→ Root of all modern language is ALGOL.
(Algorithmic language).

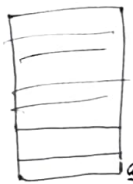
DSA Lect. 1

8 bit → 1 byte

→ computer science
→ Data & information
→ Memory → word addressable
→ byte addressable
→ 32 bit vs 64 bit ?
→ 1 word = 2^n byte

$n = 0, 1, 2, 3, \ldots$

↳ min. 2 byte



byte addressable memory

→ int a;
a = 3

why it takes 4 bytes?

→ Data type
→ How data type works?

int, char, float, double, bool

→ **Data structure** :-

- logical and mathematical model of a particular organisation of data.

using data types

- operation ÷ Traversing, searching, Insertion, deletion,
  - some special case ÷ sorting, merging

- composite data structure
  ↳ Made up of simple data structure.

* **Array** ÷
  ↳ simplest form : one dimension Array
  ↳ A finite ordered set of homogenous element.

Array of data structure

Allocating data structure ↳ Extraction & storing.

e.g. : int a[10];

Array size = Upper bound − lower bound + 1

13	14	15	16	17	18	19	20	21	22
0	1	2	3	4	5	6	7	8	9

Array [10]

a[0] → base address

index ↳ base add. + (i) × size of data

a[0]	13
a[1]	14
:	
a[9]	22

→ memory

→ Application of Array ÷ Real life use of

- Access information efficiently.
- signal Processing : Recognition, image processing
- Multimedia Application : RGB value of image
- Data mining : Represent large database
- Robotics
- Scientific computing : data processing

→ upper bound ÷ A[0]

→ lower bound ÷ A[n-1]

2D Array ÷ → Application of 2-d array.

int a[3];

int b[3][4];

array of array

Row

column

	C_0	C_1	C_2	C_3
R_0				
R_1			x	
R_2				
R_3				

int b[1][2];

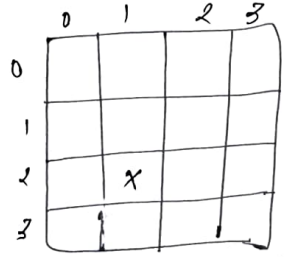
How address ?

find it

Sheet-2

int a[4][3];

find the address of a[2][1], if is the base address.



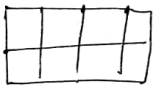
size of one element is esize.

(2 × 3 × esize + 1 × esize) + BA

. a[i][j]

base address + i × n2 × esize + j × esize

Q. int a[3][2][4], BA = 0, esize = 1
then the address of a[2][1][1]



sol: 2 × 2 × 4 + 1 × 4 + 1
= 21.

Q. int a[i][j][k]
address of a[i][j][k].

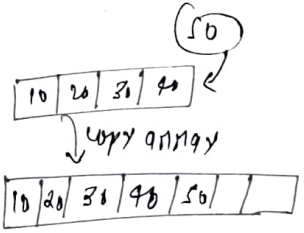
sol:

i × n2 × n3 × esize + j × n3 × esize + k × esize + BA.

n dimentional array a[n1][n2]...[nn]
BA, esize, address of a[i1][i2]...[in]

↳

* Dynamic Array :
↳ Automatic system
↳ cannot optimise



Q. 1) An array of length n stores n number of consecutive integers. You have to find sum of n element.
You need to tell how many operation you perform.

sol:# No. of operation = 1.

sum = $\frac{n}{2}$ (First element + last element).

sum = $\frac{n}{2}(2a+n-1)$

last_element = First_element + n - 1

sum = $\frac{n \times (\text{First\_element} + \text{last\_element})}{2}$

Q.2) You have an array of length n integers are store in the ascending order. You have to find the element having maximum value.

↳ T.C = O(1), No. of operation = 1.

Q.3) You have an array of integer type integers.

you have to find an element/number which is greater than atleast two no. in the array. All the elements in the array is unique. Total no. of operations?

sol: pick 3 elements & compare

sol.1) → Input n
→ sum calculation
→ operation count
→ output

sol.2) • Since the array is sorted in ascending order, the maximum element is the last element in the array.
• Access the last element directly using its index (n-1). where n is the length of the array.

sol.3)

* Stack ÷ Structurally & logical view

↳ LIFO - Last in first out.

↳ Stack : ordered collection of items.

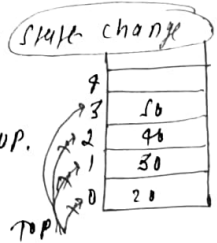
↳ insertion & deletion at one end called TOP.

↳ Stack is dynamic → state change

↳ Operation : push & pop.

insert delete

one pointer

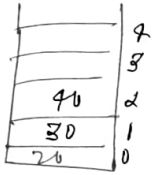


→ TOP = 1. → pointer

n = a [TOP]

n = 40

TOP = TOP - 1



→ push() ÷ O(1)

PUSH(S, n, TOP)

{

if (TOP == n-1)

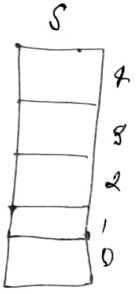
Printf ("stack is full");

else

TOP = TOP + 1

S [TOP] = x

}



TOP = -1.

→ POP() ÷ O(1)

POP(S, TOP)

{ if (TOP == -1)

Printf ("stack is empty")

else

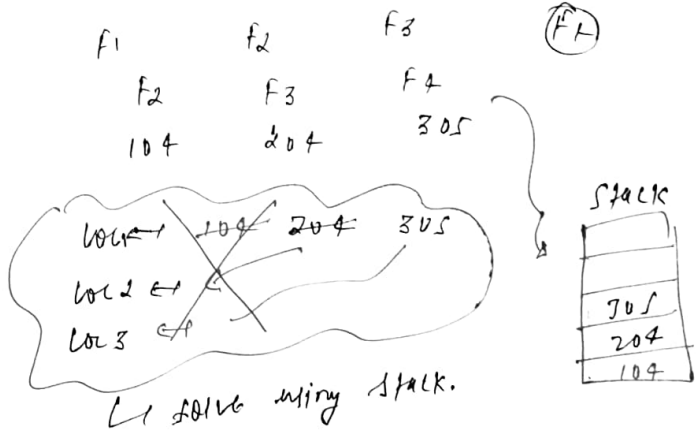
x = a[TOP]

TOP = TOP - 1

}

subroutine ? → Essential operation that allow to manage the stack efficiently ↳ push, pop, peek.

→ Repeated calls to this stack to use.



↳ solve using stack.

Sheet-3

Q.

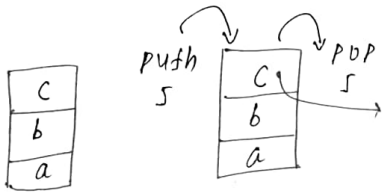
Push
Pop
] 5 unit of time.

Time interval between two stack operations: 4 unit of time

Life time of a stack element: Amount of time for which the element resides on stack.

Average life time of stack = $\frac{\sum_{i=1}^{n} \text{life time of } i}{n}$

Sol:



Push(b) | Push(c) | POP(c) | POP(b)

4 + 5 + 4 + 5 + 4

= 22.

Push(a) | Push(b) | Push(c) | POP(c) | POP(b) | POP(a)

4 5 4 5 4 5 4

= 4.

Q. In a stack push and pop operation take x unit of time and time interval b/w two operation is y units. we are going to perform n number of push operation and n number of pop operation what will be the life time of stack.

Sol:

N(x+y) - x

=

* Application of stack :-

i) Function call and recursion.

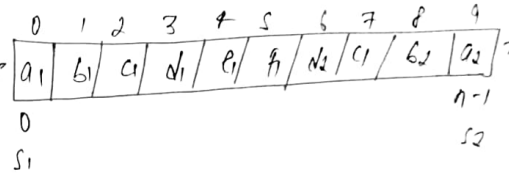
ii) Scope delimeter we nested correctly.

while (there is character in the string)
  symb = read next symbol
  if (symb = '(' || symb = '[' || symb = '{')
    push (S, symb)
  if (symb = = ')' || symb = ']' || symb = '}')
    stack is empty or not
    i = pop(S)
c, {, [→ opening → push

* scope delimeter
() →
{} →
[] →

Q. An array of N length is given. How to implement two element in the stack.

Sol:



we implement 2 stack
↳ to do this we start from both end for better utilization

* Stack is full → TOP2 - TOP1 = 1
↳ condition to check weather stack is full.

if TOP2 - TOP1 = 1
then stack is full.

* Postfix Evaluation –
↳ Mathematical Expression :-

Prefix	Infix	Postfix
+ab	a+b	ab+
x ab + cd	a x b + c x d	ab x + cd x
x + ab x cd		ab x cd +

Q. A $ B * C - D + E / F / (G + H)

find prefix and postfix.

$ → Expn. exponentiation

Sol:

Prefix → + - * $.

Postfix →

Q. A + B + C

Pre ÷ ++ABC

post ÷ AB+C+

Q. A $ B $ C

Pre ÷ $$ABC

Post ÷ A$(B$C)

Program?
- Pre
- In
- Post

(A+B) * C ——→ AB+C*

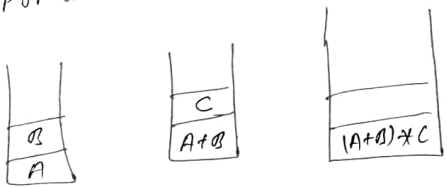
POP1 = B

POP2 = A

operator ÷ POP

operand ÷ Push

POP 2 operation POP1



Lect. 4

* Queue

↳ Two pointers ↳ FIFO (FCFS)

↳ An ordered collection of items.

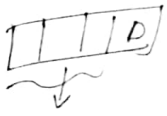
↳ Items may be inserted at one end called rear.

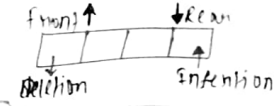
↳ Items may be deleted (removed) at one end called front.

↳ Two operations:

i) Enqueue : To insert

ii) Dequeue : To delete





→ Enqueue:

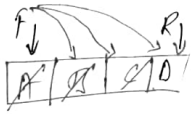
```
Enqueue (Q, N, F, R, X)
{ if (R == N-1)
  { printf ("Queue is full");
    exit (1);
  }
  if (F == -1) → if (R == F == -1)
      F = R = 0
  else R++
    Q[R] = X ;
}
```

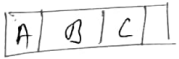
(R) F/R

→ Dequeue :

```
Dequeue (Q, N, F, R)
{
    if (F==-1)
    {
        printf("Queue is empty");
        exit(1);
    }
    y = Q[F];
    if (F==R)
        F=R=-1.
    else
        F++
    return(y);
}
```

F = -1, R = -1





→ If F=-1 then R=-1.

→ If F=R → queue have only one element (atleast one element)

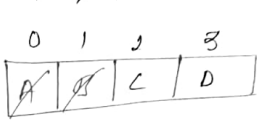
** exit() function in C

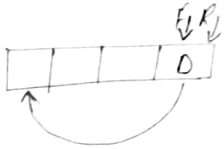
↳ used to terminate the program.
↳ part of the <stdlib.h> library
↳ 0 → indicate successful program termination.
↳ 1 → Non-zero value usually indicate error or abnormal termination

* Circular Queue :

↳ why circular is important in queue.
↳ Better memory utilization.

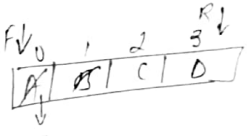
• n = 4 , n-1 = 3





• R = n-1
↳ R = 0

• R+1 % n
↳ R



→ if ((F==0 && R==n-1) || (R+1==F))

→ CQInsert :-

```
CQInsert(Q, N, F, R, x)
{ if (((R == N-1) && (F == 0)) || (R == F-1))
  {
    Printf("Queue is Full");
    exit(1);
  }
  if (R == N-1)
      R = 0
  else
      R++
  Q[R] = x
  if (F == -1)
  {
      F = 0
  }
}
```

→ CQDelete :-

```
CQDelete(Q, N, F, R)
{ if (F == -1)
  {
    Printf("Queue is empty");
    exit(1);
  }
  y = Q[F]
  if (F == R)
      F = R = -1
  else {
      if (F == N-1)
          F = 0
      else F++;
  }
  return(y)
}
```

Priority queue

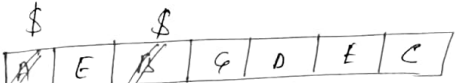
↳ A data structure in which intrinsic ordering of element determines the results.

NOTE :- i) Element can be characters, numerical and complex structure.

ii) Element may be sorted based on the ~~feed~~ field which is not the part of elements.

* Normal queue use priority queue ⇒ FIFO

*



Insertion operation → modify.
Shifting

* Recursion → Implemented by stack.

↳ A function which defines an object in terms of simpler case of itself.

↳ For atleast one arguments or group of arguments recursion function is defint in

Factorial

```
Pmod = 1
for(x=n; x>0; x--)
    Pmod = Pmod * x
return(Pmod)
```

• Non recursive exit for one element

→ n! = 1 if n = 0

→ n! = n*(n-1)! if n > 0

multiplication

a * b = a if b = 1

a * b = a(b-1) + a if b > 1

* Fibonacci sequence : 0, 1, 1, 2, 3, 5, ...
↳ Each time

fib(n) = 1 if n == 0 or n == 1.

fib(n) = fib(n-2) + fib(n-1)

→ fac(n)
↳ stack length = n.

- Drawback of Recursion ÷

↳

- Recursive vs non Recursive → storage

Lec. 5

* **Algorithm** (Algo w/o input → POP, O.S)

A finite set of instruction that specify a sequence of operations to be carried out in order to solve a specific problem or class of problems is called an algorithm.

Characteristics of algorithm ÷

- Input ÷ zero or more than zero.
- Output ÷ At least one
- Definiteness ÷
- Absence of ambiguity
- Effectiveness ÷
- Termination ÷

Algo vs Program
- Active entity
- Passive entity

Algorithm.

↳ A sequence of definite and effective instructions that terminate with the instruction of correct output for the given input for any.

Algorithm Analysis

↳ How long an algorithm is going to be to solve a problem that

↳ To choose b/w algorithm.

↳ cannot focus on execution time → focus on steps
- compiler
- O.S

↳ No. of operations ÷ Exact no. of operations, Each operation x

```
for all 256 characters do
    assign zero to counter
end for
while there are any character do
    get the next character
    increment the counter for this character
end while
```

- Initialization : 256 + 1
- Increment : 256
- condition check : 256 + 1
- Condition check = N+1
- counter increments : N

Initialization = 256 + 1 = 257

Increment = N + 256

counter = N + 258

Total no. of operations = 2N + 771

constant amount of time.

here N is fixed — (N → 100) (N → length) — पर focus करते हैं

↓ function of no. of input

Hardware

To analyse an algo. we care latter inputs

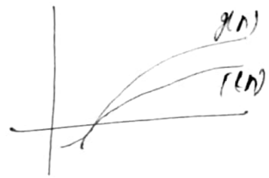
- Running time of algorithm as function of size of its input.
- How fast the function goes with input size

complexity of algorithm

- **Time complexity** : Running time of algorithm as a function of size of its input.
- **Space complexity** : Amount of space used by algorithm as a function of size of its input.

Note:

- g(n) काटे नए f(n) को cross कर सकता है / touch कर सकता है



· **Asymptotic Analysis**

- Asymptotic means a line that tends to curve which may or may not eventually touch the curve.
  It is a line that stays within
- Asymptotic notation is a form and big to write down and talk about 'fastest possible' and 'slowest possible' quality

1) **Big - Oh - Notation** : formal method to express upper bound of an algorithm.

Let f(n) and g(n) are two non-negative function. For f(n) and g(n) if there exists an integer n_0 and a constant $c > 0$ such that

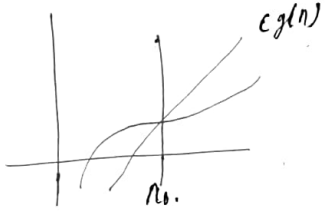
Best case? average
Random No.
Random No. [illegible] condition

for all integer $n \geq n_0$

$0 \leq f(n) \leq cg(n)$

The $f(n)$ is big-oh of $g(n)$

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



Lec. 6

Q.
```
for(i=0; i<n; i++)
    for(j=0; j<n/2; j++)
    {
        - - -
        - - -
    }
```

$f(n) = O(n \cdot \frac{n}{2})$ ✗ constant are ignored

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

Q.
```
fun(int n)
{
    for(i=1; i<n; i++){
        for(j=1; j<n; j++){
            printf("*");
        }
        , brants;
    }
}
```

$T.C = O(n^2)$

Q.
```
wid fun(int n){
    int k = 1
    while (k < n)
    k = 3 * k
}
```

$O(\log_3 n)$

→ since

$k = 3, 9, 27, \ldots$

$3^i > n$

$i > \log_3 n$

Q.
```
fun(int n)
{
    int i, count = 0;
    for(i=1; i*i<n; i++)
        count++;
}
```

$O(\sqrt{n})$

$n = 16$

for: $i^2 < n$

$i^2 > n$ → terminate

$i = \sqrt{n}$

$O(\sqrt{n})$

→ $n = 16$

i =	1	2	3	4	✗	✗	...	n
	1	4	9	16	25			i^2

Q. $f(n) = 3n^2 + 2$

→ $f(n) = O(n^3)$ ✓

→ $f(n) = O(3n^3)$ ✓

→ $f(n) = O(n^3 + 5n^3)$ ✓

→ $f(n) = O(n^2 + 9999)$ ✗

true constant does not matter.

Q. $f(n) = O(n)^m$

$f(n) =$

Q. $2^{n+1} = O(2^n)$ ✓ ÷ $2^n \cdot 2$

$2^{2n+1} = O(2^n)$ ✗ ÷ $2^n \cdot 2^n \cdot 2$

Q. $f(n) = n^2 \log n$, $g(n) = n(\log n)^{10}$

(a) $f(n) = O(g(n))$ ✗ (b) $g(n) = O(f(n))$

(c) $f(n) \neq O(g(n))$ (d) $g(n) \neq O(f(n))$

Ans:

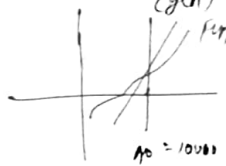
→ $f(n) \neq O(g(n))$ ✓

→ $g(n) = O(f(n))$ ✓

If $\log n^{10}$ is smaller and n^2 raised to other number of $f(n) > g(n)$

Q. $f(n) = n$, $g(n) = n^2$

$f(n) \leq O(g(n))$ and $g(n) \neq O(f(n))$



#

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

↳ n_0 = i → integer greater than zero

Array

- search ← sorted O(n) / unsorted O(n)
- traverse
- insertion
- deletion

Stack

- push() O(1)
- pop() O(1)
- isEmpty O(1)
- isFull O(1)
- size() O(1)
- deleteStack() O(1)

Queue

- enqueue() O(1)
- dequeue() O(1)
- isEmpty() O(1)
- isFull() O(1)
- size() O(1)
- deleteQueue() O(1)

Recursion

↳ space complexity

Time complexity → not $O(n!)$?

* Asymptotic notations [lec. 7]

- Big oh notation (worst)
  ↳ $f(n) \leq c\,g(n)$, $f(n) \geq c\,g(n)$
- Sigma notation (Ω) (best or lower bound)
- Theta notation ÷ $\theta(n)$ ÷ average
  ↳ $c_1 g(n) \leq f(n) \leq c_2 g(n)$

* Sorting

* without sorting, can we study the process of sorting.
  ↳ Time and space
  ↳ cost of T.C.

→ Sorting
  ↳ Internal Vs External
  ↳ Stable Vs unstable
  ↳ Inplace ÷ External memory (extra memory) → space complexity
  ↳ Sorting by address ↳ call by value / call by reference

* Stable sorting ÷

now stable means order should be maintained

→ Example?
→ Application?

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

1. Bubble sort
  ↳ Easy to understand and inefficient in general.
  ↳ In each pass compare two element and arrange them if they are not sorted (in proper order).

10, 7, 5, 8, 4, 9.

No. of comparison $\frac{n(n+1)}{2}$

10 7 5 8 4 9
7 10 5 8 4 9

→ Bubble sort unstable

A[j] < A[i]
a b
b a
after arranging

→ if a[j] > a[i] then it is stable
→ If a[j] ≥ or ≤ a[i] then it is unstable.

* Algo. of bubble sort ÷

```
void bubble (int[], int n)
{
    int hold, j, pass;
    for (pass = 0; pass < n-1; pass++)
    {
        for (j = 0; j < n-pass; j++)
        {
            if (x[j] > x[j+1])
            {
                hold = x[j];
                x[j] = x[j+1];
                x[j+1] = hold;
            }
        }
    }
}
```

↳ • if $x[j] > x[j+1]$ → stable
↳ • if $x[j] \geq x[j+1]$ → unstable
↳ T.C :- $O(n^2)$

$\frac{n}{2} \to n-1$
T.C → $O(n^2)$

* Modified bubble sort :-
↳ best case :- $O(n)$
↳ already sorted

* can bubble sort write in recursive
↳ yes

comp :-
swap :-

2. Selection sort

↳ Largest of remaining elements is place in its proper position.

index =>	0	1	2	3	4	5
	4	7	5	9	1	3

A[index]

A[i] > A[index]

index = 1

4	7	5	3	1	9

comp.	swap
n-1	1
n-2	1
n-3	1 ... n-1

↳ * selection sort is unstable.

* when swapping is costly then selection sort is used.

↳ Algorithm
↳ T.C = $O(n^2)$
↳ unstable

→ Modification is possible in selection sort ?
↳ No.

3. Quick sort

↳ Not stable

→ T.C :- $n \log n$
→ Not - stable
→ Inplace (recursive)

* If element is sorted in asc/desc
→ T.C :- n^2
→ level :- n
* S.C :-

Any sort. for T.C is $n \log n$

$i \leq j$

* Two pointer
↳ can extra space is replace.
yes, using non recursive algorithm.

* Non recursive bort algorithm → ?

Algorithm : Divide & conquer

Quick sort (A, p, r)
if $p < r$ then
q = partition (A, p, r)
quick sort (A, p, q-1)
quick sort (A, q+1, r)

** partial divide and conquer

4 **Insertion sort**

Insertion sort (A) {
for j ← 2 to length (A) {
do key ← A [j]
i = j - 1
while i ≥ 0 and A[i] > key {
do A [i+1] ← A[i]

Lec. 9 | Oct. 8

- Online keyword
- Online vs offline Problem
- Only one online sorting exist i.e. Insertion sort

Time complexity
- $O(n^2)$
- if sorted $O(n)$

- Space minimum requirement, which algo. you used
- Any Algorithm of T.C. in polynomial (n^k, n^2, n^3)

→ Array nearly sorted
→ Inplace ✓
→ stable Relative order

For sorting algorithm we prefer
1. Bubble sort
2. Quick sort (T.C)
3. insertion sort
4. selection sort (swapping)

0. C → capacity, I → item

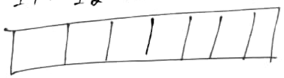
K I_1 I_2 .... I_n

C w_1, w_2 ... w_n

2^n solution

value/unit $\frac{v_1}{w_1}$, $\frac{v_2}{w_2}$ .... $\frac{v_n}{w_n}$ — Not optimal soln. → T.C = $O(n^2)$

I_1, I_2 ..... I_n

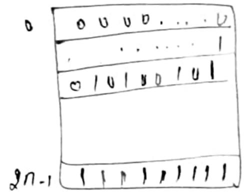


→ value : V [ ] n

weight : w [ ] n

n : no. of item / input size

2^n times / round



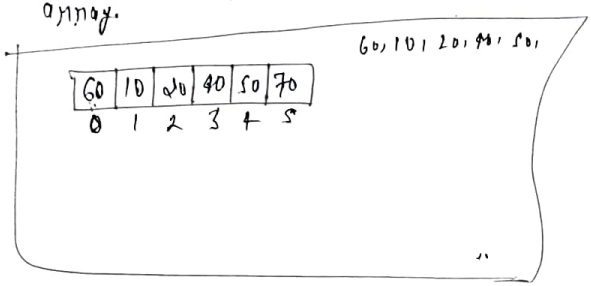
2d array : $2^n \times n$

① ② ③ ④ ⑤
0 1 0 1 0
x ↓ x ↓ x

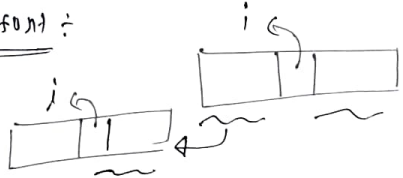
$V_2 + V_4$
$w_2 + w_4$ } $w_2 + w_4 < C$
$X \leftarrow V_2 + V_4$

Q. You have one unsorted array, you have to find kth largest element in this array.

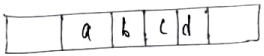
Sol:



↳ Quick sort :



Q. Find whether an array is subset of another array.



* Searching — T.C :- (open :- log n, sorted :- T.C = O(n))

To find an item with specified property in collection of items.

↳ Linear ↳ Binary

- unordered :- (T.C :- O(n)) → No. of operations
- ordered :- (T.C :- O(n)) → No. of operations

* Binary search → in sorted array

```
int binarySearch(int A[], int n, int data){
int low=0, high = n-1;

while(low ≤ high){
  mid = low + (high - low)/2;
  if(A[mid] == data)
      return mid;
  else if(A[mid] < data)
      low = mid+1;
  else high = mid-1;
}
return -1;
}
```

* Pointer

Lec. 10 | Oct. 14

C allows to access location of an object

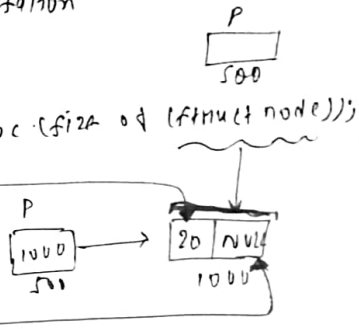
↳ pointer

```
int x = 2
int y ;
int *p;
p = &x;
y = *p;
```

```
int *pi;
pi = (int *) malloc (size of (int));
```

* **Linklist** ⇌ dynamic allocation
⇌ space utilization

```
struct node *p
p = (struct node *) malloc (size of (struct node));
(*p).data = 20;
(*p).link = null;
p → data = 20;
p → link = null;
```



* **structure**

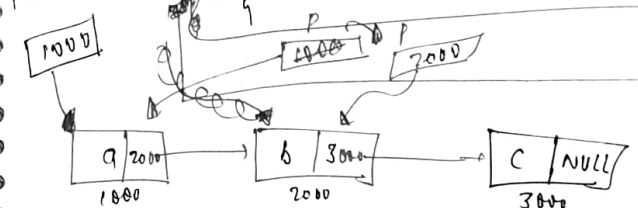
Groups of items in which each items is identified by ill own identifier.

```
struct {
    char f[10];
    char m[10];
    char l[10];
} sname, ename;
```

```
struct nametype {
    char f[10];
    char m[10];
    char l[10];
};
struct nametype, sname, ename;
struct node p1; struct node *p1;
```

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

```
void function (struct node * head)
{
    struct node *p = head
    while (p != NULL)
    {
        printf ("%d", data);
    }
    p = p → link
}
```



Q. write a function to print data of node present at even positions.

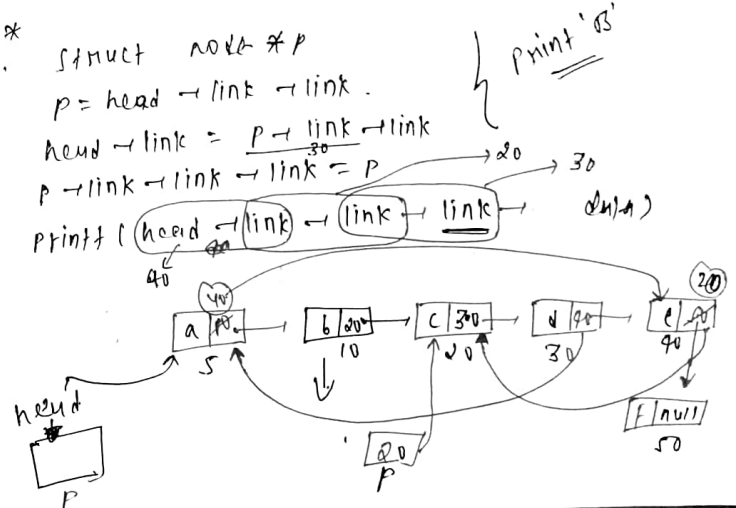
Ans:

(1)
```
void display-even(struct node *head)
{ struct node *p = head;
  while (p != null)
  {
    printf("%d ", p->data);
        p = p->link;
    if (p != null)
       p = p->link;
  }
}
```
This fail when the data/node is not present

(2)
```
wid display-even(struct node *head)
{ struct node*p = head;
  while (p->link != null && p->link->link
                                != null)
    printf("%d", p->data);
```

(2)
```
struct node *p = head;
if (head == null)
{
  exit(1);
}
while (p->link != null &&
```

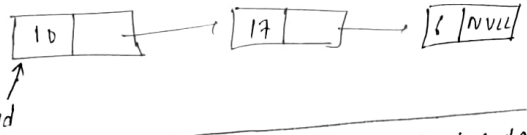
Q.
```
struct node *p
p = head->link->link.
head->link = p->link->link
p->link->link->link = p
printf(
```
print 'B'



Lec-11 Oct-15

Q. write a function to add an element at the end of the linked list.

Ans:




```
struct node *insertAtEnd (struct node *head, int data){
  struct node *head, * cur)
  temp = (struct node *)malloc (sizeof (struct node));
  temp → data = data;
  temp → next = NULL;
  * cur = head;
  if (cur == NULL){
  head = temp;
}
```

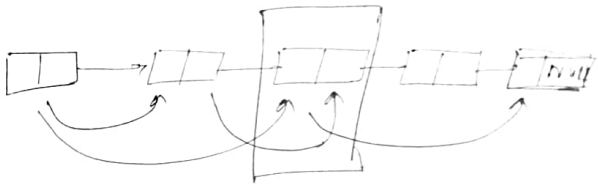
~~Q~~ Benefit of using stack with Array Vs linked list.

↳ stack pop() and push() using linked list. at ~~beginning~~ beginning

```
else{
  while (cur → next != NULL){
     cur = cur → next;
  } cur → next = temp;
  return head;
}
```

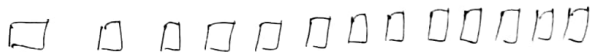
Q. write a function to find the middle element of linked list.

Ans:
- Two pointer
- Loop



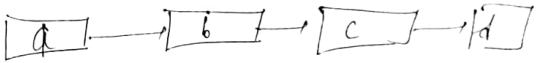
Q. ~~WAP to fin~~ write a function to find a cycle in linked list.

Ans:

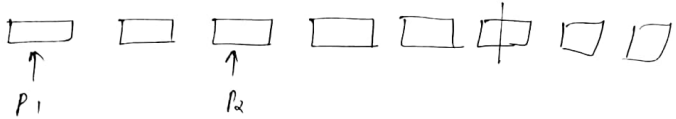


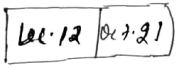
Q. write a function to find nth element from right.

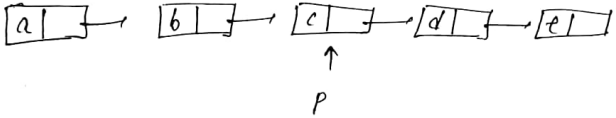
Ans:



Eg: 2nd → c, 3rd → b ; 4th → a, 1th → d





→ 

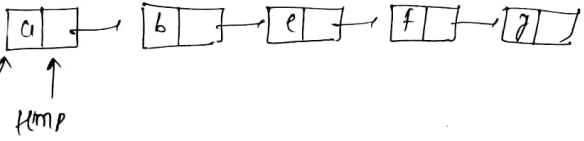
↳ Delete P

Q. Sort on linked list using bubble sort and selection sort.

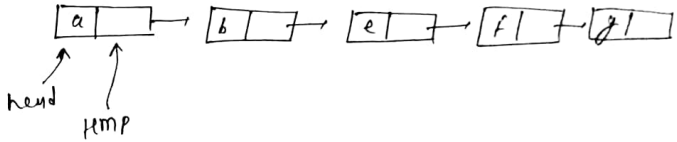
Ans: → Two loop

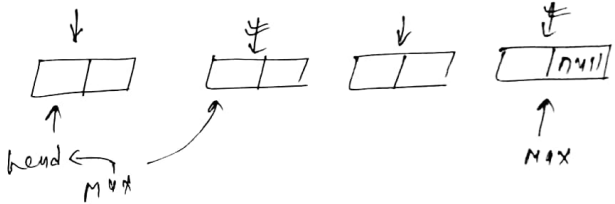
→

Linked list using bubble sort



Linked list using selection sort:

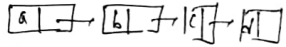




Q. Write a function to print the element of linked list in reverse order.

using stack → POP

↳ inefficient



Make stack using array.

unknown ↳ problem : ↑ ↑ ↑

↳ Array fix

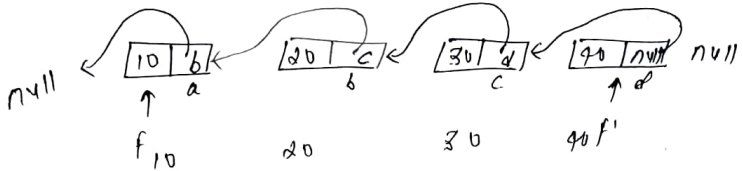
using recursion

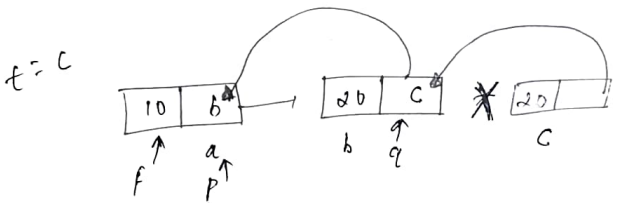
Problem :

recursion is good in this case

↳

Q. Write a function to reverse the links in linked list.





Array ← Static → Memory is fixed
Dynamic

Difference b/w array and linked list

Linked list → Utilisation bell
↳ Static
↳ Queue → two pointer ← front, rear

** How circular queue implemented by linked list
↳ we cannot discuss circular queue in linked list.
• Utilisation ↑ie.

** How to implement ~~two~~ double ended queue using linked list.

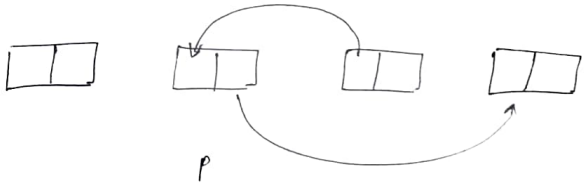
Q. Insertion and deletion using array and linked list.

• Insertion | only in ~~linked~~ list which is efficient operation.
• ~~Deletion~~

Searching ← Linear
Binary → Linked list.
• possible ✓
• Inefficient → T.C (?)
Array

Sorting
↳ Inplace ~~stability~~
↳ Stability ← Array ÷
Linked list ÷

Lec. 12 Oct. 22



Lec. 13 | Oct. 22

Tree

→ Each node points to a no. of nodes.
A way of representing hierarchical way of nature in a graphical form.

→ The root of a tree is known as the node with a parent

→ Edge: An edge preferred to link from parent to children.

→ A node with no children is called leaf node. (E, J, A, H and I).

→ Children of same parent are called siblings (B, C, and D).

→ A node P is ancestor of node Q if there exist a path from root to Q and P appear

→ The depth of a node is the length of a from the root of the node.
depth of G is 2.

→ The height of the node is the length of from the node is to biggest node.
Height of B is 2.

→ The height of tree is length of the path from the root to the deepest

→ Height of a tree is maximum height of all the nodes and the depth of the tree is the maximum depth among all nodes

For a given tree depth and height are the same but for a node both may be different.

→ Size of node is the no. of descendent it has including itself.
Size of node C is 3.

→ If every node in a tree has only one child except leaf node, then tree is called skew tree.

→ If every node has only left child the tree is called left skew tree.

→ If every node has only right child then tree is called right skew tree

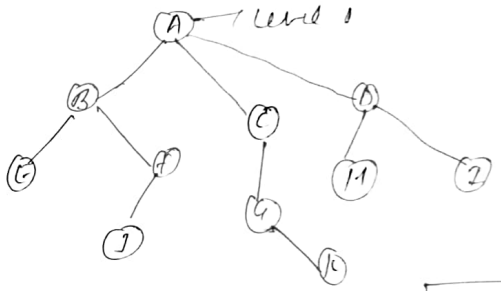
→ A tree is called binary tree if each node has zero, one or two children.

→ Empty tree is also a valid binary tree.

→ Strict binary tree: if each node has exactly two children or no children.

→ Full binary tree:- If each node has exactly two children and all leaf node are at the same level.

→ complete binary tree:- A binary tree in which every level except possibly the last, each



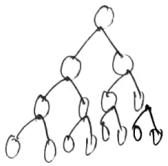
Lec.14 | Nov.19

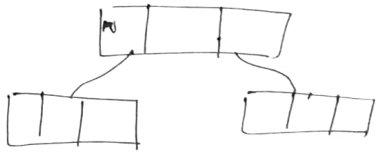
* Binary Tree

↳ No. of node in full binary tree
↳ No. of leaf nodes in full binary tree
↳ No. of NULL links in full binary tree → n+1

↳ Structure of Binary tree

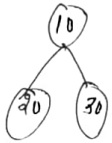
```
struct BTN{
    int data;
    struct BTN *left;
    struct BTN *right;
};
```





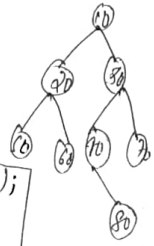
left order
inorder

- Preorder - 10, 20, 30
- Left order - 20, 30, 10
- Inorder - 20, 10, 30



** Recursion and iteration both possible. then which one used.
↳

```
void Preorder (struct BTN *root)
    if (root) {
        Printf ("%d", root → order);
        Preorder ( root → left);
        Preorder ( root → right);
    }
```

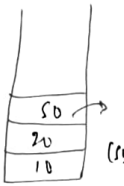


↳ using stack

① 10

② Pre (20) ⟶ 20 ③
Pre (50) ④
Pre (60)

③ Pre(30)



Print root
push root in stack
root = root → left
(5) n ← pop ()

```
void Preorder I (struct BTN * root)
    while (1) {                    → while (root) {
        Print (root)
        push (S, root)
        root = root → left
    }
    if (is Empty (S))
        break;
    root = POP(S)
    root = root → right
}
```

Q. WAF to find total no. of leaf node in a binary tree.

Sol:

```
Incorrect
    count = 1
    if (left == NULL || right == NULL)
        count = count + 1;
```

sol:

```
LeafNodecount(struct BTN *root)
{
    if (root != NULL)
    {
        if ((root->left == NULL) ff
            (root->right == NULL))
            return (1);
        else
            return(LeafNodecount(root->left)
                + LeafNodecount(root->right)
            {
            return 0
    }
```

Q. WAF to count no. of internal nodes.

sol:

```
InternalNodecount(struct BTN *root)
{
    if (root != NULL)
    { if(root->left == NULL) ff(root->right==NULL)
      return(0);
    else
      return [1 + InternalNodecount(root->left) +
              InternalNodecount(root->right)
    {
    return 0
}
```

Q. Total no. of nodes in a given binary tree.

sol:

Q. WAF to count total no. of null link in a binary tree.

sol: total number of nodes +1.

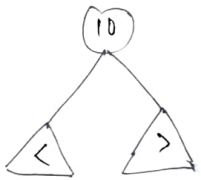
```
LeafNodecount(struct BTN *root)
{ if (root != NULL)
    {
```

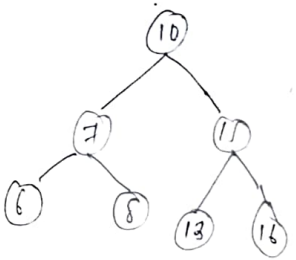
```
return (LeafNodeCount(root->left) + LeafNode(root->right)
}
return 1;
}
```

Q. WAP to find the hight of binary tree.

```
int HeightofBinaryTree(struct TreeNode *root)
{
    int leftheight, rightheight;
    if(root = NULL)
        return 0;
    else
    {
        leftheight = HeightofBinaryTree(root->left);
        rightheight = HeightofBinaryTree(root->right);
        if(leftheight > rightheight)
            return(leftheight+1);
        else
            return(rightheight+1);
    }
}
```


* Binary search Tree
  - A binary tree
  - All the elements in the left subtree should be less than the root.

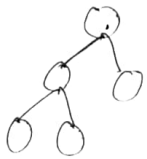




- Intention
- Deletion
- minimum — Recursive / Iterative
- maximum — Recursive / Iterative

→ minimum element :

```
void min-element ( struct BST * root)
{
    if ( root == NULL)
        return NULL;
    else if ( root → left == NULL)
        return root;
    else
        return FindMin ( root→left);
}
```



```
{
    if ( root == NULL)
        return NULL
    while (root → right != NULL)
        root = root → right
    return root;
}
```

↳ maximum

Q. WAF to find a given element in binary search tree.

Ans:

```
if (root == NULL)
    return NULL
else if (root → left == NULL)
    return root
else
```

left, right, key value

```
find ( root, data)
{
    if ( root == NULL)
        return NULL
    if ( data < root → data)
        return find (root → left, data)
    else if ( data > root → data)
        return find ( root → right, data)
    return root;
}
```

Q. WAP to insert a node(data) in BST.

Ans:

```
if (root == NULL)
    return NULL
    if (root == root->data) {
        root->left = root->right = NULL
    }
else {
    if (data < root->data) {
        return insert(root->right, data);
    }
    else if (data > root->data) {
        return insert(root->left; data);
    }
```

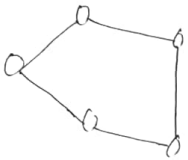
Deletion

1) If a bst two children inorder is maximum

```
    struct BST *temp;
    if (root == NULL)
        print data
}
```

Graph | Lec. 16 | Nov. 25

* Graph → A collection of vertices V and edges E.
A edge connected two vertices.

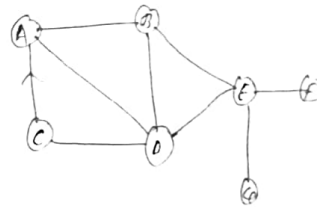


- path – A sequence of vertices to reach a vertices x from a vertex y.
- simple path – No vertex repeated.
- simple cycle – No vertex repeated except first and last vertex.
- complete graph – Each node is connected to every other node in the graph.
- Directed graph – Each edge is associated with some direction.
- Undirected graph : No specific direction.
- weighted graph – Each edge is associated with some weight.
- sparse graph – A graph with relative with few edges.
- dense graph – A graph with relatively few of the possible edges missing.

- dense graph – A graph with many edges.
- connected graph – No isolated node.
- Tree – connected graph with no cycle

Adjancy Matrix :

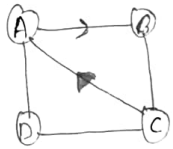
	A	B	C	D	E	F	G
A							
B							
C							
D							
E							
F							
G							

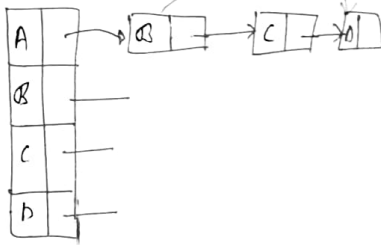


7×7 → Matrix
Application
Graphs

7×7×2
↓
Planes

Adjancy list :





** When and where to preferred Adjancy Matrix and list.

** When a graph is to be tree.

* Traversing in Graph

- Breadth first search ÷ using queue
- Depth first search using stack

1) set status = 1 for each node (ready)

2) Enqueue / push the starting node A and set it status = 2. (waiting)

3) Repeat step 4 and 5 until queue empty

4) Dequeue / pop a node N from top and process it and set its status = 3 (complete)

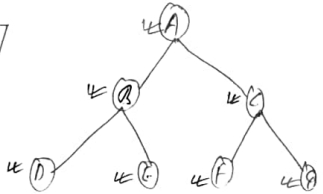
5) Enqueue / push all neighbours of N that are in ready state (status = 1) and set their status = 2

6) Exit.

#

A	B	C	D	E	F	G

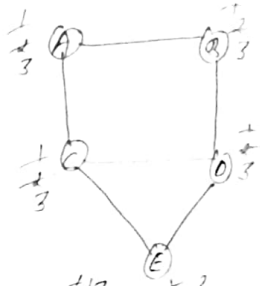
A B C D E F G



#

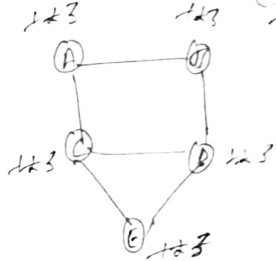
A	B	C	D	E

queue



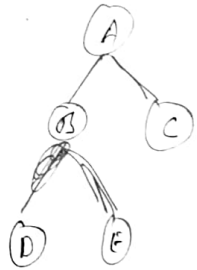
#

A B D E C



#

A B D E C



Lec 17 | Nov. 26

* Spanning Tree :

↳ subgraph of an undirected graph that contains all vertices is a tree

- No cycle
- minimally connected
- (n-1) edges

If we remove edges it divide into 2 subgraph

↳ n^{n-2} spanning tree are possible in a complete graph with n nodes

→ more than one spanning tree if possible.

↳* minimum spanning tree : (weighted graph)

→ Prim's Algorithm :

- start with a single node
- Explore all neighbours and edges with minimum weighted cauling no cycle.

→ Kruskal's Algorithm :

- select edges with ~~min~~ minimum weight and keep adding until spanning tree is created.

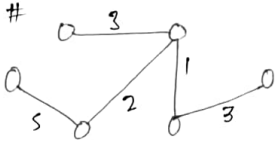
↳ Example of greedy ↳ provide optimal solution.

minimum weight's sum

key-kil node ko connect kar sakta hai!!

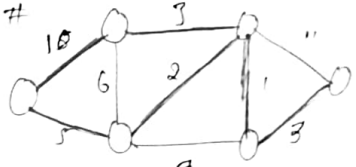
↳ Kruskal's → disconnected graph

#



↳ Prism's Algorithm

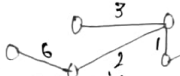
#



↳ 2 spanning tree

↳ Kruskal Algo.

#



↳ kruskal and prim can give more than ~~one~~ spanning minimum spanning if which the ~~of~~ provided prim and kruskal.

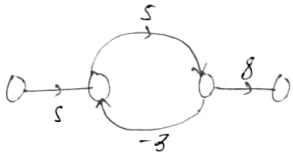
weighted ~~graph~~ cycle both +ve and -ve example.

↳ If weight is equal than there is no unique spanning tree with (n-1).

* Single - source shortest path :

↳ A shortest path from a given source to every vertex.

↳ Negative weighted cycle



→ Dijkstras Algorithm :

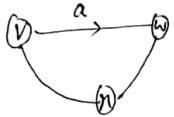
↳ All weight are non-negative, shortest path properties.

> optimal substructure

↳ All path of shortest path are shortest path.

> Thionyle inequality

$$d(v, w) \leq d(v, x) + d(x, w)$$



* minimum predecessor
  ↳ (predecessor pote hona chahiye)

Graph ↑

> initialize - single source (G, s)

for each vertex $v \in V[G]$
do $d[v] \leftarrow \infty$

* why all weighted are non-negative in Dijkstra algorithm ? infinite loop.

Lec. 18 | Dec-2

Hashing

↳ storing and retrieving data in O(1) time

- Linear search → O(n)
- Binary search → O(log n)
- Hashing → O(1)
  ↳ unnecessary utilization of memory.

↳ Hash function

↳ Hash address = key MOD 10

↳ Hash address = K MOD n

- 4 MOD 6 = 4
- 8 MOD 6 = 2

Hash function –

1) **Division modulo Method :**

↳ H(key) = key mod M

2) **Mid square Method :**

key = 135 m = 10 (No. of entries)

↳ 32 = 9 ↳ Mid of square

3) **Folding method :**

key = 123456789 → 123 + 456 + 789 = 1368 → 136 + 8 = 144

m = 10000

4) **Fold boundary Method :**

m = 1000 123456789 → 123 + 789 = 912

123 132 789 ↳ same index for keys

5) **Translation Method :**

↳ any three continuous digit

m = 1000 123456789

* characteristics of good hash functions :

- minimum collision
- Easy to calculate
- Distributed key value evenly
- use all information provided in the key

* perfect hash function :

↳ A hash function which produces always an empty until table is full.

↳ No hash function is perfect

Q. Find the no. of collisions when the following keys are kept in a hash table of size 13. keys : 10, 100, 32, 45, 58, 126, 3, 29, 200, 400, 0.

① ↳ If hash function is key are first folded by adding their digit together.

②

Sol: ① 10 % 13 = 10 ✓
100 % 13 = 9 ✓
32 % 13 = 6 ✓
45 % 13 = 6 ✓
58 % 13 = 6 ✓
126 % 13 = 9 ✓
3 % 13 = 3 ✓
29 % 13 = 3 ✓
200 % 13 = 5
400 % 13 = 10 ✓
0 % 13 = 0

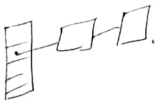
(10, 10, 9, 9, 6, 6, 6, 3, 3)
↳ No. of collisions = 5.

Sol ②:
10 = 1+0 = 1
100 1+0+0 = 1
3+2 = 5
4+5 = 9
5+8 = 13 = 1+3 = 4

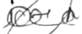
1+2+6 = 9
3
2+9 = 11 ⇒ 1+1 = 2
200 = 2+0+0 = 2
4+0+0 = 4

(1, 1, 5, 9, 4, 5, 3, 2, 2, 4, 0)
No. of collisions = 3

Collision Resolution Technique ÷



1) Chaining ÷ open Hashing — used when same indexing is lost, searching O(n)
↳ chain of NCE linked list
2) Linear Probing ÷ → worst case is O(n), static memory allocation — closed Hashing (यह space को use करता है)
↳ Next available space
3) Random Probing ÷ c = 3



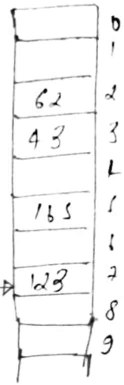
4) Quadratic Probing ÷
↳ $h + i^2$ mod n — i → No. of attempt (Prob No.)
5) Double Hashing ÷ → $(h + i^2)$ mod n

Double hashing → reversibility skill

CRT : key mod (10 - 2) + 1

43, 165, 62, 123, 143, 20, 30

→ 123
123 mod 10 = 3
123 mod (10 - 2) + 1 = 4
3 + 4 = 7.



→ 143
143 mod 10 = 3
143 mod (10 - 2) + 1 =

Rehashing → new topic
No. of Prob count → exam view
↳ find for linear, quad...
Reversible → h → key, h ← key

* In cryptographic hash
  ↳ No reversibility exist — h ← CH(x), h' ← CH(x') ✗

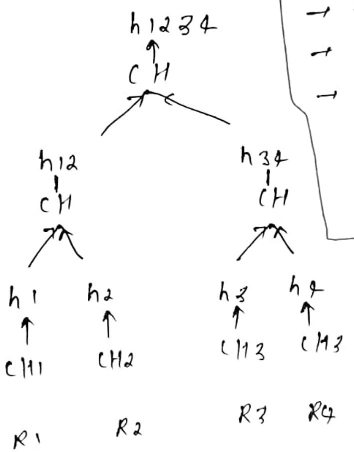
↳ Birthday paradox

Lec. 19 | Dec-3

* Merkle Tree

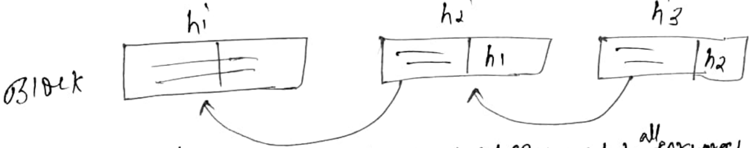
↳ Binary tree + cryptography hashing

- It is irreversible
- keep records of R1, R2, R3, R4
- If modification is done in R1 or R2 or R3 or R4 then change in h1234 is reflected.



Generate hash value
cryptography hashing

h'' ← CH(R1 R2R3R4)



↳ Block chain → chain of blocks (stored to all servers)
* modification by one server is reflected only on that particular server.

** P vs NP Problem :-

↳ P → (All the problem we have to solve) Deterministic turing machine. solve a problem in → polynomial time

↳ NP → Non-deterministic turing machine, solve a problem in polynomial time.

- 3 satisfiability problem :-
  - Bin problem
  -
  -

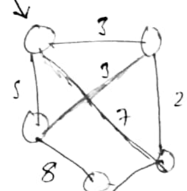
*** Knapsack

Theorem that proves profit is P/2

↳ 2 approximation algorithm
  ↳ Read Proof — Assignment of h/w

Q. Travelling salesman problem (NP) :- O(n!)

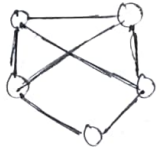
↳ A salesman has to visit n cities. All cities must be visited once and at the end the salesman returns to the city from where he/she started.



NP Problem
↳ 3 SAT Problem

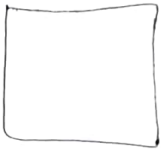
undicidable Problem
Genetic Algorithm
↳ O - P
optimal sol. → Polynomial time

Genetic Algorithm
↳ Give near optimal solution in NP problem
↳ Approximation Algo. → 2-AA



knapsack Problem
↳ fractional knapsack problem
↳ A fraction of given item can be taken.
↳ 0/1 knapsack problem
↳ either you take whole item or you can leave

n → 2^n → f.s → optimal solution
→ N.f.s



I1	I2	I3
w1	w2	w3
P1	P2	P3
P1/w1	P2/w2	P3/w3

capacity $w_1 + w_2 + w_3 < C$ else

Profit / weight check

→ Give optimal solution
→ we are giving alternate more profit
→ Greedy approach

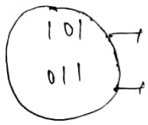
0/1 knapsack (NP) ÷
we return array

1	0	1

Item 1 (V)	Item 2 (V)	Item 3 (V)
0/1 I1	0/1 I2	0/1 I3
w1	w2	w3
P1	P2	P3

total no. of decision $= 2^3 = 8$

→ 2^n (solution) → f.s. → optimal (can be unique or more than one).
→ N.f.s.



more than one solution

(P, P) ← it form may

T.C. = $O(2^n)$

* Genetic Algorithm ÷ (we are evolving)

↳ Give near optimal solution 0 — P

knapsack 0/1 → near optimal solution

1 ..... 2^n max

0 ..... P (profit)

↳ objective ÷ to give good result (near to optimal)

* Approximation Algorithm

↳ give at least 1/2

* Both algorithm execute in polynomial time

greedy (local optimal) — current best.

| Lec. 20 | Dec. 9 |

* Dynamic programming

↳ 1. Optimal substructure

↳ 2. Overlapping subproblem

↳ It break down the complex problem into subproblems.

↳ Find optimal solution of subproblems.

↳ Stored result of subproblems

↳ Reuse the stored results.

↳ calculate the final result

$$f(n) = f_{n-1} + f_{n-2}$$

* LCS (Longest common sequence)

↳ subsequence

$S = \{A, B, B, A, A, B\}$

$S_1 = \{A, A, A, B\}$

$S_2 = \{B, A, B\}$

$S_3 = \{A, B, B, B, B\}$

Algorithm ÷

$S_1 = (A, B, B, A, B)$ m

$S_2 = (A, A, B, B, A)$ n

if $m \leq n$

$O(2^n)$ or $O(2^m)$

- problem jab hai woh subproblem ko divide ho raha hai aur woh optimal hai
- This is providing us patterns that involved the finding the optimal solution.

Q. $LCS(i,j) = \begin{cases} 0 & i=0 \text{ or } j=0 \end{cases}$

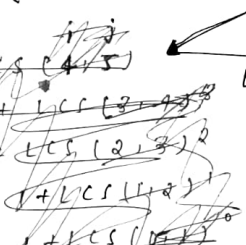
$x = A, B, B, A, B$

$y = A, A, B, B, A$

$\max(LCS(5,4), LCS(4,5))$ [Empty in table] $5\times5 = 25$

$\Rightarrow 4$

for: $LCS(i,j)$



$LCS(4,5) = 4$

$1 + LCS(3,4)$ 3

$1 + LCS(2,3)$ 2

$1 + LCS(1,2)$ 1

$1 + LCS(0,1)$ 0

we have reduced it in $O(m \times n)$

$LCS(5,4)$

$1 + LCS(4,3)$ 2

$\max(LCS(3,3), LCS(4,2))$ 1, 2

$LCS(3,3)$

$1 + LCS(2,2) \Rightarrow 1$

$\max(LCS(1,2), LCS(2,1))$ 1, 1

$LCS(1,2) \Rightarrow 1$

$1 + LCS(0,1)$ 0

$LCS(2,1)$

$\max(LCS(1,1), LCS(2,0))$ 1, 0

$1 + LCS(0,0)$ 0

$LCS(4,2)$

$1 + LCS(3,1)$

$\max(LCS(2,1), LCS(3,0))$ 1, 0

Lec. 21 | Dec. 10

* **knapsack problem**

Starting criteria
if $i=0$ or $j=0$
if $j < wi$

$V[0 \ldots n, 0 \ldots c]$

$$V[i,j] = \begin{cases} 0 \\ V[i-1, j] \\ \max(V[i-1, j], \\ \quad Vi + V[i-1, j-wi] + i+j \text{ of } Vi \end{cases}$$

- $n \rightarrow$ no. of items
- $Vi \rightarrow$ value of ith item
- $wi \rightarrow$ weight of ith item
- $c \rightarrow$ capacity of knapsack

	i \ j	1	2	3	4	5	6	7	8
$w1=3, v1=0$	0	0	0	0	0	0	0	0	0
$w2=4, v2=3$	1	0	0	0	2	2	2	2	2
$w3=5, v3=1$	2	0	0	0	2	3	3	3	5	5
$w4=6, v4=4$	3	0	0	0	2	3	3	3	5	5
	4	0	0	0	2	3	3	4	5	5

* Matrix multiplication

* use of knapsack in real life

* Greedy Algorithm

↳ Greedy can give optimal solution sometimes but not in general.

↳ 0/1 knapsack

↳ fractional knapsack

w	v	v/w	
5	3	3/5	
2	2	2/2 = 1	
4	7	7/4	may or maybe not optimal
3	1	1/3	

0/1 →
- ascending weight
- descending value
- v/w

→ greedy approach $O(n \log n)$

→ and dynamic programming $O(2^n)$

2-AA ÷ optimal ≤ 2P greedy (i.e. ka minimum)

Q. Activities selection problem (Greedy & Sorting)

Sol? n activities ÷ $A_1, A_2, A_3, \ldots A_n$

A_i has start time s_i and a final time f_i

$s_i \leq f_i$

$A_i \rightarrow [s_i, f_i)$

A_i and A_j are compatible if $[s_i, f_i) \neq (s_j, f_j)$ are non overlapping pairs.

↳ Objective – maximum no. of mutual compatible activities

→ minimum time
→ maximum no. of activities
→ knapsack
→ sort on the basis of finishes (easy)