Structures & Unions

defining structure:

```
struct book_bank {
    char title[20];
    int pages;
    float price;
};
```

Variables:

```
struct A {
} V1, V2;
```

(or)

```
struct A V1, V2;
```

Typedef struct:

```
typedef struct {

} struct_name;
```

Accessing

1) .
2) (*ptr).a → member
3) ptr → a

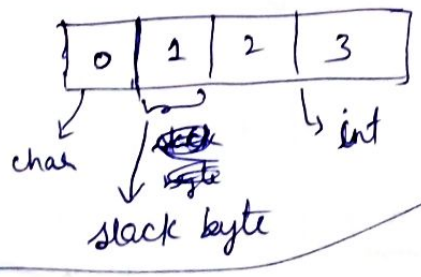
Copying & Comparing

```
struct Person P1, P2;
```

P1 = P2 ✓

P1 == P2 ×

Word Boundaries & Slack Bytes:-



* The last value assigned will be assigned to all the members irrespective of data types

* UNIONS

→ union item {
    int m;
    float x;
    char c;

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union

code.m = 379
code.c = 'a';
(cout << code.m)
↳ gives error because we should make sure that we are accessing the member who value is currently stored.

→ union item abc = {100}; ✓
union item abc = {10.75}; ✗

because during initialisation, we have to use only first member of structure.

BIT Fields:

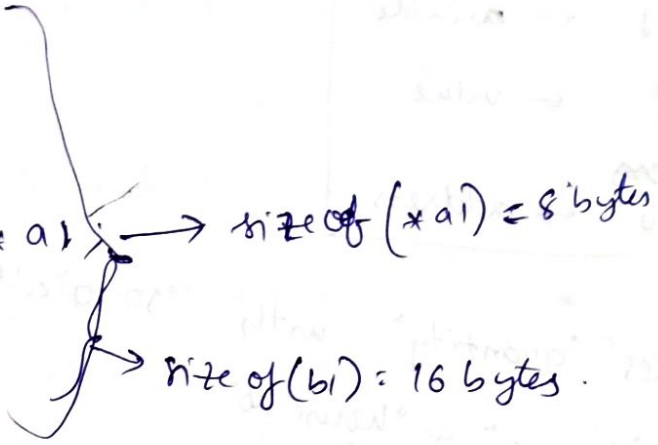
```
struct personal {
    unsigned s  : 1
    unsigned age : 7
    unsigned m  : 1
    unsigned c  : 3
    unsigned    : 4
};
```

several data items packed in a word of memory
16 bits

Structure size:

→
```
struct A {
    int a;
    struct A a1; → ✗ (invalid)
};
```

```
struct A {
    int a;
    float b;
    struct A *a1;
} b1;
```



Pointers

→ derived data type

→ contains memory addresses as their values

1) pointers can be used to access and manipulate data stored in the memory

2) return multiple values from a function

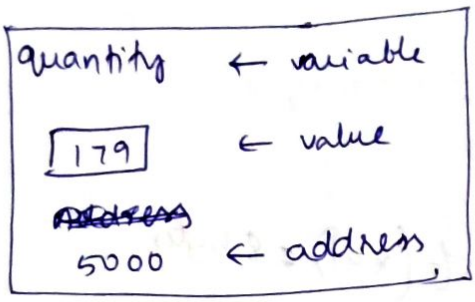
3) references to functions & thereby facilitating passing of functions as arguments to other functions

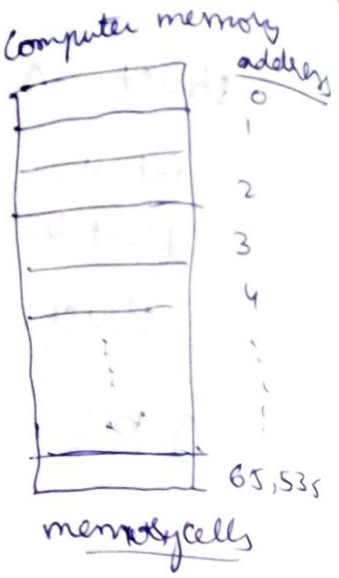
4) Support dynamic memory management

5) efficient tool for manipulating data structures such as structures, linked lists, queues, stacks, trees.

6) Reduce lengths & Complexity of programs

7) increase the execution speed and thus reduce the pgm exe. time





→ associates "quantity" with "5000" address

like "house" & "house no"

→ can access "179" either with the name 'quantity'

(or) memory address.

⇓

assigned to variables

⇓

Such, variables that hold memory addresses are

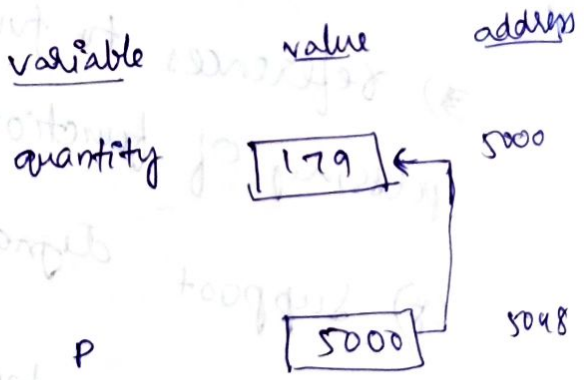
called pointer variables.

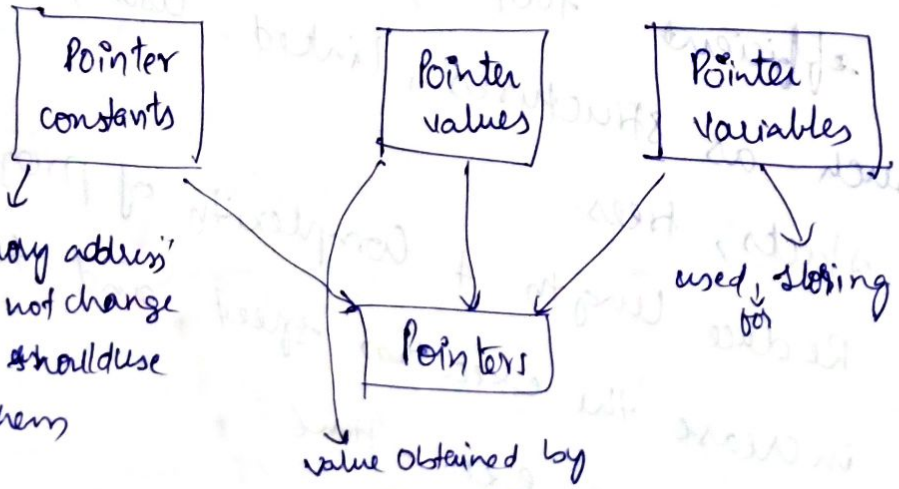
```
int *p = &quantity;
```

'p' points 'quantity'

⇓

pointer





accessing the address of a variable:

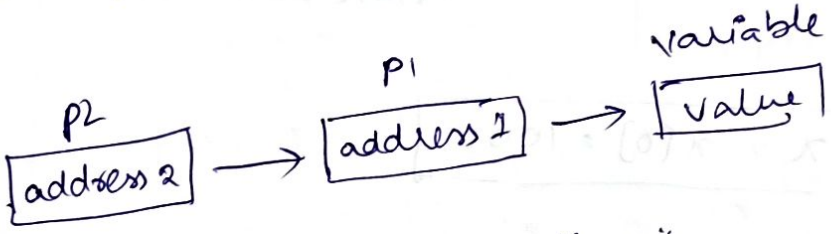
'&'

Declaring Pointer Variables:

datatype *ptrname;

Ex.. int *a;

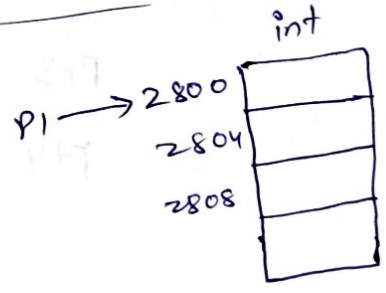
chain of Pointers:



"multiple indirections"

```
main() {
    int x, *p1, **p2;
    x = 100;
    p1 = &x;
    p2 = &p1;
    printf("%d", **p2)   // output 100
```

Pointer Increments and Scale Factor:-



p1 = 2800

(p1+1) ⇒ 2804

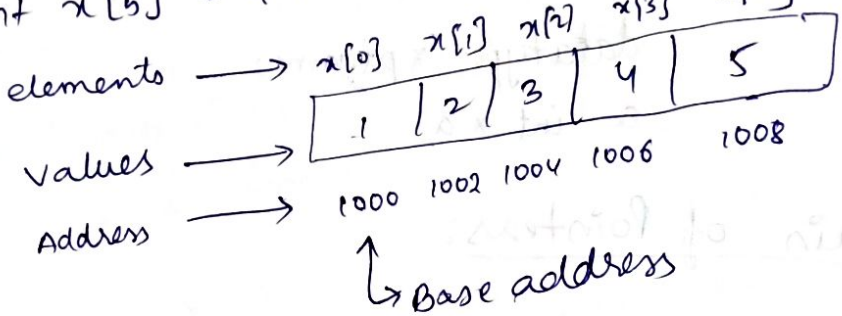
not 2801

p1++ ⇒ 2804

p1+2 ⇒ 2808 not 2802

Pointers And Arrays

int x[5] = {1, 2, 3, 4, 5}



* x = x[0] = 1000

Ex:- int *p;

x = &x[0] = 1000

p = &x; ⇒ p = &x[0]

Now,

p = &x[0] (=1000)

p+1 = &x[1] (=1002)

p+2 = &x[2] (=1004)

p+3 = &x[3] (=1006)

p+4 = &x[4] (=1008)

address of x[3] = base address + (3x scale factor of int)

= 1000 + (3x2)

= 1006

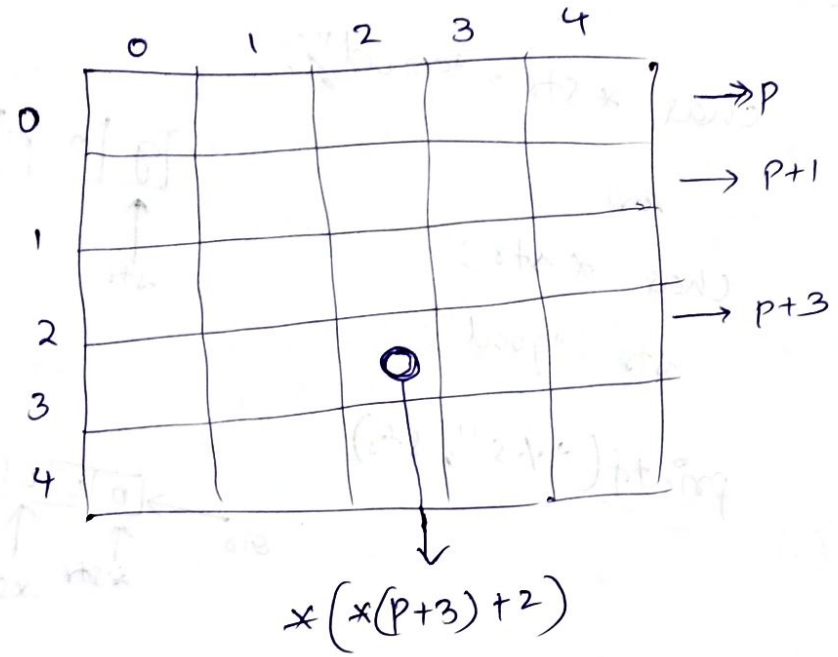
⇒ *(p+3)

Two Dimensional Array:

for 1D-array = *(x+i) or *(ptr+i)

for 2D-array = *(*(x+i)+j) or

((ptr+i)+j)



P → pointer to first row

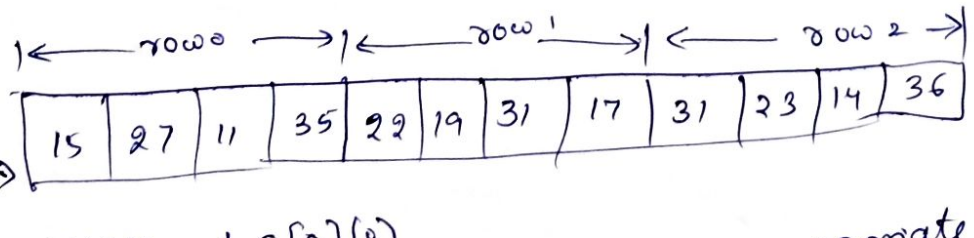
P+i → pointer to ith row

*(P+i) → pointer to first element in the ith row

*(P+i)+j → pointer to jth element in the ith row

((P+i)+j) → Value stored in the cell (i,j)

int a[3][4] = { {15, 27, 11, 35}, {22, 19, 31, 17},
{31, 23, 14, 36} };



→ compiler does appropriate mapping

Pointer and character strings :

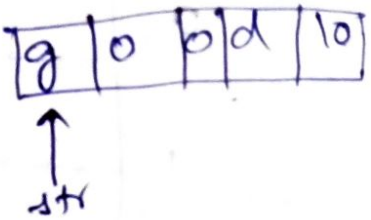
char str[5] = "good";

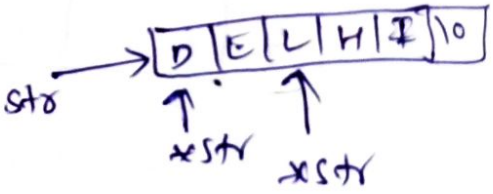
automatically '\0' ~~auto~~ is inserted at the end of string by compiler

```
char *str = "good";
```

(or)

```
char *str;
str = "good";
printf("%s", str);
```

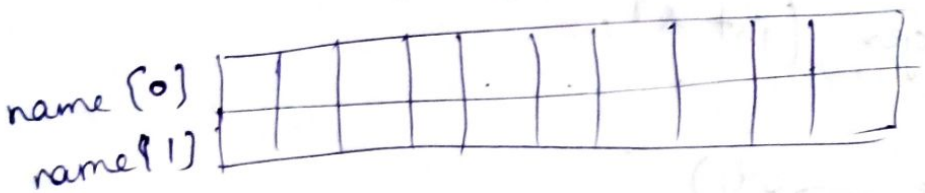





```
main() {
    char *str;
    str = "delhi";
    while (str != '\0') {
        printf("%c is stored at %d", *str, str);
        str++;
    }
}
```

Array of Pointers:-

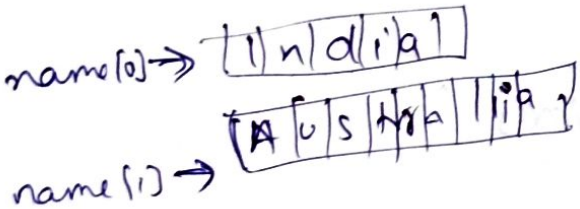
```
char name [2] [10];
```




```
char *name[2] = {"India", "Australia"}
```

name [0] → India

name [1] → Australia




```
for(i=0; i<=2; i++)
printf("%s\n", name[i]);
```

```
*(name[i]+j)
```

*p[3]
↳ an array of 3 pointers

(*p)[3]
↳ pointer to an array of three elements

because '*' has low precedence

Functions Returning Pointers :

```
(int *)larger (int *x, int *y) {

    if (*x > *y)
        return (x);
    else
        return (y);
}
```

Pointers To Functions :

if **ptr is a 2d array pointer

*ptr[0] --> elements for example

Below is the matrix structure:

*ptr[0] - [0,0] - [0,1] - [0,2]
|
*ptr[1] - [1,0] - [1,1] - [1,2]
|
*ptr[2] - [2,0] - [2,1] - [2,2]

It's wrong.
Below one is correct :

```
int **p = (int **) malloc(no_of_rows*sizeof(int));

for(i=0;i<no_of_rows;i++){
p[i] = (int *) malloc(no_of_columns*sizeof(int));
for(j=0;j<2;j++){
  cin>>*(*(p+i)+j);
}
}
```

(*ptr).age = 63 (constant)

(or)

ptr → age = 63

int *a = (int *) malloc (10 * sizeof(int));
cin>>*(a+i);

int **a = (int **) malloc (10 * sizeof(int *));

for (i=0; i<5; i++)
*(a+i) = (int *) malloc (2 * sizeof(int));
for(j=0; j<2; j++) cin>>*(*(a+i)+j);

DYNAMIC MEMORY ALLOCATION AND LINKED LISTS

→ dynamic data structures ~~with~~ in conjunction with dynamic memory management techniques

→ Process of allocating memory at run time is known as "Dynamic Memory Allocation"

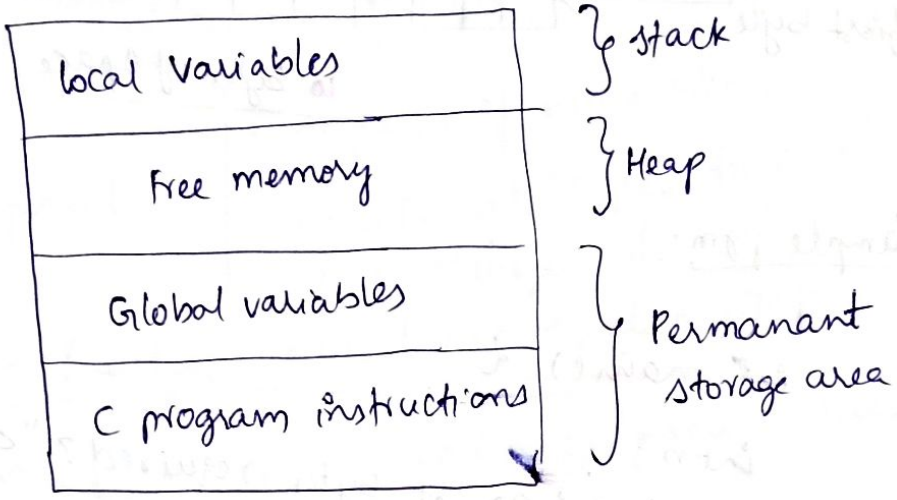
→ Four library routines known as "Memory Management Functions"

↳ allocating & freeing memory during program execution

Function	Task
malloc	Allocates request size of bytes and returns a pointer to the first byte of the allocated space
Calloc	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory
free	frees previously allocated space

realloc — modifies the size of previously allocated space

Memory Allocation Process:



→ size of heap keeps on changing when program is executed due to creation and death of variables that are local to functions & blocks

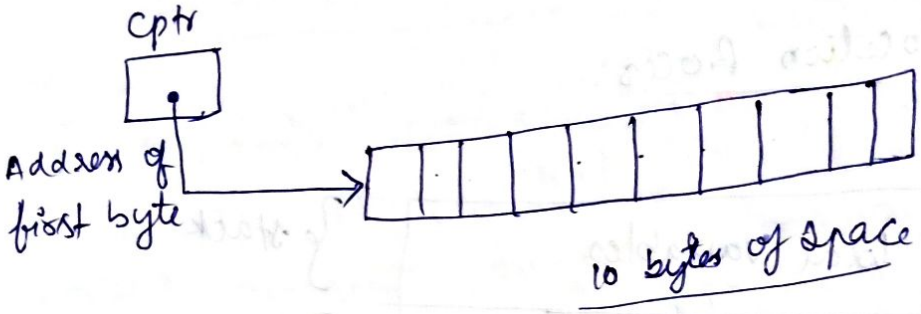
→ possibly encounter "memory overflow" during dynamic allocation. In such situation, function returns a "NULL" pointer

Allocating a Block of Memory: (MALLOC)

ptr = (cast-type *) malloc (byte-size);

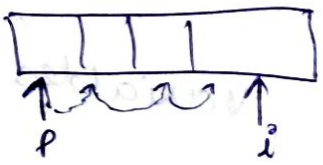
x = (int *) malloc (100 * sizeof(int));

cptr = (char *) malloc(10);



Sample pgm:

```
int main() {
  int n;
  cout << "no of bytes required ?:" << "\n";
  cin >> n;
  int * p = (int *) malloc (sizeof (int) * n);
    int *i = p;
    for (p = i; p < i+n; p++) {
      cin >> *p;
    for (p = i; p < i+n; p++)
      cout << *p;
  }
```



ALLOCATING MULTIPLE BLOCKS OF MEMORY:

CALLOC:

Calloc - generally for storing derived data types such as arrays and structures

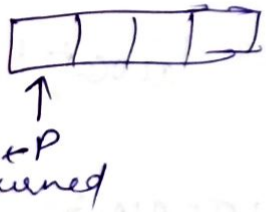
malloc	Calloc
→ single block of storage space	→ multiple blocks of storage
→ one argument	→ each of same size. → sets all bytes to zero → two arguments

ptr = (cast_type *) calloc (n, elem-size);

→ contiguous space for 'n' blocks, each of size, elem. size bytes

→ "all bytes initialised to zero"

→ pointer to the first byte of the allocated region is returned



Ex:

```
struct student {

};
(student *) st = (student *) calloc (30, sizeof(s));
```

Releasing the Used Space: FREE:

free(ptr);

1) It is not the pointer that is being released but rather what it points to

Ex:- (int *) i = (int *) malloc (sizeof(int) * 4);

free(i);

i ⇒ remains & can be used but, the memory is released

and no other pointer should point to releasing memory else it shows runtime error

2) just release the pointer, to release memory allocated to array of elements.

ALTERING SIZE OF A BLOCK : REALLOC :

ptr = (type *) malloc (size)

ptr = realloc (ptr, newsize)

1) if additional space needed (or) space to be reduced realloc is used

2) "Reallocation" is done means, create the same in an entirely new region and move the contents of old block into the new block

How to Practice a Question

1. Try to solve the problem on your own
2. Write the code for the algo on paper
3. Test code
4. Type code in comp.

Five steps to a technical question :- (Five step approach)

1. Ask your interviewer qstns to resolve ambiguity
2. Design an algorithm
3. pseudocode first
4. write your code at moderate space
5. test code and carefully fix any mistakes

Five Algorithm Approaches :-

1. Examplify:

Take an example and try to solving using example.

Ex: Given a time, calculate the angle between the hour and minutes hands.

2. Pattern matching:

Under this approach, we consider whether problems the algorithm is similar to and try to modify the solution to the related problem to develop an algorithm

3 Simplify & Generalize:

4. Base Case & Build:

$n=1$, $n=2$, $n=3$ then $n=n-1$, $n=k$

5 Data structure Brainstorm

- Check with each & every data structure whether it suits to the problem or not.

What good code looks like:

1) Appropriate data structure

2) Appropriate code Reuse

3) Modular

4) Flexible & Robust

5) Error checking [Exceptions]

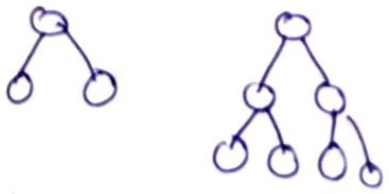
Introduction to Heaps

$O(\log n + n)$

	insert	search	find min	Delete min
Unsorted array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted array (ascending)	$O(n)$	$O(\log n)$	$O(1)$	$O(n)$
Unsorted Linked list	$O(1)$	$O(n)$	$O(n)$	$O(n+1) = O(n)$
Heap	$O(\log n)$		$O(1)$	$O(\log n)$

Heap is a binary tree (or) 3-ary .... (or) n-ary tree

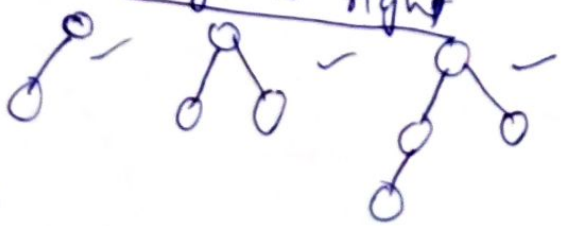
complete binary tree –



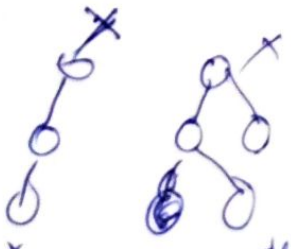
Heap (Almost Complete B-T)

↓

leaves should present only in last level (or) from last but one level & filled from left to right

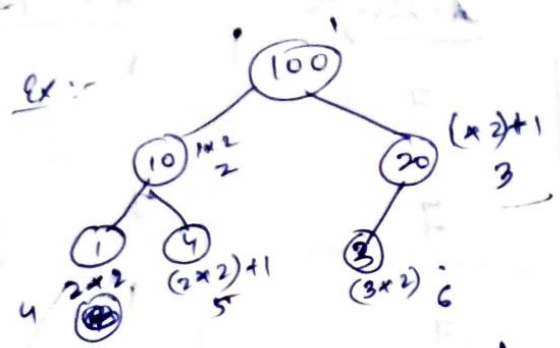


Max Heap:



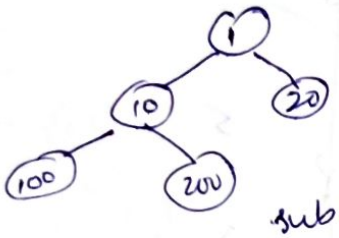
All the elements in left & right subtree should be less than root. for every node

Ex :-



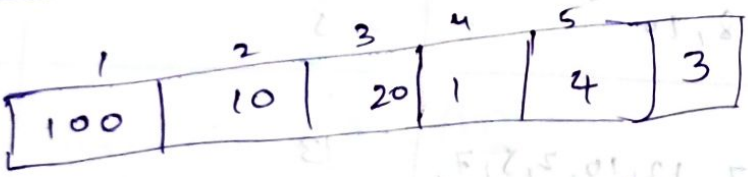
Min heap : ~~Parent~~ Parent ~~Root~~ Root → Should be ~~greater than~~ ~~child~~ its ~~right & left subtree elements~~ ~~children~~

"Root" contains min. element of entire tree.



→ All the elements in left & right subtree should greater than root for every node

Implementation



left shift

- left child - $2 \times (\text{parent index})$ = $2 * i$
- Right child - $2 * (\text{parent index}) + 1$ = $(2 * i) + 1$

↳ left shift +1

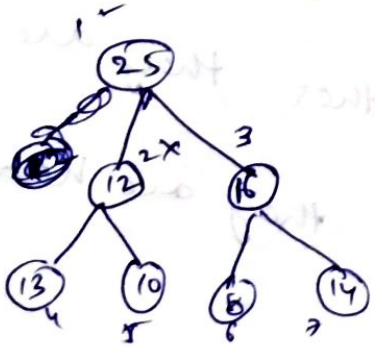
$$\text{parent} = \frac{(\text{child index})}{2} = \text{Right shift}$$

→ check below examples, whether they are heaps, if not upto what length they are heaps

		A.length	A.heapsize
①	25, 12, 16, 13, 10, 8, 14	7	1
②	25, 14, 16, 13, 10, 8, 12 ✓	7	7
3.	25, 14, 13, 16, 10, 8, 12	7	1
4	25, 14, 12, 13, 10, 8, 16	7	2
5	14, 13, 12, 10, 8 → descending order ⇒ max heap ✓	5	5
6	14, 12, 13, 8, 10 ✓	5	5
7	14, 13, 8, 12, 10 ✓ max heap	5	5
8	14, 13, 12, 8, 10 ✓	5	5
9	89, 19, 40, 17, 12, 10, 2, 5, 7,	13	2
●	11, 6, 9, 70		

A.heapsize

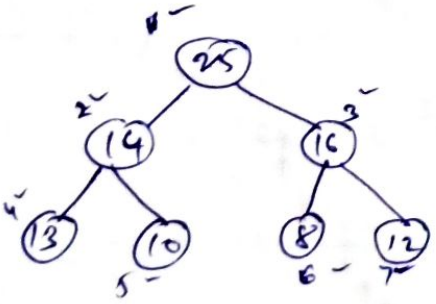
The no. of elements following heap property



⇒ heapsize = 1 [∵ only 1 element follows heap property i.e.,

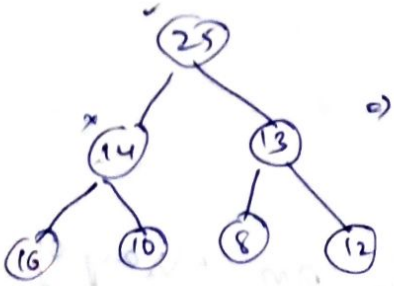
⇒ parent → children

②

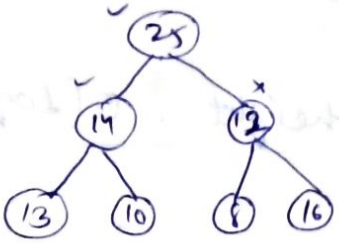


⇒ max heap

(3)



⇒ heap size = 1

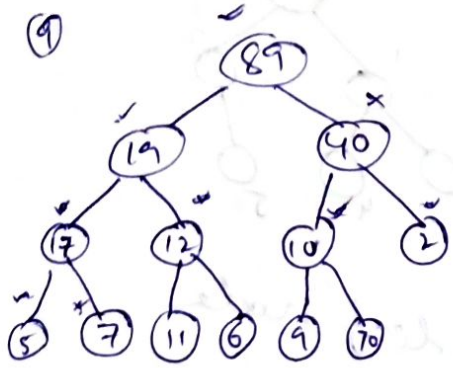


⇒ h.s = 2

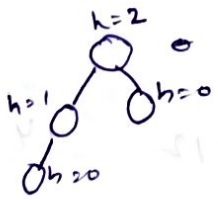
→ Array ⇒ Ascending order ⇒ ~~max~~ min heap

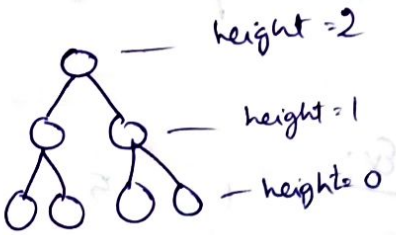
⇒ Descending order ⇒ max heap

(4)



Max Heapify Algorithm





Height of C.B.T	1	2	3	4
max no of nodes	3	7	15	31

⇒ max. no. of nodes $= (2^{h+1} - 1)$

h = height

Complete ternary tree

max no. of nodes $= 3^{h+1} - 1$

Complete n-ary tree

max. nodes $= n^{h+1} - 1$

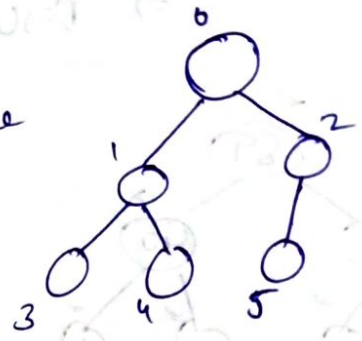
Relation b/w height & no. of nodes

$$h = \lfloor \log n \rfloor$$

height : $O(\log n)$

Construction of Max-heap from an array of elements

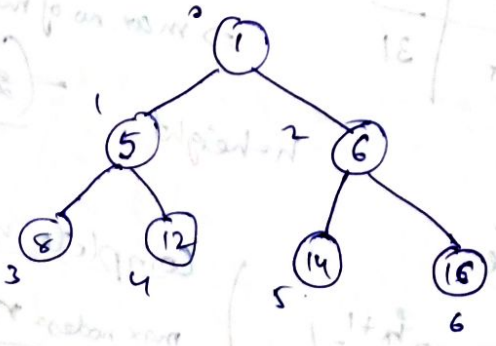
** In a complete binary tree, the elements starting from $\left(\left\lfloor \frac{n}{2} \right\rfloor + 1 \text{ to } n\right)$ ~~are leaves~~ is the largest non-leaf node



* Every leaf is a heap.

Ex:- 1 5 6 8 12 14 16

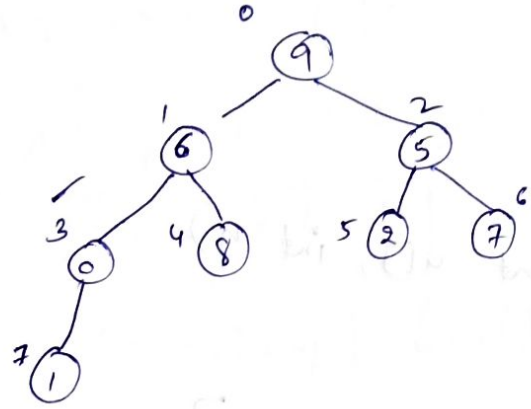
Step 1: write in binary tree



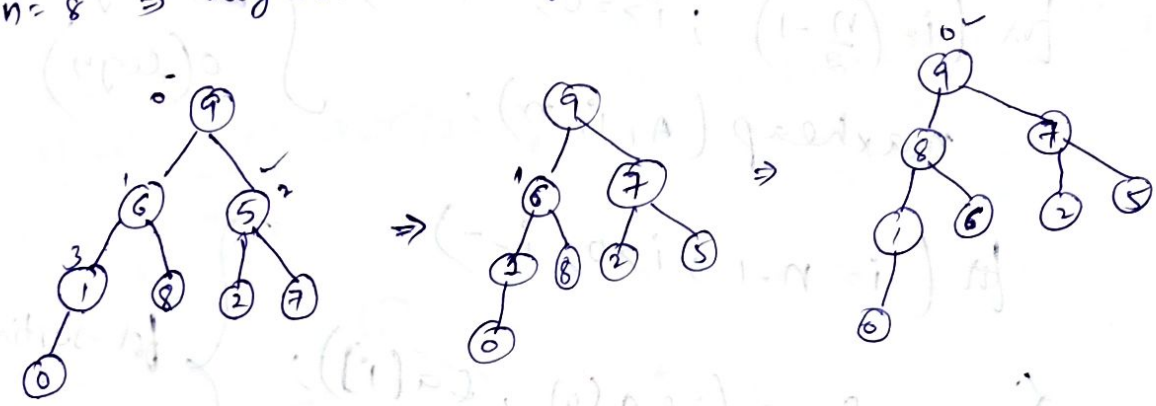
Here '2' is the largest non-leaf node

→ start from the largest non-leaf node and start changing tree to form max heap by moving towards up in tree

(2) 9 6 5 0 8 2 # 1



n = 8 ⇒ highest non-leaf node ⇒ $\frac{n}{2} - 1 = \frac{8}{2} - 1 = 3$



Refer Nagamani Notes (Pictures)

```
void maxheap (int A[], int i, int n)
{
    int child;
    int temp;
for (temp = a[i]; leftchild(i) < N; i = child)
{
    child = leftchild(i);          // first compare leftchild, if (i.e) < (rightchild) then move to rightchild
if ( child != n-1 && a[child+1] > a[child])
    child++;
    if (temp < a[child])           // compare rightchild & root
    a[i] = a[child];
    else break;
```

```
    }
    a[i]= temp;
}

void heapsort(int a[], int n)
{
    int i;
    for(i=(n/2-1); i>=0; i--)        } building maxheap
        maxheap(A, i, n);                 O(logn)

    for(i=n-1; i>0; i--)
    {
        swap(&a[0], &a[i]);          } for sorting
        maxheap(a, 0, i);                 O(n)
    }
}
```

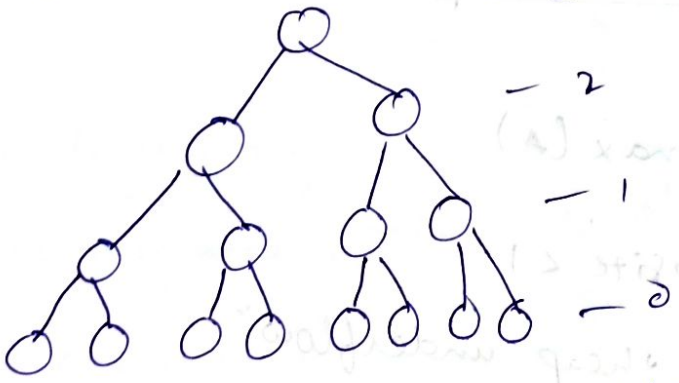
⇒ Total for sorting $O(n \log n)$

After sorting just print array. to see the output

Algorithm for heapifying

```
Build-max-heap (A)
{
    A.heapsize = A.length
    for (i = ⌊A.length/2⌋ down to 1)
        max-heapify (A, i)
}
```

→ Number of nodes of different heights in a complete binary tree



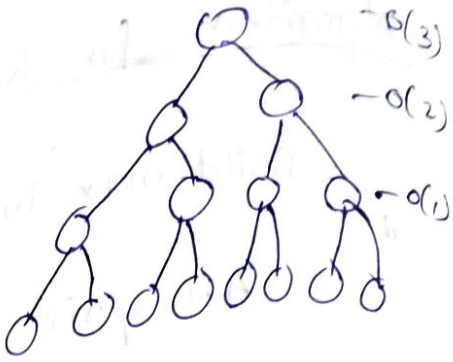
→ if no. of nodes = n; height = h

∴ Then, no. of nodes at height, h (atmost) $= \left\lceil \frac{n}{2^{h+1}} \right\rceil$

Ex: height = 0, $\left\lceil \frac{15}{2^{0+1}} \right\rceil$: 8 nodes (atmost)

h = 2; $\frac{15}{2^{2+1}} = \frac{15}{2^3} = 2$ nodes

height = 2 ⇒ 2 nodes



$$\sum_{h=0}^{\log n} \left\lceil \frac{n}{2^h \cdot 2} \right\rceil (c \cdot h)$$

$$= \frac{cn}{2} \left(\sum_{h=0}^{\log n} \left(\frac{h}{2^h} \right) \right)$$

$$\downarrow$$

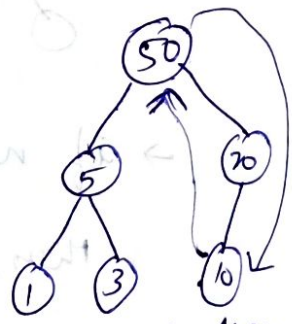
$$2$$

$$= \frac{cn}{2} \cdot 2$$

$$\Rightarrow O(n)$$

Extracting max in heap

```
Heap-extract-max (A)
{
    if A.heapsize < 1
        error "heap underflow"
    max = a[1]
    a[1] = a[a.heap_size];
    a.heapsize = a.heapsize - 1;
    max_heapify (a, 1)
    return max;
}
```



& then "remove" 50

and reduce heap size

and apply max_heapify on "10" and then it forms heap & then again if you want you can delete max element

Training

Applications of linked list:-

1. Media player
   1. volume - doubly linked list
   2. forward button -
   3. Reverse button -
   4. Play, pause -

Stacks
1. browsing history (queue)
2. Inbox

Queue :-
- customer care
- Online Reservation

Dynamic Programming

→ Intermediate soln's are stored and used to find the solution.

Generic form of printf

printf("");

printf(const char *, ...)
- ... → variable no. of arguments (list)
- * → dynamically allocated string

printf(
"%d",

→ Syntax for making a ~~building~~ built in function

pgm → compiler → Assembler → Loader → Run

C

b Ex: construction of house
1) foundation
2) pillar
3) roof 4) walls

1) Top-down

C++

1. Integration of parts

2. Bottom up

class:

User defined data type

object - instance of class

- object has life (memory allocated)

Encapsulation :-

```
class calc
{
    int a;
    int b;
public:
    int add(int a, int b)
    {
        int c;
        c = this.a + this.b;
        return c;
    }
    int sub(int a, int b)
    {
        int c;
        c = this.a - this.b;
        return c;
    }
}
```

```
int main() {
    Calc cal(a,b);
    cout << "Enter operation";
    cin >> op;   cout << "Enter a&b"; cin>>a; cin>>b;
    switch (op) {
    case 1:
        int c = cal.add(a,b);
        cout << "a+b" << c;
        break;
    case 2:
        int c = cal.sub(a,b)
        cout << "a-b": << c;
        break;
    default:
        cout << "Enter correct operation";
    return 0;
    }
    catch (Exception e)
    {
    }

try {
    if (b==0)
    throw b;
    else
    throw
```

→ try, catch block also used for testing along with exception handling

14/7/15

→ highest priority

<> → gives specific file → (search for file mode directly where 'c' is installed)

" " → gives just the path to a file.
↳ (search first mode in current folder and then where 'c' is installed)

hello.c

↓

1. Preprocessing

~~Compiling~~

↓ hello.i

2. Compiling

↓ hello.s (assembly code)

3. Assembling

↓ hello.o (object file)

4. Linking

↓ hello.exe (links & gives .exe file)

5. Loading

↓ hello

Data types

- primitive
  - char
  - int
  - long
  - double
  - float
- derived data type
  - → arrays, ptrs, structures, Unions
- user defined data type
  - → enum
  - → type modifier
  - → type qualifier

→ Int :- Int, short int, Long int, Long Long int

Data type	Size	Range
int	2	-2^{16} to $2^{16}-1$
char	1	-2^{8} to $2^{8}-1$

→ Type definition
- → Implicit casting
- → Explicit casting

typedef int awesum;

int no; ⇒ awesum no;

enum week { sun, Mon, Tue, Wed, Thu, Fri, Sat };
↳ we cannot change enum values

```
void main() {
    enum week today = Tue;
    today = Tue;
}
printf("%d", today+1);
```

enum { i=10, j=20, k=30 }
↓
'k' cannot be changed

formatted i/p o/p func :- printf(), scanf();

unformatted i/p o/p gets() ; puts();

getch() → directly displays main screen without displaying character on console

getche() → displays character on console & direct to main screen

Q

a = --2;

Right to left Associativity (only for increment/decrement)

x = (a++) + (++a);

a = 20 (21) (22)

x = 43

Q

main(){
int i=10, j=2, k=0, m;
m = ++i || ++j && ++k;
printf("%d %d %d %d %d", i, j, k, m)
}

Ans:- 11, 2, 0, 1

Sol:- when handling with '||' (or) '&&'
if we get first '1' in '||', then it will not compute remaining statement. or/ if we take '&&' - if 0.

```
printf("%d %d", ++i, ++i);
```

```
int a = {1,2,3,4,5}
int x = 1,2,3,4,5
printf("%d, %d", a, x)
```

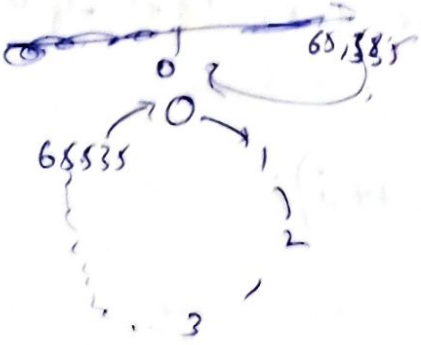
a = 5
x = 1

```
int x,a = 1,2,3,5;
```
error

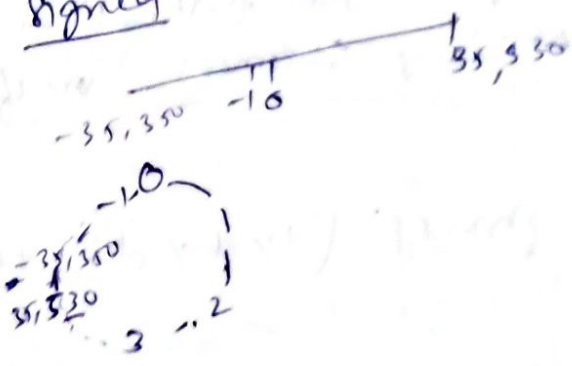
Right shift operations of negative no.

10 (hexadecimal)
= 0000 0000 0000 1010
= -10
1111 1111 1111 1010
after right shift vacant bits will be filled by '1's.

Unsigned



Signed



unsigned a = -1;

⇒ a = 65,535

Q

```
if (1, 2, 3, 5, 0)
print ("hai");
else
print ("hello");
```

a - 97 ↘ -32

A - 65

Q.

```
printf ("%.%.%.%.");
```

o/p : %.%.

\r - carriage return

\b - backspace

\a - alarm

Q

Bitwise Operator

Q

```
print ("m ks");
print ("\b mi \a");
print ("\r ha \n");
```

ks

ksmi

hai

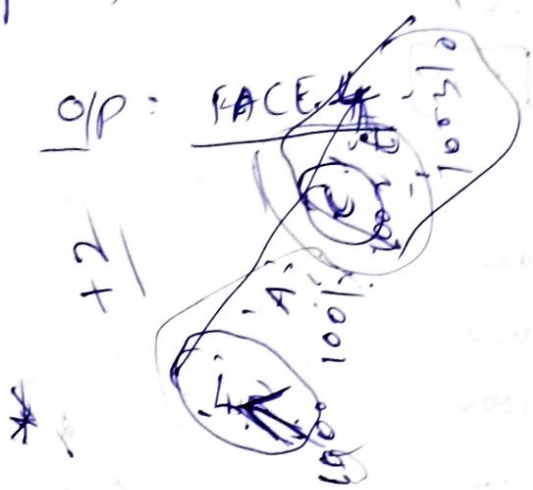
o/p:-

hai

9. printf("%d", printf("FACE")); → o/p FACE4

→ outer & inner printf

o/p: FACE4



10.

~~printf("%d", FACE~~

printf("FACE"+2); → only upto '7'

↓

ignore 1st two characters.

2.155

%2.3f → 2.100

%2.1f → 2.1

11.

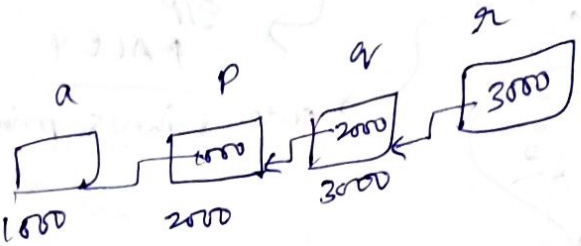
Arrays

int num[10];

int num[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9}

Pointers

int *p;



p = 1000
*p = 10
q = 2000
*q = 1000
**q = 10

r = 3000
*r = 2000
**r = 1000
***r = 10

d = 3
(*a)++
a++;

int *arr[10] → ptr to array

int (*arr)[10] → array of pointers

(int *) func()
{
}

int n = 20;
int a[n] = {1,2,3,4} X error

only we have to use macros
#define n 20;

Q)

a[2] = 2[a]

int a [n] = { 3 };

Q 4 + ~~#~~1 + 1 + 1 = 7

Q int a [3] [2] = { {1,2}, {3,4}, {5,6} };

```
1 2
3 4
5 (6)
```

*(a+2)

printf ("%d, %d, %d", a [2][1], *(a[2]+1), *(*(a+2)+1));

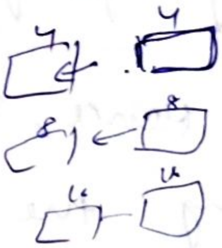
Q

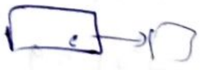
int arr2D [3] [3] = {1, 2, 3, 4, 5, 6};

printf (" ", arr2D, *arr2D, *arr2D[0]);

Q

printf ("%d", *&*p);





Bleak Number

The following numbers that are bleak, comes in a series of numbers

1, 4, 6, 13, 15, 18, 21, 23, 30, 32, 37, 39, 46..

This particular series are called Binary Self (or) Colombian ~~Series~~ Numbers.

Numbers that cannot be expressed as the sum of distinct terms of the form (2^k+1) $(k>=0$ (or) equivalently, numbers not of form. (m + sum of binary digits of m)

Hence, this plan can be solved either by checking the 2^k+1 (or) m + sum of binary digits of m

2^k+1

$2^0+1 = 2$

$2^1+1 = 3$

$2^2+1 = 5$

$2^3+1 = 9$

.

.

'4' cannot be formed
'6' cannot be formed } bleak

```
Static bool IsBleak(int m)
{
    int len = 0;
    while ((2^len + len) < m)
        len++;
    for (int i = len; i >= 0; i--)
        if (m >= (2^i + 1))
            m = m - (2^i + 1)
    return m != 0;
}
```

Dynamic Programming

→ Using the solution which is already solved by not solving it again

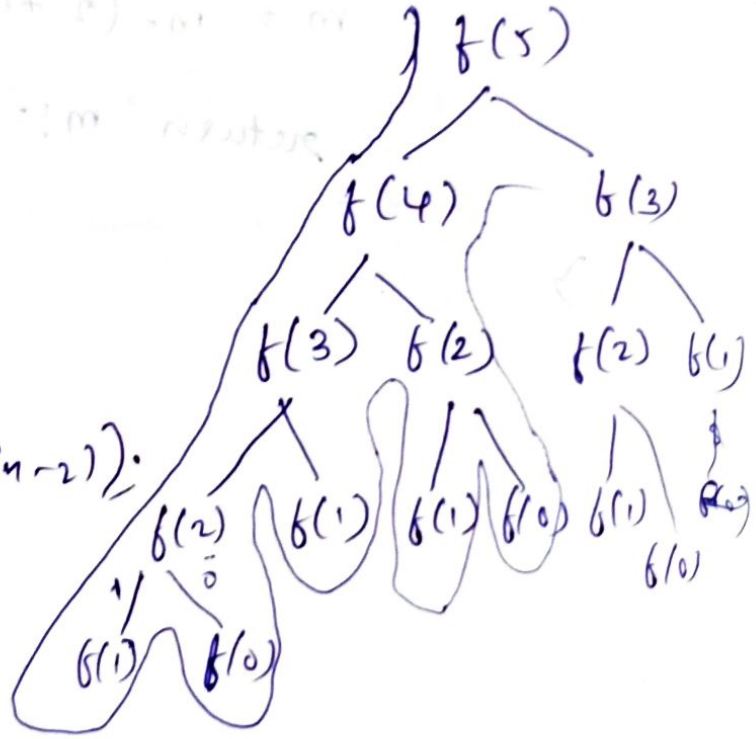
Ex: fibonacci series

$$f(n) = f(n-1) + f(n-2) \quad ; \text{otherwise}$$

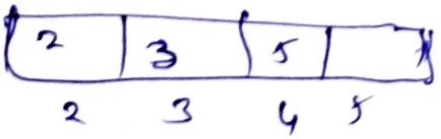
$$= 1 \quad ; n == 1$$

$$= 0 \quad ; n == 0$$

```
f(n)
    if (n==0)
      return 0;
    if (n==1)
      return 1;
    return (f(n-1)+f(n-2));
}
```



Check the table



for T(n)

if T(n) is empty

f(n)

if T(n) is empty

T(n) = f(n)

Introduction to LCS

(longest common subsequence)

Substring ≠ Subsequence

RAVINDRA
1 2 3 4 5 6 7 8

Ex: Substring

i) should be contiguous like Rav, vin, dra, ra, av...

Subsequence

2) must not be contiguous but indices should be in ascending order

Ex: RVN ↗ {1, 3, 5}, {1, 4, 6}

{3, 6, 8} VDA

{1, 8, 5} ✗ RAN

3) if m. length of string, then no of subsequences = 2^m

DNA Example:-

→ longest common subsequence, applied in finding similarities b/w DNA's

DNA components (A, C, G, T)

Ex: if D1 = AGCCTCAGT

D2 = GCCT

if (D1 == D2) then they are of same person or very much similar person

D2 is substring of D1 they are like much similar

if D2 not a substring of D1 then we try to find longest common subsequence (LCS) to measure similarity.

Ex: S1: Ravindra

S2: Ajay

subsequences = { d3, A, AA }

longest common subsequence = {AA} & length = 2

A = RAVINDRA

B = AJAY

Brute force Approach to find subsequences

time complexity:

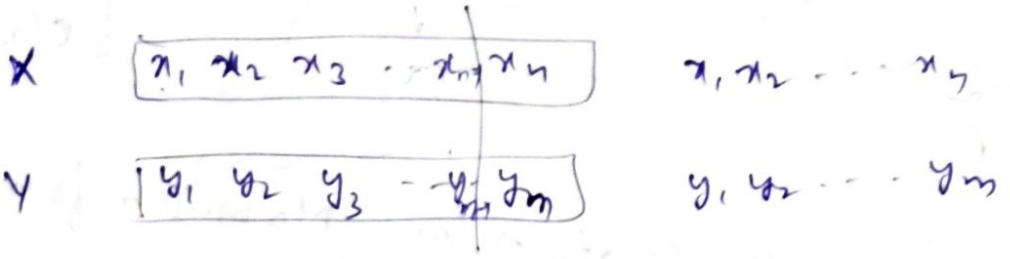
$O(2^m)$ — 1. Find all subsequences of 'A'

$O(n \cdot 2^m)$ — 2. Find for each subsequence whether it is a subseq of B'

$O(2^m)$ — 3 find the LCS

Optimal Substructure:

If the main problem is split into small problem, which can be part of main problem



$x_1, x_2, x_3, \cdots x_i$

$y_1, y_2, y_3, y_4 \cdots y_j$

$$C[i,j] = \begin{cases} 0 & ; \ i=0 \text{ or } j=0 \ (\text{if any of the string length is } 0) \\ 1 + C[i-1, j-1] & ; \ i,j>0 \text{ and } x_i = y_j \\ \max(C[i-1,j], C[i,j-1]) & ; \ i,j>0 \text{ and } x_i \neq y_j \end{cases}$$

($C[i,j]$ = length of LCS)

Recursion Tree & Unique Sub Problems

case i) if input DNA's of same length: (Best Case) & match found

$X = \{A, A, A, A\}$

$Y = \{A, A, A, A\}$

Time Compl:

$O(n)$

Space Comp:

$O(n)$

equal length

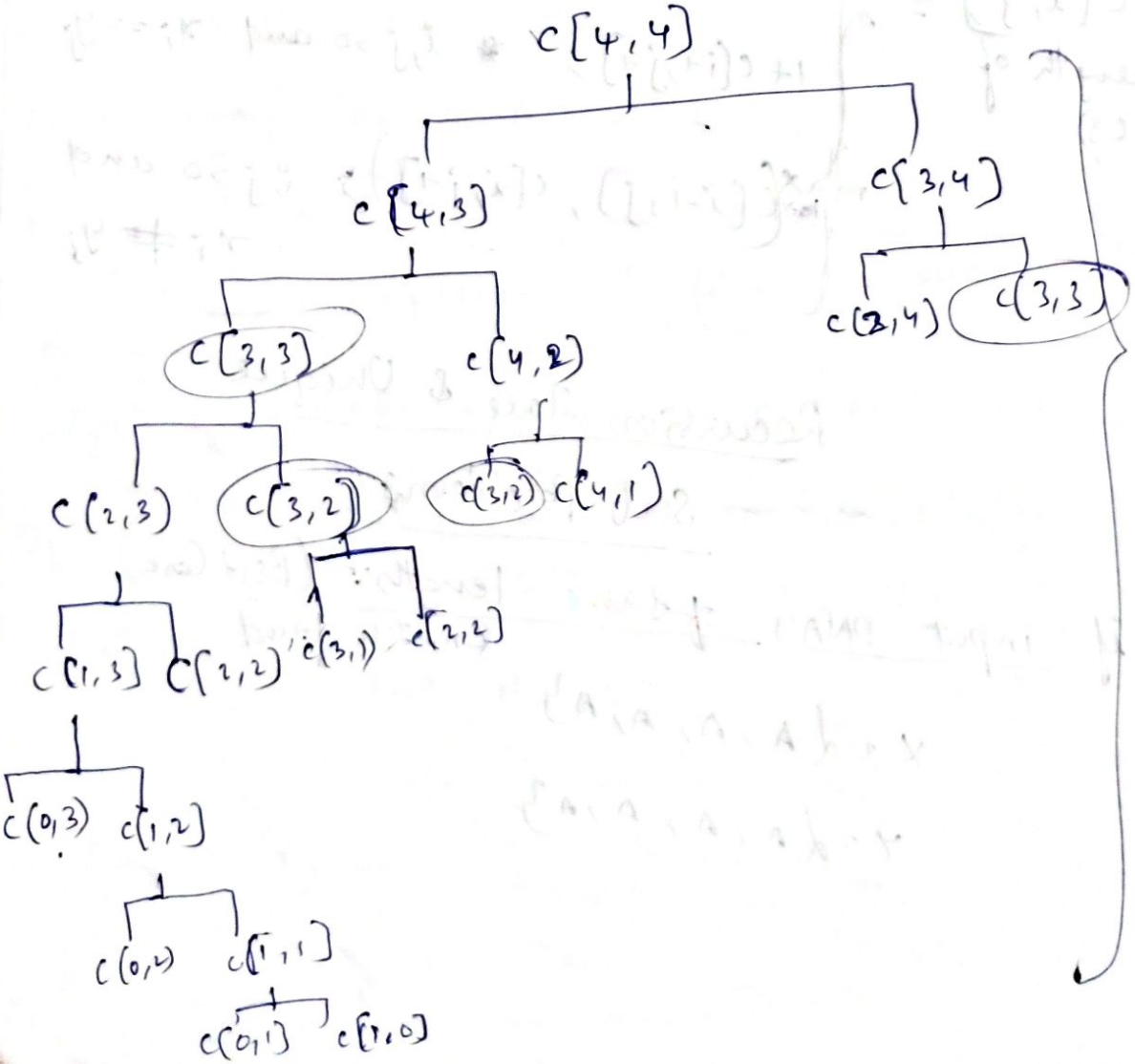
c(4,4)
|
1 + c(3,3)
|
1 + c(2,2)
|
1 + c(1,1)
|
c(0,0)

Case iii):

DNA's of diff length or same No match found:-

$X = \{A, A, A, A\}$ (1, 2, 3, 4)

$Y = \{B, B, B, B\}$ (1, 2, 3, 4)



```
c(n, m)
  |
c[n-1, m]
  |
c[n-1, m-1]
  |
c[n-2, m-1]
  |
c[n-2, m-2]
  .
  .
  .
c[0, 1]
  |
c[ , 0]
```

depth of tree

$O(n+m)$

If tree is half full

the no. of nodes

$= O(2^{n+m})$

* If 'n' length of first string, 'm' length of second string, then no. of unique solutions $= (n*m)$ [if we discard (0,1)(0,0)(1,0)]

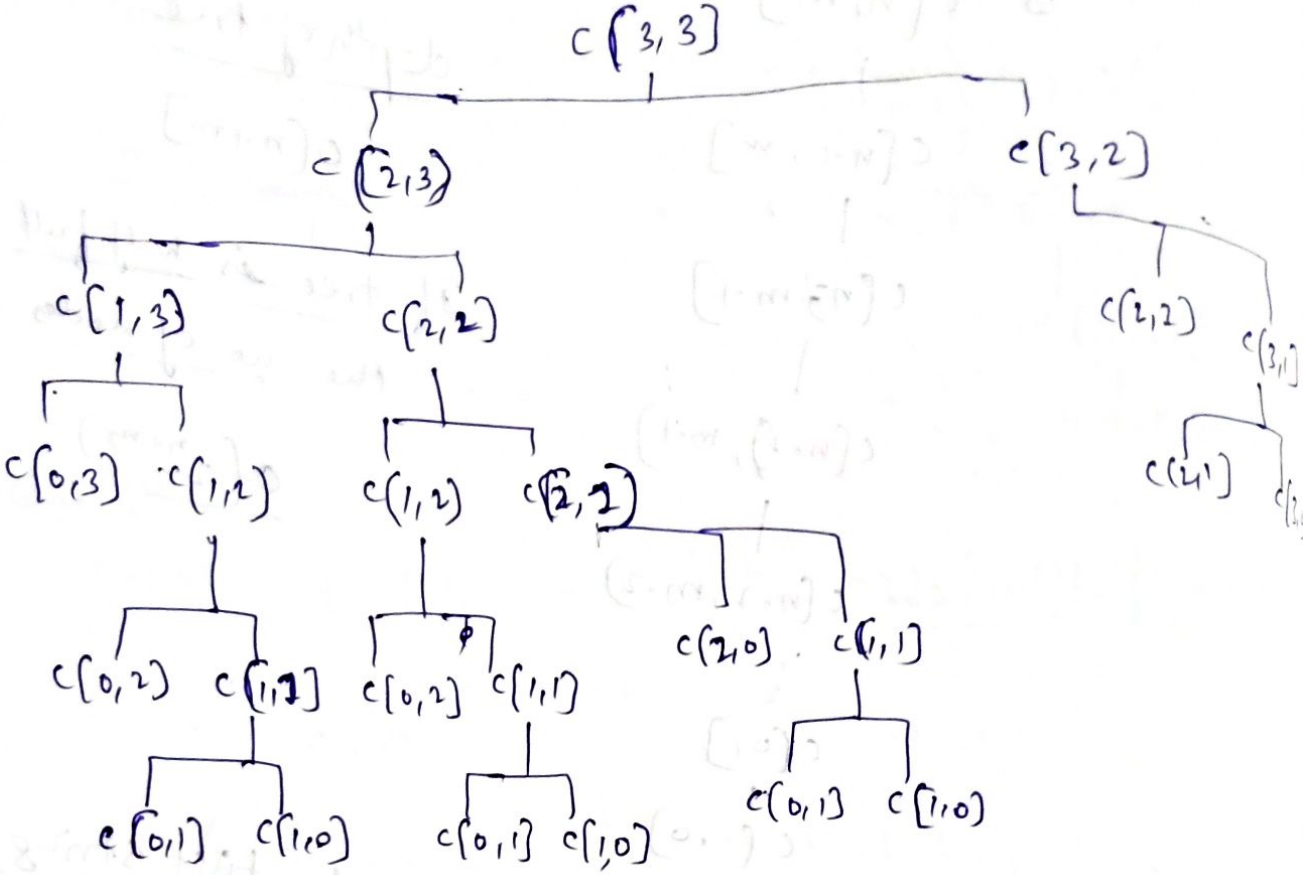
else $(n+1)*(m+1)$

ex.: (3,3)

=> unique solutions : $3*3 = 9$

(0,1)

$4*4 = 16$



unique solns

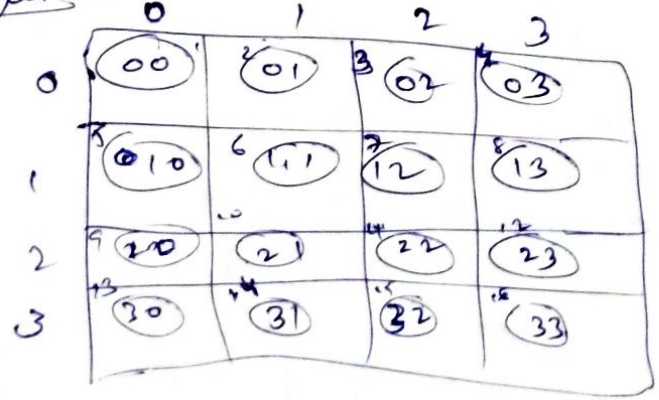
(3,3), (3,2), (3,1)
(2,3), (2,2), (2,1)
(1,3), (1,2), (1,1)
} 9

(1,0), (2,0), (3,0)
(0,1), (0,2), (0,3)
} 7

16

(0,0)

Cells required



Bottom-Up Approach: ✓ This is best.

Compute smaller problems first & then larger problems

Top-Down Approach:

Compute larger problem first & then smaller

→ Top-Down Memoization

→ same as D.P

Examples:

X : (A A B)

Y : (A C A)

		Y	0	A 1	C 2	A 3
X	0		0	0	0	0
A	1		0	1	1	1
A	2		0	1	1	2
B	3		0	1	1	2

if $x_i == y_j$: $1 + C[i-1, j-1]$

else $\max(C[i-1, j], C[i, j-1])$

Ex 2:

X = { A, B, C, B, D, A, B }
1 2 3 4 5 6 7

Y = { B, D, C, A, B, A }
1 2 3 4 5 6

Y \ X		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

① B C B A

② B D A B

③ B C A B

Matrix Chain Multiplication

Dynamic Programming

A (3×2) B (2×3)

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}_{3\times 2} \quad \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix}_{2\times 3}$$

$p \times q \qquad q \times r$

The no. of multiplications required

$$= \boxed{(p \times r) \times q}$$

$\Rightarrow (3\times 3) \times 2$ → col/n index

↓ no. of elements

Chain Multiplication

① $((A_{2\times 1}\ B_{1\times 2})\ C_{2\times 4})$

$(D_{2\times 2})$

multiplications $= (1\times 2\times 1) + (2\times 4\times 2) = 4 + 16 \Rightarrow$ 20 multiplications

② $(A_{2\times 1}\ (B_{1\times 2}\ C_{2\times 4}))$

$D_{1\times 4}$

$(2\times 4\times 1) + (1\times 4)\times 2 = 8 + 8$

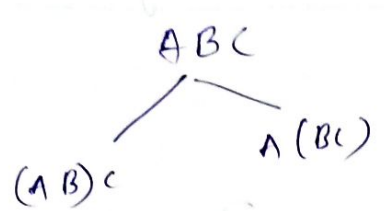
16 multiplications

In ② approach, no. of multiplications are reduced

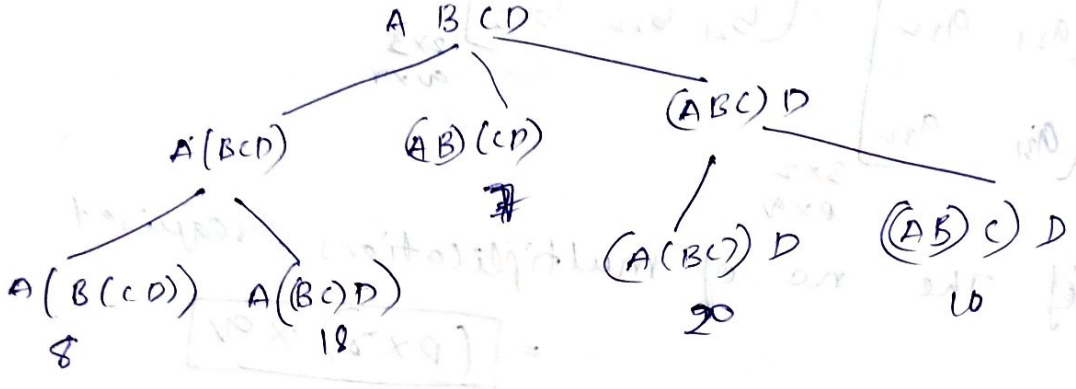
so, if we take substantially large dimensions,

it will be more effective in operations.

Examples:

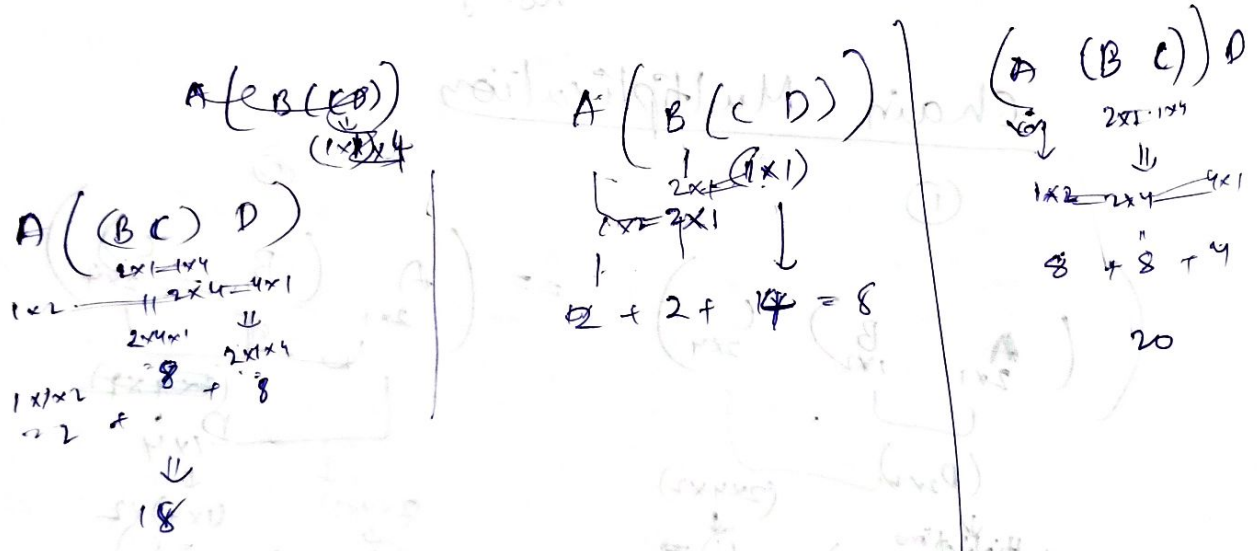


3 - matrices - 2 ways



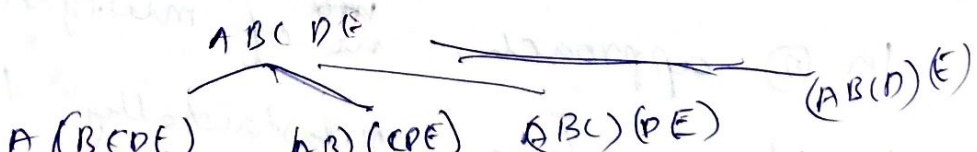
Ex..

$A_{1\times2}$ $B_{2\times1}$ $C_{1\times4}$ $D_{4\times1}$



4 matrices → 5 ways

5 matrices → 14 ways



$\frac{2n!}{(n+1)!\,n!}$ if 4 matrices $n = 3$

Optimal Substructure & Recursive Equations

Aim:- To paranthesise in such a way that minimum no of multiplications are required

matrices :- A_1 A_2 A_3 A_4 A_5 ..... A_n

dimensions of matrices : $P_0 \times P_1$ $P_1 \times P_2$ $P_2 \times P_3$ $P_3 \times P_4$ $P_4 \times P_5$ ..... $P_{n-1} \times P_n$

$A_i \; (P_{i-1} \times P_i)$

$A_i \; A_{i+1} \; A_{i+2} \; \cdots \; A_j = (A_i \; A_{i+1} \cdots A_k)(A_{k+1} \cdots A_j)$

$(i \le k \le j)$

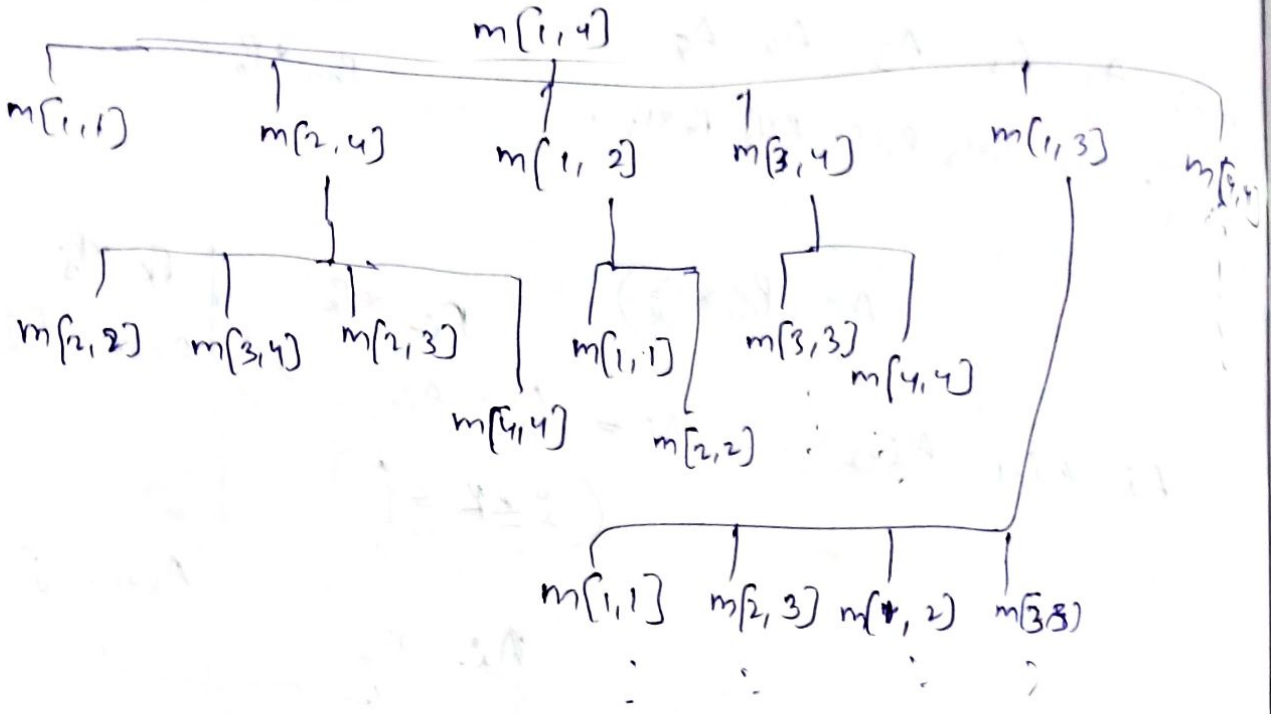
$P_{i-1} \times P_k$: $A_i \cdots k$

$P_k \times P_j$: $A_{k+1} \cdots j$

Recursive Equation

$$m[i,j] = \begin{cases} 0 & ; i = j \\ \min\limits_{i \le k \le j} \left(\underbrace{m[i,k]}_{P_{i-1} \times P_k} + \underbrace{m[k+1,j]}_{P_k \times P_j} + (P_{i-1} \cdot P_k \cdot P_j) \right) \end{cases}$$

Recursive Tree



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

The problems are repeating, so no need to compute repeatedly

D-P Bottom Up Implementation:

(1 1) ①	(2 2) ⓪	(3 3) ⓪	(4 4) ⓪	– 4	1
(1 2) ②	(2 3) ⑧	(3 4) ④		– 3	2
(1 3) ④	(2 4) ⑥			– 2	3
(1 4) ⑦				– 1	4

Subproblems

$$(1,3) = \min \begin{cases} (1,1) + (2,3) + P_0 P_1 P_3 = 16 \\ \quad 0 \quad\quad 8 \quad\quad 8 \\ (1,2) + (3,3) + P_0 P_2 P_3 = 4 \\ \quad 2 \quad\quad 0 \quad\quad 2 \end{cases}$$

$O(n^2)\, O(n) = O(n^3)$

n	1
$(n-1)$	2
$(n-2)$	3
.	.
.	.
1	n

$n + (n-1) + (n-2) + \cdots 1$

$\frac{n(n+1)}{2} \quad O(n^2)$

Bottom-up Dynamic Programming Algorithm

```
LCS (x, y)
{
    m = x.length;
    n = y.length;
    Let c[0...m, 0...n] be a new table
    for i = 0 to m
        c[i, 0] = 0
    for j = 0 to n
        c[0, j] = 0
    for i = 1 to m {
        for j = 1 to n {
            if x_i == y_i
                c[i, j] = c[i-1, j-1] + 1
            else
                c[i, j] = max(c[i-1, j], c[i, j-1])
        }
    }
    return c;
}
```

Time Comp: $O(m \times n)$

Space Comp: $O(n^2)$