.2 j=j+1

    2.3 i=i+1

$F(n) = n^2$

Frequency Count

→ To make analysis machine independent it is assumed that every instruction takes the same constant amount of time for execution.

→ Hence the determination of time complexity of a program is the matter of summing the frequency counts of all the statements.

Two Algorithms on two systems.

Algo A_1 $50(n \log n)$

Algo A_2 $2(n^2)$

Having performance

If A_2 → Super computer → 10^8 ins/sec

A_1 → P.C → 10^6 ins/sec.

for $n = 10^6$,

Super computer takes time

$$= \frac{2 \cdot (10^6)^2}{10^8} = 20,000 \text{ sec}.$$

Time taken by PC

$$= \frac{50(10^6 \log 10^6)}{10^6} = 1,000 \text{ sec}.$$

* Thus by using a fast algorithm, the personal computer gives results 20 times faster, than the result given by super computer using a slow algorithm.

Complexity

(Questions arises)

1. How fast can we solve a program?
   (depends no. of processors used)
2. There may be many algorithms for a given problem.
   Which ~~pro~~ algorithm to use?
3. What are the classical algorithm design techniques?
4. Are there problems inherently difficult to solve?
   ↳ Design And Analysis of Problems have some questions which are difficult to answer.
5. How do we express the complexity of algorithm?
   ↳ (Time and space, Complexity lower bounds
   ↙ for problems. ↘ complexity classes P, NP, etc)
   no. of instructions. amount of memory.

Lower bound → the min no. of steps for program.
like in ~~in~~ factorial it is 2 (ie 2 steps $n \times (n-1)$)
→ and upper bound is $12 \times 2 \rightarrow 24$
(bcz above 12 we can't access factorials in case of int)

* For fast execution of program, memory fetching should be reduced
(Eg for swapping 2 no.s we would go for the algo without using 3rd variable)

n depends on
- type of loops used
- input size.

✱ Asymptotic Complexity (Using mathematical notation)

- Running time of an algorithm as a function of input size n for large n.
- Expressed using only the highest-order term in the expression for the exact running time.
  - Instead of exact running time, say $\Theta(n^2)$
- Describes behavior of function in the limit.
- Written using Asymptotic Notation.

Asymptotic Notation :- O, Θ, Ω

- Defined for functions over the natural numbers.
  Eg $f(n) = \Theta(n^2)$.
  - Describes how $f(n)$ grows in comparison to n^2.
- Define a set of functions; in practice used to compare two function sizes.
- The notations describe different rate-of-growth relations between the defining function & the defined set of functions.

Categories of algorithm efficiency

Efficiency ~~constant~~	Big O
Constant	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
Linear logarithmic	$O(n \log n)$
Quadratic	$O(n^2)$
Polynomial	$O(n^x)$
Exponential	$O(c^n)$
factorial	$O(n!)$

Three Cases → Best Case
→ Worst Case
→ Average Case

$f(n)$ → function representing rate of growth of algorithm.
(A defining function)

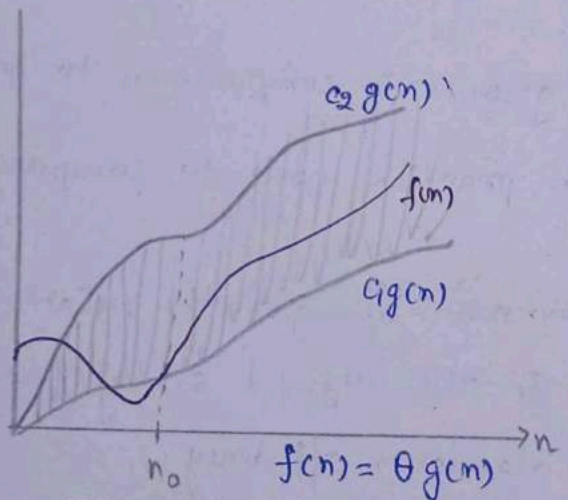
Θ → ~~Average~~ case
O → Upper bound
Ω → Lower bound. (for best case)

$g(n)$ → asymptotic ~~tight~~ bound for $f(n)$. $f(n)$ can't cross this limit.

① Θ notation → for avg. case

for function $g(n)$, $\Theta(g(n))$ is given by

$\Theta(g(n)) = \{ f(n) : \exists$ +ve constants c_1, c_2 and n_0.
such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$, $\forall\ n \ge n_0 \}$



Intuitively :- Set of all functions that have the same rate of growth as $g(n)$

n_0 = least no. that can be given as input.
Eg → factorial exists only for the nos (including 0)

c_1 & c_2 = constants

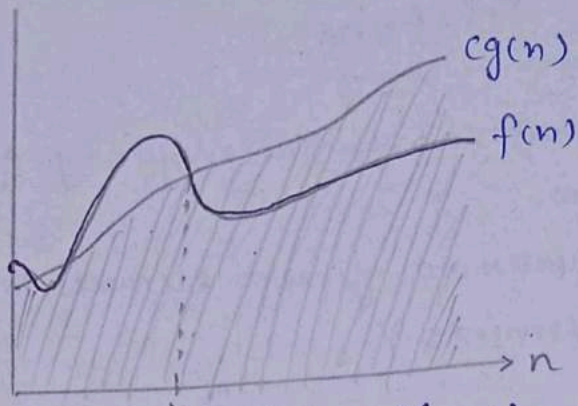
② O - notation → for worst case

for function $g(n)$, $O(g(n))$ is given by

$O(g(n)) = \{ f(n) :$ ∃ +ve constants c and n_0 such that
$0 \le f(n) \le g(n)$, $\forall\ n \ge n_0 \}$

$g(n)$ is an asymptotically ~~tight~~ upper bound for $f(n)$.

Intuitively :- Set of all functions whose rate of growth is the same as or lower than that of $g(n)$.



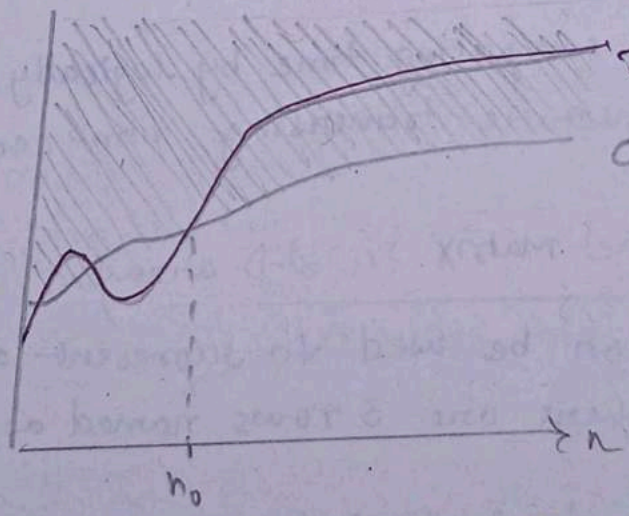
$f(n) = O(g(n))$

③ **Ω - Notation** → for best case

for function $g(n)$, $\Omega(g(n))$ is given by :

$$\Omega(g(n)) = \{ f(n) : \exists \text{ +ve constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n), \ \forall n \ge n_0 \}$$

Intuitively :- Set of all functions whose rate of growth is the same as or higher than that of $g(n)$.



$$f(n) = \Omega(g(n))$$

$g(n)$ is an asymptotic lower bound for $f(n)$.

$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$

$\Theta(g(n)) \subset O(g(n))$

$f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$

$\Theta(g(n)) \subset \Omega(g(n))$

(4)

Sparse Matrix

These are the matrices that have the majority of their elements equal to zero.
(ie matrix having greater no. of zero elements than the non-zero elements)

* Need to use a sparse matrix (instead of simple matrix)

Advantages of Sparse matrix

(i) Storage :- Since it contains lesser non-zero elements than zero. so less memory can be used to store the elements as it evaluates only the non-zero elements.

(ii) Computing time :- In case of searching in sparse matrix, we need to traverse only non-zero elements rather than traversing all the sparse matrix elements.

∴ It saves computing time by logically designing a data structure traversing non-zero elements.

How to store Sparse Matrix in 2-D array

A 2D array can be used to represent a sparse matrix in which there are 3 rows named as :-

(1) row :- it is an index of row where a non-zero element is located.

(2) Column :- it is an index of the column where a non zero elements is located.

(3) Value :- the value of non-zero element is located at the index (row, column).

This means storing non-zero elements with triples. (ie Row, Column, Value).

Example :- If we have a matrix like

0	0	1
1	0	3
0	0	2

then the sparse matrix would be like

	0	1	2	3
Row	0	1	1	2
column	2	0	2	2
Value	1	1	3	2

(∵ 4 non-zero elements thus 4 columns)

The size of the table depends upon the no. of non-zero elements in the sparse matrix.

// C program to store Sparse matrix

```
#include<stdio.h>
int main()
{ int sparse[3][3]={(0,0,1),(1,0,3),(0,0,2)};
  int size 0;
  for(int i=0; i<3; i++)
      for(int j=0; j<3;j++)
        {if(sparse[i][j]!=0)
            size++;
        }
  int matrix[3][size];      // new matrix to store the spare matrix non-zero elements.
  int k=0;
  for(int i=0; i<3; i++)
      for(int j=0; j<3; j++)
       { if(sparse[i][j]!=0)
            matrix[0][k]=i;
            matrix[1][k]=j;
            matrix[2][k]=sparse[i][j];
            k++;
       }
```

```
// Now displaying the output-
for(int i=0; i<3; i++)
    { for(int j=0; j<size; j++)
        { printf("%d \t", matrix[i][j]);
        }
        printf("\n");
    }
// closing main.
```

✱ Calculating the Address of the random element of a 2D Array.

Row - Major Order

$$Add(arr[i][j]) = BA + (i - L1) * (U2 - L2 + 1) * size + (j - L2) * size$$

BA = Base Address or Address of 0th location.
L1 = Lower bound of row L2 = Lower bound of Column
U1 = Upper bound of row U2 = Upper bound of Column

Column - Major Order

$$Add(arr[i][j]) = BA + (j - L2) * (U1 - L1 + 1) * size + (i - L1) * size,$$

Arrays Vs Linked Lists

Problems with Arrays → size of array is fixed
(we can't change after declaration)
→ Requires contiguous memory
→ Mapping by compiler is carried out in constant time.

Basis for Comparison	Array	Linked List
Basic	It is consistent set of a fixed no. of data items.	It is an ordered set consisting of a variable number of data it
Size	Specified during declaration.	No need to specify, grow & shrink during execution
Storage Allocation	Element-location is allocated during compile time.	Element-position is assigned during run time
Order of elements.	Stored consecutively.	Stored randomly.
Accessing the elements	Direct or randomly accessed ie specify the array index or subscript	Sequentially accessed ie Traverse starting from the first node in the list by the pointer.
Insertion/deletion of element	Slow relatively as shifting is required.	Easier, fast and efficient
Memory required	Less	More (bcz of nodes, pointe
Memory utilization	Inefficient	Efficient
Searching	Binary search & linear search.	Linear search.

STATIC Vs DYNAMIC

Static Memory allocation	Dynamic memory Allocation.
(1) Memory is allocated during compilation time	Memory is allocated during execution time.
(2) Used only when the data size is fixed and known in advance before processing.	Used only for unpredictable memory requirement.

MALLOC Vs CALLOC

Basis of Comparison	Malloc( )	Calloc( )
No. of blocks.	Allocates an only single block of requested memory.	Allocates multiple blocks of the requested memory.
Syntax	void * malloc(size_t size);	void * calloc (size_t num, size_t size);
Initialization	malloc( ) doesn't clear & initialize the allocated memory.	calloc( ) initializes the allocated memory to zero.
Manner of Allocation.	malloc( ) function allocates memory of size 'size' from the heap (use sizeof to determine the size of object)	Calloc() function allocates memory of the size of which is equal to num*
Speed	fast	Comparatively slow.

free(ptr) :- it deallocates memory allocated by malloc.
→ takes pointer as an argument.

Self-Referential Structure

Self-referential structures

→ Structure that contains a pointer to a structure of the same type

→ Can be linked together to form useful data structures such as lists, queues, stacks and trees.

→ Terminated with a NULL pointer.

```
struct node {
        int data;
        struct node * next;
        }
```

next :- points to an object of type node
- Referred to as a link.
ie Ties one node to another node.

Linked Lists

It is → Linear collection of self-referential structure called nodes
→ Each node contains two parts :- INFO & NEXT *
→ INFO contains data & NEXT contains address of NEXT node.
→ All nodes connected by pointer links (NEXT)
→ Accessed via a pointer to the first node of the list (START)
→ Link pointer in the last node is set to NULL to mark the list's end.

Use of Linked list over an array when
→ You have an unpredictable number of data items.
→ Your list needs to be sorted quickly.

Advantages of linked lists

→ Stores 'item' sequentially' without restrictions on location,
→ access any item as long as external link to first item maintained.
→ insert new item without shifting
→ deletion of existing item without shifting
→ Size can be expanded / contracted throughout the use.

Disadvantages

→ overhead of links : used only internally, pure overhead.
→ if dynamic, must provide destructor, copy constructor
→ no longer have direct access to each element of the list.
→ O(1) access becomes O(n) access, since we must go through first element and then second, and then second, and then third and so on.

* List-processing algorithm that require fast access to each element cannot (usually) be done as efficiently with linked list, eg:-
  → Binary Search
  → Sorting
  → Append item (add to end of list)

C program to manipulate Linked List

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

typedef struct node
{ int data;
  struct node *next;
}node;

node * start=NULL, *newptr, *ptr, *save;

Void creation();
//Void insertion();
        void begininn();
        Void middleinp(); void middleinv();
        void endin();
//void deletion();
        void beginD();
        Void middleD();
        Void endD();

void traversal();
void reversal(node *head);

int main()
{
  :
  :
}
```

```
Void creation( )
{
    int i, n;
    printf("Enter the no. of nodes to be created \n");
    Scanf("%d", &n);
    newptr = (node *) malloc (sizeof (node));
    printf("Enter data for node 1 \n");
    Scanf("%d", & newptr → data);
    fflush(stdin);
    newptr → next = NULL;
    Start = newptr;
    ptr = start;
    for (i=2; i<=n; i++)
    {
        newptr = (node *) malloc (sizeof (node));
        printf("Enter data for node %d \n ", i);
        scanf("%d", & newptr → data);
        newptr → next = NULL;
        ptr → next = newptr;
        ptr = ptr → next;
    }
```

```
void beginin()
{
    newptr = (node*) malloc (sizeof (node));
    newptr -> next = NULL;
    printf ("Enter data to be inserted at the beginning \n");
    scanf ("%d", & newptr -> data);
    ptr = start;
    if (start == NULL)
        { printf (" List is empty \n");
          start = newptr;
          printf (" Data is store in 1st node succesfully \n");
        }
    else
      { newptr -> next = start;
        start = newptr;
      }
    fflush (stdin);
}
void endin()
{ newptr = (node*) malloc (sizeof (node));
  newptr -> next = NULL;
  printf (" Enter the value to be inserted at the end \n");
  scanf (" %d", & newptr -> data);
  ptr = start;
  while (ptr -> next != NULL)
      ptr = ptr -> next;
  ptr -> next = newptr;
  ptr = newptr;
  fflush (stdin);
}
```

```
// Position based Insertion of an element
Void middleinp()
{ int n, pos, i;
  node *newptr, *ptr, *save ;
  newptr = (node*) malloc (sizeof(node));
  Save = (node*) malloc (sizeof(node));
  if (start == NULL)
        printf ("Linked list is empty \n");
  else
  { printf(" Enter the position where you want to insert \n");
    scanf ("%d", &pos);
    printf ("Enter the data to be inserted \n");
    scanf ("%d", &newptr->data);
    newptr->next = NULL;
    ptr = start ;
    n=1;
    while (ptr->next != NULL)
          n++;
    if (pos <n && pos>1 )
     { for (i=1; i<pos ; i++)
          { Save =ptr;
            ptr = ptr -> next;
          }
       Save->next = newptr;
       newptr->next = ptr;
     }
 }
 fflush(stdin) ;
}
```

```
// Value based insertion
void middle_inv()
{ int val, ch;
  if (start == NULL)
      printf ("List doesn't exist \n");
  else
  { printf ("where you want to insert \n");
    printf ("Press 1: to insert before an item \n");
    printf ("Press 2: to insert after an item \n");
    scanf ("%d", &ch);
    if (ch == 1)
    { printf ("Enter the value before which you want to insert- \n");
      scanf ("%d", &val);
      printf ("Enter the data to be inserted \n");
      scanf ("%d", &newptr->data);
      newptr->next = NULL;
      ptr = start;
      while (ptr->data != val && ptr->next != NULL)
      { save = ptr;
        ptr = ptr->next;
      }
      newptr->next = ptr;
      save->next = newptr;
    }
    else
    { printf ("Enter the value after which you want to insert \n");
      scanf ("%d", &val);
      printf ("Enter your data to be inserted \n");
      scanf ("%d", &newptr->data);
      newptr->next = NULL;
      ptr = start;
      while (ptr->data != val && ptr->next != NULL)
          ptr = ptr->next;
```

```
            newptr->next = ptr->next;
            ptr->next = newptr;
        }//closing of else (inner)
    }// closing outer else.
    fflush(stdin);
}// closing middleinv() function.

Void traversal()
{ if(start==NULL)
        printf("List is empty \n");
  else
  {   ptr= start;
      while(ptr->next!=NULL)
      {  printf("%d=>", ptr->data);
         ptr = ptr->next;
      }
     printf("%d", ptr->data);
     printf("\n");
  }
}

void reversal(node * head)        // passing start on calling
{ if(head==NULL)
    { printf("NULL =>");
      return;
    }
  reversal(head->next);
  printf("%d =>", head->data);
}
```

// use of stack

```
Void deletion ()
{ int a;
  printf("Enter your choice \n");
  printf ("Press 1: Deletion in the beginning \n Press 2: Deletion at
          position in the middle \n Press 3: Deletion at end \n")
  scanf ("%d", &a);
  if(a==1)
     beginD();
  if (a==2)
     middleD();
  if(a==3)
     endD();
  else printf("Wrong Input \n");
  fflush(stdin);
}

void beginD()
{ newptr = (node *) malloc (sizeof(node));
  newptr->next = NULL;
  printf("Enter data to be inserted at the be
  ptr = start;
  if(start == NULL)
        printf("List is empty \n Deletion not possible");
  else
  { if(start->next == NULL)
     { start = NULL; // or free(ptr);
       printf("Only element of the list is deleted successfully \n
     }
    else
    { ptr = start;
      start = start->next;
      free(ptr);
  } } printf("Deletion is successful \n");
}
```

```
void middleD( )
{ int x;
  newptr =(node *)malloc(sizeof(node));
  printf("Enter your choice \n");
  printf(" Press 1: for location based Deletion \n Press 2: for
          position based deletion \n ");
  scanf("%d",&x);
  if(x==1)
  { save=(node*)malloc(sizeof(node));
    int pos, n=1, i;
    if (start==NULL)
          printf("List is empty \n");
    else
    { printf("Enter the position where you want to delete item \n");
      scanf("%d", &pos);
      ptr=start;
      while (ptr->next!=NULL)
          n++;
      if(pos<n)
      { for(i=1; i<=pos; i++)
          { save=ptr;
            ptr=ptr->next;
          }
          save->next=ptr->next;
          free(ptr);
      }
    Else
        printf("Deletion not possible\n");
    } //closing of upper else.
} //closing of if
fflush(stdin);
//closing function.
```

```
if (x==2)
{ save =(node*)malloc(sizeof(node));
  int val;
  if (start==NULL)
        printf("List isempty\n");
  else
   { ptr = start;
     printf("Enter the value to be deleted \n");
     scanf("%d", &val);
     while(ptr->data!=val && ptr->next!=NULL)
     {   save =ptr;
         ptr = ptr->next;
     }
     save->next = ptr->next;
     ptr->next =NULL
     free(ptr);
     printf("Deletion successful \n");
   }
  fflush(stdin);
}
} //closing function.
```

```
void endD()
{
  int val;
  if (start == NULL)
      printf("List doesn't exist \n");
  else
  {  printf("Enter the value to be del
     ptr=start;
     while(ptr->next!=NULL)
          { save=ptr;
            ptr=ptr->next;
          }
     save->next=NULL;
     free(ptr);
  }
}
```

Doubly - Linked list (back-tracking is possible).

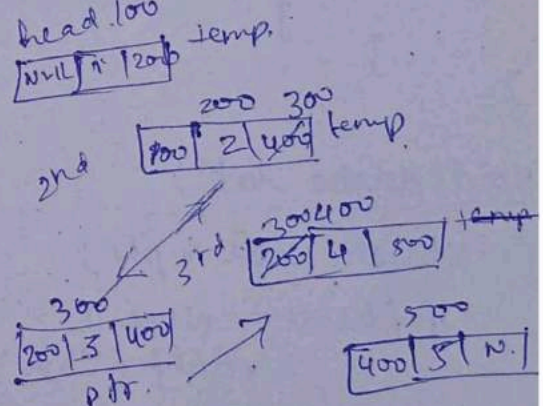
```
#include <stdio.h>
# include<stdlib.h>

struct node
{
    struct node * prev;
    struct node * next;
    int data;
};

struct node *head;
void insert_begin();
void insert_last();
void insert_pos();
void delete_bfgin();
void delete_last();
void delete_pos();
void display();
void search();
void main()
{
    int choice=0;
    while(choice!=9)
    {
        printf("Choose one option from the following list...\n");
        printf("\n1. Insert in beginning \n 2. Insert at last \n
                3. Insert at any random location \n
                4. Delete from Beginning \n 5. Delete from last \n
                6. Delete the node after the given data \n
                7. Search \n 8. Show \n 9. Exit \n");
        printf("Enter your choice : \n");
        scanf("\n%d",&choice);
```

```
switch(choice)
{
  case 1:
     {insert_begin();
      break;}
  case 2:
     {insert_last();
      break;}
  case 3:
     {insert_pos();
      break;}
  case 4:
     {delete_begin();
      break;}
  case 5:
     {delete_last();
      break;}
  case 6:
     {delete_pos();
      break;}
  case 7:
     {search();
      break;
     }
  case 8:
     {display();
     }break;
  case 9:
        exit(0);
        break;
  default:
        printf("Please enter valid choice.");
  }
 } // closing of while
} // closing of main.
```

```
void insert_begin()
{
    struct node *ptr;
    int item;
    ptr = (struct node *) malloc (sizeof (struct node));
    if (ptr == NULL)
        printf("\n Overflow)
    else
    { printf("\n Enter item value to be inserted \n");
      scanf("%d", &item);
      if (head == NULL)
      { ptr->next = NULL;
        ptr->prev = NULL;
        ptr->data = item;
        head = ptr;
      }
      else
      { ptr->data = item;
        ptr->prev = NULL;
        ptr->next = head;
        head->prev = ptr;
        head = ptr;
      }
      printf("Node Inserted \n");
    }
}
```

```
id insert_last()

struct node *ptr, *temp;
int item;
ptr = (struct node*) malloc (sizeof (struct node));
if (ptr == NULL)
        printf ("Overflow");        // as no more memory can be allocated.
else
  { printf ("Enter value \n");
    scanf ("%d", &item);
   ptr->data = item;
    if (head == NULL)
    { ptr->next = ptr->prev = NULL;
      head = ptr;
    }
    else
    { temp = head;
      while (temp->next != NULL)
      {   temp = temp->next;
      }
      temp->next = ptr;
      ptr->prev = temp;
      ptr->next = NULL;
    }
  }
```

```
void insert_pos( )
{ struct node *ptr, *temp;
int item, loc, i;
    ptr=(struct node *) malloc (sizeof (struct node));
    if (ptr == NULL)
            printf ("Overflow");
    else
      { temp = head;
        printf ("Enter value\n");
        scanf ("%d", &item);
        ptr->data = item;
        printf("Enter the location");
        scanf ("%d", &loc);
        for (i=1; i<loc; i++)
        {   temp = temp->next;
            if (temp == NULL)
              { printf (" There are less than %d elements", loc);
                return;
              }
        }
        ptr->next = temp->next;
        ptr->prev = item;
        temp->next = ptr;
        (temp->next)->prev = ptr;
        printf ("Node inserted \n");
}
```



```
void delete_begin()
{ struct node *ptr;
  if(head==NULL)
        printf("Underflow\n");
  else
    { if(head->next == NULL)
      {head = NULL;
       free(head);
       printf("Only node deleted \n");
      }
      else
       { ptr=head;
         head = head->next;
         head->prev = NULL;
         free(ptr);
         printf("node deleted \n");
       }
    }
}
void delete_last()
{ struct node *ptr;
  if(head==NULL)
        printf("\n Underflow \n");

  else
    { if(head->next==NULL)
      { head = NULL;
        free(head);
        printf("Only node deleted \n");
      }
     else
      { ptr=head
        while(ptr->next!=NULL)
              ptr=ptr->next;
       ptr->prev->next=NULL;
       free(ptr);
      } printf("node deleted \n");
```

```
void delete_pos()
{   struct node *ptr, *temp;
    int val;
    printf("Enter the data after which the node is to be
            deleted \n");
    scanf("%d", &val);
    ptr = head;
    while(ptr->data != val)
        ptr = ptr->next;
    if(ptr->next==NULL)
        printf("can't delete \n");
    else
    {   if(ptr->next->next == NULL)
        {   ptr->next = NULL;
        }
        else
        {   temp = ptr->next;
            ptr->next = temp->next;
            temp->next->prev = ptr;
            free(temp);
            printf("node deleted\n");
        }
    }
}

void display()
{   struct node *ptr;
    printf("printing the values ....\n");
    ptr = head;
    while(ptr!=NULL)
        {   printf("%d\n", ptr->data);
            ptr = ptr->next;
        }
}
```

```
Void search()
{ struct node *ptr;
  int item, i=0, flag;
  ptr = head;
  while
  if (ptr==NULL)
        printf("Empty List \n");

  else
    { printf("Enter item which you want to search \n");
      scanf("%d", &item);
      while (ptr!=NULL)
          { if(p->data==item)
              { printf("Enter Item which found at %d", i+1);
                flag=0;
              } break;
            else
                flag=1;
            i++;
            ptr = ptr->next;
          }
  if (flag==1)
    { printf("Item not found \n");
    }
}
```

DATA STRUCTURES APPLICATIONS

(1) Singly Linked List & Doubly Linked List

In computer

1- Implementation of stacks & queues.

2- Implementation of graphs : Adjacency list representation of graphs is most popular which uses linked list to store adjacent vertices.

3- Dynamic memory allocation → of free blocks.

4- Maintaining directory of names

5- representing sparse matrices.

In real world

1- Image viewer → Previous & next images are linked.

2- Previous & next page in web browsers.

3- Music player

(2) Circular linked list

(1) to keep the track of turns in a multi player game

(2) In PCs, where multiple applications are running All the running applications are kept in a circular linked list & the OS gives a fixed time slot to all for running. The OS keeps on iterating over the linked list until all the applications are completed.

(3) In Multiplayer games.

(4) Can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memo all the time, where as in Circular linked list, only one pointe is required (ie End)

(3) Stacks

(LIFO) → (1) During function calls & Recursive Algorithms
(2) Expression Evaluation.
(3) Undo feature in computer keyboard.
(4) Converting an infix to postfix
(5) During Depth first search (DFS) and Backtracking Algorithms etc.
(6) To reverse a word.
(7) A stack of plates / books
(8) Wearing / Removing Bangles.

(4) Queues

(FIFO) (1) At ticket window.
(2) Phone answering system in Call center.
(3) Patients waiting outside the doctor's clinic.
→ (4) Serving requests on a single shared resource like printer, CPU tasks scheduling etc.
(5) To handle congestion in networking.
(6) Sending an E-mail, it will be queued.
(7) Uploading & downloading photo's

Arrays

(1) In phones → When we want to add or delete any contact, the changes will be similar to addition/deletion of elements in an array.
⇒ Music playlist, here playlist is an array & the songs are elements.

(2) Matrices.

(3) Log in passwords.

Stack Vs Heap Memory Allocation

(1) Stack Allocation :- The allocation happens on contiguous blocks of memory. We call it stack memory allocation bcz the allocation happens in the function call stack. The size of memory to be allocated is known to the compiler and whenever a function is called, its variables get memory allocated on the stack. And whenever the function call is over, the memory for the variables is de-allocated. This all happens using some predefined routines in the compiler, the programmer need not to worry about memory allocation & de-allocation of stack variab

- It's a temporary memory allocation scheme where the data members are accessible only if the method () that contained them is currently is running.

(2) Heap Allocation :- The memory is allocated during the execution of instructions written by programmers.
It is not heap data structure, it is called heap because it is pile of memory space available to programmers to allocate & de-allocate.

Everytime when we made an object it always creates in Heap-space & the referencing information to these objects are always stored in stack-memory.

- Here no automatic de-allocation feature is provided. We need to use a Garbage collector to remove the old unused objects in order to use the memory efficiently.
- Heap-memory is accessible / exists as long as the whole application runs.

Key-differences between Stack and Heap Allocations.

<1> In stack, the allocation and de-allocation are automatically done by the compiler whereas in heap, it needs to be done by the programmer manually.

<2> Handling of heap frame is costlier than the handling of stack frame.

<3> Memory shortage problem is more likely to happen in stack whereas the main issue in heap memory is fragmentation.

<4> Stack frame access is easier as the stack have small region of memory & is ~~cache~~ cache friendly, but in case of heap frames which are dispersed throughout the memory so it causes more cache misses.

<5> A stack is not flexible, the memory size allotted can't be changed whereas heap is flexible, and the allotted memory can be altered.

<6> Accessing time of heap takes, is more than a stack.

Parameter	Stack	Heap.
Basic	Memory allocation in contiguous block.	Memory is allocated in any random order.
Allocation & De-allocation	Automatic by compiler instructions	Manual by programmer.
Cost	Less	More
Implementation	Easy	Hard
Access time	faster	Slower
Main Issue	Shortage of memory	Memory fragmentation.
Locality of reference	Excellent	Adequate
Safety	Thread safe, data stored can only be accessed by owner	Not thread safe, data stored visible to all threads.
flexibility	fixed size	Resizing is possible
Data type Structure	Linear	Hierarchical.

Circular Singly Linked List

(As back-tracking was not possible ie to access the previous element in singly linked list but here it is possible by repeated traversing).

In circular singly linked list, we take an external pointer that points to the last node of the list. If we have a pointer last pointing to the last node, then last→next will point to the 1st node.

① INSERTION

```
typedef struct node{
        int data;
        struct node * next;
        }node;

node * start = NULL, * ptr, * newptr;

void insertion_begin();
void insertion_end();
void insertion_middle();

int main()
{ int ch;
  printf("Enter your choice: \n");
  printf("1. Insertion at beginning \n 2. Insertion at end \n
          3. Insertion at middle \n");

  scanf("%d", &ch);
  switch(ch){
      case 1:
          insertion_begin();
          break;
      case 2:
          insertion_end();
          break;
      case 3: insertion_middle(); break;
}
```

```
void insert_begin()
{
    node *
    newptr = (node*) malloc(sizeof(node));
    printf("Enter the data to be inserted \n");
    scanf("%d", &newptr->data);
    if(start == NULL)
        { newptr = start;
          start->next = newptr;
        }
    else
        { ptr = start;
          while(ptr->next != NULL)
            {   ptr = ptr->next;
            }
          newptr->next = start;
          ptr->next = newptr;
          start = newptr;
        }
}

void insertion_end()
{   printf("Enter data \n"); scanf("%d", &newptr->data);
    ptr = start;
    while (ptr->next != NULL)
        ptr = ptr->next;
    ptr->next = newptr;
    newptr->next = start;
}
```

★ Circular doubly linked list is also same as doubly linked list

just last->next = start;

Stacks

features

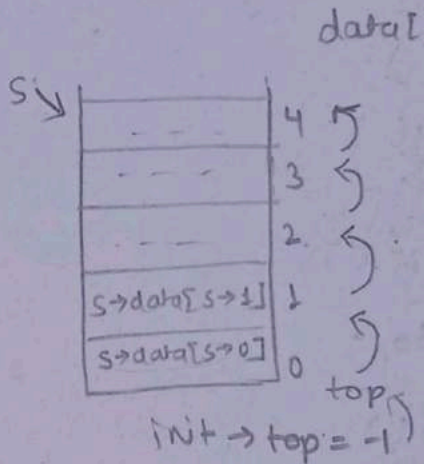
1. A stack is an ordered collection of data items where access is possible only at one end called 'top' of the stack
2. Works on the LIFO principle.
3. Insertion operation is called PUSH
4. Deletion operation is called POP.

Array Implementation of Stack

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5
typedef struct stack {
            int data[MAX];
            int top;
          }stack;

void init(stack *);
int empty(stack *);
int full(stack *);
int pop(stack *);
void push(stack *, int);
void print(stack *);

int main()
{ int ch, x;
  printf init(&s);
  stack s;
  do
   { printf("Enter your choice.\n");
     printf("1. Push\n 2. POP\n 3. Print\n 4. Quit\n");
     Scanf("%d", &ch);
     Switch(ch)
     {
```

```
case 1:
        printf("Enter the data to be inserted \n");
        scanf("%d", &x);
        if(!full(&s))
                push(&s, x);
        else
                printf("stack is full \n");
        break;

case 2:
        if(!empty(&s))
                { x = pop(&s);
                  printf("Poped element is : %d \n", x);
                }
        else
                printf("Stack is empty \n");
                ~~bbb~~
        break;

case 3:
        print(&s);
        break;

    }
  } while(ch!=4);
}
void init(stack *s)
{
  s->top = -1;
}
```

```
int empty (stack *s)
{ if(s->top == -1)
        return 1;
  return 0;
}
int full (stack *s)
{ if (s->top == MAX-1)
        return 1;
  return 0;
}
int pop (stack *s)
{ int x;
  x = s->data[s->top];
  s->top = s->top-1;
  return(x);
}
void push (stack *s, int x)
{ s->top = s->top+1;
  s->data[ s->top] = x;
}
void print(stack *s)
{ int i;
  for(i=s->top; i>=0; i--)
      printf("%d\n", s->data[i]);
}
```

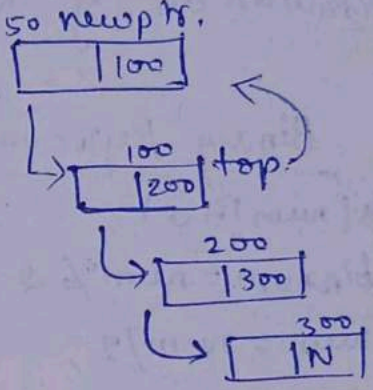


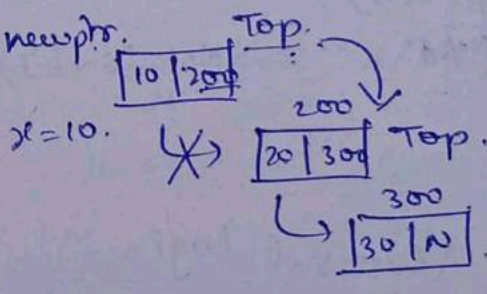
Linked List Implementation of Stack

// no need of full() function as size doesn't matter in linked list

```
typedef struct stack
    {   int data ;
        struct stack * next ;
    } Stack;

void init ( );
int empty ( );
int pop ( );
void push(int );
Void print ( );

Stack * top;

int main ( )
{  int x, ch;
   init( );
   do
   { printf(" Enter your choice \n ");
     printf(" 1. Push\n 2. Pop \n 3. Print \n 4. Quit");
     scanf("%d", &ch);
     switch(ch)
     {
       case 1:
             printf("Enter the data \n ");
             scanf("%d", &x);
             push(x);
             break;

       case 2:
             if(!empty())
                { x=pop();
                  printf ("Popped element is %d \n", x);
                }
             else
                printf (" Underflow \n ");
                break;

       Case 3:
             print();
             break;
   }
```

```
    }while(ch!=4);
  }
void init()
{ top=NULL;
}
void push(int x)
{ stack * newptr;
  newptr=(stack*) malloc(sizeof(stack));
  newptr->data=x;
  newptr->next=top;
  top=newptr;
 }
int empty()
 { if(top==NULL)
      return 1;
   return 0;
 }
int pop()
{  int x;
   stack * newptr;
   newptr=top;
   x=newptr->data;
   top=newptr->next;
   free(newptr);
   return(x);
}
void print()
{ while(top!=NULL)
    { printf("%d\n",top->data);
    }top=top->next;
}
```





Applications of Stacks

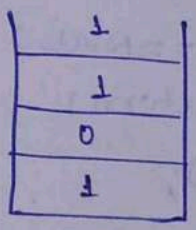
- Binary representation of a number
- String reversal
- Reverse Polish Notation
  - → Infix to prefix
  - → Infix to Postfix
- Expression evaluation
- Simulation of Recursion.

Binary Representation Logic

```
while(num!=0)
{ binnum = num%2;
  num = num/2;
  push(binnum);
}
i=top;
for(;i>=0;i--)
  { x=pop(&s);
    printf("%d\n", s->data[s->i]);
  }
```

2	13	1
2	6	0
2	3	1
	1	

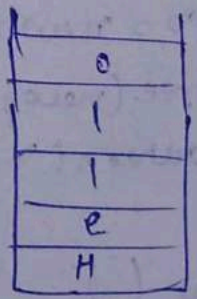
⇒ 1101



String Reversal Logic

[H|e|l|l|o|\0]

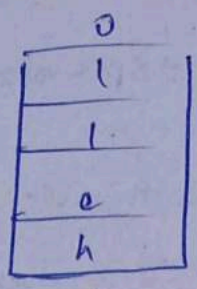
```
for(i=0; str[i]!='\0'; i++)
    push(&s, str[i]);
printf("Reverse string is \n");
i=s->top;
for(; i>=0; i--)
  { x=pop(&s);
    printf("%c\n", s->data[s->i]x);
  }
```

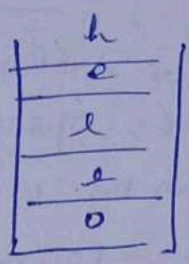


O/P ⇒ O l l e H

Palindrome

Input a string str
the store it in stack 1
then pop top element
of S1 and push into S2.





store as it is

then check (S1->data[s->top] == S2->data[s->top)
till i>=0
if fulfilled then palindrome
otherwise not.

* Now a days, every procesor uses postfix notations to solve the arithmetic operations.
→ (we can maintain priority of operators without using parenthesis in this expression)
thus memory reduction ie no need to store parenthesis
→ Here we don't need to check the priority also, it is automatically in correct order.

RPN → Reverse Polish Notation.

Precedence of operators

#	highest	3
^		
*, /		2
+, -		1
$	lowest.	

Reverse Polish Notation (RPN) or Postfix Expr.

→ A notation for arithmetic expressions ie operators written after the operands.

→ Expression can be written without using parenthesis.

→ Developed by Polish logician, Jan Lukasiewics in 1950's.

Converting Infix to RPN

(Stack Algorithm)

1. Initialize an empty stack of operators.
2. while no error has occurred & end of infix expression not reached.
   a. Get next Token in the infix expression
   b. if Token is
      i) a left parenthesis :- Push it onto the stack
      ii) a right parenthesis :- Pop & display Stack elements until a left parenthesis is encountered, but do not display (
      (Error if stack empty with no left-parenthesis found.)

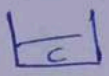
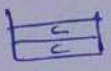
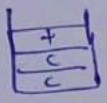
      (iii) an operator :

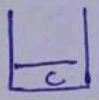
      If stack empty or token has higher priority than top stacks element, push Token on stack ('(' has lower priority than operators)

      Otherwise, pop & display the top stack element ~~If stack empty or Token has higher priority than top stack element, push~~ then repeat the comparison of Token with new top stack item.

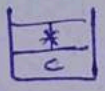
      (iv) an operand :- Just Display it.
3. When end of infix expression reached, pop & display stack items until stack is empty.

Expression :- ((A+B)*C)/(D-E)

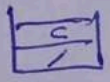
1. Push C 
2. Push C 
3. Display A → A
4. Push + 
5. Display B → AB
6. Right parenthesis. thus pop + and display. Pop until left-parenthesis but don't display it.

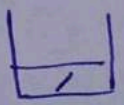


AB+

7. Push * 
8. Display C → AB+C
9. Right parenthesis thus pop until left-parenthesis and display *

AB+C*

10. Push /
11. Push C 
12. Display D → AB+C*D
13. Push - 
14. Display E → AB+C*DE
15. Pop ~~everything~~ until '(' and display. firstly -

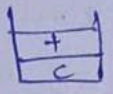
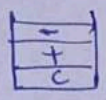


AB+C*DE-

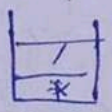
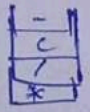
Now our String is exhausted, thus check if(Stack != Empty) → if True thus start poping every element

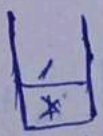
16. Display / → **AB+C*DE-/** is the postfix notation.

Expression:- (A+B-C)*(D-E)/(F-G+H)

1. Push (
2. Display A ⟶ A
3. Push + 
4. Display B ⟶ AB
5. push - 
6. Display C ⟶ ABC
7. Right parenthesis thus pop until (
8. Display ABC-+
9. Push * 
10. Push (
11. Display D ⟶ ABC-+D
12. Push - 
13. Display E ⟶ ABC-+DE
14. Right parenthesis thus pop until (

ABC-+DE-

15. Push / 
16. Push (
17. Display F ⟶ ABC-+DE-F
18. Push - 
19. Display G ⟶ ABC-+DE-FG
20. Push + 
21. Display H ⟶ ABC-+DE-FGH
22. Right parenthesis thus pop until)



ABC-+DE-FGH+-

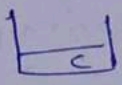
Now pop everything

ABC-+DE-FGH+-/*

is the postfix expression.

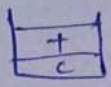
Expression:- (A + B ^ C) * D + E

1. Push (



2. Display A ⟶ A

3. Push +

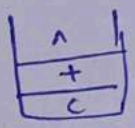


4. Display B ⟶ AB

5. ~~Check priority of ^ with +~~ (^ > +)
thus ~~pop +~~ and then push ^

6. Display AB +
6. Display C ⟶ ~~AB+C~~ AB + C

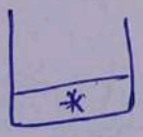
5. Push ^



6. Display C ⟶ ABC

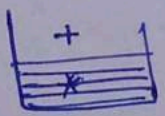
7. Pop until (
and display ⟶ ABC^+

8 Push *



7. Display D ⟶ ABC^+D

8 ~~Pop~~ * and display the push +



⟶ ABC^+D*E

9 Display E

Now string is exhausted
Now pop everything till stack is not Empty.

Postfix :- Expression. ABC^+D*~~E~~+

Infix Expression :- P*Q^R+S ⟶ (P*Q^R+S)

Priority order
^ > * > +

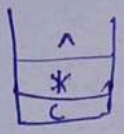
Push (

1. Display P
2. Push *

3. Display Q ⟶ PQ
4. Check priority of ^ with *

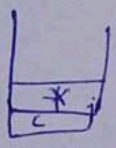
(^ > *)

thus push ^



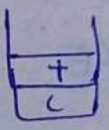
5 Display R ⟶ PQR

6 Pop ^ and display ⟶ PQR^



7. Pop * and display ⟶ PQR^*

8. Push +



9. Display $ ⟶ PQR^*$

String is exhausted now
thus pop everything till stack is not empty and display.

Postfix Expr :- **PQR^*S+**

Queues → an ordered collection of data items.

FIFO based structure.

Insertion in queue → Enqueue (at Rear)
Deletion in queue → Dequeue. (at front)

Applications in

① I/O buffers

② scheduling queues in a multi-user computer system

Eg printer queue
→ when files are submitted to a printer, they are placed in the printer queue.

Queues
- → Resident queue → On disk, waiting for memory
- → Ready queue → In memory - needs only CPU to run
- → Suspended queue → waiting for I/O transfer or to be re-assigned the CPU.

Queues Vs Stacks

Stacks	Queues.
• Stacks are a LIFO container => Store data in the reverse of order received.	Queues are a FIFO container => Store data in the order received.
• Stacks then suggest applications where some sort of reversal or unwinding is desired.	Queues suggest applications where Services is to be rendered relative to order received.

★ Stack & Queues can be used in conjunction to compare different orderings of the same data set.

★ From an ordering perspective, Queue are the 'opposite' of stacks.

Array Implementation

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

typedef struct queue
{
    int f, r;
    int data[MAX];
}queue;

void create(queue *);
void int empty(queue *);
int full(queue *);
void enque(queue *, int);
void int deque(queue *);
void print(queue *);

int main()
{
    queue q;
    create(&q);
    char y = 'y';
    while(y == 'y')
    {
        int ch, x;
        printf("Enter your choice \n");
        printf("\n Press 1: Enqueue \n Press 2: Dequeue \n
                Press 3: Print the elements \n");
        scanf("%d", &ch);
```

```
switch(ch){
    case1:
        { if(!full(&q))
          { printf("Enter the element to be inserted \n");
            scanf("%d", &x);
            enque(&q, x);
          }
          else
              printf("Element can't be inserted \n");
          break;
        }
    case2:
        { if(!empty(&q))
          { x = deque(&q);
            printf("Dequeued element is %d \n", x);
          }
          else
              printf("Element can't be removed since
                      queue is empty \n");
          break;
        }
    case3:
        { printf(&q);
          break;
        }
    default:
          printf("Invalid input \n");
}
fflush(stdin);
printf("Do you want to input again? y/n \n");
scanf("%c", &y);
} //closing of while
//closing of main
```

```
void create(queue *q)
{
    q->f = q->r = -1;
}

int empty(queue *q)
{
    if (q->f == -1)          // or it can be q->r == -1 also.
        return 1;
    return 0;
}

int full(queue *q, int x)
{
    if (q->r == MAX-1)
        return 1;
    return 0;
}

void enque(queue *q, int x)
{
    if (q->r == -1)
    {
        q->r = q->f = 0;
        q->data[q->r] = x;
    }
    else
    {
        q->r = q->r+1;
        q->data[q->r] = x;
    }
}

int deque(queue *q)
{
    int x;
    x = q->data[q->f];
    if(q->r == q->f)
        q->r = q->f = -1;
    else
        q->f = q->f+1;
    return x;
}
```

```
void print(queue *q)
{  int i;
   if (!empty(q))
      { for( i=q->f ; i<=q->r ; i++)
            printf("%d \n", q->data[i]);
      }
   else
     printf("Queue is empty \n");
}
```

Queue Implimentation by Linked List

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

typedy struct node
{   int data;
    struct node * next;
}node;

typedy struct queue
{   node *r, *f;
}que;

void init(que* );
int empty (que *);
void enque ( que *, int );
int deque ( que * );
void print ( que * );
que q;
int main()
{   int ch, x;
    do
    {  printf("\nEnter your choice \n");
       printf( "\n 1. Enqueue\n 2. Dequeue\n 3. Print \n 4. Exit\n ");
       scanf( "%d", &ch);
       switch(ch)
       {
        case 1 :
              printf( " Enter the data to enqueue \n");
              scanf("%d", &x);
              enqueue(&q, x);
              break;
    case 2:
              x = deque(&q);
              printf( "Dequeued Element is : %d \n ", x);
              break;
```

```
        case 3:
                print(&q);
                break;
        case 4:
                printf (" Process End now \n ");
        default:
                printf(" Please Enter Valid Input \n ");
        }
    }while(ch!=4);
    fflush(stdin);

}
void init ( que *q )
{  q->r = q->f = NULL;
}
int empty( que *q )
{ if(q->f == NULL)
        return 1;
  return 0;
}
void enque( que *q , int x)
{ node *newptr;
  newptr = (node*)malloc(sizeof( node ));
  newptr -> data = x;
  newptr->next = NULL;
  if (empty(q))
        q->f = q->r = newptr;
  else
    { (q->r)->next = newptr;
      q->r = newptr;
    }
}
```

```
int deque(que *q)
{
  int x;
  if(q->r==NULL)
     { printf("Queue is Empty");
       x=0;
     }
  else
     {
       x=(q->r)->data;
     if(q->r==q->f)
          q->r=q->f=NULL;
     else
          q->f=(q->f)->next;
     }
  return x;
}
void print(que *q)
{  if(q->f!=empty)
     {   node *ptr;
         ptr=q->f;
         while(q->f!=NULL)
           {  printf("\n%d\n",(q->f)->data;
              q->f=(q->f)->next;
           }
         q->f=ptr;
         free(ptr);
     }
  else
      printf("Queue is Empty\n");
}
```

TREE DS

Tree → it is a non-linear data structure in which data is organized in a hierarchical manner (hierarchical order to store the relationship b/w different data elements).

→ It is a collection of elements called 'nodes', linked to each other on the basis of parent children relationship

→ starts with a specially designated node called as 'root' with no parent.

→ It don't contains any cycle

→ No duplicate values are allowed.

Key terms

- Tree :- a subset of trees. $T = \{t_1, t_2, t_3, \ldots t_n\}$
- Root :- specially designated node without any parent through which the entire tree is referenced.
- Node :- a block containing data along with links to its children.
- Children :- All immediate successors of a node
- Subtree :- A partial tree of an existing tree. Each node itself a tree.
- Siblings :- Nodes with the same parent
- Leaf Node :- The outer most node having no children in tree. They have degree of zero and are also known as external nodes.
- Height :- No. of levels within a tree. Root node at level 0.
- Degree of node :- No. of children of a node
- Degree of a tree :- Max degree of a node within the tree.

Binary tree

→ A tree is binary if each node of the tree can have maximum of two children.

→ A binary tree is said to be complete binary tree if each of its node has either two children or no child at all.

→ No. of nodes at any level 'i' in a complete binary tree is given by 2^i

Total no. of nodes 'n' in complete binary tree of height 'h'

$$n = 2^0 + 2^1 + 2^2 + \cdots + 2^h$$

$$\boxed{n = 2^{h+1} - 1}$$

★ Representation of Binary tree using Array

→ Nodes are numbered (labelled) sequentially level by level & from left to right.

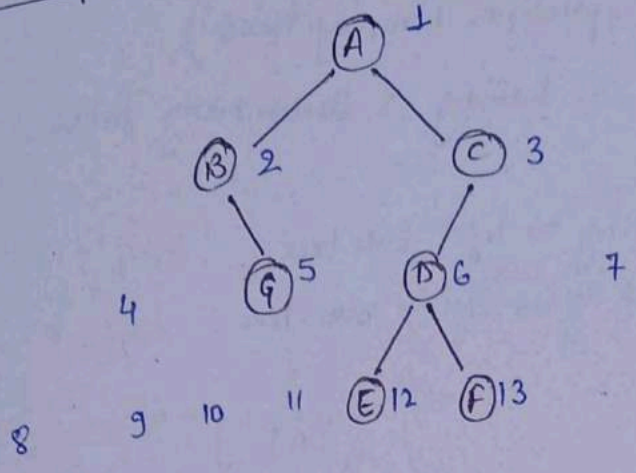
→ Empty nodes (ie non-existing one) are also numbered.

⇒ When the data of the tree is stored in an array, then the number appearing against the node will work as indices of the node in an array.

★ Location '0' stores the size of the tree in terms of total no. of nodes (existing or non-existing both)

- Index of left child of a node $i = 2 * i$
- Index of right-child of a node $i = 2 * i + 1$
- Index of parent of a node $i = i/2$
- Index of sibling of a node i, if i is a left child = $i+1$
- " " " " " " " " if i is a right child = $i-1$

Example.



Array

0	1	2	3	4	5	6	7	8	9	10	11	12	13
13	A	B	C	\0	G	D	\0	\0	\0	\0	\0	E	F

size of the tree

All non-existing children are shown by '\0' in the array.

① Index of left-child of a node $i = 2*i$

Eg $\underset{(3)}{C} \rightarrow 2\times(3) = 6.$

② Index of right-child of a node i

Eg $\underset{(2)}{B} = 2*i+1 = 5$

③ Index of parent of a node $i = i/2$

Eg. parent of D (6) = $6/2 = 3$. ie C.

Advantages

① Given a child node, its parent node can be determined immediately. If a child node is at location n in the array then its parent is at n/

② It can be implemented easily in languages in which only static memory allocation is directly available.

Disadvantages

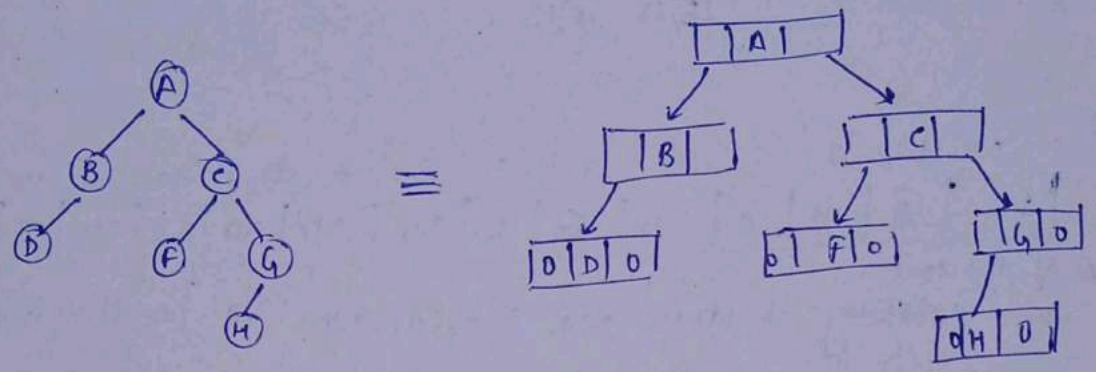
① Insertion/Deletion of a node causes considerable data movement up and down the array, using an excessive amount of processing tim

② Wastage of memory due to partially filled trees.

Linked List Representation

(most commonly represents binary trees)

* Each node can be considered as having 3 elementary fields.
  - → a data field
  - → left pointer, pointing to left-sub tree
  - → right pointer, pointing to right sub-tree.



Disadvantage

1) Wastage of memory for null pointers.
2) Given a node, it is difficult to determine the parent of the node.
3) Its implementation algorithm is more difficult in languages that do not offer dynamic storage techniques.

```
typedef struct treenode
{   int data;
    struct treenode *left, *right;
} tnode;
```

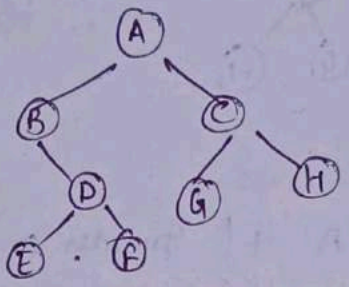
Binary Tree Traversal

- Preorder — NLR
- Inorder — LNR
- Postorder — LRN

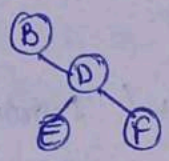
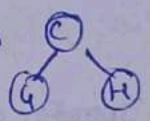
① Preorder Traversal (NLR)

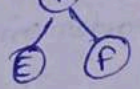
- Visit root node (N)
- Traverse left subtree (L) in preorder
- Traverse right subtree (R) in preorder.

Eg The functioning of pre-order traversal of a non-empty binary tree



Apply preorder on tree.

① A + (preorder on ) + (preorder on )

② A + (B + preorder on ) + (C + preorder on G + preorder of H)

③ A + (B + (D + preorder on E + preorder on F)) + (C + G + H)

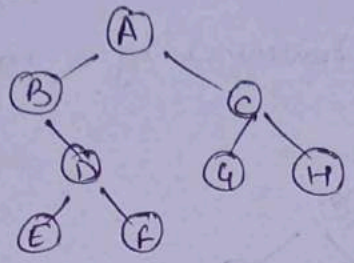
④ A + (BDEF) + (CGH)

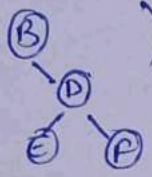
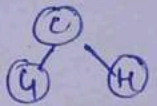
⑤ ABDEFCGH

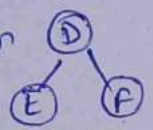
② Inorder Traversal (LNR)

- Traverse left subtree in inorder
- Visit root node
- Traverse right subtree in inorder.

Eg The functioning of inorder traversal of a non-empty binary tree



① (Inorder on ) + A + (Inorder on )

② (B + Inorder on ) + A + (Inorder on G + C + Inorder on H)

③ (B + (Inorder on E + D + Inorder on F)) + A + (G C H)

④ (BEDF) + A + (GCH)

⑤ BEDFAGCH

Recursive

```
void inorder( treenode *T)
{
 if(T!=NULL)
 {  inorder(T->left);
    printf("%d", T->data);
    inorder(T->right);
 }
```

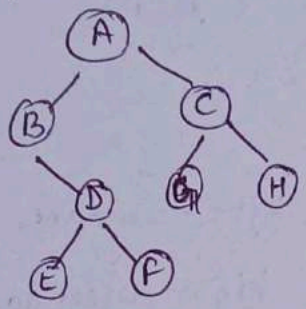
Non Recursive

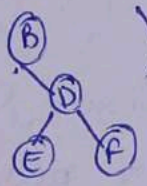
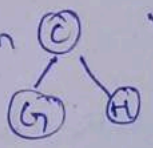
```
void non_rec_inorder( tnode *T)
{ stack s;
  init(&s);
  printf("\n");
  if(T==NULL)
        return;
  while(T!=NULL)
      { push(&s, T);
        T = T->left;
      }
  while( !empty(&s))
      { T=pop(&s);
        printf("%d", T->data);
        T = T->right;
        while( T!=NULL)
            { push(&s, T);
              T = T->left;
            }
      }
}
```

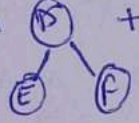
③ Postorder Traversal (LRN)

- Traverse left subtree in Postorder
- Traverse right subtree in Post order
- Visit root node

Eg



① (Post order on ) + (post order on ) + A

② (post order on  + B) + (post order on G + post order on C) + A

③ ((post order on E + post order on F + D) + B) + (GHC) + A

④ (EFDB) + GHC + A

⑤ EFDBGHCA

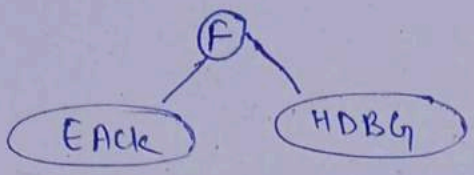
Creating a Binary Tree

① from preorder & inorder traversals.

Inorder: EACKFHDBG (LNR)

Preorder: FAEKCDHGB (NLR)

Step 1



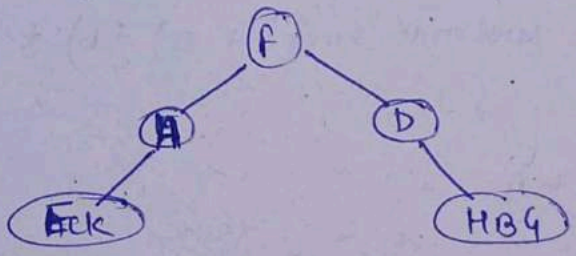
Left descendants = EACK of root F

Right descendants = HDBG.

Step 2 Among EACK, A comes first in preorder traversal.

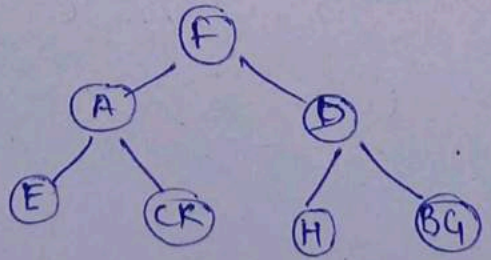
∴ A is root of the subtree with nodes EACK

Similarly D comes first in right descendants in Pre-order traversal thus D is the root of the subtree with nodes HDBG.



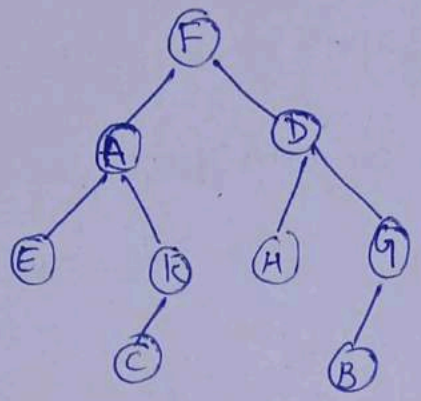
Step 3 Now for root A, its left left descendant is E & right " is CK

Similarly for D, H is left & BG is right descendant as per Inorder.



Step 4 Among CK, K comes 1st in preorder traversal. Thus, K is root & C is left child.

Similarly among BG, G comes first in pre-order traversal thus G is root node & B is left child.

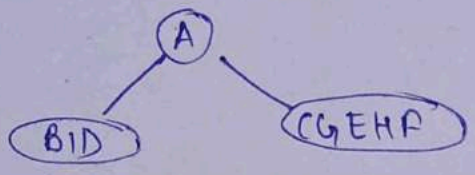


② from post order & in order traversal.

Inorder : BIDACGEHF (LNR)

Post order : IDBGCHFEA (LRN)

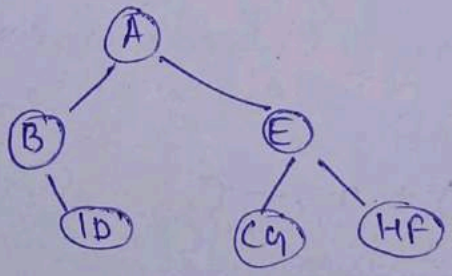
Step 1



On evaluating both orders.

Step 2 In BID, B comes last in post order thus it is node for BDI nodes

Similarly among CGEHF, E comes last in post order thus it is root for nodes CGEHF.

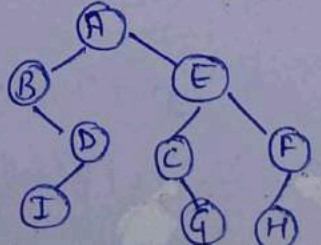


Step 3

Among ID; D comes last in post order thus D is root node & in inorder I comes first D thus it is ~~on~~ left child.

Similarly

Among CG & HF, C & F are root nodes as per post order & as per inorder G is right child & H is left child.



Recurrences

A recurrence is an equation or inequality.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Eg $a = 2,\ b = 2,\ f(n) = n$

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

Various methods to solve recurrence eqn. to get the time complexity.

1) Substitution method
2) Iterative method
3) Master method.
4) Recursion Tree

(1) SUBSTITUTION METHOD

Steps

1 - Guess the form of the solution
2 - Verify by mathematical induction
3 - Solve for constant.

We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values. Hence the name "substitution method"

Eg $T(n) = 4T(n/2) + n$ — (1)

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

Substituting $T(n/2)$ in eqn (1)

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

$T(n) = 16T(n/4) + 2n + n$

$T(n) = 16T(n/4) + 3n$ — (2)

$= 2^{2n} T(n/2^n) + (2^2 - 1)n$

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

now $n \to n/4$ in eq (1)

$$T(n/4) = 4\left[T(n/8)\right] + 3n/4$$

Substituting this value in eq^n (2)

$$T(n) = 16\left[4T\left(\frac{n}{8}\right) + \frac{3n}{4}\right] + 3n$$

$$= 64T\left(\frac{n}{8}\right) + 4n + 3n$$

$$= 256\,T\left(\frac{n}{16}\right) + 15n$$

$$= 2^{2^3} T\left(\frac{n}{2^4}\right) + (2^4 - 1)n$$

$$T(n) = 64\,T(n/8) + 7n$$

$$= 2^{2\times 3} T\left(\frac{n}{2^3}\right) + (2^3 - 1)n$$

General eqn is

$$\therefore T(n) = 2^{2i} T\left(\frac{n}{2i}\right) + (2^i - 1)n$$ //

Let $\frac{n}{2i} = 1$

$2^i = n$

$\log n = i$

$$\therefore T(n) = n^2 T(1) + (n-1)n$$

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

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

$\therefore$ Time complexity $\to O(n^2)$ //

Eg by substitution.

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad \text{---(1)} \quad , \quad T(1) = 1.$$

Replace $n \to n/2$, Eq(1) becomes

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

Substituting this value in eq (1)

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

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

$$T(n) = 4T(n/4) + 2n \quad \text{---(2)} \quad = 2^2 T(n/2^2) + 2n$$

Now $n \to n/4$ in eq (1)

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

Substituting this value in eq^n (2)

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

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

$$T(n) = 8T(n/8) + 3n \quad = 2^3 T(n/2^3) + 3n$$

∴ General eq^n is

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

now let $\frac{n}{2^i} = 1 \qquad (\because T(1) = 1)$

$$n = 2^i$$

$$\log_2 n = i$$

$$\therefore T(n) = nT(1) + n\log n$$

$$= n + n\log n$$

∴ Time complexity is $O(n\log n)$ //

By Iterative Method

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

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

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

$$= 2^2[2T(n/8) + n/4] + 2n$$

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

∴ After k iteration, $T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$ ——— (1)

Sub problem size is 1 after $\frac{n}{2^k} = 1$

$$2^k = n$$

$$\log n = k$$

∴ After long iterations, the sub-problem size will be.

So, when $k = \log n$ is put in eqn (1),

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

$$= nC + n\log n \quad [\text{say } c = T(1)]$$

$\therefore O(n\log n)$

is the time complexity.

By Recursion Tree Method

* Recursion tree show successive expansions of recurrences using trees.

* Recursive tree model the costs (time) of a recursive execution of an algorithm that is composed of two parts :-

  ① cost of non-recursive part
  ② cost of recursive call on smaller input size.

* A tree node represents the cost of a sub-problem (recursive function invocation).

* To determine the total cost of the Recursion Tree, evaluate :-

  ① cost of individual node at depth 'i'
  ② sum up the cost of all nodes at depth 'i'
  ③ sum up all per-level costs of the Recursion tre

Some formulae

$$a^{\log_y b} = b^{\log_y a}$$

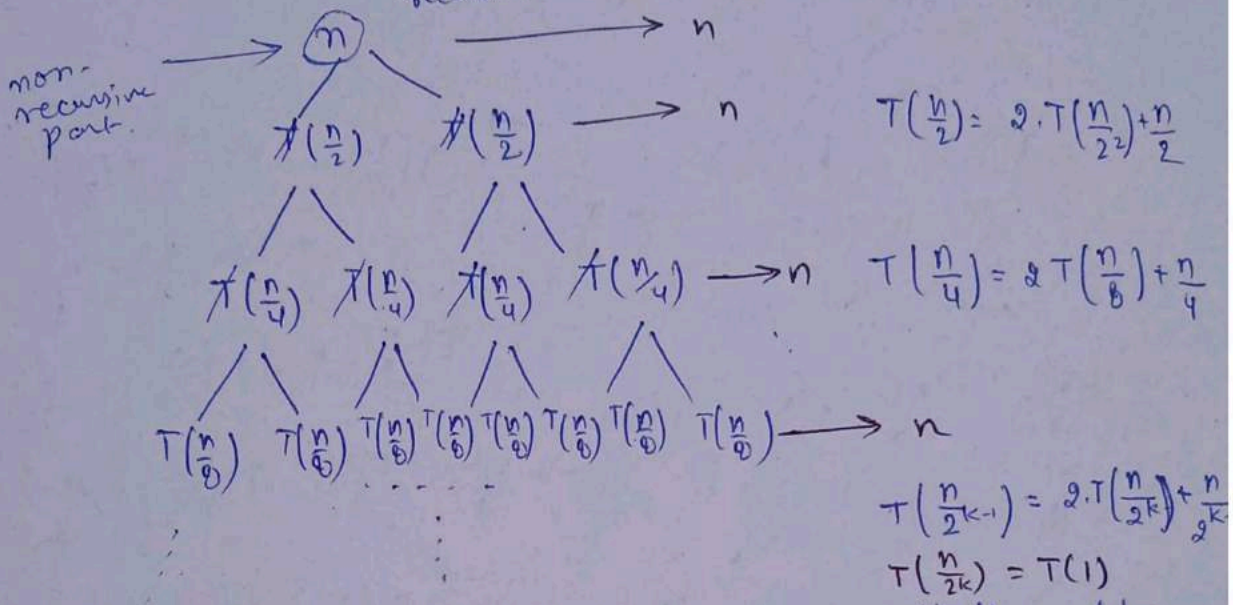
$$x^0 + x^1 + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1} \quad ; x \neq 1$$

$$x^0 + x^1 + x^2 + \cdots = \frac{1}{1 - x} \quad ; |x| < 1$$

Example.

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

recursive. non-recursive part



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

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

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

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

continue expanding until the problem size reduces to 1.

Total cost = cost of leaf nodes + cost of internal nodes

Total cost = (cost of a leaf node × total leaf nodes) + (sum of costs at each level of internal nodes)

Total cost $= L_c + I_c$

$\because \frac{n}{2^k} = 1 \quad \log_2 n = k \quad$ or $\quad k = \lg n$

$L_c = 2^k = 2^{\lg n} \Rightarrow n^{\lg 2} \Rightarrow n$

$\therefore L_c = n$

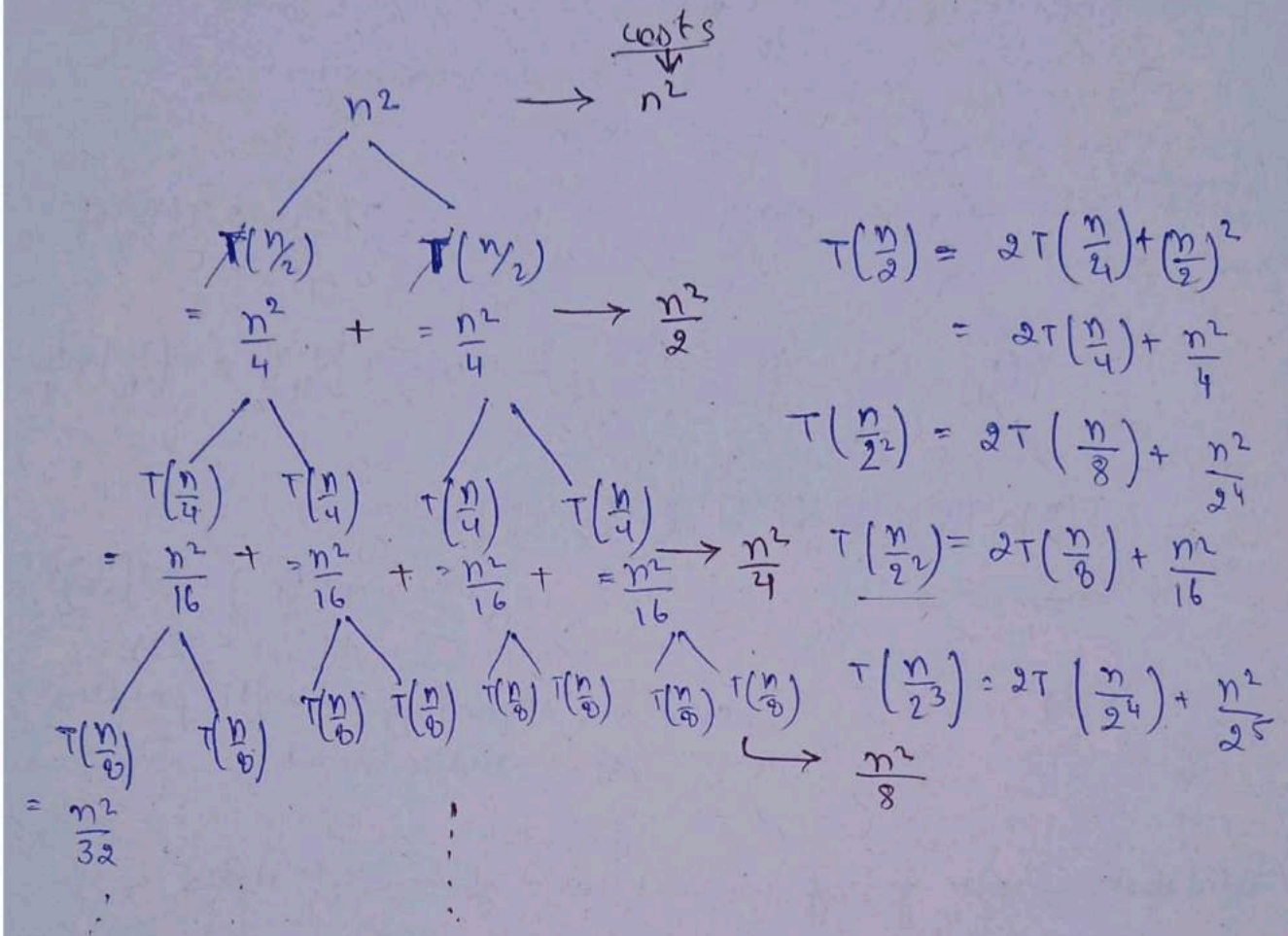
& $I_c = k \cdot n \quad (\because k$ levels$)$

$I_c = n \lg n$

$\therefore$ Total cost $= L_c + I_c = n + n \lg n$

$\therefore O(n \lg n)$ //

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



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

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

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

$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{8}\right) + \frac{n^2}{16}$

$T\left(\frac{n}{2^3}\right) = 2T\left(\frac{n}{2^4}\right) + \frac{n^2}{2^5}$

→ $\frac{n^3}{2^{k-1}}$ (k = levels)

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

$n = 2^k$

$\log_2 n = k$

$\lg n = k.$

∵ Total cost $= L_c + I_c$

for $L_c = 2^k$ (as leaf ↑ as the order of 2 thus at kth level we will have 2^k leaves)

$2^{\lg n} = L_c$

$n^{\lg 2} = L_c$

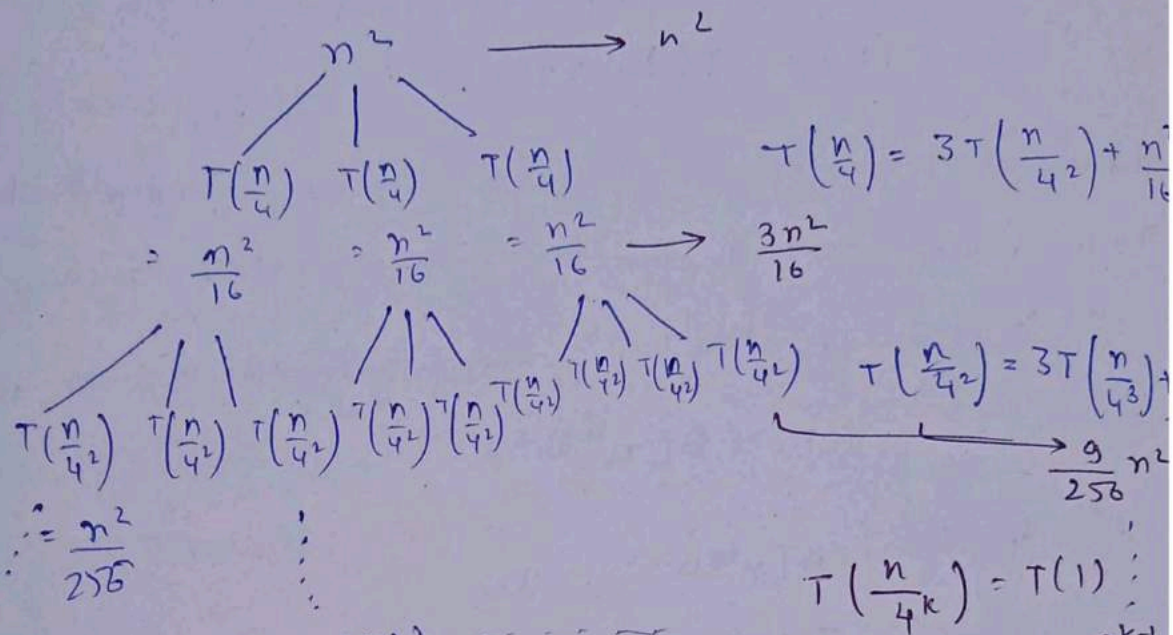
$L_c = n$

for $I_c = n^2\left[\left(\frac{1}{2}\right)^0 + \left(\frac{1}{2}\right)^1 + \left(\frac{1}{2}\right)^2 + \cdots + \left(\frac{1}{2}\right)^{k-1}\right]$

$= n^2\left[\frac{1}{1-\frac{1}{2}}\right] \Rightarrow 2n^2$ (using formula)

∴ Total cost $= L_c + I_c = n + 2n^2 \Rightarrow T(n) \in O(n^2)$

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



$T\left(\frac{n}{4}\right) = 3T\left(\frac{n}{4^2}\right) + \frac{n^2}{16}$

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

$T\left(\frac{n}{4^k}\right) = T(1)$

$4^k = n$

$k \log_4 4 = \log_4 n$

$k = \log_4 n$

$L_c = 3^k$

$= 3^{\log_4 n}$

$L_c = n^{\log_4 3}$ (Total no. of leaf cost)

$\therefore$ Total cost $= L_c + I_c$

where $I_c = n^2 \left[\left(\frac{3}{16}\right)^0 + \left(\frac{3}{16}\right)^1 + \left(\frac{3}{16}\right)^2 + \cdots + \left(\frac{3}{16}\right)^{k-1}\right]$

$= n^2 \left[\frac{1}{1 - 3/16}\right] \Rightarrow \frac{16}{13} n^2$

$\therefore$ total cost $= n^{\log_4 3} + \frac{16}{13} n^2$

$(\because \log_4 3 < 1)$

$\therefore T(n) \in O(n^2)$

Master Method

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $a >= 1$, $b > 1$ are constants & $f(n)$ is asymptotically +ve fu

To use master method, we have to remember 3 cases :-

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

$$T(n) = \theta(n^{\log_b a})$$

(2) If $f(n) = \theta(n^{\log_b a})$ then

$$T(n) = \theta(n^{\log_b a} \log n)$$

(3) If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ and if

$a * f\left(\frac{n}{b}\right) <= c * f(n)$ for some constant $c < 1$ & all sufficiently large n, then

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

Simplified way

$$T(n) = a\,T\left(\frac{n}{b}\right) + \theta(n^k \log^p n)$$

$a \geq 1$, $b > 1$, $k \geq 0$, & $p \in R$

(1) If $a > b^k \rightarrow T(n) = \theta(n^{\log_b a})$

(2) If $a = b^k$

(a) if $p > -1$, $T(n) = \theta(n^{\log_b a} \log^{p+1} n)$

(b) if $p = -1$, $T(n) = \theta(n^{\log_b a} \log \log n)$

(c) if $p < -1$, $T(n) = \theta(n^{\log_b a})$

(3) If $a < b^k$

(a) if $p \geq 0$, $T(n) = \theta(n^k \log^p n)$

(b) if $p < 0$, $T(n) = O(n^k)$

Case 1 $T(n) = 4T(\frac{n}{2}) + n$

$a = 4, \quad b = 2, \quad f(n) = n$

if $f(n) = O(n^{\log_b a - \epsilon})$

$f(n) = O(n^{2-\epsilon})$ if $\epsilon > 0$

then $T(n) = \theta(n^{\log_b a})$

$T(n) = \theta(n^{\log_2 4}) = \theta(n^{\log_2 2^2})$

$T(n) = \theta(n^{2.1})$

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

Case 2 $T(n) = 4T(\frac{n}{2}) + n^2$

$a = 4, \quad b = 2, \quad f(n) = n^2$

If $f(n) = \theta(n^{\log_b a})$

$f(n) = \theta(n^2 - \epsilon)$ when $\epsilon = 0$.

then $T(n) = \theta(n^{\log_b a} \log n)$

$= \theta(n^{\log_2 4} \log n)$

$T(n) = \theta(n^2 \log n)$

if we try case 1

$f(n) = O(n^{\log_b a - \epsilon})$

$= O(n^{\log_2 4 - \epsilon})$

$= O(n^{2-\epsilon})$

but $f(n) = n^2$

∴ it is possible when $\epsilon = 0$

thus case 2 is applicable

Case 3 $T(n) = 4T(\frac{n}{2}) + n^3$

$f(n) = n^3$ $\quad a = 4, \quad b = 2.$

$f(n) = \Omega(n^{\log_2 4 + \epsilon})$

$f(n) = \Omega(n^{2+\epsilon})$ when $\epsilon > 0$

∴ $T(n) = \theta f(n)$

$= \theta(n^3)$

Q/ Use the M.M to determine Asymptotic bounds for the following recurrences.

1. $T(n) = 4T(n/2) + n$
2. $T(n) = 4T(n/2) + n^2$
3. $T(n) = 4T(n/2) + n^3$

Soln

1) $T(n) = \theta(n^{\log_2 4})$

$= \theta(n^2)$

$a = 4, b = 2$

$f(n) = n$

$f(n) = O(n^{\log_2 4 - \varepsilon})$

$= \theta(n^{2-1})$ at $\varepsilon = 1$

$= O(n)$

$\therefore T(n) = \theta(n^2)$ $\because \varepsilon > 0$

2) $\therefore T(n) = \theta(n^{\log_b a} \log^{p+1} n)$

$= \theta(n^{\log_2 4} \log n)$

$= \theta(n^2 \log n)$

$a = 4, b = 2$

$a = b^2$

$f(n) = n^2$.

$f(n) = \theta(n^{\log_2 4 - \varepsilon})$

$= \theta(n^{2-\varepsilon})$

$\varepsilon > 0$.

$\therefore T(n) = \theta(n^{\log_2 4} \log n)$

3) $T(n) = \theta(n^k \log^p n)$

$= \theta(n^3 \log^0 n)$

$= \theta(n^3)$

$a = 4, b = 2$

$k = 3$

$a < b^k$

$n^{\log_b a} = n^{\log_2 4}$

$= n^2$

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 & with the min possible total edge weight.

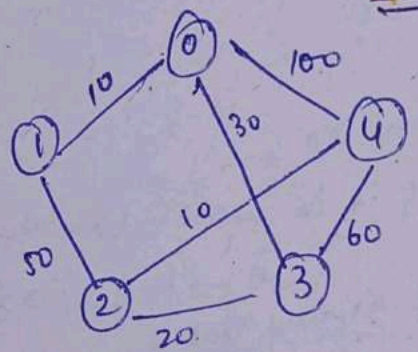
Such trees can be found with algorithms such as Prim's or Kruskal's Algorithm

Dijkstra's Algorithm.

Single source shortest path (single source multiple destination)

⇒ finding shortest paths b/w nodes in a graph from the source node.

Eg 1.



Initial configuration

	0	1	2	3	4
0	∞	10	∞	30	100
1	10	∞	50	∞	∞
2	∞	50	∞	20	10
3	30	∞	20	∞	60
4	100	∞	10	60	∞

We maintain two arrays for every step.

Initially {

Distance array.

0	1	2	3	4
0	10	∞	30	100

Visited nodes Array

0	1	2	3	4
1	0	0	0	0

1st Iteration

(1) Select vertex 0

(2) re adjust the distances of vertex

2nd Iteration

(1) select vertex 1 (as it is at min distance from 0)
(no need to go at node 0 again)

→ cost of visting to Vertex 2 (V_2) from source Vertex V_0 via V_1

$= 10 + 50 = 60$

→ cost of visiting V_3 from V_0
(it is worst case to follow $0 \to 1 \to 2 \to 3$)

✓ no so need to update the array value

→ cost of going to V_4 =
(again it is worst case to follow $0 \to 1 \to 2 \to 3 \to 4$)

✓ so no need to update the existing value..

∴ Dist Array =

0	1	2	3	4
0	10	60	30	100

Visiting Array

0	1	2	3	4
1	1	0	0	0

3rd iteration

(2) Select vertex (3) V_3

→ cost of going to V_2 via V_3 from source Vertex V_0
$= 30 + 20 = 50$. ie < existing value
thus update the array.

→ cost of visiting V_4 from V_0 via V_3

= 30+60 = 90 ie < existing value

thus update the array.

Now the array is

	0	1	2	3	4
Dist	0	10	50	30	90

4th Iteration

Select vertex (2)

cost of going to V_4 from V_0 via V_2

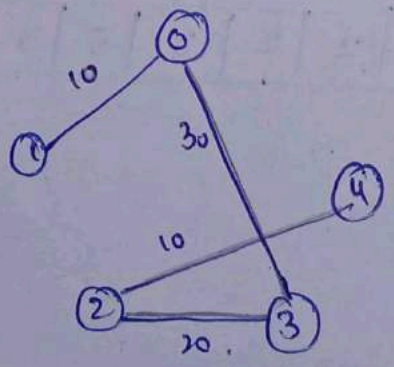
= 10+50+10 = 70 ie < 90

thus Arrays are

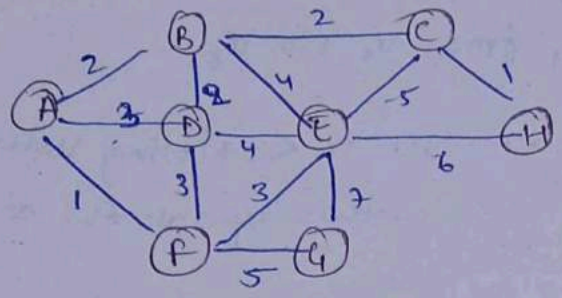
	0	1	2	3	4
Dist →	0	10	50	30	70

⇐ final min dist Array.

	0	1	2	3	4
Visited array →	1	1	1	1	0



Eg 2

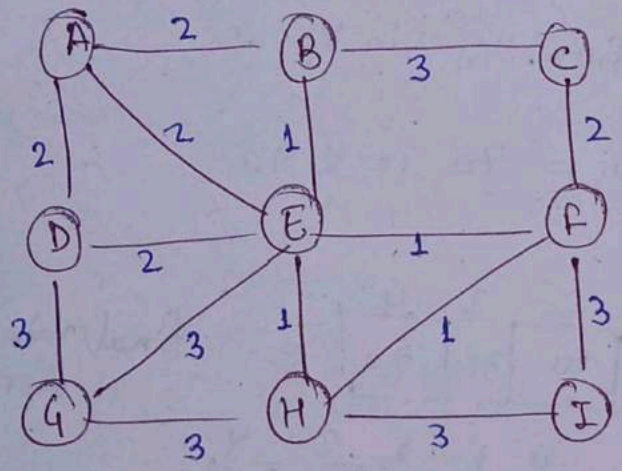


(Apply same algorithm and you will get the final array like this)

taking source vertex as A

A	B	C	D	E	F	G	H
0	2	4	3	4	1	6	5

Eg

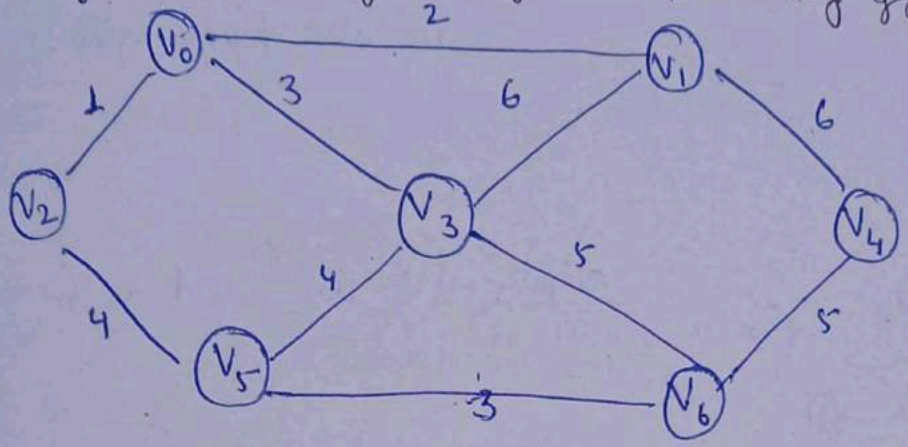


taking A as source node

A	B	C	D	E	F	G	H	I
0	2	5	2	2	3	5	3	6

Kruskal's Algorithm

Q Using Kruskal's Algorithm, find a minimum cost spanning tree of the following graph:-



STEP 1 :- List the edges with path cost:

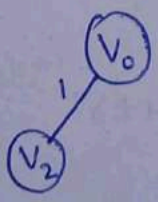
V_0V_1	V_0V_2	V_0V_3	V_1V_3	V_1V_4	V_2V_5	V_3V_5	V_3V_6	V_4V_6	V_5V_6
2	1	3	6	6	4	4	5	5	3

Step 2 Sort the edges in ascending order of their weight.

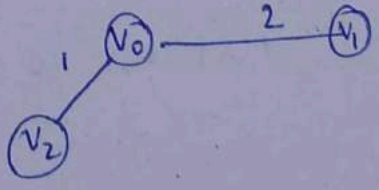
1	2	3	3	4	4	5	5	6	6
V_0V_2	V_0V_1	V_0V_3	V_5V_6	V_2V_5	V_3V_5	V_3V_6	V_4V_6	V_1V_3	V_1V_4

Step 3 Edges are added in order of weight to the spanning trees. If an edge forms a cycle, it is discarded.

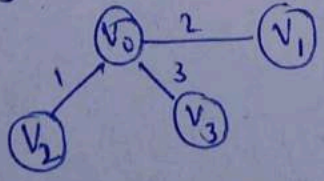
Step 4 Add V_0V_2



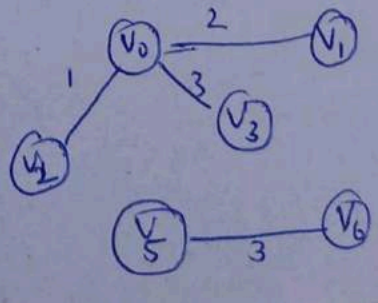
Step 5 Add V_0V_1



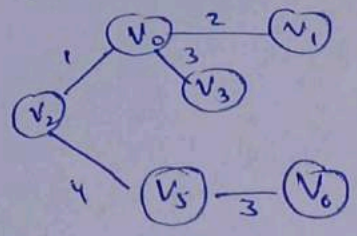
Step 6 Add V_0V_3



Step 7 Add V_5V_6

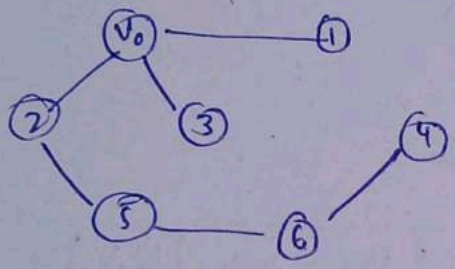


Step 8 Add V_2V_5



Step 9 V_3V_5 is discarded as it creates cycle. V_3V_6 is discarded as it also forms cycle.

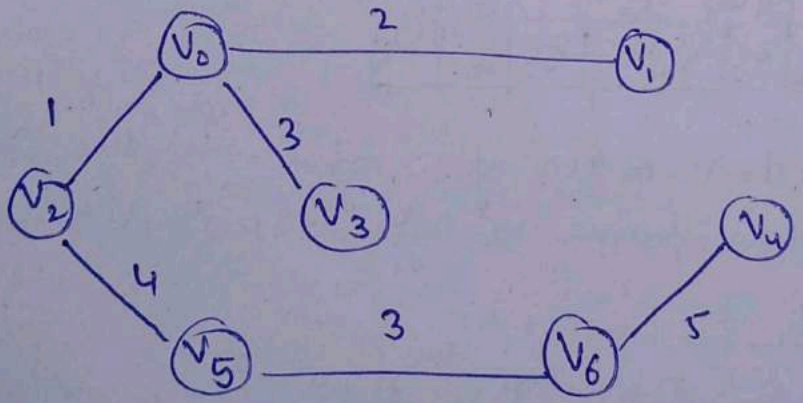
Step 10 Add V_4V_6



Step 11. V_1V_3 discarded as it creates cycle.

Step 12 V_1V_4 is discarded as it creates cycle.

∴ Minimal spanning tree is :-



MST cost is = 1+2+3+3+4+5

= (18)

Prim's Algorithm

(It is not as much effective as Kruskal's)

(greedy Algorithm)

that finds a minimum spanning tree for a weighted undirected graph.

→ Here we have to visit all the nodes arbitrarily.

① First let's take ①

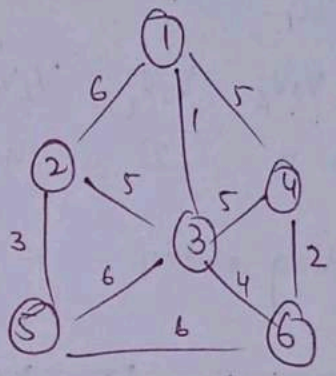
min path for (3) among the adjacent vertex.

from (3)

min path = for (6)

from (6)

min path for (4)



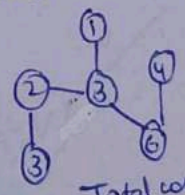
from (4)

Adj vertices are ①, ③, ⑥. but now, there will be a cycle get of created thus we can't move ahead. (as tree has no cycles).

Thus go 1 step back.

At (6) ⟶ or go to (3) then to (2) ie double back track.
↳ go to (5)

in this case tree will be



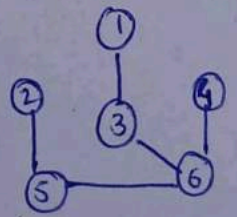
Total cost = 1+4+2 +5+3 ⇒ (15) //

from (5)

Adj vertices = ②, ③, ⑥

reach (2)

∴ Resultant tree will be like

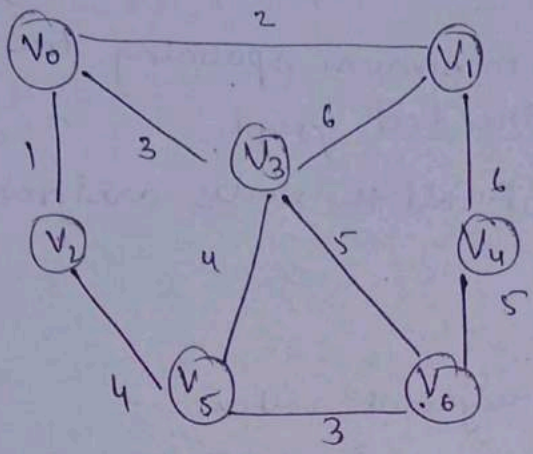


1+4+2+6+3 = 16

min cost = 16 //

Similarly take one by one all the vertexes as source node & among them whichever will be least, that will be our final MST.

Eg

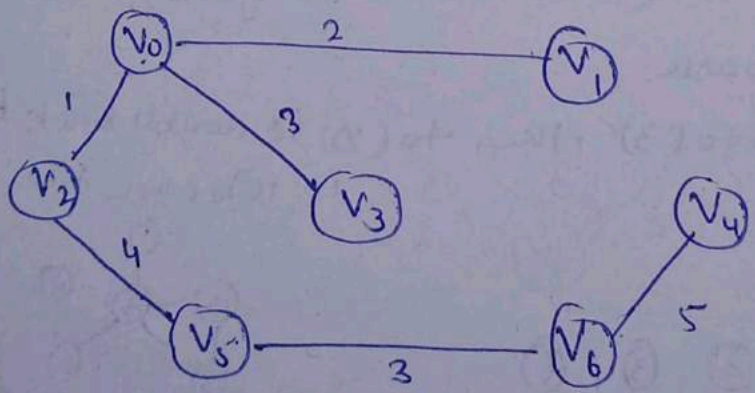


V_0V_2	V_0V_3	V_0V_1	V_2V_5	V_1V_3	V_1V_4	V_3V_5	V_3V_6	V_5V_6	V_6V_4
1	3	2	4	6	6	4	5	3	5

In Ascending order

V_0V_2	V_0V_1	V_0V_3	V_5V_6	V_2V_5	V_3V_5	V_3V_6	V_6V_4	V_1V_3	V_1V_4
1	2	3	3	4	4	5	5	6	6
					X	X		X	X

Now edges are added in order of the weights to the spanning tree.



~~*~~ is the final tree.

with min cost-

1+2+3+4+3+5

⇒ 18 //

Binary Search Tree

Left key < root < Right-key

(Huffman coding is an application)
based on variable code length.

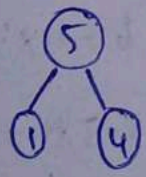
Example.
Draw a huffman tree for the following symbols whose frequency of occurrence in a message is given as follows -

A	B	C	D	E	F	G	H	I
15	6	7	12	25	4	6	1	15

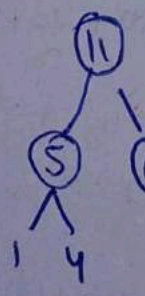
Step 1 Sort them in ascending order

H	F	B	G	C	D	A	I	E
1	4	6	6	7	12	15	15	25

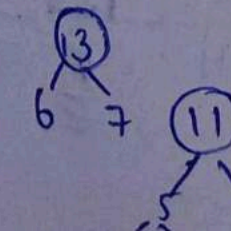
Step 2 Merge the min weight node.
and repeat the same process.

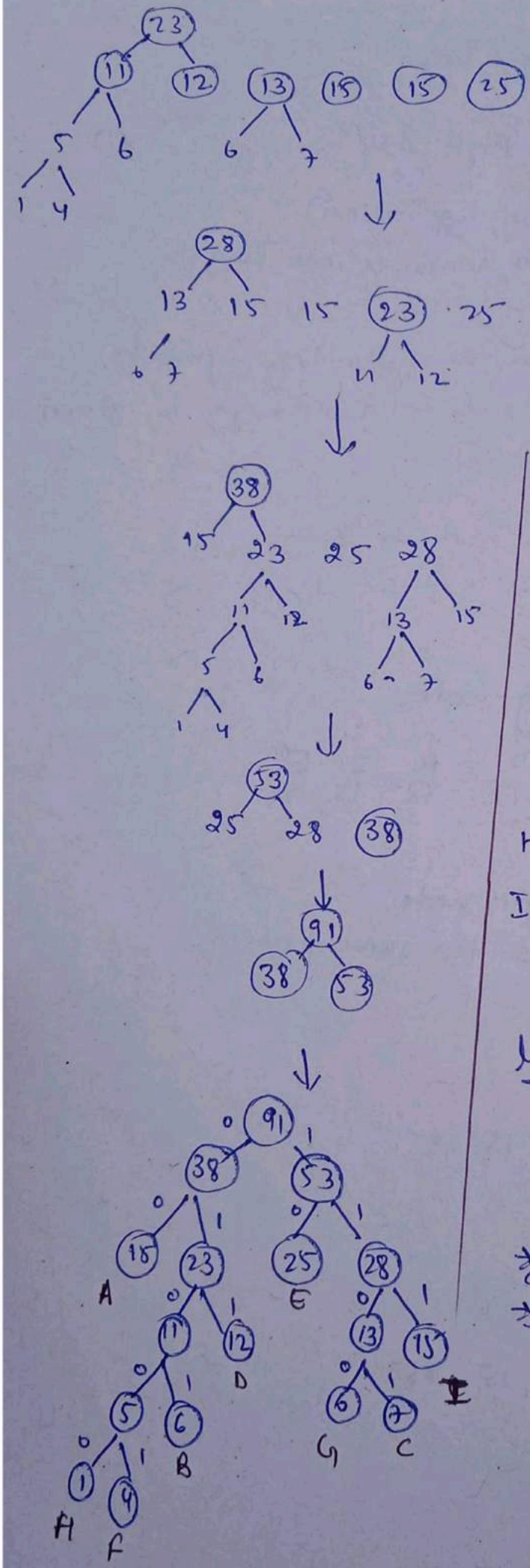


6 6 7 12 15 15 25



6 7 12 15 15 25





Variable code length

A = 00
B = 0101
C = 1101
D = 011
E = 10
F = 01001
G = 1100
H = 01000
I = 111

~~2x CODE~~

length - $2\times15 + 4\times6 + 4\times7 + 3\times12 + 2\times25 + 5\times4 + 4\times6 + 5\times1 + 3\times15$

⇒ 30 + 24 + 28 + 36 + 50 + 20 + 24 + 15 + 15

⇒ 262.

Decoding the code

1 1 1 01000 10 111 011

I H E I D

Approach ← Sequential → In which some consecutive steps of inst. are executed in a chronological order to solve a problem.
Divide & Conquer → Merge sort, Quicksort, Binary search, Bellman Ford's Algo, LCS
Greedy → Kruskal's algo, Prim's algo, Huffman Code, Dijkstra's Algo (for constructing min spanning trees)

GREEDY ALGORITHM

also known as single source shortest path

(1) They are short sighted in their approach that they take decisions on the basis of information in and without worrying about the effect, these decisions may have in the future.

(2) They are easy to invent, easy to implement & most of the time efficient.

(3) They are used to store optimization problems.

Optimization (in our favour)

→ to maximize or to minimize as per our goal (need).

↑ constraints ↑ objective func^n.

decided ~~the~~, other than

G.A has two parts
- Optimal Sub-structure
- Greedy choice property.

(It starts with a locally optimized choice & continue it till the goal achieved.)

① Optimal sub-structure → A Global optimal solution can be obtained by making a locally optimal greedy choice.

We have 2 sets

Set 1 : contains chosen items.

Set 2 : contains rejected items.

(2) Greedy choice property → It starts with a locally optimal choice & continues making locally optimal choice until solution is found.

A G.A consists of 4 functions to find optimal soln.

1) A function that checks whether chosen set of items provide a solution.

2) A function that checks feasibility of the sets.

3) Selection function that tells which of the candidates is most promising (favourable ones)

4) An objective function which doesn't appear explicitly, gives the value of the soln.

Feasibility :- A feasible set is promising if it can be extended to produce not merely a solution but an optimal soln to the problem.

(follows on particular instance that takes to optimality)

Optimality :- (it is for whole scenario, ie whole task)

An optimal solution is the solution that either minimises or maximizes the given constraints for the objective function. (source → destination cost checks)

Knapsack

Knapsack problems can be stated as follows :-

1) Suppose there are n objects from $i = 1, 2, \ldots n$

2) each object i has some weight w_i and profit value v_i associated with it.

3) It carry at most weight- w.

Greedy Approach

Goal ⇒ (1) choose only those objects that have maximum profit

(2) Total weight of chosen object ≤ w (knapsack weight)

Mathematically

we can obtain set of feasible solution ie

maximise $\sum_{i=1}^{n} p_i x_i$

(chosing objects from 1 to)

Constraints / subjected :- $\sum_{1 \le i \le n} w_i x_i \le w$

and knapsack can carry the fraction x_i of an object such that $0 \le x_i \le 1$ &

$1 \le i \le n$ (no. of items available)

There are 3 greedy strategies :-

(Methods to apply G.A)

① Max profit value → arrange in decreasing values.

(ie sort the objects in descending order and pick them in that order)

② Min weight → arrange in increasing order.

(ie sort in ascending order & pick with lowest)

③ Max [$\frac{\text{profit}}{\text{weight}}$] → sort the objects in decreasing order & pick the max.

Eg:

$w = 50$

$n = 3$

$(P_1, P_2, P_3) = (60, 100, 120)$

$(w_1, w_2, w_3) = (10, 20, 30)$

item	P_i (profit value)	w_i (weight)	Ratio (p_i/w_i)
1	60	10	6
2	100	20	5
3	120	30	4

On the basis of profit value

$I_2 + I_3 = 100 + 120 = 220$.

(as their respective weight gives $20 + 30 = 50 = w$)

On the basis of weight- criteria.

~~$3I_1 + I_2 = 180 + 100 = 280$.~~

$60 + 100 + \underbrace{4 \times 20}_{\text{taking partial fraction of 120. (20 part of } \frac{120}{30}\text{)}} = 160 + 80 = 240$. //

so that $\sum w_i \leq w$

On the basis of Ratio

$60 + 100 + \underbrace{4 \times 20}_{\text{partial fraction}} = 240$ //

Ques

$W = 60$

$n = 4.$

$(A, B, C, D) = (280, 100, 120, 120)$

$(w_1, w_2, w_3, w_4) = (40, 10, 20, 24)$

(if repetition allowed)

Item	Profit	weight	Ratio
A	280	40	7
B	100	10	10
C	120	20	6
D	120	24	5

On the basis of Profit-

$280 + 120 = 280 + 120 = 400$

$w. \quad (40) + 2(10) = 60 = w.$

On the basis of weight-

$w = 2(10) + 2(20) \Rightarrow p = 200 + 240 = 440$ //

On the basis of profit/weight Ratio

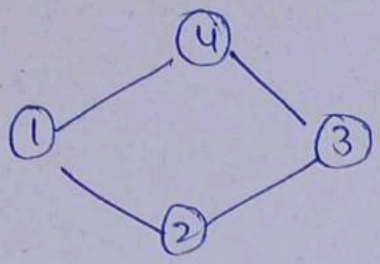
$2B + 1A = 280 + 200 = 480$ //

Max profit in Ratio basis.

Algorithm (Apply for all the 3 methods)

1. $i \leftarrow 1$ to n (item selection)
2. do $x[i] \leftarrow 0$
3. weight $\leftarrow 0$ (current weight get initialized)
4. weight $\leq w$
5. do $i \leftarrow$ best remaining item.
6. if weight + $w[i] \leq w$
7. then $x[i] = 1$
8. weight $\leftarrow$ weight + $w[i]$
9. else
10. $x[i] \leftarrow (w - \text{weight}) / w[i]$
11. weight $\leftarrow w$
12. return x

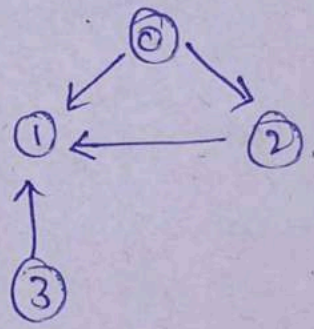
Adjacency Matrix of graph (Undirected)



Spanning

	1	2	3	4
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	1	0	1	0

Directed graph



	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	0	1	0	0
3	0	1	0	0

In weighted graphs, instead of 1 & 0 we place their values.

* Each individual node is sub-graph (part of graph)
* Degree of vertex = for ⓪ = 2.
  - indegree = 0 (incoming edge sum)
  - outdegree = 2 (outgoing edge sum)

Degree of graph = degree of the vertex having largest degree.
(in this case of node ①)

Q Find the optimal solution for Knapsack problem where $n=7$, $m=15$

$(P_1, P_2, P_3, P_4, P_5, P_6, P_7) = (10, 5, 15, 7, 6, 18, 3)$

$(w_1, w_2, w_3, w_4, w_5, w_6, w_7) = (2, 3, 5, 7, 1, 4, 1)$

item	P_i	w_i	P_i/w_i
1	10	2	5
2	5	3	1.67
3	15	5	3
4	7	7	1
5	6	1	6
6	18	4	4.5
7	3	1	3.

On the basis of P_i (profit value)

$18 + 15 + 10 + 4 \times (\frac{7}{7}) = 18+15+10+4 = 47$

$w \rightarrow (4 + 5 + 2 + \frac{7}{7} \times 4 = 15)$

On the basis of w_i

$3 + 6 + 10 + 5 + 18 + \frac{15}{5} \times 4 = \frac{3+6+10+5+18+12}{42} = 54$

$w \rightarrow (1 + 1 + 2 + 3 + 4 + \frac{5}{5} \times 4)$

On the basis of P_i/w_i (ie Ratio basis)

$6 + 10 + 18 + 15 + 3 + \frac{5}{3} \times 2 = 34 + 18 + 3.34$

$w \rightarrow (1 + 2 + 4 + 5 + 1 + 1.67 \times 2) = 52 + 3.34 = 55.34$ //

$\therefore$ Max profit in Ratio basis.

4/10/21

Activity Selection Problem

It states that given a set of n activities, with their start and finish times, we need to select maximum number of non-conflicting activities that can be performed by a single person given that the person can handle only one activity at a time.

Eg.

A	A_1	A_2	A_3	A_4	A_5	A_6
S	5	1	3	0	5	8
F	9	2	4	6	7	9

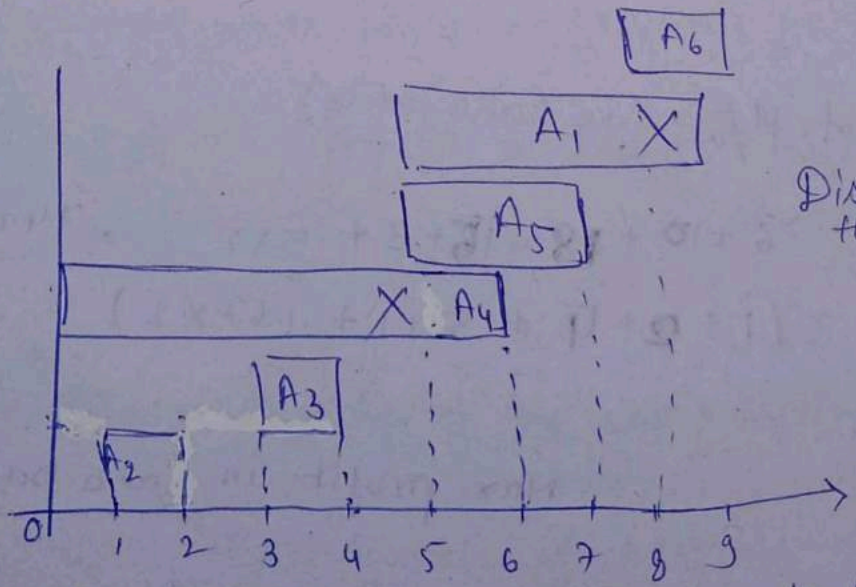
Condition.

$A_i \leq f_j$
or
$S_j \geq f_i$

Step 1
Sort according to finish time in ascending order.

	A_2	A_3	A_4	A_5	A_1	A_6
f:	2	4	6	7	9	9
s:	1	3	0	5	5	8

Manage a time chart



Discarded on the basis of overlapping.

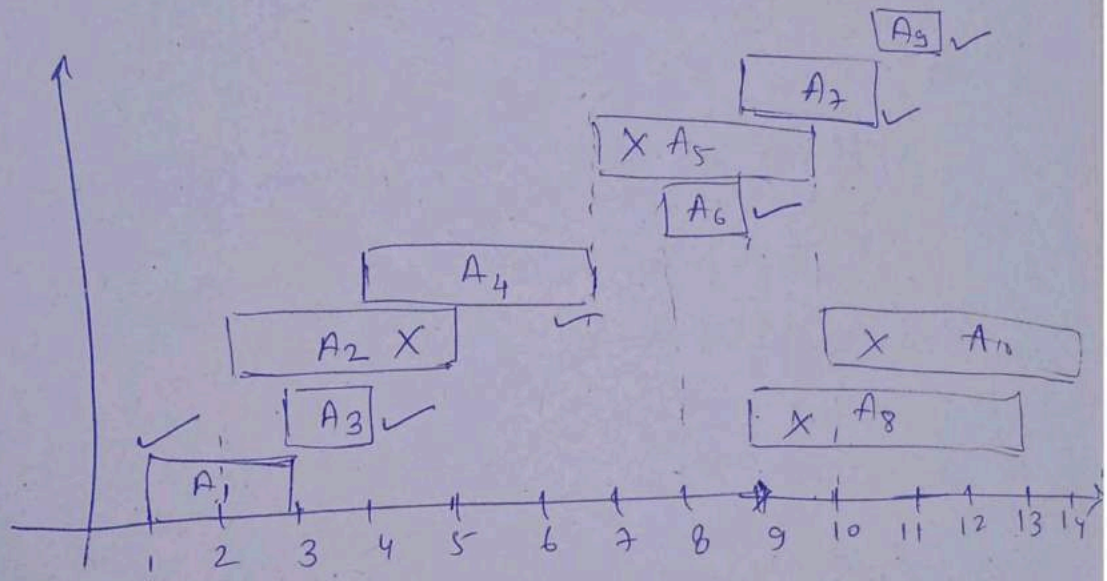
The set of activities to be performed on the basis of greedy approach are A_2 A_3 A_5 A_6.

Que.

A	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}
s_i	1	2	3	4	7	8	9	9	11	10
f_i	3	5	4	7	10	9	11	13	12	14

Sorting

	A_1	A_3	A_2	A_4	A_6	A_5	A_7	A_9	A_8	A_{10}
s_i	1	3	2	4	8	7	9	11	9	10
f_i	3	4	5	7	9	10	11	12	13	14



Possible activities are :-

A_1 A_3 A_4 A_6 & A_7 A_9

Dijkistra's Algo with Greedy ~~Algorithm~~ Approach.

(Single source shortest path)

Initialize single source (G, S)
↓ Goal ↘ source.

Algorithm

1. for each vertex V ∈ set of vertex ∈[G]
   i.e V ∈ V[G]

2. do

   2.1 d[V] ← ∞ // distance from source to that vertex

   2.2 Π[V] ← nil

   2.3 d[S] ← 0.

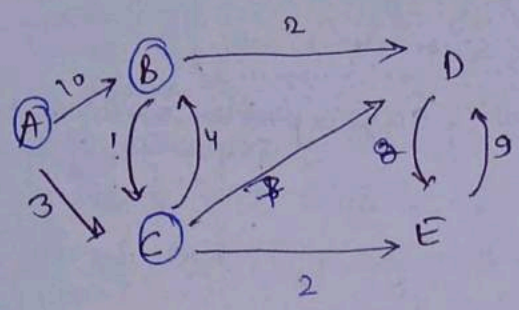
Relax (u, v, ω)

1. if (d[u] + ω(u,v) < d[v])

2. then d[v] ← d[u] + ω(u,v)

3. Π[v] ← u

Relax func for feasibility func.
(it tells how can we minimize the cost)



from A.

A	B	C	D	E
0	∞	∞	∞	∞
Dist.	10	3	-	-
	7	-	11	5

from C

reaching B 3+4 = 7 thus replace.

reaching D = 3+7 = 11 "

" E = 3+2 = 5. "

A	B	C	D	E
0	7	3	11	5

∴

from E

reaching D = 9+2+3 = 14 no need.

{A, C, E}

Dynamic Problemming used problems —

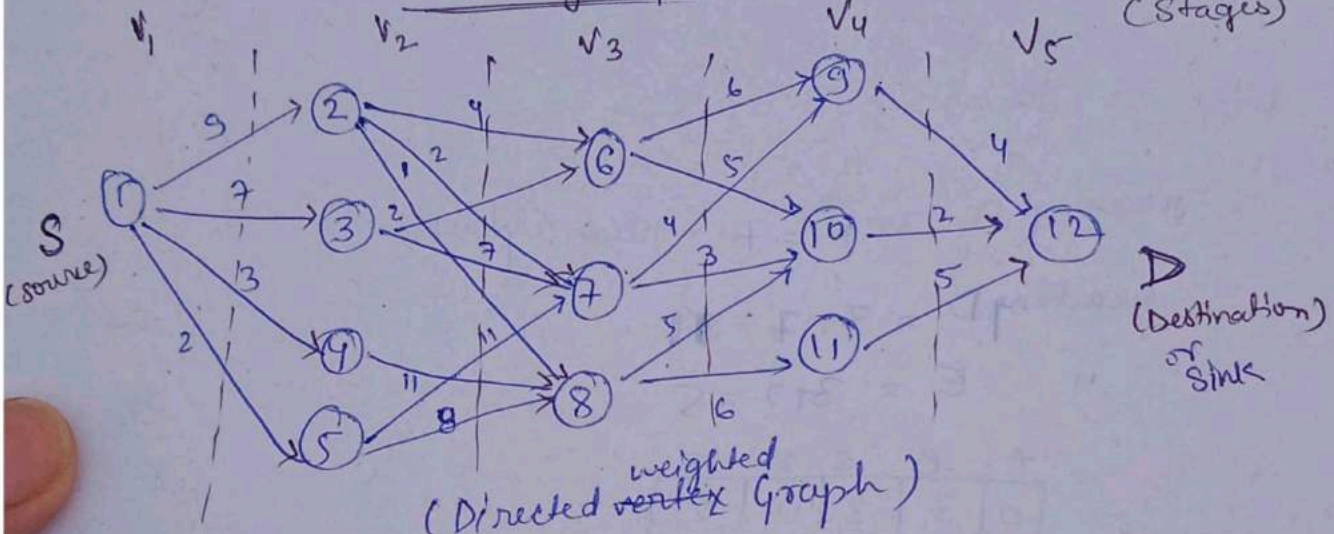
- 0/1 knapsack
- Matrix chain multiplication
- Edit- distance.
- Bellman-Ford Algo for single source shortest path.
- Floyd Warshall Algo for all pairs shortest path.

Dynamic Programming < Optimal Sub-structure. / Overlapping sub-structure. → It uses memoization technique.

DP works on principle of optimality & optimality says a problem must be solved in sequence of decision.

Sub-problem overlapping → if you have done that problem earlier then you need not to do again the same.

Multi-Stage Graph (for larger values set)



- 1st step → divided into sub-programs ($V_1 - V_5$)
- principle of optimality works on cost minimisation of path.

V	1	2	3	4	5	6	7	8	9	10	11	12
cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2/3	7	6	8	8	10	10	10	12	12	12	12

cost (Stage, vertex)
↓ (V_1, V_2, V_3, V_4, V_5 or 1,2,3,4,5)
→ (1-12)

due to 2 → cost is 7
due to 3 → cost is 9
thus final one is (2)

cost this is bcz of which vertex (node) (Approaching node)

- Start from Destination. (Bottom-up Approach)

cost (V_5 , 12) = 0 (as no work is done here till then)

cost (4, 9) = 4

cost (4, 10) = 2

cost (4, 11) = 5

c = weight of an edge.

cost(3, 6) = min (C(6,9) + cost(4, 9), C(6,10) + cost(4,10))

= 6+4, 5+2 ← min cost from 6 to 12

cost (3, 7)

cost (3, 8)

↳ min (C(7,9) + cost(4,9), C(7,10) + cost(4,10))

4+4, 3+2 ← min cost from 7 to 12 due to 10

min (C(8,10) + cost(4,10), C(8,11) + cost(4,11))

min (5+2, 6+5)

= 7 due to node 6

cost(2, 2) = min { C(2, 6) + cost(3,6), C(2,7) + cost(3,7) + C(2,8) + cost(3,8) }.

= min (4+7, 2+5, 1+7) = min (11, 7, 8)

cost (2,3) = min (C(3,6) + cost(3,6), C(3,7) + cost(3,7) +

= min (2+7, 7+5) = min (9, 12) = 9 due to 6

cost (2, 4) = min (C(4,8) + cost(3,8)) = 11+7 = 18.

cost (2,5) = min (C(5,7) + cost(3,7), C(5,8) + cost(3,8))

= min (11+5, 8+7)

= min (16, 15)

= 15 due to 8

cost(1,1) = min (C(1,2) + cost(2,2), C(1,3) + cost(2,3), C(1,4) + cost(2,4), C(1,5) + cost(2,5))

= min (9+7, 7+9, 3+18, 2+15) // don't have to recalculate

= min (16, 16, 23, 17)

= 16

formula for forward method

$$cost(i, j) = \min \{ c(j, l) + cost(i+1, l) \}$$

$\langle j, l \rangle \in \{E\}$

$l \in V_{i+1}$ (ie next stage)

i → stage, j → vertex, E → Edge, l = some vertex ~~in the i+1~~ (longest)

| d(stage, vertex) |

d(1,1) = 2

d(2,2) = 7

d(3,7) = 10 (node)

d(4,10) = 12

min path 1 → 2 → 7 → 10 → 12

shortest path.

Dynamic Programming Solution ✳

Longest Common Subsequent (LCS)

suppose we have two strings (no overlapping)

never come back for matching.

ie no intersecting.

S1 : A B C B D A B

S2 : B D C A B A

∴ LCS = BDAB

OR

S1 : A B C B D A B

S2 : B D C A B A

LCS = BCAB

OR

many matching strings like AA, BB, CAB

11/11/21

S1: a b d a c e
S2: b a b c e

LCS: bace

OR

LCS: a b c e

S1: a b c d e f g h i
S2: c d g i

LCS: cdgi

if S2: e cdgi

LCS: cdgi

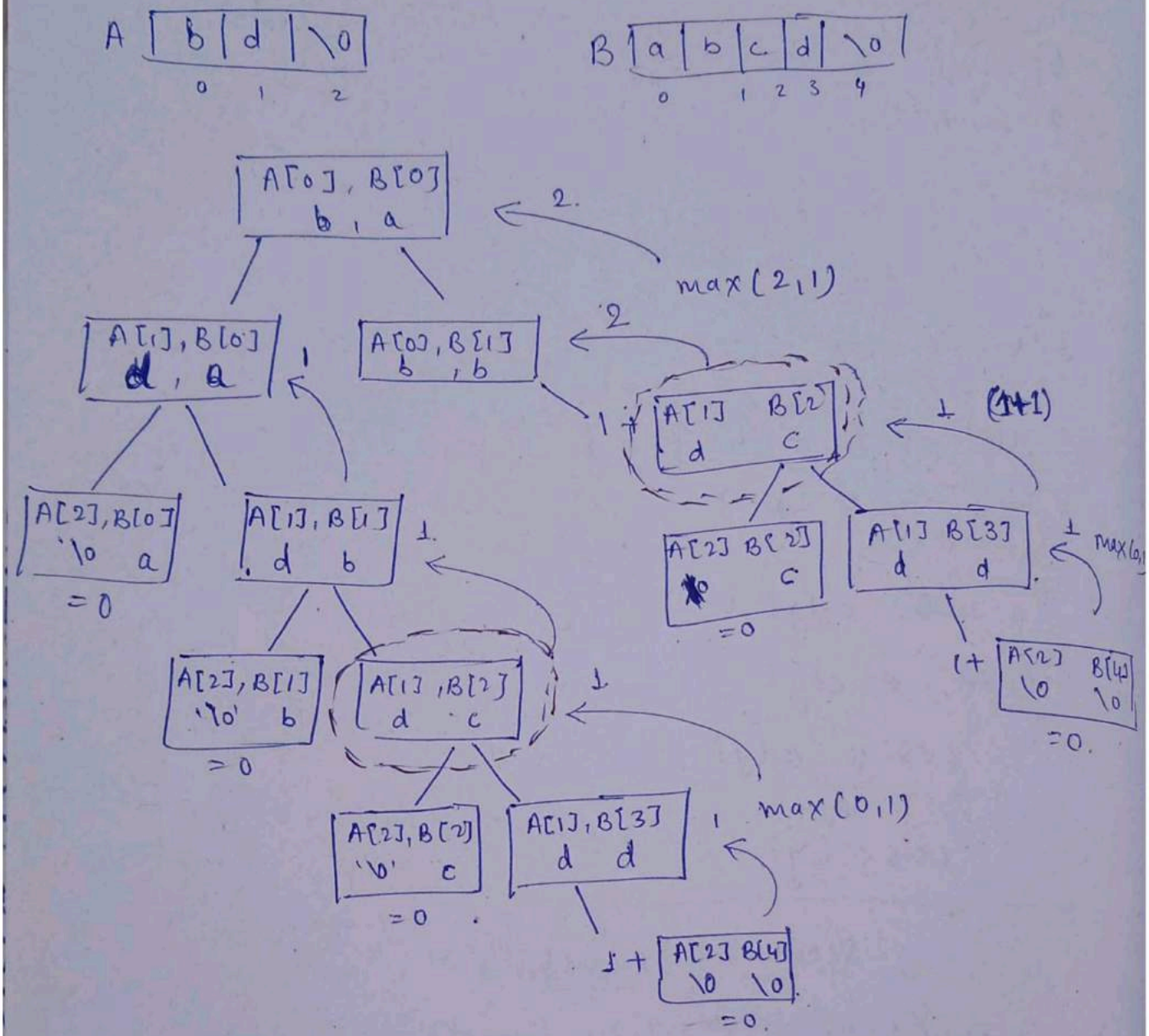
OR

LCS: egi

Recursive function

```
int LCS (i, j)
{ if (A[i] == '\0' || B[i] == '\0')
      return 0;
  else if (A[i] == B[j])
      return (1 + LCS(i+1, j+1));
  else
      return max (LCS(i+1, j), LCS(i, j+1));
```

Time complexity → 2^m



$\therefore$ 1+0 = 1.

$\therefore$ Length of LCS = 2.

Memoization

(Stores the previous solved problems)

Once we have entered the data we don't need to solve again, ie overlapping.

1st String	a	b	2nd string c	d	\0
	0	1	2	3	4
b 0	2	2			
d 1	1	1	(1)	1	
\0 2	0	0	0		0

15

Algo of D.P

if (A[i] == B[j])

LCS [i,j] = 1 + LCS [i-1, j-1]

else

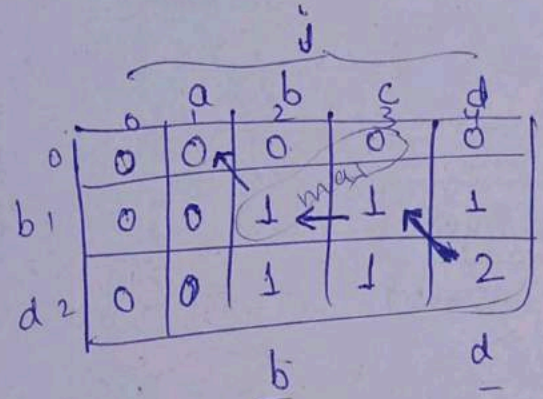
LCS[i,j] = max (LCS [i-1, j], LCS [i, j-1])

Sample

A = | b | d |

B = | a | b | c | d |

Table



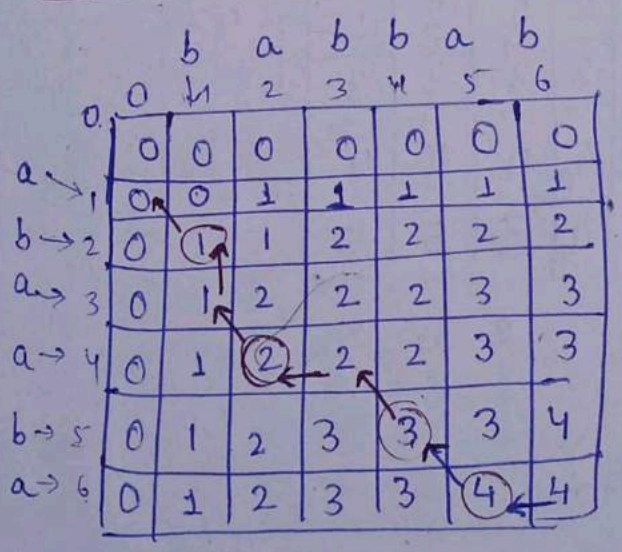
(0,0) for initialization with some value like 0.

bd

Ex

x = | a | b | a | a | b | a |

y = | b | a | b | b | a | b |



from bottom

b a b a

∴ LCS (longest common subsequence using D.P.)

= baba

Eg $x = [c\,o\,w]$ $y = [b\,r\,o\,w\,n]$

		b	r	o	w	n
	0	0	1	2	3	4	5
	0	0	0	0	0	0	0
c	1	0	0	0	0	0	0
o	2	0	0	0	1	1	1
w	3	0	0	0	1	2	2

o w

LCS = ow

Eg

$x = [stone]$ $y = [strongest]$

		s	t	r	o	n	g	e	s	t
	0	0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1	1	1	1
t	2	0	1	2	2	2	2	2	2	2	2
o	3	0	1	2	2	3	3	3	3	3	3
n	4	0	1	2	2	3	4	4	4	4	4
e	5	0	1	2	2	3	4	4	5	5	5

s t o n e

LCS = stone

$V[4,8] = \max\{V[3,8], V[3,8-5]+6\}$

$= \max[7, V[3,3]+6]$

$= \max[7, 2+6]$

$= \max[7, 8]$

$= 8$

0/1 (Dynamic) Knapsack problem not fraction as in case of greedy approach.

0 = not included
1 = included.

(0/1) Knapsack Problem using DP

$$V[i,w] = \max\{ V(i-1], w), \; V[i-1, w-w[i]] + P[i] \}$$

Conditions $\max\{ p_i x_i$

$$\left\{ \sum w_i x_i \leq w \right.$$

Eg $w = 8$ (limit) max capacity. $n = 4$ (no. of items)

$x = \{1, 2, 3, 4\}$ (item no.)

$w = \{2, 3, 4, 5\}$ (weight)

$P = \{1, 2, 5, 6\}$ (profit)

possible ways for combination is 2^4 i.e 2^n.

$O(2^n)$

p_i	w_i	x_i	w (weight) → 0	1	2	3	4	5	6	7	8
			0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
	5	4	0	0	1	2	5	6	7	7	8

profit comp. max

include this one bcz it is not present in the above row.

highest profit. due to item no. 4 & 2.

& since it is not present in the previous row thus it is sure that it is due to 4th item.

now 8 − 6 = 2

check for this profit

here 2 is present but 2 is also present in above row thus it is not due to 3rd item hence don't include x_3.

x_1 x_2 x_3 x_4

{ 0 1 0 1 }.

can't choose as 2 & 4 already chosen and now it will exceed w.

can't chose as $w_3 + w_4 > 8$.

$$V[i,w] = \max\{ V[i-1, w], V[i-1, w-w[i]] + P[i] \}$$

~~max [V[4-1, 5], V[3]]~~

$V[4,1] = \max[V[3,1], V[3, 1-5] + 6]$

$= 0$.

$[3, -4]$ X not possible

$V[4,5] = \max[V[3,5], V[3, 5-5] + 6]$

$\max[5, 0+6]$

We are having $W = 25$, $n = 4$

$x = \{A, B, C, D\}$

$w = \{24, 10, 10, 7\}$

$P = \{24, 18, 18, 10\}$

P_i	w_i	x_i
24	24	A
18	10	B
18	10	C
10	7	D

weight → / x_i	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	24	24
B	0	0	0	0	0	0	18	18	18	18	18	18	18	18	18	18	18	18	18	24	24
C	0	0	0	0	0	0	18	18	18	18	18	18	18	18	18	18	18	18	18	24	24
D	0	0	0	10	10	10	18	18	18	18	18	18	18	18	18	18	18	18	18	24	24

A B C D

{ 1 0 0 0 }

OR 3D ⇒ $w = 21$, $P = 30$

we are having $W = 10$, $n = 4$

$x = \{A, B, C, D\}$

$w = \{4, 5, 3, 2\}$

$P = \{24, 18, 18, 10\}$

P_i	w_i	x_i
24	4	A
18	5	B
18	3	C
10	2	D

weight → / x_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	24	24	24	24	24	24	24
B	0	0	0	0	24	18	24	24	24	42	42
C	0	0	0	18	24	24	24	42	36	42	42
D	0	0	10	18	24	28	34	28	36	42	42

A B C D

{ 1 0 0 0 }

P_i	w_i	x_i
24	4	A
18	5	B
18	3	C
10	2	D

weight → / x_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	24	24	24	24	24	24	24
B	0	0	0	0	24	18	24	24	24	42	42
C	0	0	0	18	24	24	24	42	42	52	46
D	0		10	18	24	24	24	42	42	52	52

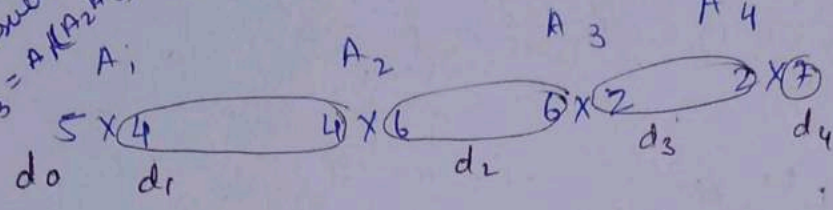
bcz of A, C

As a programmer we would focus on reducing the cost of multiplication.

Matrix multiplication is associative so parenthesization doesn't change result.
$(A_1 A_2) A_3 = A_1 (A_2 A_3)$

Matrix Chain Multiplication

Time complexity
$= n^2 \times n = n^3$
$O(n^3)$.

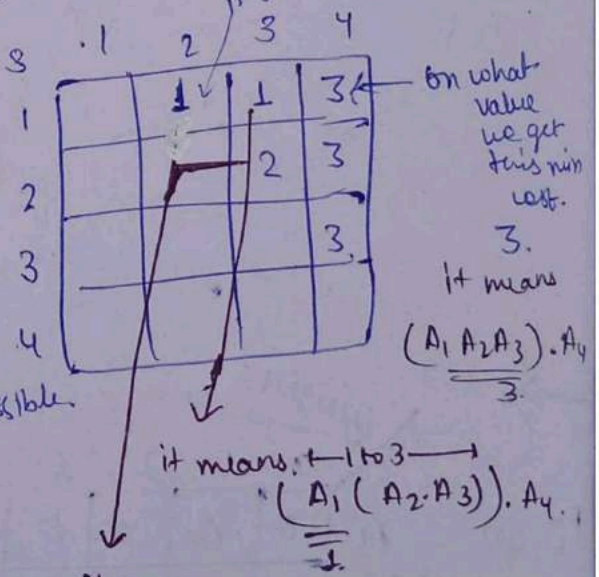


Ways
- $((A_1 \times A_2) \times A_3) A_4$ ✓
- $(A_1 \times A_2) \times (A_3 \times A_4)$
- $A_1 (A_2 \times A_3) \times A_4$
- $A_1 (A_2 \cdot (A_3 \times A_4))$

$M(1,1) \rightarrow A_1$ ie no multiplication with anything. ∴ cost = 0.

	1	2	3	4
1	0	120	88	158
2	·	0	48	104
3			0	84
4				0

Lower A will always have nothings bcz no multiplication possible.



(1)
$m[1,2] = m[1,1] + m[2,2] + 5 \times 4 \times 6$
$A_1 \cdot A_2$
$5 \times 4 \quad 4 \times 6$
$5 \times 6 \times 4$
$= 120$

$m(3,4)$:
$A_3 \cdot A_4$
$= 6 \times 7 \times 2$
$= 84$

$m[2,3]$
$A_2 \cdot A_3$
$= 4 \times 6 \times 2$
$= 48$

(2)
$m[1,3] \leftarrow \min \begin{cases} m[1,1] + m[2,3] + d_0 d_1 d_3 \\ m[1,2] + m[3,3] + d_0 d_2 d_3 \end{cases}$

$(A_1 (A_2 \cdot A_3))$ | $(A_1 \cdot A_2) \cdot A_3$.

$4 \times 6 \cdot 6 \times 2$ | $5 \times 4 \quad 4 \times 6 \quad 6 \times 2$

$m[1,1] + m[2,3] + 5 \times 4 \times 2$ | $m[1,2] + m[3,3] + 5 \times 6 \times 2$

Total no. of products.
$0 + 48 + 40$
$= 88$ ✓ min.

$120 + 0 + 60$
$= 180$

$m[2,4]$ → Range $= \min_{2 \le k < 4} \{ m[2,k] + m[k+1,4] + d_1 d_k d_4 \}$

$\min \begin{cases} m[2,2] + m[3,4] + d_1 d_2 d_4 \\ m[2,3] + m[4,4] + d_1 d_3 d_4 \end{cases}$

$A_2 \times (A_3 \times A_4)$ | $(A_2 \times A_3) A_4$

$4\times6 \quad 6\times2 \quad 2\times7$ | $4\times6 \quad 6\times2 \quad 2\times7$

$m[2,2] + m[3,4] + 4\times6\times7$ | $m[2,3] + m[4,4] + 4\times2\times7$

$= 0 + 84 + 168$ | $= 48 + 0 + 56$

$= 252$ | $= 104$ ✓ min

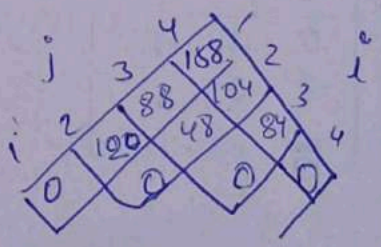
$m[1,4]$ → Range

$$\min \begin{cases} m[1,1] + m[2,4] + 5\times4\times7 \\ m[1,2] + m[3,4] + 5\times6\times7 \\ m[1,3] + m[4,4] + 5\times2\times7 \end{cases}$$

$(A_1 \cdot A_2 \cdot A_3) \cdot A_4$

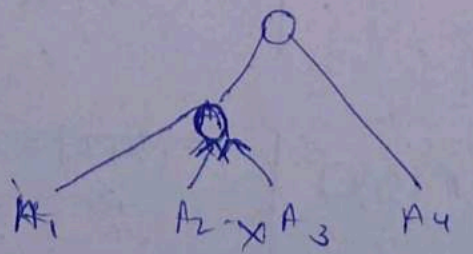
$$\min \begin{cases} 0 + 104 + 140 \\ 120 + 84 + 210 \\ 88 + 0 + 70 \end{cases}$$

$$\min \begin{cases} 244 \\ 414 \\ 158 \end{cases} = 158$$



$(A_1 \cdot A_2 \cdot A_3) A_4$.

$(A_1 \cdot (A_2 \cdot A_3)) \cdot A_4$ bcz $1 \to 3$ we get min value due to 1



formula

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + d_{i-1} \times d_k \times d_j \} & i < j \end{cases}$$

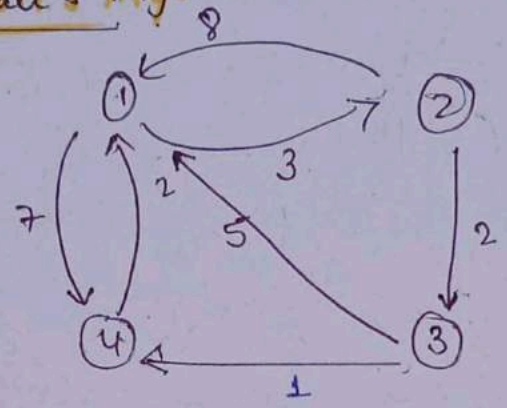
for some k, $A_{i \cdots j} = A_{i \cdots k} A_{k+1 \cdots j}$

$m[i,j] = m[i,k] + m[k+1,j] + d_{i-1} d_k d_j$ $\quad i \le k < j$

Dijkstra's Algorithm using D.P

All pairs Shortest Path (D.P) $O(n^3)$

Floyd Warshall's Algo



1 ← 2, 3, 4 ; 2 ← 1, 3, 4 ; 3 ← 1, 2, 4 ; 4 ← 1, 2, 3

$$A_0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & \infty \\ 3 & 5 & \infty & 0 & 1 \\ 4 & 2 & \infty & \infty & 0 \end{array}$$

self loop → 0
no paths exist → ∞

$$A_1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & \infty & 7 \\ 2 & 8 & 0 & 2 & 15 \\ 3 & 5 & 8 & 0 & 1 \\ 4 & 2 & 5 & \infty & 0 \end{array}$$

maintaing corresponding values for 1. via (1).

$A^0[2,3] = A^0[2,1] + A^0[1,3]$

$\min(2 \lessgtr 8 + \infty = \infty)$

$2 < \infty$

$= 2.$

$A^0[2,4] = A^0[2,1] + A^0[1,4]$

$\infty = 8 + 7 = 15$

$\min(\infty > 15)$

$= 15$

$A^0[3,2] = A^0[3,1] + A^0[1,2]$

$\infty = 5 + 3 = 8.$

$\min(\infty > 8)$

$\therefore 8.$

$A^0[3,4] = A^0[3,1] + A^0[1,4]$

$1 = 5 + 7$

$1 = 12$

$\min(1, 12)$

$= 1$

~~$A^0[4,2] = A^0[4,1] + A^0[1,2]$~~

~~$\infty = 2 + 3 = 5.$~~

~~$\min(\infty, 5) = 5.$~~

$A^0[4,3] = A^0[4,1] + A^0[1,3]$

$= 2, \infty$

$= \infty$

for A^2 =

	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	5	8	0	1
4	2	5	7	0

making constant row 2 & colum 2 (via 2 vertex)

$A[1,3] = A'[1,2] + A'[2,3]$

$\infty > 3 + 2$

$\min(\infty > 5)$

$= 5.$

$A[1,4] = A'[1,2] + A'[2,4]$

$7 = 3 + 15$

$\min(7 < 15)$

$= 7$

$A[3,1] = A'[3,2] + A'[2,1]$

$5 = 2 + 8$

$5 = 10.$

$= 5$

$A[3,4] = A'[3,2] + A'[2,4]$

$1 < 8 + 15$

$= 1.$

$A[4,3] = A[4,2] + A[2,3]$

$\infty = 5 + 2$

$\infty = 7$

$\min(7)$

$= 7.$

via (3)

	1	2	3	4
1	0	3	5	6
2	7	0	2	3
3	5	8	0	1
4	2	5	7	0

$A[1,4] = A^2[1,3] + A^2[3,4]$

$7 = 5 + 1$

$7 > 6.$

$A[2,1] = A[2,3] + A[3,1]$

$8 = 2 + 5$

$8 > 7.$

$A[2,4] = A[2,3] + A[3,4]$

$15 = 2 + 1$

$= 3.$

$A[4,1] = A[4,3] + A[3,1]$

$2 = 3 + 5$

$= 2$

$A[4,2] = A[4,3] + A[3,2]$

$5 = 7 + 8.$

$= 5.$

A^3 via c

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	9	7	0	1
d	6	16	9	0

$A^3[a,b] = A[a,c] + A[c,b] = 3 + 7$

$= 10$

$A[a,d] = A[a,c] + A[c,d]$

$= 3 + 1$

$= 4$

$\therefore 4$

$A[b,a] = [b,c] + [c,a]$

$= 5 + 9$

$A[b,d] = [b,c] + [c,d]$

$= 5 + 1 = 6$

~~$A[c,a] = 9$~~

$A[d,a] = A[d,c] + A[c,a] = 9 + 9$

$6 < 18$

$A[d,b] = A[d,c] + A[c,b] = 9 + 7$

$\infty > 16$

~~AE~~

A^4 =

via d

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	7	7	0	1
d	6	16	9	0

$A[a,b] = A[a,d] + [a,b] = 4 + 16$

$10 < 16$

A:

$A[c,a] = A[c,d] + A[d,a]$

$9 = 3 + 6$

7

$\therefore 7 < 9$