Master Theorem:-

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

$a \geq 1$, $b > 1$, $f(n)$ → +ve

Three Cases:

① If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ (very small no), $T(n) = \Theta\left(n^{\log_b a}\right)$

② If $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$, $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$

③ If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ & $f(n) \to RC$

$$a f\left(\frac{n}{b}\right) \leq c\, f(n), \quad c < 1$$

Ex: ① $T(n) = T(n/2) + n$ → 3rd Case → $T(n) = \Theta(n)$

② $T(n) = 2T(n/2) + n$ → 2nd Case → $T(n) = \Theta(n \log n)$ (Merge / Quick)

③ $T(n) = 2T(n/2) + n \log n$ → 2nd → $T(n) = \Theta(n \log^2 n)$

④ $T(n) = T(n/2) + 1$ → 2nd Case → Binary Search, $T(n) = \Theta(\log n)$

Practice: ① $T(n) = 3T(n/2) + n^2$

$a = 3, b = 2, f(n) = n^2$

$n^{\log_b a} \Rightarrow n^{\log_2 3} = n^{1.5}$

Case ① $n^2 = O(n^{1.5-\epsilon})$ not satisfied

Case ② $n^2 = \Theta(n^{1.5} \log^k n)$ not satisfied for any $k \geq 0$

Case ③ $n^2 = \Omega(n^{1.5+\epsilon})$ Satisfied for small +ve values of ϵ

RC: $af(n/b) \leq c\, f(n)$

$3\left(\frac{n^2}{4}\right) \leq c * n^2$

$0.75n^2 \leq cn^2$

$c < 1$

Satisfied for $c = 0.9999$

$\underline{T(n) = \Theta(n^2)}$

② $T(n) = 4T(n/2) + \log n$

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

$n^{\log_b a} = n^{\log_2 4} = n^2$

Case ① $\log n = O(n^{2-\epsilon})$ Satisfied for small +ve ϵ

$\underline{T(n) = \Theta(n^2)}$

③ $T(n) = 16T(n/4) + n!$ or 2^n

Case 1: $a = 16, b = 4, f(n) = n!$

$n^{\log_b a} = n^{\log_4 16} = \underline{n^2}$

$T(n) = \Theta(2^n)$
$T(n) = \Theta(n!)$

Case 1: $n! = O(n^{2-\epsilon})$ not satisfied

Case 2: $n! = O(n^2 \log^k n)$ not satisfied for any $K \geq 0$

Case 3: $n! = \Omega(n^{2+\epsilon})$ Satisfied for small values of ϵ

RC: $16 * (\frac{n}{4})! \leq c * n!$

$0.66 \left[\frac{2}{3}\right] n! \leq (c) * n!$ Satisfied

$c \rightarrow 0.8$

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

Practice Questions:—

① $T(n) = 7T(n/3) + n^2$ — 3rd $T(n) = O(n^2)$

② $T(n) = 64T(n/8) - n^2 \log n$ → MT can't be applied

③ $T(n) = \sqrt{2}T(n/2) + \log n$ → 1st Case $T(n) = O(\sqrt{n})$

④ $T(n) = 0.5T(n/2) + \frac{1}{n}$ → MT can't be applied

⑤ $T(n) = 3T(n/3) + \sqrt{n}$ → $T(n) = O(n)$

⑥ $T(n) = 2T(n/4) + n^{0.51}$ → $T(n) = O(n^{0.51})$

Question: form a RR for the following code & solve it using MI.

```
T(n)
{ if (n≤1) return 1;
  else return T(√n);
}
```

RR

$T(n) = T(\sqrt{n}) + 1$

$T(1) = 1$

$T(n) = T(\sqrt{n}) + 1$

Take $n = 2^m \Rightarrow m = \log n$

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

Assume $S(m) = T(2^m)$

$S(m/2) = T(2^{m/2})$?

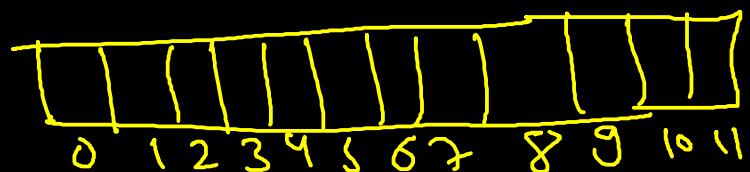
$S(m) = S(m/2) + 1$

$S(m) = O(\log m)$

$T(2^m) = O(\log m)$

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

① Linear Search:-



Time — Non-Sorted

Best – O(1)
Avg – O(n)
Worst – O(n)

Space – Auxiliary OR Extra Space that algo takes for its execution

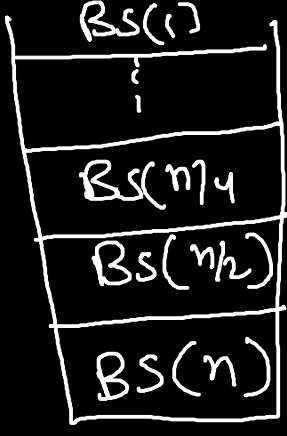
n = 100 → Extra vars. 2
n = 10^5 → " " 2

Constant Space – O(1)

Binary Search:-

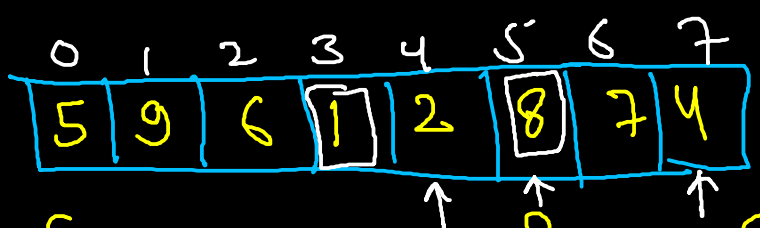
① Iterative :- Time – O(log n) Space – O(1)

② Recursive:- Time – O(log n) Space – O(log n)



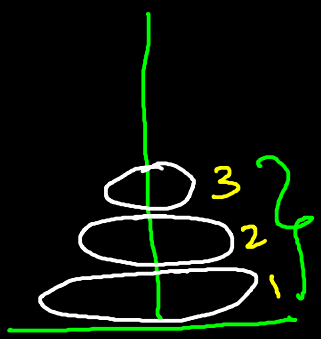
log n stack frames in worst Case

Que: If we apply Binary Search on a non-sorted array, Is it possible that it finds the key element?

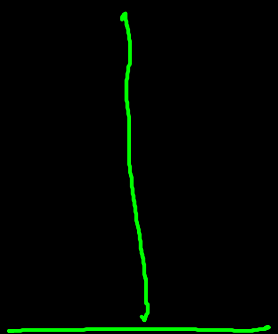


Can you find 4 using Binary Search?
If yes, in how many steps?
If no, then after how many steps algo determines that element is not present in the array?

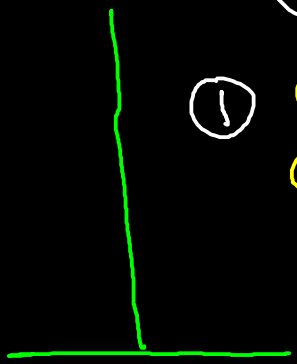
* Tower of Hanoi :-



Source



Aux. (Temp)

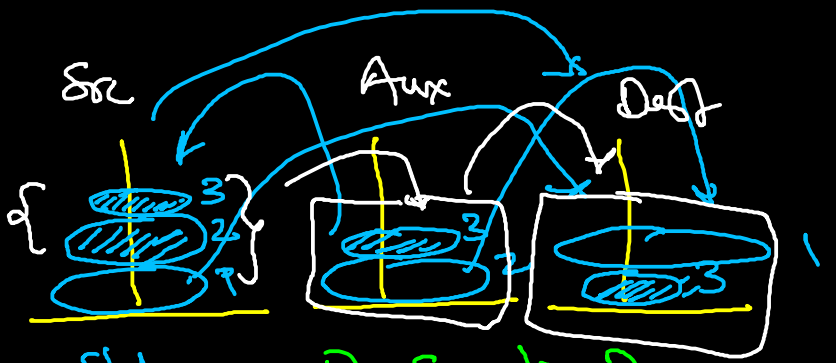


Dest.

Constraints:

① only one disc can be moved at a time.

② At any point of time the large disc should not be above smaller disc.



Steps

① S to D
② S to A
③ D to A
④ S to D
⑤ A to S
⑥ A to D
⑦ S to D

7 steps

3 disc — 7 steps (2^3-1)

4 discs — 15 steps (2^4-1)

n discs — (2^n-1)

Source Dest Aux

```
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod){
    if (n == 1){
        printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
        return;
    }
    towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
    towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}
```

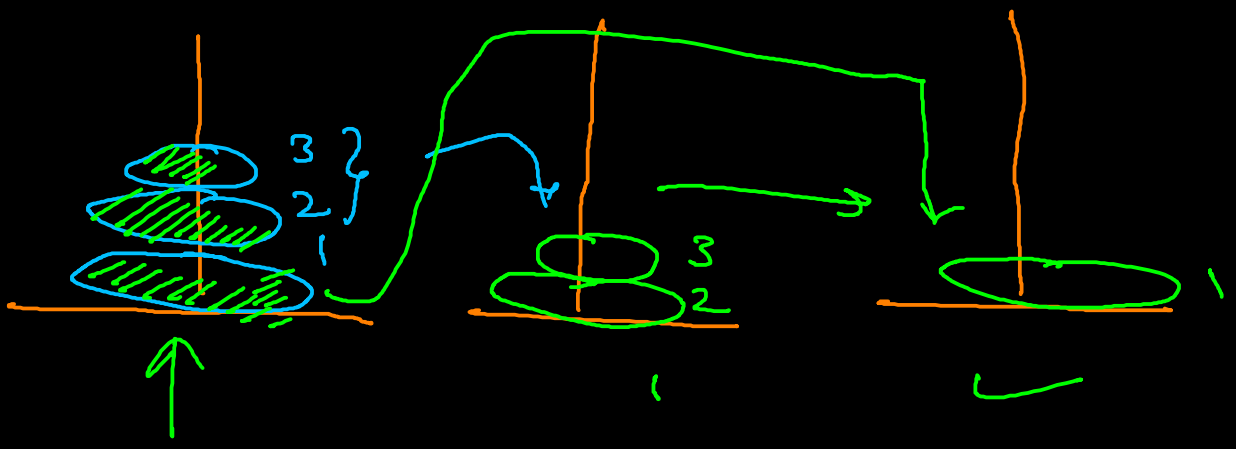
T(n)

O(1)

Src Dest Temp(Aux)

Src Dest

T(n-1) T(n-1)



ToH(3, '1', '2', '3')

ToH(2, '1', '3', '2')

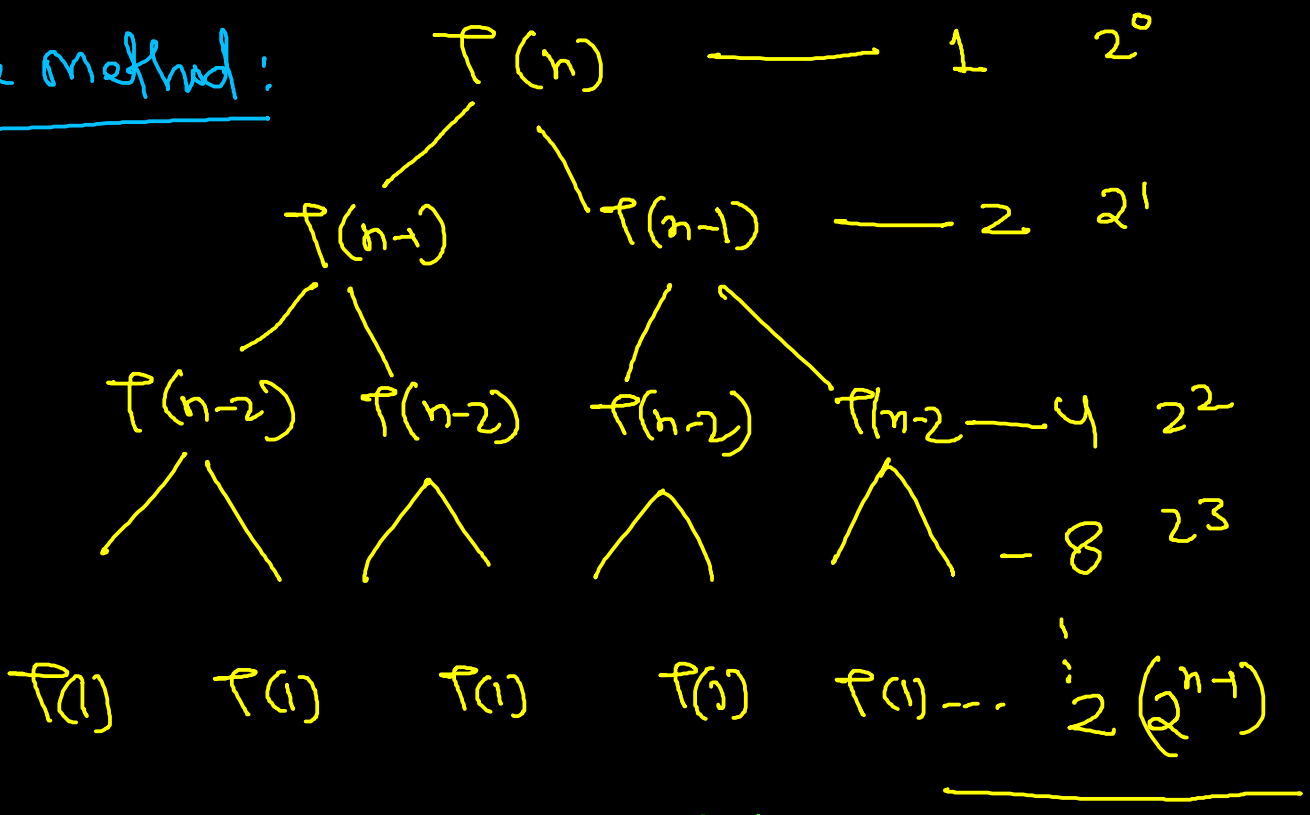
ToH(2, '2', '3', '1')

ToH(1, '1', '2', '3')

ToH(1, '2', '1', '3')

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

Tree method:



Qp: $1 + 2 + 4 + 8 + \cdots + 2^{n-1}$

$$\frac{a(r^n - 1)}{(r-1)}$$

$r = 2$
$a = 1$

Exponential

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

* Run Program for $n = 3, 4, 5$ & see how much time it takes.

Fibonacci Series:-

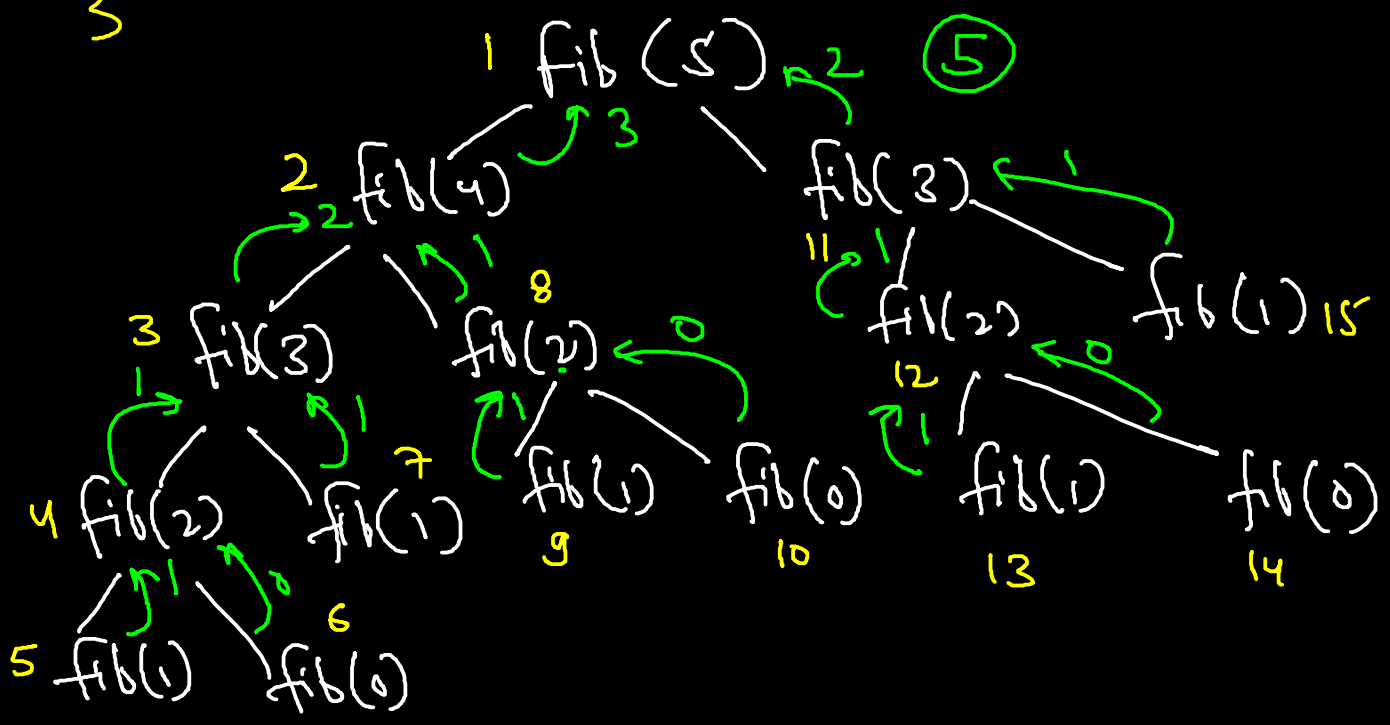
0	1	2	3	4	5	6	nth
0,	1,	1,	2,	3,	5,	8,	- - - - -

find the nth no. & fibo. Series:-

① Recursive ② iterative

```
int fib(int n)
{   if (n<=1) return n;
    return fib(n-1) + fib(n-2);
}
```

fib(5)



RR: $T(n) = T(n-1) + T(n-2) + 1$ — ①

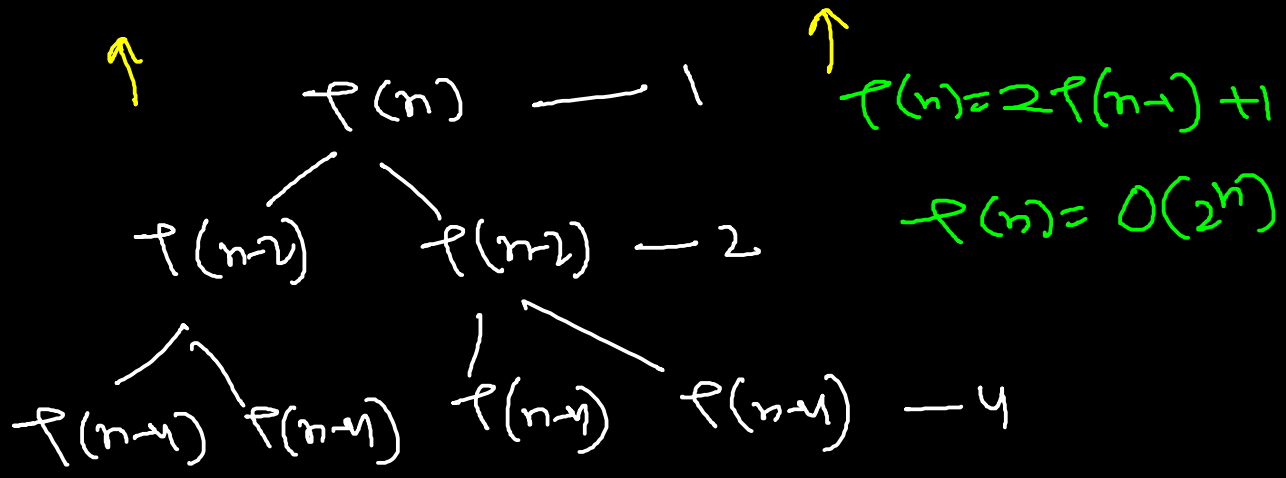
$T(n) = T(n-1) + T(n-1) + 1$ — ②

$T(n) = 2T(n-1) + 1$ — ②

$T(n) = T(n-2) + T(n-2) + 1$ — ③

$T(n) = 2T(n-2) + 1$ — ③

eqn(3) $\leq$ eqn(1) $\leq$ eqn(2)



$T(1)\quad T(1)\quad \cdots\quad T(1)$ — 2^{n-1}

$1+2+4+8+\cdots\cdots 2^{n-1}$

$n/2$

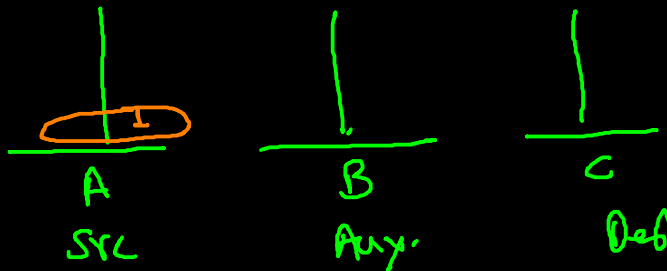
$\frac{a(r^k-1)}{r-1}$ $\qquad (2^{n/2}-1)$

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

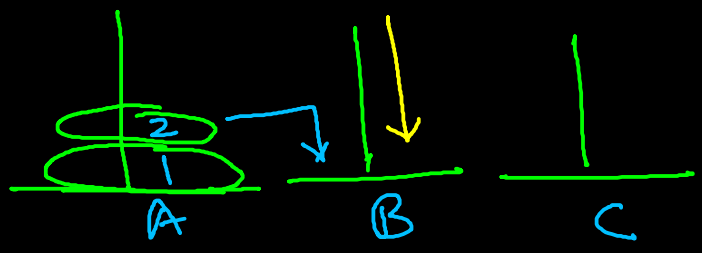
$T(n) = O(2^n)$

* Towers of Hanoi:-

* move a disc from A to C



* Two discs :-

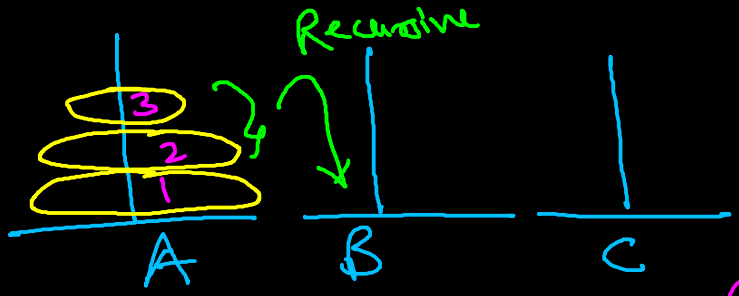


① move 1 disc from A to B using C

② move a disc from A to C

③ move 1 disc from B to C using A

* 3 discs



steps for 3 discs-

① move 2 discs from A to B using C

② move a disc from A to C

③ move 2 discs from B to C using A

* n discs :-

steps :- ① move (n-1) discs from A to B using C

② move a disk from A to C

③ move (n-1) discs from B to C using A

```
void TOH (int n, int A, int B, int C)
{   if (n > 0)
    {   TOH(n-1, A, C, B);
        printf("move a disk from A to C");
        TOH(n-1, B, A, C);
    }
}
```

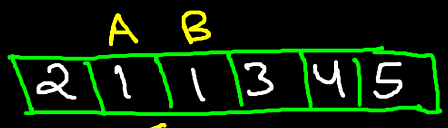
(Src, Aux, Dest)

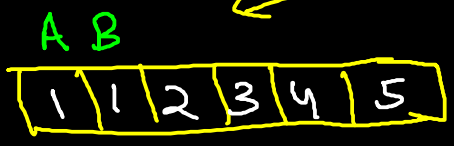
3 3

Unit-2 Sorting Algorithms

Sorting :- Arranging elements in to non-increasing or non-decreasing order.

Ex:





Non-decreasing order

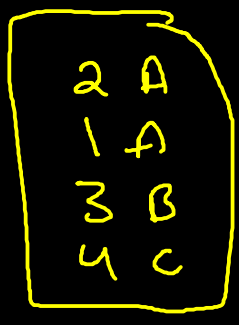
Non-increasing order

(1) Stable Sort – Relative order doesn't change

(2) Unstable Sort – Relative order might change
or
No guarantee to maintain relative order.

Ex: Bubble Sort, insertion Sort, selection Sort

↓ Stable ↓ Stable ↓ ?



Unstable Sort

③ Inplace Sorting:- Sorting algo that doesn't take extra space.

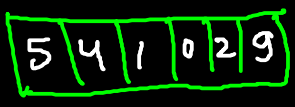
Ex: Bubble sort, Selection sort, Insertion sort

④ Online Sorting:- Sorting algo that doesn't need whole array before implementation starts. It can process elements one by one when elements arrive.

Ex: Insertion sort

Bubble sort ✗ Selection sort ✗

find max



If we get one element at a time we can still find max

⑤ Internal & External Sorting algos:-

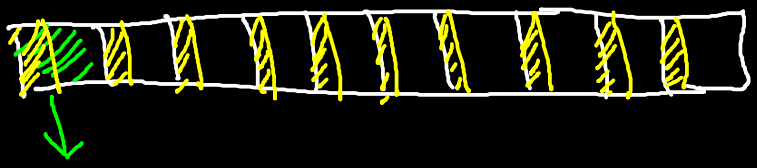
↓ Need whole array in m/m during execution

↓ Don't need whole array in m/m

↓ Physical m/m (RAM) Primary m/m

Ex: Suppose you have 1 GB RAM & 10 GB file having numbers which you need to sort. How you are going to do that?

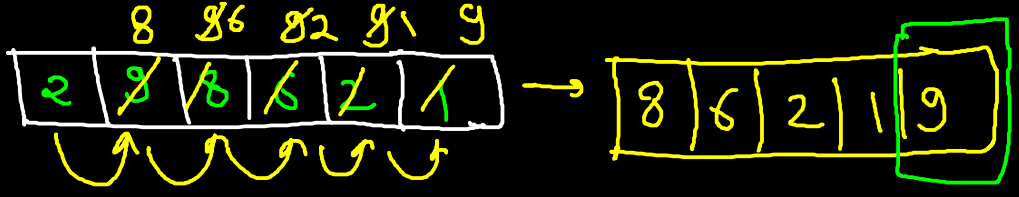
} Refer. Wikipedia Page for Exp.



Put it in RAM, Sort the array, (which algo?)
Put it back to HDD (virtual M(m)
Merge all sorted parts into one

* Bubble sort:- ① Compare & swap alternate elements if ith element is bigger than jth element where $i<j$

→ after 1st iteration ?



This element is at its Correct position

Algo:-

```
for i=1 to n-1
    for j=1 to n-i
        if (a[j] > a[j+1])
            swap(a[j], a[j+1]);
```

→

i=1, n-1
i=2, n-2
i=3, n-3
⋮
i=n-1, 1

$1+2+3+\cdots\cdots+(n-1)$

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

Space $= O(1)$

Loop invariant Condition :- A loop invariant is a condition that is necessarily true immediately before & immediately after each iteration of a loop.

Ex:
```
int j = 10;
for (int i=0; i<11; i++)
    j--;
```
$i+j == 10$

What is loop invariant for above code?

Ex: 54, 26, 93, 17, 77, 31, 44, 55, 20

what will be the array after 3 passes?

1st Pass 26, 54, 17, 77, 31, 44, 55, 20, 93

2nd Pass 26, 17, 54, 31, 44, 55, 20, 77, 93

3rd Pass 17, 26, 31, 44, 54, 20, 55, 77, 93

after 8th pass, this array will be sorted.

Bubble sort is stable sorting algo.

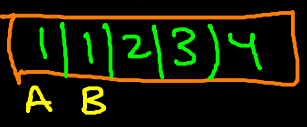
Ex: 4 | 1A | 1B | 3 → 1A, 1B, 3, 4

stable

Ex: Unsorted array

What can you say about this sorting algo?

↓ Sorting algo



(1) Algo is stable
(2) Algo is unstable
(3) Algo might be stable ✓
(4) Algo is not unstable

Insertion Sort:-

```
for j=2 to n
    key = a[j]
    i = j-1
    while i>0 and a[i]>key
        a[i+1] = a[i]
        i = i-1
    a[i+1] = key
```

n times

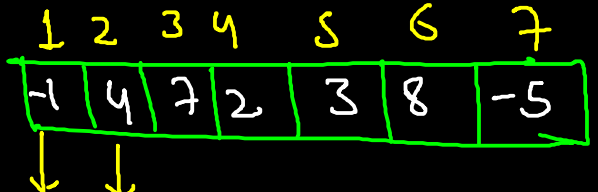
Space - O(1)

Time - O(n²)

j=2, 1
j=3, 2
j=4, 3
j=n, n-1

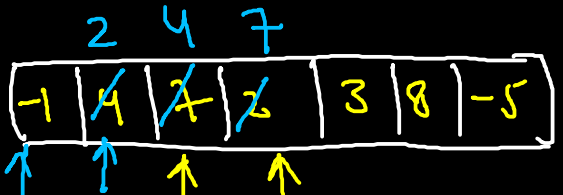
* Bubble sort have too many swaps
* insertion sort avoids too many swaps & exchanges elements only once after an iteration

Ex: ①



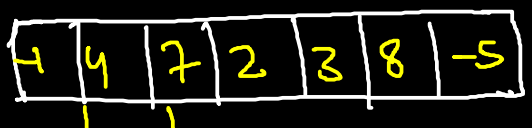
i=1 j=2 key=4

③



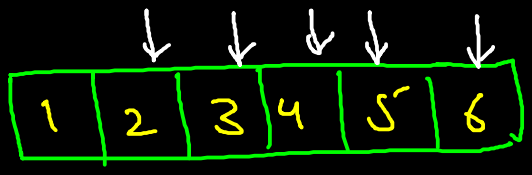
i=1 i=2 i=3 j=4 key=2

②



j=2 j=3 key=7 ✓

a[i+1] = key
a[1+1] = 2

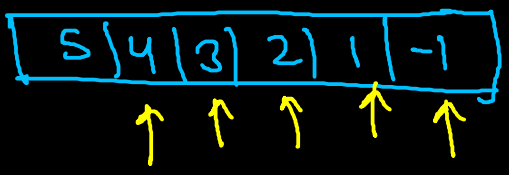
Ex: 

Sorted array

Sort this array using insertion sort.

Best Case – $O(n)$

* insertion sort works really well for nearly sorted array.

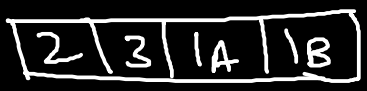
Ex: 

Sort this array using insertion sort.

Worst Case – $O(n^2)$

* Reverse Sorted array

* Insertion sort is stable sorting algo.

Ex: 

No change after 1st pass

after 2nd pass – 1A, 2, 3, 1B

after 3rd pass – 1A, 1B, 2, 3 — final Sorted array

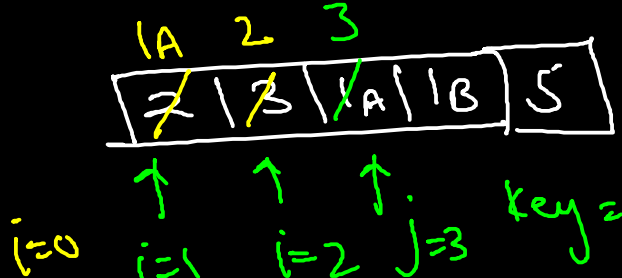
* Loop Invariant :- first i elements will be sorted after i^{th} iteration

* Insertion sort works well for sorted or nearly sorted array.

* Insertion sort — inplace — space O(1)

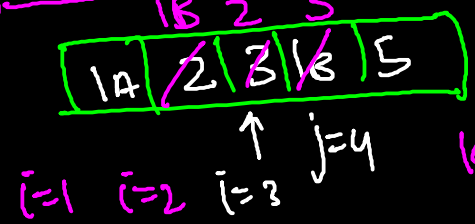
* stable? Yes

Ex: (1)



a[i+1] = key

→ i>0 and a[i] > key



a[i+1] = key

What if Condition is changed?

i>0 and a[i] >= key

Algo will become unstable

Selection Sort:-

```
for j=1 do n-1
    min = j                          O(1)
    for i = j+1 to n                 O(1)
        if a[i] < a[min]             O(1)
            min = i
    Swap( a[j], a[min] )             O(1)
```

j=1, (n-1)
j=2, (n-2)
j=3, (n-3)
:
j=n-1, 1

* It works by finding out min elements (1st min, 2nd min, 3rd min and so on) and arranging them in increasing order.

Time - $O(n^2)$ ↓ max Swapping - $O(n)$

Comparisons in Selection sort → $O(n^2)$

Comparisons — $n*(n-1)/2$

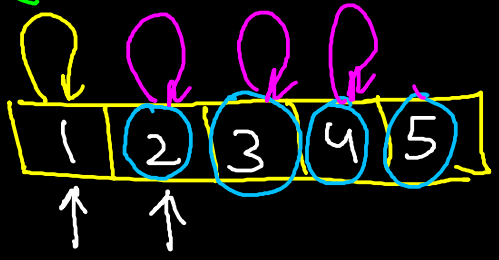
for n=5 $\frac{5 \times 4}{2} = 10$

Best Case - $O(n^2)$
Avg Case - $O(n^2)$
Worst Case - $O(n^2)$

↑ 4 Swapping

Sorted array

Example ①



min = 1

In this array Swapping will be done with same element. So this can be avoided by testing a condition.

Swap(a[i], a[i])
↓
if (j != min)
↓
then Swap

Comparisons :- 1st Pass - 4, 2nd Pass - 3
3rd Pass - 2, 4th Pass - 1 Total = 10

Ex:②

1	2	3	4	5
5	4	3	2	1

← min

Comparisons = 4+3+2+1 = 10

Swaps = 1+1+1+1 = 4

$j=1$ $i=2-5$, min = 1 ≠ 3, 4, 5

After 1st Pass — Swap(a[1], a[5])

1	2	3	4	5
1	4	3	2	5

Swap(a[2], a[4])

$j=2$, $i=3-5$, min = 2 ≠ 3 ≠ 4

1	2	3	4	5
1	2	3	4	5

$j=3$, $i=4-5$, min = 3

Inplace — Selection Sort is inplace as it takes no extra space O(1)

Stable ?

Ex:①

2	1A	3	1B	5

↓ Selection

1A	1B	2	3	5

Ex.①

3	4	2A	5	2B

↓ Selection

2A	2B	3	4	5

Ex.③

1_A	3_A	3_B	1_B

$j=1$ $i=2$ min = 1

No change after 1st Pass

→

1	2	3	4
1A	3A	3B	1B

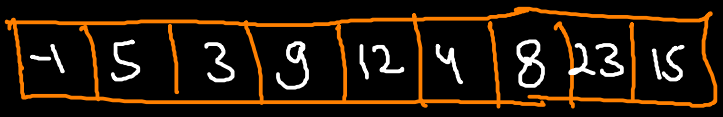
$j=2$ $i=3$ min = 2 ≠ 4

1A	1B	3B	3A

$j=3$, $i=4$, min = 3

Unstable

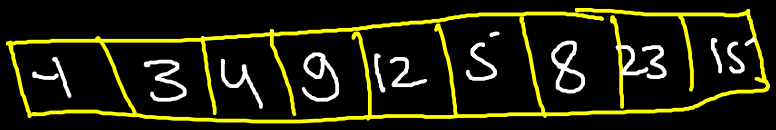
Ex: What will be the array after 3rd Pass.



1st Pass — No change after 1st Pass

2nd Pass — Exchange 5 & 3 — swap(a[2], a[3])

3rd Pass — Exchange 5 & 4 — swap(a[3], a[4])



Counting Sort:- Non. Comparison Based algo

Bubble, Selection, insertion, Quick, merge, Heap } Comparison Based algo

Assumptions:- Numbers to be sorted are integers in {0, 1, 2, ---k}

+ve integers in range (0-k)

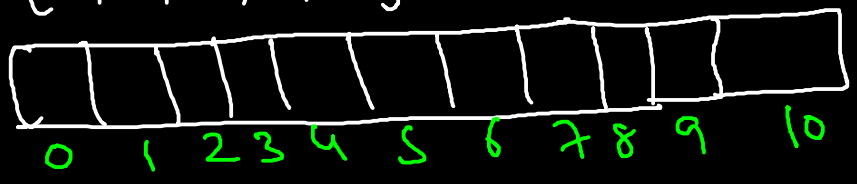
Input:- A[1--n] where A[j] ∈ {0, 1, 2, ---k} for j = 1, 2, --- n

Output:- B[1---n] sorted, B is assumed to be already allocated

Auxillary storage:- C[0---k]

Ex: A = [5, 1, 2, 3, 10]

B = [1, 2, 3, 5, 10]

C = 

Size of Auxiliary array (storage) depends on the highest no in the list of elements to be sorted.

* What if highest no is 10^6 & all other no. are very less than 10^6

Ex: 

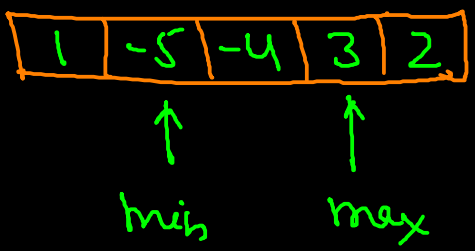
Sort this array using Counting sort.

for this you need 10^6 extra space

Space — $O(n+k)$

This algo is efficient only if $n \approx k$

* This algo only works for +ve integers so you can't use it for sorting negative no.

Ex: 

Counting_Sort (A, B, n, k)

let C[0..k] be the new array

```
for i = 0 to k                    } O(k)
    C[i] = 0
for j = 1 to n                    } O(n)
    C[A[j]] = C[A[j]] + 1
for i = 1 to k                    } O(k)
    C[i] = C[i] + C[i-1]
for j = n downto 1                } O(n)
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Time = $O(n+k)$

Time = $O(n)$ if $n \geq k$

Ex:

0	1	2	3	4	5	6	7
2_A	5	3_A	0_A	2_B	3_B	0_B	3_C

max = 5

C

2	0	2	3	0	1
0	1	2	3	4	5

⇓

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

This array shows Cumulative sum, where each element shows how many no. are less than or equal to numbers at that index.

A shows that there a 4 no. which less than or equal to 2.

B

0_A	0_B	2_A	2_B	3_A	3_B	3_C	5
1	2	3	4	5	6	7	8

C

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

~~1~~0 ~~3~~2 ~~6~~4 7

$B[C[A[6]]]$

$B[2] = \underline{A[6]}$

0

* $B[C[A[5]]] = A[5]$

3

3_B

$B[6]$

* $B[C[A[3]]] = A[3]$

0

$\uparrow 0$

1

$\underline{B[1] = 0}$

* Counting sort is stable sorting algo.
* Is it inplace ? No.

Quick Sort:

- Based on divide & Conquer strategy.

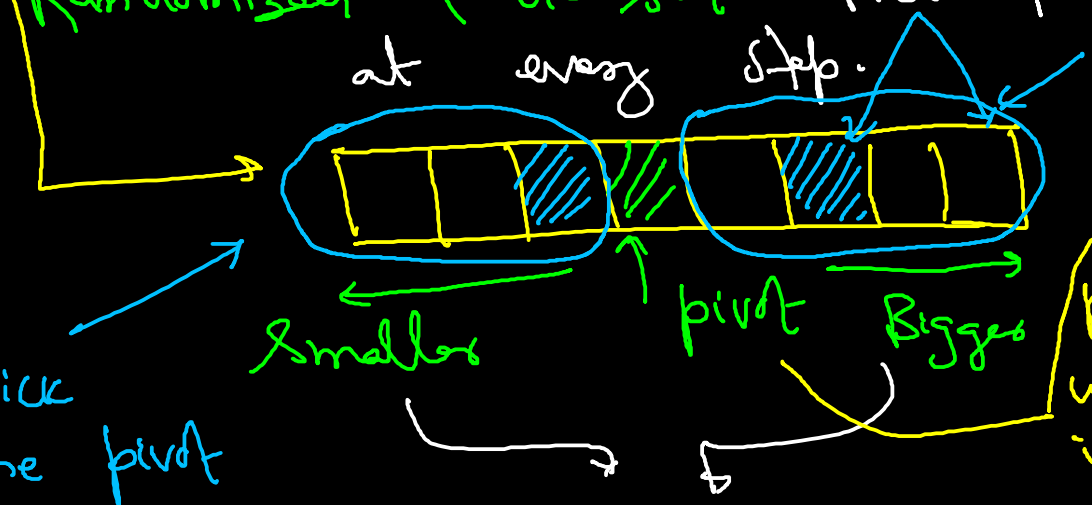
- A pivot(x)-element is picked & array is partitioned with respect to the pivot element i.e. elements smaller than x will be in one partition and elements greater than x will be in another partition.

- Which element could be pivot element?

  Any element of the array.

  But right now we will take last element as pivot element.

Randomized quick sort - Pick pivot randomly at every step.



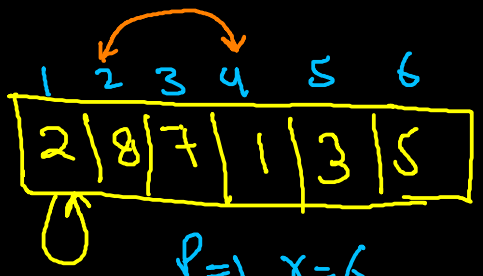
pick one pivot & sort this partition

These parts are still to be sorted

QuickSort(A, p, r) → T(n)

```
if (p<r)
O(n)   q = partition(A,p,r)   // partition array & fix position of pivot element
T(k)   QuickSort(A, p, q-1)   → Call quicksort on left parts
T(n-k+1) QuickSort(A, q+1, r) → Call quicksort on right parts
```

```
Partition(A, p, r)
    x = A[r]   // choose pivot
    i = p-1
    for j = p to r-1
        if A[j] ≤ x        O(n)
            i = i+1
            Swap(A[i], A[j])
    Swap(A[i+1], A[r])
    return i+1
```



p=1, r=6

QuickSort(A, 1, 6)

Partition(A, 1, 6)

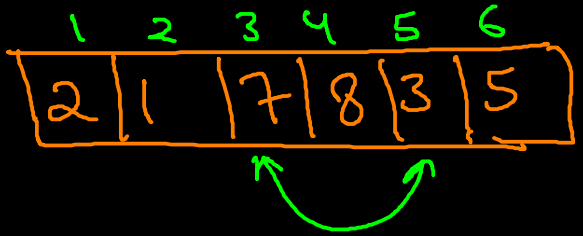
x=5, i=0

① { j=1 to 5
A[1] ≤ 5
i=1

② { j=2, A[2] ≤ 5
i=1

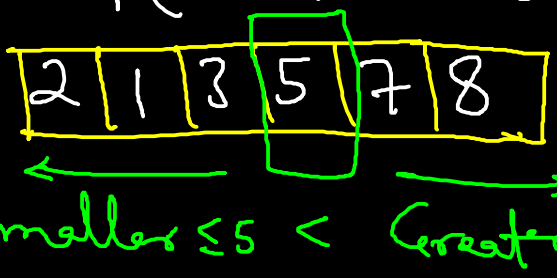
③ { j=3, A[3] ≤ 5
i=1

④ j=4, A[4] ≤ 5
i=2
Swap(A[2], A[4])



⑤ j=5, A[5] ≤ x
i=3
Swap(A[3], A[5])

⑥ Swap(A[4], A[6])

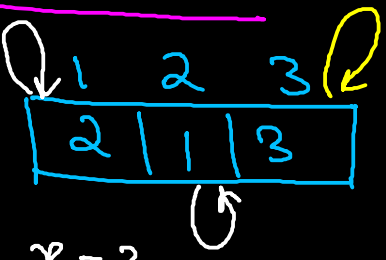


→ return 4

Smaller ≤ 5 < Greater

Next Steps

Apply QuicksaA(A, 1, 3)



x = 3

i = 0, j = 1 to 2

① A[j] ≤ 3

A[1] ≤ 3

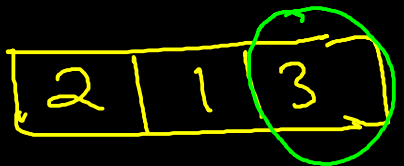
i = 1

② A[2] ≤ 3, i = 2

* Swap(A[i+1], A[r])

Swap(A[3], A[3])

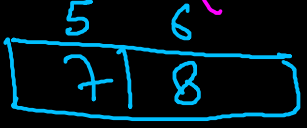
return 3



QuicksaA(A, 1, 2) → 1 2

QuicksaA(A, 4, 3)

QuicksaA(A, 5, 6)



↓

No change

Divide and Conquer:-

↓ Break problem into subproblems of same type → Recursively solve these problems

Combine – Combine the answers

Time Complexity Analysis:-

① $T(n) = T(k) + T(n-k+1) + O(n)$

② $T(n) = T(n/2) + T(n/2) + O(n)$ (Best Case)

③ $T(n) = T(n-1) + O(n)$ (Worst Case)

④ $T(n) = T(\frac{9n}{10}) + T(\frac{n}{10}) + O(n)$ (Algorithm)

Best Case: $T(n) = 2T(n/2) + O(n)$

MT: $f(n) = n$, $a=2, b=2$

$n^{\log_1 9} = n^{\log_2 2} = n$

2nd Case $f(n) = \Theta(n^{\log_b a} \log^k n)$

then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

$n = \Theta(n \log^k n)$ for $k=0$, it is satisfied

$T(n) = \Omega(n \log n)$ (Best Case)

Worst Case :- $T(n) = T(n-1) + O(n)$

$T(n) \longrightarrow n$

$T(n-1) \longrightarrow n-1$

$T(n-2) \longrightarrow n-2$

$T(1) \longrightarrow 1$

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

$+ 3 + 2 + 1$

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

Quick Sort has Complexity of $O(n^2)$ in worst Case.

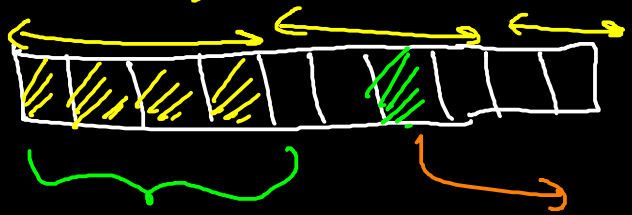
Still Qs is Considered best algo for practical purposes. Why?

Merge, Heap $\Rightarrow$ Avg, Worst, Best $\Rightarrow O(n \log n)$

① Efficient use of Cache

② Can be implemented in $O(n \log n)$ using randomized QS for almost all the cases and then it rarely achieve $O(n^2)$ complexity.

③ locality of reference



all these elements have chances to be accessed one after another

If not in m/m it will result into page fault

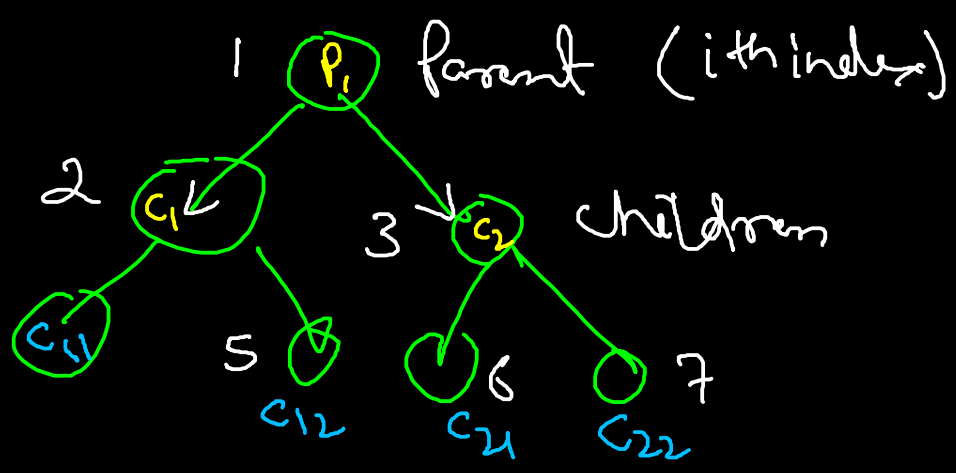
* If something is not found in m/m, it needs to be retrieved from HDD (virtual m/m) which takes around 1000 times time in accessing the data.

Ex: m/m access - 1ns

Page fault (HDD Access) - 1000 ns / 1 ms

HDD Access is very costly.

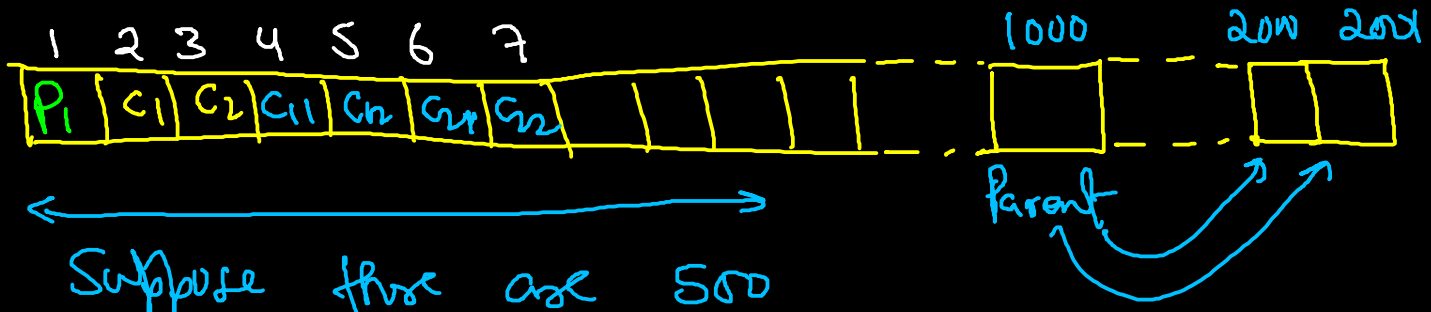
Heap Sort:



Parent index - i

Children index

$2i, 2i+1$



Suppose there are 500 elements in m/m, to access other elements Page fault will occur.

Ex:

	0	1	2	3
0	00	01	02	03
1	10			
2	20			
3	30			33

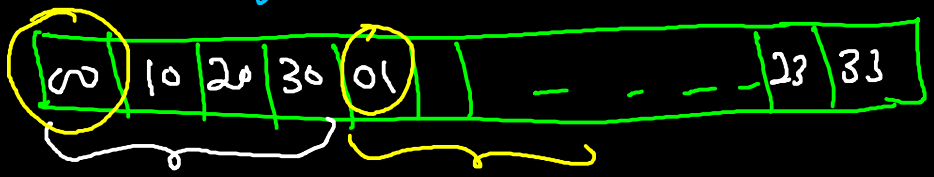
Row major

Column major

* Add all no. of methods —

If your programming lang. is implementing matrix (System) using Row major storage - which of the above method will be better or more efficient to add all nos?

* what if matrix is stored as Column major format?



* Run above example for a size of $10^6 \times 10^6$ matrix & compare the performance.

* $T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + n$

Solve above equ. using tree method and find out the time complexity of Quick Sort. Also find out the difference between heights of 2 extreme branches of the tree.

Merge Sort : Divide & Conquer Strategy

Divide - Divide n elements sequence to be sorted into subsequences of n/2 elements each

Conquer - Sort two subsequences recursively using merge sort

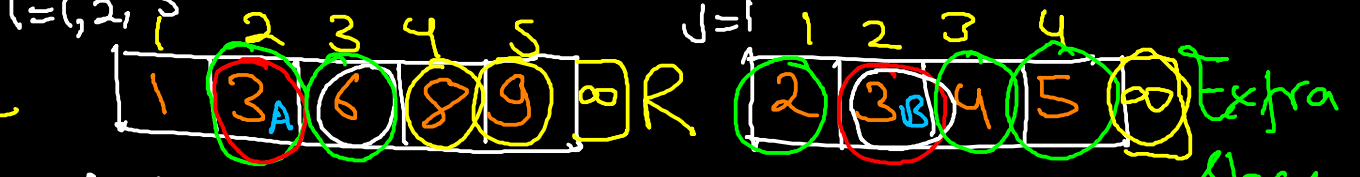
Combine - merge two sorted subsequences to produce the sorted answer.

Ex:

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

$p=1, r=9, mid = \left\lfloor \frac{p+r}{2} \right\rfloor = 5$

Take two temp arrays L & R and Consider that two subparts are sorted



Original Array

K=1	2	3	4	5	6	7	8	9
1	2	3A	3B	4	5	6	8	9

MergeSort (A, P, r) T(n)

if (p < r)

$q = \lfloor (p+r)/2 \rfloor$

→ MergeSort (A, P, q) → $T(n/2)$

→ MergeSort (A, q+1, r) → $T(n/2)$

Merge (A, P, q, r) → $O(n)$

$\begin{cases} L[i] \le R[j] \\ A[k] = L[i] \end{cases}$

Merge (A, p, q, r)

$n_1 = q - p + 1$

$n_2 = r - q$

let $L[1, 2, \ldots n_1+1]$ & $R[1, 2, \ldots n_2+1]$

for $i=1$ to n_1

$L[i] = A[p+i-1]$

for $j=1$ to n_2

$R[j] = A[q+j]$

$L[n_1+1] = \infty$

$R[n_2+1] = \infty$

$i=1, j=1$

for $k=p$ to r

if $(L[i] \leq R[j])$ $O(n)$

$A[k] = L[i]$

$i = i+1$

else $A[k] = R[j]$

$j++$

If last element is not ∞

while $(i \leq n_1)$

$A[k] = L[i]$

$i++$

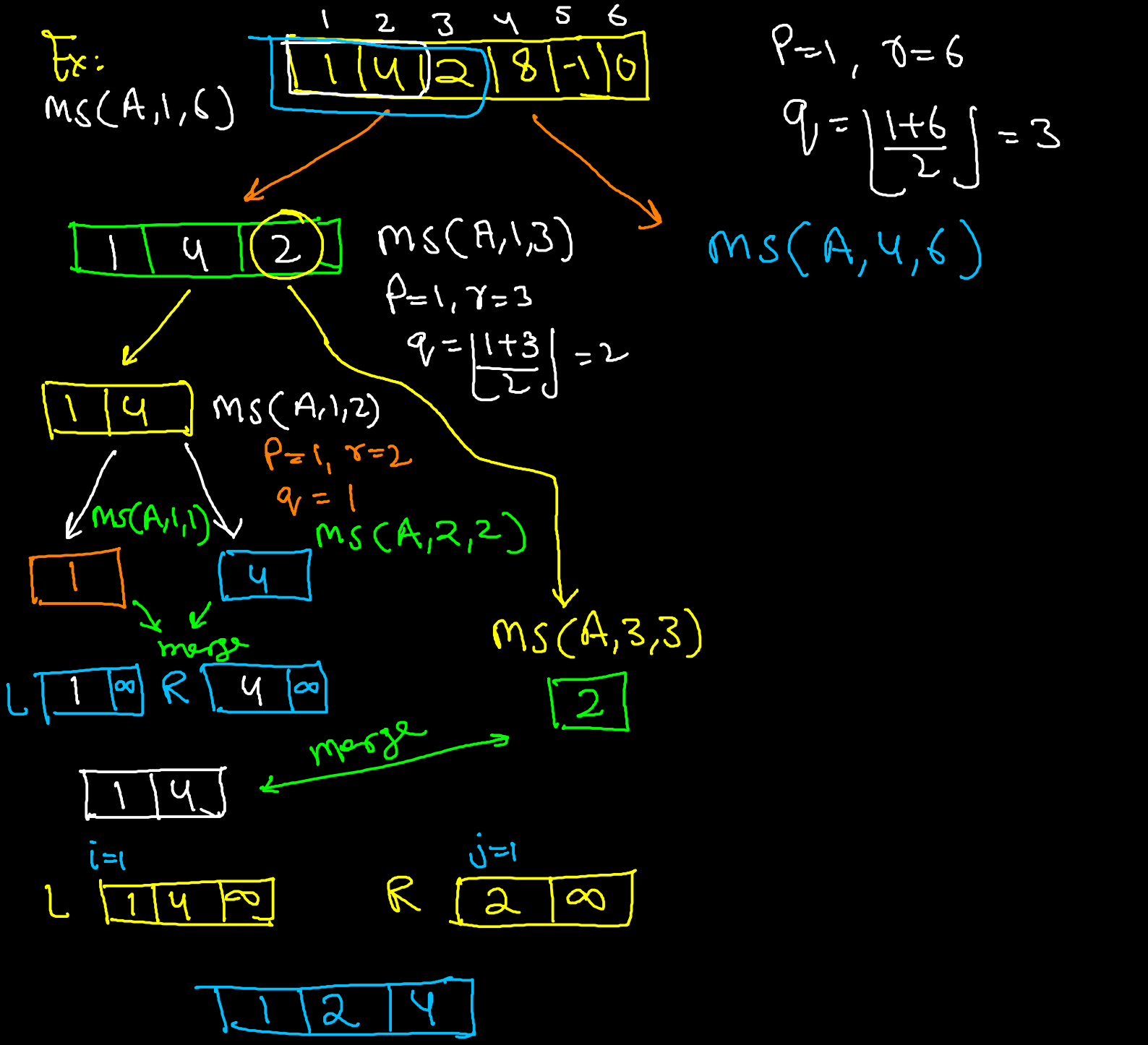
$k++$

* while $(j \leq n_2)$

$A[k] = R[j]$

$j++$

$k++$



Time Complexity: $T(n) = 2T(n/2) + n$

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

* Space – $O(n)$

Inplace – No.

* Stable – Yes

Quick Sort: Inplace – QS doesn't take any extra space for input manipulation & therefore it can be considered inplace But it takes extra space during recursive implementation & for that reason its not inplace.

Stable – NO.

Ex: $3_A, 7, 3_B, 1_A, 3_C, 1_B, 4, 1_C$

↓

$1_A, 7, 3_B, 3_A, 3_C, 1_B, 4, 1_C$

↓

$1_A, 1_B, 3_B, 3_A, 3_C, 7, 4, 1_C$

↓

$1_A, 1_B, 1_C, 3_A, 3_C, 7, 4, 3_B$

↓

$1_A, 1_B, 1_C, 3_A, 3_C, 3_B, 4, 7$ ← Unstable

Heap Sort:- It uses a data structure called Binary Heap.

What is a Heap (Binary Heap) — Complete binary tree whose items are stored in such a way that parent node is greater or smaller than values in its two child nodes.

- Max Heap
- Min Heap

* Parent at ith index

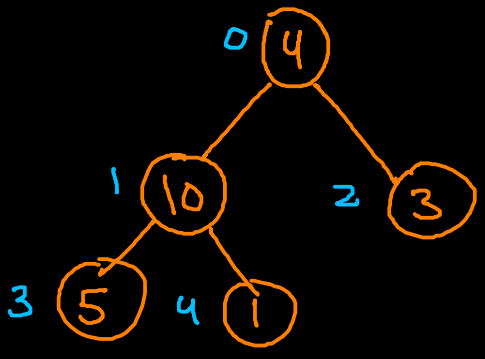
Left child at $2i+1$ → $2i$

Right child at $2i+2$ → $2i+1$

→ when index starts from zero

→ Index starts at One

0	1	2	3	4
4	10	3	5	1

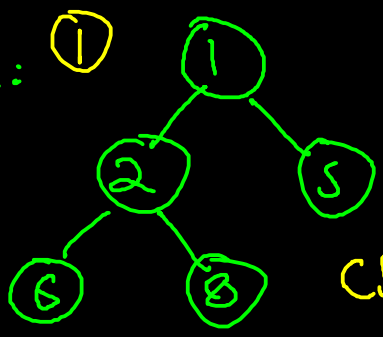


Is it Max or Min Heap?

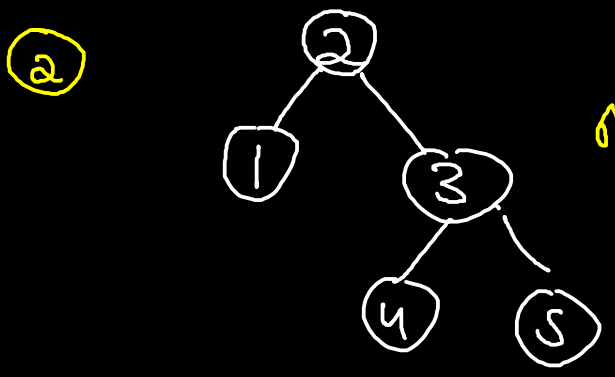
We can apply Heapify to make it a max or min heap.

Complete Binary Tree (CBT): A BT is CBT if all levels are completely filled except possibly the last level & has all keys as left as possible.

Ex:

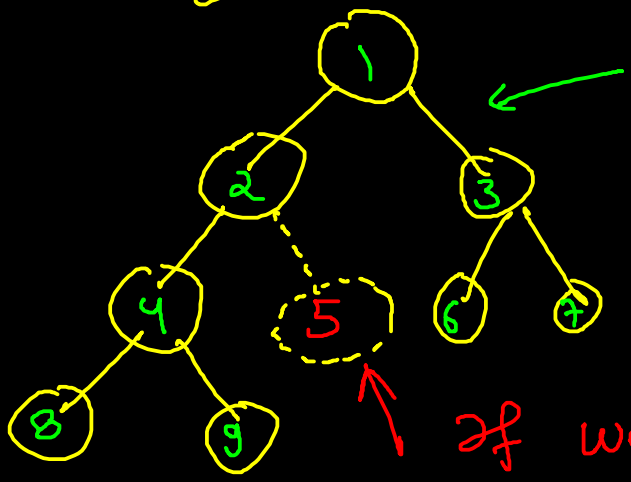


CBT ✓



No

Hint:

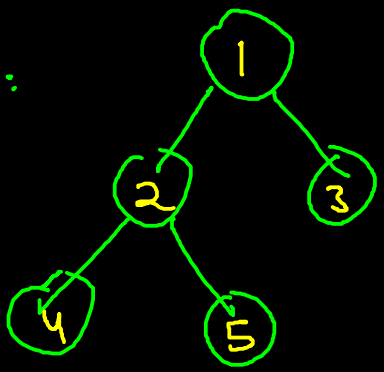


This is CBT if we have all the nodes including (5)

If we remove this node then its not CBT.

* Full Binary Tree :- A tree is FBT if every node has 0 or 2 children.

Ex:



FBT ✓

CBT ✓

* Which of the following is/are Correct?

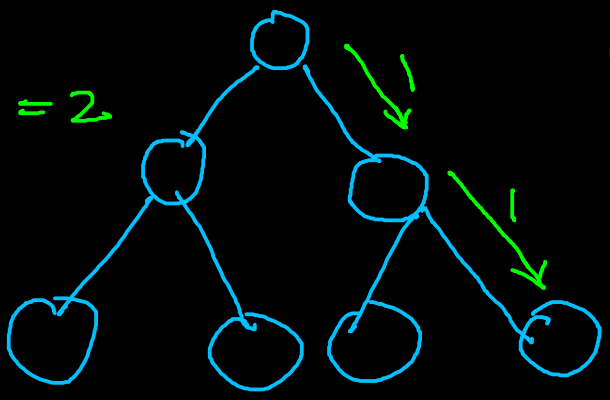
(1) All CBTs are FBTs — False

(2) All FBTs are CBTs — false

(3) All Max Heaps/Min heaps are CBTs — True

(4) All Heaps are FBTs — False

* Perfect Binary Tree (PBT): All internal nodes have two children & all leaf nodes are at the same level.



CBT ✓
FBT ✓
PBT ✓

* The no. of leaf nodes in PBT is equal to no. of internal nodes plus 1.
* Height of PBT - $(\log_2 n)$
* if height is h then how many nodes in tree? $2^{(h+1)} - 1$
* Balanced Binary Tree — A tree is balanced if height is $\log n$

Ex: ① AVL Tree

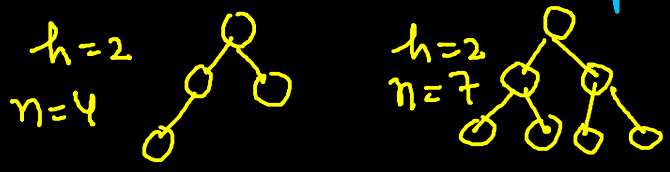
$$|\text{left height} - \text{Right height}| \quad 0,1$$

AVL ✓

(2) Red Black Trees are also balanced binary trees.

* BBTs are very efficient as they provide O(logn) time for search, insert & delete.

Qu 1. What is min & max no. of elements in a heap of height h?

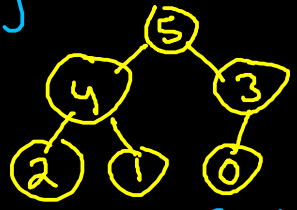


$min - 2^h$

$max - 2^{h+1} - 1$

Qu 2: Is an array that is reverse sorted a heap?

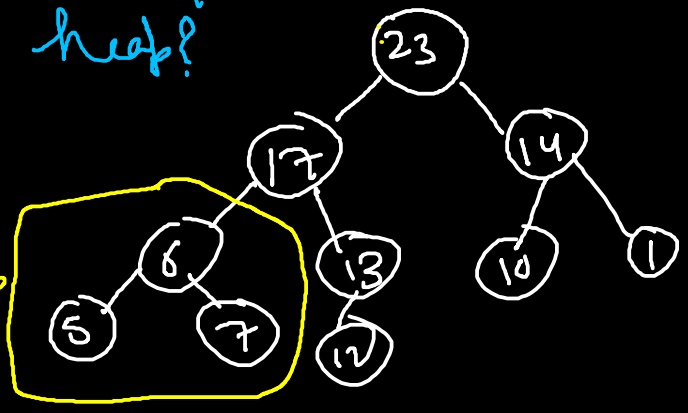
5, 4, 3, 2, 1, 0



Max Heap Yes

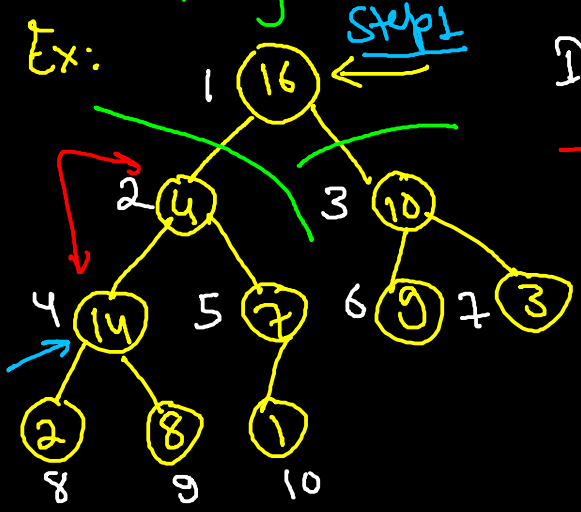
Qu-3 Is the sequence 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 a heap?

Violates the property of max heap



Heapify (Maintaining the heap property)

Ex:



Is it a heap?

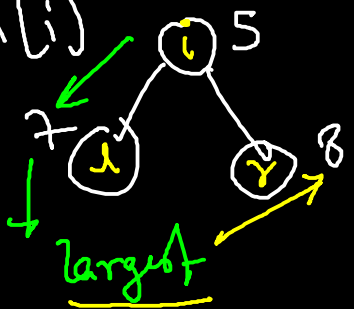
The problem is due to element 4 at index 2, So we can run heapify function at index 2.

→ The function of heapify is to float down the value at index 2 which is violating property of max heap.

```
Heapify (A, i)
    l = Left(i)  // 2i   (index of left child)
    r = Right(i) // 2i+1 (index of right child)
```

find largest element in a node of its 2 children

```
if l ≤ heap-size(A) and A[l] > A[i]
        largest = l
else    largest = i
if r ≤ heap-size(A) and A[r] > A[largest]
        largest = r
```



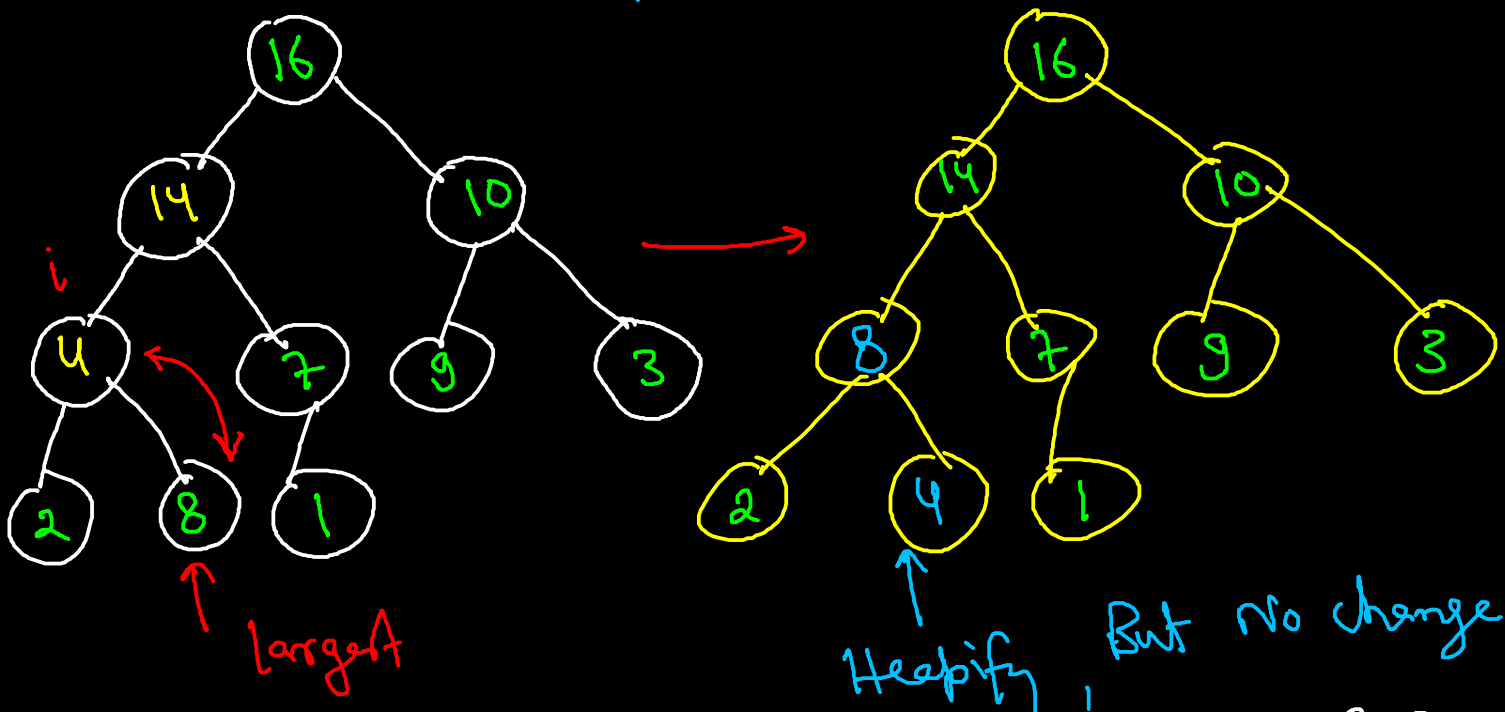

```
if largest ≠ i
    Swap(A[i], A[largest])
    Heapify(A, largest)
```

Time → $T(2n/3)$

RR

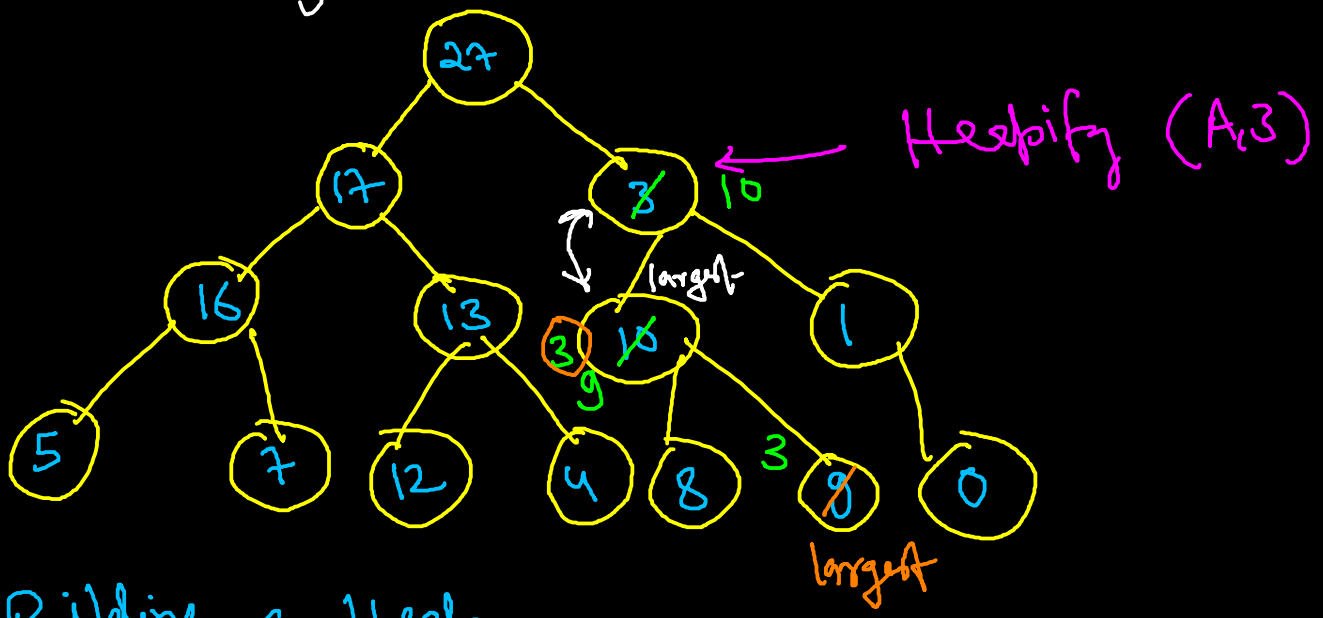
$$T(n) \leq T(2n/3) + O(1)$$

Step 2 after Heapify :-



Ex: A = 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0

Illustrate the operation of Heapify (A, 3) on array A where index starts with 1.



* Building a Heap

Build Heap (A)

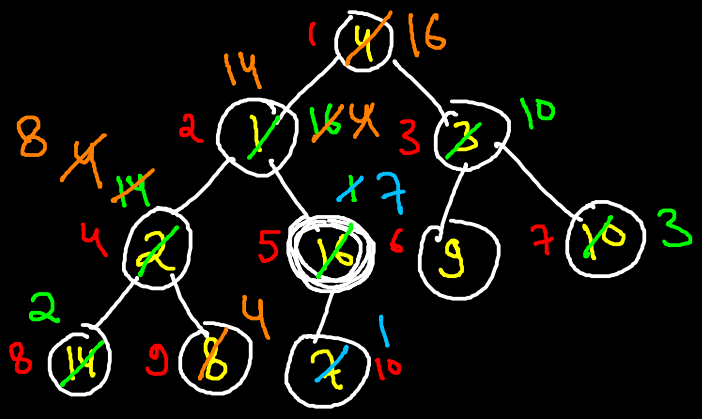
heap-size [A] = length (A) —— O(1)

For i = ⌊length(A)/2⌋ down to 1 (n/2)

Heapify (A, i) ⟵ (log n)

* Why Build_Heap starts / begins heapify from $\lfloor length(A)/2 \rfloor$

Ex: 4, 1, 3, 2, 16, 9, 10, 14, 8, 7



$n = \lfloor length[A)/2 \rfloor = 5$

so heapify will begin at index 5.

No change

① → Heapify (A, 4)

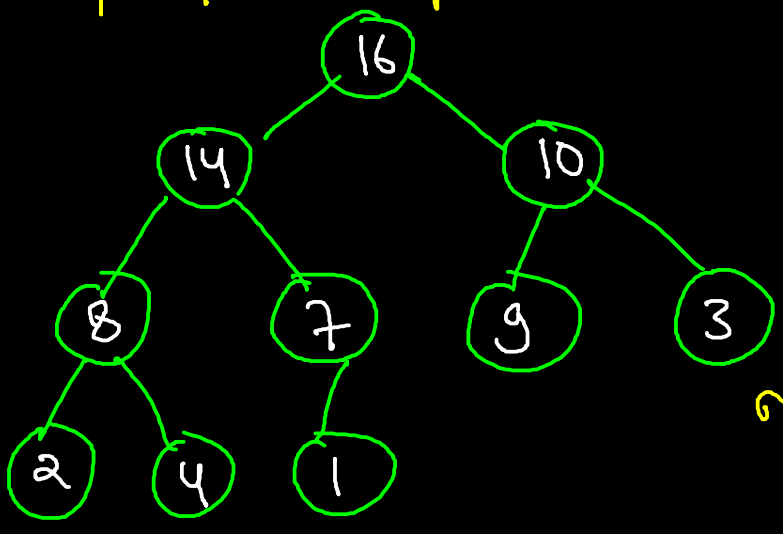
② → Heapify (A, 8) no change

③ Heapify (A, 3)
Heapify (A, 7) no change

④ Heapify (A, 2)
Heapify (A, 5)
Swap (A[5], A[10])
Heapify (A, 10)
No change

⑤ Heapify (A, 1)
Swap (A[1], A[2])
Heapify (A, 2)
Swap (A[2], A[4])
Heapify (A, 4)
Swap (A[4], A[8])
Heapify (A, 9) no change

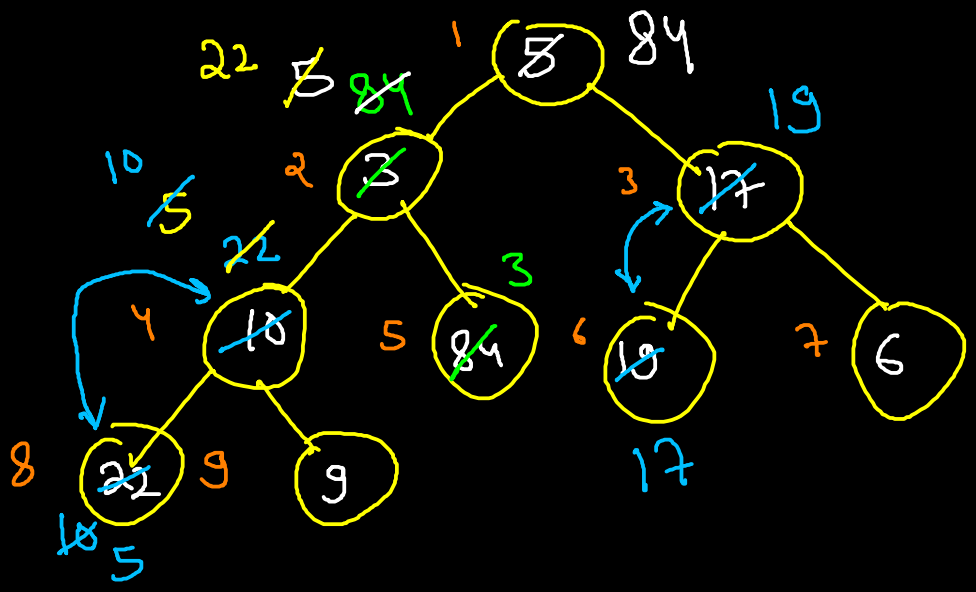
final Heap



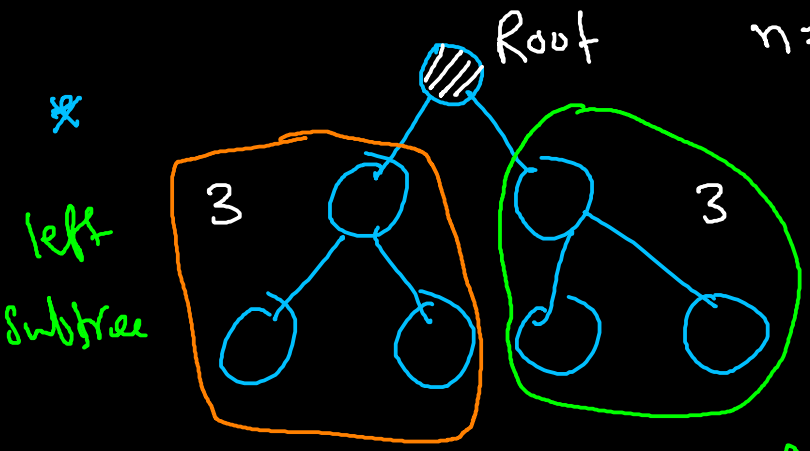
now its a max Heap

Que: Apply Build Heap on

A = 5, 3, 17, 10, 84, 19, 6, 22, 9



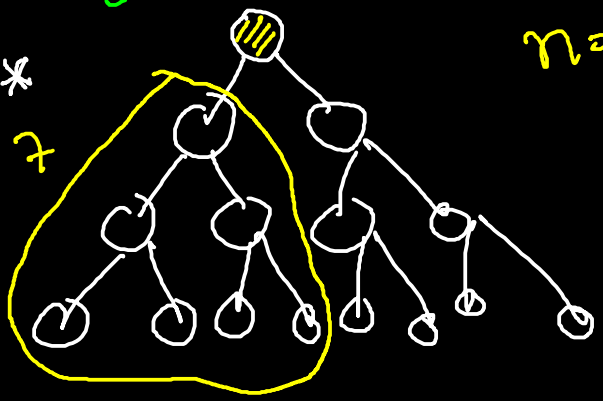
Build Heap will begin Heapify at index = 4



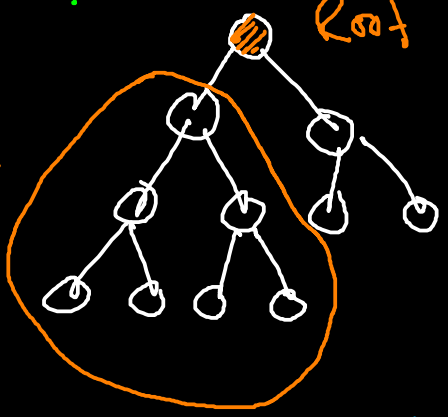
* left subtree

Right Subtree

What will be max no. of nodes in left & right subtree of a node?



* 3,4,5,6,7



* The subtrees at a node have at most $2n/3$ nodes. The worst case occurs when the last row of the tree is exactly half full.

Solve: $T(n) = T(2n/3) + O(1)$ (Heapify)

$a = 1 \quad b = 3/2 \quad f(n) = 1$

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0$

Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ for $k=0$

$1 = \Theta(n^0 \log^0 n)$

Time Complexity of Heapify: $T(n) = \Theta(\log n)$

* Time of Build Heap:- $O(n \log n)$

* This is upper bound & is not asymptotically tight.

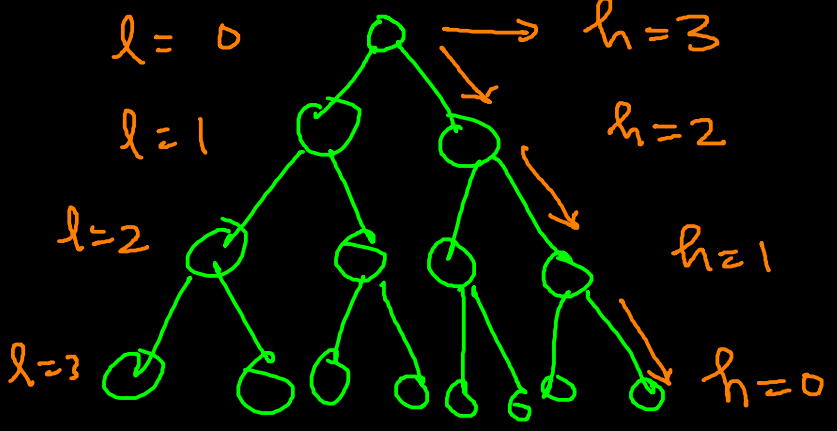
* Time for heapify varies with the height of the node in the tree, and the heights of the most nodes are small.

Property:- In an n-element heap there are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of height h.

— Time for Build Heap will come out

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

* Hence, we can build a heap from an unordered array in linear time.



$$\left\lceil \frac{n}{2^{h+1}} \right\rceil = \left\lceil \frac{15}{2} \right\rceil$$

(8)

nodes & height 1 = $\left\lceil \frac{15}{2^2} \right\rceil = \left\lceil \frac{15}{4} \right\rceil = 4$

Complexity of heapify – $O(h)$

$$T(n) = \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil * O(h)$$

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

$$T(n) = O\left(n * \sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$

formulae (1) $\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$ — By Differentiating & multiplying by x on both sides

(2) $\sum_{n=0}^{\infty} n x^n = \frac{x}{(1-x)^2}$

$n = h, \; x = \left(\frac{1}{2}\right)$

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

$$= O\left(n * \frac{\frac{1}{2}}{\frac{1}{4}}\right) = O(2n)$$

$T(n) = O(n)$ Time Complexity of Build heap is $O(n)$

HeapSort (A) → $O(n)$ Time - $O(n \lg n)$

BuildHeap (A)

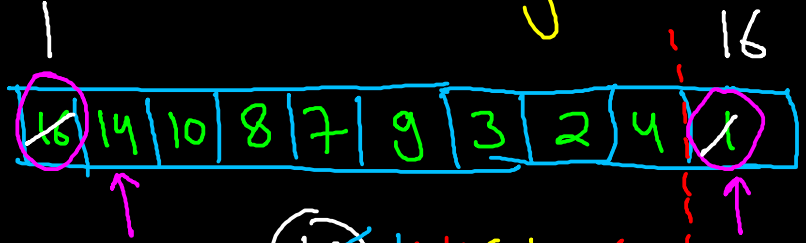
for i = length(A) down to 2 → 10 in this Example

swap (A[1], A[i])

heapsize [A] = heapsize[A] - 1

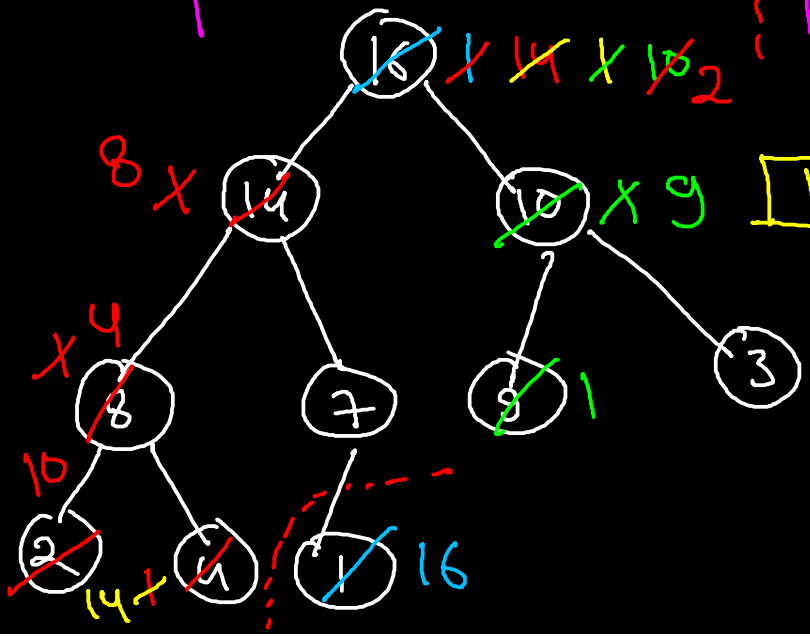
Heapify (A, 1)

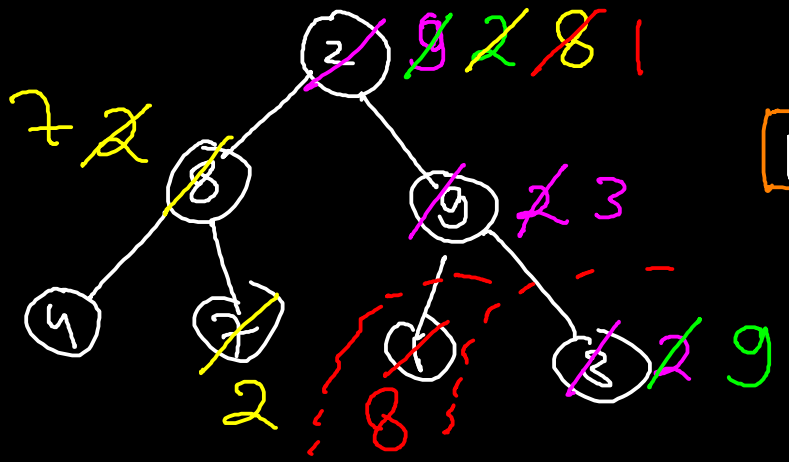
} n lgn

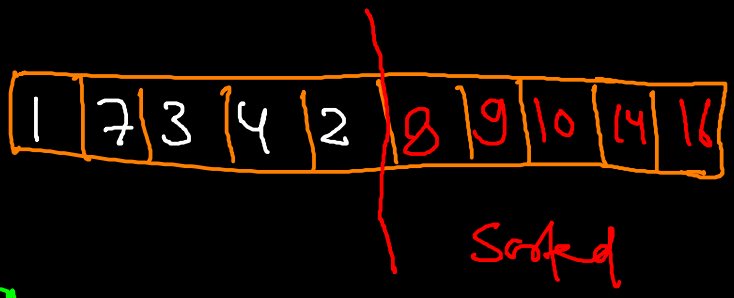


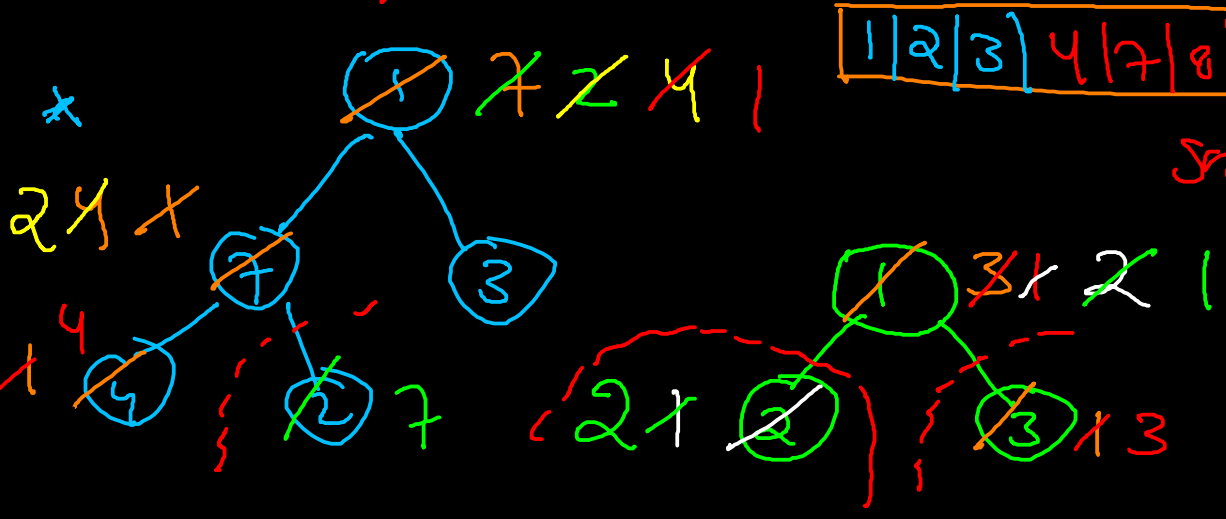
Is it max Heap?

Yes









Sorted array is

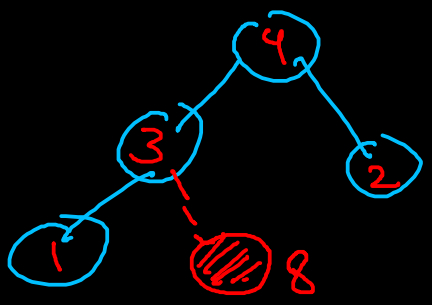
1, 2, 3, 4, 7, 8, 9, 10, 14, 16

Heap can be used as priority Queue.

Priority Queue:- PQ is a data structure for maintaining a set S of elements each with an associated value called a key.

Operations on PQ:

1. Insert (S, x) — insert key — O(log n)
2. Maximum (S) — return max — O(1)
3. Extract-max (S) — Remove & return max — O(log n)



Applications: —

1. Priority scheduling algo in OS
2. Graph. Algos. Dijkstra & Prim

Graphs:

Graphs consists of following components -

1. finite set of vertices (nodes)
2. finite set of ordered pair of the form (u,v) called as edge. $(u,v) \neq (v,u)$

* **Applications:** ① Social network - facebook, linkedin, instagram, twitter etc.

Ex:

*

② Electric Circuit

③ Telephone n/w

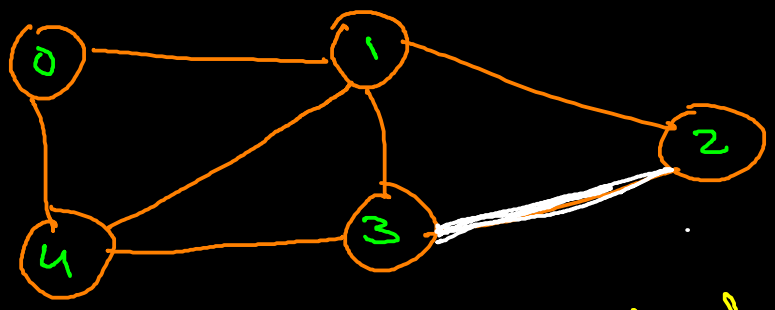
④ Paths in a city

Representation of Graphs

1. Adj. matrix
2. Adj list

① Adj matrix:- 2D array of size $V \times V$ where V — no. of vertices in graph G.

Ex:



adj[2][3] = 1
adj[3][2] = 1

* adj[i][j] = 1 (indicates an edge)

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

* adj matrix for an undirected graph is always symmetric.

* for weighted graph. adj[i][j] = w

Pros: ① Easy implementation
② Remove edge — O(1)
③ Find is an edge? O(1)

Cons: ① O(V^2) space

Sparse / Dense in both cases

Sparse → mostly 0's

Dense → mostly 1's in adj matrix

Ex: Students in a class - 60

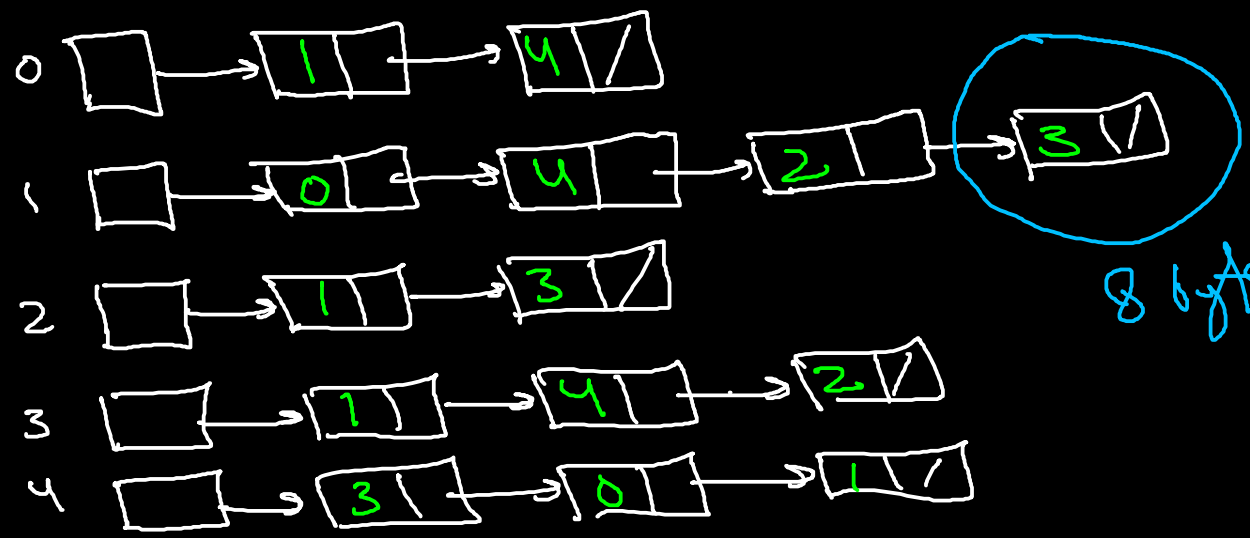
matrix - 60 x 60 = 3600 elements

1 entry takes 4 bytes - 14400 bytes

② Adding vertex - $O(V^2)$

② Adj list: An array of list is used.

* size of array = no. of nodes



* adj list is better for sparse graphs.

* adj matrix is good for dense graphs.

Pros: ① Space $O(V+E)$

② Adding a vertex is easier

Cons:- ① Whether there is an edge from vertex u to v - $O(V)$

* `vector<int> adj[V];`

`addEdge(u,v)`

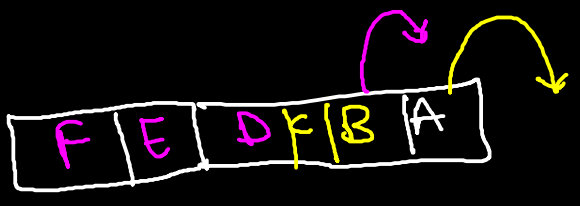

```
adj[u].push_back(v)
adj[v].push_back(u)
```

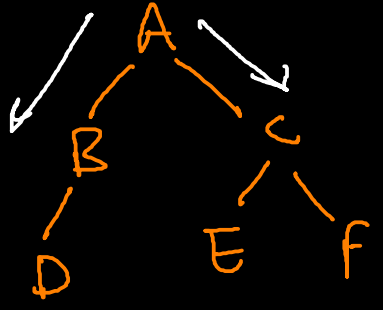
* In real life most graphs are sparse.
  - Therefore adj list is used in most cases.

BFS & DFS

BFS: - vertex based technique for finding the shortest path in the graph.

- Queue





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

DFS: - Edge based technique
- Stack

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

BFS

1. Queue
2. Suitable for finding vertex closer to source
3. adj list. $O(V+E)$
   adj matrix $O(V^2)$
4. Requires more space
5. Siblings are visited before children
6. Find single source shortest path, because in BFS we reach a vertex with min. no. of edges from a source vertex.
7. Applications:- Bipartite graph, shortest path

DFS

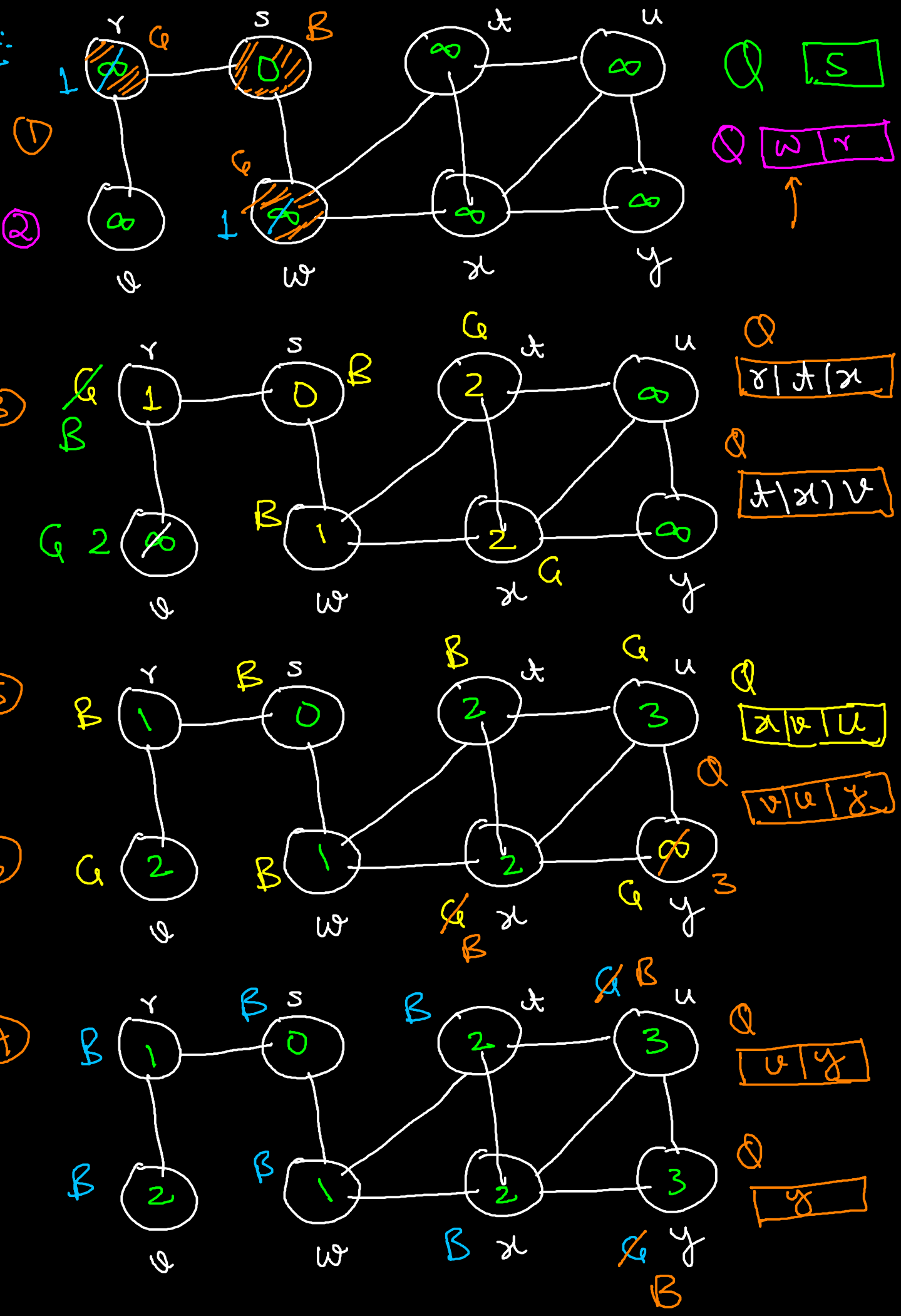
1. Stack
2. Suitable when there are nodes away from source
3. ✓ ✓
4. Less space
5. Children are visited before the siblings
6. We might traverse through more edges to reach a destination vertex from a source
7. Topological order, Acyclic graphs

Pseudocode :

```
BFS(G,S)
    For each vertex u ∈ G.V - {S}          } O(V)
        u.color = WHITE
        u.d = ∞
        u.π = NIL
    S.color = GRAY
    S.d = 0
    S.π = NIL
    Q = ϕ
    ENQUEUE(Q,S)
    while Q ≠ ϕ
        u = DEQUEUE(Q)
        for each v ∈ G.adj[u]              } O(E)
            if v.color == WHITE
                v.color = GRAY
                v.d = u.d + 1
                v.π = u
                ENQUEUE(Q,v)
        u.color = Black
```

Time - $O(V+E)$

Ex:



DFS: Colors:

① White :- Vertices are white before discovery

② Gray :- vertices are gray when they are discovered but their outgoing edges are still in the process of being explored.

③ Black :- Vertices are black when the entire subtree starting at the vertex has been explored.

Events: (d/f)

① disc(u) :- The time at which we discover vertex u ie. the time at which it becomes gray.

② finish (u) :- The time at which we finish exploring all edges out of vertex u ie. the time at which it becomes black.

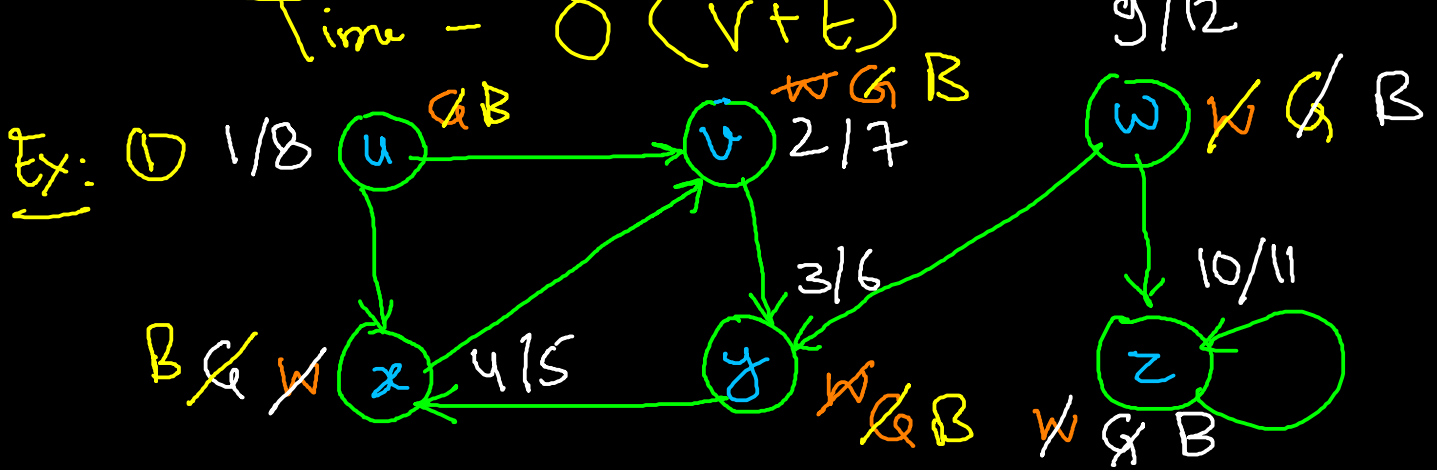
```
DFS(G)
    ∀u ∈ V:   color(u) = white
    time = 0
    for all u ∈ V
        if color(u) == white
            DFS-visit(u)
```

O(V)

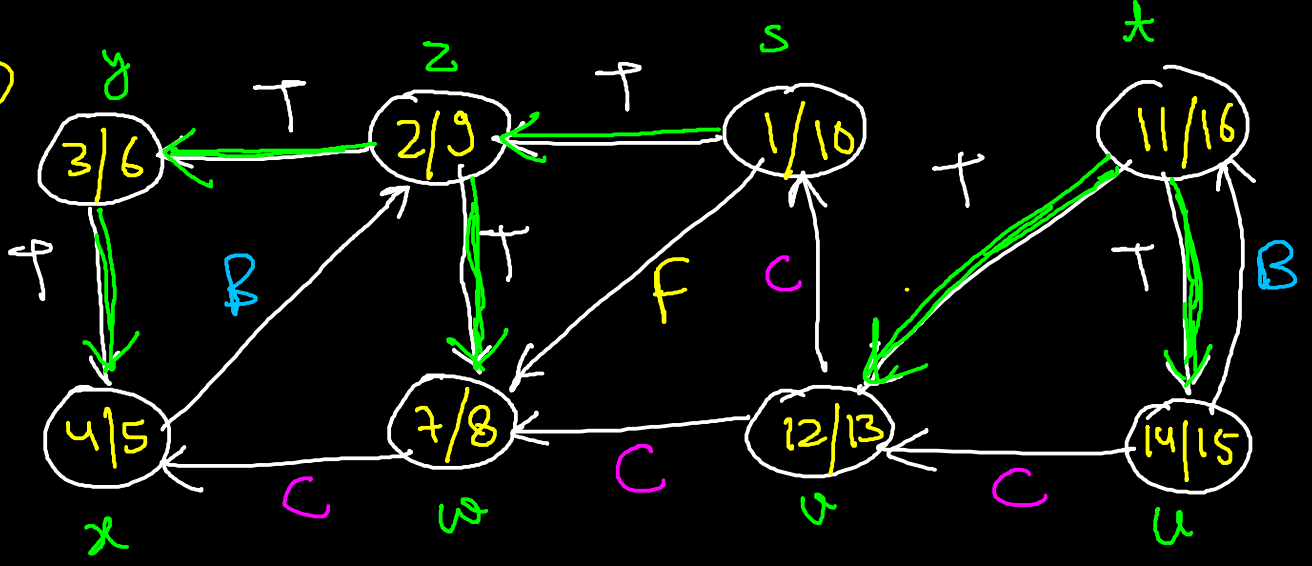
```
DFS-Visit(u)
    color(u) = gray
    time = time + 1
    disc(u) = time
    for all v ∈ adj[u]
        if color(v) == white
            DFS-visit(v)
    color(u) = Black
    time = time + 1
    finish(u) = time
```

O(E)

Time = O(V+E)



Ex: ②



classification of edges:-

① Tree Edges:- Edge (u,v) is TE if v was discovered first by exploring edge (u,v).

② Back Edges:- Those edges (u,v) connecting a vertex u to an ancestor in a depth first tree.

③ forward Edges:- Those non-tree edges (u,v) connecting a vertex u to a descendent v in a depth first tree.

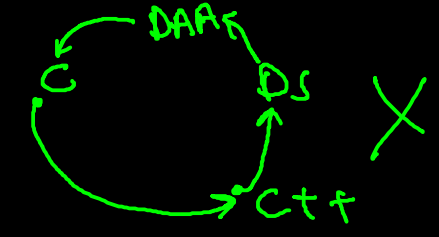
④ Cross Edges:- All other edges

they can go between vertices in the same depth first trees, as long as one vertex is not an ancestor of the other or they can go between vertices in different depth first trees.

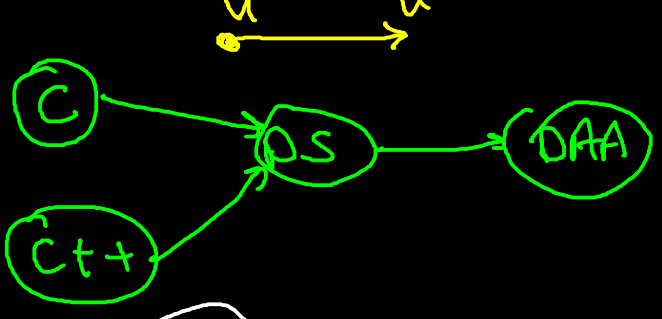
Topological Sort:- A topological sort of a DAG (Directed acyclic graph) $G=(V,E)$ is a linear ordering of all its vertices such that if G contains an edge (u,v) then u appears before v in the ordering.

* If G contains a cycle, then no linear ordering is possible.

Ex: ①



X



② Instruction Scheduling

$C = a+b$

$d = b+c$

C, C++, DS, DAA
C++, C, DS, DAA

③ Resolving symbol dependencies in linkers

④ Software updates

⑤ Project management (Indicate precedence among various events)

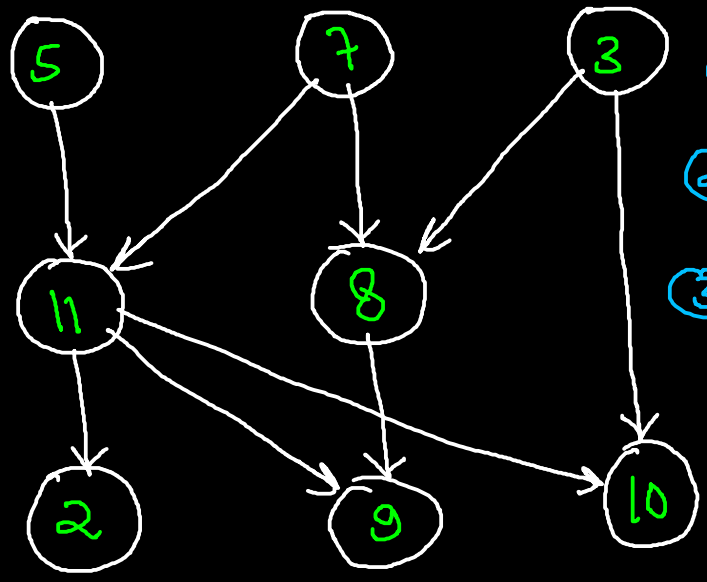
Algo: ① Call DFS(G) to compute finishing time v.f for each vertex v — $O(V+E)$

② As each vertex is finished, insert it onto the front of a linked list — $O(V)$

③ return the linked list of the vertices

Time – $O(V+E)$

Ex: ①



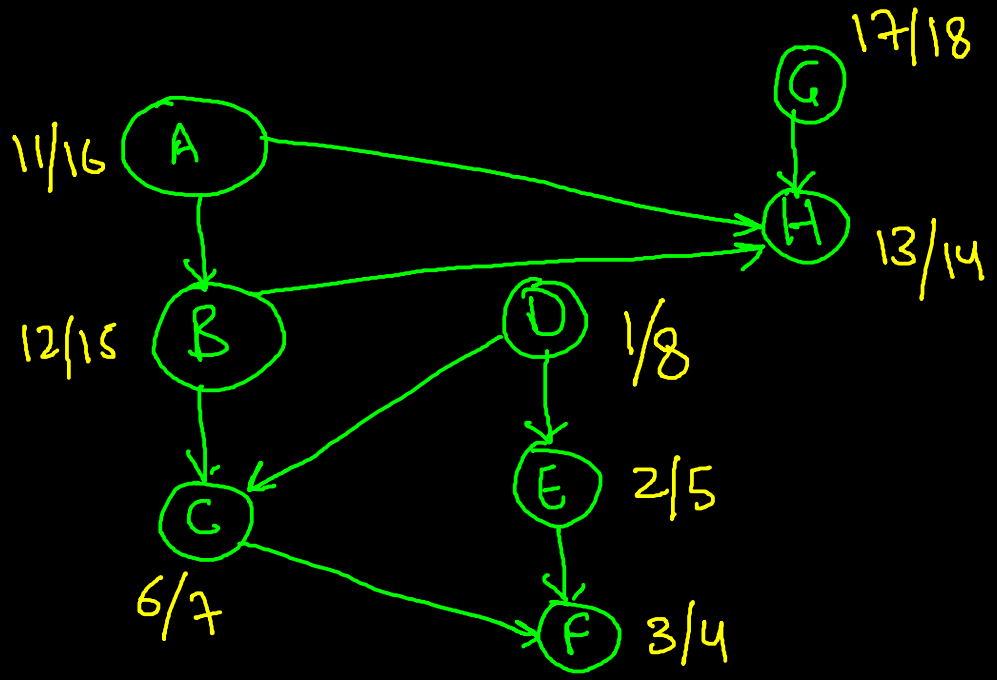
① 5, 7, 3, 11, 8, 2, 9, 10

② 3, 5, 7, 8, 11, 2, 9, 10

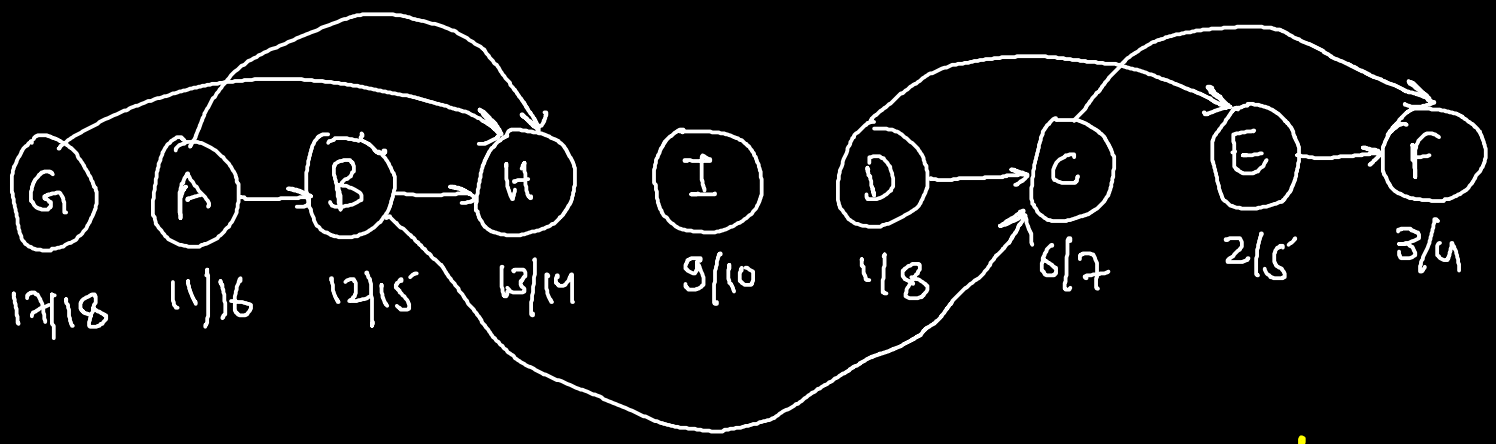
③ 5, 7, 3, 8, 11, 10, 9, 2

④ 3, 7, 8, 5, 11, 10, 9, 2

Ex ②



I 9/10



Topologically sorted Graph, with its vertices arranged from left to right in order of decreasing finish time.

Disjoint set DS:- A disjoint set DS maintains a collection $S = \{S_1, S_2, \ldots S_k\}$ of disjoint dynamic sets.

$S_1 = \{1, 2\}$ $S_2 = \{3, 4\}$

- Identify each set by a representative which is some member of the set.
- Ask for representative of a dynamic set twice without modifying the set — you will get same answer.

Operations

1. **Make-set (x):-** Creates a new set whose only member & thus representative is x.

S_1 = make-set (1) = {1}, 1

S_2 = make-set (2) = {2}, 2

② Union (x, y): Unites the dynamic sets that contain x & y say S_x & S_y into a new set that is union of two sets.

– Choose a new representative

S = Union (1, 2) = {1, 2}

new representative – 1

③ find-set (x):- Returns a pointer to the representative of the set containing x.

find-set (2) = 1

Ex: S_1 = {1, 2, 3} → 1

S_2 = {4, 5, 6} → 4

find-set (3) = 1

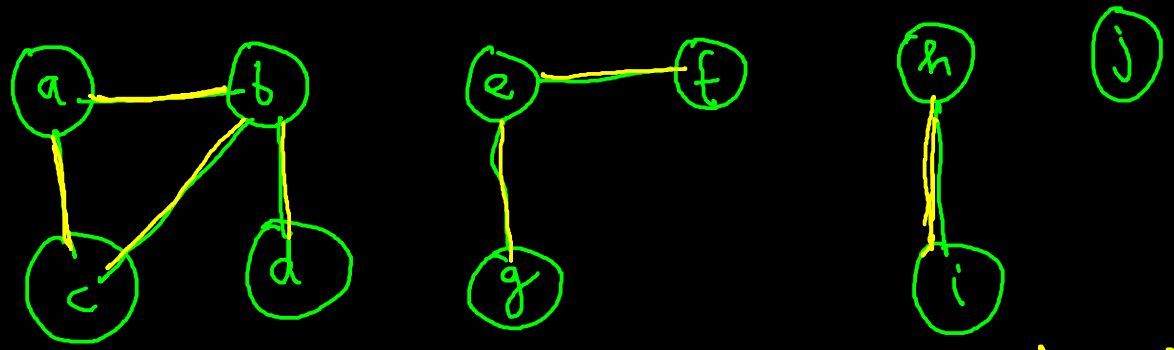
Union (3, 6) = {1, 2, 3, 4, 5, 6} → 1

Application: ① Connected Components

Connected Components (G)

```
for each vertex v ∈ G.V
    make-set (v)
for each edge (u,v) ∈ G.E
    if find-set (u) ≠ find-set (v)
        Union (u,v)
```

Ex:



Edge Processed	Collection of disjoint sets
Initial sets	{a} {b} {c} {d} {e} {f} {g} {h} {i} {j}
bd	{a} {b,d} {c} {e} {f} {g} {h} {i} {j}
eg	{a} {b,d} {c} {e,g} {f} {h} {i} {j}
ac	{a,c} {b,d} {e,g} {f} {h} {i} {j}
hi	{a,c} {b,d} {e,g} {f} {h,i} {j}
ab	{a,b,c,d} {e,g} {f} {h,i} {j}
ef	{a,b,c,d} {e,f,g} {h,i} {j}
fc	

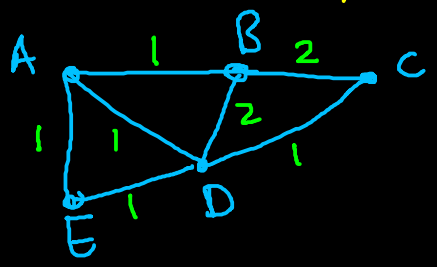
There are 4 Components

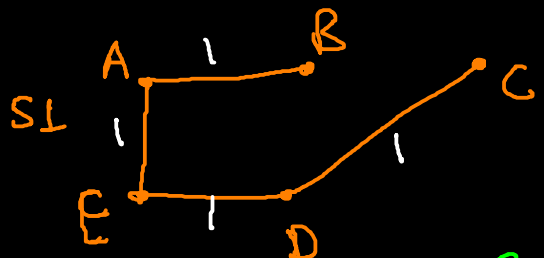
* **Same-Component (u,v)**

```
if find-set (u) == find-set (v)
    return True
else return false
```

Minimum Spanning Tree (MST): Subgraph that is tree & connects all vertices

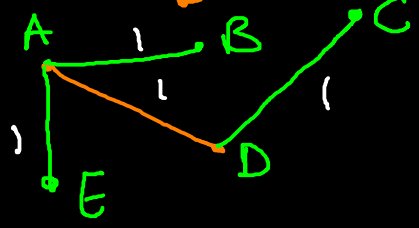
→ Sum of weights of all edges should be minimum among all spanning trees of the graph.





MST weight 4

AB, AE, AD, ED, DC, BC, BD



cycle formation

(DFS, disjoint set DS)

Kruskal → $O(E \lg V)$ using ordinary Binary Heap

Prim's → Fibonacci heap — $O(E + V \lg V)$

↓

Better if $|V|$ is much smaller than $|E|$

Loop Invariant :- Prior to each iteration A is subset of some MST.

At each step, we determine an edge (u,v) that we can add to A without violating this invariant, in the sense that $A \cup \{(u,v)\}$ is also a subset of MST.

Such an edge is a safe Edge for A, since we can add it safely to A while maintaining the invariant.

$A \rightarrow$ set of Edges

```
Generic-MST (G, w)
    A = ∅
    while A doesn't form a spanning tree
        find an edge (u,v) that is safe for A
        A = A ∪ {(u,v)}
    return A
```

```
MST-Kruskal (G,w)
    A = ∅
    for each vertex v ∈ G.V }                O(V)
        make-set (v)
(E lg E) sort the edges of G.E into non-decreasing order by w
    for each edge (u,v) ∈ G.E, taken in  "  "  "  "
O(E)        if find-set (u) ≠ find-set (v)
                A = A ∪ {(u,v)}
```

Union(u,v)

return A

Time - $O(E \lg E)$
- $O(E \lg V)$

Prim's Algo:

MST-Prim(G, w, r)

for each $u \in G.V$ } $O(V)$

$u.key = \infty$

$u.\pi = NIL$

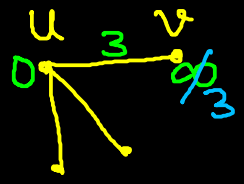
$r.key = 0$

$Q = G.V$

while $Q \neq \phi$

$u = Extract\text{-}min(Q)$ → $O(V \lg V)$

for each $v \in adj[u]$



$O(E)$

if $v \in Q$ and $w(u,v) < v.key$

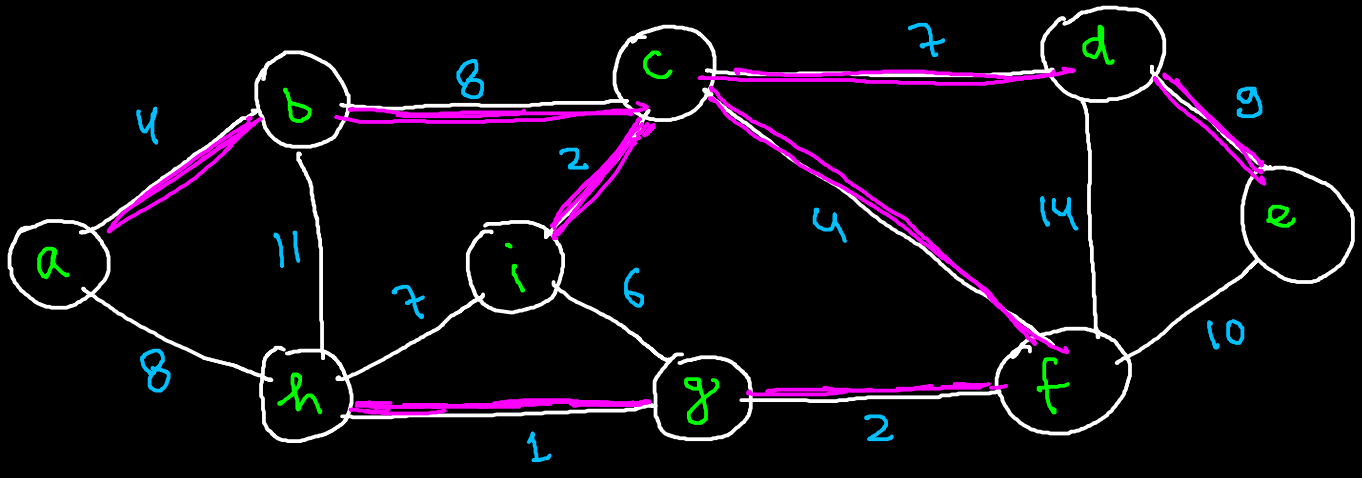
$v.\pi = u$

$v.key = w(u,v)$ $O(\lg V)$

Time - $O(V \lg V + E \lg V) = O(E \lg V)$

Ex:1

Kruskal

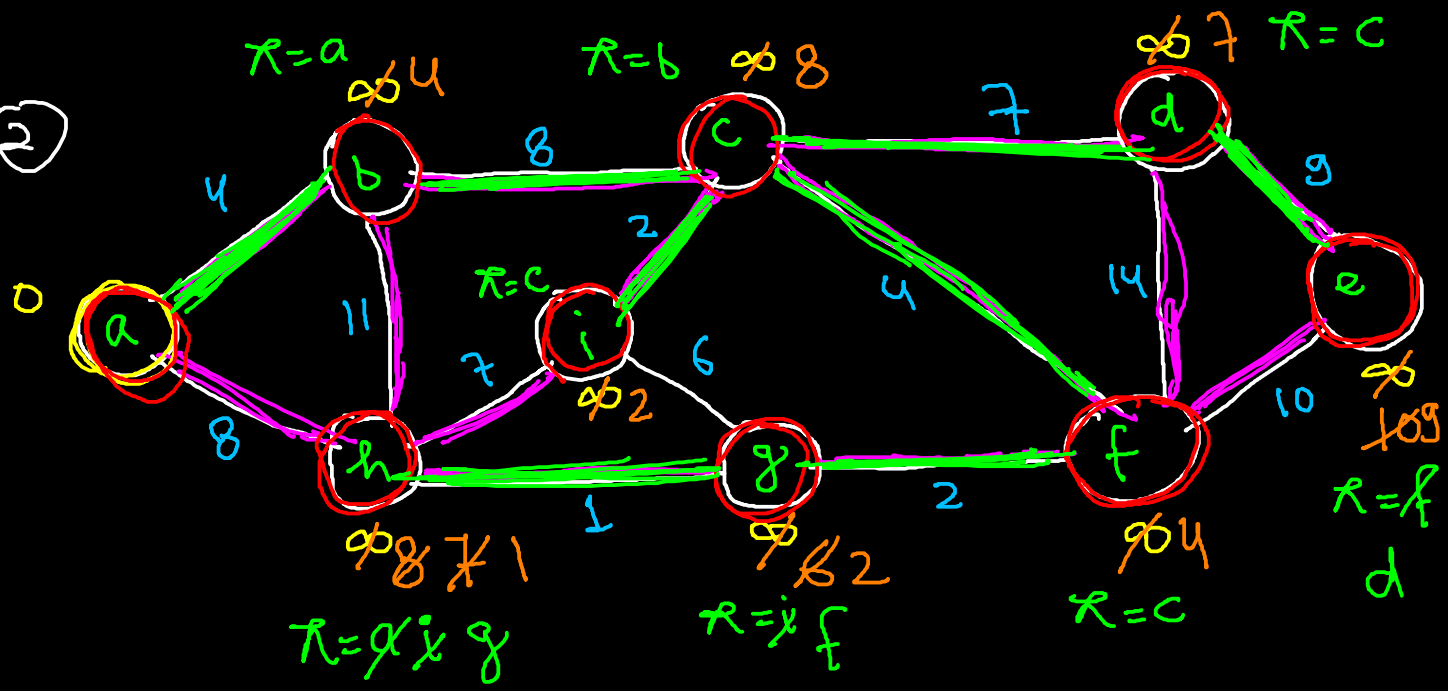


hg, gf, ci, ab, cf, ig, hi, cd, bc, ah, de, ef
1 2 2 4 4 6 7 7 8 8 9 10

bh, 11 df, 14

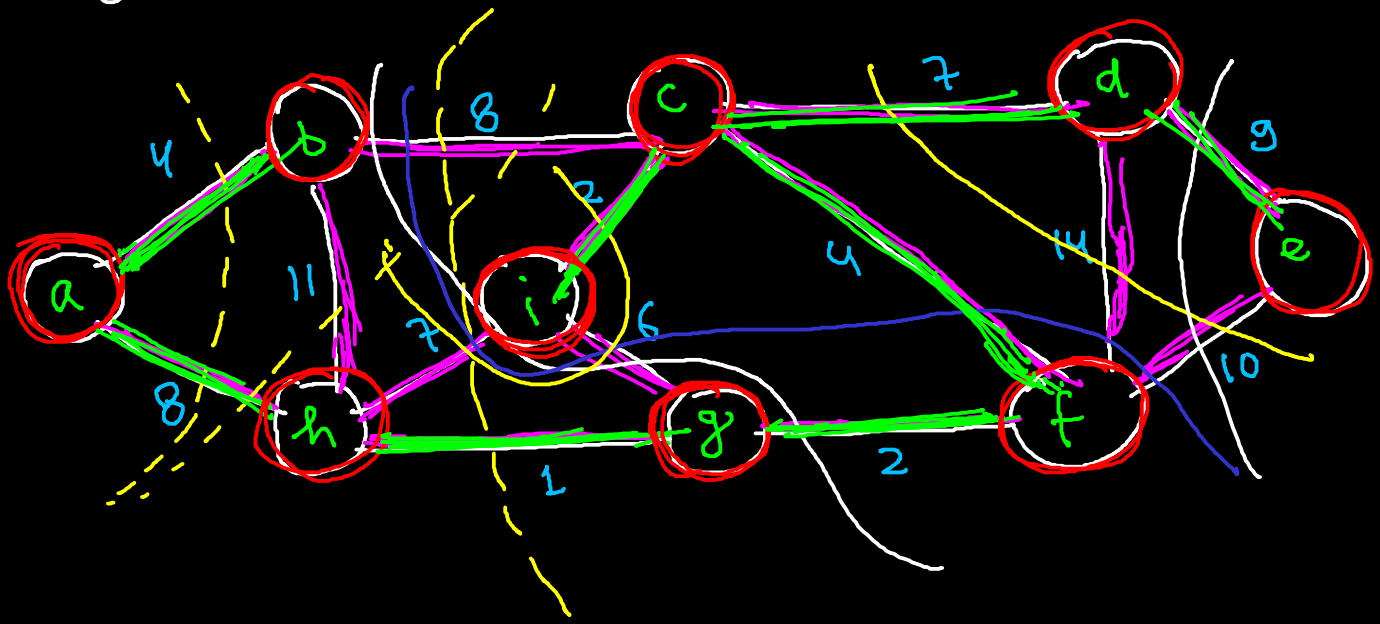
MST weight = 1+2+2+4+4+7+8+9 = 37

Ex: 2



MST weight = 4+8+2+1+7+4+2+9 = 37

Ex: Using cutsets

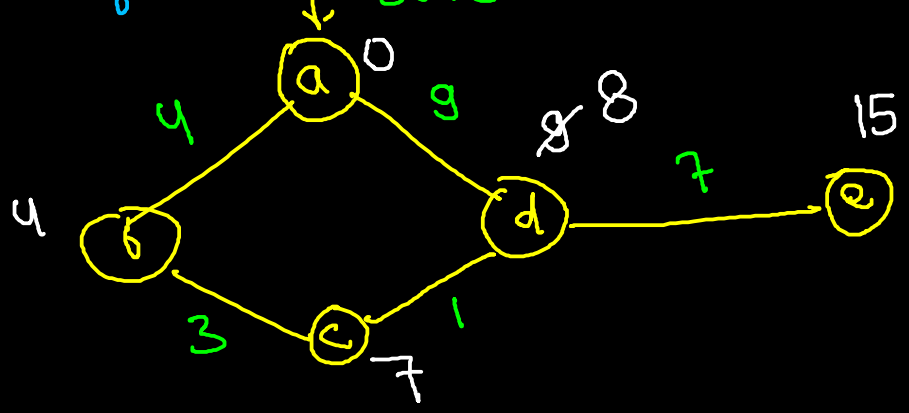


Single Source shortest paths Algorithms:-

① **Dijkstra's Algo:-** It solves the single source shortest paths problems on a weighted directed graph.

- It maintains a set S of vertices whose final shortest paths weights from the source s have already been determined.
- The algo repeatedly selects vertex $u \in V-S$ with min shortest path distance estimate, adds u to S & relaxes all edges leaving u.

Ex: ①



Dijkstra (G, w, s)

Initialize-single Source (G, s) → for each vertex $v \in G.V$: $v.d = \infty$, $v.\pi = NIL$; $s.d = 0$

$S = \phi$

$Q = G.V$ → $O(V)$

while $Q \neq \phi$

$u =$ Extract-min (Q) → $O(\lg V)$

$S = S \cup \{u\}$

for each vertex $v \in G.adj[u]$ → $O(E)$

RELAX (u, v, w)

Test whether we can improve shortest path to v found so far by going through u & if so update v.d & v.π.

```
if v.d > u.d + w(u,v)
   v.d = u.d + w(u,v)
   v.π = u
```

Assumption:- Non-negative edges $w(u,v) \geq 0$

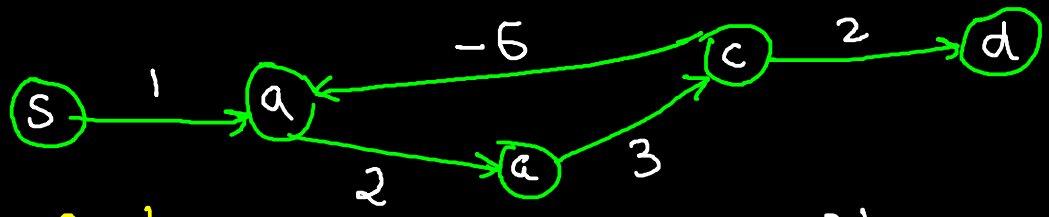
Question: Can Dijkstra's algo give correct answer if there is a -ve weight edge in the graph?

- Yes, it can, but it doesn't guarantee that answer will always be correct in case of -ve edges.

But it will never work in case of -ve edge cycle.

Another algo, Bellman ford works in case of -ve weight edges but it also fails in case of -ve edge cycle.

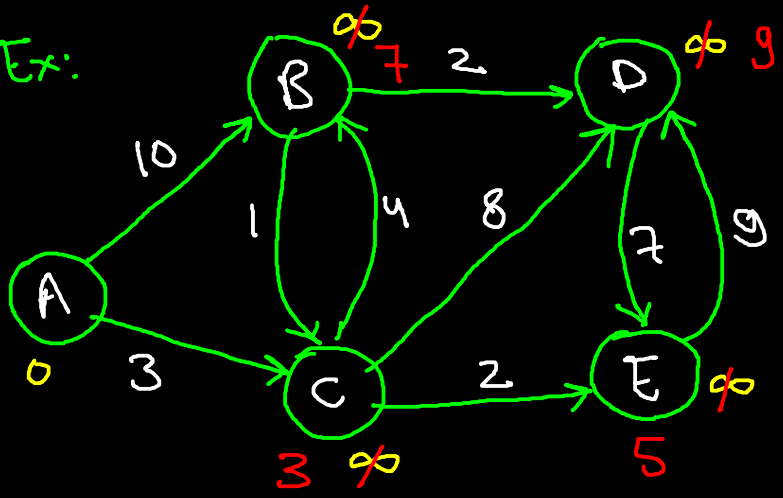
Ex:



Time Complexity: Binary Heap - $(E+V)\log V \rightarrow E\log V$

Array - $O(V^2)$ Fibo Heap - $E + V\log V$

Ex:

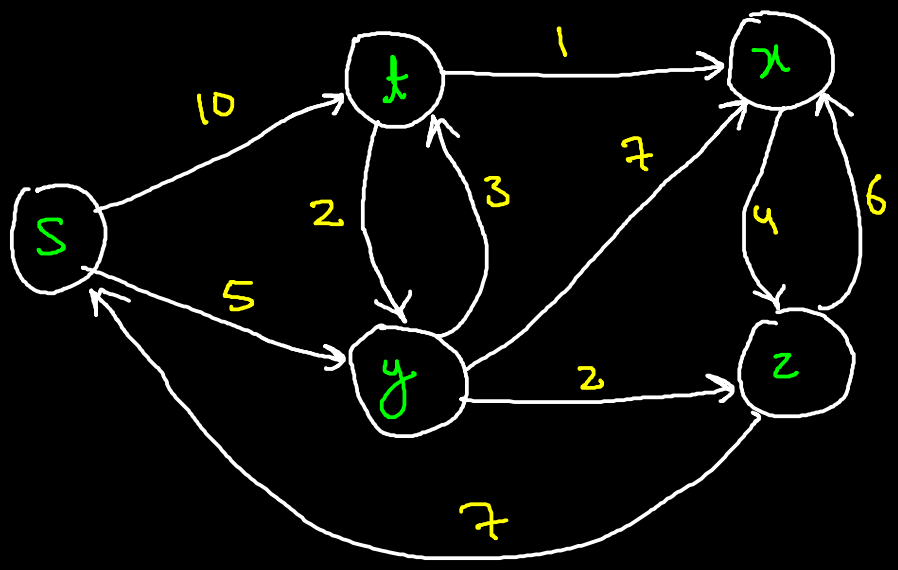


$S = \{\}$

$Q = \{A, B, C, D, E\}$

S	A	B	C	D	E
	0	∞	∞	∞	∞
{A}	0	10	3	∞	∞
{A,C}	0	7	3	11	5
{A,C,E}	0	7	3	11	5
{A,B,C,E}	0	7	3	9	5

Ex:



Dijkstra
↓
Greedy Algo

Bellman ford
↓
Dynamic Programming

Bellman ford: Single source shortest path algo.

- Algo returns a boolean value indicating whether or not there is a -ve weight cycle that is reachable from the source.

If cycle → no solution exists

If no cycle → Returns shortest path

```
BellmanFord (G, W, S)
    Initialize-single-source (G,S)  ——— O(V)
    for i=1 to |G.V| - 1
        for each edge (u,v) ∈ G.E      } O(VE)
            RELAX (u,v,w)

    for each edge (u,v) ∈ G.E
        if v.d > u.d + w(u,v)          } O(E)
            return false
    return True
```

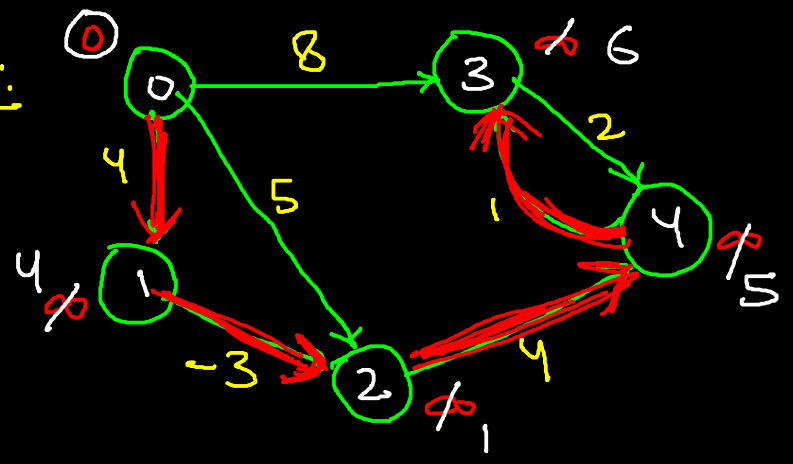
Check for a -ve weight cycle & return the appropriate boolean value.

Time — O(VE)

* Algo calculates the shortest distances which have atmost one edge in the path. Then it calculates shortest path with atmost i edges after ith iteration.

— There can be max |V|-1 edges in any simple path that's why the outer loop runs |V|-1 times.

Ex:



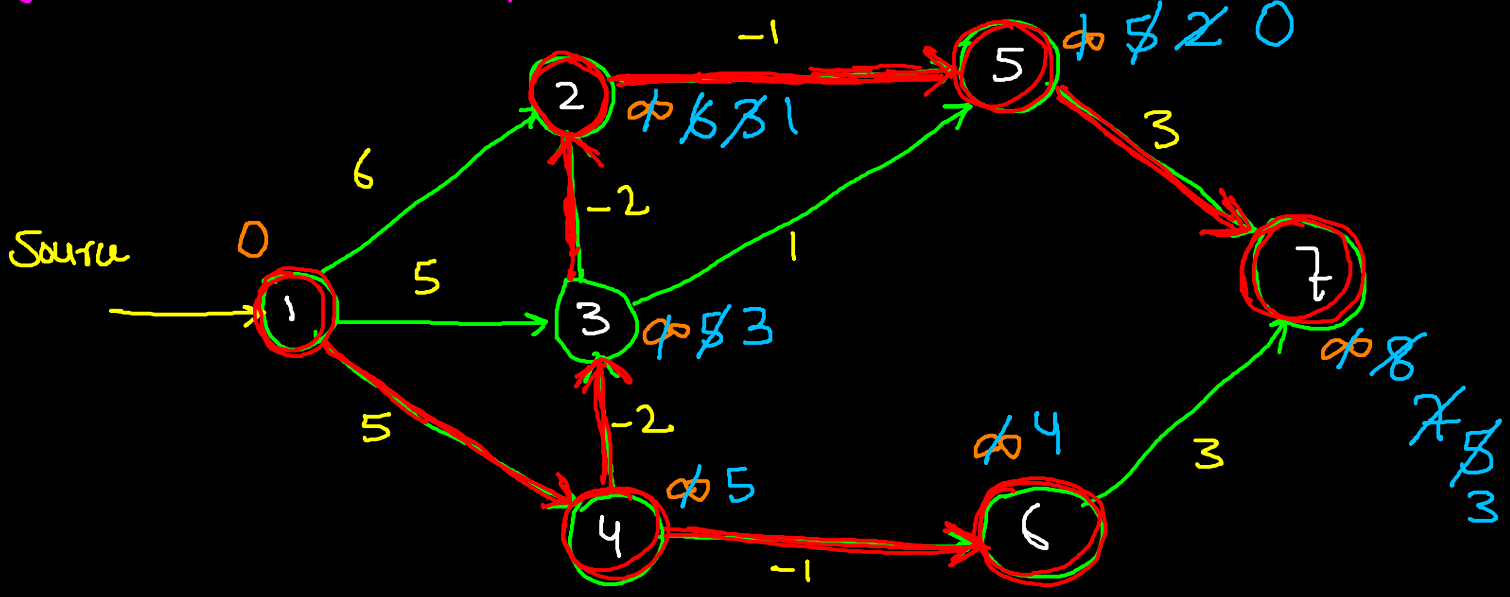
vertex	dis	parent
0	0	N
1	∞ 4	N 0
2	∞ 5 1	N 0 1
3	∞ 8 6	N 0 4
4	∞ 10 9 5	N 3 2 2

Edge to be processed

1st 2nd 3rd 4th

- 3 → 4 = 2
- 4 → 3 = 1
- 2 → 4 = 4
- 0 → 2 = 5 ✓
- 1 → 2 = 3
- 0 → 3 = 8 ✓
- 0 → 1 = 4 ✓

Ex: Bellman ford



Edge to be processed

(1,2), (1,3), (1,4), (2,5), (3,2), (3,5), (4,3), (4,6), (5,7), (6,7)

Relax:

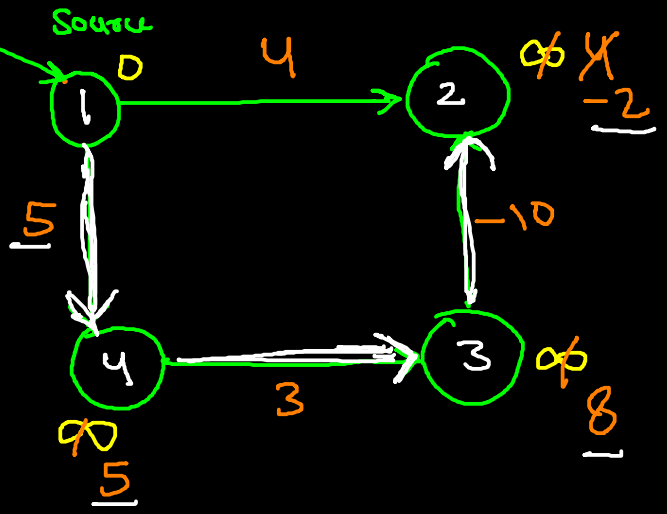
if $v.d > u.d + w(u,v)$

$v.d = u.d + w(u,v)$

$v.\pi = u$

$d(2) > d(1) + d(1,2)$

Ex:



Edges

(1,2), (1,4), (4,3), (3,2)

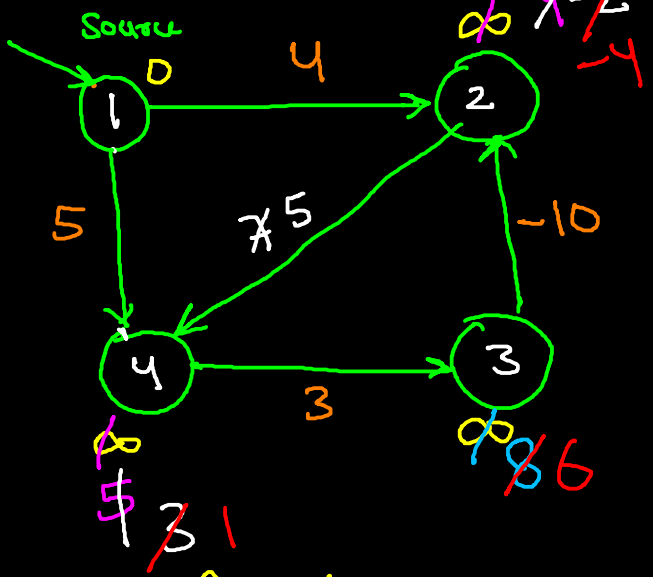
Edge to be processed

(3,2), (4,3), (1,4), (1,2), (2,4)

cycle length = d(4,3) + d(3,2) + d(2,4)

= 3 + (-10) + 7 = 0

Ex:



now change d(2,4) = 5

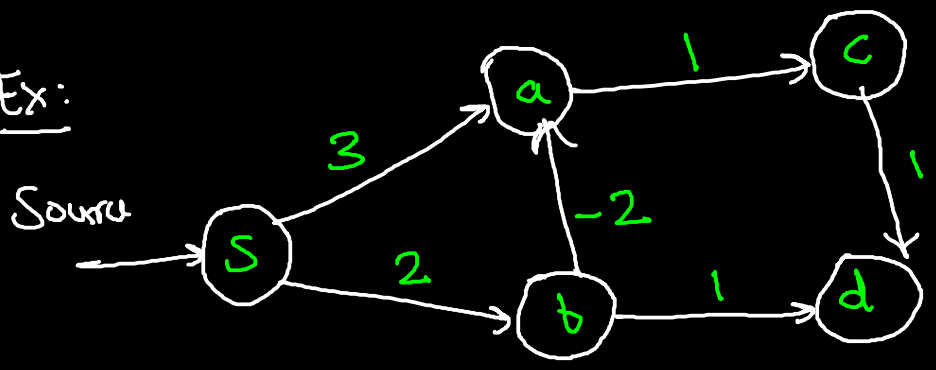
cycle length = 3 + (-10) + 5

= -2

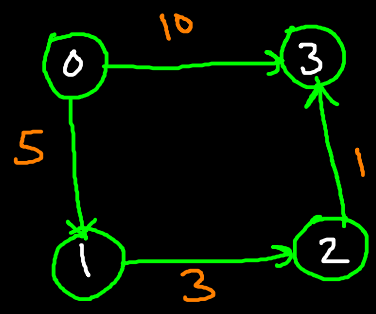
* Neither Dijkstra nor Bellman ford will work in case of negative weight cycle.

This is called negative weight cycle.

Ex:



Floyd Warshall Algo — All pair shortest path



graph[i][j] = 0 if i == j

& graph[i][j] = ∞ if there is no edge from i to j

	0	1	2	3
0	0	5	∞	10
1	∞	0	3	∞
2	∞	∞	0	1
3	∞	∞	∞	0

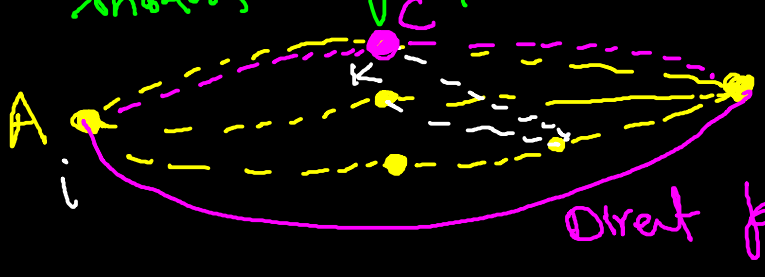
floyd warshall →

	0	1	2	3
0	0	5	8	9
1	∞	0	3	4
2	∞	∞	0	1
3	∞	∞	∞	0

* This algo is to find shortest paths in a weighted graph with +ve or -ve edge weights (but no negative cycles)

* This can also be used to find transitive closure of a relation R or widest paths between all pairs of vertices in a weighted graph.

* The idea is to pick all the vertices one by one & update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.



d(AC) + d(CB) < d(AB)

floydWarshall (int graph[V][V])

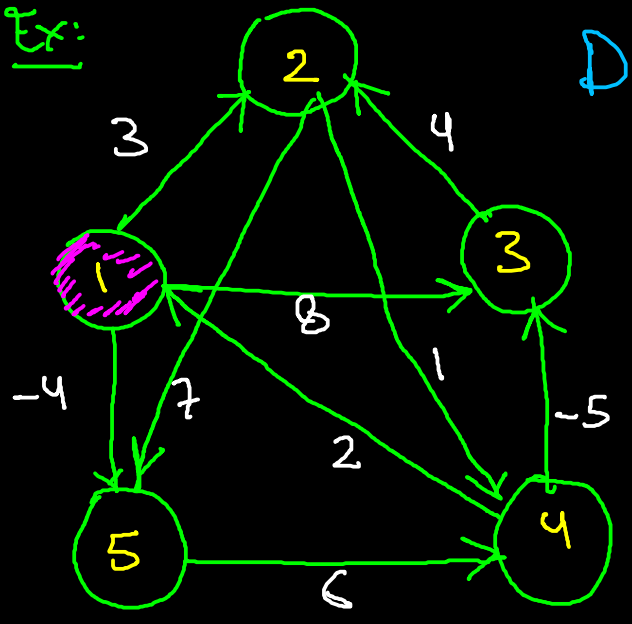
```
int dist[V][V], i, j, k;
for (i=0 to V)
   for (j=0 to V)
      dist[i][j] = graph[i][j];
```

// Initialize the dist. matrix same as graph

```
for (k=0 to V)
   for (i=0 to V)
      for (j=0 to V)
         if (dist[i][k] + dist[k][j] < dist[i][j])
            dist[i][j] = dist[i][k] + dist[k][j];
```

Time - $O(V^3)$

Ex:



D_0 =

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞ (5)	-5	0	∞ (2)
5	∞	∞	∞	6	0

2,3

2,1 = ∞

1,3 = 8

2,1 + 1,3 < 2,3

∞ 8 ∞

D_1 =

	1	2	3	4	5
1	0	3	8	∞ (4)	-4
2	∞	0	∞	1	7
3	∞	4	0	∞ (5)	∞ (11)
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

$D_2 =$

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

$D_4 =$

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

$D_3 =$

	1	2	3	4	5
1	0	3	-1	4	-4
2	∞	0	-4	1	-1
3	∞	4	0	5	3
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

$D_5 =$

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

Unit-4

Greedy Algorithms & Dynamic Programming

Greedy Algorithms:- Always makes the choice that seems best at the moment.

It means that, greedy algo makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

* It never goes back & reverses the decision.
* On the other hand DP can go back & changes the decisions, if needed.

* Greedy algo Adv. & Disad.

(1) Quite easy to come up with a greedy algo (or multiple greedy algo) for a problem.

(2) Analyzing run time is easier for greedy algo.

(3) With greedy algo it's harder to understand correctness of the algo. (Proving that a correct greedy algo is correct is difficult)

Ex: Greedy Algo:

1. Dijkstra's algo
2. Kruskal
3. Prim's algo
4. Activity Selection
5. Job scheduling
6. fractional knapsack
7. Huffman Code

Where to use Greedy Algo (GA):

A problem must follow these 2 properties —

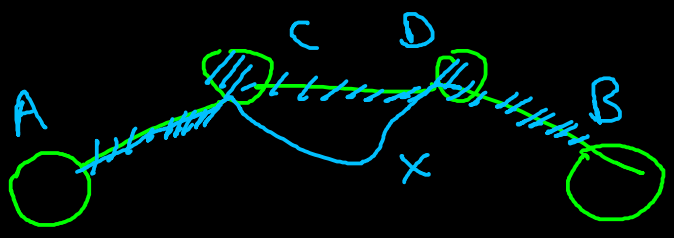
① Optimal Substructure Property —

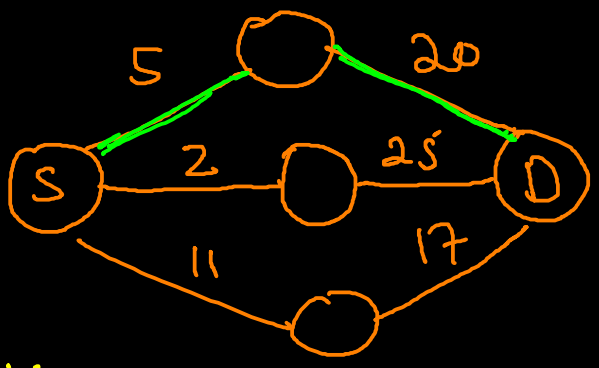
A problem is said to have OS if an optimal solution can be constructed from optimal solution of its subproblems.

* This property is used both in Greedy problems & DP

* There is another property that is used by DP — Overlapping Subproblems

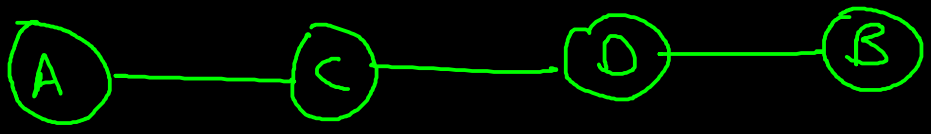
Ex: * will finding shortest path from city A to city B exhibit optimal substructure property? Yes





If there is shortest path between city A & B that passes through city C & D (first C then D), then shortest path from C to B must pass through D.

Ex:



Suppose there is min fare of flight from city A to B that passes through C & then D. Does it means fare of flight from C to B will be min through D only ?

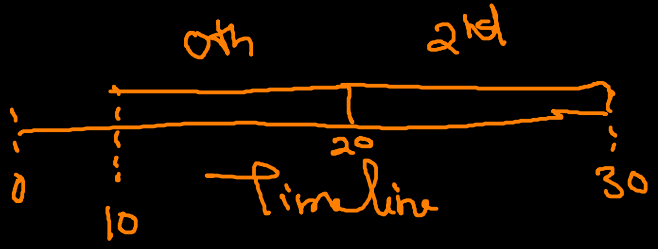
- It might not follow optimal sub. property.

Activity Selection Problem:- you are given n activities with their start & finish times. Select the max no. of activities that can be scheduled / performed by a person, assuming that a person can only work on a single

activity at a time.

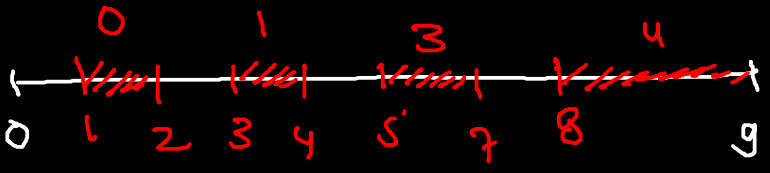
Ex:

	0	1	2
start	10	12	20
finish	20	25	30



Ex:

	0	1	2	3	4	5
s	1	3	0	5	8	5
f	2	4	6	7	9	9
	✓	✓	X			



Algo:

① Sort the activities according to their finish time.

② Select the first activity from the sorted list of activities & print it.

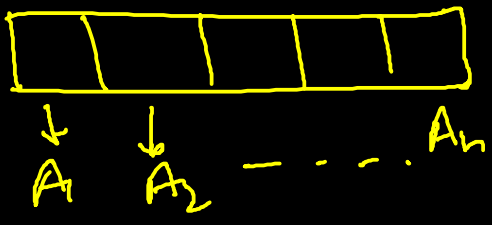
③ Do the following for the remaining activities —

③.1 If the start time of this activity is ≥ finish time of the previous activity, then select this & print it.

Need to create

* Array of structures.

```
struct Activity
{ int start, finish;
};
```




```
bool ComparisonFunction (Activity A1, Activity A2)
{
    return A1.finish < A2.finish;
}
```

```
ActivitySelection ( Activity arr[], int n)
    sort ( arr, arr+n, Comparison function);
    int i=0;
    cout << arr[i].start << arr[i].finish;
    for (int k=1; k<n ; k++)
    { if (arr[k].start >= arr[i].finish)
      { cout << arr[k].start << arr[k].finish;
        i=k;
      }
    }
```

Time - $O(n \log n)$

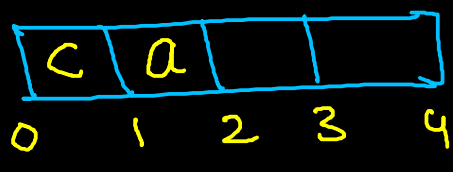
Job Sequencing Problem:- Given an array of jobs where every job has a deadline & associated profit if the job is finished before the deadline.

Every job takes a single unit of time.

How to maximize the profit if only one job can be scheduled at a time.

Ex:

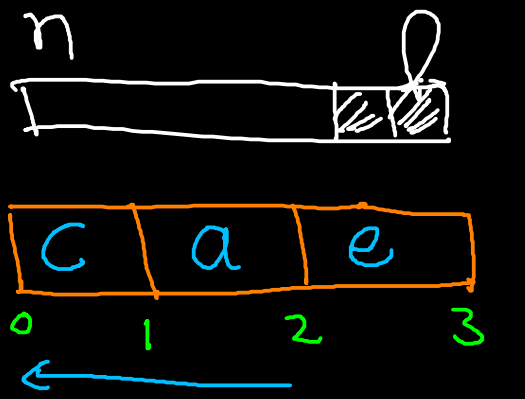
id	deadline	profit
a	4	20
b	1	10
c	1	40
d	1	30



40+20 = 60

Ex:

	id	deadline	profit	
✓	a	2	100	①
X	b	1	19	④
✓	c	2	27	②
X	d	1	25	③
✓	e	3	15	⑤



100 + 27 + 15 = 142

* Generate all subsets & keep track of max profit among all feasible subsets.

Exponential

Algo

① Sort all jobs in decreasing order of profit.

② iterate jobs in " " " "

(2.1) find a time slot i, such that slot is empty & i < deadline and i is the greatest. Put the job in this slot & mark this slot filled.

(2.2) If no such i exists, ignore the job.

* struct Job
{ char id;
int deadline, profit;
};

```
bool Compare (Job j1, Job j2)
{
return j1.profit >
        j2.profit;
}
```

```
JobScheduling ( job arr[], int n)
    Sort ( arr, arr+n, Comparator);
    int result [n];
    bool slot [n]; // make all these false
                   // in the beginning

    for (int i=0; i<n ; i++)
    {  for(int j=min(n,arr[i].deadline; j>=0 ; j--)
       {  if (slot [i] == false)
          {  result [i]= i;
             slot[i] = true;
             break;
          }
       }
    }
```

Time - O(n2)

3 3 3

* Print the result whose slots are empty.
* Try Combining slot & result arrays into one.
* you can optimize the solution further by using PriorityQueue (max Heap)

That will reduce the time complexity to — O(n log n) Space - O(n)

Fractional Knapsack - Given the weights & values of n items, we need to put these items in a knapsack of capacity W to get the max value in the knapsack.

* There is another variant of the problem called 0/1 knapsack, where we are not allowed to break the items in fraction.

Ex:

value	60	100	120
weight	10	20	30
value/weight	6	5	4

W = 50

$10*6 + 20*5 + 20*4$

$= 240$

* Brute force — inefficient

* Calculate the ratio of value/weight & sort in decreasing order ← $O(n \log n)$
* Keep on adding items till knapsack is full or until we can't add the next item as a whole — $O(n)$
* At the end add the next item as much as possible.

← Time - $O(n \log n)$

```
* struct Item
{ int value, weight;

  Item (int value, int weight)
  { this->value = value;
    this->weight = weight;
  }
};
```

// you can also use
// pair of STL to
// represent an item
// & then create a
// vector of pairs

* you can also create
a class that represents
an items.

```
bool Compare (struct Item a, struct Item b)
  { double ratio1 = (double) a.value / (double) a.weight;
    double ratio2 = (double) b.value / (double) b.weight;
    return ratio1 > ratio2;
  }

FractionalKnapsack (int W, struct Item arr[], int n)
{ Sort (arr, arr+n, Compare);
  // create a loop to print new order of items

  double result = 0.0;
  for (int i=0; i<n; i++)
    { if (arr[i].weight <= W)
        { W = W - arr[i].weight;
          result += arr[i].value;
        }
```

```
        else
            result += (double)arr[i].value / (double)arr[i].weight * W;
            break;
        }
    }
    return result;
}
```

Time - O(nlogn)

Huffman Coding :- Huffman Coding is a lossless data Compression algorithm.

Ex: you are given a text file of 1000 characters

The only chars in file are - a, b, c, d, e, f

we need 3 bits to represent these chars.

eg. a = 000, b = 001, c = 010 d = 011, e = 100, f = 101

— f(a) = 450, b = 130, c = 120, d = 160, e = 90, f = 50

using 1st method we will be using 3000 bits.

Now using Variable length codes —

a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

How many bits it will take? 2240

* The idea is to assign variable length codes to input chars.

* Length of the assigned codes are based on the frequencies of corresponding chars.

* The variable length codes assigned to I/P chars are Prefix Codes - The codes are assigned in such a way that the code assigned to one char is not the prefix of code assigned to any other char. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstreams.

Ex: a = 00, b = 01, c = 0, d = 1

Compressed bit stream - 0001

find out the original / decompressed output -

ab, cccd, ccb, acd, cad

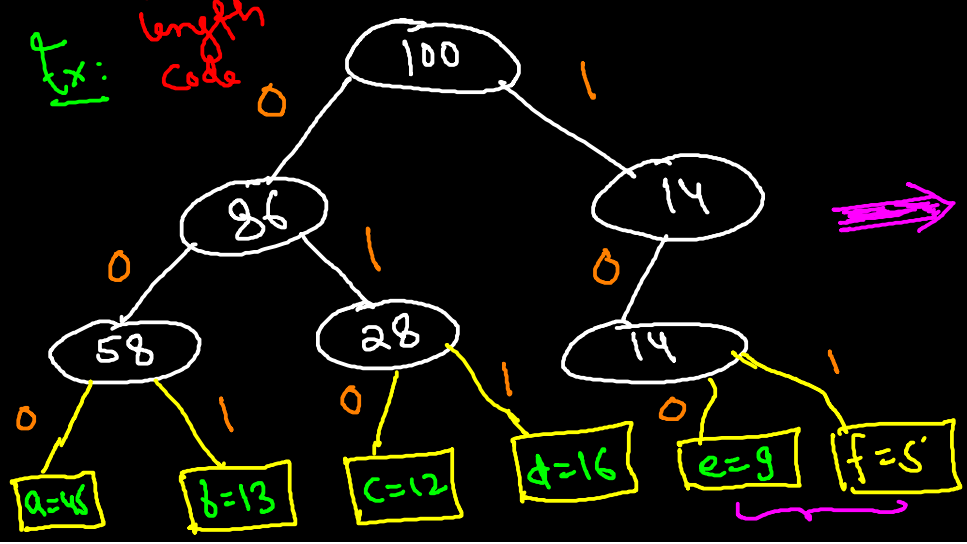
This is ambiguous encoding because code of c is prefix of a & b.

* Ex. for variable length codes given above

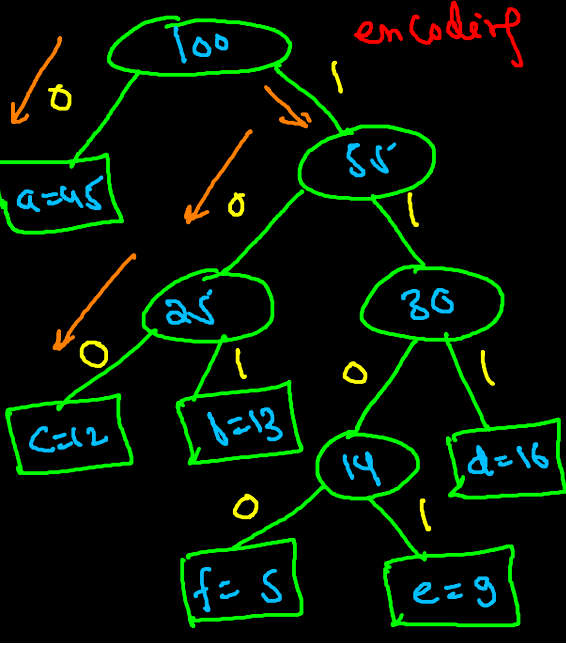
Compressed bit stream - 1000101

output - cab

Ex: fixed length code



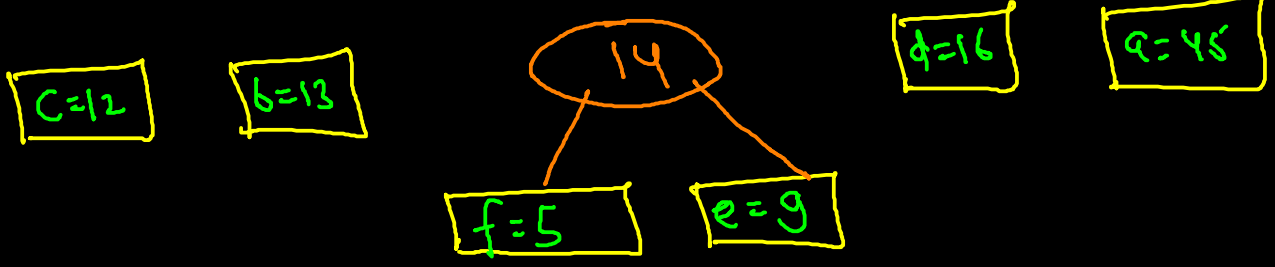
Variable length encoding

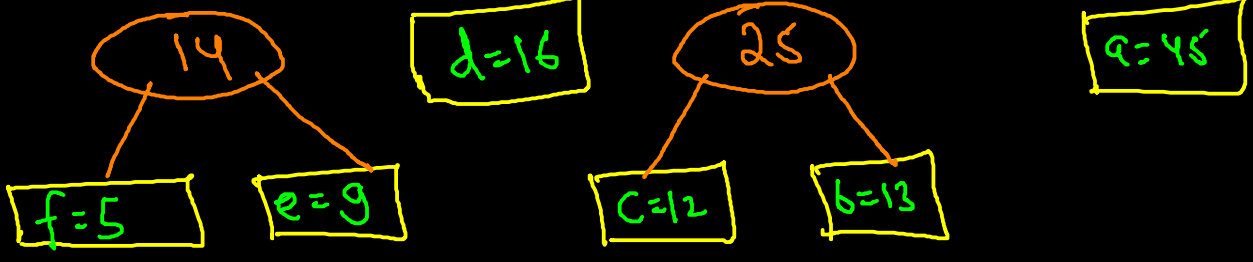


* A code for a file can be represented using a binary tree.
— As many leaves as there are diff. chars in file
- Each internal node will have 2 children and it is labelled with the sum of frequencies of the leaves in its subtree

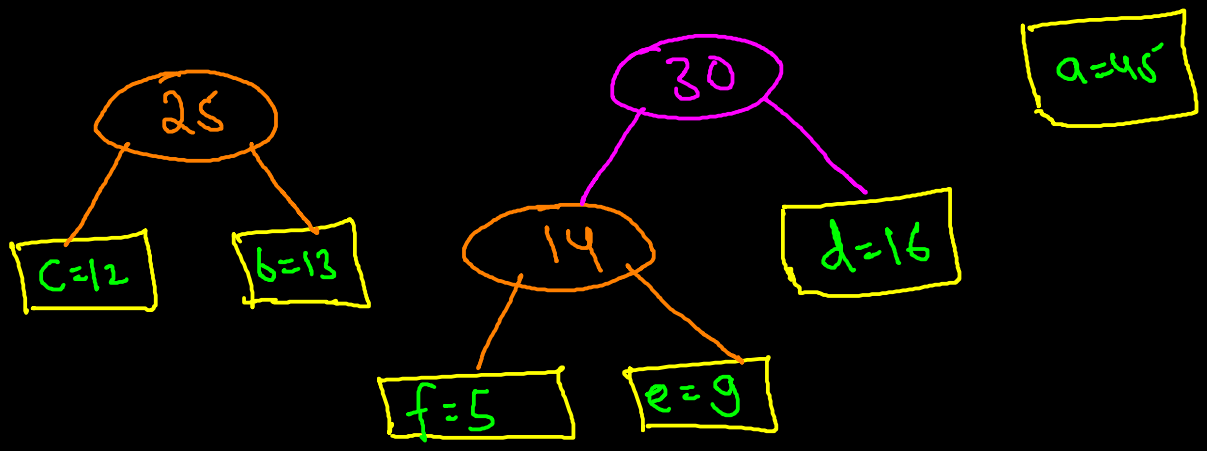
Algo: ① find the freq. for each char
② for each char create a node & store its freq. in the node.
③ Find the two least freq. nodes x & y.
④ create a new node z & make its children x & y. The freq. of z is the sum of the freq. of its children.
⑤ Repeat (x & y will not be considered again) until there is only one node left (Root of tree)

Ex: (1) 

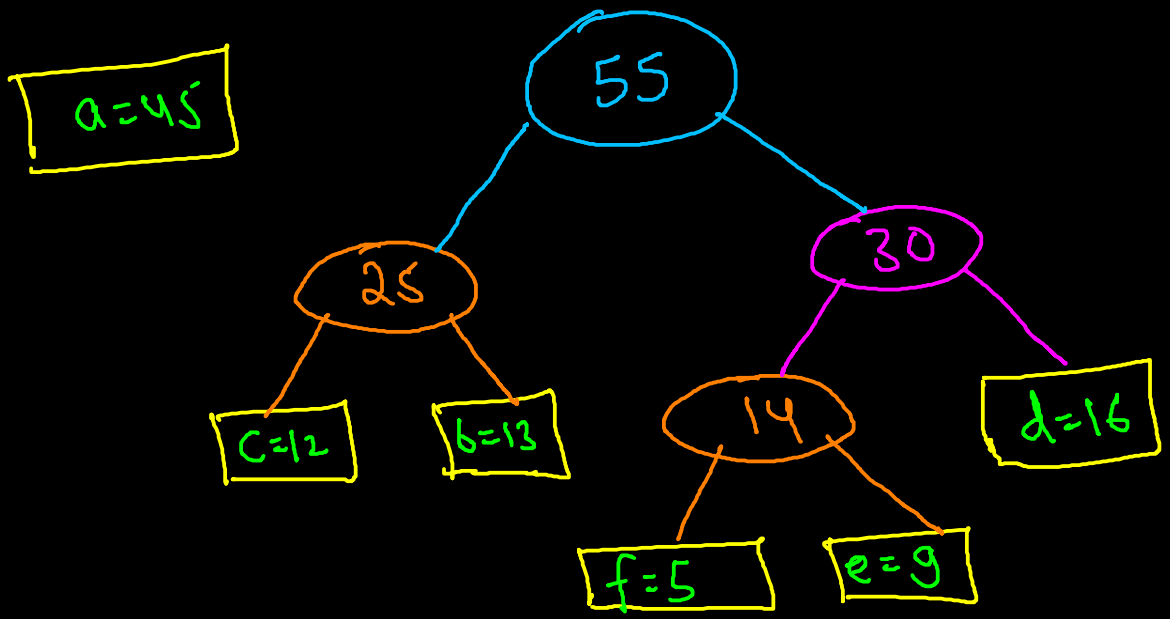
(2) 

(3) 

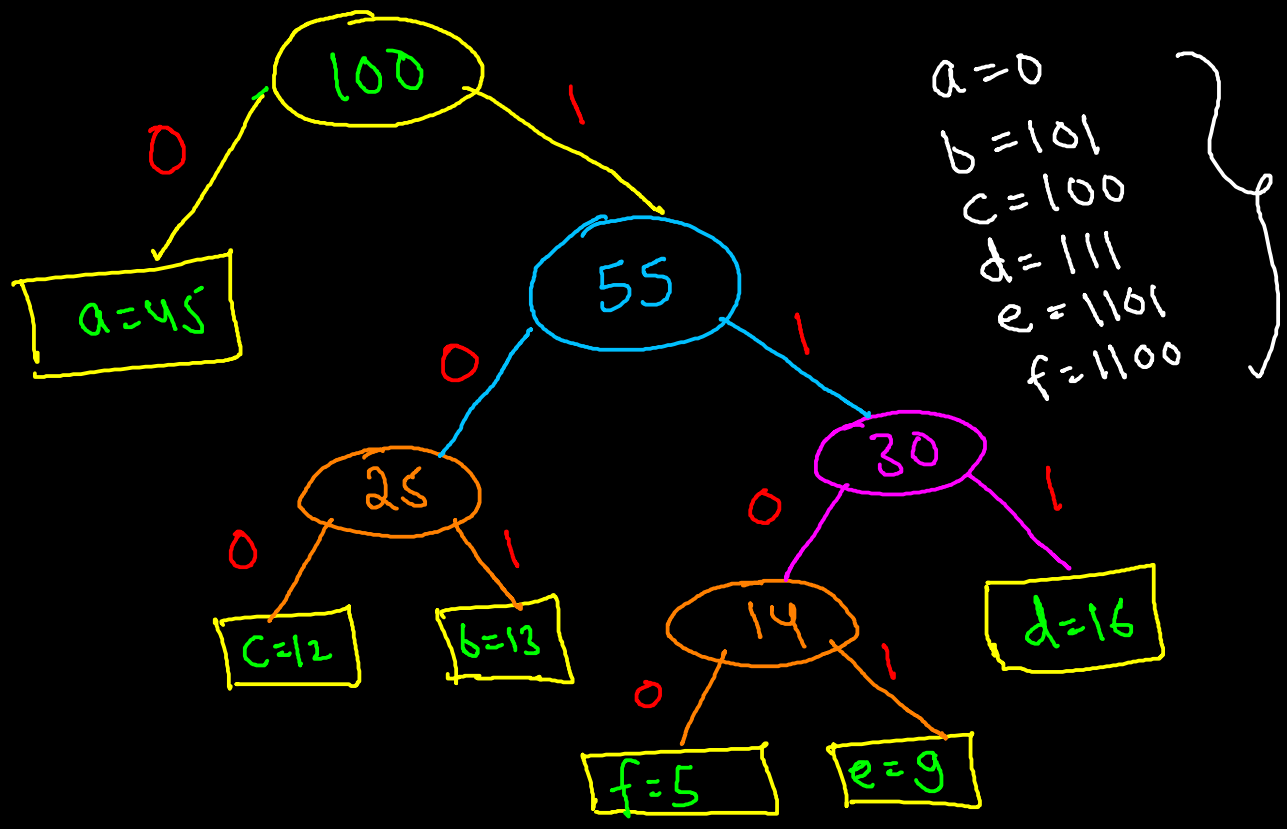
(4)



(5)



(6)



a=0
b=101
c=100
d=111
e=1101
f=1100

Pseudocode:

Input:- a collection C of objects containing a char & its freq.

Output: The root of a Huffman tree

Data structure - Priority Queue (min Heap) Q

Time - $O(n \log n)$

```
n = |C|
Q = C
for i=1 do n-1
    z = new treeNode
    left(z) = Extract min (Q)
    right(z) = Extract min (Q)
    freq(z) = freq(left(z)) + freq(right(z))
    insert (Q, z)
return Extract Min (Q)
```

Ex:

chars	a	b	c	d	e	f	g
freq	10	15	12	3	4	13	1

a - 111 f - 01
b - 10 g - 11000
c - 00
d - 11001
e - 1101

- Create a Binary (Huffman) tree for given freq. of characters.
- Determine code for each char.
- What will be avg. code length ? - 2.52
- length of Huffman encoded message (in bits) 146

① Avg. code length per char $= \frac{\sum (\text{freq}_i * \text{code length}_i)}{\sum \text{freq}_i}$

② Total no. of bits in Huffman encoded message = Total chars in message * Avg. code length

$= \sum (\text{freq}_i * \text{code length}_i)$

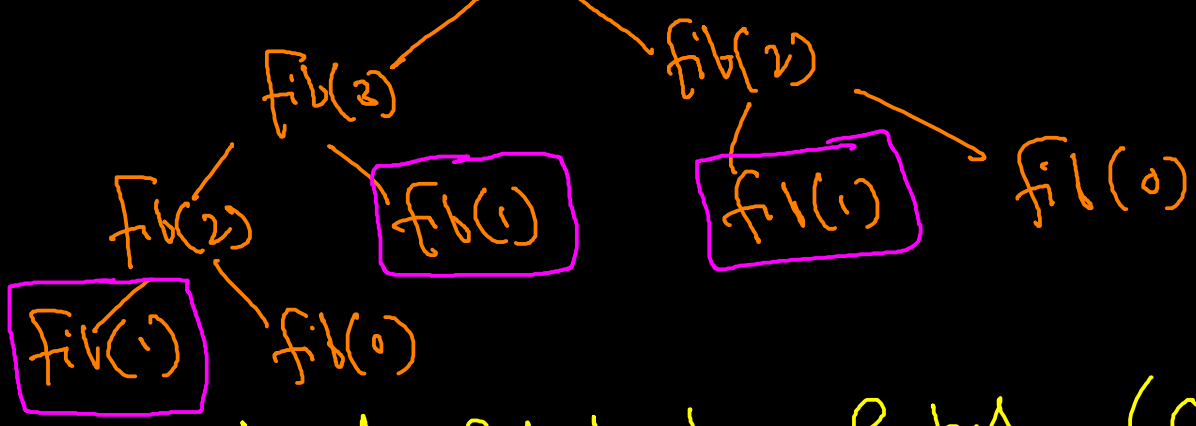
Dynamic Programming:- Breaking down an optimization problems into simpler sub-problems and storing the solution to each sub-problem so that each sub-problem is solved only once.

There are 2 properties that DP must follow -

① **Overlapping Subproblems:-** DP is useful when computed solutions to subproblems are stored in a table so that these don't have to be recomputed.

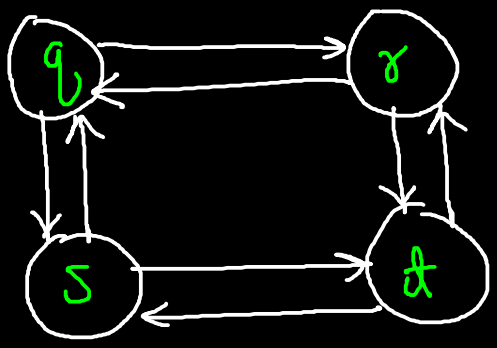
* DP is not useful when there are no common (overlapping) subproblems.

Ex. fib(4)



② **Optimal Substructure Property (OSP):-**
- Global solution can be constructed with optimal solutions of subproblems.

Ex: Shortest path problem has OSP
On the other hand longest path problem doesn't exhibit OSP.



longest paths from q → t

① q → r → t

② q → s → t

q → s → t → r → t → r → q → s → t

q → s → t → r → q → s → t

* There are 2 ways to store the values so that the value of a sub-problems can be reused.

① Tabulation : Bottom up

② Memoization : Top Down

Ex: State for DP — dp[x]

Base case — dp[0] → Bottom up, if you start with dp[0]

Transition → Definition — dp[n] (find out this value) → Top down, if you begin here.

① Tabulation :-

* factorial (x)

```
int dp[SIZE];
int dp[0]=1;
for (int i=1; i<=n; i++)
    dp[i] = dp[i-1] * i;
```

fill array with values -1

// int dp[SIZE];

② Memoization :-

*

```
int fibo (int x)
{ if (x==0 || x==1) return x;
  if (dp[x] != -1)
      return dp[x];
```

return (dp[x] = fibo(x-1) + fibo(x-2));

}

* int fact (int x)

{ if (x==0) return 1;

if (dp[x] != -1) return dp[x];

return (dp[x] = x * fact(x-1));

}

Ex: Given 3 numbers (1,3,5), what are total no. of ways we can form a no. N=8 using the sum of the given 3 no.s. (allowing repetitions & diff. arrangements)

- 1+1+1+1+1+1+1+1 5+1+1+1

State (1) 1 ways to form 1

State (2) 11 ways to form 2

:

state (7)

State (8) 11 ways to form 8

- Adding 1 to all possible combinations of State (7)
- Adding 3 to all possible combinations of state (5)
- Adding 5 to all possible combinations of state (3)

$state(8) = state(7) + state(5) + state(3)$

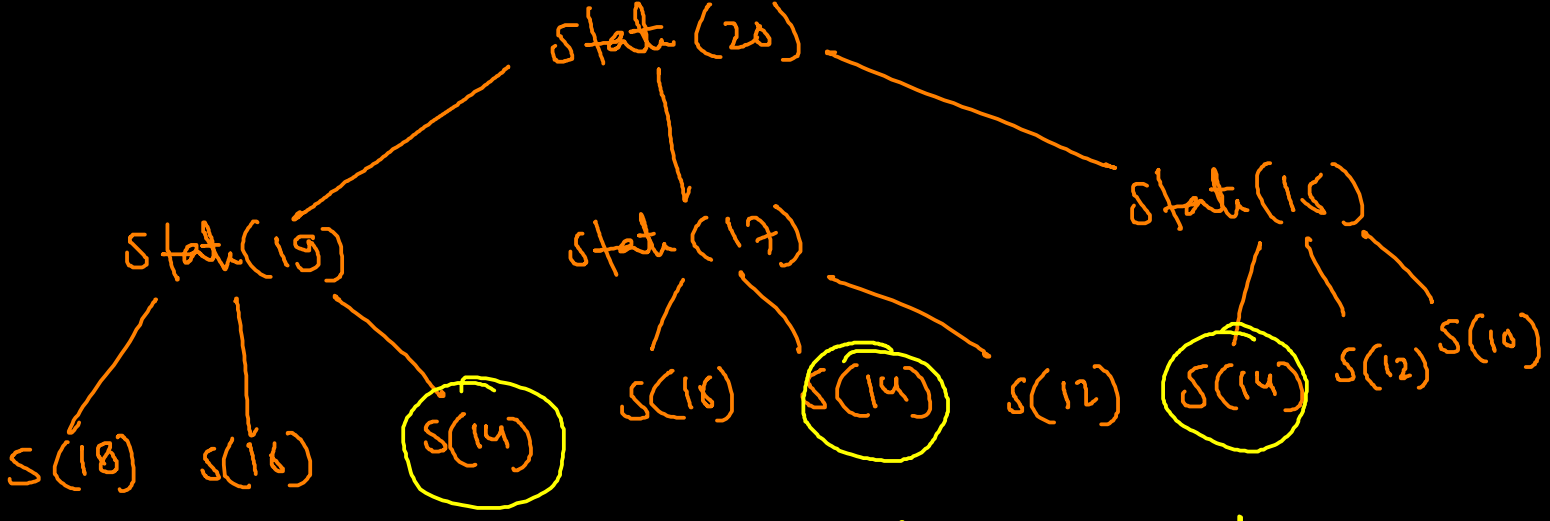
$state(8) = state(8-1) + state(8-3) + state(8-5)$

$state(n) = state(n-1) + state(n-3) + state(n-5)$

* function will be similar to solving fibonacci series.

* Base Case – $n \leq 0$ return 0

$n = 1$ return 1



These are overlapping solutions so its better to add memoization & avoid recomputing the solutions.

```
int dp[SIZE];
int state (int n)
{ if (n<=0) return 0;
  if (n==1) return 1;
  if (dp[n] != -1) return dp[n];
  return dp[n] = state(n-1) + state(n-3) + state(n-5);
}
```

Some standard DP Problems that you need to focus on —

1. fibonacci no.
2. Longest Common subsequence (LCS) (git diff)
3. Longest Increasing subsequence (LIS)
4. Edit Distance
5. Min Cost path
6. Coin change
7. matrix chain multiplication
8. 0/1 knapsack
9. Egg dropping Puzzle
10. Longest palindrome subsequence
11. Cutting a rod
12. Max. Sum increasing subsequence
13. Bellman ford
14. floyd warshall
15. Largest Sum Contiguous subarray
16. Subset Sum Problem
17. Count ways to reach nth stair
18. Permutation / Combination related problems

1. Longest Common Subsequence (LCS):

Application — DNA strands

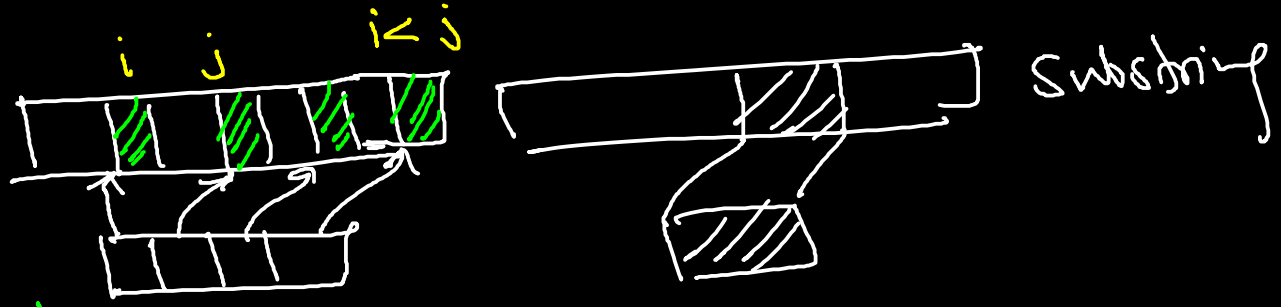
* Exact match rarely occurs because of small changes in DNA replication, so we are interested in finding out / computing

Similarities between strings that don't match exactly.

Two ways to do this — ① Edit Distance ② LCS

* Given 2 sequences $X = \langle x_1, x_2, \ldots x_m \rangle$ and $Z = \langle z_1, z_2, \ldots z_k \rangle$ we say that Z is subsequence of X if there is strictly increasing sequence of k indices $\langle i_1, i_2, \ldots i_k \rangle$ $(1 \le i_1 < i_2 < \ldots i_k \le n)$ such that $Z = (x_{i_1}, x_{i_2}, \ldots x_{i_k})$

Ex: $X = \langle ABRACADABRA \rangle$
$Z = \langle AADAA \rangle$
Here Z is subsequence of X.

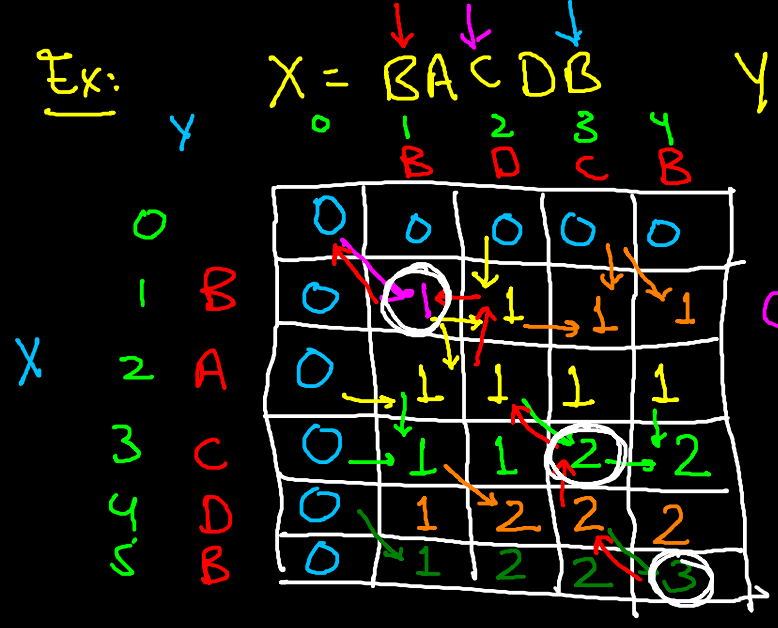


* Given 2 strings X & Y, the LCS of X & Y is a longest sequence Z which is both a subsequence of X & Y.

$X = \langle ABRACADABRA \rangle$
$Y = \langle YABBADABBADOO \rangle$

LCS for X & Y is Z = ABADABA

Ex: X = BACDB Y = BDCB LCS = BCB



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

Ex: X = ABCDGH, Y = AEDFHR } ADH

Ex: X = AGGTAB, Y = GXTXAYAB } GTAB

```
int LCS (char *X, char *Y, int m, int n)
{
  if (m==0 || (n==0)) return 0;
  if (X[m-1] == Y[n-1])
      return 1+ LCS (X, Y, m-1, n-1);
  else
      return max (LCS (X, Y, m, n-1), LCS(X, Y, m-1, n));
}
```

* Now implement above code by using DP.
* we generally make these arrays global. { Initialize with all -1 }
* Create a 2D array / vector dp.

```
int LCS(char *X, char *Y, int m, int n)
{
    if (m==0 || n==0) return 0;
    if(X[m-1]==Y[n-1])
     return dp[m][n] = 1+ LCS(X,Y,m-1,n-1);
    if(dp[m][n]!=-1) return dp[m][n];
    return (dp[m][n] = max(LCS(X,Y,m,n-1),
                           LCS(X,Y,m-1,n));
}
```

Time - $O(mn)$

Space - $O(mn)$

* DP Converts exponential solutions into solutions of polynomial time Complexity.

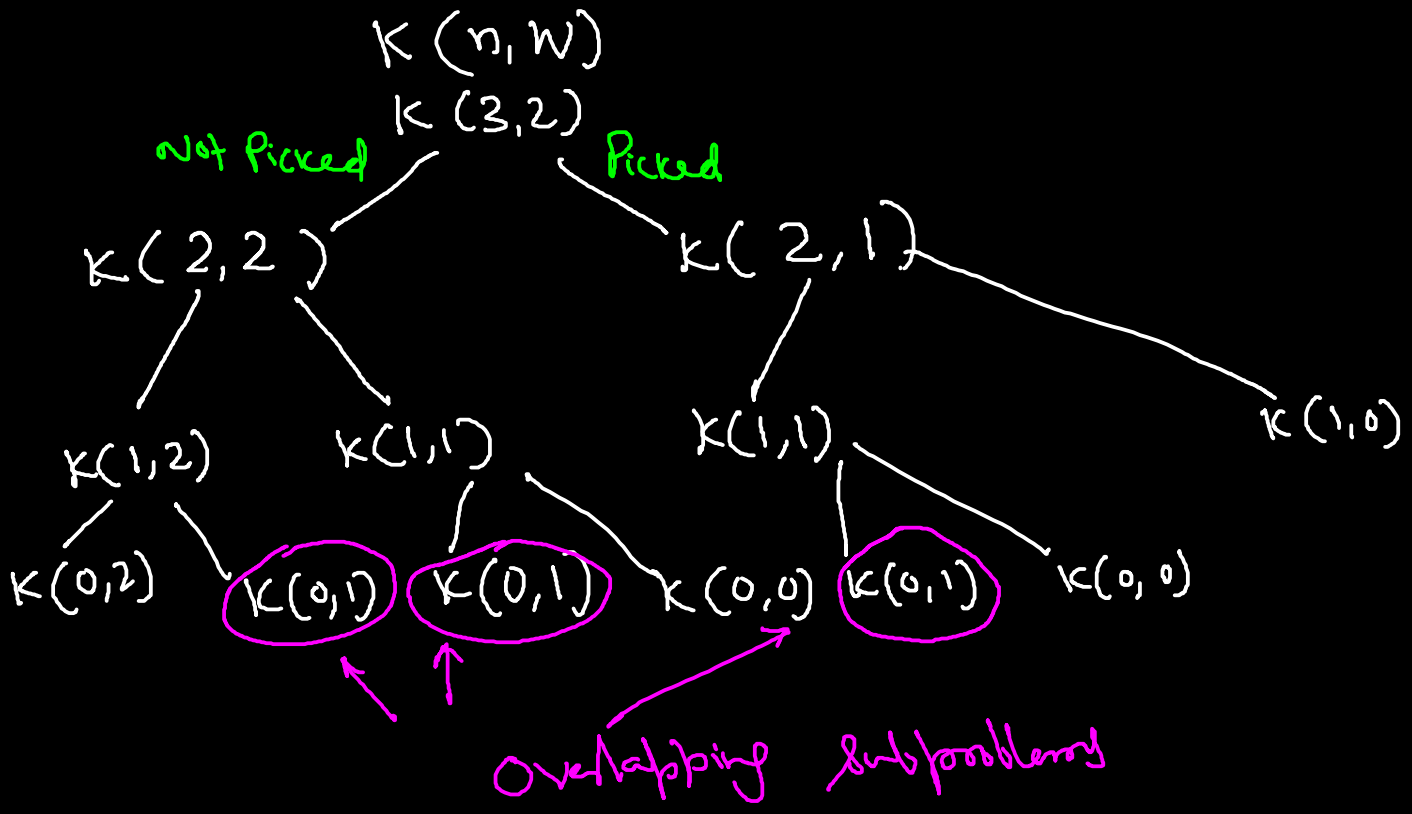
0/1 knapsack Problem:- Given weights & values of n items, put these items in a knapsack of Capacity W to get the max value.

- you Can't break an item, either you need to pick the Complete item or leave it (0/1)

Ex: W = 10, 20, 30
V = 60, 100, 120

W = 50 fractional knapsack - 240
0/1 knapsack - 220

Ex: wt[] = {1,1,1} val[] = {10,20,30} W = 2



* Optimal substructure :-
  1. Include item
  2. Exclude item

  } max

* Value from n items — max { val(n-1) + Include nth item ; val(n-1) + Exclude nth item }

* (1) Max. value obtained by (n-1) items & W weight of knapsack (Exclude)
  (2) value obtained by (n-1) items + including nth item (W - wt of nth item)

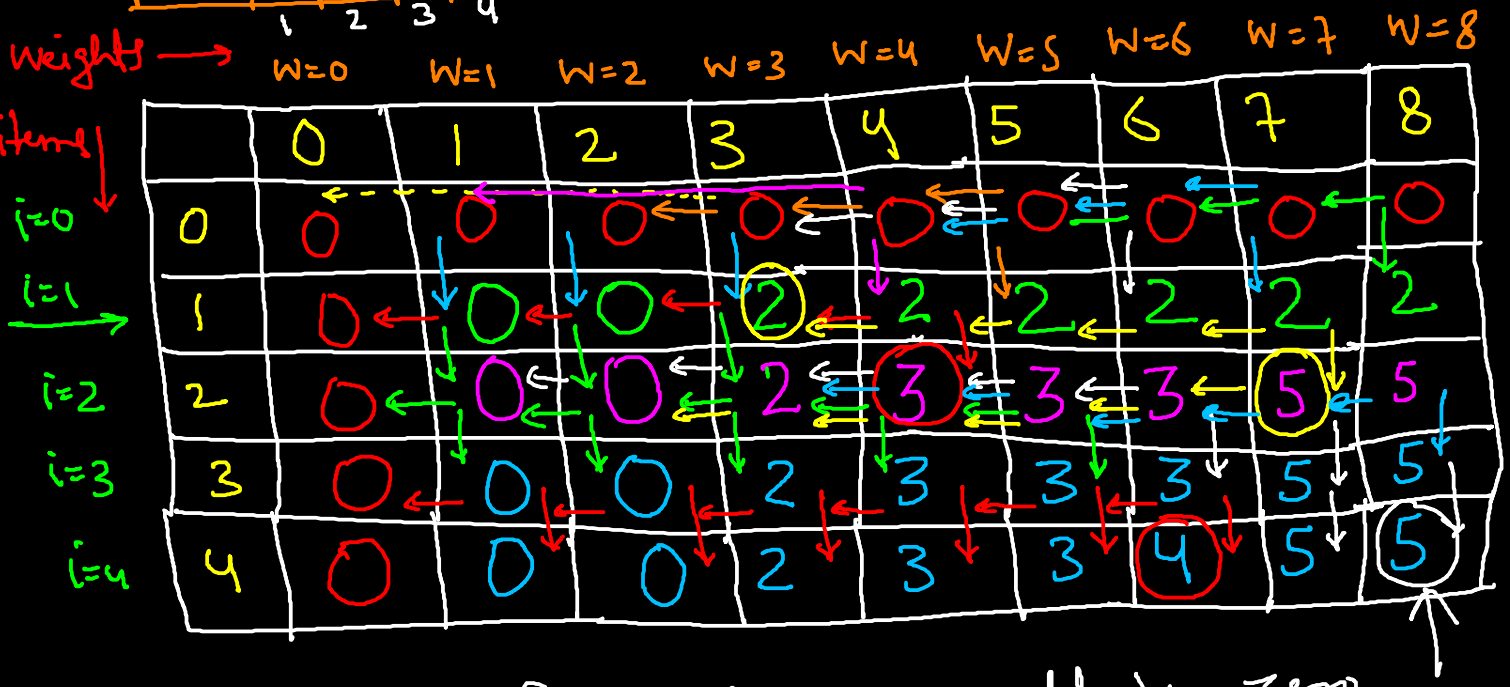
Recursive solution Time — $O(2^n)$

Dynamic Programming Approach:-

Ex:

	0	1	2	3
Weight	3	4	5	6
Value	2	3	1	4
	1	2	3	4

W = 8 Total Profit - 5



* first row & first column would be zero as there is no items for W=0.

Recursive version:-

```
int K ( int W, int wt[], int val[], int n)
{   if (n==0 || W==0)  return 0;
    if ( wt[n-1] > W)
        return K(W, wt, val, n-1);
    else
        return max (K(W, wt, val, n-1),
                    K(W-wt[n-1], wt, val, n-1));
}
```

* As this problem has overlapping subproblems we can use stored solutions of subproblems.
for that we need to create a table.

```
int K(int W, int wt[], int val[], int n)
{
    int i, w;
    int T[n+1][W+1];
    for (i=0 to n)
    {  for (w=0 to W)
       {  if (i==0 || w==0)
              T[i][w] = 0;
          else if (wt[i-1] <= w)
              T[i][w] = max(val[i-1] + T[i-1][w-wt[i-1]],
                            T[i-1][w]);
          else
              T[i][w] = T[i-1][w];
       }
    }
    return T[n][w];
}
```

Time - O(n W)

no of items

Capacity

Include

Exclude

---

i=1

wt[i-1]=3

w = 1

T[1][3] = T[0][3]

2 + T[0][3-3]

2 + T[0][0]

i=2, W=0,1,2, ... 8

wt[1] ≤ 1 ⇒ 4 ≤ 1

T[2][1] = T[1][1]

T[2][2] = T[1][2]

T[2][3] = T[1][3]

wt[1] ≤ 4 when w=4

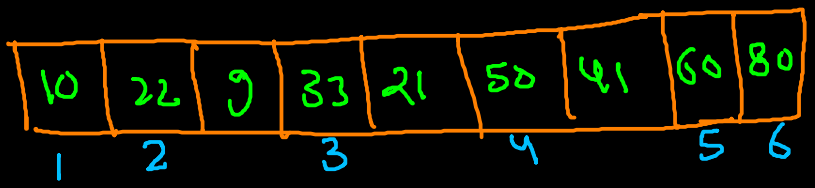
max(3 + T[1][0], T[1][4])

max(3 + 0, 2)

max(3, 2) = 3

Longest Increasing Subsequence (LIS):- LIS is used to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.

Ex:



if $j<i$ and $a[j]<a[i]$ then Consider that (i) element for LIS

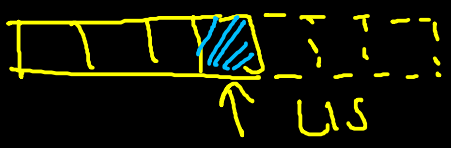
Ex: 5 4 1 2 3 → {1, 2, 3}

Ex: 7 15 → {7}, {5}

First method is to use Recursion-

Optimal Substructure:- Let arr[0..n-1] be the input array and L(i) be the length of LIS ending at index i such that arr[i] is the last element of LIS.

L(i) can be written as follows-



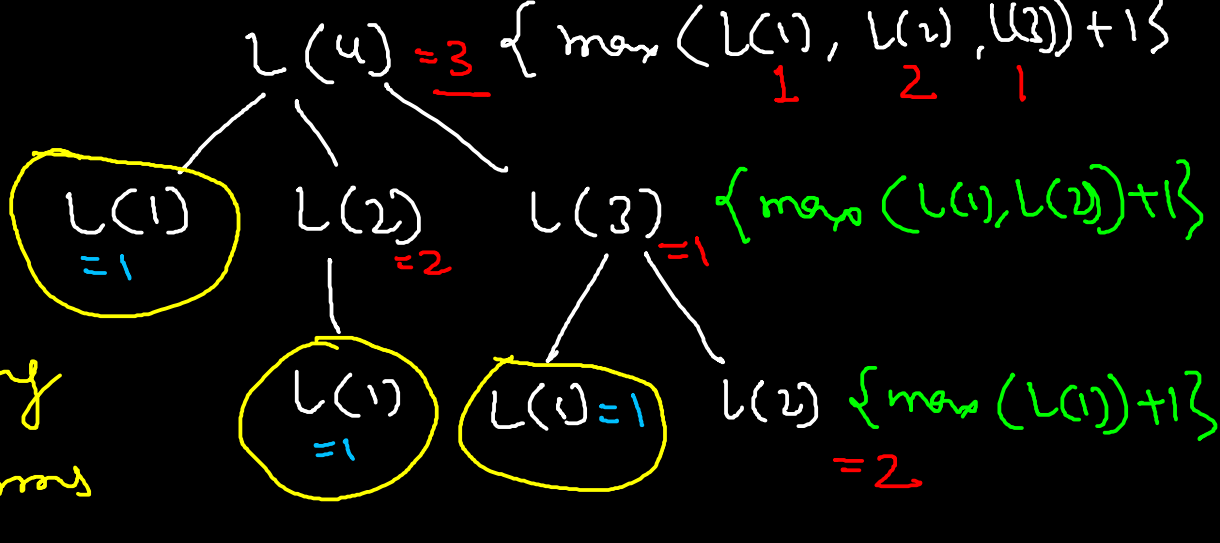
$LIS(i) = 1 + \max(LIS(j))$ where $0<j<i$ and $arr[j] < arr[i]$

$LIS(i) = 1$, if no such j exists

Return $\max(LIS(i))$ where $0<i<n$

* The length of the LIS ending at index i, will be 1 greater than the max of lengths of all LIS ending at indices before i, whose arr[i] < arr[i] (j<i). Thus, LIS satisfies the OSP as large problem can be solved using its subproblems.

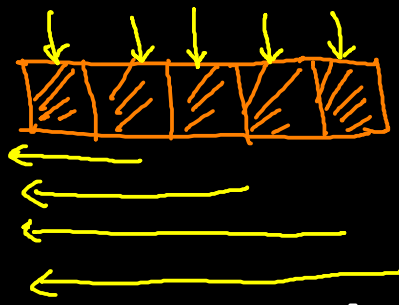
Ex: arr[] = { 3, 10, 2, 11 } (1 2 3 4) LIS = {3, 10, 11}




```
* int LIS ( int arr[], int n)
  {   int result = 1;
      for (i=0 to n-1)
          result = max(result, lis_ending_here (arr, i));

      return result;
  }
```

```
* int lis_ending_here ( int arr[], int curr)
  {  int (curr == 0) return 1;
     int result = 1;
     for (i = curr - 1 to 0)
         if (arr[i] < arr[curr])
             result = max (result, 1 + lis_ending_here (arr, i));
  }  return result;
```



Ex: ① arr[] = { 3, 10, 2, 11 } — 3

Ex: ② arr[] = { 10, 22, 9, 33, 21, 50, 41, 60, 80 } — 6

DP Code:

```
int LIS (int arr[], int n)
{   int L[n];
    L[0]=1;
    for(int i=1; i<n; i++)
    {   L[i]=1;
        for(int j=0; j<i; j++)
        {   if (arr[j]<arr[i]&&L[j]+1>L[i])
                L[i] = L[j] +1;
        }
    }
    return *max_element (L, L+n);
}
```

Time - $O(n^2)$

Space - $O(n)$

* You can further optimize the solution by using Greedy approach & Binary search.

Time - $O(n \log n)$

Space - $O(n)$

Matrix chain Multiplication (MCM):- we are given a seq. of matrices to be multiplied

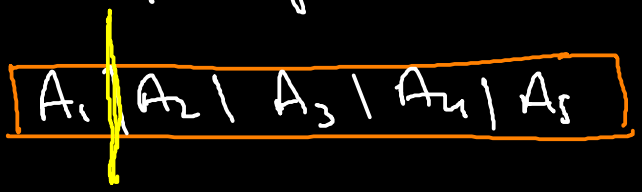
$A_1 \; A_2 \; \cdots \; A_n$

- Matrix multiplication is associative so all the parenthesizations yield the same product.

Ex: $A_1 A_2 A_3 A_4$

- $(A_1 (A_2 (A_3 A_4)))$
- $((A_1 A_2)(A_3 A_4))$
- $(((A_1 A_2) A_3) A_4)$
- $((A_1 (A_2 A_3)) A_4)$

* The way we parenthesize can have a huge impact on the cost of evaluating the product.



Ex: $(A_1 \; A_2 \; A_3)$

A_1: 10×100, A_2: 100×5, A_3: 5×50

$A_1 A_2$

1st way - $((A_1 A_2) A_3)$ — $(10 \times 100 \times 5)$ + $(A_1 A_2) A_3$ $(10 \times 5 \times 50)$ = 7500

$A_2 A_3$

2nd way - $(A_1 (A_2 A_3))$ — $(100 \times 5 \times 50)$ + $A_1 (A_2 A_3)$ $(10 \times 100 \times 50)$ = 75000

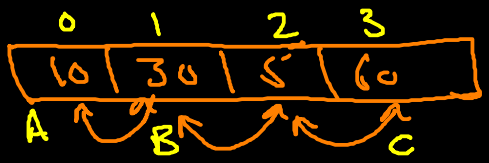
The MCM problem can be stated as follows- Given a chain $(A_1 A_2 \cdots A_n)$ of n matrices where

For $i = 1, 2, \ldots n$ matrix A_i has dimension $P_{i-1} \times P_i$, fully parenthesize the product $A_1 A_2 \ldots A_n$ in a way that minimizes the no. of scalar multiplications.

Ex: A - 10×30, B - 30×5, C - 5×60

1st way - (AB)C — 4500
2nd way - A(BC) — 27000

} Clearly 1st way is better



Recursive Implementation:-

* Matrix A_i - dimension $P[i-1] \times P[i]$

```
int matrixchain ( int P[] int i, int j)
{   if (i == j) return 0;
    int k, count, min = INT_MAX;
    for (k=i; k<j; k++)
    { count = matrixchain (P, i, k) +
              matrixchain (P, k+1, j) +
              P[i-1] * P[k] * P[j];
      if ( count < min)
          min = count;
    }
    return min;
}
```

i & j are first & last indices of array P

Time - Exponential

① Optimal substructure:- In a chain of n matrices we can place the first set of parenthesis in (n-1) ways.

Ex: ABCD

① (A)(BCD) ② (AB)(CD)

③ (ABC)(D)

So, by doing this we have divided the problem into smaller subproblems.

Therefore, this problem has OSP.

* min no. of multiplication needed = min of all (n-1) placements

* DP Code

```
int dp[100][100]; // Initialize with -1
int matrixChainDP ( int *P, int i, int j)
{   if( i==j )   return 0;
    if( dp[i][j] != -1) return dp[i][j];
    dp[i][j] = INT_MAX;
    for (int k=i ; k<j ; k++)
    { dp[i][j] = min (dp[i][j],
        matrixChainDP (P, i,k) + mcd(P, k+1, j)
            + P[i-1] * P[k] * P[j] ;
    }
```

return dp[i][j];

}

Ex:① arr[] = {1, 2, 3, 4}

min multiplications = 18

Ex:② arr[] = {1, 2, 3, 4, 3}

min multiplications = 30

Ex:③ arr[] = {4, 10, 3, 12, 20, 7}

P_0 P_1 P_2 P_3 P_4 P_5

M_1 M_2 M_3 M_4 M_5

for M_i –
P(i-1) as row
P[i] as Column

We need to Compute m[i][j], $0 \le i, j \le 5$

m[i][i] = 0 for all i

	1	2	3	4	5
1	0	120			
2		0	360		
3			0	720	
4				0	1680
5					0

* Calculation of product of 2 matrices

① $m(1,2) = m_1 \times m_2$

$4 \times 10 \times 3 = 120$

② $m(2,3) = m_2 \times m_3$

$10 \times 3 \times 12 = 360$

③ $m(3,4) = m_3 \times m_4$

$3 \times 12 \times 20 = 720$

④ $m(4,5) = m_4 \times m_5$

$12 \times 20 \times 7 = 1680$

* Calculation of 3 matrices

1	2	3	4	5
0	120	264		
	0	360	1320	
		0	720	1140
			0	1680
				0

① $m[1,3] = m_1 m_2 m_3$

– $\underline{(m_1 m_2) m_3}$, $m_1 (m_2 m_3)$

$$m[1,3] = \min \begin{cases} m[1,2] + m[3,3] + p_0 p_2 p_3 = 120 + 0 + 4 \cdot 3 \cdot 12 = 264 \\ m[1,1] + m[2,3] + p_0 p_1 p_3 = 0 + 360 + 4 \cdot 10 \cdot 12 = 840 \end{cases}$$

② $m[2][4] = m_2 m_3 m_4$

$(m_2 m_3) m_4$, $m_2 (m_3 m_4)$

p_1 p_2 p_2 p_3 p_3 p_4

$$m[2,4] = \min \begin{cases} m[2,3] + m[4,4] + p_1 p_3 p_4 = 360 + 0 + 10 \cdot 12 \cdot 20 = 2760 \\ m[2,2] + m[3,4] + p_1 p_2 p_4 = 0 + 720 + 10 \cdot 3 \cdot 20 = 1320 \end{cases}$$

③ $m[3,5] = m_3 m_4 m_5$

$\underline{(m_3 m_4) m_5}$, $m_3 (m_4 m_5)$

$$m[3,5] = \min \begin{cases} m[3][4] + m[5,5] + p_2 p_4 p_5 = 720 + 0 + 3 \cdot 20 \cdot 7 = 1140 \\ m[3,3] + m[4,5] + p_2 p_3 p_5 = 0 + 1680 + 3 \cdot 12 \cdot 7 = 1932 \end{cases}$$

* multiplying 4 matrices

$m[1,4] = m_1 m_2 m_3 m_4$

$(m_1 m_2 m_3) m_4$

$m_1 (m_2 m_3 m_4)$

$(m_1 m_2)(m_3 m_4)$

	1	2	3	4	5
1	0	120	264	1080	
2		0	360	1320	1350
3			0	720	1140
4				0	1680
5					0

$$m[1,4] = \min \begin{cases} 264 + 0 + P_0 P_3 P_4 = 264 + 4 \cdot 12 \cdot 20 = 1224 \\ 0 + 1320 + P_0 P_1 P_4 = 1320 + 4 \cdot 10 \cdot 20 = 2120 \\ 120 + 720 + P_0 P_2 P_4 = 840 + 4 \cdot 3 \cdot 20 = 1080 \end{cases}$$

$m[2,5] = m_2 m_3 m_4 m_5$

$(m_2 m_3 m_4) m_5$

$m_2 (m_3 m_4 m_5)$

$(m_2 m_3)(m_4 m_5)$

$$m[2,5] = \min \begin{cases} 1320 + 0 + P_1 P_4 P_5 = 1320 + 10 \cdot 20 \cdot 7 = 2720 \\ 0 + 1140 + P_1 P_2 P_5 = 1140 + 10 \cdot 3 \cdot 7 = 1350 \\ 360 + 1680 + P_1 P_3 P_5 = 2040 + 10 \cdot 12 \cdot 7 = 2880 \end{cases}$$

* on multiplying 5 matrices

$m[1,5] = m_1 m_2 m_3 m_4 m_5$

$(m_1 m_2 m_3 m_4) m_5$

$m_1 (m_2 m_3 m_4 m_5)$

$(m_1 m_2 m_3)(m_4 m_5)$

$(m_1 m_2)(m_3 m_4 m_5)$

1	2	3	4	5
0	120	264	1080	1344
	0	360	1320	1350
		0	720	1140
			0	1680
				0

$$m[1,5] = \min \begin{cases} 1080 + 0 + p_0 p_4 p_5 = 1080 + 4 \cdot 20 \cdot 7 = 1544 \\ 0 + 1350 + p_0 p_2 p_5 = 1350 + 4 \cdot 10 \cdot 7 = 1630 \\ 264 + 1680 + p_0 p_3 p_5 = 1944 + 4 \cdot 12 \cdot 7 = 2016 \\ 120 + 1140 + p_0 p_2 p_5 = 1260 + 4 \cdot 3 \cdot 7 = 1344 \end{cases}$$

* $(m_1 m_2)((m_3 m_4) m_5)$

Unit-5

Hashing :- Hashing is the process of converting a given key into another value for O(1) retrieval time.

* Hashing is used when keys can have large range of values and the total no. of keys to be stored in the database may be less.

Ex: ① { (1,7), (11,8), (101,20) }

(1 = Key, 7 = Value)

Ex: ② All students have unique roll nos inside University

Ex: ③ In libraries, each book is assigned a unique no., which can be used to determine the details about book.

— In both the examples above, Roll no. & Book no. is hashed to a unique no.

— In Hashing large keys are converted into small keys by Hash functions.

— The values are then stored in a data structure called Hash Table.

Hashing is implemented in Two steps

① An element is converted into an integer by using a hash function.

② The element is stored in the hash table for quick retrieval.

hash = hash fun (key)

index = hash % arraySize

Requirements of a good hash function :-

① Easy to Compute

② Uniform distribution — should not result in clustering

③ less Collisions — should be avoided

Note:- Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a HT it is important to manage Collisions through various Collision resolution techniques.

Ex: Symbol Table in Compiler — keys of ST are some strings that correspond to the identifiers in the language.

Ex: md5 hashing algo / SHA256

Hashing is one way function.

How password matching happens during login? Match Hash

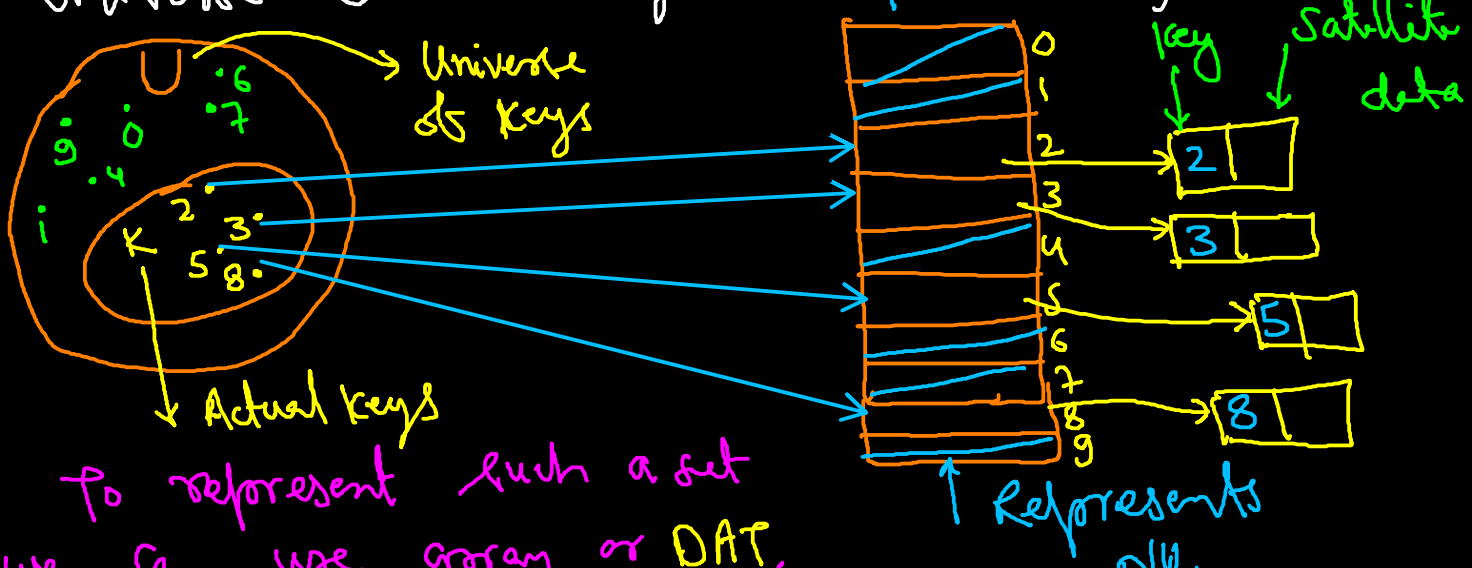
* Is there any chance that 2 passwords generate same Hash?

← There is a rare chance, but it's possible.

* Searching for an element in a hash can take — O(n) in worst case
Under reasonable assumptions, the expected time to search is — O(1)

(DAT)

Direct Address Tables:- Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.



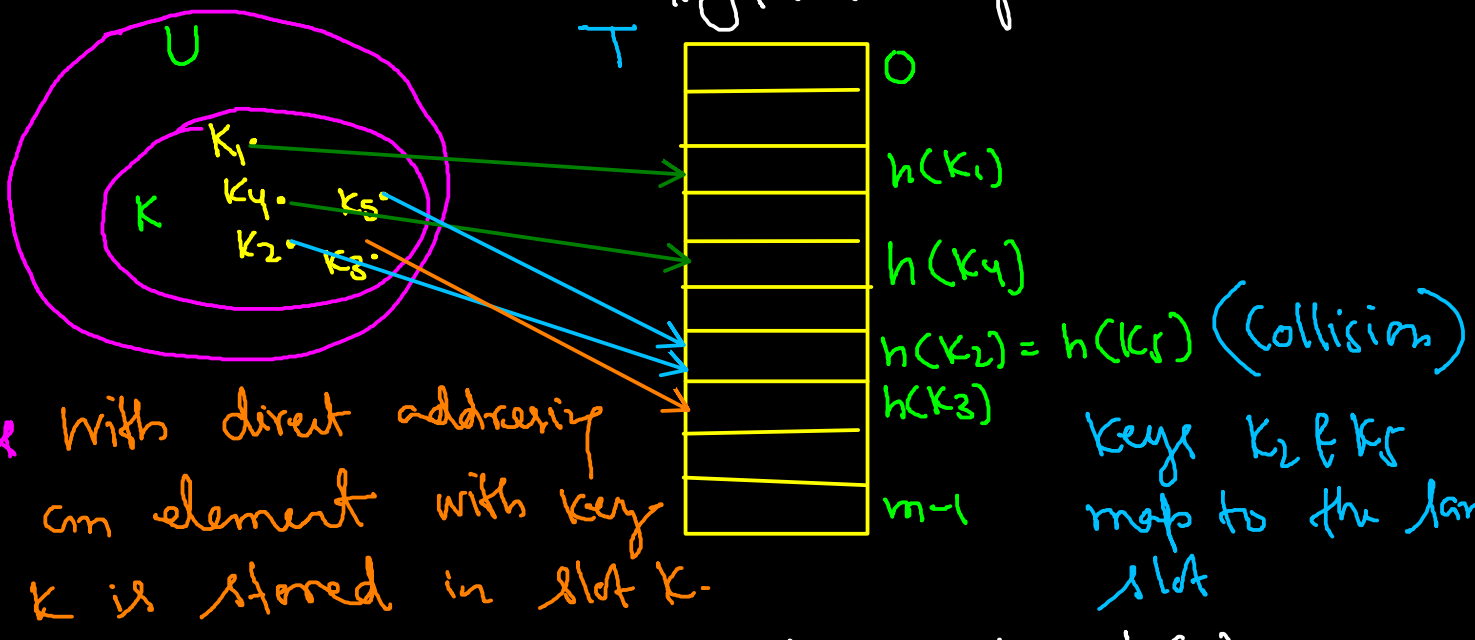
* To represent such a set we can use array or DAT, T[0.....m-1], in which each position or slot corresponds to a key in the universe U.

– Search, Insert, Delete — O(1)

Disadvantage – ① not suitable for large range of values, so we can use HT for such range of values.

② Sometimes keys may be so small relative to U that most of the space allocated to T would be wasted.

Hash Table (HT):- Data structure to store key/value pairs.



* With direct addressing an element with key k is stored in slot k.

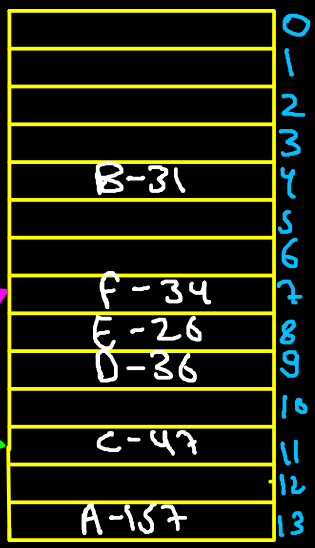
keys k_2 & k_5 map to the same slot

With Hashing it is stored in slot h(k).

So h maps the Universe U of keys into the slots of a hash table $T[0 \ldots m-1]$

Ex: Hash fun. is the sum of digits K

A	B	C	D	E	F
K = 157	31	47	36	26	34
h(K) = 13	4	11	9	8	7



Search for K=47

4+7 = 11

clashing

Search for 1123 — 7

Insert 1123 — 7

clashing:- $h(k_1) = h(k_2)$ where $k_1 \neq k_2$

What to do in case of clashing?

There are 2 ways to handle collision.

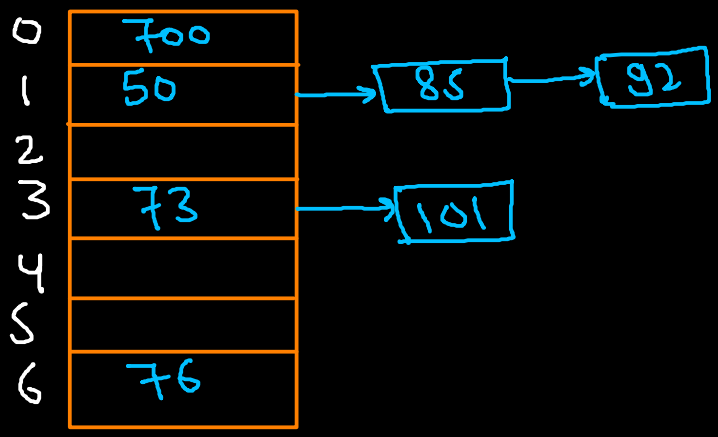
① Separate chaining (Open Hashing)

② Open Addressing (closed Hashing)

① **Separate chaining:-** - most common technique
- Implemented using linked lists (LL)
- Each element of the HT is LL
- If collision between two elements, store both in same LL.

Ex: keys: 50, 700, 76, 85, 92, 73, 101

Hash function - key mod 7



Qu: Demonstrate the insertion of keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a HT with collision resolved by chaining.
HT have 9 slots & Hash fun is $h(k) = k \bmod 9$

Advantages:- ① Simple to implement
② HT never fills up, we can always add more elements
③ Mostly used when it is unknown how many & how frequently keys may be

inserted or deleted.

④ Less sensitive to the hash function or load factor.

Disadvantages:-

① Poor cache performance (Open addressing has better cache ")

② Wastage of space

③ Searching may become O(n) if chain is too long.

④ Use extra space for links.

② Open Addressing:- All the elements are stored in the HT itself.

– Size of HT must be ≥ Total keys

It can be done in following ways

① Linear Probing

② Quadratic "

③ Double Hashing

① Linear Probing:- – Generate $h_i(K) = K \bmod n$

– Store in $h_i(K)$ if $h_i(K)$ is free,
otherwise store in $h_i(K)+1$ if free,
else " " $h_i(K)+2$ if free.

Ex:

	0	1	2	3	4	5	6	7	8	9	10	11	12
Hash Status	E O	E	E O	E	E	E O	E	E	E O	E O	E O	E	E O
Table Value	26		15			18			47	35 (D) 22	9		64

Hash status ⇒ E - Empty, D - Deleted
O - occupied

insert 18, insert 26, insert 35, insert 9, insert 15,
find 48, delete 35, find 9, insert 64,
↓ insert 47, find 35, insert 22
Can't find

Probe sequence :- is a seq. of table entry generated to search / insert / delete a key.

Ex: m = 13
insert keys (18, 41, 22, 44, 59, 32, 31, 73, 19, 20)
Delete - 22
Insert - 34

probe seq. (find 19) — <6, 7, 8, 9, 10, 11, 12>
probe seq (delete 22) — <9>

	0	1	2	3	4	5	6	7	8	9	10	11	12
Hash status	E O	E	E O	E	E	E O	E O	E O	E O	E O D O	E O	E O	E O
Table Value	20		41			18	44	59	32	~~22~~ 34	31	73	19

Disadvantage - Subject to primary clustering

② Quadratic Probing :- - store in h(k) if free
- $(h(k)+1^2)$ mod m, store here if free
- $(h(k)+2^2)$ mod m, store here if free

	0	1	2	3	4	5	6	7	8	9	10	11	12
Hash Status	~~E~~ O	E	E	E	E	~~E~~ O	E	E	E	~~E~~ O	~~E~~ O	E	~~E~~ O
Table Value	22					31				9	10		35

Insert(10), insert(9), insert(22) insert(31)

↓ h(k) = 9

① 22 mod 13 = 9 (not free)

② $(9+1^2) \bmod 13 = 10$ (not free)

③ $(9+2^2) \bmod 13 = 0$

insert(35)

① h(k) = 35 mod 13 = 9 (NF)

② $(9+1^2) \bmod 13 = 10$ (NF)

③ $(9+2^2) \bmod 13 = 0$ (NF)

④ $(9+3^2) \bmod 13 = 5$ (NF)

⑤ $(9+4^2) \bmod 13 = 12$ ✓

* Probe Sequence to insert 35

$< 9, 10, 0, 5, 12 >$

* Quadratic probing suffers from secondary clustering.

Primary Clustering:- Many Consecutive elements form groups and it starts taking time to find a free slot or to search an element.

Secondary clustering:- This is less severe, Two records only have the same collision chain (Probe Sequence) if their initial position is the same.

③ Double Hashing:- The intervals that lie between probes are computed by another hash function.

- Reduces clustering in an optimized way
- The increments for the probing seq. are computed by using another hash function.

Ex: $h(K,i) = (h_1(K) + i \cdot h_2(K)) \bmod m$

$h_1(K) = K \bmod 13$

$h_2(K) = K \bmod 7$

* insert (9), insert (22), insert (10), insert (61), insert (46)

9 mod 13 = 9 - 22 mod 13 (NF)

- $(9 + 1 \cdot 22 \bmod 7) \bmod 13 = 10$

* insert 10 - 10 mod 13 = 10 (NF)

- $(10 + 1 \cdot 10 \bmod 7) \bmod 13 = 0$

* insert 61 – $61 \bmod 13 = 9$ (NF)

$(9 + 1 \cdot 61 \bmod 7) \bmod 13 = 1$

* insert 48 – $48 \bmod 13 = 9$ (NF)

$(9 + 1 \cdot 48 \bmod 7) \bmod 13 = 2$

Note: (1) Linear probing has the best cache performance, but suffers from clustering. It is Easy to Compute.

(2) Quadratic Probing suffers from secondary clustering but is better than primary clustering. It lies between other two in terms of cache performance & clustering.

(3) Poor cache performance but no clustering. It requires more computation time as two hash functions need to be Computed.